Günther Ruhe · Claes Wohlin *Editors*

Software Project Management in a Changing World

Springer

Software Project Management in a Changing World

Günther Ruhe • Claes Wohlin
Editors

Software Project Management in a Changing World

Springer

Editors

Günther Ruhe	Claes Wohlin
University of Calgary	Blekinge Institute of Technology
Calgary, AB	Karlskrona
Canada	Sweden

ISBN 978-3-642-55034-8 ISBN 978-3-642-55035-5 (eBook)
DOI 10.1007/978-3-642-55035-5
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014949137

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Foreword

When the software field was growing up, the software being developed dealt mainly with relatively stable applications. These involved relatively stable business and scientific applications and software involved in controlling relatively stable hardware devices. As experiences in defining requirements for hardware devices found that design solutions would often become requirements and overconstrain the solution space, the software field followed the hardware field in postponing the design until the requirements were completely and consistently defined. This led to the dominance of the sequential, top-down, requirements-first, reductionist waterfall approach used to define, develop, and manage software projects.

One of my jobs at TRW in 1976–1977 was to lead a project to formalize this approach into a set of corporate software development policies and standards. These were inculcated in the company via training materials, courses, and a 40-question equivalent of the California driver's-license test that TRW software developers needed to pass. We also highlighted this material in a public relations campaign to show our mastery of software development and management.

This worked very well for a while, but by the early 1980s the assumption of stable, predetermined requirements began to lose its validity. In particular, graphic-user-interactive (GUI) terminals began to become economically viable. Users much preferred this way of operating, but our requirements engineers found that (1) it was hard to specify graphic layouts in requirement documents and (2) it was hard to get users to define how they wanted to interact. We encountered the IKIWISI syndrome: "I can't tell you how I want it, but I'll know it when I see it."

Our more creative software engineers began to develop rapid-prototyping capabilities that potential customers found very helpful in resolving IKIWISI requirements. However, when we tried to emphasize rapid prototyping in competitive procurements, we found that we had so thoroughly brainwashed many of our senior software engineers that they would pound on the table and say, "You can't do that! It's programming before we've defined the requirements, and it violates our policies!" Further, we found that several government agencies had adopted and adapted our policies and standards as their way of doing business. And if undoing corporate policies was difficult, undoing government policies and standards was virtually impossible.

Since then, further trends have made the sequential, reductionist approach less and less viable. Requirements have become more emergent with system use. With COTS products and cloud services, their capabilities drive the system requirements rather than prespecified requirements. Time-to-market pressures and rapidly evolving products such as cell phones have made sequential definition and development processes uncompetitive in the marketplace, along with increasingly rapid changes in technology, organizations, and user preferences. Yet, many organizations cling to the sequential, reductionist approach as a security blanket. Increasingly, they take several years to deliver a system and then find that its technology is obsolete and that its users' needs have become very different.

Thus, the appearance of this book, *Software Project Management in a Changing World*, is very timely. It focuses on how people and organizations can make their processes more change-adaptive. It is good in emphasizing in its chapters on cost estimation and risk/opportunity management that unpredictable change requires probabilistic approaches, using range vs. point estimates, late binding of product content decisions, and evolutionary development. It has good guidance on agile project management, using principles such as minimum critical specifications, autonomous teams, skills redundancy, and use of feedback and post release reflection.

The book is also strong on quality management and on balancing lightweight agile methods with the use of empirical methods, using Goal-Question-Metric and Experience Factory-type approaches to the management and use of project knowledge. Its chapters on global project management and global team motivation are strong in identifying and employing knowledge on personnel motivation and on the importance of investments in team building and trust, although the chapter on human resource allocation focuses more on algorithmic methods of project staffing.

The strong emphasis on how to make software processes more change-adaptive could have done more on how to make software products more change-adaptive. A good example is the approach in David Parnas' paper on Designing Software for Ease of Extension and Contraction. This involves identifying sources of change and encapsulating them into modules, so that change effects are largely confined to individual modules, rather than rippling through the rest of the product. This also involves identifying evolution requirements as well as current-snapshot requirements for the initial product. Other good product-adaptive approaches include open interface standards, use of design patterns and generics, judicious selection of COTS products that are change-adaptive without destabilizing their users, and emphasizing simplicity via Occam's Razor or Einstein's guidance, "Everything should be as simple as possible, but no simpler."

That said, the book is also strong in identifying sources of change in software technology and their implications for software management. These include big data and search technology that can enhance project knowledge and social media technology that can enable better multidiscipline and distributed-stakeholder collaboration in software requirements negotiation, change handling, and concurrence at decision gates. Also, improved process simulation technology can be used to better understand the likely effects of alternative project decisions and to determine the domains of applicability of various software “laws,” such as Brooks’ Law: Adding people to a late software project will make it later (not always true if foreseen and done early). It is also strong in identifying alternative software development methods and their management differences, such as open source, inner source, distributed and global software development, and agile methods.

Overall, I found the book to be a pleasure to read and a valuable source of guidance on how to cope with the proliferating sources of change we all will face in the future. I hope that you will benefit from it in similar ways.

February 2014 Barry Boehm

Author Biography

Barry Boehm is the TRW Professor in the USC Computer Sciences, Industrial and Systems Engineering, and Astronautics Departments. He is also the Director of Research of the DoD-Stevens-USC Systems Engineering Research Center, and the founding Director of the USC Center for Systems and Software Engineering. He was director of DARPA-ISTO 1989–1992 at TRW 1973–1989, at Rand Corporation 1959–1973, and at General Dynamics 1955–1959. His contributions include the COCOMO family of cost models and the Spiral family of process models. He is a Fellow of the primary professional societies in computing (ACM), aerospace (AIAA), electronics (IEEE), and systems engineering (INCOSE) and a member of the U.S. National Academy of Engineering.

Acknowledgments

Editing a book is a major undertaking; it may sound like it is much less work than authoring your own book. And maybe it is less work, but, foremost, it is different. It requires a lot of coordination, and hence editors become highly dependent on the contributors, reviewers, and others providing support. This book is no different.

We would like to express our gratitude to all the contributors for sharing their expertise and being responsive to our comments, enquiries, and requests. We are grateful to all reviewers helping us further improve the content of the book. Maleknaz Nayebi was of tremendous support in preparing supplementary literature studies. We would also like to express our gratitude to the Springer team for their support and in particular to Ralf Gerstner for his guidance and valuable input on practical matters.

We are grateful to Professor Barry Boehm for agreeing to write the foreword and for sharing his vast experience with us. Barry not only provides his personal perspective on the evolution of software development as a whole but also focuses on software project management.

Last but not least, we would like to express our sincere thanks to Kornelia Streb. She re-created most of the art work and did an excellent job in handling a large variety of tasks in the process of editing the book.

February 2014

Günther Ruhe
Claes Wohlin

Contents

Part II Supporting Areas

Part III New Paradigms

Part IV Emerging Techniques

List of Contributors

Andreas S. Andreou Department of Electrical Engineering, Computer Engineering and Informatics, Cyprus University of Technology, Lemesos, Cyprus

Sarah Beecham Department of Computer Science & Information Systems, Lero—The Irish Software Engineering Research Centre, University of Limerick, Limerick, Ireland

Barry Boehm University of Southern California, Los Angeles, USA

Sjaak Brinkkemper Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands

Darren Dalcher National Centre for Project Management, University of Hertfordshire, Hatfield, UK

Alexander Delater Institute of Computer Science, University of Heidelberg, Heidelberg, Germany

Torgeir Dingsøyr SINTEF, Trondheim, Norway

Ton Dobbe UNIT4, Sliedrecht, The Netherlands

Tore Dybå SINTEF, Trondheim, Norway

Christof Ebert Vector Consulting Services GmbH, Stuttgart, Germany

Filomena Ferrucci DISTRA, University of Salerno, Salerno, Italy

Mark Harman Department of Computer Science, University College London, London, UK

Rachel Harrison Computing and Communication Technologies, Oxford Brookes University, Oxford, UK

Jens Heidrich Fraunhofer IESE, Kaiserslautern, Germany

Tom-Michael Hesse Institute of Computer Science, University of Heidelberg, Heidelberg, Germany

Martin Höst Department of Computer Science, Lund University, Lund, Sweden

Erik Jagroep Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands

Michael Kläs Fraunhofer IESE, Kaiserslautern, Germany

Tim Menzies Lane Department of Computer Science & Electrical Engineering, West Virginia University, Morgantown, USA

Nils Brede Moe SINTEF, Trondheim, Norway

Alma Oručević-Alagić Department of Computer Science, Lund University, Lund, Sweden

Barbara Paech Institute of Computer Science, University of Heidelberg, Heidelberg, Germany

Dietmar Pfahl Institute of Computer Science, University of Tartu, Tartu, Estonia

Dieter Rombach Technische Universität Kaiserslautern, Kaiserslautern, Germany

Fraunhofer Institute for Experimental Software Engineering (IESE), Fraunhofer-Platz 1, Kaiserslautern, Germany

Günther Ruhe Department of Computer Science & Electrical Engineering, University of Calgary, Calgary, Canada

Federica Sarro Department of Computer Science, University College London, London, UK

Martin Shepperd Information Systems and Computing, Brunel University, Uxbridge, UK

Darja Smite Blekinge Institute of Technology, Karlskrona, Sweden

Ioannis Stamelos Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece

Klaas-Jan Stol Lero—The Irish Software Engineering Research Centre, University of Limerick, Limerick, Ireland

Constantinos Stylianou Department of Computer Science, University of Cyprus, Lefkosia, Cyprus

Varsha Veerappa Department of Computing and Communication Technologies, Oxford Brookes University, Oxford, UK

Inge Van de Weerd Department of Information, Logistics and Innovation, VU University Amsterdam, Amsterdam, The Netherlands

Claes Wohlin Blekinge Institute of Technology, Karlskrona, Sweden

Chapter 1
Software Project Management: Setting the Context

Günther Ruhe and Claes Wohlin

Abstract This chapter is designed as the introduction to the book. It provides the motivation for studying software project management as a response to the increasing variety of software development methodologies. The chapter characterizes software projects and presents ten knowledge areas in software project management. This body of knowledge is described in the software edition of the Project Management Body of Knowledge (PMBOK). The chapters of the book are classified in terms of their contribution to these knowledge areas.

The chapter also discusses the multidisciplinary nature of the project management discipline. Based on some predicted trends for the future of software engineering, a prediction on the future of software project management is given. Finally, an overview of the content and structure of the whole book is presented.

1.1 Motivation

The world is continuously changing. Software and software-intensive systems are among the key drivers of this trend. The speed and magnitude of all these changes is breathtaking. What would happen today if any of the existing telecommunication, health-care, financial, or logistic systems were not performing securely, safely, and reliably? The rapid growth in technology in combination with the strong dependence of products and services on software raises the demand on managing the development and evolution of such systems.

G. Ruhe (✉)
University of Calgary, Calgary, Canada
e-mail: ruhe@ucalgary.ca

C. Wohlin
Blekinge Institute of Technology, Karlskrona, Sweden
e-mail: claes.wohlin@bth.se

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_1, © Springer-Verlag Berlin Heidelberg 2014

Project management is one of the youngest, most vibrant, and most dynamic fields among different management disciplines. According to the PMBOK, project management is the "application of knowledge, skills, tools and techniques to project activities to meet the project requirements" (PMI 2013a). Project management is accomplished through the application and integration of 47 logically grouped project management processes divided into five process groups: initiating, planning, executing, monitoring and controlling, and closing.

Software is a direct product of the cognitive processes of individuals engaged in innovative teamwork. Many of the procedures and techniques used in software project management are designed to facilitate communication and coordination among team members engaged in an intellectually intensive work. Software development is often characterized as a learning process in which knowledge is gained and information generated during the project. Dealing with people and conflicts, team building, knowledge sharing, and communication will be the determinants of good project management.

Software project management deals with software projects and the challenges of human-based development (as opposed to the more deterministic processes in traditional projects). The higher flexibility in software development approaches puts new demands on the capabilities of software project management. Weaknesses in planning, organizing, staffing, directing, and controlling are hard to be counterbalanced by more efficiency in technical development work. As Fred Brooks stated in 1987, "... today's major problems with software development are not technical problems, but management problems" (Brooks 1987).

The principal nature of the challenges in software project management has not changed dramatically in the last 25 years. However, software-intensive systems of the twenty-first century increasingly vary in their content, size, complexity, and their degree of interaction with other systems. The technological and communication infrastructure to develop these systems is hard to compare with that available in the past. As a consequence, the concrete content of the project management challenges looks different from that of 25 years ago.

Beginning from the 1970s and 1980s, traditional plan-driven software development has been replaced and complemented by more adaptive and dynamic approaches. Global (or distributed) software development, open source development, and the application of the different variants of adaptive development techniques have proven successful under various circumstances. The Internet has dramatically enhanced the ability of individuals, teams, and organizations to manage projects across continents and cultures in real time (Kwak and Anbari 2008). New paradigms (such as inner-source project management) or emerging techniques (such as social media collaboration) provide new opportunities for conducting software project management more successfully than before.

1.2 Characteristics of Software Projects and Why Software Project Management Is Difficult

Software development is both human-intensive and knowledge-intensive, which makes people the most important asset in any software development endeavor. *Software projects* are different from other projects in a number of ways. Consequently, management of software projects cannot be done in the same way as in traditional project management and needs to be adjusted correspondingly. Following (PMI 2013b), some of the main differentiating factors are as follows:

- Software is an intangible product.
- Software is a cognitive and human-based development process that requires sharing of documents.
- There is a higher degree of uncertainty in the project and product scope.
- Communication and coordination within software teams and with project stakeholders often lacks clarity.
- The intellectual capital of software personel is the primary asset of software projects and organizations.
- There is a degree of change of requirements in the course of the software project.
- The creation of software requires innovative problem solving to create unique solutions.
- Initial planning and estimation of software projects is challenging because these activities depend on requirements that are often imprecise or based on lacking information.
- The development and evolution of software-intensive systems is challenging because of the high complexity of software based on the enormous number of logical paths in program modules and all the combinations of interface details.
- Exhaustive testing of software is impractical because of the time and related complexity constraints.
- Software development often involves interactions of different vendor products and interfaces with other software.
- Software security is a large and growing challenge.
- Objective measurement and quantification of software quality is difficult.
- Learning and knowledge creation in software development is more difficult because processes, methods, and tools are constantly evolving.
- The execution of software is platform-dependent and is often an element of a system consisting of diverse hardware, other software, and manual procedures.

Software project life cycles are models of how software projects pass through the phases of development, from their initiation to their closure. The software extension of the PMBOK describes the continuum of software project life cycles ranging from highly predictive to highly adaptive (PMI 2013b). The variation between them is described by the degree of change in requirements (from being specified during initiation and planning to elaboration at frequent intervals during development), the

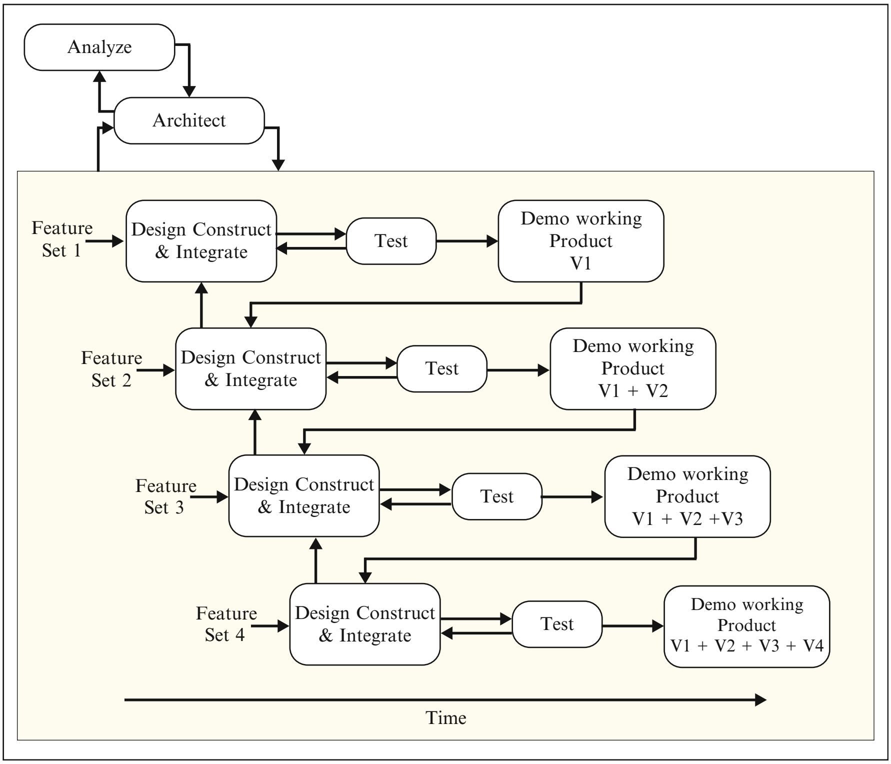

Analyze
Architect
Feature Set 1
Design Construct & Integrate
Test
Demo working Product V1
Feature Set 2
Design Construct & Integrate
Test
Demo working Product V1 + V2
Feature Set 3
Design Construct & Integrate
Test
Demo working Product V1 + V2 +V3
Feature Set 4
Design Construct & Integrate
Test
Demo working Product V1 + V2 + V3 + V4
Time

Fig. 1.1 Incremental software product development (PMI 2013a)

control of cost and risk, and the involvement of key stakeholders (from involvement at scheduled milestones to continuous involvement).

Practically, all modern development approaches are iterative and incremental (Larman and Basili 2003) in their key nature. The notion of iteration refers to a phase within the development process, while the term *increment* describes a certain stage of the product evolution. An example of an incremental software product development process is given in Fig. 1.1.

Kruchten (2011) has proposed a *conceptual model of software development*. The main pillars of that model are four core entities called *intent*, *product, work*, and *people*, which are common entities across all software development projects. In brief, the entities give answers to the following questions:

Intent:	What is the project trying to achieve?
Product:	What is the outcome of the project?
Work:	How to build the targeted product?
People:	Who will be available to perform the work?

Each of these four core entities has three attributes: time, quality, and risk. The *time* attribute refers to a description of the execution of the project over time. *Quality* is applied as well to all the four entities. For example, the quality of the people refers to the competence, diligence, and dedication of the staff assigned to the project. The *risk* attribute describes the inherent uncertainty of all the entities.

In addition, there is a *value* attribute for the intent and product, as well as a *cost* attribute for people and work. The majority of the cost is associated with the cost of the people performing the work. The intended value can be defined in terms of functional and quality requirements. The resulting value can be expressed in financial terms by the net present value of the product.

Besides the common core concepts, there are factors used to differentiate projects. All these factors can be grouped into *organization-level* and *project-level project factors*:

1. Organization-level factors:
   (a) **Business domain**: For what domain is the software (–based) product developed?
   (b) **Number of instances**: How many instances of the software (–based) product will be deployed?
   (c) **Maturity of organization**: How mature are the processes of the software developing organization?
   (d) **Level of innovation**: How innovative is the organization?
   (e) **Culture**: In which culture are the projects developed?
2. Project-level factors:
   (a) **Size**: How big is the system under development?
   (b) **Stable architecture**: Is a stable architecture in place?
   (c) **Business model**: Under which business model is the software (–based) product developed?
   (d) **Team distribution**: How many teams are working in the project and in which configuration?
   (e) **Rate of change**: How stable is your business environment and how many risks and uncertainties are you facing?
   (f) **Age of system**: Greenfield (from scratch) vs. brownfield (evolving) software system development?
   (g) **Criticality**: How many people's safety will be threatened if the system fails?
   (h) **Governance**: Who manages the project managers? How much governance is applied to the project?

This classification can serve as guidance to approach the question: which software project management approach has proven successful and is recommended for which given configuration of project-level and organization-level factors? We do not expect having a single answer in most cases. We also expect having clusters of configurations where certain approaches are recommended. As the pathway to accumulate the knowledge needed to make these decisions, the *empirical paradigm* suggests the replicated application of making the observations, modeling the real-world phenomena, measuring and analyzing, validing hypotheses, and defining another cycle with slightly changed parameters to confirm the existing and creating the new knowledge (Basili et al. 1999).

1.3 Ten Knowledge Areas of Software Project Management

The PMBOK (PMI 2013a) as a set of standard terminology and guidelines for project management is familiar to most professionals and researchers. A PMBOK knowledge area contains the processes that need to be accomplished within its discipline in order to achieve an effective project management program. The software extension of the PMBOK (PMI 2013b) is based on the PMBOK but provides an extension toward commonly accepted practices for managing software projects. Software project management processes are grouped into *initiating, planning, executing, monitoring and controlling,* and *closing*.

The whole body of knowledge of software project management is described by ten knowledge areas. Each area is defined by a set of associated processes. In what follows, and aligned with the software extension of the PMBOK (PMI 2013b), we describe the main content of these knowledge areas. For five areas that are the closest to the content of the book, a more detailed description, including inputs, outputs, and the main tools and techniques (to be) used for the respective processes, is given. These knowledge areas are:

- Time management
- Cost management
- Human resource management
- Communications management
- Risk management

1.3.1 Integration Management

This area coordinates other areas to work together throughout the project and includes the processes and activities to identify, define, combine, unify and coordinate the various processes and project management activities within the project management process groups. Integration management contains management of processes for controlling the project during its life-cycle from execution through completion, as well as controlling successful management of stakeholder expectations. It also includes making choices about resource allocation, making trade-offs among competing objectives and alternatives, and managing the interdependencies among the project management knowledge areas. This knowledge area emphasizes the generally accepted role of a project manager: performing coordination and bringing all the pieces (the deliverables of the project) together. It also refers to the integration of processes and activities (PMI 2013b).

1.3.2 Scope Management

A set of processes is used to ensure that the project includes all the requirements and that no new requirements are added in a way that could harm the project. This knowledge area includes the processes required to ensure that the project includes all the work required, and only the work required, to complete the project successfully and includes setting clearly defined project objectives, defining major project deliverables, and controlling changes to those deliverables. Managing the scope primarily concerns defining and controlling what is and what is not included in the project.

For software, the definition of product scope includes features and quality attributes that are needed and desired by stakeholders. The product scope can be used to estimate the project scope, and constraints on the project scope may determine the product scope. Constraints on both of these may require trade-offs among features, quality attributes, schedule, budget, resources, and technology.

1.3.3 Time Management

Processes are required to manage the timely completion of the project and to ensure that the project is completed on schedule. In software projects, this knowledge area is driven by risk, resource availability, business value, and the scheduling methods used. Specific scheduling methods for software projects introduced by PMI (2013b), including structured scheduling, schedule as an independent variable, iterative scheduling with a backlog, on-demand scheduling, and portfolio management scheduling. All the respective processes with their inputs, outputs, and tools and techniques (to be) used are summarized in Table 1.1.

1.3.4 Cost Management

Project cost management includes the processes involved in planning, estimating, budgeting, financing, funding, managing, and controlling costs so that the project can be completed within the approved budget (PMI 2013a). In other words, this knowledge area includes processes to ensure that the project is completed on budget. Cost management for software projects includes making initial estimates and updating them periodically and may include identifying and forecasting the

Table 1.1 Overview of project time management processes (PMI 2013b)

Process name	Inputs	Outputs	Tools and techniques
Plan schedule management	– Project management plan – Project charter – Enterprise environmental factors – Organizational process assets – Safety and security issues	– Schedule management plan	– Expert judgment – Analytical techniques – Meetings
Define activities	– Schedule management plan – Scope baseline – Enterprise environmental factors – Organizational process assets – Additional factors	– Activity list – Activity attributes – Milestone list	– Decomposition – Rolling wave planning – Experts judgment – Story breakdown structure – Storyboards – Use cases
Sequence activities	– Schedule management plan – Activity list – Activity attributes – Milestone list – Project scope statement – Enterprise environmental factors – Organizational process assets – Architectural and IV & V constraints – Safety and security analyses	– Project schedule network diagrams – Project document updates – Feature sets – Release plans – Architectural and nonfunctional dependencies	– Precedence diagramming method (PDM) – Dependency determination – Applying leads and lags – SAIV and time boxing – Work in progress limits and classes of service – Feature set evaluation – Service-level agreements
Estimate activity resources	– Schedule management plan – Activity list – Activity attributes – Risk register – Activity cost estimates – Resource calendars – Enterprise environmental factors – Organizational process assets	– Activity resource requirements – Resource breakdown structure – Project document updates	– Experts judgment – Alternatives analysis – Published estimating data – Bottom-up estimating – Project management software
Estimate activity durations	– Schedule management plan – Activity list – Activity attributes – Activity resource requirements – Resource calendars – Project scope statement – Risk register – Resource breakdown	– Activity duration estimates – Project document updates	– Experts judgment – Analogous estimating – Parametric estimating – Three-point estimating – Group decision-making techniques – Reserve analysis

(continued)

Table 1.1 (continued)

Process name	Inputs	Outputs	Tools and techniques
	structure – Enterprise environmental factors – Organizational process assets		
Develop schedule	– Schedule management plan – Activity list – Activity attributes – Project schedule network diagrams – Activity resource requirements – Resource calendars – Activity duration estimates – Project scope statement – Risk register – Project staff assignment – Resource breakdown structure – Enterprise environmental factors – Organizational process assets	– Schedule baseline – Project schedule – Schedule data – Project calendar – Project management plan updates – Project document updates – Release and iteration plan updates	

cost of maintaining and evolving a software product plus listening or updating commercially acquired components over many years (PMI 2013b). All the respective processes with their inputs, outputs, and tools and techniques (to be) used are summarized in Table 1.2.

1.3.5 Quality Management

Project quality management includes the processes and activities of the performing organization that determine the quality policies, objectives, and responsibilities so that the project will satisfy the needs for which it was undertaken. Project quality management uses policies and procedures to implement, within the project's context, the organization's quality management system and supports continuous process improvement activities as undertaken on behalf of the performing organization (PMI 2013a). This area also includes developing plans to ensure that project requirements, including product requirements, are met and validated. Quality management can ultimately establish a quality policy, help understand quality principles introduced by quality experts, develop quality assurance processes, and control the quality of all project deliverables.

Table 1.2 Project cost management overview of processes (PMI 2013b)

Process name	Inputs	Outputs	Tools and techniques
Plan cost management	– Project management plan – Project charter – Enterprise environmental factors – Organizational process assets	– Cost management plan – Accuracy of estimate – Units of measure – Cost performance measurement methods	– Expert judgment – Analytical techniques – Meetings
Estimate costs	– Cost management plan – Human resource management plan – Scope baseline – Project schedule – Risk register – Enterprise environmental factors – Organizational process assets – Software size and complexity – Rate of work	– Activity cost estimates – Basis of estimates – Project document updates	– Experts judgment – Analogous estimating – Parametric estimating – Bottom-up estimating – Three-point estimates – Reserve analysis – Cost of quality – Project management software – Vendor bid analysis – Group decision-making techniques – Time-boxed estimating – Function point and source line of code estimating – Story point and use-case point estimating – Estimating reusable code effort – Price to win
Determine budget	– Cost management plan – Scope baseline – Activity cost estimates – Basis of estimates – Project schedule – Resource calendars – Risk register – Agreements – Organizational process assets	– Cost baseline – Project funding requirements – Project document updates	– Cost aggregation – Reserve analysis – Experts judgment – Historical relationships – Funding limit reconciliation
Control costs	– Project management plan – Project funding requirements – Work performance data – Organizational process assets	– Work performance information – Cost forecast – Change requests – Project management plan updates – Project document updates – Organizational process assets updates	– Earned value management – Forecasting – To-complete performance index – Performance reviews – Project management software

Software quality has been a fundamental issue from the early days of developing algorithms. Software quality models include process quality, internal and external product quality, quality in use, data quality, and quality of the software code. The complexities of software quality have led to a number of quality models, such as those in ISO/IEC 25000 and other standards (PMI 2013b).

1.3.6 Human Resource Management

Project human resource management includes the processes that organize, manage, and lead the project team. This knowledge area includes all the processes used to develop, manage, and put the project team together. This involves identifying project stakeholders, developing the project team, motivating the team, and understanding management styles and organizational structure.

Software project staffs collaborate to solve novel problems with incomplete information. Software project managers usually put less emphasis on directing the work and more on facilitating the efficiency and effectiveness of project teams, and solving the fitness problem of each team member within the team is critical due to the interaction and communication needs of software projects. All the respective processes with their inputs, outputs, and tools and techniques (to be) used are summarized in Table 1.3.

1.3.7 Communications Management

Project communications management includes the processes that are required to ensure the timely and appropriate planning, collection, creation, distribution, storage, retrieval, management, control, monitoring, and ultimate disposition of project information (PMI 2013a). This knowledge area determines what information is needed, how that information will be sent and managed, and how project performance is reported. This involves planning and distributing information correctly and to the appropriate stakeholders, reporting the performance, managing stakeholders, and developing processes to ensure effective transfer of information. Communications management is mainly about effective communication among those involved in the project, from stakeholders to internal project interests at different levels and with different views.

The role of project communication is a primary consideration for software projects because teams of individuals who engage in closely coordinated, intellectual activities develop software. With no physical product to reference, effective communication is paramount for keeping team members productively engaged and stakeholders informed (PMI 2013b). All the respective processes with their inputs, outputs, and tools and techniques (to be) used are summarized in Table 1.4.

Table 1.3 Project human resource management overview of processes (PMI 2013b)

Process name	Inputs	Outputs	Tools and techniques
Plan human resource management	– Project management plan – Activity resource requirements – Enterprise environmental factors – Organizational process assets	– Human resource management plan	– Organization charts and position descriptions – Networking – Organizational theory – Expert judgment – Meetings
Acquire project team	– Human resource management plan – Enterprise environmental factors – Organizational process assets	– Project staff assignments – Resource calendars – Project management plan updates	– Preassignment – Negotiation – Acquisition – Virtual teams – Multi criteria decision analysis
Develop project team	– Human resource management plan – Project staff assignments – Resource calendars	– Team performance assessments – Enterprise environmental factors updates	– Interpersonal skills – Training – Team-building activities – Ground rules – Colocation – Recognition and rewards – Personnel assessment tools – Additional tools and techniques
Manage project team	– Human resource management plan – Project staff assignments – Team performance assessments – Issue log – Work performance reports – Organizational process assets	– Change requests – Project management plan updates – Project documents updates – Enterprise environmental factors updates – Organizational process assets updates	– Observation and conversation – Project performance appraisals – Conflict management – Interpersonal skills – Additional consideration

1.3.8 Risk Management

Project risk management includes the processes of conducting risk management planning, identification, analysis, response planning, and risk control in a project. The objectives of project risk management are to increase the likelihood and impact of positive events and decrease the likelihood and impact of negative events in the project (PMI 2013a). This area includes identifying potential project risk events, using qualitative and quantitative analysis to prioritize potential risks, respond to risk situations, and develop risk monitoring and controlling processes. This area can

Table 1.4 Project communications management overview of processes (PMI 2013b)

Process name	Inputs	Outputs	Tools and techniques
Plan communication management	– Project management plan – Stakeholder register – Enterprise environmental factors – Organizational process assets	– Communication management plan – Project documents update	– Communication requirements analysis – Communication technology – Communication models – Communication methods – Meetings
Manage communication	– Communication management plan – Work performance reports – Enterprise environmental factors – Organizational process assets – Release and iteration plans	– Project communications – Project management plan updates – Project documents updates – Organizational process assets updates – Special communication tools – Update information radiators	– Communication technology – Communication models – Communication methods – Information management systems – Performance reporting – Information radiators – Velocity – Historical velocity – Online collaboration tools
Control communication	– Project management plan – Project communications – Issue log – Work performance data – Organizational process assets – Prioritized backlog – Velocity statistics and projections	– Work performance information – Change requests – Project management plans updates – Organizational process assets updates – Iteration and release plan updates – Reprioritized backlog	– Information management systems – Expert judgment – Meetings – Considerate communications – Automated systems

be defined as a proactive approach to risk management in which the project team and the project manager actively discuss potential risk situations that will make difference between a smooth-flowing project and a project filled with surprises and potential disasters.

Each software development project has different uncertainties and risks, because each project is a unique combination of requirements, design, and construction, resulting in distinct software products (the uncertainty arises from a lack of information, and risk is a potential issue). Software risk management aims to improve the probability of achieving the project goals; software opportunity management aims to exceed the project goals. Opportunity management is commonly applied in software project management, especially in adaptive projects that have the opportunity to respond to customer-requested changes, apply new technology, or receive additional resources (PMI 2013b). All the respective processes with their inputs, outputs, and tools and techniques (to be) used are summarized in Table 1.5.

1.3.9 Procurement Management

Project procurement management includes the processes necessary to purchase or acquire products, services, or results needed from outside the project team to complete the project objectives. This includes determining what goods and services should be purchased or developed internally by an organization, planning purchases, and developing procurement documentation such as requests for proposals. It also involves determining appropriate contract types, negotiating terms, selecting sellers, managing contracts through implementation, and then managing project closure and contractual closure.

This knowledge area addresses planning, conducting, controlling, and closing out software project procurements. It also addresses the acquisition of commercially available software for use in a software project. Licensing of software packages, obtaining of rights to modify open source software, the reuse of existing components, and the purchase of specialty services to build software are all elements of software procurement. Software may also be procured as a service. Just as with commercially available software, it is important to understand the exact nature of the services provided; how they might evolve over time; and what control the customer retains over the data provided to be processed under the service, the results obtained, and any security obligations. These considerations are usually covered in a service-level agreement. Often, the standard agreement issued by the provider may not meet the acquirer's specific needs (PMI 2013b).

1.3.10 Stakeholder Management

Project stakeholders include anyone influenced by or influencing the result of a (software) project. The involvement of different types of customers, developers, management, shareholders, competitors, as well as standards and legislations are important for the execution and success of the project. Project stakeholder management includes the processes to ensure the identification of stakeholders and to plan, manage, and control their engagement, which is required to identify the people, groups, or organizations that could impact or be impacted by the project, to analyze stakeholder expectations and their impact on the project, and to develop appropriate management strategies for effectively engaging stakeholders in project decisions and execution. Stakeholder management also focuses on continuous communication with stakeholders to understand their needs and expectations,

Table 1.5 Project risk management overview of processes (PMI 2013b)

Process name	Inputs	Outputs	Tools and techniques
Plan risk management	– Project management plan – Project charter – Stakeholder register – Enterprise environmental factors – Organizational process assets	– Risk management plan	– Analytical techniques – Expert judgment – Meetings – Additional consideration
Identify risks	– Cost management plan – Schedule management plan – Quality management plan – Human resource management plan – Scope baseline – Activity cost estimates – Activity duration estimates – Stakeholder register – Project documents – Procurement documents – Enterprise environmental factors – Organizational process assets – Risk taxonomies	– Risk register	– Documentation reviews – Information gathering techniques – Checklist analysis – Assumptions analysis – Diagramming techniques – SWOT analysis – Expert judgment – Retrospective meetings
Perform qualitative risk analysis	– Risk management plan – Scope baseline – Risk register – Enterprise environmental factors – Organizational process assets	– Project documents updates	– Risk probability and impact assessment – Probability and impact matrix – Risk data quality assessment – Risk categorization – Risk urgency assessment – Expert judgment – Additional consideration
Perform quantitative risk analysis	– Risk management plan – Cost management plan – Schedule management plan – Risk register – Enterprise environmental factors – Organizational process assets	– Project management plan updates – Project documents update – Additional consideration	– Data gathering and representation techniques – Quantitative risk analysis and modeling techniques – Expert judgment

(continued)

Table 1.5 (continued)

Process name	Inputs	Outputs	Tools and techniques
Plan risk responses	– Risk management plan – Risk register	– Project management plan updates – Project documents update – Additional consideration	– Strategies for negative risks or treats – Strategies for positive risks or opportunities – Contingent response opportunities – Expert judgment – Additional consideration
Monitor and control risks	– Project management pan – Risk register – Work performance data – Work performance reports	– Work performance information – Change requests – Project management plan updates – Project documents updates – Organizational process assets updates	– Risk reassessment – Risk audits – Variance and trend analysis – Technical performance measurement – Reserve analysis – Meetings

addressing issues as they occur, managing conflicting interests, and fostering appropriate stakeholder engagement in project decisions and activities.

Stakeholder management is critical for achieving successful outcomes for software projects because software has no physical presence and is often novel. Software is difficult to visualize until it is demonstrated. In addition, there often exists a gulf of expectation between what a customer or product owner states and what the developer interprets. Misalignments among stakeholders represent a major risk to the successful completion of software projects (PMI 2013b).

1.4 The Book's Coverage of the PMBOK Knowledge Areas

In Table 1.6, we define each chapter's main scope related to the established PMBOK knowledge areas. It shows that the collection of chapters is in good match with the project management knowledge areas introduced by the PMBOK (PMI 2013a). Some of the chapters are related to more than one knowledge area. Chapter 8 is the only exception to this. This chapter is more related to "The Standard for Portfolio Management" (PMI 2013c), published recently in response to the increasing acceptance of portfolio management. On the other hand, almost all the knowledge areas are covered by at least one chapter, with most of them having multiple connections. Again, there is one exception, and this is the area of procurement management, being outside the scope of this book.

Table 1.6 Book chapters vs. PMBOK knowledge areas

Chapter	Title	PMBOK knowledge areas
2	Rethinking Success in Software Projects: Looking beyond the Failure Factors	– Communications management – Time management – Cost management – Scope management – Risk management – Stakeholder management
3	Cost Prediction and Software Project Management	– Cost management
4	Human Resource Allocation and Scheduling for Software Project Management	– Human resource management – Time management
5	Software Project Risk and Opportunity Management	– Risk management – Cost management
6	Model-Based Quality Management of Software Development Projects	– Quality management
7	Supporting Project Management Through Integrated Management of System and Project Knowledge	– Integration management – Scope management – Time management – Quality management – Communications management – Risk management – Stakeholder management
8	A Framework for Implementing Product Portfolio Management in Software Business	– Not in the domain of PMBOK knowledge areas, but related to portfolio management
9	Managing Global Software Projects	– Communications management – Human resource management
10	Motivating Software Engineers Working in Virtual Teams Across the Globe	– Human resource management
11	Agile Project Management	– Time management – Human resource management – Communications management – Integration management – Stakeholder management
12	Distributed Project Management : Ten Misconceptions That Might Kill Your Distributed Project	– Communications management – Time management – Cost management – Human resource management
13	Management and Coordination of Free/Open Source Projects	– Human resource management – Communications management – Integration management
14	Inner Source Project Management	– Communications management – Integration management – Time management – Cost management
15	Search-Based Software Project Management	– Human resource management – Time management – Cost management
16	Social Media Collaboration in Software Projects	– Communications management – Stakeholder management

(continued)

Table 1.6 (continued)

Chapter	Title	PMBOK knowledge areas
17	Process Simulation: A Tool for Software Project Managers?	– Time management – Cost management – Risk management
18	Occam's Razor and Simple Software Project Management	– Cost management

1.5 The Multidisciplinary Nature of Project Management

Project management (PM) is seen by Kwak and Anbari (2008) as the integration and application of what was called *allied disciplines*. The authors analyzed the past, current, and future trends of the allied disciplines by exploring, identifying, and classifying the top management journal articles related to project management research. The authors defined eight main areas (allied disciplines) and conducted some research trend analysis. They studied 537 articles published between 1950 and June 2007. A ranking of the disciplines according to the number of studies is presented in Table 1.7.

Kwak and Anbari (2008) predicted that project management becomes more multidisciplinary and flexible in adopting tools from other disciplines. As documented by the book, this also applies to software project management.

Accurate planning and estimation of cost and schedule is difficult for all kinds of projects, but it is particularly difficult for software projects because of the cognitive nature of work, the wide variety of productivity among individuals, the poor requirement definition, and the data inaccuracy of past projects. In the study of allied disciplines, the authors predict that *Strategy/Integration/Portfolio Management/Value of Project Management and Marketing* and *Quality Management/Six Sigma/Process Improvement* should have a growing impact on project management as business strategies are developed and qualities are measured and analyzed to plan and implement effective project management. Chapters 6, 7, and 8 are contributions in this direction.

According to PMI (2013b), the creation of software requires innovative problem solving to create unique solutions. Software projects are more akin to research and development projects than to construction and manufacturing projects. From an allied disciplines point of view, *Technology Applications/Innovation/New Product Development/Research and Development* as well as *Performance Management/Earned Value Management/Project Finance and Accounting* are poised to make major breakthroughs given the recent organizational interest and institutional determination in achieving project success (Kwak and Anbari 2008). Chapters 2, 3, and 5 are contributions in this direction.

Operations Research/Decision Sciences/Operation Management/Supply Chain Management, Performance Management/Earned Value Management/Project

Table 1.7 Ranking of PM allied disciplines according to Kwak and Anbari (2008)

Rank	Discipline	Percentage of publications
1	Strategy/Integration/Portfolio Management/Value of Project Management and Marketing	30
2	Operations Research/Decision Sciences/Operation Management/ Supply Chain Management	23
3	Organizational Behavior/Human Resources Management	13
4	Information Technology/Information Systems	11
5	Technology Applications/Innovation/New Product Development/ Research and Development	11
6	Performance Management/Earned Value Management/Project Finance and Accounting	7
7	Engineering and Construction/Contracts/Legal Aspects/Expert Witness	3
8	Quality Management/Six Sigma/Process Improvement	2

Finance and Accounting, Information Technology/Information Systems, and Technology Applications/Innovation/New Product Development/Research and Development will work together to deliver tools and techniques to allow the “science” of planning, scheduling, and cost control to function in a real project delivery environment. Chapters 4, 15, 17, and 18 are contributions in this direction.

1.6 The Future of Software Engineering

Boehm predicted that in response to the increasing criticality of software within systems and the increasing demands being put on twenty-first century systems, systems and software engineering processes will evolve significantly over the next two decades (Boehm 2006a). By analyzing today’s trends, Goff stated that the average enterprise in 2025, even with innovation, market pressures, and significant effort, will still be struggling to adapt with today’s leading technologies (Goff 2009).

In the case of predicting the future in software project management, both trends of project management and software engineering could be helpful. The ten trends in software engineering discussed in Boehm and Lane (2010) are

1. Increasing emphasis on rapid development and adaptability
2. Increasing software criticality and the need for quality assurance
3. Increased complexity, global systems of systems, and the need for scalability and interoperability
4. Increased needs to accommodate Commercial-Off-The-Shelf (COTS) software services and legacy systems

5. Increasingly large volumes of data and ways to learn from them
6. Increased emphasis on users and end value
7. Computational plenty and multicore chips
8. Increasing integration of software and systems engineering
9. Increasing software autonomy
10. Combinations of biology and computing

Technology trends and the continuing need for product differentiation, globalization, and its effect on the market and processes for new technology introduction accelerate system change, which makes the trade-off between the speed development and the costs of development a necessity. Nidiffer and Dolan (2005) stated that the ever-increasing growth and complexity of software-intensive systems and the appearance of geographically distributed systems are today's trends. Geographically distributed projects let managers compress schedules by employing larger workforces than could fit in a single location, using time zone differences to increase the number of productive work hours in a day, and securing scarce resources such as knowledge experts and other specialized resources no matter where they reside. However, these benefits come with increased risks because of the lack of face-to-face communication, in particular the potential loss of trust, collaboration, and communication richness.

On the other hand, as a probable situation, computational plenty will spawn new types of applications and platforms. These will present process-related challenges for specifying their configurations and behavior; generating the resulting applications; verifying and validating their capabilities such as performance and dependability; and integrating them into even more complex systems of systems. All these may need minor changes in the domain of project management. Boehm (2006b) indicates that key challenges include cross-cultural bridging; the establishment of a common shared vision and trust; contracting mechanisms and incentives; handovers and change synchronization in multi-time zone development; and culture-sensitive collaboration-oriented groupware.

1.7 Software Project Management: Past and Future

While the era of modern project management as a discipline started in the 1950s, it was by about 1980 that the software development industry started implementing some established project management practices. During this period, Barry Boehm defined the whole field of software engineering economics by introducing COCOMO, the Constructive Cost Model for software (Boehm 1981). In 1984, The *Project Management Institute* (PMI) was offering its certification program for the first time. By the 1990s, project management theories and practices accepted widely, and project management was known as a profession.

More recently, project management has become a factor present at all levels, in all divisions of many companies. Goff (2009) discussed visions for the project management software industry from an industrial point of view. The following trends were projected:

- Project portfolio management
  Currently there are several vendors that create tools to support portfolios in the case of managing integration and it should support the future of project management.
- Collaboration with the means of virtual teams and social networks
  Combination of virtual teams with social networks in conjunction with more advanced tools supporting collaboration and communication creates new opportunities of working together as virtual teams, not being colocated. This is projected to result in a massive increase in the market size for PM software.
- Mastery of real-time tracking
  Time tracking based on a time sheet is a critical success factor for many teams. Earned value management becomes more useful if retrospective analysis moves from being done one month after the work towards being done only one day afterwards, in combination with having much more accurate and reliable information available.
- Capture and reuse of project knowledge
  Developing the project schedule and project plan once would be really great; in fact all successive projects reuse the materials as templates. This reuse could be applied in project documents such as test plans or requirements, too.
- Dashboards and project intelligence
  Project tracking may be based on easy-to-measure trailing indicators instead of critical indicators. Monitoring the project status in an efficient way will help in project communication improvement and smarter decisions.
- Project management absorbed into enterprise systems
  More and more connectivity to other enterprise system functionality. Strongest ties are to enterprise resource planning (ERP) systems, but links and increasing degree of overlap also exist with customer relationship management (CRM), supply chain management (SCM), and product life cycle management (PLM) systems.

1.8 This Book

Software Project Management in a Changing World is not “just another book” in the area of software project management. It brings together the various current directions within the discipline, which have been so far presented mainly in individual articles or books. Whenever appropriate, the content of the book is based on evidence coming from empirical evaluation of the proposed approaches.

There is already a great variety of books and publications offering guidelines and best practices. To keep the content of the book focused, the implicit assumption here is that we do not address small-scale projects. For professionals, the book is intended to be a source of inspiration to refine their project management skills into new areas. For researchers and graduate students, the book presents some of the most recent methods and techniques to accommodate the new challenges of the discipline. The goal of the book is to find a good balance between new results and putting together existing material to allow its usage in new contexts.

The book consists of four parts, preceded by a general introduction. Each of the parts consists of a sequence of chapters. All the four parts start with a brief overview outlining its content.

Introduction

1. Software Project Management: Setting the Context by Günther Ruhe and Claes Wohlin

Part I: Fundamentals

2. Rethinking Success in Software Projects: Looking Beyond the Failure Factors by Darren Dalcher
3. Cost Prediction and Software Project Management by Martin Shepperd
4. Human Resource Allocation and Scheduling for Software Project Management by Constantinos Stylianou and Andreas S. Andreou
5. Software Project Risk and Opportunity Management by Barry Boehm

Part II: Supporting Areas

6. Model-based Quality Management of Software Development Projects by Jens Heidrich, Dieter Rombach and Michael Kläs
7. Supporting Project Management through Integrated Management of System and Project Knowledge by Barbara Paech, Alexander Delater and Tom-Michael Hesse
8. A Framework for Implementing Product Portfolio Management in Software Businesses by Erik Jagroep, Sjaak Brinkkemper, Inge van de Weerd and Ton Dobbe
9. Managing Global Software Projects by Christof Ebert
10. Motivating Software Engineers Working in Virtual Teams across the Globe by Sarah Beecham

Part III: New Paradigms

11. Agile Project Management by Tore Dybå, Torgeir Dingsøyr and Nils Brede Moe
12. Distributed Project Management by Darja Šmite
13. Management and Coordination of Free/Open Source Projects by Ioannis Stamelos
14. Inner Source Project Management by Martin Höst, Klaas-Jan Stol and Alma Oručević-Alagić

Part IV: Emerging Techniques

15. Search-Based Software Project Management by Filomena Ferrucci, Mark Harman and Federica Sarro
16. Social Media Collaboration in Software Projects by Rachel Harrison and Varsha Veerappa
17. Process Simulation: A Tool for Software Project Managers? by Dietmar Pfahl
18. Occam's Razor and Simple Software Project Management by Tim Menzies

Software project management is a dynamically evolving discipline, which is constituted of a wide range of sub disciplines. Each of them covers specific practices, methods, and tools. Even though we have covered a broad range of topics, the book is not intended to be comprehensive. The selection of the chapters was primarily based on the perceived importance of the topics, although it was impossible to cover all topics of interest. For example, even though there exists a large variety of proprietary and open source tools (Pereira et al. 2013), we considered software project management tools in general being outside the scope of the book. Also, some recent trends such as the one of utilizing the power of predictive analytics (Hassan 2013) or of visualization of project data (Novais et al. 2013) for the purpose of qualifying project management were not included.

The book is intended to attract readers from both academia and practice. The targeted benefits for the readers are

- Getting an overview of the most recent methods and techniques
- Support for better decision-making and providing inspiration when conducting the activities related to project management in new contexts
- Learning about the most recent trends and understanding the implications of their implementations

References

Basili VR, Shull F, Lanubile F (1999) Building knowledge through families of experiments. IEEE Trans Softw Eng 25(4):456–473

Boehm BW (1981) Software engineering economics. Prentice Hall, Englewood Cliffs, NJ

Boehm BW (2006a) Some future trends and implications for systems and software engineering processes. Syst Eng 9(1):1–19

Boehm BW (2006b) A view of 20th and 21st century software engineering. In: 28th international conference on software engineering, New York, pp 12–29

Boehm BW, Lane JA (2010) Evidence-based software processes. In: International conference on software process, ICSP 2010. LNCS, vol 6195. Springer, Berlin, pp 62–73

Brooks FP (1987) No silver bullet: essence and accidents of software engineering. IEEE Comput 20(4):10–19

Goff SA (2009) Visions for the project management software industry. In: Cleland D, Bidanda B (eds) Project management circa 2025. Project Management Institute, Athabasca University, Newtown Square, PA, pp 1–15

Hassan AE (2013) Software analytics: going beyond developers. IEEE Softw 30(4):53

Kruchten P (2011) The frog and the octopus – a model of software development. CSI Commun 35 (4):12–15

Kwak YH, Anbari FT (2008) Impact on project management of allied disciplines: trends and future of project management practices and research. Project Management Institute, Newtown Square, PA

Larman C, Basili VR (2003) Iterative and incremental development: a brief history. Computer 36 (6):47–56

Nidiffer KE, Dolan D (2005) Evolving distributed project management. IEEE Softw 22(5):63–72

Novais RL, Torres A, Mendes TS, Mendonça M, Zazworka N (2013) Software evolution visualization: a systematic mapping study. Inf Softw Technol 55:1860–1883

Pereira AM, Goncalves RQ, von Wangenheim CG, Buglione L (2013) Comparison for open source tools for project management. Int J Softw Eng Knowl Eng 23:189–209

PMI (2013a) A guide to the project management body of knowledge (PMBOK® guide), 5th edn. Project Management Institute, Newtown Square, PA

PMI (2013b) Software Extension to the PMBOK Guide, 5th edn. Project Management Institute, IEEE Computer Society, Newtown Square, PA

PMI (2013c) The Standard for Portfolio Management, 3rd edn. Project Management Institute, Newtown Square, PA

Biography Günther Ruhe holds an Industrial Research Chair in Software Engineering at University of Calgary. Dr. Ruhe received a doctorate rer. nat degree in Mathematics with emphasis on Operations Research from Freiberg University and a doctorate habil. nat. degree (Computer Science) University of Kaiserslautern. From 1996 until 2001, he was the deputy director of the Fraunhofer Institute for Experimental Software Engineering (Fh IESE). Ruhe received an iCORE research award for the period 2001 to 2007. Since 2007, he had served as an Associate Editor of the *Journal of Information and Software Technology*, published by Elsevier. His main research interests are in the areas of Product Release Planning, Software Project Management, Empirical Software Engineering as well as Search-based Software Engineering. He is a Senior member of IEEE and a member of the ACM. Dr. Ruhe is the Founder and CEO of Expert Decisions Inc., a University of Calgary spin-off company created in 2003.

Claes Wohlin is Professor of Software Engineering at Blekinge Institute of Technology, Sweden. From January 1, 2014, he has been serving as the Dean for the Faculty of Computing. Professor Wohlin is a guest professor at Shandong University at Weihai in China. Prior to joining BTH in 2000, he held professor chairs at Lund and Linköping Universities. He has been a Visiting Professor at Chalmers University of Technology in Göteborg (2005–2008) and at the University of New South Wales in Sydney, Australia (2009–2011). Professor Wohlin received a Ph.D. degree in Communication Systems from Lund University in 1991. Since January 2008, Professor Wohlin has been Editor-in-Chief of the *Journal of Information and Software Technology*, published by Elsevier, and he has been the Co-Editor-in-Chief of the journal since 2001. In 2011, Claes Wohlin was elected member of the Royal Swedish Academy of Engineering Sciences. He is a senior member of IEEE.

Part I
Fundamentals

Introduction

Software project management as such implies managing a set of fundamental aspects in relation to the development and evolution of software as outlined in Chap. 1. In this part of the book, we have invited some of the leading experts in relation to a set of fundamental areas for a software project manager to master, and they share their knowledge, insights and accompanying recommendations and conclusions in four chapters in this part of the book.

In Chap. 2, Darren Dalcher challenges us to rethink the definition of software project management success. The chapter starts off by summarizing some of the literature in reporting on the failures in relation to software projects. The data clearly indicates that it is a daunting task to succeed with your software project, in particular if it is a large project. He goes on to explain how the project manager all too often is not really in charge of the main success factors: software performance in terms of what the software should achieve, cost of development and delivery time. Dalcher describes the need to have multiple categories for success, for example, project success and product success. Based on this reasoning, the chapter presents a four-level model of success in relation to software projects and their output. Some examples are presented to highlight how the perception of failure or success may change with the levels as well as over time.

Martin Shepperd provides a review of software project cost prediction in Chap. 3. He starts by discussing some of the main reasons for the problems in relation to cost prediction: 1) complexity of the development, 2) software development is a design activity, 3) estimates are needed early and 4) the development is often put under social and political pressure. The chapter continues by reviewing some of the techniques for cost prediction both formal models and expert judgment. Challenges in relation to both techniques for cost prediction are discussed along with the need to consider both the people and formal aspects. Particular emphasis is given to the problems that arise from cognitive biases, that is, heuristics that cause even experts to deviate from optimal or logical decision-making. Based on the discussion, Shepperd provides some recommendations in relation to improving the practice of software cost prediction.

In Chap. 4, Constantinos Stylianou and Andreas S. Andreou address some of the issues in relation to human resource allocation and scheduling in software projects. Given that software development is a design activity, the human aspect of development and management is crucial. Stylianou and Andreou focus on human resources from a planning perspective, including assigning developers and teams to tasks within the development. The chapter provides an overview of some recent approaches to human resource allocation and scheduling. The most common general approaches in research relate to using different specific techniques in relation to mathematical modelling and computational intelligence. The chapter provides summaries and references to specific approaches to human resource allocation and scheduling. It ends with a discussion on the shift towards also incorporating non-technical factors in human resource allocation and scheduling and, in particular, the adoption of a more human-centric approach by using software developers' personality types to allocate tasks and form teams.

Barry Boehm has authored the final chapter (Chap. 5) in this part. The chapter discusses risk and opportunity management in relation to software projects. Risk is an uncertain condition and event, which may jeopardize the success of the software project. Boehm describes a number of aspects to be addressed to manage risk. He discusses the duality between risk and opportunity, where risks may generate losses and opportunities may result in gains. He describes and illustrates how risk and opportunity exposure can be managed. Boehm highlights that not only risks must be managed, but also the opportunities. He continues by discussing the joint management of risks and opportunities. Furthermore, he introduces the concept of lean risk management plans and discusses risk tracking.

The four chapters in this part give an in-depth insight into some of the challenges related to a selection of fundamental areas for anyone conducting research into software project management or managing a software project.

Chapter 2
Rethinking Success in Software Projects: Looking Beyond the Failure Factors

Darren Dalcher

Abstract The notions of success and failure in software projects are confusing. Failure is often considered in the context of the iron triangle as the inability to meet time, cost, and performance constraints. While there is a consensus around the prevalence of project failure, new projects seem destined to repeat past mistakes. This chapter tries to advance the discussion by offering a new perspective for reasoning about the meaning of success and the different types of software project failures.

In order to court project success, practitioners need to rise beyond a fixation with the internal parameters of efficiency, thus bringing forth the effectiveness required to secure project success. The chapter begins by discussing the limited insights from existing project failure surveys, before offering a four-level model addressing the essence of successful delivery and operation in software projects. Following consideration of outcomes and time, the chapter offers a series of vignettes and mini case studies that highlight the rich interplay between the four levels of success, before addressing the types of measures underpinning the four levels and the need to develop a multi-dimensional perspective to obtain a more accurate picture regarding the success of a project.

2.1 The Extent of Software Project Failures

The popular computing literature is awash with stories of software development failures and their adverse impacts on individuals, organisations, and societal infrastructure. Indeed, contemporary software development practice is regularly

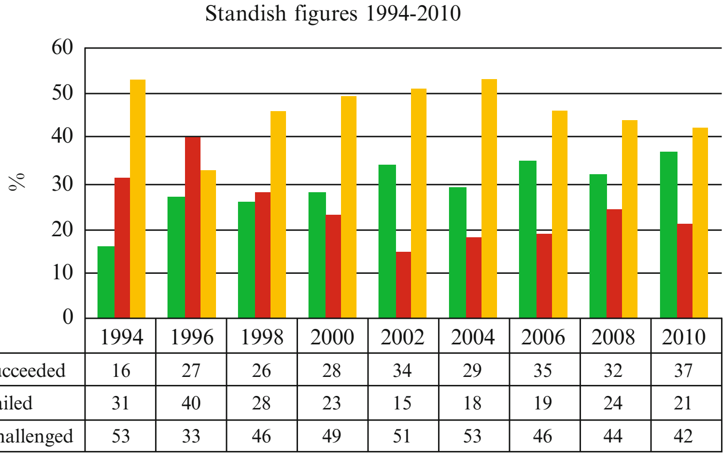
D. Dalcher (✉)
NCPM, University of Hertfordshire, MacLaurin Building, 4 Bishops Square, Hatfield, Hertfordshire AL10, 9AB, UK
e-mail: d.dalcher2@herts.ac.uk

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_2, © Springer-Verlag Berlin Heidelberg 2014

characterised by runaway projects, late delivery, exceeded budgets, reduced functionality, and questionable quality that often translate into cancellations, reduced scope, and significant rework cycles (Dalcher 1994). The net result is an accumulation of waste typically measured in financial terms. For example, in 1995, failed U.S. projects cost $81 billion, with an additional $59 billion of overspend, totalling $140 billion (Standish 2004). Capers Jones contended that the average U.S. cancelled project was a year late, having consumed 200 % of its expected budget at the point of cancellation (1994). In 1996, failed projects alone totalled an estimated $100 billion (Luqi and Goguen 1997). In 1998, 28 % of projects failed, at a cost of $75 billion, while in 2000, 65,000 U.S. projects were reported to be failing (Standish 2000). McManus and Wood-Harper (2008) reported that the cost of software project failure across the European Union in 2004 was €142 billion. More recently, a McKinsey–Oxford survey of more than 5,400 software projects revealed that half of all projects significantly fail on budgetary assessment, while 17 % of projects actually threaten the very existence of the company, with the average project running 45 % over budget and 7 % behind schedule, while delivering 56 % less functionality than predicted (Bloch et al. 2013). According to the report, achieving $15 million in benefits now requires an average spending in excess of $59 million.

Yet, software project failure is not a new phenomenon. The first indications of the problem and mention of the term ‘software crisis’ were made during the NATO conferences in 1968 and 1969 (Naur and Randell 1968; Buxton et al. 1969). Indeed, conference attendees reported a set of symptoms that would resonate with the issues raised by developers and managers today. Over 30 years ago, a GAO report in the USA (Anon 1979) showed that there were serious problems associated with the development of software. Less than 2 % of the total value of contracts could be used efficiently as delivered and a further 3 % could only be used after changes. The rest of the projects had the software delivered but never successfully used; the software paid for but not delivered; or the software used but extensively reworked or later abandoned. Moreover, the first edition of the best-selling book in software engineering tells the story of a huge IBM software project with major cost and schedule delays which teetered on the brink of disaster for a number of years from the perspective of the project manager trying to stabilize the project (Brooks 1975). Indeed, the OS360 project came close to bankrupting IBM.

Consultancies and polling organisations have attempted to collect market data about the prevalence of failure. The Standish Group, for example, has been compiling an annual failure survey since 1994. In 1995, 31.1 % of U.S. software projects were cancelled, while 52.7 % were completed late, over budget (cost 189 % of their original budget), and lacked essential functionality (Standish 2000). Only 16.2 % of projects were completed on time and within budget; only 9 % were in larger companies, where completed projects had an average of 42 % of desired functionality (ibid.). The 1996 cancellation figure rose to 40 % (ibid.) before improving to around 15 % in 2002 (see Fig. 2.1). However, the most recent figures reveal that the current failure rate is 21 % (Standish 2011) with 63 % of overall projects labelled as not successful. Note that problems associated with cost



	1994	1996	1998	2000	2002	2004	2006	2008	2010
🟩 Succeeded	16	27	26	28	34	29	35	32	37
🟥 Failed	31	40	28	23	15	18	19	24	21
🟨 Challenged	53	33	46	49	51	53	46	44	42

Fig. 2.1 Standish figures 1994–2010

estimation and the apparent optimism of software project managers are covered in Chap. 3.

While the research approach used by the Standish Group has been challenged over the methodology adopted and its rigour (Glass 2005, 2006; Jørgensen and Moløkken 2006; Eveleens and Verhoef 2010), the figures provide a well-referenced baseline related to the extent of software project failures. Other studies appear to confirm the high failure rates. For example, Taylor (2000) reported that only 130 projects out of 1,027 were considered successful, while a 2004 PriceWaterhouse-Coopers study surveyed 10,640 projects and revealed that only 2.5 % of companies achieve budget, scope, and schedule targets on all their projects. Sauer and Cuthbertson (2003) reported that 16 % of IT projects (with a major emphasis on software development) were considered successful, however Sauer et al. (2007) noted that 67 % of the projects were nonetheless delivered close to budget, schedule, and scope expectations. More recently, McManus and Wood-Harper (2008) discovered that only one in eight IT projects can be considered truly successful, with almost a quarter (23.8 %) cancelled due to issues related to requirements, change, communication, business process alignment, and overspend. Using similar definitions, IBM (2008) reported that only 40 % of projects experienced by 1,500 change management executives met their schedule, budget, and quality targets, while KPMG (2010) observed that 70 % of surveyed organisations in New Zealand had experienced a failure in the previous 12 months. Following interviews with 600 developers, Geneca (2011) reported that 75 % of project participants lacked confidence in project success, admitting that their projects are ‘doomed right from the start’.

Not surprisingly, larger and longer projects fare worse. Figure 2.2 shows the probability of success mapped against project duration relating to a body of 23,000 projects accumulated by the Standish Group. The diagram confirms that the

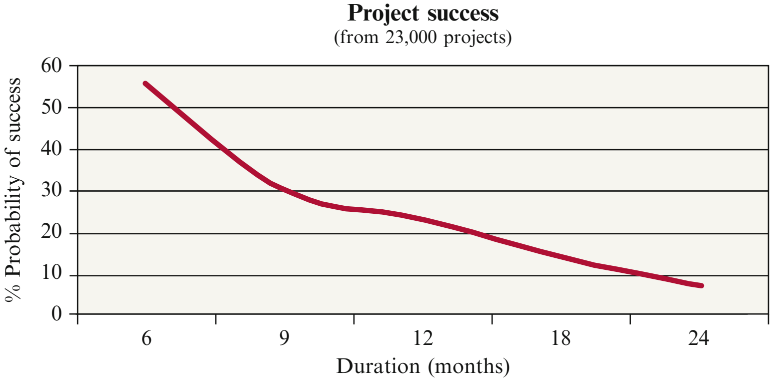


Fig. 2.2 Standish figures: success against project duration

probability of success is much higher for smaller projects; for longer projects, the likelihood of a successful outcome is significantly decreased. Jones (2007) confirms from his detailed studies that the risk of cancellation or major delays rises rapidly as the overall application size (measured in function points) goes up.

Jones (2007, 2008) investigated the likelihood that the average U.S. software project will be cancelled, typically due to cost and schedule overruns, failure to meet requirements, poor planning, estimating, quality control, or excessive requirements creep, relative to size. The results indicate that none of the eight domains investigated are fully successful for large systems of above 10,000 function points in size, showing the average probability of cancellation at 36 %. He warned that the 'development of large applications in excess of 10,000 function points is one of the most hazardous and risky undertakings of the modern world' (Jones 2007). Applications in the region of 100,000 function points are more likely to fail with an average cancellation likelihood reading of 51 %, with some sectors such as Management Information Systems displaying higher failure rates (70 %). Jones (2008, p. 308) concluded that: 'Cancellations, major delays in excess of one calendar year, and cost overruns in excess of 100 % remain endemic problems for software applications in the 100,000 function point size range, and larger.' Jones (2010) further added that: 'large software projects are almost always over budget, usually delivered late, and are filled with bugs when they're finally delivered. Even worse, as many as 35 % of large applications in the 10,000 function point or more size range will be cancelled and never delivered at all.'

Flyvbjerg and Budzier (2011) contended that IT projects are now so big and their influence so wide ranging across many aspects of the organisation, that they pose a singular new kind of risk that can sink entire corporations, cities, and even nations. Their global survey of 1,471 IT change projects showed that while the average cost overrun on large initiatives was 27 %, one in six projects showed a cost overrun of 200 %, on average, and a schedule overrun of almost 70 %. As software is integrated into bigger products and systems, the concerns can become magnified. 'The software industry has the highest failure rate of any so-called engineering field. An occupation that runs late on more than 75 % of projects and cancels as many as 35 % of larger projects is not a true engineering discipline' (Jones 2010).

2.2 Beyond Simple Success Measures

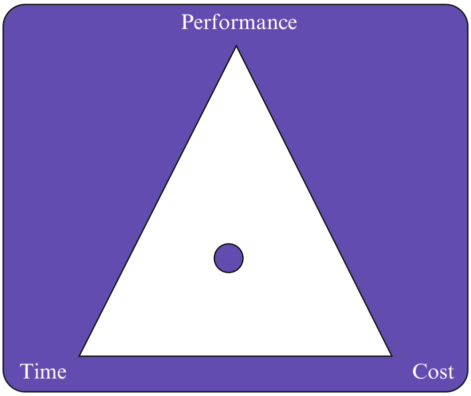
The relationship between success and failure is not clear. Some view the relationship as a binary function so that a project is either successful, or not. The research by McManus and Wood-Harper describes failure as 'those projects that do not meet the original time, cost and requirements criteria'. The Standish Group makes a further distinction between 'failed projects' and 'challenged projects'. *Failed projects* are cancelled before completion, never implemented, or scrapped following installation. *Challenged projects* are completed and operational projects which are over-budget, late, and with fewer features and functions than initially specified. *Successful projects*, in contrast, are completed on time, on budget, with all specified features. Figure 2.1 also shows the relationship between successful, challenged, and failed projects. Observing the Standish figures over the past 19 years would appear to indicate a rough rule of thumb, suggesting a split of 25 % of projects being successful, 50 % being challenged, and 25 % failing.

The Oxford Dictionary defines success as: a favourable outcome; doing what was desired or attempted; the accomplishment of an aim or purpose; or the attainment of wealth or fame or position. Failure is broadly defined as lack of success supporting the idea of a binary relationship. In an attempt to make further sense of the relative positions of success and failure, software surveys have clearly found it useful to introduce the idea of partial failure (or challenged projects) as an intermediate position between success and failure, potentially indicating dissatisfaction with a two-state explanation. Indeed many project outcomes do not fall directly into either category.

The majority of the studies mentioned above define success as meeting all the criteria associated with the budget, schedule, and functionality; with failure viewed as a failure to meet *all* of the same criteria. This implies that if a project is finished on time, within budget, whilst offering the expected functionality, it can be viewed as successful. Conversely, failing to meet any of the criteria will deem it a failure. The view is predicated on the traditional measures applied in project management and generally known as the triple constraint, the golden triangle, or the iron triangle. This idea presupposes high estimation accuracy with regard to the initial formulation of the variables of the triple constraint (Eveleens and Verhoef 2010) when the degree of uncertainty is at its greatest.

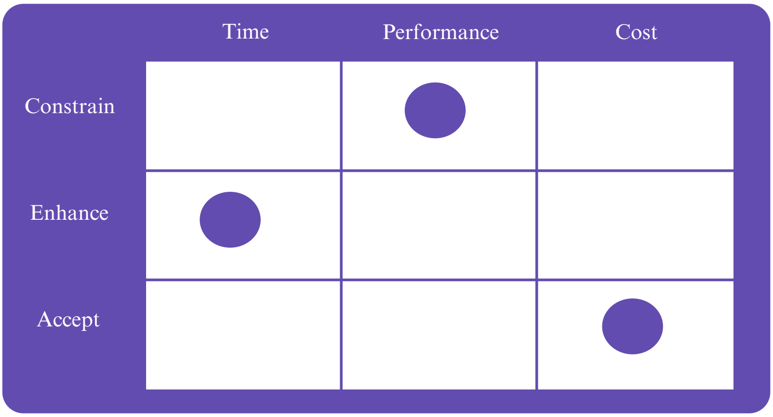
Traditional project management theory holds that optimising the three criteria will result in ideal performance on a project. Typical projects thus require a balancing act between the so-called triple constraints of time, cost, and functionality as expressed in the original triangle conceived by Martin Barnes in 1969 (see Fig. 2.3). Note that the third corner is named 'performance'. The original release named that corner 'quality', but this was soon corrected to performance 'to reflect

Fig. 2.3 Budget, time, and performance trade-off



whatever the finished product was supposed to achieve' (Barnes 2013). Performance means satisfactory function of the product, which has to be fully defined. This could be specified in terms of rate of return, profit, beat the enemy, impress visitors, or in the case of software the project scope and expected functionality. The whole point of the triangle is that the spot can be placed at such a point that its closeness to each corner represents its relative importance and helps the project manager to make informed decisions about the project. Trade-offs and adjustments are therefore made by restricting, adding to, or adjusting the cost, time, and performance associated with a project. The triangle enables managers to consider each decision and its implications on the dimensions of time, cost, and performance and integrate the different project management functions. For example, the more that is requested in terms of performance, the more it is likely to cost and the longer the expected duration. If the client needs to have a certain performance delivered very rapidly, this will increase the cost due to the need to work faster and have more resources involved in the development, albeit with increased communication costs. The more features expected from a system, the higher the cost and the longer the expected duration. Conversely, if the costs need to be kept to a minimum, one may need to consider the essential performance, or the overall project scope, and compromise there (Dalcher and Brodie 2007).

Many managers quickly discover that the triangle is not flexible. The three factors are closely entwined and project managers are expected to balance the **what** (performance) with the **when** (time) and the **how much** (cost). Kloppenborg and Mantel (1990) discerned that the importance assigned to the factors varies systematically according to the life cycle stage. Optimisation and trade-offs will thus depend on the phase during which decisions are made. While tools such as project priority matrices (see Fig. 2.4) may be utilised to prioritise the constraints underpinning the decision process, managers would need to consider the overall priorities and recognise that priorities may shift according to the stage of development. The priority matrix offers a simplifying mechanism for making decisions. The horizontal axis depicts the three key constraints of time, performance, and cost, while the vertical axis covers the suggested treatment options (constrain, enhance,



	Time	Performance	Cost
Constrain		●	
Enhance	●		
Accept			●

Fig. 2.4 Project priority matrix

or accept). In the example below, the proposed project offers a new type of market-leading functionality which must be available in full; that is, it MUST meet the performance criteria—hence fixing, or constraining, that aspect. Given the intention to release before the competition, every effort should be spent on activities that enhance or help project delivery to ensure the product is first to market. The trade-off, therefore, requires that reasonable additional costs are accepted in order to optimise the time criterion, whilst strictly adhering to the performance brief. The matrix offers a structured way of considering the impacts offering a simplified version of the trade-off triangle.

In practice, performance is often determined prior to the project. Moreover, project managers often inherit the overall budget from the contracting activities that may even have imposed a fixed-price contract structure. A fixed overall budget may also exclude typical remedies like the hiring of specialists and the addition of human resources. The only remaining latitude for leverage is in the schedule. However, this may also be imposed on a project through a fixed date for delivery with little regard to the complexity of the intended system or the risks it embodies. Once both budget and schedule are fixed, there appears to be little capacity for compromises and trade-offs.

The three factors clearly play a key part in determining the degree to which a project is challenged (or even deemed a failure); yet they may be uncontrollable by the project manager. Indeed, Capers Jones observed that the most common constraints encountered are: fixed delivery dates; fixed-price contracts; staffing or team size limitations; and performance or throughput constraints (Jones 2000) i.e., fixed time, price, staffing level, performance, and scope. Many managers are thus looking to control other factors that may alter the outcome of the project, in particular as the constraints often occur in concert. Measuring success on the basis of preestablished parameters that cannot be adjusted is therefore of limited value.

Before addressing additional factors in the next section it is also useful to point out that the artefacts of projects, especially the final delivered systems, interact with organisations, customers, stakeholders, and other systems. Their impacts, regardless of whether or not they are delivered on time, can be crucial and perhaps even fatal in financial or real terms. Dalcher reports on the impact of an ambulance despatch system that was delivered to the users, the citizens of a major metropolis (on the third attempt), yet failed in action subsequently, potentially leading to loss of life (Dalcher 2010). Another illustration is a UK disaster which followed an earlier, yet unrelated failure. The delay in introducing the Nirs2 system into the Inland Revenue beginning in 1995 meant that additional backlogs were building up. The backlogs caused the Inland Revenue to stop sending reminders to up to a third of the UK working force, warning them that they needed to top up their national insurance contributions. As a result, around 10 million people face a state pension shortfall. The impacted party includes the lowest paid workers in the UK. While the backlog resulted from a delayed system that itself cost taxpayers millions of pounds, the additional loss will be borne by individuals and only count as a hidden backlog indirectly stemming from another failure. The true cost to individuals is likely to be £15 billion and the hardship that ensues as a result (BBC 2003). Therein lies the complexity of counting the costs of failure. Numerous other compilations have been published identifying major impacts on individuals, organisations, and society at large (see for example, Dalcher 1994, 2012; Flowers 1996; Charette 2005). The success, and failure, of software projects therefore cannot be delimited by a simplified set of factors and constraints associated with the delivery effort.

2.3 Rethinking Project Success

Project success is a rather nebulous concept and the focus on the triple constraint can be too limiting. Indeed, Linberg (1999) asserted that a whole new theory of project success is needed. Pinto and Slevin (1988) noted that success combines issues related to the project itself with issues related to the client. Moreover, software developers and systems analysts have recognised long ago that user involvement, satisfaction and buy-in are crucial to the success of software projects. Prototyping and user-driven approaches were developed to maximise the potential for satisfaction for various stakeholders and thus increase the likelihood of user acceptance of the ultimate system.

Baccarini identified the need to distinguish between project management success and the success of the product which entails dealing with the effects of the project's final delivered product (1999), thereby allaying the need to define a further dimension concerned with client expectations which have already been expressed in the desired performance functionality. Ironically, this chimes with the original (but often misunderstood) intention of the Barnes' triangle (Fig. 2.3) to capture the agreed upon definition of the purpose of the project or how the complete project would perform. Given that the product will be utilised by the client there is a degree of correspondence between the dichotomies put forward by Pinto and Slevin and by Baccarini. Indeed, de Wit (1988) observed that measuring progress and cost are part of project control, which should not be confused with measuring success. Cooke-Davies (2002) likewise made a distinction between the focus on project performance and the need to look at the success of a project.

Having multiple categories of success would suggest that it is possible to be successful in some areas and not successful in others. It thus makes it possible to understand mismatches between the different criteria and groups. Moreover, it implies that the traditional triple constraints of cost, time, and performance only reveal part of the picture. In other words, it may be possible to maximise the traditional criteria and yet deliver a product that is not valued by the users. Likewise, it is also possible to exceed the traditional criteria but deliver a product that is valued and adopted by the user community, despite exceeding the budget or the schedule, or even both.

The discussion thus far indicates that at least two different levels of success can be identified. Indeed, according to Munns and Bjerimi (1996) it is possible to achieve a successful project even when management has failed, and also possible to deliver a failed project following successful management. However most studies and surveys of software project failures tend to focus on the traditional criteria of efficiency embedded through the triple constraints of time, cost and performance. They thus ignore the deeper aspects associated with the delivered product, its perceived utility and value, the expectations and needs of stakeholders, the intended performance of the product, and the project context.

Further evidence of the need to look beyond the traditional criteria is provided in Table 2.1, which summarises an extended and refined set of common issues that were originally identified across six project failures covered in detail in Dalcher and Genus (2003) and extended through a sequence of workshops with practitioners, the mapping of factors in 150 failed projects and a series of four international surveys resulting in the revised figure presented in this chapter. Note: Escalation of commitment and its impact on projects is explained in detail in Sect. 11.3.

The obvious message from the set of issues is that the traditional efficiency criteria as embedded in the triple constraint do not appear to have played a part in the build-up to any of the failures. Instead, the issues identified were more concerned with the product (as well as the assumptions and expectations surrounding it) and the overall business success.

It may also be instructive to scrutinise other domains and sectors. When the UK Government recognised that the construction sector was underachieving, it assigned a task force to determine the causes of the shortfall. The study recommended substantial changes in the culture and structure of the sector (Egan 1998). Crucially, it perceived the need to replace competitive tendering with long-term relationships to address the growing dissatisfaction of both private and public sector clients. The main criticism was reserved for the way projects were assessed as the focus on time, cost, and quality was recognised to be wholly inadequate. Overall, the task force acknowledged that the construction sector had thus far failed

Table 2.1 Groupings of crucial issues observed in failures

Area	Typical additional issues
Relationship management	Vendor–client disagreements, partnerships, long-term perspective, respect, joint working
Trust	Lack of trust, reliance, co-operation
Communication	Information, barriers, exchange, ambiguities
Management of expectations	Stakeholder engagement, needs assessment, involvement
Politics	Organisational politics, blocks, defensive routines
Escalation of commitment	Sunken costs, pressure, escalating investment
Risk management	Exchanging risks, effective transfer
Contract management	Contractual engagement, multiple interpretations, expected obligations

to meet the needs of modern business by failing to look beyond the traditional success criteria to determine the true performance of projects. A 10-year retrospective review re-affirmed that progress had yet to materialise (Egan 2008).

2.4 Towards Multiple Levels of Success

Success, it would appear, needs to be understood at multiple levels in order to appreciate the complex dynamics and subtle impacts. A tabular representation of four levels of success, which builds on the earlier discussion, is offered in Table 2.2.

Level 1 represents project management success and is thus concerned with internal efficiency and performance measurement and optimisation at the project level through the tracking of the cost, schedule and performance parameters. Level 1 success is therefore to do with project delivery against the constraints or measures imposed on the project.

Level 2 is focused on the overall effectiveness of the project through the lens of what is actually being delivered. Success is measured through the utility and acceptability of the output that has been delivered. The benefits of the projects and the achievement of the objectives are thus assessed in terms of the satisfaction of the customer and the different stakeholder groups. Level 2 success reflects the acceptability and impact of the resulting artefact, the benefits that it delivers, the degree to which it is used, the quality built into it, the match with the project objectives, needs and requirements, the relationship with the different stakeholder groups, and the overall impact on the customer.

Level 3 is centred on the business efficiency which is assessed through the creation and delivery of internal value. The outcome of the project contributes to business success through the satisfaction of business objectives that have been realised. Success equates to maximisation of financial and business efficiency measures, such as sales, profits or ROI as well as delivered value measures.

Table 2.2 Levels of success

Levels of project success	Focus
Level 1: Project management success	– Efficiency and performance
Level 2: Project success	– Objectives, benefits, stakeholders
Level 3: Business success	– Value creation and delivery
Level 4: Future potential	– New markets, skills, opportunities

Level 4 is forward looking and opportunistic and enhances the business horizon by projecting future gains and opening new avenues, capabilities, skills, and markets. Strategic opportunities require a continuous and long-term approach that seeks to derive not just immediate benefit but also maximise opportunities for cornering the market, creating killer applications, and building the potential for self-enhancing positive feedback loops to secure future growth. Level 4 success is achieved through the realisation of new opportunities and harnessing of new potential. It may include new uses or ideas that were not originally considered as well as the development of new competence or capability.

The focus identified in Table 2.2 provides a clue as to the nature of measurements required at each level. Measurement at Level 1 focus on determining the progress and efficiency of the project management effort, for example, through the use of earned value management. Measures for Level 2 are concerned with benefits realisation and measuring the achievements of objectives, requirements, and expectations. Measures for Level 3 emphasise the business value using traditional economic measures such as sales, revenue, and delivered value. Measures for Level 4 require a more creative measurement of opportunities, capabilities, and market position. The combined levels offer a richer way of conceptualising and making sense of the complex phenomena surrounding success in and around projects.

2.5 Mapping Success

It is interesting to note the horizon of activity for each of the levels of success. At Level 1, project management success is concerned with the execution of the project itself based on performance against internally established constraints. Success at this level is determined upon the delivery of the project, often achieved through the incremental delivery of partial targets. It is primarily concerned with the task of project control. This is what most failure surveys assess and, therefore, where most failures are observed.

Level 2 success is more deeply entwined with the design activities resulting in the product or deliverables; indeed this is where utility and quality provide the key to the assessment of success. Both levels can be said to be output driven as they look at the complementary aspects of technical action and management within established and imposed constraints. Level 2 success can extend to cover the entire

Table 2.3 Focus vs. output/outcome

Focus vs. output/ outcome	Outputs	Outcomes
Internal focus	Level 1: Project management success Internal measures and constraints	Level 3: Business success Internal business value realised following project investment
External focus	Level 2: Project success Utility/usefulness of deliverables and other key outputs to stakeholders	Level 4: Future potential Business potential, opportunities and wider implications

operational life of the project. After all, delivering a bridge that stands for 6 months before collapsing is far from being a mark of utility, quality, or success.

The levels of success show a shift from focusing on internal efficiency and outputs, to outcomes and value as more strategic considerations come into play. The main distinction is that *outputs* occur as a result of a process as they relate to the specified deliverables and artefacts (that are delivered within time, cost, and performance constraints) viewed in terms of tangible products and services, such as a new web interface. Outcomes are the effects of change and how it translates into value for the entire business even beyond the reach of the original project. Outcomes happen as a result of the work and will often support the realisation of strategy through new capabilities or capacities, such as improved access to services or systems resulting from the new web interface. This relationship is depicted in Table 2.3.

Encouraging long-term thinking is important from a strategic perspective as it enables organisations to realise corporate strategies. It also fits with the need to deal with extended life cycles and consider deployment, extended use, and decommissioning of artefacts alongside benefiting from new opportunities and market possibilities. Moreover, it also chimes with the idea of viewing software development as the development of a continuous service (implying the fostering of long-term relationships and the dedication of attention to strategic concerns at the operational end) rather than the delivery of a single artefact. It is worth noting that while Levels 1 and 2 are primarily concerned with the delivery of a single project, the remaining levels look beyond a single delivery view by utilising a more strategic lens that considers operation and investment cycles.

A further important distinction is the separation between efficiency and effectiveness. Project managers and software developers have shown a tendency to focus on efficiency and its implications, as is reflected by the continuing obsession with failure studies. However meaningful solutions emerge from consideration of effectiveness. Efficiency is essentially viewed as an internally oriented productivity metric and method of evaluation, as it is concerned with following procedures and processes, adhering to constraints or achieving with minimum resources. Effectiveness on the other hand, relates to the overall utility; the ends to effectiveness' means.

Table 2.4 Efficiency/effectiveness vs. timing orientation

Efficiency/ effectiveness vs. timing orientation	Short term	Long term
Efficiency	Level 1: Project management success Efficiency of project: internal efficiency in delivery within constraints; minimising resources; procedural focus; project execution	Level 3: Business success Determining financial efficiency, business value, and return on investment
Effectiveness	Level 2: Project success Utility and quality of output; completeness, and conformance to requirements addressing true needs and concerns of stakeholders	Level 4: Future potential Achieving enterprise objectives; best quality horizon as focus for improvement; investment as greater benefit

In a nutshell, efficiency focuses on optimisation of the available resources by ‘doing things right’, while effectiveness revolves around the fulfilment of objectives and the contribution to achieving organisational goals by ‘doing the right things’. This is depicted in Table 2.4 which also shows the types of measures required.

Failure studies and surveys seem to focus on criteria concerned with the efficiency of projects, while ignoring the effectiveness aspects, and thus sidestepping the major issues associated with the outcomes of change. Indeed, even the bodies of knowledge in project management seem concerned with the track record of projects as a measure of quality. However, this is not the case as achievement in line with the triple constraints measures the ability to predict deadlines when uncertainty is at its highest and stick to them. This is not a measure of quality and is therefore addressed as Level 1 success. To attain project success, one needs to relate to the benefits, impacts, and quality aspects and perspectives related to the effectiveness of a project.

Quality, utility, and success are judged by different stakeholders in different ways, employing different criteria, over different timescales (Morris and Hough 1987; Wateridge 1995; Turner 2009)—see also Chap. 6. Recently, there has been a tendency to let the customer define quality. The Kodak organisation defines quality as ‘those products or services that are perceived to meet or exceed the needs and expectations of the customer at a cost that represents outstanding value’ (Kerzner 2009). The interesting point to note with this definition is how the customer viewpoint impacts on a project: a project must take great care that it accurately defines the customers’ needs and expectations, as the ultimate power about deciding on quality is given to the customers. So with this type of definition, conformance to requirements is not necessarily sufficient—the customer must be satisfied with the resulting product or service. Further, in order to maintain the satisfaction of customers and their loyalty and to ensure higher levels of success, products need to be revised and adjusted to reflect shifting needs and expectations (as well as market trends and competition). Consequently, maintaining quality becomes a continuous process of product (and process) improvement (Dalcher and Brodie 2007).

2.6 Illustrative Examples

To highlight the distinctive features of the levels of success and the differences between them it might be instructive to focus on thumbnail sketch examples of projects from a range of sectors and reflect on their relative success in comparison with the four levels.

Story 1: The operation was successful but the patient died. Level 1 success—Level 2 failure. This chapter began by describing a number of failure surveys focused on project management failure (i.e., the inability to meet time, cost, and performance criteria). Project management success is no guarantee of project success as many targets are assigned arbitrarily at an early phase. For example, the third attempt to deliver a working ambulance despatch system for London was delivered by the agreed deadline (Level 1 success), but stopped working a few days later resulting in potential loss of life (Dalcher 2010). This experience will chime with other software projects that are delivered on time and within budget, covering the agreed scope, which are ultimately never used by the users, or fail within a few weeks of the handover date.

Story 2: The Millennium Dome: Early partial failure becomes Level 4 success. The project to deliver a dome-shaped building to house a 1-year exhibition to celebrate the millennium had to be delivered in time for the new Millennium. The building itself and the infrastructure enabling Londoners to experience the exhibition were just about finished on time, but only following an unexpected injection of additional funds. On opening night, many of the exhibits were not functioning and dignitaries were left to queue outside for hours. Expected visitor numbers as specified in the business case exceeded one million per month, but in practice only about a third of the expected visitors turned up. The exhibition had to be kept open for a further year (in clear violation of the stated intention) to try and recoup some of the costs, while the entry fee was ultimately cut in half in order to attract additional visitors. Following the end of the exhibition, the site was mothballed at a cost of £190,000 a year adding to the accumulated losses. However, once the sale was finally concluded, the renamed O2 Centre became the biggest and most successful sports and entertainment arena complex in Europe. Level 4 success through innovative use of the structure thus managed to make up for the earlier disappointments (albeit in the hands of a new owner).

Story 3: The Sydney Opera House: Technical failure—architectural marvel. An even more heralded failure which clearly failed in terms of project management, project, and business. The Sydney Opera House came in at 14 times over budget, a clear project management failure. The building was unsuitable for its original purpose as the acoustics made it impossible to have concerts inside the hall. However the building has become an icon and is considered to be an architectural marvel. It is attracting tourists from all over the world and generating revenue, not least for the entire city. The revenue is not generated under the original intention but the new potential has been utilised to the full. Interestingly, the building was not fit for its purpose and hence was not of acceptable quality, yet it managed to generate a new purpose for which it was fit enough.

Story 4: Project Orion: Level 1 & 2 innovation success—long-term failure. A massive effort to develop Kodak's new Advantix photographic system was considered a big success on completion. The product was selected by Business Week as one of the best new products of 1996 (suggesting project success). It also won the Project Management Institute International Project of the Year award, confirming that it was also a project management success. The only problem was that Kodak failed to anticipate the accelerating switch to digital photography which made the product redundant. A successful output that won multiple awards was thus destined to become a failure as an outcome as it failed to deliver the promised value. In terms of utility the resulting product was an award winner but at the wrong end of the utilisation and relevance curve just as the technology slid out of fashion and was replaced by a new innovation.

The brief vignettes highlight the complexity of success and the interplay between the different levels involved. Success is never simple and the mini cases help in shedding further light on the rich, interconnected, and intricate (and sometimes temporary) nature of success.

2.7 The Impact of Time

Time is often viewed as a strategic resource. In some cases, it is viewed as being more important than money. This brief section focuses on the temporal nature of success and failure through the lens of time. In essence, we all recognise that things change. Perceptions, values, priorities, and objectives shift over time. Views about success and failure may also be similarly impacted. What is considered to be success, or a failure at a given point, may need to be adjusted to reflect new perceptions or changing views. Some of the individual cases described above hint at the change of perception over time. The following mini-case emphasises that point further.

Story 5: Raising the Vasa: Celebrating failure. The Vasa was the pride of the Swedish Royal Navy on August 1628 when it set sail on its maiden voyage. According to Fairley and Wilshire (2003), it was Sweden's most expensive project ever. The ship was much needed to bolster the Navy following a number of defeats in the Baltic Sea during the war with Poland. It was also meant to outperform a Danish ship being built at the same time. The completion of the ship was heralded as a great success and thousands turned up for cheer. Yet, definite success turned to failure when the ship capsized and sank after managing to sail less than a nautical mile out of Stockholm on a calm day. The crowd, which included foreign ambassadors invited to witness the great occasion, watched the ship disappear only 120 m from shore, with the loss of 30–50 sailors. In the space of minutes the successful release had turned into a catastrophic failure, accompanied by the loss of human life. The Vasa 333 years later, was raised and ultimately installed in a Museum close to where it sank. The Vasa is now one of Sweden's most popular tourist attractions as an embarrassing episode was radically transformed into a long-term success by 'celebrating' and analysing the failure.

The moral of the story is that the perception of success, and failure is wholly dependent on the point at which it is assessed. An undertaking can shift from being viewed as a success, as in the case of a standing bridge, or a floating ship, to being regarded as a total failure following structural, or even human failure. Nonetheless, over time new opportunities emerge and the perception can once again shift. Understanding and labelling of phenomena is therefore clearly aligned to the period in which it is perceived, observed, and categorised. The benefit of hindsight, the burden of maintenance, or the chimes of progress can revive, question, and re-enliven previous pronouncements.

It is also worth noting that outcomes persist beyond the life of a given project or programme. Increasingly, there is talk about the legacy of projects which delays the assessment of the impacts and results on initiatives. For example the legacy targets of the London 2012 Olympic games include local regeneration, increased participation in sports activities, enhanced enrolment in local clubs, and a reduction in rates of obesity among young people (BBC 2012). Such targets and outcomes can only be evaluated longitudinally. The reasons for the longer time horizon include the following:

- Effects following significant interventions are only temporary and partial.
- Evaluation requires a longer time horizon.
- Benefits were designed to be measured over longer time periods.
- Perceptions and priorities shift with time.
- Impact is measured in terms of changes to adaptive patterns over time.

2.8 Measuring Success

Determining the success of a project is not simple. It is often said that success is in the eye of the beholder, and can mean different things to different people. Consequently, analysing the dimensions of success and failure is a complicated task (Cooke-Davies 2002; Shenhar and Dvir 2007; Dalcher 2009). Measuring success requires an understanding of the different levels of success and what each one can offer:

Project management success implies tracking data related to predicted cost, time, and scope. Measuring performance against efficiency considerations is relatively straightforward. Determining progress through monitoring the achievement of milestones (e.g., using Earned Value Methods) enables project managers to track the achievement of predefined targets. It is a very useful focus when there is little residual uncertainty or when the project is clearly understood. However, it is debateable whether the measurement of an arbitrarily predefined target is completely meaningful, especially when project managers play little, if any, part in the initial estimation. Typical measures would focus on the efficiency of the process emphasising milestones, identified defects, and delivery and change management measures (including approved change requests), as well as earned value management measures showing project management progress, cost and schedule variances, cost performance index, and estimate to complete.

Project success relates to the effectiveness of a project and is normally considered in terms of realised benefits. In order to provide meaningful values, measures should relate to the requirements identified and be established and acknowledged as part of the needs assessment and requirements management processes. Stakeholder management is central to the identification and assessment of the concerns of different stakeholder groups and the issues impacting the development team. de Wit (1988) defined success as encompassing a high level of satisfaction concerning the project result amongst key stakeholders and users, while Lyytinen and Hirschheim (1987) framed the effort in terms of meeting stakeholders' expectations in terms of the balance between the objectives, constraints, and benefits. Project success can therefore be viewed as either realising the stream of benefits allocated to the project as they are cascaded down from the strategic objectives, or equalling and exceeding the expectations of clients, users, and stakeholder groups, thereby emphasising elements of stakeholder satisfaction and management.

Drevin and Dalcher (2011a, b, c) reported on the different concerns and issues expressed by different stakeholder groups. Using both narrative and antenarrative approaches to make sense of the stories of different communities, unveiled significantly differing interpretations of the purpose of a project, the approaches for dealing with alternative communities and the understanding of the perception of success and failure as they percolate and aggregate in different stakeholder communities. Typical project success measures would identify realised benefits from the project, achieved project requirements, satisfaction levels, recorded complaints, usage figures for the delivered artefacts, and met expectations.

Business success pertains to the funds raised from the initiative, or more generally to the organisational value derived in terms of finance, environmental and social concerns, and their balancing. The perspective often requires a longer timeframe that considers value creation and delivery over investment cycles and the contributions made towards the achievement of strategic objectives of the organisation. Business success can refer to the payback period, but often extends to consider the accumulated benefits accruing from an investment in a project or initiative. Business success measures typically address the delivered value, the rate of return, break-even calculations, payback calculations, sales achieved, revenue measures, environmental and social targets, and increasingly may focus on reputation, influence marketing, and sustainability ratings. Note that some of the calculations may be done from the project portfolio management, a multi-project or an enterprise-wide perspective. Occasionally, project managers are involved in devising some of these calculations during the initiation phase of projects, however the business success readings are likely to materialise well after the delivery phase of the project.

Future potential extends the time horizon of consideration into the longer-term utilisation of the outcomes and results of projects and actions. It allows the accumulation of longer-term benefits that result from adjustments and re-balancing. The intention is to seek to increase the accrued value from projects by exploring and exploiting opportunities beyond the formalised project baseline. Given the longterm focus it cannot be assumed that project assumptions, and originally intended outcomes remain relevant over time. The aim therefore is to maximise organisational value in accordance with the strategic direction. When projects are completed under conditions of uncertainty, they are often subject to positive feedback cycles, systems dynamics, and complex interactions that uncover new opportunities and strategic openings. Potential opportunities can often lead organisations to explore new directions, expand into a particular market, or occupy a certain leading position within a sector, and adjust their strategic intentions to match their new ambitions. Measures will focus on the identification and utilisation of emerging opportunities and adaptation to new market conditions that result from the experience, learning, and re-positioning related to the project and its outcomes.

Table 2.5 identifies some of the specific measures that can be utilised within each of the success levels. The measures were derived through a series of workshops with over 120 practicing project managers in four countries. The purpose of the sessions was to validate the model proposed in this chapter and to identify specific measures suited to each of the levels. Groups worked independently through the levels and the results were compared and developed further during a second series of facilitated sessions.

The accomplishment of any of the measures can be absolute or relative. The factors can be combined into a compound measure for a particular level, or a judgement statement regarding the achievement of each specific measure can be independently derived and assessed. In any project, one or more of the measures may be critical and multi-attribute aggregation methods can be used to combine, and trade-off specific measures.

Measurement of success requires understanding of the relevant levels. The identification of meaningful dimensions and the agreement regarding relevant factors could lead to a richer mapping of the relative success merits of projects alongside the multiple levels. More complex profiles can be devised by combining additional measures and considerations within each of the dimensions. Projects can thus be ranked and rated along four, or any other number of dimensions, providing an alternative to the simplistic measures which are currently being over used in failure studies. By shifting the focus to success and recognising the multi-factor

Table 2.5 Measures for determining success by level

Success level	Measures
Project management success	– Project cost – Project time – Full scope – Milestones – Functionality – Project performance data – Number of defects – Earned value management – Use of resources (in terms of resource constraints) – Agreed scope changes – Change requests
Project success	– Benefits – Satisfied objectives – Satisfied project requirements – Satisfied stakeholder needs – Stakeholder satisfaction – Client satisfaction – Complaints – Product/result usable – Product/result in use (usage figures) – Product/result useful – Fulfilled expectations
Business success	– Delivered value – Return on investment (ROI) – Break-even time (BET) – Break-even after release (BEAR) – Net present value (NPV) – Internal rate of return (IRR) – Economic value added (EVA) – Payback calculations – Shareholder value – Environmental targets – Social or societal targets – Sustainability considerations – Revenue measures – Sales – Improving operating margins – Reputation
Future potential	– New business opportunities – New benefits – Additional business – New markets – Derived products – Competitive advantage – New or expanded core competency – New system capability – New people-related capability – Recognition in new market or segment – New strategy – Improved processes – Image – Enhanced reputation

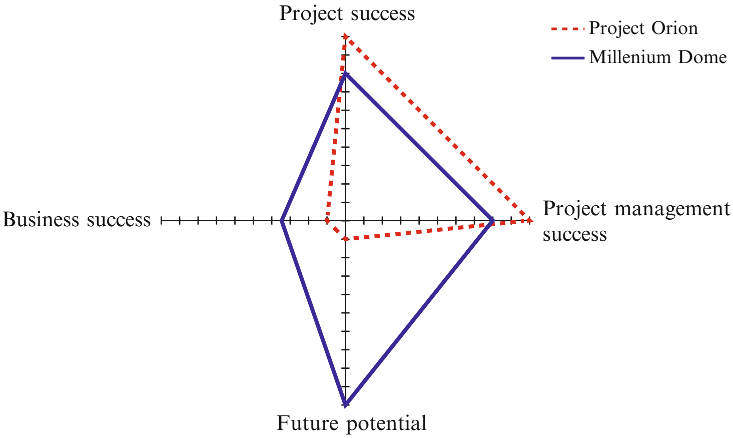


Fig. 2.5 Project success radar chart

nature of the levels, it thus becomes possible to chart a more meaningful picture of success using the levels as dimensions (see Fig. 2.5).

The radar chart will be able to rate the success at each level independently and provide more sophisticated snapshots of projects and initiatives capable of counterbalancing the simplistic measures used to gauge success in project work hitherto. Figure 2.5 provides a relative benchmarking between Project Orion and the Millennium Dome, described earlier, showing how each performed along the four project success dimensions.

2.9 Conclusions

Project failures have been used to highlight the need to improve IT software project practice. Many of the studies and surveys focus on project management success (or failure), which can be described as a subset of internal efficiency measures and imposed constraints ignoring the impact on the project and the business. In order to improve project performance, project managers need to look beyond such measures and focus on project success—an area concerned with the effectiveness and quality of the project output. Project managers are also increasingly asked to consider the value derived from the project, the sustainability implications as well as issues related to environmental, social, and societal impacts.

Success is a complex and multi-layered concept that needs to be understood at different levels and time frames. Indeed, the impact of success often extends beyond a single project. This chapter offers a wider perspective, which takes in a range of project success levels thus enabling practitioners to move beyond the simplistic measures that continue to be offered. The success view determines actions and colours new developments. Increased attention to enterprise objectives and utility, rather than simply endeavouring to optimise correctness according to preimposed constraints, can open a new dialogue about the needs of a profession seeking to fundamentally and essentially improve its track record and enable project management practice to rise beyond the continuous obsession with failure.

Further work is needed to encourage the research and practice communities to consider project management success at a number of levels. Practitioners will need to make links between strategy, business, and project management delivery functions, while researchers are likely to try and make sense of requirements and expectations that emerge from a multi-level model that invites new types of surveys to make sense of the success and failure in software projects. Ultimately, in order to overcome failure we must learn to appreciate success and grow up enough to look beyond the simplest manifestations of an imperfect practice.

> *Success is not final, failure is not fatal: it is the courage to continue that counts.*
> Winston Churchill

References

Anon (1979) Contracting for computer software development—serious problems require management attention to avoid wasting additional millions, general accounting office report to the congress by the comptroller general of the United States. FGMSD 80–84

Baccarini D (1999) The logical framework method for defining project success'. Proj Manag J 30 (4):25–32

Barnes M (2013) Private communication, September 2013 BBC (2003) BBC Radio 4 news, 15.5.2003. http://www.silicon.com/news/500022/1/4169.html

BBC (2012) London 2012 legacy plan published, 18 September 2012. http://bbc.co.uk. Accessed 12 Nov 2013

Bloch M, Blumberg S, Laartz J (2013) Delivering large-scale IT projects on time, on budget and on value. McKinsey Finance 45:28–35

Brooks F (1975) The mythical man-month: essays on software engineering. Addison-Wesley, Reading, MA

Buxton JN, Naur P, Randell B (1969) Software engineering techniques. In: Proceedings, NATO conference, scientific affair division, Brussels

Charette RN (2005) Why software fails. IEEE Spectr 42(9):42–49

Cooke-Davies T (2002) The "real" success factors on project. Int J Proj Manag 20(3):185–190

Dalcher D (1994) Falling down is part of growing up; the study of failure and the software engineering community. In: Proceedings of 7th SEI education in software engineering conference. Springer, New York, pp 489–496

Dalcher D (2009) Making sense of IS failures. Encyc Inf Sci Technol 5:2476–2483

Dalcher D (2010) The LAS story: learning from project failure. In: Turner RJ, Huemann M, Anbari FT, Bredillet CN (eds) Perspectives on projects. Routledge, New York

Dalcher D (2012) The nature of project management: a reflection on the anatomy of major projects by Morris and Hough. Int J Manag Proj Bus 5(4):643–660

Dalcher D, Brodie L (2007) Successful IT projects. Thomson, London

Dalcher D, Genus A (2003) Avoiding IS/IT implementation failure. Tech Anal Str Manag 15 (4):403–407

de Wit A (1988) Measurement of project management success. Int J Proj Manag 6(3):164–170

Drevin L, Dalcher D (2011a) Antenarrative and narrative: the experience of actors involved in the development and use of information systems. In: Boje DM (ed) Storytelling and the future of organisations: an antenarrative handbook. Taylor and Francis, New York, pp 148–162

Drevin L, Dalcher D (2011b) Narrative methods: success and failure stories as told by information systems users. In: Standing conference for management and organization enquiry, SC’MOI conference, Philadelphia, PA

Drevin L, Dalcher D (2011c) Using antenarrative approaches to investigate the perceptions of Information Systems’ actors regarding failure and success. In: Pokorny J, Repa V, Richta K, Wojtkowski W, Linger H, Barry C, Lang M (eds) Information systems development business systems and services: modeling and development. Springer, New York, pp 207–218

Egan J (1998) Rethinking construction, the report of the construction task force. Department of Environment, Transport and the Region, London

Egan J (2008) Sir John Egan’s speech to the house of commons, 21st April, 2008

Eveleens JL, Verhoef C (2010) The rise and fall of the chaos report figures. IEEE Softw 27 (1):30–36

Fairley RE, Wilshire MJ (2003) Why the Vasa Sank: 10 problems and some antidotes for software projects. IEEE Softw 20(2):18–25

Flowers S (1996) Software failure, management failure. Wiley, Chichester

Flyvbjerg B, Budzier A (2011) Why your IT project may be riskier than you think. Harv Bus Rev 89(9):83–85

Geneca (2011) Doomed from the start? Why a majority of business and IT teams anticipate their software development project will fail. Geneca, Oakbrook Terrace, IL

Glass R (2005) IT Failure Rates—70% or 10-15%. IEEE Softw 22(3):110–112

Glass R (2006) The Standish report: does it really describe a software crisis? CACM 49(8):15–16

IBM (2008) Making change work. IBM Global Services, Somers, NY

Jones C (1994) Assessment and control of software risks. Prentice-Hall, Englewood Cliffs, NJ

Jones C (2000) Software assessments, benchmarks and best practices. Addison-Wesley, Upper Saddle River, NJ

Jones C (2007) Estimating software costs: bringing realism to estimating. McGraw Hill, New York

Jones C (2008) Applied software measurement: global analysis of productivity and quality. McGraw Hill, New York

Jones C (2010) Software engineering best practices: lessons from successful projects in the top companies. McGraw Hill, New York

Jørgensen M, Moløkken K (2006) How large are software cost overruns? A review of the 1994 Chaos report. Inform Softw Technol 48(8):297–301

Kerzner H (2009) Project management: a systems approach to planning, scheduling and controlling, 10th edn. Wiley, Hoboken, NJ

Kloppenborg T, Mantel SJ (1990) Tradeoffs on projects: they may not be what you think. Proj Manag J 21(1):13–20

KPMG (2010) KPMG New Zeland project management survey 2010. Http://kpmg.co.nz

Linberg K (1999) Software developer perceptions about software project failure: a case study. J Syst Softw 49:177–192

Luqi, Goguen JA (1997) Formal methods: promises and problems. IEEE Softw 14(1):73–85

Lyytinen K, Hirschheim R (1987) Information systems failures—a survey and classification of the empirical literature. Oxf Surv Inf Technol 4:57–309

McManus J, Wood-Harper T (2008) A study in project failure. http://www.bcs.org/server.php?show=ConWebDoc.19584

Morris PWG, Hough G (1987) The anatomy of major projects: a study of the reality of project management. Wiley, Chichester

Munns AK, Bjerimi BF (1996) The role of project management in achieving project success. Int J Proj Manag 14(2):81–87

Naur P, Randell B (1968) Software engineering. In: Proceedings. NATO Scientific Affairs Division, Brussels

Pinto JK, Slevin DP (1988) Critical success factors in effective project implementation. In: Cleland DI, King WR (eds) Project management handbook, 2nd edn. Van Nostrand Reinhold, New York

Sauer C, Cuthbertson C (2003) The state of IT project management in the UK 2002–2003. Templeton College, Oxford

Sauer C, Gemino A, Reich BH (2007) The impact of size and volatility on IT project performance. CACM 50(11):79–84

Shenhar AJ, Dvir D (2007) Reinventing project management: the diamond approach to successful growth and innovation. Harvard Business School Press, Boston, MA

Standish Group (2000) Chaos 2000. Standish, Dennis, MA

Standish Group (2004) Chaos 2004. Standish, Dennis, MA

Standish Group (2011) Chaos 2011. Standish, Dennis, MA

Taylor A (2000) IT projects sink or swim. The Comp Bulletin, January, Brit Comp Society

Turner JR (2009) The handbook of project based management, 3rd edn. McGraw-Hill, New York

Wateridge JH (1995) IT projects: a basis for success. Int J Proj Manag 13(3):169–172

Biography Darren Dalcher is Professor of Project Management at the University of Hertfordshire and Founding Director of the National Centre for Project Management. He edits two international project management book series as well as the *Journal of Software: Evolution and Process*. He has published over 150 refereed papers and book chapters. He is an Honorary Fellow of the Association of Project Management and a Chartered Fellow of the British Computer Society.

Chapter 3
Cost Prediction and Software Project Management

Martin Shepperd

Abstract This chapter reviews the background and extent of the software project cost prediction problem. Given the importance of the topic, there has been a great deal of research activity over the past 40 years, most of which has focused on developing formal cost prediction systems. The problem is that presently there is limited evidence to suggest formal methods outperform experts, therefore detailed consideration is given to the available empirical evidence concerning expert performance. This shows that software professionals tend to be biased (optimistic) and over-confident, and there are a number of deep cognitive biases which help us understand why this is so. Finally, the chapter describes how this might best be tackled through a range of simple, practical and evidence-based methods.

3.1 Introduction

Cost estimation[1] has been viewed as a challenging and important part of software project management for almost 60 years. Interestingly, Benington (1956) writes of his experiences developing, what was back in the mid-1950s, a large air defense system comprising half a million lines of code (LOC). In it he tabulates what he termed 'reasonable production costs' and although the headings such as computer

[1] There is something of a proliferation of terminology. Whilst the majority of writers refer to cost modelling or prediction, strictly speaking the usual focus is upon labour or effort which forms the dominant part of costs and is usually the hardest to predict. Such costs may or may not be reflected in the price charged to the client or user. This chapter will use the term in this particular sense. Likewise, estimation and prediction are used interchangeably since we're only concerned with future events.

M. Shepperd (✉)
Department of Computer Science, Brunel University, Middlesex UB8 3PH, UK
e-mail: martin.shepperd@brunel.ac.uk

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_3, © Springer-Verlag Berlin Heidelberg 2014

and paper costs might no longer be seen as relevant, others such as specification, coding and testing remain pertinent. The outcome was 'the schedule slipped by a year', something that remains distressingly familiar!

So how bad is the problem? Apart from the anecdotal, evidence is surprisingly elusive probably due to the commercially sensitive nature of poor project cost estimation. Jørgensen and Moløkken-Østvold (2006) reviewed multiple sources of evidence and concluded that a typical cost estimation error was 'in the range of about 30 %'. Another indicator that not all is well comes from the 2005 and 2007 surveys conducted by El Eman and Koru (2008) who from a total of 388 responses found 'the most critical performance problem in delivered software projects is therefore estimating the schedule and managing to that estimate'. An independent study by the European Services Strategy Unit of 105 large public ICT projects (Whitfield 2007) found more than half to show cost overruns with the average cost being 30.5 %, a figure very much in line with Jørgensen and Moløkken-Østvold (2006).

The question therefore arises, as to why are software project costs difficult to estimate? There are many reasons. First and foremost is complexity. Many projects are extremely large undertakings with multiple stakeholders in a setting characterised by uncertainty, inconsistency and change. Second, software development is best viewed as a design type activity and it is emphatically not concerned with production. This means the sub-tasks and activities are not routine so simple linear extrapolation is seldom a safe guide. Third, estimates are required at a very early stage when little is known and requirements are still to be discovered, arbitrated, let alone documented. Finally, there are many subtle, and not so subtle, social and political pressures upon those responsible for cost modelling. In his analysis of a wide range of projects, Flyvbjerg refers to this tendency to under-estimate costs and over-estimate benefits in order to secure funding for a proposed project as 'strategic misrepresentation' (Flyvbjerg 2008).

Clearly, these problems with predicting software project costs have significant ramifications. First, we see a tendency for errors in one direction, i.e., bias or a propensity for over-optimism. Second, poor cost prediction will severely hamper meaningful cost-benefit analysis and the consequent unnecessary cancellation of projects that should not have been commissioned in the first place. Conversely, under-estimation might lead to missed opportunities or sub-optimal procurement decisions.

3.2 A Review of State-of-the-Art Techniques

The first thing to consider is what is an estimate? Although it can easily be forgotten, it must be stressed that an estimate is a probabilistic statement (DeMarco 1982; Kitchenham and Linkman 1997), and consequently, to simply report an estimate as a point value masks important information. As an example, if a project manager makes a prediction that the Integration Testing will take 150 person-hours, we do not know with what confidence he or she makes this statement; it could be with near certainty or it could be a wild guess with almost no certainty. Thus there are two components. Jørgensen and Sjøberg (2003) recommend a simple approach based on an interval and a confidence level. Based on the Integration Testing example, the project manager (if highly confident) might state 140–160 person-hours at 90 % confidence, or (if lacking confidence) 50–250 person-hours at 50 % confidence. Note the trade-off between the interval size so it is possible to increase confidence by enlarging the interval or to decrease the confidence value and reduce the interval accordingly.

An alternative approach sometimes used in industry is derived from the critical path analysis technique Program Evaluation and Review Technique (PERT) (Willis 1985) and is known as 3-point estimation. It is based on the idea that an estimate is actually a probability distribution, and a simple characterisation is as a triangle based on the best case, worst case and most likely case or mode.

Figure 3.1 shows an example of a 3-point estimate depicted as a triangular probability distribution. The shaded area shows the region within which the true or actual effort value will fall (assuming of course that the distribution is correctly estimated). The estimation interval is the range between the worst case (i.e., the highest possible value) for effort and the best case (i.e., the lowest possible value) for effort. In addition, the distribution shows likelihood or the probability p on the y-axis. This reveals that the highest or modal point on the distribution is the most likely, i.e., it has the greatest chance of actually occurring. The distribution also reveals another interesting property, that it is skewed or biased since the region above or to the right of the most likely value is considerably greater than the region below the mode. The implication is that even if the distribution were accurately estimated, to use the most-likely value as the actual estimate will lead to a tendency to under-estimate over time. This is a phenomenon that we observe (as noted in Sect. 3.1).

Although thinking of an estimate as a distribution enables a far richer analysis, empirically we are hindered by the fact that we are obliged to construct the distribution from a single observation. The situation can be further complicated by the fact that projects are seldom static and so one has to be clear whether the estimates refer to a project as intended at its inception as compared with the actual project as delivered which could conceivably have functionality added or removed. These problems are further explored by Grimstad et al. (2006).

There is surprisingly little systematic analysis of what software practitioners actually do. Studies such as those by Heemstra (1992) and Hughes (1996) have reported that expert judgment is the dominant method amongst software practitioners and there is little to suggest matters have changed radically since the 1990s.

One source for identifying what is perceived as good practice is the Software engineering Body of Knowledge (SWEBOK; Abran and Bourque 2004), which was the culmination of the work of a team of software development experts. Interestingly, the section on effort, schedule and cost estimation is relatively brief; however, a number of principles emerge:

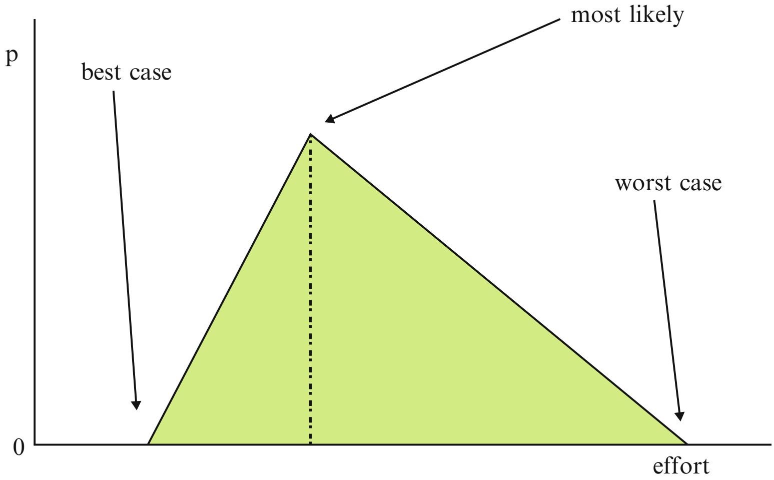


Fig. 3.1 Three point estimates as probability distribution

1. Estimates can be derived *top-down* or by means of some breakdown of tasks.
2. For each such task the expected effort [cost] *range* can be derived from a cost model which needs *calibration* to the local environment using *historical* data if available. Otherwise, an alternative is needed such as expert judgment.
3. The individual estimates should be summed across the entire project.
4. Estimates need to be *revised* iteratively until agreement is reached amongst all stakeholders, which the SWEBOK identifies as principally software engineers and management.

In this list of steps I have italicised some key concepts, which will be explored in more detail.

The idea behind a top-down or a decomposition approach to cost estimation is that of divide and conquer. In other words, it is easier to estimate the cost of a small task than a large one. Moreover, it is easier to match a smaller task to some repertoire of previously completed tasks, than it is for a large task where the combinatorial explosion militates against this possibility. Often, the idea is formalised into work breakdown charts. The chief difficulty is the fact that some activities do not easily fit into neat hierarchical breakdowns.

The next point of note is the SWEBOK recommendation to consider representing an estimate as a range. As previously discussed, in order to view an estimate as a probabilistic statement a point value is inadequate. However, to attach additional meaning minimally, we need a confidence level in the range. Provision of 3-point estimate provides an even richer picture.

SWEBOK also recommend the use of formal models, and although no examples are specified, widely used models include COCOMO 81, which is based on a non-linear relationship between estimated LOC and effort, implying diseconomies of scale. The fundamental relationship is modified by the type of project and, initially, 14 cost drivers. This was subsequently modified and extended as COCOMO II, although, unfortunately unlike COCOMO 81, the database from which this model is derived is not in the public domain.

Although COCOMO is widely used and there are many free implementations, it has come in for criticism. Firstly, accurate estimates of LOC may not be available at an early stage of a software project. Secondly, there is mixed empirical evidence as to whether software projects exhibit diseconomies (as many commentators assert), economies or simple linearity with respect to scale (Kitchenham 2002). Third, there is limited evidence that COCOMO performs well using the off the shelf settings on data other than that with which it was developed, for example, Kocaguneli et al. (2012b) reported that the model was ranked 92 out of 102 different combinations of models and pre-processors that were evaluated in a major empirical study. Likewise Kemerer (1987) reported mean absolute relative errors in excess of 600 % for a different data set of 15 software projects. Interestingly, he found that COCOMO performed best (least badly?) in its simplest form, and additional sophistication of the model harmed its accuracy. This has led many researchers, in line with the SWEBOK, to recommend tailoring and calibration to a local environment. Gulezian (1991) describes how multiple regression analysis can be used to calibrate the weights for the various cost drivers. The systematic review by Jørgensen (2004) identified individual primary studies and the *only* ones that showed formal prediction systems to outperform experts involved the use of calibration. More recently, Yang et al. (2013) described a calibration procedure to handle local bias, thereby improving the usability of cross-company data sets and demonstrated this with respect to COCOMO II. The value of calibration was again highlighted by the analysis of Menzies et al. (2013). Nevertheless, despite some of the reservations COCOMO or a similar approach is often used as some form of sanity check.

Another important part of the SWEBOK recommendations is the need to revisit any prediction. This has often been neglected by researchers who tend to see a software project as a static snapshot, which of course does not reflect the realities of (1) a growing understanding of the requirements and challenges as the software project plays out, converging upon certainty on the day of delivery and (2) the changing environment in which the project is embedded. MacDonell and Shepperd (2003a) in a rare study of re-estimation in a commercial setting found no support for the idea that there are ‘standard proportions’ of effort for particular development stages, e.g., specification and design. However, in most cases simple linear regression combined the managers’ estimates led to improvements in predictive accuracy. These results indicate that, in this organisation, prior-phase effort data is useful and revising estimates worthwhile.

3.3 A Review of Cost Estimation Research

Because of the need for effective software cost estimation, this has been the subject of a good deal of research. From the outset, the aim has been to replace the subjectivity of project managers and other professionals, generally referred to as expert judgment with more objective and formal approaches. This was, or still is, seen as a good thing because this provides opportunities for scrutiny, it is more repeatable and can militate against the loss of knowledge and insight if experts leave an organisation.

Early approaches tended to be based on some function between size either measured as estimated LOC or Function Points (Albrecht and Gaffney 1983) and a variant known as Mk II Function Points (Symons 1988). Generically, these take the form:

$$\mathrm{E} = f(\mathrm{S}^{\mathrm{a}})$$

where E is effort or cost, S is size (typically measured by LOC or Function Points) and *a* an exponent representing economies or diseconomies of scale. Typically, this overall relationship is then modified by a set of productivity or cost factors. COCOMO 81 (as described in Sect. 3.2) is a good example of this approach. An interesting recent study by Kocaguneli et al. (2012a) has suggested that in many cases, the use of a size measure may be less important than previously supposed. It may be that other features act as a proxy for size, e.g., the different application types may tend to be of different sizes. Nevertheless, it is an thought-provoking point that size may be less essential than has been previously supposed.

Early models were postulated based on the beliefs of the inventor, however, the 1990s heralded a more data-driven approach to modelling. Often, multiple regression methods sometimes using a stepwise approach[2] were deployed in order to isolate the important factors, specific to some software development environment as captured by a data set of historical project data. Kitchenham and Kansala (1993) used multiple regression to re-estimate weightings for the standard values for Function Points with considerable benefit. They also reminded researchers of the dangers of constructing models when many of the components are strongly correlated, i.e., multicollinearity is present which if uncorrected leads to highly unstable models.

Given the emphasis of learning from historical data, different machine learning techniques became popular from the 1990s onwards. In all cases the underlying principle is to reason inductively from the particular to the general. For cost prediction the idea is to learn from past, completed software projects in order to predict for new, unseen projects. One technique is lazy learning[3] based on analogical or case-based reasoning (Shepperd and Schofield 1997; Keung et al. 2008) which is often referred to as Estimation by Analogy (EBA). The simplicity of the idea—that history repeats itself, but not exactly—has attracted a good deal of attention not least because to be acceptable to practitioners, prediction systems

[2] The regression model is constructed one independent variable at a time or iteratively until no new variable *significantly* contributes to the model fit.

[3] A lazy learner only makes an inductive generalization when actually presented with the new problem to solve. This can be advantageous when trying to learn in the face of noisy training cases and much uncertainty.

benefit from good explanatory value since decisions arising from the prediction will be of high value (Mair et al. 2000). Despite these strengths, EBA was not found by a Systematic Review (Mair and Shepperd 2005) of all available empirical studies to outperform simpler regression models with 9 studies supporting, 4 equivocal and 7 against.

Because of the relative ease of fitting regression models these are now often used as a benchmark with which to compare more elaborate methods, e.g., Mair et al. (2000) compared various machine learning methods (artificial neural nets (ANNs), case-based reasoners (CBR) and rule induction) with stepwise regression. Interestingly, the basic regression approach outperformed the rule induction algorithms although not CBR or ANNs.

The last decade could be characterised by research that has explored more advanced prediction systems. Examples include through the use of ensembles of learners coupled with some decision making logic (Minku and Yao 2013) and new approaches like Grey Relational Algebra (Song and Shepperd 2011). This has been supported by more research into such things as data pre-processing as many prediction methods are vulnerable to excessive noise, extreme outliers and missing observations. Consequently, appropriate pre-processing can have a substantial impact upon predictive performance, Strike et al. (2001), Song and Shepperd (2007), Liu and Mintram (2005).

Another area of concern and of some progress is developing frameworks as to how we meaningfully compare the proliferating number of cost estimation approaches. Until the empirical studies of Myrtveit and Stensrud (1999) which set out to independently compare regression modelling, EBA and the unaided human expert, it was not customary to perform any statistical testing. Subsequently, inferential test such as *t*-tests and Mann–Whitney U became the norm, however, methodological problems such as correcting[4] the α threshold for null hypothesis significance testing in the face of large numbers of tests and using inappropriate measures of predictive accuracy remained. Mittas and Angelis have proposed a method that is not too conservative but reduces the number of tests required by means of clustering the results into groups (Mittas and Angelis 2013). More generally, various authors have proposed remedies and strong arguments as to why to proper procedures are required in order to derive sound conclusions. For example, Shepperd and MacDonell (2012) show that inappropriate evaluation hid the fact that various published prediction techniques such as regression to the mean coupled with EBA actually performed worse than guessing!

After the event, when evaluating the quality of a prediction there are three dimensions that need to be assessed (1) error (2) bias and (3) variance or scatter.

[4] Essentially, the point is that when conducting a significance test for a hypothesis, there are two dangers: One can wrongly reject the null hypothesis or wrongly fail to reject the null hypothesis. It is customary to set the chances of wrongly rejecting the null hypothesis (denoted by α) at 0.05. However, if many tests are performed, the probability of at least once committing such an error grows with the number of tests. For this reason, the α threshold needs to be reduced to take this danger into account.

Even accuracy is often misunderstood in the software engineering community and inappropriately assessed by accuracy statistics such as Mean Magnitude of Relative Error (MMRE). Elsewhere researchers show how this is flawed both theoretically as it is merely an asymmetric measure of spread (Kitchenham et al. 2001) and empirically through Monte Carlo simulation (Foss et al. 2003). Without a clear conceptual understanding of accuracy it is difficult for the community to review or improve their prediction practice since there is no systematic basis for evaluating different approaches to cost estimation. Indeed, MMRE has the rather perverse characteristic of favouring optimistic predictions over pessimistic ones. Given the widespread use of MMRE this may be another contributor to the biases we observe in industry practice described in Sect. 3.1. Therefore, unless there is good reason to the contrary, it is recommended (Shepperd and MacDonell 2012) that researchers seek to minimise the absolute sum of the residuals, consider performance relative to guessing and be aware of the effect size. The effect size is a means of capturing the practical or real world effect of the particular intervention, for example by moving from cost estimation technique A to B what actual benefit does this yield? This is a very different question from how likely is the effect to have arisen by chance since large numbers of observations will render even small effects highly significant (Armstrong 2007; Ellis 2010).

The final development, and one that warrants a section in its own right, is the realisation that formal prediction or cost models have not succeeded in replacing humans and therefore there is a need to research into how practitioners make predictions. This section has of necessity been brief. For a more detailed overview see the mapping studies in Jørgensen (2004) and Jørgensen and Shepperd (2007).

3.4 The Interaction Between People and Formal Techniques

As the previous section has shown there has been no shortage of ideas or research into constructing formal prediction systems for software project costs. Unfortunately, as systematic reviews (Mair and Shepperd 2005; Jørgensen and Shepperd 2007; and simulation work Shepperd and Kadoda 2001) demonstrate, no single technique dominates. In particular, formal model performance seems closely linked with the specific characteristics of the historical data that are used to train or calibrate the prediction system (Shepperd and Kadoda 2001). This has led some researchers such as Menzies et al. to suggest that we should focus on finding prediction systems that are ‘good enough’ rather than the ‘best’ (Menzies et al. 2010). Nevertheless, Jørgensen (2004) reported that formal models do not consistently outperform their human counterparts and frequently do less well. Specifically, in his systematic review of 15 primary studies he reports that 5 favoured formal models, 5 were equivocal and 5 favoured expert judgement over the formal model. Looking in more detail, Jørgensen suggests that those studies using local calibration or where the estimators lacked expertise yielded the best results for formal models. Similarly, in a software maintenance setting, the systematic review of Riaz et al. (2009) found that 'there is little evidence on the effectiveness of software maintainability prediction techniques and models'. Moreover, formal models do not appear to be very widely used in practice and expert judgement remains the dominant estimation technique (Jørgensen 2004). Consequently, Jørgensen and his co-workers have been exploring over the past decade why this might be so.

The first thing to appreciate is the nature and use of cost estimates. Software projects are generally high value and relatively infrequent events since typical durations are many months through to years. Therefore the estimate matters and in a way that predicting if a supermarket customer chooses a cabbage will they also purchase carrots, does not. The career prospects of an individual may be impacted by an estimate and the associated decision-making, e.g., to initiate/cancel a software project. In extremis the financial health or viability of the software development may be impacted. Such awareness may skew the estimation process of individuals. More than 20 years ago Lederer and Mendelow (1999) in their study of cost estimation within information systems projects observed how organisational politics can be inimical to good estimation. Flyvbjerg et al. (2003), Flyvbjerg (2008) in a study of a number of major projects—whilst not specifically related to software—found considerable evidence to support the notion of strategic misrepresentation. This typically manifests itself as a tendency to under-estimate costs and over-estimate benefits because of the desirability of the end goal. In terms of software it may be that professionals might see the potential opportunities of a new project, e.g., improved work prospects, personal development or intellectual challenge. The interesting thing is that formal models may not offer any protection against such phenomena since these models require inputs, many of which must be estimated, for instance COCOMO (as previously indicated) requires the user to estimate delivered LOC which will not normally be known at the point of prediction. Likewise many machine learning techniques are heavily parameterised with little deep theory to guide the user, thus rendering such methods rather experimental in their approach. This can encourage a 'suck it and see' philosophy. Jørgensen and Gruschke (2005) termed this 'expert judgment in disguise'.

The problem of obtaining useful predictions is compounded by the strong tendency for professionals to display both over-optimism, e.g., Buehler et al. (1994) and over-confidence, e.g., Jørgensen (2010). Because these phenomena are so widespread the causes of bias have been extensively investigated by cognitive psychologists in various domains over the past three decades since the seminal work of Kahneman and Tversky (1979). This has led to the identification of a number of cognitive biases that appear to be both deeply ingrained and widespread. Four such biases are now considered.

One problem is the so-called 'planning fallacy' which is the tendency to underestimate project completion times as a consequence of spending time on detailed planning aspects. Buehler et al. (1994) examined the underlying cognitive processes and found that a narrow focus on future plans for the target task led to neglect of other useful sources of information. In other words, an illusion of control leads to significant over-optimism. Therefore we might expect detailed top-down planning methods such as work breakdown to be vulnerable to this particular bias.

Another source of bias is a preference for case-specific (and recent) evidence over distributional evidence (Tversky and Kahneman 1974; Griffin and Buehler 1999). For example, data suggesting that 8 out of 10 projects are delivered late (i.e., costs and schedule have been under-estimated) might be neglected in preference to evidence suggesting this specific project will be different because staff will be motivated to work harder or because there will be reuse of some software components. This helps us understand why professionals struggle to learn lessons from the past because deep down we believe it will be different next time. The problem is the distributional or frequency related evidence says otherwise and this is usually correct!

A closely related phenomenon is the peak-end rule where the most recent experience dominates even when it is highly atypical. This has been demonstrated in many different arenas including the experiment described in Kahneman et al. (1993) where participants were subjected to modest pain (a hand in icy water) and preferred the worse (in terms of temperature and duration) experience when for the final period the water temperature was raised. In terms of software projects, professionals may recall the final experiences of getting software to work, as opposed to the lengthy previous experiences of failures and debugging. Again this bias can lead to distributional evidence being ignored or neglected and the consequent impact upon estimates.

A third, relevant cognitive theory is the dual-process theory of cognition which leads to a tendency to trust analytic justifications (explanations) over intuitive ones yet to prefer intuitive judgments over analytic ones. One implication is that this is another reason why formal prediction systems can turn into ‘expert judgment in disguise’ (Jørgensen and Gruschke 2005) as the estimator is seeking ‘objective’ evidence to support his or her intuitive judgement.

A fourth bias is known as anchoring where data in the request for an estimate can be highly influential even when the estimator is told to ignore it. An example is the experiment by Jørgensen and Grimstad (2012) where professional participants were randomly allocated to two groups, one of which was primed with a high anchor and another with a very low anchor. They were then asked to estimate the same task, namely their own productivity in LOC per work-hour over their last project. Remarkably, the difference in median response between the two groups was almost sevenfold (15 LOC per hour versus 100 LOC per hour). This stable finding—repeated by a number of independent studies—indicates just how vulnerable humans are to these biases and is clearly a major contributor to the some of the cost estimation problems reported at the beginning of this chapter.

These biases are common to many problem domains, and seem independent of individual differences, e.g., the traits of optimism and procrastination (Buehler and Griffin 2003). The limited work investigating de-biasing strategies, e.g., utilising previous experience, such as past project databases, Personal Software Process

(Humphrey 2000) and lessons learned sessions, have not been all that successful, particularly in the field of software engineering prediction. Interestingly, Jørgensen and Grushka found that software professionals were better able to learn lessons for the estimates of others than for their own estimates (Jørgensen and Gruschke 2009).

There are both theoretical and empirical reasons why software practitioners make consistently sub-optimal predictions within software engineering. However, the vast bulk of the psychological research has been conducted using student participants working on problems that are not industry-related (Mair et al. 2009) and therefore Jørgensen's work using software developers has been quite unusual. In addition, the literature has predominantly focused upon understanding factors that contribute to bias. We need to also explore factors that promote de-biasing in realistic settings. In parallel, much research has been undertaken into metacognition (i.e., thinking about thinking), particularly in the domain of learning. There is a considerable body of evidence showing that increased metacognitive awareness leads to increased learning and enhanced performance, e.g., Coutinho found a relationship between metacognitive awareness and educational performance (Coutinho 2007). Other researchers have shown that metacognitive skills can be taught (Borkowski et al. 1987; Dawson 2008) and these can potentially militate against some of the cognitive biases described above.

Metacognition can be divided into metacognitive knowledge and metacognitive skills. The former relates to declarative knowledge of the interactions among self, task, and strategy characteristics (Flavell 1979) that can be inaccurate and resistant to change. Clearly, this will be an inhibitor to improving prediction performance. Metacognitive skills on the other hand refer to procedural knowledge for self-regulating problem solving and learning activities and include feedback (reflection) on metacognitive knowledge. This division between metacognitive knowledge and skills is related to that of single and double loop learning popularised by Argyris and Schön (1996).

'Single-loop learning' occurs when goals, values, plans and rules are taken for granted and put into operation rather than questioned. It reduces risk and affords greater control, but severely limits growth and learning. By contrast, 'double-loop learning' involves questioning the fundamental systems that underlie goals and strategies. It results in the questioning of governing variables and may lead to fundamental changes. This double-loop learning is necessary if practitioners and organisations are to make informed decisions in changing and uncertain contexts.

Reflection is a metacognitive skill important for personal and professional development, see for example, Schön (1983), Moon (1999), and it plays a key role in both single and double loop learning. However critical reflection, as demonstrated in double loop learning, is essential for growth and change. Critical reflection demands focusing on the cognitive aspects and challenging the strategies that led to particular actions, and the outcomes and lessons learned from those actions for future application.

Unfortunately, previous studies of software project cost prediction suggest that feedback on performance and the typical methods for reflecting on experience, e.g., unaided lessons learned sessions, do not necessarily lead to improvement in accuracy or assessment of the uncertainty (Jørgensen and Gruschke 2009). The lack of training in both reflecting on one's own thinking and the fundamental causes of suboptimal outcomes (double-loop learning) can be a major obstacle. As an illustration, in a previous study where software professionals described reasons for their estimation errors (Jørgensen and Gruschke 2009; Moløkken and Jørgensen 2004), most were shallow and corresponded to single-loop learning. In particular the participants (all software professionals) exclusively focused on reasons for their estimation inaccuracy and at the expense of their confidence. Indeed, participants only identified means to improve their accuracy (e.g., add more time for unknown events). The alternative, which would have been to change their level of confidence in the effort estimates, was not considered in terms of documented reflections. This lack of double-loop learning would seem to be a key contributor to the robust findings on over-optimism and over-confidence among software developers (Note, in contrast Chap. 7 takes a more organisational perspective to learning. It also uses the device of a decision rationale to support future learning.)

Hence it is important to consider estimation approaches that are underpinned by theories of meta-cognition and double-loop learning. Specifically, we need to better understand the impact of enhanced metacognitive awareness on the ability to improve project cost prediction and confidence (uncertainty assessment) within a software engineering context. To summarise,

1. Formal prediction systems are not consistently reliable or superior to the unaided human expert. Moreover, their inputs and parameters must be manipulated by humans with a consequent loss of their raison d'être, i.e., objectivity.
2. There is a strong tendency for professionals to display over-optimism and over-confidence. A number of experiments and empirical studies help us to understand the cognitive basis for this bias.
3. De-biasing strategies based upon utilising previous experiences, such as lessons learned sessions, have not led to noticeable improvement in prediction accuracy or the realism of uncertainty assessment.
4. There are opportunities to apply recent results from metacognition research to counteract this natural bias and consequently improve performance.

It is therefore evident that more attention needs to be paid both by researchers and practitioners into the cognitive aspects of cost estimation. To ignore this aspect is to severely limit the reach and impact of any initiatives to improve cost estimation practice. As has already been noted, formal models such as those based on machine learning algorithms have their place, but they still depend upon inputs and parameters supplied by, and outputs utilised by, software professionals who are subject to the same cares, concerns and biases of all human beings.

3.5 Practical Recommendations

Thus far, this chapter has noted the importance of effective cost estimation for software projects and contrasted this with the widespread challenges that are faced. In particular, endemic over-optimism has led to costs being systematically underestimated and over-confidence, causing estimators to believe they are more accurate than they really are. We have then reviewed the problems that are currently being experienced in terms of cost estimation, most notably the tendency to be over-optimistic (i.e., under-estimate costs) and to be over-confident (i.e., be less accurate than anticipated). This has triggered a good deal of research to try to overcome these problems, in particular through proposing formal prediction systems or models. After initial work based on the idea of generally applicable models such as COCOMO (Boehm 1984) and COCOMO II (Boehm et al. 2000), the dominant idea driving formal models has been to derive them from historical data either through statistical analysis such as regression modelling or through induction using one of the many machine learning techniques available. Despite this activity, it is not possible to strongly recommend any one formal technique, for the simple reason of a lack of consistent evidence. Thus, any recommendations must be grounded in the understanding that human judgement plays a substantial contribution.

Whilst not intended to be exhaustive, the following is a list of six practical recommendations that are supported by empirical evidence and could usefully be deployed in real-life projects:

1. Data driven
2. Sensitivity analysis
3. Multiple techniques
4. Group estimates
5. Training and reflection
6. Estimation and confidence

Data-driven estimation requires the availability of historical data on previously completed projects. Such data can be useful in three different ways. First, for analogical reasoning that can be formalised as case-based reasoning (Shepperd and Schofield 1997) or used more informally. Second, local historical data can be used for calibration purposes since there is widespread evidence to indicate that off-the-shelf approaches are problematic and that general purpose models benefit from calibration to the specific or local problem domain (Cuelenaere et al. 1987; Jeffery and Low 1990; Gulezian 1991; Yang et al. 2013). Third, for direct predictive model building, relevant, local data is necessary for training, i.e., inductive learning purposes. Naturally, the question arises about the situation when no local data is available. This might be because the software development organisation is new or because no relevant past data exists. Is the assumption that global data is inferior to local data well founded? This has vexed researchers for some time and two systematic reviews (Kitchenham et al. 2007; MacDonell and Shepperd 2007) have concluded that the evidence is mixed, and from primary studies available no definitive answer is possible. In some ways, the question of local vs. global data is somewhat artificial and more relevant is how relevant is the global or cross-company data? However, a recommendation is to inform any cost estimation with local data, including past estimation performance data wherever possible. If circumstances do not allow this, then global data, after careful consideration of its relevance, is the next best option.

Sensitivity analysis is not common practice, yet in the face of uncertainty, it is a very useful means of determining the vulnerability of an estimate to particular assumptions and the level of confidence that can be placed in that estimate. Such analysis can be highly sophisticated (Saltelli et al. 2000) or use simple Monte Carlo methods (Fishman 1996). Wagner (2007) illustrates how these ideas can be deployed using a COCOMO model and finds that the code size estimate dominates the effort prediction, but less obviously that there are significant second order effects between the different cost drivers due to the multiplicative nature of the model. This kind of analysis can also be valuable when the uncertainty surrounding an estimate is unacceptable, thereby helping the estimator identify the most important sources of variability and could then take steps to reduce this uncertainty through further investigation, simulation, etc. of the key parameters or inputs.

Using more than one estimation method or *multiple techniques* is another important consideration. Although an obvious recommendation for practitioners, this has not been widely researched and the evidence base is quite limited. Kitchenham et al. (2002) conducted an empirical study of 145 projects at a large software house where estimators were required to use a minimum of two techniques and then select one estimate to be the basis of client-agreed budget. The advantage, over simply using the mean is that if one estimate is misleading it will not ‘contaminate’. MacDonell and Shepperd (2003b) explored a similar question and also found that not only was no one technique best but using the mean was also sub-optimal. By selecting one technique, or perhaps investigating more deeply, requires more consideration and discussion than the formulaic application of an averaging technique.

Group estimates should also be considered as a practical estimation technique. Again, surprisingly considering they have been promoted since Boehm’s seminal *Software Engineering Economics* (Boehm 1981) described a wideband Delphi process, but there has been limited research and therefore evidence. Taff et al. (1991) proposed a related approach that they termed Estimeetings, however, little empirical support is offered in terms of their effectiveness. Passing and Shepperd (2003) investigated the impact of group discussion and iterated estimates and found that both checklists and group discussions significantly contribute to improved estimation. The limitation of this study was that it involved Masters students rather than professionals and was in an artificial setting. Reporting similar results, Moløkken and Jørgensen (2004) found a significant and substantial effect in terms of the tendency for group estimates to be less optimistic both for group decisions and the individual post-group discussion decisions to be less optimistic than the original estimates.

The lack of systematic *training and reflection* is another improvement opportunity. As Jørgensen puts it 'the focus on learning estimation skills from software development experience seems to be very low' (Jørgensen 2004). The challenges are that the various cognitive biases described in Sect. 3.4 are deeply ingrained and de-biasing strategies not necessarily effective. Consequently, emphasis should be given to reflection but structured in order to guide estimators beyond the shallow reflections that some researchers have found, such as 'the estimate was too low because insufficient time was allocated'! Researchers have also found that emphasising metacognitive skills can also significantly improve performance.

Finally, practitioners need to keep in mind that because an estimate is a probabilistic statement, it has two dimensions (the *estimate* and *confidence*) and therefore, it is not well represented by a single point value even if this is required as the final outcome of the decision making process, e.g., the bid value. To give an example, estimating 1,000 person-hours ± 10 person-hours is a very different proposition to 1,000 person-hours ± 500 person-hours. Even this may not be adequate since it is unclear whether it means that an actual effort of 1,510 person-hours is deemed impossible or merely very unlikely. Moreover such a formulation imposes a symmetric distribution which may not properly reflect the estimator's beliefs. Jørgensen recommends a confidence value in a range, e.g., 80 % confidence between 500 and 1,500 person-hours. This allows some simple trade-offs between precision and confidence to be exploited. A richer picture still is obtained by describing the estimate as a probability distribution, e.g., as a 3-point estimate and a triangular distribution. Either way, failing to regard estimates as probabilities indicates a failure to appreciate their true nature and therefore the opportunity to learn and improve.

The above list contains some simple, practical, general and evidence-based recommendations for software cost estimation. It is not a panacea, and there are many other challenges that have not been fully addressed. Nevertheless, given the importance of software, software projects and effective cost management, they may offer some useful steps forward.

3.6 Follow-Up Sources of Information

There are several comprehensive systematic reviews on research into cost estimation. Jørgensen and Shepperd (2007) gives general coverage of the different research activities being undertaken and Simula have continued to update the database of sources subsequent to its publication.[5] A second, more specialised on the role of human experts, and slightly older systematic review is by Jørgensen (2004). The review by Riaz et al. (2009) focuses on cost estimation in a software maintenance context.

[5] The bibliographic database can be found at www.simula.no/BESTweb

Cost estimation generally takes place in the wider setting of a software project. There are many good textbooks, such as Hughes and Cotterell (2009) on project management and Sommerville (2010) on software engineering and the set of guidelines published as the SWEBOK (Abran and Bourque 2004).

In terms of making sense of published empirical research comparing different formal models, and for designing new experiments, Shepperd and MacDonell (2012) set out a framework based on three research questions that need to be addressed.

Glossary

Absolute residuals a simple and robust means of assessing the predictive accuracy of a prediction system. It is defined simply as: $|y_i - \hat{y}_i|$ where y_i is the true value for the ith project and $\hat{y}_i$ the estimated value. This gives the error, irrespective of direction, i.e., an under- or over-estimate. The mean residual (keeping the direction of error) gives a measure of the degree of bias.

Cognitive bias these are patterns of thinking about problem solving or decision-making that distort and lead people to 'sub-optimal' choices. Because of the ubiquity of many such biases, they are classified and named, e.g., the anchoring bias. See the pioneering work of Tversky and Kahneman (1974).

Double loop learning this differs from ordinary or single-loop learning in that one not only observes the effects of the process, but also understands the external factors that influence the effects. This was initially promoted by Argyris and Schön as a way of promoting effective organisational behaviour (Argyris and Schön 1996).

Estimation by Analogy (EBA) uses some form of case-based reasoning where a new or target case which is to be solved is plotted in feature space (one dimension per feature) and some distance metric used to determine past proximal cases from which a solution can be derived. For a general account of CBR see the pioneering work by Kolodner (1993) and for its application to software engineering see Shepperd (2003).

Expert Judgement this is something of a catch all description for a range of informal approaches to estimation. Jørgensen describes it as 'unaided intuition ("gut feeling") to expert judgment supported by historical data, process guidelines and checklists ("structured estimation")' (Jørgensen 2004). Despite it being a widespread estimation approach, it can still be criticised for its reasoning not being open to scrutiny since the reasoning process is 'non-recoverable' (Jørgensen 2004), not repeatable or easily transferable from existing experts to others.

Formal prediction system or formal model for cost prediction is characterised by repeatability so that different individuals applying the same inputs should generate the same outputs (with the exception of prediction systems based on stochastic search [also see Chap. 15 on search-based project management] where this will tend to be true over time (Clark et al. 2007), but not necessarily for a single utilisation). Examples of formal systems range from simple algorithmic models, such as COCOMO, to complex ensembles of learners.

Machine Learning this is a branch of applied artificial intelligence based on inducing prediction systems from historical data, i.e., reasoning from the particular to the general. There are a wide range of approaches including neural networks, case-based reasoning, rule induction, Bayesian methods, support vector machines and population search methods such as genetic programming. Standard textbooks that provide overviews of these techniques include Witten et al. (2011).

Mean magnitude of relative error (MMRE) this is a widely used, although now heavily criticized (Kitchenham et al. 2001; Foss et al. 2003; Shepperd and MacDonell 2012), measure of predictive accuracy defined as:

$$MMRE = \sum_{1}^{n} \left(\left| {(x_i - \hat{x}_i)} \Big/ {x_i} \right| \right) \Big/ n$$

where x is the true cost for the ith project, $\hat{x}$ is the estimated cost and n the total number of projects.

Metacognition this refers to 'thinking about thinking' (Flavell 1979) and is an awareness and monitoring of one's thoughts and performance. It encompasses the ability to consciously control the cognitive processes involved in learning such as planning, strategy selection, monitoring and evaluating progress towards a particular goal and adapting strategies as, and when, necessary to reach that goal (Ridley et al. 1992).

Over-confidence refers to the tendency of an estimator to value precision over accuracy. Typically, one might express confidence in an estimate as the likelihood that the true value falls within a specified interval. For example, stating that one is 80 % confident that the actual effort will fall within the range 1,000–1,200 person-hours implies that this will occur 8 out of 10 times. If the true value falls into the range less frequently this implies over-confidence. Jørgensen et al. (2004) reported that over-confidence was a widespread phenomenon and that at least one contributor was the fact that managers often interpret wide intervals as conveying a lack of knowledge and prefer narrow but less accurate estimates.

Over-optimism refers to the situation where the estimation error is biased towards an under-estimate. Many studies indicate that this is the norm in the software industry with a figure of 30 % being seen as typical (Jørgensen 2004).

Prediction whilst 'prediction' and 'estimation' are often used interchangeably, we use 'prediction' to mean a forecast or projection, and 'estimate' to connote a guess or rough and ready calculation.

Single-loop learning Argyris and Schön (1996) characterise this as focusing on restrictive feedback so that the individual or organisation only endeavours to improve a single metric without external reflection upon the process, i.e., double loop learning.

References

Abran A, Bourque, P (2004) SWEBOK: guide to the software engineering body of knowledge. IEEE Computer Society

Albrecht AJ, Gaffney JR (1983) Software function, source lines of code, and development effort prediction: a software science validation. IEEE Trans Softw Eng 9:639–648

Argyris C, Schön D (1996) Organizational learning II: theory, method and practice. Addison-Wesley, Reading, MA

Armstrong S (2007) Significance tests harm progress in forecasting. Int J Forecast 23:321–327

Benington HD (1956) Production of large computer programs. In: Symposium on advanced computer programs for digital computers, ACR-15

Boehm BW (1981) Software engineering economics. Prentice-Hall, Englewood Cliffs, NJ

Boehm BW (1984) Software engineering economics. IEEE Trans Softw Eng 10:4–21

Boehm B, Abts C, Brown W, Chulani S, Clark BK, Horowitz E, Madachy R, Reifer D, Steece B (2000) Software cost estimation with COCOMO II. Pearson/Prentice Hall, Englewood Cliffs, NJ

Borkowski JG, Carr M, Pressley M (1987) Spontaneous strategy use: perspectives from metacognitive theory. Intelligence 11:61–75

Buehler R, Griffin D (2003) Planning, personality, and prediction: the role of future focus in optimistic time predictions. Organ Behav Hum Decis Process 92:80–90

Buehler R, Griffin D, Ross M (1994) Exploring the "Planning Fallacy": why people underestimate their task completion times. J Pers Soc Psychol 67:366–381

Clark J, Dolado JJ, Harman M, Hierons RM, Jones B, Lumkin M, Mitchell B, Mancoridis S, Coutinho SA (2007) The relationship between goals, metacognition, and academic success. Educate 7:39–47

Coutinho SA (2007) The relationship between goals, metacognition, and academic success. Educate 7:39–47

Cuelenaere A, van Genuchten M, Heemstra F (1987) Calibrating a software cost estimation model - why and how. Inf Softw Technol 29:558–567

Dawson TL (2008) Metacognition and learning in adulthood. Developmental Testing Service LLC, Northampton, MA

DeMarco T (1982) Controlling software projects: management, measurement and estimation. Yourdon Press, New York

El Emam K, Koru G (2008) A replicated survey of IT software project failures. IEEE Softw 25:84–90

Ellis P (2010) The essential guide to effect sizes: statistical power, meta-analysis, and the interpretation of research results. Cambridge University Press, Cambridge

Fishman G (1996) Monte Carlo: concepts, algorithms, and applications. Springer, New York

Flavell JH (1979) Metacognition and cognitive monitoring: a new area of cognitive-developmental inquiry. Am Psychol 34:906–911

Flyvbjerg B (2008) Curbing optimism bias and strategic misrepresentation in planning: reference class forecasting in practice. Eur Plan Stud 16:3–32

Flyvbjerg B, Bruzelius N, Rothengatter W (2003) Megaprojects and risk: an anatomy of ambition. Cambridge University Press, Cambridge

Foss T, Stensrud E, Kitchenham B, Myrtveit I (2003) A simulation study of the model evaluation criterion MMRE. IEEE Trans Softw Eng 29:985–995

Griffin D, Buehler R (1999) Frequency, probability, and prediction: easy solutions to cognitive illusions? Cogn Psychol 38:48–78

Grimstad S, Jørgensen M, Moløkken-Østvold K (2006) Software effort estimation terminology: the tower of Babel. Inf Softw Technol 48:302–310

Gulezian R (1991) Reformulating and calibrating COCOMO. J Syst Softw 16:235–242

Heemstra FJ (1992) Software cost estimation. Inf Softw Technol 34:627–639

Hughes RT (1996) Expert judgement as an estimating method. Inf Softw Technol 38:67–75

Hughes RT, Cotterell M (2009) Software project management. McGraw-Hill, London

Humphrey W (2000) Introducing the personal software process. Ann Softw Eng 1:311–325

Jeffery DR, Low GC (1990) Calibrating estimation tools for software development. Softw Eng J 5:215–221

Jørgensen M (2004) A review of studies on expert estimation of software development effort. J Syst Softw 70:37–60

Jørgensen M (2010) Identification of more risks can lead to increased over-optimism of and over-confidence in software development effort estimates. Inf Softw Technol 52:506–516

Jørgensen M, Grimstad S (2012) Software development estimation biases: the role of interdependence. IEEE Trans Softw Eng 38:677–693

Jørgensen M, Gruschke T (2005) Industrial use of formal software cost estimation models: expert estimation in disguise? In: Proceedings of EASE, Keele, UK

Jørgensen M, Gruschke T (2009) The impact of lessons-learned sessions on effort estimation and uncertainty assessments. IEEE Trans Softw Eng 35:368–383

Jørgensen M, Moløkken-Østvold K (2006) How large are software cost overruns? A review of the 1994 CHAOS report. Inf Softw Technol 48:297–301

Jørgensen M, Shepperd M (2007) A systematic review of software development cost estimation studies. IEEE Trans Softw Eng 33:33–53

Jørgensen M, Sjøberg DIK (2003) An effort prediction interval approach based on the empirical distribution of previous estimation accuracy. Inf Softw Technol 45:123–136

Jørgensen M, Teigen KH, Moløkken K (2004) Better sure than safe? Overconfidence in judgment based software development effort prediction intervals. J Syst Softw 70:79–93

Kahneman D, Tversky A (1979) Intuitive prediction: biases and corrective procedures. TIMS Stud Manag Sci 12:313–327

Kahneman D, Fredrickson B, Schreiber C, Redelmeir D (1993) When more pain is preferred to less: adding a better end. Psychol Sci 4:401–405

Kemerer CF (1987) An empirical validation of software cost estimation models. Commun ACM 30:416–429

Keung J, Kitchenham B, Jeffery R (2008) Analogy-X: providing statistical inference to analogy-based software cost estimation. IEEE Trans Softw Eng 34:471–484

Kitchenham BA (2002) The question of scale economies in software - why cannot researchers agree? Inf Softw Technol 44:13–24

Kitchenham BA, Kansala, K. (1993) Inter-item correlations among function points. In: 1st International symposium on software metrics. IEEE Computer Society Press, Baltimore, MD

Kitchenham BA, Linkman SG (1997) Estimates, uncertainty and risk. IEEE Softw 14:69–74

Kitchenham BA, MacDonell SG, Pickard L, Shepperd MJ (2001) What accuracy statistics really measure. IEEE Proc Softw Eng 148:81–85

Kitchenham BA, Pfleeger SL, McColl B, Eagan S (2002) An empirical study of maintenance and development estimation accuracy. J Syst Softw 64:57–77

Kitchenham B, Mendes E, Travassos G (2007) Cross versus within-company cost estimation studies: a systematic review. IEEE Trans Softw Eng 33:316–329

Kocaguneli E, Menzies T, Hihn J, Kang H (2012a) Size doesn't matter? On the value of software size features for effort estimation. In: Proceedings of the 8th international conference on predictive models in software engineering, New York

Kocaguneli E, Menzies T, Keung J (2012b) On the value of ensemble effort estimation. IEEE Trans Softw Eng 38:1403–1416

Kolodner JL (1993) Case-based reasoning. Morgan-Kaufmann, San Mateo, CA

Lederer A, Mendelow A (1999) The impact of the environment on the management of information systems. Inf Syst Res 1:205–222

Liu Q, Mintram R (2005) Preliminary data analysis methods in software estimation. Softw Qual J 13:91–115

MacDonell S, Shepperd M (2003a) Using prior-phase effort records for re-estimation during software projects. In: 9th IEEE international metrics symposium

MacDonell S, Shepperd M (2003b) Combining techniques to optimize effort predictions in software project management. J Syst Softw 66:91–98

MacDonell S, Shepperd MJ (2007) Comparing local and global software effort estimation models – reflections on a systematic review. In: 1st international symposium on empirical software engineering and measurement, Madrid

Mair C, Shepperd M (2005) The consistency of empirical comparisons of regression and analogy-based software project cost prediction. In: 4th international symposium on empirical software Engineering (ISESE) Noosa Heads, Australia

Mair C, Kadoda G, Lefley M, Keith P, Schofield C, Shepperd M, Webster S (2000) An investigation of machine learning based prediction systems. J Syst Softw 53:23–29

Mair C, Martincova M, Shepperd M (2009) A literature review of expert problem solving using analogy. In: 13th international conference on evaluation and assessment in software engineering (EASE), British Computer Society, Swinton, UK

Menzies T, Jalili M, Hihn J, Baker D, Lum K (2010) Stable rankings for different effort models. Autom Softw Eng 17:409–437

Menzies T, Butcher A, Cok D, Marcus A, Layman L, Shull F, Turhan B, Zimmermann T (2013) Local versus global lessons for defect prediction and effort estimation. IEEE Trans Softw Eng 39:822–834

Minku L, Yao X (2013) Ensembles and locality: insight on improving software effort estimation. Inf Softw Technol 55:1512–1528

Mittas N, Angelis L (2013) Ranking and clustering software cost estimation models through a multiple comparisons algorithm. IEEE Trans Softw Eng 39:537–551

Moløkken K, Jørgensen M (2004) Group processes in software effort estimation. Empir Softw Eng 9:315–334

Moon J (1999) Reflection in learning and professional development: theory and practice. Kogan Page, London

Myrtveit I, Stensrud E (1999) A controlled experiment to assess the benefits of estimating with analogy and regression models. IEEE Trans Softw Eng 25:510–525

Passing U, Shepperd M (2003) An experiment on software project size and effort estimation. In: ACM-IEEE international symposium on empirical software engineering (ISESE 2003)

Riaz M, Mendes E, Tempero E (2009) A systematic review of software maintainability prediction and metrics. In: 3rd international symposium on empirical software engineering and measurement, ACM Computer Press, pp 367–377

Ridley D, Schutz P, Glanz R, Wernstein C (1992) Self-regulated learning: the interactive influence of metacognitive awareness and goal-setting. J Exp Educ 60:293–306

Saltelli A, Tarantola S, Campolongo F (2000) Sensitivity analysis as an ingredient of modeling. Stat Sci 15:377–395

Schön DA (1983) The reflective practitioner. Basic Books, New York

Shepperd M (2003) Case-based reasoning and software engineering. In: Aurum A, Jeffery R, Wohlin C, Handzic M (eds) Managing software engineering knowledge. Springer, Berlin

Shepperd MJ, Kadoda G (2001) Comparing software prediction techniques using simulation. IEEE Trans Softw Eng 27:987–998

Shepperd M, MacDonell S (2012) Evaluating prediction systems in software project estimation. Inf Softw Technol 54:820–827

Shepperd MJ, Schofield C (1997) Estimating software project effort using analogies. IEEE Trans Softw Eng 23:736–743

Sommerville I (2010) Software engineering. Pearson, Hemel Hempstead, UK

Song Q, Shepperd M (2007) Missing data imputation techniques. Int J Bus Intell Data Mining 2:261–291

Song Q, Shepperd M (2011) Predicting software project effort: a grey relational analysis based method. Expert Syst Appl 38:7302–7316

Strike K, El Emam K, Madhavji N (2001) Software cost estimation with incomplete data. IEEE Trans Softw Eng 27:890–908

Symons CR (1988) Function point analysis: difficulties and improvements. IEEE Trans Softw Eng 14:2–11

Taff LM, Borcering JWB, Hudgins WR (1991) Estimeetings: development estimates and a front-end process for a large project. IEEE Trans Softw Eng 17:839–849

Tversky A, Kahneman D (1974) Judgment under uncertainty: heuristics and biases. Science 185:1124–1131

Wagner S (2007) An approach to global sensitivity analysis: FAST on COCOMO. In: 1st International symposium on empirical software engineering and measurement (ESEM 2007). IEEE Computer Society, pp 440–442

Whitfield D (2007) Cost Overruns, delays and terminations: 105 outsourced public sector ICT contracts. The European Services Strategy Unit

Willis R (1985) Invited review: critical path analysis and resource constrained project scheduling—theory and practice. Eur J Oper Res 21(2):149–155

Witten I, Frank E, Hall M (2011) Data mining: practical machine learning, tools and techniques. Morgan Kaufmann, Burlington, MA

Yang Y, He Z, Mao K, Li Q, Nguyen V, Boehm B, Valerdi R (2013) Analyzing and handling local bias for calibrating parametric cost estimation models. Inf Softw Technol 55:1496–1511. Software Engineering Body of Knowledge (SWEBOK). Software Engineering Body of Knowledge (SWEBOK) Home. http://www.computer.org/portal/web/swebok/home

Biography Martin Shepperd received a Ph.D. in computer science from the Open University in 1991 for his work in measurement theory and its application to empirical software engineering. He is Head of Department and holds the chair of Software Technology and Modelling at Brunel University, London, UK. He has published more than 150 refereed papers and 3 books in the areas of software engineering and machine learning. He is a fellow of the British Computer Society.

Chapter 4
Human Resource Allocation and Scheduling for Software Project Management

Constantinos Stylianou and Andreas S. Andreou

Abstract Software project management consists of a number of planning, organizing, staffing, directing and controlling activities. Human resources feature prominently in all of these activities and, as a consequence, they can affect and determine project management decisions. Therefore, in order to help guarantee the success of a software project, managers must take into consideration this type of resource when performing the aforementioned activities. This chapter specifically investigates human resources from a planning perspective and, in particular, focuses on the responsibilities of allocating developers and teams to project tasks, scheduling developers and teams, as well as forming development teams. These responsibilities are often challenging to undertake because they are accompanied by time, budget and quality constraints, which software project managers find difficult to balance correctly. The purpose of the chapter is to explore the most recent research work in the field of human resource allocation and scheduling, and to specifically examine the motivation behind each approach and the goals and benefits to real-world practitioners. In addition, the chapter investigates development team formation, which can be considered as an indirect method of allocating human resources to a software project. This perspective, in particular, sheds light on current and future trends, which lean towards incorporating human-centric aspects of software development in planning activities.

C. Stylianou
Department of Computer Science, University of Cyprus, Lefkosia, Cyprus
e-mail: cstylianou@cs.ucy.ac.cy

A.S. Andreou (✉)
Department of Electrical Engineering/Computer Engineering and Informatics, Cyprus University of Technology, 31 Archbishop Kyprianou Avenue, P.O. Box 50329, Lemesos 3036, Cyprus
e-mail: andreas.andreou@cut.ac.cy

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_4, © Springer-Verlag Berlin Heidelberg 2014

4.1 Introduction

Human resource allocation involves assigning a developer to carry out a task and attempts to answer “Who will work on what?”, whereas human resource scheduling involves specifying the time frame in which a developer will work on a task and tries to answer “Who will work when?”. They are both part of a software project manager’s planning activities, but depending on the practices followed by each software development company and the information regarding the actual software project, the way they are carried out can vary. In some cases, a software project manager is required to allocate and schedule one or more developers to each task, whereas in other cases, tasks are distributed to already predefined teams of developers. Finally, there are times when a project manager needs to only put together a group of developers without assigning them to specific tasks or scheduling them, in effect carrying out team formation. Generally, they are both carried out at the initial phases of a software project. However, in most cases, information at the beginning of a software project is often imprecise or unavailable. As a result, they are considered two of the most challenging responsibilities of software project managers, and they are part of the first activities that can significantly affect the progression and overall success of a software development project. Adding further complexity for project managers is the fact that allocating developers to tasks and scheduling tasks and developers are not independent activities and that treating them so may be considered unsuitable (Chang et al. 2008). Allocation and scheduling both affect the availability of developers, so in order to avoid conflict, both activities need to be worked on simultaneously.

To help them, software project managers make use of their own past experiences and previously acquired knowledge together with the wide range of available commercial tools and techniques, such as Microsoft Project, Project KickStart, Basecamp, MatchWare MindView and RationalPlan MultiProject. A study of the impact of project management information systems by Raymond and Bergeron (2008) found that such systems improve efficiency and effectiveness with respect to project planning and control activities, as well as general project performance and overall success. However, not many of these available applications are tailor-made for the software development industry. A survey conducted by McBride (2008) highlights this, especially for monitoring, controlling and coordination activities, in which project managers use a number of different mechanisms for a given activity but also use the same mechanism for a number of activities.

As a result of a lack of specific tools, software project managers still seek the aid of more practical and intelligent models and techniques to overcome the challenges posed within these activities. Many research studies have contributed to the attempt to solve the problem of human resource allocation and scheduling specifically for the software industry, in particular, by utilizing specialized operational research techniques found in the fields of mathematical modelling and computational intelligence. The first part of the literature review in this chapter (Sect. 4.2), therefore, is dedicated to providing an overview of some of the most recent approaches proposed that specifically focus on the use of these techniques in order to handle the most common human resource allocation and scheduling issues and to provide an automated way of supporting software project managers with practical benefits to the software industry.

Given that even to this day a large percentage of software projects are severely challenged or considered to have failed, combined with the fact that human resources are considered the only type of resource in a software project, one of the directions that the software engineering research community is trying to establish for the software development industry to follow involves including human-centric aspects. Specifically, this direction attempts to promote non-technical aspects of software development as equally important as technical aspects in approaches for human resource allocation, scheduling and team formation.

A leading trend focuses on taking into account the personality types of developers when assigning them to tasks and also when grouping them into development teams. More and more studies are now being performed aiming to observe the effects of personality types on performance, productivity, software quality and job satisfaction. Also, there have been attempts to determine the personality types required of different software development professionals, such as system analysts, programmers, testers, etc. Therefore, the second part of the literature review in this chapter (Sect. 4.3) presents various human resource allocation and scheduling approaches, as well as team formation strategies for software development teams, that incorporate personality types and results of their application in the software industry.

The aim of this chapter is to present a better understanding of research efforts for human resource allocation, scheduling and team formation in the software industry and to highlight the trends emerging. By conducting this survey, the aim is to identify any gaps in current research of software project human resource allocation in order to suggest possible areas for further investigation. To facilitate this, an initial search of digital libraries was carried out to extract primary studies and survey papers concerning software human resource allocation scheduling and team formation. The electronic databases used included IEEE Xplore, ScienceDirect, ACM Digital Library, SpringerLink, Google Scholar and Wiley Interscience. Other sources were later incorporated by using the reference lists of retrieved studies, as well as conference proceedings and technical reports. Irrelevant or duplicate articles and papers that the initial search generated were then eliminated. Due to the different terminology often used to describe human resource allocation and scheduling, a number of keyword search strings were used to obtain related articles. Specifically, conjunctions of different phrases were built by selecting from a list of various related keywords, including terms such as (1) “resource,” “human resource,” “developer,” “software developer,” “team,” “software team”; (2) “scheduling,” “planning,” “allocation,” “formation,” “assignment”; and (3) “software,” “software project,” “software development,” “software management.”

4.2 Human Resource Allocation and Scheduling Approaches

The various models and techniques adopted in the proposed attempts belong to the fields of mathematical modelling and computational intelligence as these generally contain the most commonly used methods to solve problems in the field of operational research. Operational research consists of different complex decision-making problems in various fields, such as natural sciences and engineering as well as social sciences, which are solved by locating optimal or near-optimal solutions. Human resource allocation and scheduling can be considered an operational research problem and, thus, most research efforts are focused on providing solutions using mathematical modelling and computational intelligence methods. A short description of each approach is provided so that the reader can identify the possible benefits and shortcomings of using it for allocation and/or scheduling in the real world.

Mathematical models make use of mathematical notations and concepts, such as variables, operators, equations and functions, to represent a problem and then attempt to solve the model as an optimization problem or to use the model as a prediction scheme. The types of models include linear programming, which is optimizing a linear objective function, statistical modelling and queuing theory. On the other hand, computational intelligence techniques comprise a range of nature-inspired methodologies and algorithms aiming to solve real-world problems that are both complicated and complex. Their goal is to imitate the individual and collective behaviours and qualities of living beings concerning reasoning, logic and inference, learning and processing knowledge, as well as reproduction and evolution to achieve specific goals. The most well-known techniques include evolutionary algorithms and swarm intelligence.

In general, the available methods and techniques in both categories have been adopted as means to help solve several important problems in the field of software engineering. For example, techniques that are capable of carrying out prediction have been used in models for estimating software costs and effort (Heiat 2002), classification schemes have been utilized for evaluating software quality (Khoshgoftaar and Seliya 2004), clustering algorithms have been part of attempts to group and retrieve software components in repositories (Stylianou and Andreou 2007), and methods for optimization have been applied to automatically generate test-cases (Michael et al. 1997).

4.2.1 Mathematical Modelling Approach

4.2.1.1 Linear Programming

The first type of mathematical modelling concerns linear programming, which requires a linear objective function to be minimized or maximized in order to find optimal solutions to problems described by linear relationships subject to certain problem-specific restrictions. Kantorovich (1940), a Soviet mathematician during World War II, first introduced this approach as a means to solve several planning problems for the military, including how to optimally assign, schedule and transport resources based on their availability and cost so that army expenses are reduced while enemy losses are increased. Consequently, linear programming has been considered by researchers as a suitable technique for helping software project managers in their planning activities also.

Li et al. (2007) used integer linear programming to help software development organizations cope with the pressures of limited resources and decreased time-to-market intervals by proposing two models concerning requirement scheduling and software release planning. Their first model takes into account the precedence dependencies of requirements and the skills of available teams of developers to generate a project schedule for the development of requirements of a new release within the shortest possible make-span, whereas their second model integrates requirements selection and software release planning of a project with a fixed deadline to maximize revenues in addition to providing an on-time delivery schedule. One of the assumptions of this attempt is that requirements are assigned to teams of developers to implement and not to individual developers. Additionally, for testing their proposed approach, the authors used both example and real-world data sets. The authors do point out, however, that a mathematical model cannot stand alone as a project management decision support system since other real-world factors influence the decision-making process, such as psychological, personality and political factors.

Another methodology using linear programming is presented by Otero et al. (2009) and was developed to tackle the issue of project manager subjectivity in human resource allocation. The authors highlight that ineffectual resource allocation can lead to many problems for development organizations, such as "*schedule overruns, decreased customer satisfaction, decreased employee morale, reduced product quality, and negative market reputation*" (Ejnioui et al. 2012). They, therefore, propose the best-fitted resource methodology that works to measure the suitability between the skills required by tasks and the skills possessed by the available resources. Project managers can then use the results from the methodology to decide on the most suitable (optimal) allocation of resources based on their capabilities. To test their approach, the authors provided a small sample resource allocation scenario to 30 subjects consisting of software engineers and project managers from the industry and also computer science students and professors from universities, and asked them to perform a ranking of the available resources based on the developers capabilities in the required skills. The results of this survey were then compared to the results obtained from their methodology, showing that such an approach had potential in allocating developers to tasks.

Otero et al. (2010) presented another similar multicriteria decision-making methodology for software task assignment. Here, they state that there is evidence that ineffective human resource project planning is the main reason that software development projects fail (Tsai et al. 2003). The methodology uses a desirability function as a means of assigning tasks to developers in cases where there are no optimally suitable developers in the existing workforce. It takes into account the capabilities of resources in skills, the required levels of expertise as well as the level of significance of skills required by tasks and task complexities. A significant aspect of this approach is that it can be extended to take into account project-specific factors that a software project manager decides are important according to the needs of the project. An artificial case study was used to demonstrate the methodology, consisting of a scenario where a task needed to be assigned to one of ten candidate developers based on their skill assessment and associated cost with respect to the required skills of the task. On a practical level, the authors state that the approach can be adopted by software project managers using a simple spreadsheet implementation. However, no formal description of a tool is provided. Although it seems sensible to exploit the strengths of developers based on what each task requires, this is only realistically possible if the developer is available to carry out a task. The approach, however, does not address the issue of availability when computing the desirability function and does not deal with human resource scheduling, which often influences or comes hand-in-hand with allocation.

4.2.1.2 Probabilistic Modelling

Probabilistic modelling is a mathematical modelling approach that uses data (usually historical data) to forecast the conditions of different future states of a problem by calculating the probability of certain outcomes. A characteristic of this approach is that one or more of the variables in the model can be random.

Padberg (2001) presented a probabilistic project scheduling model, which focused on using scheduling strategies to help software development organizations to manage their human resources more effectively, arguing that software developers are the most valuable resources and that software project managers need a useful scheduling support tool as opposed to a common cost estimation tool that simply predicts the overall development effort needed to carry out a project. Specifically, in the approach scheduling strategies represent, in quantitative terms, the effect of decisions regarding development costs and duration on the current state of a project. Once a strategy is fixed it is inserted into the model to compute a probability distribution estimating the completion time and cost by using several technical and non-technical factors, such as scheduling constraints, adopted software processes and the complexity of components to be developed, as well as the skills and experience of the human resources. Stochastic optimization techniques are then applied to optimize the expected duration or the cost of the project with regard to the allocated resources. It is important to model the intrinsic uncertainty that is part of the software process regarding the duration of activities and also the events that occur during a project. The author, therefore, claims that using a probabilistic approach can help deal with the fact that events in a project can occur with a particular likelihood. The approach considers a project to be broken down into components to which only one team is assigned at any given time. An advantage to the approach is that it allows a team to interrupt their work on a component in order to rework a previously completed component. In addition, it takes into account the availability of development teams as well as the precedence relationships between components. However, overall this approach can only be applicable in software companies that have predefined teams of developers, with each team possessing the know-how to undertake the development of the component. For small-to-medium sized companies that do not often have such luxury, this could be impractical.

Padberg (2002, 2003) later implemented this previous probabilistic scheduling model as a discrete simulation model for project managers to use as a tool to provide feedback and comparisons among varying strategies and also implemented a variation of the value iteration algorithm to generate optimal scheduling policies in the model (Padberg 2004, 2006). The premise of these works remains the same as in his previous approaches: that uncertainty inherent in the task durations can only allow a software project manager to create a schedule wherein the duration and cost are "*likely*" to be minimized, and so it is vital for software project managers to be able to apply dynamic scheduling policies.

4.2.1.3 Queuing Theory

Queuing theory can be used as a mathematical model to simulate a system providing services to customers (human or otherwise) as they wait in line to be served. In general, this method attempts to minimize the duration and size of delays subject to constraints and, therefore, has practical applications in problems such as scheduling, employee allocation, facility design and management, and traffic flow management.

Antoniol et al. (2004a) used this technique in their approach concerning the allocation of resources in a large software maintenance project. Specifically, the authors made use of stochastic simulations of queuing networks as an instrument to evaluate the probability that the project meets its deadline as the project is being carried out.

Jalote and Jain (2004) implement a critical path/most immediate successor first approach to resource allocation targeting software projects that are to be developed by multiple teams across different geographically distributed time zones. With a rise in the number of organizations adopting global software development, project managers face new communication and coordination issues in addition to technical and managerial problems. Therefore, they suggest a 24-hour software factory model that utilizes project task precedence graphs and available resources to satisfy three types of constraints: operational, skill and resource, in order to generate a near-to-optimal software project schedule with the shortest make-span. Further reading on global software development is available in Chaps. 9 and 10, which discuss in detail various aspects of managing IT projects developing software across the globe and motivating virtual team members involved in global IT projects, respectively.

4.2.1.4 Constraint Satisfaction

Constraint satisfaction is a method that is adopted as a means of modelling and finding solutions to combinatorial problems by imposing conditions on variables in mathematical functions that are all required to be satisfied. They feature in many artificial intelligence fields and other disciplines, including planning, scheduling and logistics. Well-known examples of problems that can be solved using this method include map colouring, job shop scheduling and even Sudoku puzzles. With respect to the software industry, the constraints regarding development projects predominantly concern the budget, the schedule and the quality of the software products. Therefore, this method is adopted in order to attempt to satisfy the restrictions surrounding these issues.

Barreto et al. (2005) proposed the use of constraint satisfaction as an optimization approach to software project staffing, stating that process productivity and product quality are highly associated with the abilities of the available resources. The abilities taken into consideration included skills, knowledge, experience, capabilities and roles, and together with the characteristics of a project's activities and any development organization constraints, various utility functions can be maximized or minimized depending on the project manager's needs. The possible optimizers implemented consisted of most or least qualified team, cheapest team, smallest team, and best partial solution team. It is assumed that tasks are broken down into small units of work to which only one developer can be assigned. Once the software project manager decides what these tasks are, the tool performs optimization in order to locate the developer assignments that best fit the chosen utility function. The approach concentrates solely on the allocation of resources, while the starting and finishing times of tasks are known beforehand.

As an extension to their previous approach, Barreto et al. (2008) incorporated a mechanism to also handle developer productivity. The authors state that the time taken to carry out a task is affected by the developer's level of productivity. Hence, the approach proposes various productivity modifiers computed based on the experience, the profession or the activity itself. A software project manager selects to apply one of these modifiers, and then a new duration for each task is estimated accordingly (either increasing or decreasing it based on the developer assigned). A new utility function was subsequently implemented to enable assignments yielding the fastest team. The ability to factor in productivity is very important for software companies as the accuracy of budgets and schedule estimates can be improved.

4.2.2 Computational Intelligence Approaches

4.2.2.1 Evolutionary Algorithms

Evolutionary algorithms are a class of population-based algorithms that stem from the theory of natural evolution. They are most widely used to solve search-based problems that require some form of optimization since they are able to explore and exploit a problem's search space more efficiently and effectively to locate the best or near-best solutions. Consequently, evolutionary algorithms have been commonly applied directly or indirectly to the problem of human resource allocation and scheduling in software development projects. In order to find the optimal or near-optimal solutions, evolutionary approaches evaluate each individual in the population, which represents a candidate solution, using an objective function that rates the fitness of solutions and checks whether they satisfy various constraints. Stronger candidates are passed into subsequent generations, whereas weaker ones are discarded, leading to the detection of (near-)optimal solutions. Chapter 15 discusses a number of approaches that adopt such algorithms, though several have been selected to be presented in the remainder of this section.

One of the earliest instances of using evolutionary algorithms for software project allocation and scheduling is found in the work of Chang et al. (1994), who formalized a model for software project management, namely SPMNet, in the mid-1990s. Their approach focuses on the fact that software development organizations fail to assign the right developers to the right tasks due to the difficulties faced by project managers in handling the high level of complexity involved in finding optimal or near-optimal schedules. Their approach employs a single-objective genetic algorithm as a "*schedule optimizer*" aiming to minimize the total duration and cost of a software project through a process of assigning software developers to tasks (Chang et al. 1994, 1998). One of the practical benefits of the formal software management model proposed is that it allows software project managers to track the progress of a project by working together with developers and customers. It also addresses the issue of risk management by enabling the pre-execution of SPMNet and, hence, predicting the future states of a project. Over the years, this model has been significantly extended to support features to deal with additional software project management issues, such as partial assignment of developers to tasks, developer overload and multiple project scheduling (Chang et al. 2001), in addition to developer reassignments, task suspensions and resumptions, learning and task-specific deadlines (Chang et al. 2008).

Ge and Chang (2006) used the schedule optimizer mentioned above to implement a capability-based scheduling framework in which task durations are calculated through system dynamics simulation that focused on the capabilities of the available personnel. The authors state that it is important to consider developers' capabilities because they can influence a team's average productivity, which is determined by factors such as individual productivity, overworking and communication overhead. Being able to simulate the effect of an assignment based on the capabilities of the developer going to carry out a task could provide software project managers vital information at any stage during the project. However, exact details on how system dynamics simulation manages to generate task durations are not provided, which severely limits the assessment of its applicability in real-world settings.

An extension to the capability-based scheduling framework is suggested by Jiang et al. (2007), who incorporate personnel risks based on historical data during the assignment process. This addition is aimed at helping software project managers identify, analyse and monitor possible risk factors arising from human activities (for example, late-in-the-day coding) and allow them to regulate resource assignment. Furthermore, the authors adapt the previous genetic algorithm to a multi-objective schedule optimizer employing a weighted sum method to allow for tradeoff solutions to be generated. Another approach using multi-objective optimization was implemented again by Ge (2009) to provide a framework for scheduling and rescheduling software projects. The approach takes into account the skills and capabilities of available developers and attempts to provide an optimal project schedule based on efficiency (minimum cost and duration) and also stability factors (minimum impact of disruptions caused by rescheduling developers).

Alba and Chicano (2005, 2007) also employed genetic algorithms to develop an automated tool to allocate resources to tasks taking into account duration, resource skills, cost and global complexity. Their research work was centered on the fact that one of the goals of software project managers is to reduce both the cost and the duration of software development projects even though these two goals can be conflicting. Each individual in the population is an assignment matrix representing the allocation of developers to tasks. The quality of each assignment matrix regarding cost and duration is evaluated through two objectives using the salary of each developer, the degree of dedication each developer is permitted to work on each task and the effort required for each task. The project's schedule is constructed directly as a result of the developers allocated to each task. As the algorithm executes, solutions converge to the optimal/near-optimal allocations and schedules. By allowing project managers to adjust weights according to the problem at hand, they have the ability to perform different scenario analyses and make better decisions regarding the software project. This is a significant feature because the importance of each criterion is subject to the software being developed within the project, thus it is reasonable to expect a software project manager in some cases to want to give emphasis on minimizing the cost of the project and in other cases to want to focus on minimizing the project's duration, depending on which criterion he or she considers more important. One drawback to the approach, however, relates to the way that developer skills are handled. Specifically, the skills possessed by developers are treated as Boolean; either a developer possesses a certain skill or does not, and this information is used to evaluate whether the skills required by the project's tasks are satisfied in the form of a constraint. However, in reality, most project managers do not treat skills in such a way but rather take into account that developers possess skills at varying levels. Therefore, the approach would make more sense to address this as part of the evaluation of objectives (i.e., as an additional criterion for assigning developers to a task) rather than as part of the assessment of the constraints. A comparison of several multi-objective evolutionary algorithms using various quality indicators was subsequently performed in Luna et al. (2011) and Chicano et al. (2011) using the same representation, that is, with each solution comprising a series of developers possessing a set of skills, which are matched against the skills required by the project's tasks. None of the experiments in this group of approaches, however, has been tested on real-world software projects. Instead, they have only been applied to a collection of simulated projects, which were created by an instance generator that randomly creates a set of tasks (with associated costs and required skills) and a set of developers (with associated salaries and skills possessed). The randomness of the generated software projects may not always accurately reflect, for example, the correlation between the skill set and salary of a developer where higher-skilled employees are more likely to be paid more.

In the approach proposed by Duggan et al. (2004), project managers supply the complexity of the packages to be developed (using McCabe's (1976) cyclomatic complexity measure) and the proficiency (from novice to expert) of the available software engineers in each of the packages. Using a multi-objective genetic algorithm, the approach aims to find an optimal solution that minimizes the number of defects per unit of complexity and minimizes the duration of the project with a specific assignment of developers. However, software project managers may find it difficult to adopt this approach because it is strictly focused on allocating and scheduling resources regarding implementation tasks of a development project and only if the project is developed using an object-oriented approach.

Kapur et al. (2008) proposed a hybrid approach, which employs integer linear programming in conjunction with genetic algorithms for resource scheduling and allocation, targeting planning product releases. The authors emphasize the fact that software developers have different levels of skills and so their goal is to help project managers assign the most qualified developers to the required tasks in order for them to achieve maximum productivity, which in turn leads to a product release offering features that maximize business value. The optimization carried out using the genetic algorithm helps software companies decide which features should be included in a particular release for its customers. The two-phase method was applied to a real-world project carried out at Chartwell Technology, which specializes in developing online gaming and gambling software, demonstrating how change requests, user requirements and improvements were planned and ordered. This approach, however, can only be used for allocating and scheduling human resources for software projects developed incrementally. Ngo-The and Ruhe (2009) further develop this two-phase approach again aimed at incremental software development. The authors use integer linear programming to fix an upper bound to the maximum possible achievable business value according to stakeholders' satisfaction and then employ a genetic algorithm to evaluate this value and subsequently find an optimal or near-optimal assignment and schedule of developers to tasks in order to plan which features are to be included in each release and which are to be postponed. The approach also allocates non-human resources, such as capital, during the assignment procedure. One of the benefits of the approach, as stated by the authors, is that project managers can replan features and reschedule resources if requirements are changed or new requirements are introduced by simply using the same two-phase approach with modified inputs and parameters.

Several attempts carried out by Antoniol et al. (2004b, 2005) had the sequence of execution of work packages and the assignment of teams to work packages evaluated using a hybrid of queuing simulation and a single-objective genetic algorithm. A shift to a multi-objective genetic algorithm was then made in the approach suggested in Gueorguiev et al. (2009), which highlights the difficulties in constructing project schedules with regard to risk. The main objective of this approach focuses on the conflicting objectives of robustness and completion time, but the approach can be used implicitly for resource usage maximization. Furthermore, the adoption of queuing simulation for task staffing and optimization for scheduling tasks are also part of a later approach in Di Penta et al. (2011), where additional features are implemented to deal with fragmentation, software developer specialization and work package dependencies. Ren et al. (2011) opted for a different approach to optimizing the sequence of execution of work packages and assigning developers to tasks by adopting a cooperative co-evolutionary method, which tries to evolve two populations of individuals simultaneously through collaboration, rather than having individuals in a single population compete against each other.

In an alternative approach, Yannibelli and Amandi (2011) proposed a knowledge-based genetic algorithm to aid project managers at the early stages of scheduling to staff software projects with the most effective employees. Specifically, the approach uses available knowledge about employees' previous participation in projects to evaluate how effective a set of resources will be if assigned to a specific activity and how effective each individual in that set will be. With this knowledge, the algorithm attempts to find feasible and optimal project schedules satisfying the precedence relationships between the activities and the human resource requirements. An important aspect of this approach is that allocations are based not only on the skills of developers but also on the level of effectivity that is the result of two or more developers working together on the same task. This is an attempt to reflect real-world practices since a software project manager may be hesitant to assign a task to a pair of developers when he or she is aware that the pair is less effective working together, even though individually the developers possess a higher level of skills than another pair of developers. It might be preferable to allocate two developers who are less skilled, but more effective working together in order to be more productive. What the authors do not make clear, however, is whether the duration of a task is specified knowing the exact number of developers to be assigned to it. What would be more flexible, if this is not the case, is having the duration of a task to actually shorten or stretch depending on the final level of effectivity resulting from the developers assigned.

Some of the approaches mentioned in this section are revisited in Chap. 15, which provides an overview of how different areas and problems of software project management have been reformulated to be solved with computational search and optimization techniques.

4.2.2.2 Swarm Intelligence

Swarm intelligence algorithms are a specific group of computational intelligence methods inspired by the behaviour of biological systems found in nature, such as the flocking of birds and the schooling of fish. The aim of swarm intelligence algorithms is to mimic how each individual in the swarm acts and interacts with other individuals in its environment to achieve a common goal shared by all individuals. In particular, swarm intelligence algorithms, such as ant colony optimization and particle swarm optimization, work similarly to evolutionary algorithms by assessing the quality of the solution that each individual in the swarm represents. In the case of human resource allocation and scheduling for software development, these types of algorithms are only just now beginning to be applied, though the general goals of the approaches still focus on minimizing the cost and duration of software projects in a similar fashion to evolutionary algorithms.

Chen and Zhang (2013) recently proposed a model that combines an event-based scheduler with ant colony optimization, aiming to provide solutions consisting of reduced project costs and more stable workload assignments. Essentially, the model considers that developer allocations are affected by specific events: the starting time of the project, the time when developers join or leave the project, and the time when developers are released from completed tasks. When one of these events occurs during the project, the event-based scheduler modifies the allocation of developers based on the priority given to tasks, the skill proficiency of developers and the current availability of the developers. Then, the method goes on to construct a new schedule by using ant colony optimization, where artificial ants are iteratively dispatched to build project plans. The practical benefits with this method are that it allows a software project manager to have the flexibility to pre-empt tasks, but also to be able to handle and avoid resource conflicts. Experiments were carried out on 80 artificial projects and 3 real-world business software projects of a departmental store, and the results demonstrated that the combination of event-based scheduling with ant colony optimization was effective in yielding solutions with the lowest project cost.

Xiao et al. (2013) also presented an approach using swarm optimization to allocate and schedule developers in a software project. The authors adopted a similar approach as Alba and Chicano (2005), but instead of using a genetic algorithm to generate solutions with optimal developer assignments and project schedules, they adopted ant colony optimization. They used the same objectives, that is, to minimize cost and duration, subject to the precedence relationships of tasks and skills of developers. The authors show through the results of optimization on 30 randomly generated project instances that this approach outperformed the original.

A particle swarm optimization algorithm was used by Gerasimou et al. (2012) in an initial investigation in maximizing human resource usage. The proposed approach aims to assign tasks to software developers based on their experience in the skills required by the tasks and, simultaneously, to generate the shortest project make-span by scheduling tasks with respect to task dependencies and developer availability. A software project manager gives each objective a weight denoting its level of importance so that the particles, which represent the start days of tasks and the developers allocated to it, can converge to an optimal as possible solution that balances the two objectives according to the weights provided. The reasoning behind this approach is that, ideally, a software project manager would want to assign a task to the developer most suitable in terms of skill. However, if the most skilful developer is assigned conflictingly to two tasks that are scheduled to execute in parallel, a software project manager is faced with the dilemma of how to handle such a conflict. Does he or she allocate a different, possibly less skilled, developer to one of the tasks but keeps the schedule the same? Or does he or she leave the assignments as they are and sets one of the tasks to start as soon as the developer becomes available again, possibly increasing the duration of the project? The challenge with implementing such an approach in real-world projects is that the cost of the project is not taken into consideration, which could also affect the allocation criteria. Furthermore, project managers may not be able to quantify the experience a developer has in particular skills easily. It is, however, critical to take into account the non-interchangeable nature of software developers considering that each developer possesses a different skill set with different levels of proficiency.

4.2.2.3 Fuzzy Logic

Fuzzy logic is regarded as a control system for solving problems based on information that is imprecise, ambiguous, uncertain or even missing, and is used to imitate the human decision-making process on a linguistic (descriptive) rather than a numeric basis. The goal is to model the vagueness of variables that do not possess a clear and crisp distinction between its possible values. Instead, it divides the variable into (usually) overlapping fuzzy sets and with the use of membership functions determines the degree to which a specific value falls into each set. It has been applied in many disciplines, such as robotics, medicine and management, where it has helped overcome the subjectivity of the decision maker.

One attempt at using fuzzy logic for software project scheduling was proposed as a decision support system by Hapke et al. (1994), who claim that, due to the uncertainty of time parameters, software project managers can only approximate the durations of development activities. The fuzzy project scheduling system proposed, therefore, creates intervals representing possible durations of tasks and aims to assign software engineers to development phases taking into consideration the completion time and maximum lateness of a software project. The time criterion is cut into lower and upper bounds generating a set of optimistic and pessimistic scenarios, which are then optimized using priority heuristic rules. Because the approach only handles the minimization of the duration of projects, its applicability in the industry may be limited. The fact, however, that human resources are considered renewable resources severely increases its limitations since it does not accurately reflect the impact that developers' capabilities can have on allocation and scheduling.

Fuzzy logic was also employed as a means for project scheduling by Callegari and Bastos (2009) in order to handle the difficulties present in pure mathematical models, for example, "*the partial loss in meaning in terms of knowledge representation*." The multi-criteria resource selection method proposed employs multi-valued logic and a set of inference rules to rank available resources according to their suitability to specific tasks, thus allowing project managers to assign resources to tasks. Specifically, a fuzzy rule matrix is constructed that stores how suitable an assignment is based on the skill level expected by a task and the current skill level possessed by the assigned developers. If-then rules then help software project managers allocate developers in order to meet the requirements of each task. One advantage of this approach is that the rules can help avoid poor utilization of developers, which is considerably important for software development companies as highly experienced developers are not wasted on tasks requiring low levels of skills. However, one criticism is its inability to handle the scheduling of developers simultaneously. This is one of the few approaches that also demonstrate a prototype tool to show how a software project manager can adopt the approach in the industry.

The approaches discussed in Sect. 4.2 are summarized in Table 4.1. They are grouped by the method/technique adopted in each proposed human resource scheduling and allocation attempt explored. As can be seen, the majority of attempts employ computational intelligence methods as a form of optimization, with the most popular technique being evolutionary algorithms.

4.2.3 Discussion

Not getting the right people to do the right job at the right time can be detrimental to the success of a software project. Various techniques borrowed from several fields have been used to help avoid this through different approaches allocating and scheduling human resources in software projects. But despite the evolution over the years, the problem still remains unsolved largely because there is no consensus on the criteria that these research approaches need to target to create a successful human resource allocation and scheduling tool. There are several notable points regarding the approaches that need to be addressed.

Firstly, even though there have been many approaches proposed, their ability to be applied in real-world environments is not always clear. First and foremost, any approach should be accompanied with some sort of tool to show exactly how the approach could be adopted by software project managers and not provide only a description of the underlying mechanisms. Additionally, the information needed to

Table 4.1 Summary of human resource allocation and staffing approaches

Technique/Method	Research Attempt	Goals/Objectives	Constraints	Data Used
Linear Programming	– Li et al. (2007)	– Minimize project duration – Maximize revenues	– Requirement dependencies – Team availability	– Simulated – Real-world
	– Otero et al. (2009) – Otero et al. (2010)	– Maximize suitability of developers	– Skill/expertise requirements – Resource requirements	– Simulated
Probabilistic Modelling	– Padberg (2001; 2002; 2003; 2004; 2006)	– Minimize project duration	– Skill/expertise requirements	– Simulated
Queuing Theory	– Antoniol, Cimitile et al. (2004)	– Minimize risk of delay	– N/A	– Real-world
	– Jalote and Jain (2004)	– Minimize project duration	– Task dependencies – Skill/expertise requirements – Resource requirements	– Simulated – Real-world
Constraint Satisfaction	– Barreto et al. (2005; 2008)	– Minimize project cost – Maximize/minimize team quality – Minimize team size – Minimize project duration	– Resource requirements – Developer availability	– Simulated
Evolutionary Algorithms	– Chang et al. (1994) – Chang et al. (1998) – Chang et al. (2001) – Chang et al. (2008)	– Minimize project cost – Minimize project duration – Minimize amount of overtime	– Developer availability – Developer overtime limit – Hard deadlines – Resource requirements	– Simulated
	– Ge and Chang (2006)	– Minimize project cost	– Developer availability – Developer overtime limit – Task dependencies	– Simulated

Technique/Method	Research Attempt	Goals/Objectives	Constraints	Data Used
Evolutionary Algorithms (continued)	– Jiang et al. (2007)	– Minimize project cost – Minimize project risk	– Developer availability – Developer overtime limit – Task dependencies	– N/A
	– Ge (2009)	– Maximize efficiency – Maximize stability	– Developer availability – Developer overtime limit – Task dependencies	– Simulated
	– Alba and Chicano (2005; 2007) – Luna et al. (2011) – Chicano et al. (2011)	– Minimize project cost – Minimize project duration	– Developer overtime limit – Task dependencies – Resource requirements – Skill/expertise requirements	– Simulated
	– Duggan et al. (2004)	– Maximize project duration – Minimize software defects	– Package dependencies – Team utilization – Communication overhead	– Simulated
	– Kapur et al. (2008) – Ngo-The and Ruhe (2009)	– Maximize business value	– Feature dependencies – Task dependencies – Developer availability – Release deadlines – Feature release requirements – Resource requirements	– Simulated – Real-world
	– Antoniol, Di Penta et al. (2004; 2005)	– Minimize project duration	– N/A	– Real-world
	– Gueorguiev et al. (2009)	– Minimize project duration – Minimize project overruns	– Work package dependencies	– Real-world

(continued)

Table 4.1 (continued)

Technique/Method	Research Attempt	Goals/Objectives	Constraints	Data Used
Evolutionary Algorithms (continued)	– Di Penta et al. (2011)	– Minimize project duration – Minimize schedule fragmentation	– Work package dependencies – Work package assignment – Skill/expertise requirements	– Real-world
	– Ren et al. (2011)	– Minimize project duration	– Work package dependencies – Resource requirements	– Real-world
	– Yannibelli and Amandi (2011)	– Maximize effectivity levels of teams	– Task dependencies – Resource requirements	– Simulated
Swarm Intelligence	– Chen and Zhang (2013)	– Minimize project cost	– Task dependencies – Developer overtime limit – Resource requirements	– Simulated
	– Xiao et al. (2013)	– Minimize project cost – Minimize project duration	– Developer overtime limit – Task dependencies – Resource requirements – Skill/expertise requirements	– Simulated
	– Gerasimou et al. (2012)	– Minimize project duration – Maximize suitability of developers	– Developer availability – Task dependencies – Resource requirements – Skill/expertise requirements	– Simulated
Fuzzy Logic	– Hapke et al. (1994)	– Minimize project duration	– Task dependencies – Developer availability	– Real-world
	– Callegari and Bastos (2009)	– Maximize suitability of developers	– N/A	– Simulated

execute any approach should be easily obtainable and measurable where necessary by software project managers, such as the dependency relationships between tasks in order to validate the feasibility of a schedule. But, for example, things like units of complexity may not be able to be provided by a software project manager, especially at the initial stages of the project. Also, some attempts have put their approach to the test using simulated or artificial projects only, without obtaining results from experiments on real-world cases. This may negatively influence a project manager's perception of the practicality of the approach.

Secondly, the majority of the research works approach the problem as a (multi-) optimization problem in that they aim to minimize/maximize several objectives, with genetic algorithms being the most prevalent of approaches. The most popular objectives involve the cost and duration of the project—two of the three dimensions/constraints of software project success—through allocating and scheduling developers in such a way that the assignments yield a balance between the two.

Thirdly, some approaches consider software developers as interchangeable resources, especially when it comes to dealing with the skills required by tasks and the skills possessed by developers. Just because two developers possess the same skill, it does not mean that they will carry out a task in the same way or within the same time. Software developers are knowledge workers, and it is with this knowledge that software is built. The varying levels of skill proficiency and experience between developers can be directly related to the salary of developers as well as to the time it takes to carry out a task. Therefore, in approaches trying to allocate and schedule developers, it is important for software project managers to be able to factor in the variance caused by different levels of performance and productivity of developers.

Skill proficiency and experience levels are not the only things that differentiate developers. Performing human resource allocation and scheduling using only these technical aspects of software development means that other, non-technical aspects are neglected. Amrit (2005) argues that approaches that are based strictly on skills and experience may be inadequate for project managers to help them handle issues like interpersonal relationships among developers. Such human, social and cultural aspects are strongly exhibited in software development companies, especially as they become more reliant on teamwork and collaboration and the emergence of distributed development. For this reason, more and more research work is being carried out that tries to incorporate non-technical aspects, especially human-centric factors, involved in software development. The following section discusses one of the current directions in this trend and, in particular, the impact of the personality type of software developers. There have been many studies surrounding this human-centric factor, and additionally, various approaches have been proposed attempting to incorporate it into human resource management either as part of team formation strategies or as part of allocation and scheduling activities.

4.3 The Implication of Software Development Personality Types

4.3.1 Personality and Type Assessment

Personality psychology is the area of psychology that examines the person—the human individual. Every individual has a set of characteristics, both organized and dynamic, that come into action or that are expressed in certain situations regarding a person's cognitive, motivational and behavioural patterns. Over the years, many categories of personality theories have been developed, including trait theories, type theories, humanistic theories and behaviourist theories, all of which aim either to understand an individual's distinctive personality features or to identify general rules applying to different individuals. During the twentieth century, with the rapid growth of the field of personality psychology, there was an equal interest in the field of personality testing. The intensive research in the field has led to many additions and modifications of personality assessment instruments, both in approach and application.

One of the most widely administered personality tests is the Myers-Briggs Type Indicator (MBTI) (Briggs Myers et al. 1998), which scores individual preferences based on the works of Carl Gustav Jung (1923). Individuals answer a psychometric questionnaire that assesses preferences relating to four dichotomies: extraversion/introversion, sensing/intuition, thinking/feeling and judging/perceiving. The personality type of an individual is determined by which alternative of each dichotomy is preferred by answering a series of forced-choice questions. It should be noted that preference of one option does not mean that the other is never used—it is simply less preferred. As a result, there are 16 possible combinations of personality types.

Another well-known personality test is the Keirsey Temperament Sorter (Keirsey and Bates 1984), which is closely related to the MBTI but, instead, uses temperaments rather than attitudes and functions. The four temperaments (artisan, guardian, idealist and rational) can be subsequently broken down into roles and further into role variants based on an individual's preference towards behaviours that are concrete or abstract, cooperative or pragmatic, directive or informative and assertive or responsive. An individual's temperament and character type is measured using a 70-item forced-answer questionnaire.

The Revised NEO-Personality Inventory (NEO-PI-R) (Costa and McCrae 1992) was introduced to measure the Five-Factor Model (FFM) personality traits of individuals (Tupes and Christal 1961). The assessment determines emotional, interpersonal, experiential, attitudinal and motivational styles represented by the five domains—neuroticism, extraversion, openness to experience, agreeableness and conscientiousness—and their subdomains (facets). The 240 items comprising this psychological personality inventory contain descriptions of behaviours that are answered using a five-point scale.

Personality tests have been used extensively in a number of academic and application disciplines that have required the adoption of personality measures. More importantly, personality tests may also be utilized for career and personnel assessment. Even as far back as the early twentieth century, tests were used to investigate the desirable psychological abilities and traits an employee was required to have. For example, Münsterberg's (1913) theories and research work in the field of industrial/organizational psychology led to the widespread development and adoption of a personality test to measure and evaluate candidate employees. His test was put into practice by the Boston Elevated Company for selecting conductors as well as by the American Tobacco Company for choosing travelling salesmen. After World War II, the use of personality testing shifted its focus on assessing employees most suited for managerial and executive positions, and many companies, including IBM, started developing their own employee personality tests. Eventually, during the 1960s, testing restarted being practised at all levels and over a wider variety of occupations (Cox 2003).

Investigating whether or not a specific type of job can be executed by a particular personality type, especially for the heavily people-oriented field of software development, is very appealing to many organizations. It can help supervisors decide on issues such as pay rises and promotions or, in a negative light, disciplinary action and dismissal. As a result, several studies have been carried out to investigate whether software development professionals possess a specific type of personality. The outcomes of these investigations can shed light on the type(s) of personality that are drawn towards a career in software development. It also enables to explore whether different professions within the industry appeal to different personality types.

4.3.2 Personality Types of Software Development Professionals

One of the earliest studies of personnel in software engineering-related occupations was performed by Moore (1991). The study was based on the Sixteen Personality Factor Questionnaire (16PF), a popular measurement tool in personality-occupation studies and extensively used to assemble personality profiles for people in various occupations (Cattell et al. 1993). In the study, the author compiled the 16PF questionnaire for four software development occupation categories—application programmers, systems analysts, technical programmers and data processing managers—in an attempt to answer the question "*Do these groups of information systems professionals share a common personality profile, or are there significant differences?*" After multiple analyses, the authors found that managers and application programmers were most similar in that they are more inclined to experiment and think freely, thus allowing them to use their imagination more, while at the same time being more outspoken and comfortable with whatever happens.

In contrast with application programmers, however, managers are more likely to be laid-back and spontaneous, more forceful and competitive, and more capable of abstract thinking. Another finding showed that systems analysts and technical programmers have a tendency to be more practical, careful and conservative than data processing managers because their work is often highly visible, not only within data processing but throughout the company. Mistakes can be costly but also embarrassing. Additionally, the study also identified managers as "*less concerned with social rules than most people*" and more likely to pursue their own desires.

Wynekoop and Walz (1998) attempted to explore the differences between information systems professionals in order to determine whether or not differences existed in personality characteristics with the rest of the general population. They surveyed three oil and gas companies with a total of 114 programmer analysts, systems analysts and project managers and administered the California Psychological Inventory Adjective Check List (ACL) (Gough and Heilbrum 1983) on the employees. The results showed that managers and systems analysts are more similar to each other than to programmers. In addition, managers and systems analysts differ from the general population on more scales than programmers but also on different scales. Another finding was that managers tend to be more logical, compliant and with more confidence than the general population, whereas analysts are more willing to keep friendly relationships with others. Generally, the study shows that IT professionals have more leadership skills, are more ambitious and reasonable and have more self-esteem and can be more disciplined than other professionals. Similarly, Smith (1989) also carried out research on IT professionals, though his work concentrated only on the personality types of systems analysts. Based on MBTI type tests, the author concluded that a high majority of systems analysts tend to prefer sensing and thinking in addition to being more introvert rather than extrovert.

Capretz (2003) attempts to provide a personality profile of software engineering employees by distributing the MBTI instrument to 100 software engineers working for the government or for private companies and students of private or public universities and comparing the results to the distribution of MBTI types of the general US adult population. The motivation behind the author's research is the fact that the majority of software engineering professionals are typecast as "*nerds*"—introverts working alone in a corner and with no intentions to interact with others. However, over the years, software development has become more complex and has given rise to specialization within the profession (such as systems analysts, designers, programmers, testers, etc.), and as a result, each role requires a corresponding personality type. Furthermore, at the time of the study, there had been very little research carried out on the degree of job satisfaction among software professionals; any profile of the software engineer constructed may have been modified due to the growth of the field's popularity. The results of the author's survey showed that the majority of software engineers are technically oriented and prefer working with facts and reason rather than with people. It was also noted that systems analysts possessed a personality type that preferred to communicate with other people and to use their enhanced thinking ability to solve organizational problems. On the other hand, programmers exhibit a personality type that excels at spotting the centre of a problem and seem to find practical solutions. Conversely, some have a high need to achieve although a low drive to socialize with other people. It is a fact that the software development field is dominated by introverts, who typically have difficulty in communicating with end users. The greatest difference, according to the author, between software engineers and the general population is that the majority of software engineers take action based on what they think rather than what somebody else feels. This, however, does not help bring software developers closer to the users.

A more recent comprehensive investigation can be found in an analysis by Varona et al. (2012), which surveys existing studies that try to profile software development professions, in order to properly understand the human resources working in the software industry, as well as to spot possible trends and changes.

4.3.3 Allocating Developers to Tasks Based on Personality Types

Even if a specific personality type can be distinguished for each software development profession, the most important question is how to make use of this information in practice when trying to allocate and schedule human resources or form software development teams. There has been a gradual rise in the number of approaches aiming to help answer this question, and this section presents some of these approaches.

The personality type of a developer can play a significant role in determining which tasks he or she is assigned to because particular individual traits can help certain developers to be more adept in coping with the requirements and characteristics of a specific task. Furthermore, a more suitable personality type assigned to a task can have a direct influence on individual performance and team efficacy (Peeters et al. 2006; Capretz and Ahmed 2010b) and group conflict and team cohesion (Karn et al. 2007), as well as contribute to the overall quality of the final software product (Fernández-Sanz and Misra 2011). In addition, when a developer is assigned to a task that suits his or her personality, then his or her level of job satisfaction can increase leading to higher productivity (Acuña et al. 2009).

Dafoulas and Macaulay’s (2001) approach to assigning developers to tasks uses dynamic role allocation to maximize productivity and performance and takes into account certain role criteria (such as the goals and objectives, skills and knowledge, as well as any personality and culture requirements) so that project managers can assign/reassign roles or activities to team members according to their suitability.

Acuña and Juristo (2004) also consider roles and human capabilities in their attempts. Their proposed model first determines the intra-personal, organizational, interpersonal and management capabilities of team members and then performs role assignment to team members based on the capabilities required by the roles and the capabilities of the available resources. Each capability is allotted a number of personality traits required to be possessed using the 16PF test as a psychometric instrument. The goal is to assign those employees possessing the 16PF personality traits nearest to the 16PF personality traits required by the role (Acuña et al. 2006).

Similarly, André et al. (2011) developed a formal model for human resource allocation focusing on the assignment of developers to roles. In this approach, rules are generated to undertake the team formation process based on the roles and competencies of developers assessed through psychological tests. These team formation rules were converted into a formal model comprising four objective functions (competence, team compatibilities, availability and distance cost) and 12 constraint types to perform human resource assignment to roles by employing heuristic algorithms (random restart hill climbing, simulated annealing, tabu search and various other combinations of heuristic approaches).

Capretz and Ahmed (2010a, b) presented an attempt at human resource allocation suggesting a mapping of job requirements and skills to personality characteristics of employees, stating that the diversity of psychological types improves effectiveness and fulfillment of software developers. Because employees are more likely to perform better if they are assigned roles that their personality traits are best suited to, the authors associate hard skills (in the form of job requirements) to soft skills (in the form of personality requirements) for various software professionals: systems analysts, designers, programmers, testers and maintenance staff. The soft skills are then matched with specific personality characteristics based on MBTI personality types, and this can allow project managers to select team members with the same personality types and assign them to the roles required in the project.

Stylianou and Andreou (2012) employed a multi-objective genetic algorithm to simultaneously allocate and schedule software developers to tasks based on the technical skills and their personality types. One of the assumptions in this approach is that the schedule of tasks is fixed, and so the allocation of developers is constrained by the time that each task has been set to execute. This approach was recently developed into a prototype intelligent decision support tool in Stylianou et al. (2012) and was extended to also accommodate situations where a software project manager wishes to allocate and schedule software developers without having the project tasks scheduled beforehand. The results obtained in these two approaches appear highly promising and demonstrate the significance of human-centric developer assignment.

4.3.4 Allocate Developers to Team Based on Personality Types

While some may argue the importance of getting a developer to work on the right task, others may argue the significance of getting developers to work right together. The approaches mentioned in Sect. 4.3.3 all focus on the relationship between developers and tasks. However, what about the relationship between developers themselves? After all, software projects are undertaken by teams and require collaboration, coordination and communication between members. The answer to this question has been explored by several groups of researchers, all trying to identify how the personality type of developers influences various facets of team work and investigate whether certain combinations of personality types improve aspects such as performance, productivity and even quality. From a software project manager's perspective, this could help him or her to understand and exploit this underlying factor effectively when deciding on allocating developers to tasks.

One area of study concerns the heterogeneity of personality types, that is, the diversity of traits possessed by developers. Rutherfoord (2001) examined the impact of diversity by comparing teams comprising different personality types with teams composed of the same personality type using the Keirsey Temperament Sorter. The results showed that groups with members of the same personality type were having more personal problems, rather than technical. The surveys revealed that members seemed to want to elaborate the project by themselves and had problems with members that did not have much of a sharing discipline. On the other hand, groups with members of different personality types seemed to have more problems at a technical level. It was also noticed that groups where all members possessed a "*supervisor*" personality type were spending too much time discussing on how tasks will be assigned, despite this matter having already been decided previously. Groups where all members possessed an "*inspector*" personality type were very quiet, and interaction between them did not seem to exist. These groups appeared, however, much more focused and responsible. Groups with different personality types among their members were very active, had robust discussions and provided different kinds of ideas. The authors also noticed that groups with "*supervisor*" personality types were very opinionated and preferred to "*follow a traditional path.*" Research by Neuman et al. (1999) investigated the relationship between work team effectiveness and two other factors: team personality elevation (TPE), defined as "*the average level of a given trait within a team,*" and team personality diversity (TPD), described as "*the variability or differences in personality traits found within a team.*" Predicting job performance using personality has conventionally been based only on the elevation, or magnitude, of traits within the group, and this has been the foundation of selection and placement strategies. Nevertheless, the authors claim that team-based designs may also require taking into account the diversity, or variability, of traits within the group in order to find correlations between personality and job performance. The research used the FFM to examine the relationship between team personality composition and work team performance. Based on the authors' interpretation of the results, teams perform better when members differ in terms of extraversion and emotional stability rather than when members are similar in terms of these traits. Conversely, team performance is likely to increase if team members possess similarly high levels of traits regarding conscientiousness, agreeableness and openness to experience. Therefore, project management decisions on employee selection can be supported by taking into account the similarity of certain traits and the dissimilarity of others within a team.

Interestingly, a large number of research studies concentrate on the effects of personality in agile methodologies, which is in itself a relatively new development approach in the field of software engineering. Project management for agile methodologies is explored in Chap. 11, which describes how agile methodologies transform the way in which communication, collaboration and coordination practices in software development projects are carried out towards a more "*people-oriented*" approach where software teams are self-managing and share the decision-making. In this chapter, the discussion focuses particularly on pair programming and how personality types are implicated. This activity involves two developers working together on one task as they alternate between the roles of "*driver*"—the developer who codes—and "*navigator*"—the developer who reviews the code. Immediately, there is a need for social interaction (in the form of communication, collaboration and cooperation) among the developers in order to reach a common goal of delivering the unit produced on time and with the required quality. Hence, this is the reason why, especially in the past several years, studies have been carried out to investigate the impact that personality types have on the performance and productivity of the pairs.

Sfetsos et al. (2006) concentrated on the diversity of personality traits and came to the conclusion that pairs with heterogeneous personalities and temperaments exhibit better performance and collaboration-viability than pairs with similar personality traits. Software project managers, therefore, can take into account personality types when allocating developers to tasks and try to match developers so as to optimize the pair's effectiveness. Similarly, Choi et al. (2008) investigated which combination of personality types yields higher pair productivity. Specifically, they tested pairs of developers with alike, opposite and diverse combinations of personality types and found that the latter combination outperformed, in terms of code productivity, the other two.

In practice, a software development company may find it easier and cheaper if developers are left to team up by themselves. Oftentimes, however, pairs will be formed based on friendships and common interests and not on optimizing productivity. If personality types are taken into account, a software project manager can assign tasks to developers yielding maximum effect with relatively little time and cost.

Acuña et al. (2009) explored the relationship between each of the five factors of the FFM and job satisfaction, performance, team cohesion, task conflict and quality in agile settings. Their quasi-experiment produced a variety of results. Firstly, they observed that the quality of the end product is positively correlated to the preferred interpersonal style of the developers. This means that teams with a high average level of extraversion will enjoy the social interaction that is promoted through agile methodologies, and all members share the same goal of making the project a success. They also noted that developers with positive attitudinal and motivational styles are also more likely to be satisfied with their job. Developers in a team that share the same high level of agreeableness and conscientiousness feel more content with their career. Staying with the factors of the FFM, Salleh et al. (2010) explored how they especially affected pair programming. The main findings here were that pairings of developers with high levels of traits relating to openness to experience were conducive to the effectiveness of the pairings. Hannay et al. (2010) provide a comprehensive survey of the research investigating the effects of personality on pair programming and its ability to predict job performance.

4.3.5 Discussion

There are two schools of thought concerning the inclusion of information regarding the personality types of developers for human resource allocation and scheduling activities in software development. On the one hand, there is a view that a developer should be assigned to a task that he or she is more suitable based on the requirements of the task and the personality type of the developer. The claim is that each software development task has a set of characteristics and requirements that can be associated to a desired set of personality traits. For example, requirements elicitation tasks involve a high level of social engagement and the ability to identify with clients to understand their needs. Therefore, an introverted individual may struggle to perform these tasks as they are more reserved and prefer working alone rather than in environments requiring high social interaction. The research does not claim that a developer cannot carry out a task if he or she does not have the right personality type; it claims that he or she would not prefer to carry out the task. Consequently, a better task-fit for a developer would result not only in better performance but also in higher job satisfaction. The more fulfilled a developer is while working on a task that he or she is suited to, the more productive and efficient he or she is. Of course, this can only work if the developer is capable of carrying out the task in the first place with regard to technical skills, knowledge and expertise, so as not to jeopardize the quality of the software being developed.

On the other hand, there is the standpoint that a developer should be assigned with other developers so that the resulting combination of personality types leads to increased performance and effectiveness. The claim here is that there are certain combinations of personality traits that can improve the productivity of the team and increase the probability of success. Some traits, such as conscientiousness, should be present in all team members, while other traits, such as extraversion, should be diverse. Contrariwise, if several developers are assigned to work together on a task, their combination of personality types may not foster the most efficient and productive working environment. This does not mean that the job cannot get done; it may just mean that a more appropriate mixture of developers in terms of personality type may be able get the job done with improved levels of communication, collaboration and coordination, which are governed by an individual's personality type. Inevitably, if this personality type blend is not "*effectual*" there will be several knock-on effects, such as lowered productivity, job satisfaction and, ultimately, software quality.

Overall, there are a limited number of approaches that attempt to incorporate personality types of software developers in order to assign them to tasks, which is expected as this is still a relatively new direction. Some do not treat the allocation and scheduling of developers as an operational research problem and, therefore, do not employ the specialized techniques and methods as the approaches presented in Sect. 4.2 do. Those that do, attempt to optimize the allocation so that the developer whose personality type is closest to a desired profile is assigned. Interestingly, all but one approach overlook dealing with the problem of resource/task scheduling altogether, which as previously mentioned, is tough to separate from allocation as both activities are affected by developer availability constraints. Hence, the ability of approaches to provide an integrated tool may be considered limited unless they are able to accommodate scheduling also.

4.4 Further Research Trends and Challenges

Incorporating aspects of personality types in allocation and scheduling is still at a young and exploratory stage, and so the applicability of approaches lacks the backup of empirical evidence demonstrating their practical benefits in order to promote their adoption by real-world software development companies. A systematic evaluation of the effect is still required to be carried out to gather such evidences, and if these continue to indicate promising results, only then can a significant evolution in team formation, as well as allocation and scheduling strategies, occur.

The desired personality types of roles, tasks or activities that form the basis of assigning suitable developers in a number of approaches are not always justified empirically. It is important that the desired personality type of a task is correctly identified in order to allocate a suitable developer, but this may pose a challenge given the different personality measures and frameworks available to assess personality types and preferences. There is currently no consensus as to which personality instrument is the most capable of providing a task's desired personality accurately.

There is a difference of opinion with respect to how personality types can be utilized—for assigning tasks or for staffing teams. Either way, the emphasis remains on gathering evidence whether taking into account personality types of developers constitutes a legitimate way forward to help software project managers make staffing decisions aiming to increase the probability of success. Ideally, future approaches will be able to support both these valid research viewpoints.

Considering the use of personality types does not aim to single out developers or discriminate against them. Instead, it is supposed to provide software project managers with additional and complementary information to help them in the allocation and scheduling of resources or, in general, task-independent team formation. Additionally, it should not substitute or force to disregard important technical factors such as knowledge, skills and experience. However, some developers may still consider such an approach intrusive, so it is therefore important to provide reassurances that the goal is to utilize this human-centric factor to achieve maximum resource usage through the strengths of developers. Ultimately, the goals and objectives of any approach are to eliminate those risks in software project management preventing development organizations from delivering their products on time, within budget and with the required level of quality.

One of the other biggest challenges for the research community is trying to find a way of blending or "*marrying*" the two styles of solution. The interdisciplinary nature of the area requires many fields to come together to provide adequate and practical solutions for the software development industry. On the one hand, optimizing technical project criteria, such as cost, duration and number of defects, is attempted to be solved as an operational research problem, whereas the human-centric approaches tend to be handled as team formation strategies. Therefore, the ideal direction for research would be to concentrate on providing a hybrid of the two solution styles in a unified software project allocation and scheduling framework, on the one hand taking advantage of the benefits of underlying techniques (mathematical modelling or computational intelligence) and on the other hand targeting technical as well as non-technical, human-centric criteria. One of the obstacles to achieving this is quantifying and measuring human-centric criteria.

4.5 Concluding Remarks

The purpose of this chapter was to give the reader an insight of the most recent research approaches to human resource allocation and scheduling in software projects. What is observed is that there is a shift from the traditional operational research approach of allocating and scheduling developers based on conventional technical criteria, such as cost and duration, towards focusing on team formation and allocation strategies, aiming to make use of both the technical abilities of developers (e.g., skills and experience) as well as their personality types for improving other criteria, such as performance, productivity and software quality.

The traditional approaches to human resource allocation and scheduling consider the attempt to solve the problem as an optimization problem and make use of mathematical modelling techniques, such as linear programming and probabilistic modelling, in addition to computational intelligence methods, such as evolutionary algorithms and swarm intelligence. In order to determine the best allocation and scheduling plan, a software project manager would have to exhaustively evaluate all the possible permutations given the project's tasks, their durations, their dependency relationships and the skills they require, in addition to the available developers, their cost and the skills they possess. Especially with larger-sized software projects, this may prove overwhelming and time-consuming for a project manager. Approaches that adopt these methods therefore attempt to provide a quicker and easier alternative.

Allocating and scheduling human resources has started to move into a more human-centric direction, with a growth in the research area investigating the addition of non-technical aspects of software development to help software project managers increase the rate of software project success. One such aspect involves the use of personality types in allocation and scheduling. Research approaches, however, are still currently limited, but as this trend is becoming more popular, evidence is accumulating showing that personality types could indeed be used to help assess how well a developer would perform certain jobs and tasks and also how effective and/or productive he or she will be with other developers. Also, some studies have concluded that caution must be given in forming diverse software development teams as these appear to generally perform better than less heterogeneous teams. Overall, this particular area of research is very promising as it contributes to dealing with the important issue of helping software projects succeed by focusing on the most important, if not the only, resource involved in software development.

References

Acuña ST, Juristo N (2004) Assigning people to roles in software projects. Softw Pract Exp 34 (7):675–696

Acuña ST, Juristo N, Moreno AM (2006) Emphasizing human capabilities in software development. IEEE Softw 23(2):94–101

Acuña ST, Gómez M, Juristo N (2009) How do personality, team processes and task characteristics relate to job satisfaction and software quality? Inf Softw Technol 51(3):627–639

Alba E, Chicano JF (2005) Management of software projects with GAs. Paper presented at the 6th metaheuristics international conference, Vienna, Austria, 22–26 August, 2005

Alba E, Chicano JF (2007) Software project management with GAs. Inf Sci 177(11):2380–2401

Amrit C (2005) Coordination in software development: the problem of task allocation. Paper presented at the 27th international conference on software engineering, St. Louis, MO, 15–21 May, 2005

André M, Baldoquín MG, Acuña ST (2011) Formal model for assigning human resources to teams in software projects. Inf Softw Technol 53(3):259–275

Antoniol G, Cimitile A, Di Lucca GA, Di Penta M (2004a) Assessing staffing needs for a software maintenance project through queuing simulation. IEEE Trans Softw Eng 30(1):43–58

Antoniol G, Di Penta M, Harman M (2004) Search-based techniques for optimizing software project resource allocation. Paper presented at the 2004 genetic and evolutionary computation conference, Seattle, WA, 26–30 Jun 2004

Antoniol G, Di Penta M, Harman M (2005) Search–based techniques applied to optimization of project planning for a massive maintenance project. Paper presented at the 21st IEEE international conference on software maintenance, Budapest, Hungary, 26–29 Sept 2005

Barreto A, Barros MO, Werner CML (2005) Staffing a software project: a constraint satisfaction approach. ACM SIGSOFT Softw Eng Notes 30(4):1–5

Barreto A, Barros MO, Werner CML (2008) Staffing a software project: a constraint satisfaction and optimization-based approach. Comput Oper Res 35(10):3073–3089

Briggs Myers I, McCaulley MH, Quenk NL, Hammer AL (1998) MBTI® Manual: a guide to the development and the use of the Myers-Briggs type indicator®, 3rd edn. Consulting Psychologists, Mountain View, CA

Callegari DA, Bastos RM (2009) A multi-criteria resource selection method for software projects using fuzzy logic. Paper presented at the 11th international conference on enterprise information systems, Milan, Italy, 6–10 May 2009

Capretz LF (2003) Personality types in software engineering. Int J Hum Comput Stud 58(2):207–214

Capretz LF, Ahmed F (2010a) Making sense of software development and personality types. IT Prof 12(1):6–13

Capretz LF, Ahmed F (2010b) Why do we need personality diversity in software engineering? ACM SIGSOFT Softw Eng Notes 35(2):1–11

Cattell RB, Cattell AK, Cattell HEP (1993) 16PF fifth edition questionnaire. Institute for Personality and Ability Testing, Champaign, IL

Chang CK, Chao C, Hsieh S et al (1994) SPMNet: a formal methodology for software management. Paper presented at the 18th annual international computer software and applications conference, Taipei, Taiwan, 9–11 Nov 1994

Chang CK, Chao C, Nguyen TT, Christensen MJ (1998) Software project management net: a new methodology on software management. Paper presented at the 22nd annual international computer software and applications conference, Vienna, Austria, 19–21 Aug 1998

Chang CK, Christensen MJ, Zhang T (2001) Genetic algorithms for project management. Ann Softw Eng 11(1):107–139

Chang CK, Jiang H, Di Y et al (2008) Time-line based model for software project scheduling with genetic algorithms. Inf Softw Technol 50(11):1142–1154

Chen W, Zhang J (2013) Ant colony optimization for software project scheduling and staffing with an event-based scheduler. IEEE Trans Softw Eng 39(1):1–17

Chicano F, Luna F, Nebro AJ et al (2011) Using multi-objective metaheuristics to solve the software project scheduling problem. Paper presented at the 13th annual conference on genetic and evolutionary computation, Dublin, Ireland, 12–16 Jul 2011

Choi KS, Deek FP, Im I (2008) Exploring the underlying aspects of pair programming: the impact of personality. Inf Softw Technol 50(11):1114–1126

Costa PT Jr, McCrae RR (1992) NEO inventories professional manual. Psychological Assessment Resources, Inc., Odessa, TX

Cox AM (2003) I am never lonely: a brief history of employee personality testing. Stay Free! 21:22–24

Dafoulas GA, Macaulay LA (2001) Facilitating group formation and role allocation in software engineering groups. Paper presented at the 2001 ACS/IEEE international conference on computer systems and applications, Beirut, Lebanon, 25–29 Jun 2001

Di Penta M, Harman M, Antoniol G (2011) The use of search-based optimization techniques to schedule and staff software projects: an approach and an empirical study. Softw Pract Exp 41 (5):495–519

Duggan J, Byrne J, Lyons GJ (2004) A task allocation optimizer for software construction. IEEE Softw 21(3):76–82

Ejnioui A, Otero CE, Otero LD (2012) A multi-attribute decision making approach for resource allocation in software projects. In: Arabnia HR, Reza H, Xiong J (eds) Proceedings of the 2012 international conference on software engineering research and practice, Las Vegas, 12–16 June 2012

Fernández-Sanz L, Misra S (2011) Influence of human factors in software quality and productivity. Paper presented at the 2011 international conference on computational science and its applications, Santander, Spain, 20–23 Jun 2011

Ge Y (2009) Software project rescheduling with genetic algorithms. Paper presented at the 2009 international conference on artificial intelligence and computational intelligence, Shanghai, China, 7–8 Nov 2009

Ge Y, Chang CK (2006) Capability-based project scheduling with genetic algorithms. Paper presented at the 2006 international conference on computational intelligence for modelling, control and automation and international conference on intelligent agents web technologies and international commerce, Sydney, Australia, 28 Nov–1 Dec 2006

Gerasimou S, Stylianou C, Andreou AS (2012) An investigation of optimal project scheduling and team staffing in software development using particle swarm optimization. Paper presented at the 14th international conference on enterprise information systems, Wrocław, Poland, 28 Jun–1 Jul 2012

Gough HG, Heilbrun AB Jr (1983) The adjective checklist manual. Consulting Psychologists Press, Inc., Palo Alto, CA

Gueorguiev S, Harman M, Antoniol, G (2009) Software project planning for robustness and completion time in the presence of uncertainty using multi objective search based software engineering. Paper presented at the 11th annual conference on genetic and evolutionary computation, Montréal, Canada, 8–12 Jul 2009

Hannay JE, Arisholm E, Engvik H, Sjoberg DIK (2010) Effects of personality on pair programming. IEEE Trans Softw Eng 36(1):61–80

Hapke M, Jaszkiewicz A, Slowinski R (1994) Fuzzy project scheduling system for software development. Fuzzy Sets Syst 67(1):101–117

Heiat A (2002) Comparison of artificial neural network and regression models for estimating software development effort. Inf Softw Technol 44(15):911–922

Jalote P, Jain G (2004) Assigning tasks in a 24-hour software development model. Paper presented at the 11th Asia-Pacific software engineering conference, Busan, Korea, 30 Nov–3 Dec 2004

Jiang H, Chang CK, Xia J, Cheng S (2007) A history-based automatic scheduling model for personnel risk management. Paper presented at the 31st annual international computer software and applications conference, Beijing, China, 24–27 Jul 2007

Jung CG (1923) Psychological types (H. Godwin Baines Trans.). London, England: Routledge; Kegan Paul Ltd

Kantorovich LV (1940) A new method of solving some classes of extremal problems. Doklady Akad Sci USSR 28:211–214

Kapur P, Ngo-The A, Ruhe G et al (2008) Optimized staffing for product releases and its application at Chartwell technology. J Softw Maint Evol R 20(5):365–386

Karn JS, Syed-Abdullah S, Cowling AJ, Holcombe M (2007) A study into the effects of personality type and methodology on cohesion in software engineering teams. Behav Inf Technol 26 (2):99–111

Keirsey D, Bates M (1984) Please understand me: character and temperament Types. Prometheus Nemesis Book Company, Del Mar, CA

Khoshgoftaar TM, Seliya N (2004) Comparative assessment of software quality classification techniques: an empirical case study. Empir Softw Eng 9(3):229–257

Li C, van den Akker J. M., Brinkkemper S, Diepen G (2007) Integrated requirement selection and scheduling for the release planning of a software product. Paper presented at the 13th international working conference on requirements engineering: foundation for software quality, Trondheim, Norway, 11–12 Jun 2007

Luna F, Gonzalez-Alvarez DL, Chicano F, Vega-Rodriquez MA (2011) On the scalability of multi-objective metaheuristics for the software scheduling problem. Paper presented at the 11th international conference on intelligent systems design and applications, Córdoba, Spain, 22–24 Nov 2011

McBride T (2008) The mechanisms of project management of software development. J Syst Softw 81(12):2386–2395

McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng 2(4):308–320

Michael CC, McGraw GE, Schatz MA, Walton CC (1997) Genetic algorithms for dynamic test data generation. Paper presented at the 12th international conference on automated software engineering, Lake Tahoe, 1–5 Nov 1997

Moore JE (1991) Personality characteristics of information systems professionals. Paper presented at the 1991 ACM SIGCPR conference on computer personnel research, Athens, GA, 8–9 Apr 1991

Münsterberg H (1913) Psychology and industrial efficiency. The Riverside Press, Cambridge, USA

Neuman GA, Wagner SH, Christiansen ND (1999) The relationship between work-team personality composition and the job performance of teams. Group Organ Manag 24(1):28–45

Ngo-The A, Ruhe G (2009) Optimized resource allocation for software release planning. IEEE Trans Softw Eng 35(1):109–123

Otero LD, Centeno G, Ruiz-Torres AJ, Otero CE (2009) A systematic approach for resource allocation in software projects. Comput Ind Eng 56(4):1333–1339

Otero CE, Otero LD, Weissberger I, Qureshi A (2010) A multi-criteria decision making approach for resource allocation in software engineering. Paper presented at the 12th international conference on computer modelling and simulation, Cambridge, England, 24–26 Mar 2010

Padberg F (2001) scheduling software projects to minimize the development time and cost with a given staff. In: Anonymous eighth Asia-Pacific software engineering conference (APSEC 2001), Macao, China, 4–7 Dec 2001. IEEE Computer Science Press, Los Alamitos, CA, pp 187–194

Padberg F (2002) Using process simulation to compare scheduling strategies for software projects. Paper presented at the 9th Asia-Pacific software engineering conference, Gold Coast, Australia, 4–6 Dec 2002

Padberg F (2003) A software process scheduling simulator. Paper presented at the 25th international conference on software engineering, Portland, OR, 3–10 May 2003

Padberg F (2004) Computing optimal scheduling policies for software projects. Paper presented at the 11th Asia-Pacific software engineering conference, Busan, Korea, 30 Nov–3 Dec 2004

Padberg F (2006) A study on optimal scheduling for software projects. Softw Process Improv Pract 11(1):77–91

Peeters MAG, van Tuijl HFJM, Rutte CG, Reymen IMMJ (2006) Personality and team performance: a meta-analysis. Eur J Pers 20(5):377–396

Raymond L, Bergeron F (2008) Project management information systems an empirical study of their impact on project managers and project success. Int J Proj Manag 26(2):213–220

Ren J, Harman M, Di Penta M (2011) Cooperative co-evolutionary optimization of software project staff assignments and job scheduling. Paper presented at the 2011 international symposium on search based software engineering, Szeged, Hungary, 10–12 Sept 2011

Rutherfoord RH (2001) Using personality inventories to help form teams for software engineering class projects. ACM SIGCSE Bull 33(3):73–76

Salleh N, Mendes E, Grundy J, St. John Burch G (2010) An empirical study of the effects of conscientiousness in pair programming using the five-factor personality model. Paper presented at the 32nd ACM/IEEE international conference on software engineering, Cape Town, South Africa, 2–8 May 2010

Sfetsos P, Stamelos I, Angelis L, Deligiannis I (2006) Investigating the impact of personality types on communication and collaboration-viability in pair programming – an empirical study. Paper presented at the 7th international conference on extreme programming and agile processes in software engineering, Oulu, Finland, 17–22 Jun 2006

Smith DC (1989) The personality of the systems analysts: an investigation. ACM SIGCPR Comput Pers 12(2):12–14

Stylianou C, Andreou AS (2007) A hybrid software component clustering and retrieval scheme using an entropy-based fuzzy k-modes algorithm. Paper presented at the 19th IEEE international conference on tools with artificial intelligence, Patras, Greece, 29–31 Oct 2007

Stylianou C, Andreou AS (2012) A multi-objective genetic algorithm for software development team staffing based on personality types. Paper presented at the 8th IFIP WG 12.5 international conference on artificial intelligence applications and innovations, Halkidiki, Greece, 27–30 Sept 2012

Stylianou C, Gerasimou S, Andreou, AS (2012) A novel prototype tool for intelligent software project scheduling and staffing enhanced with personality factors. Paper presented at the 24th IEEE international conference on tools with artificial intelligence, Athens, Greece, 7–9 Nov 2012

Tsai H, Moskowitz H, Lee L (2003) Human resource selection for software development projects using Taguchi's parameter design. Eur J Oper Res 151(1):167–180

Tupes EC, Christal RE (1961) Recurrent personality factors based on trait ratings. J Pers 60 (2):225–251

Varona D, Capretz LF, Piñero Y et al (2012) Evolution of software engineers' personality profile. SIGSOFT Softw Eng Notes 37(1):1–5

Wynekoop JL, Walz DB (1998) Revisiting the perennial question: are IS people different? ACM SIGMIS Database 29(3):62–72

Xiao J, Ao X, Tang Y (2013) Solving software project scheduling problems with ant colony optimization. Comput Oper Res 40(1):33–46

Yannibelli V, Amandi A (2011) A knowledge-based evolutionary assistant to software development project scheduling. Expert Syst Appl 38(7):8403–8413

Biography Constantinos Stylianou is a Ph.D. student at the Department of Computer Science of the University of Cyprus. His research focuses on aspects of software project management and specifically on the use of intelligent techniques for human resource scheduling and allocation in addition to human-centric factors in software development. He is also a research member of the Software Engineering and Intelligent Information Systems Research Lab of the Cyprus University of Technology.

Andreas S. Andreou is an Associate Professor and Vice-Chair of the Department of Electrical Engineering/Computer Engineering and Informatics of the Cyprus University of Technology. He is the Director of the Software Engineering and Intelligent Information Systems Research Lab, where his areas of interest include Software Engineering, Web Engineering, Electronic and Mobile Commerce and Intelligent Information Systems.

Chapter 5
Software Project Risk and Opportunity Management

Barry Boehm

Abstract Risk is an uncertain event or condition that has a positive or negative effect on project objectives. Risk management includes the processes of planning, identification, analysis, resource planning, and controlling risk in a project. This chapter focuses on recent insights and approaches within risk management. A positive counterpart to risk management has emerged, called opportunity management. The duality between the two concepts is explained, and the fundamentals of risk–opportunity management are discussed. Furthermore, risk and opportunity management methods, processes, and tools are presented.

5.1 Introduction

Compared to 20–30 years ago, very few projects end up delivering what were initially defined as its requirements. Or if they do, the delivered system has become a poor match to the organization's current needs. Thus, most projects will have to address project uncertainties and risk management right at the start and throughout the project.

Risk management is all about uncertainty and value. Its fundamental quantity to be managed is *risk exposure*, calculated as the project's *probability of loss* (a measure of uncertainty) times its *impact of loss* (a measure of stakeholder value). This is initially obtained by identifying the project's initial sources of uncertainty called *Prob(Loss)* and adding up their contributions to overall risk exposure. Risk exposure then continues to be updated as the project encounters further sources of uncertainty.

B. Boehm (✉)
University of Southern California, Los Angeles, CA, USA
e-mail: barryboehm@gmail.com

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_5, © Springer-Verlag Berlin Heidelberg 2014

The remainder of this chapter elaborates on recent new insights and approaches:

- The duality of risk and opportunity management
- Risk management elements: risk assessment and risk control
- Fundamental risk quantities: risk exposure and risk reduction leverage
- Fundamental risk management strategies: risk understanding, avoidance, transfer, reduction, and acceptance
- Evidence- and value-based milestone decision processes (shortfalls in evidence are uncertainties and risk probabilities; stakeholder value propositions are determinants of risk impact)
- Risk-based "how much is enough" analyses (balancing the risk of doing too little and the risk of doing too much)

Also, it is important to recognize that not all uncertainty is bad. A positive counterpart to risk management has emerged, called opportunity management, which enables projects to balance risk and opportunity prospects.

5.2 The Duality of Risks and Opportunities

The generally accepted quantity for reasoning about risk is *risk exposure* (RE) defined as

$$\mathrm{RE} = \mathrm{Prob(Loss)}^{*}\mathrm{Impact(Loss)}$$

Opportunity can be considered as a dual to risk as a decision not to pursue an opportunity has a negative expected value equal to the probability of success for the opportunity times its impact of gain if it succeeds. This could be called its "missed opportunity" risk exposure, but it is more positive to call it its *opportunity exposure* (OE) defined as

$$\mathrm{OE} = \mathrm{Probability\ (Gain)}^{*}\mathrm{Impact(Gain)}$$

Charette (1999) defines such a holistic view as risk *entrepreneurship*. Risk entrepreneurs are not extreme thrill seekers; they are careful, judicial, and value-aware decision makers. They understand the possible costs of risks as well as the return from the opportunities. However, where the business value is sufficiently large and the risk is both understandable and within the bounds of the organization's capability to manage it if something goes awry, they have the ability and the courage to embrace the risk.

Another form of opportunity management that emphasizes the duality with risk management involves identifying, evaluating, and deciding on opportunities to reduce the project's risk exposure. This is discussed below with respect to the concept of *risk reduction leverage*.

It should be noted that in managing opportunities as well as risks, negative risks generally have a much stronger impact than pure opportunities (those not paired with a risk). The INCOSE/IEEE System Engineering Body of Knowledge, SEBoK (Pyster et al. 2013) states that

> In principle, opportunity management is the duality to risk management (. . .). Thus, both should be addressed in risk management planning and execution. In practice, however, a positive opportunity exposure will not match a negative risk exposure in utility space, since the positive utility magnitude of improving an expected outcome is considerably less than the negative utility magnitude of failing to meet an expected outcome.
>
> Canada (1971), Kahneman and Tversky (1979).

In other words, the utility function usually applied to project and program managers is strongly weighted toward the negative; that is, making a $1 million loss has about the same negative utility for a project manager as has the positive utility of making a $5 million gain. Edmund Conrow's comprehensive book, Effective Risk Management (Conrow 2003) has some further cautions about overenthusiastic opportunity management.

5.3 Fundamentals of Risk–Opportunity Management

This section provides a summary of the fundamental risk–opportunity management activities (risk assessment and risk control); the two primary decision criteria associated with risk management (risk exposure and risk reduction leverage); and the five fundamental risk management strategies (risk understanding, risk avoidance, risk transfer, risk reduction, and risk acceptance). We use risk as the overarching term, but we always consider the risk–opportunity dualism in the discussion.

5.3.1 Risk Assessment: Identification, Analysis, and Prioritization

Risk Identification produces lists of potential project-specific risks or opportunities likely to change a project's outcome. Typical identification techniques include checklists, decomposition, comparison with experience, conflicts among the success-critical stakeholders' value propositions, and examination of decision drivers. Another key contribution to risk identification involves evidence-based decision criteria, in that shortfalls in evidence of project and product feasibility are uncertainties or probabilities of loss if the project proceeds with inadequate evidence of feasibility. This Prob(Loss) when multiplied by the Impact(Loss) becomes Risk Exposure, as above.

Risk Analysis produces assessments of the gain/loss-probability and gain/loss-impact associated with each of the identified risks and assessments of the compounding that may occur if risks interact to change probabilities or impacts when more than one happens to occur. Typical techniques include network analysis, decision trees, cost models, performance models, and statistical decision analysis.

Even for medium-size (20–30 people) projects, *Risk Identification* can identify up to 100 candidate risks to address. However, about 10–15 project-wide risks are about the most that a medium-size, medium-complexity project should take on simultaneously. The overall objective of risk analysis is therefore to determine the relative risk exposure of each identified risk, initially just accurately enough to separate the "big rocks" from the "pebbles," and subsequently to provide more insight on how best to reduce the big-rock probabilities and impacts of loss. If these quantities cannot be accurately determined, the project has two main choices. One is to assign relative values for Prob(Loss) and Impact(Loss), generally on a scale of 0–10. The other is to buy information to reduce risk, via stakeholder interviews, surveys, prototypes, or high-level models.

Risk Prioritization produces a prioritized ordering of the risks to be subjected to budgeted risk mitigation and control, and which are to be monitored for opportunistic risk mitigation or growth to become a serious risk requiring budgeted risk mitigation. The emphasis on risk mitigation budgets reflects the reality that projects have limited budgets, and that the relative cost of risk mitigation is as important as the relative risk or opportunity exposure. This brings into play the second fundamental risk quantity called *risk reduction leverage* (RRL). RRL is defined as the ratio of risk exposure reduction to the cost of achieving the reduction for a given risk reduction alternative.

Thus, having a high risk exposure may not be a good criterion for choosing risks to be mitigated. If an emerging technology is so immature that a very large budget or long schedule would be required to reduce its risks of performance, reliability, scalability, or maintainability, it would be best to defer consideration of its use until it reached a higher level of maturity. Typical risk prioritization techniques include RRL analysis, particularly involving cost-benefit analysis, and Delphi or group-consensus techniques.

5.3.2 Risk Control: Risk Mitigation Planning, Risk Mitigation, Risk Monitoring and Corrective Action

Risk Mitigation Planning produces plans for addressing each risk (e.g., via risk understanding, avoidance, transfer, reduction, or acceptance), including the coordination of the individual plans with each other and with the overall project plan. As above under Risk Prioritization, it is important to address situations where a risk mitigation threatens a project's critical path, a particularly frequent occurrence with highly interactive risks that are pushing the technology frontiers in several directions at once. In such cases, it is often best to defer some of the capabilities, or if several are needed for success, to reflect this in the project's schedule. A good example was the CCPDS-R project described in Royce (1998), which deferred its Preliminary Design Review from a traditional Month 6 to Month 14, by which time all of the risks had been mitigated.

Typical techniques include checklists of risk mitigation techniques, cost–benefit analysis, and standard risk management plan outlines, forms, and elements. In many cases, though, the standard forms may be overkills; a lean alternative is provided below.

Risk Mitigation produces a situation in which the risks are eliminated or otherwise resolved (e.g., risk avoidance via relaxation of requirements; opportunity acceptance via strategic partnership agreements). Typical techniques include prototypes, simulations, benchmarks, mission analyses, key-personnel agreements, design-to- cost approaches, and incremental development for risk avoidance. Opportunity assurance analysis may use technology trend analyses, multi-corporate analyses, prototypes, and agreements, and acquisition analyses.

Risk monitoring and corrective Action involves tracking the project's progress towards resolving its risk exposure and taking corrective action where appropriate. Typical techniques include risk management plan milestone tracking and a top ten risk event list whose progress is reviewed and acknowledged at each weekly, monthly, or milestone project review.

A framework and checklist of goals, critical success factors, and questions for feasibility evidence assessment and associated risk monitoring at decision points is provided in (Boehm et al. 2009). A complementary continuous risk monitoring framework is the INCOSE System Engineering Leading Indicators Guide (INCOSE 2012).

5.3.3 The Fundamental Risk Quantities: Risk Exposure and Risk Reduction Leverage

Having defined risk exposure (RE) and opportunity exposure (OE) in terms of "loss," to be able to use these in project situations, we need a definition of "loss." Given that all projects involve several classes of stakeholders (customer, developer, user, maintainer, and often others such as suppliers, distributors, interoperators, regulators, and venture capitalists), each with somewhat different but highly important value propositions or satisfaction criteria, we can see that "loss" is multidimensional. For customers and developers, budget overruns and schedule slips can result in losses of value. For users, products with the wrong functionality, user interface shortfalls, performance shortfalls, or reliability shortfalls can result in losses of value. For maintainers, poor quality and bad design can result in losses of value.

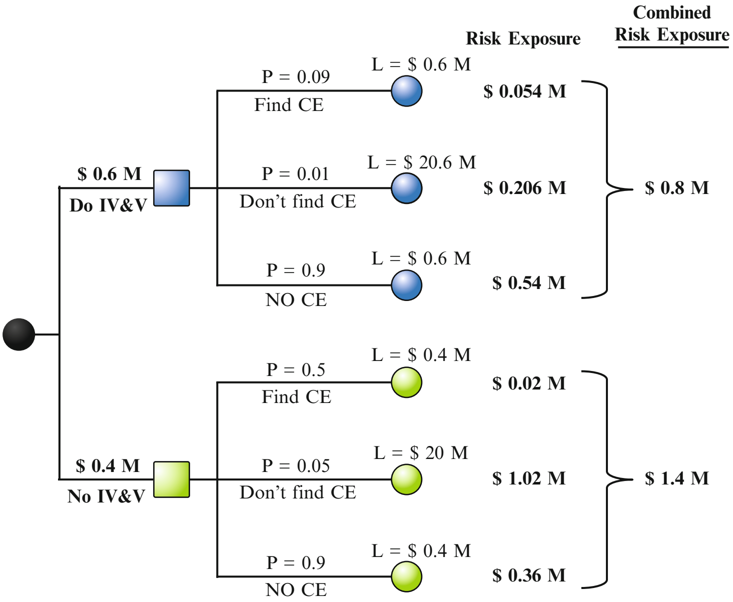


Fig. 5.1 Decision tree for spacecraft experiment IV&V

One way to look at risk exposure in context is to construct a decision tree similar to that shown in Fig. 5.1. It illustrates a potentially risky situation involving the software controlling a spacecraft experiment. The software has been under development by an experiment team that understands their experiment well, but is inexperienced and somewhat casual about software development. As a result, the satellite platform manager has obtained an estimate that there is a probability Prob (Loss) of 0.1 that the experimenters' software will have a critical error: one which will wipe out the entire experiment and cause an associated loss Size(loss) of the total $20 million investment.

The satellite platform manager identifies two major options for reducing the risk of losing the experiment:

- Convincing and helping the experiment team to apply better software development methods. This incurs an additional analysis and test cost of $400 K, and from previous experience the manager estimates that this will reduce the failure probability Prob (Loss) to 0.05
- Hiring a contractor to independently verify and validate (IV&V) the software. This costs an additional $300 K, plus another $300 K to fix the defects found by IV&V. Based on the results of similar IV&V efforts, the manager estimates that this will reduce the error probability Prob(Loss) to 0.01

The decision tree in Fig. 5.1 then shows, for each of the two major decision options, the possible outcomes, their probabilities, the losses associated with each outcome, the risk exposure associated with each outcome, and the total risk exposure (or expected loss) associated with each decision option. In this case, the total risk exposure associated with the experiment-team option is \$1.4 M. For the strongest IV&V option, the total risk exposure is only \$0.8 M; thus it represents the more attractive option.

Besides providing individual solutions for risk management situations, the decision tree also provides a framework for analyzing the sensitivity of preferred solutions to the risk exposure parameters. Thus, for example, the experiment-team option would be preferred if the loss due to a critical software error were less than \$5 M, if the experiment team could reduce their critical-software-error probability to less than 0.02, if the IV&V team option cost more than \$1,200 K, or if there were various partial combinations of these possibilities. However, even with this sort of sensitivity analysis, there may not be enough information available to quantify the risk exposure parameters well enough to perform a precise analysis.

A similar approach can be accomplished to evaluate the benefit of a pure opportunity or to analyze the risk-reward balance of compound opportunities and risks.

The *second fundamental risk quantity* is the leverage available based on mitigation or assurance costs. Risk reduction leverage RRL is defined as follows:

$$\mathrm{RRL} = (\mathrm{RE}_{\mathrm{before}} - \mathrm{RE}_{\mathrm{after}})/\text{risk reduction cost}$$

With an $\mathrm{RE}_{\mathrm{before}}$ of 0.1 * \$20M = \$2M, the corresponding RRL values for the "No IV&V" and "Do IV&V" options are:

$$\mathrm{RRL}\ (\text{No IV\&V}) = (2\mathrm{M} - 1\mathrm{M})/400\mathrm{K} = 2.5;\ \text{and}$$

$$\mathrm{RRL}\ (\text{Do IV\&V}) = (2\mathrm{M} - 0.2\mathrm{M})/600\mathrm{K} = 3.$$

Even stronger outcomes could be achieved via value-based prioritization of IV&V activities (Li et al. 2010), as frequently 20 % of the defects impact 80 % of the business value (Bullock 2000).

5.3.4 The Fundamental Risk Mitigation Strategies

There are five fundamental strategies that can be used to mitigate a risk: Risk understanding via buying information, risk avoidance, risk transfer, risk reduction and Risk Acceptance. Like the decision tree, these are equally useful in investigating opportunities:

- ***Risk Understanding.*** Sometimes, the best way of mitigating a risk is to gain more insight into the problem. Buying information via prototyping to learn more about requirements or COTS product interoperability can reduce or eliminate risks.

- ***Risk Avoidance.*** Avoiding a risk means taking actions that remove the risk from the critical path or the project. For example, negotiating with the customer to reduce a risky performance goal can effectively avoid the risk.
- ***Risk Transfer.*** Transferring risk involve an action that moves the risk from one party to another, or shares the risk exposure between several parties such that no single party is overly burdened.
- ***Risk Reduction.*** Actions can be taken that reduce the risk exposure by lowering the probability or the magnitude of loss.
- ***Risk Acceptance.*** This is a decision that the risk exposure is low enough that the project can succeed even if the risk occurs, or that the opportunity exposure for success more than compensates for the risk exposure for failure. Accepting the risks still implies that they need to be managed.

Here is an example of how each of these were used for a COTS selection options analysis originally described in (Boehm and Bhuta 2008). Suppose that at the beginning of a project, there is an opportunity to choose either a higher-performance COTS product B or a comparable but lower-performance COTS product C. Without further evidence about the relative merits of COTS products B and C, the project would choose B. However, in this case, the project then finds out during integration that B has serious architectural mismatches with another project-essential COTS product A. This will cause the project to overrun by 3 months and $300 K.

At this point, the probability of this is 1.0, so it is not a risk but a problem. But earlier, its nature was uncertain and should have been identified as a risk item during an early milestone review by the experts reviewing the evidence that the selected COTS products would successfully interoperate. It would then need to be covered by a risk management plan in order to meet the review success criteria. A risk management plan can use one or more of the main risk mitigation strategies, In general, the best one to try first is Risk Understanding via buying information, which will provide more insight on which of the other strategies to employ.

Risk Understanding. The project decides to spend $30 K prototyping the integration of COTS packages B and C with COTS package A. It finds the architectural mismatches between A and B, and the likely resulting costs and schedules needed to resolve them, and also finds that COTS package C would integrate easily with A, but with a 10 % performance loss. This information enables the stakeholders to better evaluate the other risk mitigation strategies.

Risk Avoidance. This option would be best if the customer agrees that the reduction in performance is preferable to the prospect of late delivery, and agrees to go with COTS product C rather than B.

Risk Transfer. However, if the customer decides that the increase in performance is worth the extra time and money, the customer should establish a risk reserve of 3 months and $300 K to be used to the extent that it will be needed during integration, but with award fees for the developer if less than the full risk reserve will be needed.

Risk Reduction. In order to eliminate the schedule risk, the developer and customer agree to perform a parallel integration of A and B early in the project with the added cost but with no delay in delivery schedule.

Risk Acceptance. The developer decides that having a proprietary solution to integrating A and B will provide them with a competitive edge on future projects, and decides to fund and patent the solution, while giving the customer a royalty-free license to use it. From a risk/opportunity standpoint, the business case for having a proprietary solution to integrating A and B would have given the developer a sufficiently high payoff to generate a positive Opportunity Exposure for taking this option.

Usually, some combination of the risk mitigation strategies will prove mutually acceptable to all of the stakeholders. However, there will be some situations in which there are irreconcilable differences between the stakeholders that leave the project with no feasible options. In this case, it is best to have found this out early rather than at the end of the project, and to discontinue the project with no further expenditure of stakeholders' resources, or to redefine the project scope in a way that is mutually satisfactory to the stakeholders.

5.4 Risk and Opportunity Management Methods, Processes, and Tools

This section elaborates on several particularly valuable risk and opportunity management methods, processes, and tools mentioned above. They are evidence- and risk-based decision reviews, lean risk management plans, top-10 risk tracking, and risk-balanced activity levels.

5.4.1 Evidence- and Risk-Based Decision Reviews

A major recent step forward in the management of outsourced projects has been to move from schedule-based reviews to event-based reviews. A schedule-based review says basically that, "The contract specifies that the Preliminary Design Review (PDR) will be held on April 1, 2014, whether we have a design or not."

In general, neither the customer nor the developer wants to fail the PDR, so the project goes into development with the blessing of having passed a PDR, but with numerous undefined interfaces and unresolved risks. Such shortfalls are the primary source of large amounts of late rework, generally resulting in major project overruns and incomplete deliveries.

An event-based review says, "Once we have a preliminary design, we will hold the PDR." Such a review will generally consist of exhaustive presentations of sunny-day briefing charts and UML diagrams. But in general, it will still have numerous unidentified shortfalls that will cause extensive project rework, overruns, and incomplete deliveries. These shortfalls are uncertainties and probabilities of loss, which when multiplied by size of loss become Risk Exposure.

Evidence- and risk-based decision reviews use evidence criteria embodied in a *Feasibility Evidence Description* (FED). It includes *evidence* provided by the developer and validated by independent experts that, if the system is built to the specified architecture it will

1. Satisfy the specified operational concept and requirements, including capability, interfaces, level of service, and evolution
2. Be buildable within the budgets and schedules in the plan
3. Generate a viable return on investment
4. Generate satisfactory outcomes for all of the success-critical stakeholders
5. Identify shortfalls in evidence as risks, and cover them with risk mitigation plans

A FED does not assess a single sequentially developed system definition element (operational concept, requirements specification, architecture, development plan, integration and test plan, etc.), but the consistency, compatibility, and feasibility of several concurrently engineered elements. To make this concurrency work, a set of milestone decision reviews are performed to ensure that the many concurrent activities are synchronized, stabilized, and risk-assessed at the end of each phase. Each of these reviews is focused on developer-produced and expert-validated evidence, documented in the FED (or preferably to provide pointers to the results of feasibility analyses, to avoid excessive documentation), to help the system's success-critical stakeholders determine whether to proceed into the next level of commitment. Hence, they are called Commitment Reviews.

The FED is based on evidence from simulations, models, or experiments with planned technologies and increasingly detailed analysis of development approaches and project productivity rates. The parameters used in the analyses should be based on measured component performance or on historical data showing relevant past performance, cost estimation accuracy, and actual developer productivity rates. A Data Item Description for contractual purposes is defined in Boehm et al. (2013), based on earlier contributions such as (AT&T 1993; Boehm 1996; Royce 1998; Kruchten 1999; Maranzano et al. 2005).

A shortfall in feasibility evidence indicates a level of program execution uncertainty or probability of loss, and a source of program risk. It is often not possible to fully resolve all risks at a given point in the development cycle, but known, unresolved risks need to be identified and covered by risk management plans, including the necessary staffing and funding to address them. The nature of the evidence shortfalls, the strength and affordability of the risk management plans, and the stakeholders' degrees of risk acceptance or avoidance will determine their willingness to commit the necessary resources to proceed. A program with risks is not necessarily bad, particularly if it has strong risk management plans. A program with no risks may be high on achievability, but low on ability to produce a timely payoff or competitive advantage.

1.	**Objectives**	**(the “Why”)**
2.	**Deliverables and Milestones**	**(the “What” and “When”)**
3.	**Responsibilities**	**(the “Who” and “Where”)**
4.	**Approach**	**(the “How”)**
5.	**Resources**	**(the “How Much”)**

Fig. 5.2 Lean risk management plan outline

5.4.2 Lean Risk Management Plans

The word “Plan” often conjures up a heavyweight document full of boilerplate and hard-to-remember sections. Risk management plans should be particularly lean and risk-driven as illustrated in the outline presented in Fig. 5.2 (a generally useful outline for other plans as well).

Figure 5.3 (Boehm 1991) provides an example of how the outline might be used to develop a risk management plan to conduct fault tolerance prototyping to mitigate identified risks that the fault tolerance features might seriously compromise performance aspects such as throughput, real-time deadline satisfaction, and power consumption. It also shows the examples of the types of information to be provided for responsibilities, the risk management approach, and the needed funding resources.

5.5 Top-10 Risk Item Tracking

As discussed in Sect. 5.3.1, even medium-sized projects can identify up to 100 potential risks, but on the order of 10 is best for top-management focus. The number 10 is not magical; it could be 7 or 12. The others can be put on a watch list, and promoted to the top-10 if they become serious.

An example format for the top-10 risk item list is shown in Fig. 5.4. Each risk item should identify the risk, and note its current and previous rank along with the number of months (or other periodic project review frequency) it has been on the list. The top-10 list should be the first item discussed in the monthly project review, as it is the best vehicle for the project manager to indicate to her or his superiors doing the review where the project will need their help. If this is done at the end of the review, time will often run out before the needed help can be discussed and acted on.

1. Objectives

Determine and reduce the level of risk of the fault tolerance features causing unacceptable performance (e.g., throughput, response time, power consumption). Create a description of and a development plan for a set of low-risk fault tolerance features.

2. Deliverables and Milestones

By week 3:
Evaluation of fault tolerance options, assessment of reusable components, draft workload characterization, evaluation plan for prototype exercise, and description of prototype.
By week 7:
Operational prototype with key fault tolerance features, workload simulation, instrumentation and data reduction capabilities. Draft description, plan for fault tolerance features.
By week 10:
Evaluation and iteration of prototype, revised description and plan for fault tolerance features.

3. Responsibilities

- System Engineer: G. Smith assigned to Tasks 1, 3, 4, 9, 11 and to support of Tasks 5, 10
- Lead Programmer: C. Lee assigned to Tasks 5, 6, 7, 10 and to support of Tasks 1, 3
- Programmer: J. Wilson assigned to Tasks 2, 8 and to support of Tasks 5, 6, 7, 10

4. Approach

Design-to-Schedule prototyping effort, driven by hypotheses about fault tolerance-performance effects, using multicore processor, real-time OS, add prototype fault tolerance features.

Evaluate performance with respect to representative workload.
Refine Prototype based on results observed.

5. Resources

$60K	Full-time system engineer, lead programmer, programmer (= 10 weeks)*(3 staff)*($2K/staff-week)
$0K	3 Dedicated workstations (from project pool)
$0K	2 Target processors (from project pool)
$0K	1 Test co-processor (from project pool)
$10K	Contingencies
$70K	**Total**

Fig. 5.3 Sample lean risk management plan

	Mo. Ranking			
Risk Item	This	Last	# Mo.	Risk Resolution Progress
Replacing sensor-control software developer	1	4	2	Top replacement candidate unavailable
Target hardware delivery delays	2	5	2	Procurement procedural delays
Sensor data formats undefined	3	3	3	Action items to software, sensor team; Due next month
Staffing of design V&V team	4	2	3	Key reviewers committed; Need fault-tolerance reviewer
Softare fault-tolerance may compromise performance	5	1	3	Fault tolerance prototype successful
Accommodate changes in data bus design	6	-	1	Meeting scheduled with data bus designers
Testbed interface definitions	7	8	3	Some delays in action items; Reviewer meeting scheduled
User interface uncertainties	8	6	3	User interface prototype successful
TBDs in experiment operational concept	-	7	3	TBDs resolved
Uncertainties in reusable monitoring software	-	9	3	Required changes small, successfully made

Fig. 5.4 Top-10 risk items list assuming monthly risk reassessment (Boehm 1991)

5.6 Risk-Balanced Activity Levels

How much evidence generation is enough? The amount of evidence for a commitment decision should be proportional to the amount of risk involved in making the decision. This may seem like circular reasoning, and it is in a way. But it can be approached incrementally, which is the process used in the Incremental Commitment Spiral Model (Boehm et al. 2014).

The risk-based decision heuristic of balancing the risks of doing too little evidence generation with the risks of doing too much can be applied to most decisions involved in systems and software definition, development, and evolution. How much system scoping, planning, prototyping, COTS evaluation, requirements detail, spare capacity, fault tolerance, safety, security, environmental protection, documenting, configuration management, quality assurance, peer reviewing, testing, use of formal methods, and feasibility evidence is enough? The best answer can generally be found by considering and balancing the risks of doing too little with the risks of doing too much. And the answer will generally not be the same for all parts of the system. The higher-risk parts of the system will need more attention to detail than the lower-risk parts, in order to reduce both the probability and size of loss involved in getting it wrong.

This *Meta-Principle of Risk Balancing: Balancing the risk of doing too little and the risk of doing too much of anything will generally find a middle course sweet spot that is about the best you can do*, is arguably the strongest contribution of risk management to software project management.

Of course, there is nothing new about it. It is what Herb Simon was talking about in preferring satisficing to optimizing; what Aristotle was talking about with the golden mean; what the Confucians talk about with the doctrine of the mean, and what the Buddhists talk about with the middle way. With all of these advocates, it seems like a pretty good underlying meta-principle.

5.7 Summary and Conclusions

The fundamental quantity to be managed in risk or opportunity management is risk or opportunity exposure. It is the product of the probability that a risk or opportunity item will be realized (a measure of uncertainty) times the loss or gain impact of the risk or opportunity item on the resulting system's effectiveness (a measure of value). Further aspects of uncertainty are addressed in Chaps. 2 and 3 on Cost and Schedule Estimation and Chap. 11 on Agile Project Management.

Further aspects of value are addressed in Chap. 2 on defining success, Chap. 6 on Quality Management, and Chap. 8 on Portfolio Management.

As uncertainty increases with changes in technology, competition, the environment, and social systems, and impact increases as software becomes increasingly success-critical for systems (Neumann 1985), risk and opportunity exposure become increasingly critical to software project management. Future technologies likely to change and improve software risk and opportunity management are search-based techniques as addressed in Chap. 15 and social media collaboration support as addressed in Chap. 16.

Several techniques are presented on the components of risk assessment (risk identification, analysis, and prioritization) and risk control (risk mitigation planning, execution, monitoring, and corrective action); options for risk mitigation (risk understanding, avoidance, transfer, reduction, and acceptance); risk management methods, processes, and tools (evidence-based decision reviews, decision trees, lean risk mitigation plans, and top-10 risk item tracking). Arguably, the strongest contribution that risk considerations can provide to software project management is the *meta-principle of risk balancing: balancing the risk of doing too little and the risk of doing too much of anything will generally find a middle course sweet spot that is about the best you can do*.

References

AT&T (1993) Best current practices: software architecture validation. AT&T, Murray Hill, NJ
Boehm B (1991) Software risk management: principles and practices. IEEE Softw 8(1):32–41
Boehm B (1996) Anchoring the software process. IEEE Softw 1996:73–82

Boehm B, Bhuta J (2008) Balancing opportunities and risks in component-based software development. IEEE Softw 15(6):56–63

Boehm B, Ingold D, Dangle K, Turner R, Componation P (2009) Early identification of SE-related program risks. SERC Technical Report RT-EM

Boehm B, Lane J, Koolmanojwong S, Turner R (2013) An evidence-based systems engineering (SE) data item description. In: Proceedings, CSER 2013, Elsevier

Boehm B, Lane J, Koolmanojwong S, Turner R (2014) The incremental commitment spiral model: principles for creating successful systems and software. Addison-Wesley, Reading, MA

Bullock J (2000) Calculating the value of testing. Softw Testing Qual Eng (May/June):56–62

Canada R (1971) Intermediate economic analysis for management and engineering. Prentice-Hall, Englewood Cliffs, NJ

Charette R (1999) The competitive edge of risk entrepreneurs. IEEE IT Prof 1:67–71

Conrow E (2003) Effective risk management: some keys to success, 2nd edn. AIAA, Reston, VA

INCOSE (2012) System engineering leading indicators guide, version 2.0. INCOSE-TP-2005-001-03

Kahneman D, Tversky A (1979) Prospect theory: an analysis of decision under risk. Econometrica 47(2):263–292

Kruchten P (1999) The rational unified process. Addison-Wesley, Reading, MA

Li Q, Shu F, Boehm B, Wang Q (2010) Improving the ROI of software quality assurance activities: an empirical study. In: Proceedings of international conference on software process (ICSP 2010), Paderborn, Germany, Jul 2010, pp 357–368

Maranzano JF, Rozsypal SA, Zimmermann GH, Warnken GW, Wirth PE, Weiss DM (2005) Architecture reviews: practice and experience. IEEE Softw 22:34–43

Neumann P (1985) Risks to the public in computers and related systems. Softw Eng Notes. ACM SIGSOFT, 1985-continuing

Pyster A et al (2013) Guide to the systems engineering body of knowledge, risk management. http://sebokwiki.org/1.1.1/index.php?title=Risk_Management. Retrieved 16, 2013

Royce WE (1998) Software project management. Addison-Wesley, Reading, MA

Biography Barry Boehm is the USC Distinguished Professor of Computer Sciences, Industrial and Systems Engineering, and Astronautics, and the TRW Professor of Software Engineering. He is also the Chief Scientist of the DoD-Stevens-USC Systems Engineering Research Center and the founding Director of the USC Center for Systems and Software Engineering. His contributions include the COCOMO family of cost models and the Spiral family of process models.

Part II
Supporting Areas

Introduction

Software project management is conducted in a context where project management actions will vary to reflect differences in this context. This includes areas such as quality management systems, knowledge management, product management, global software development and motivation of software engineers. In this part of the book, we have invited some of the leading experts on the areas listed to present some of the latest experiences and results in relation to these five areas. The authors share their knowledge, insights and accompanying recommendations and conclusions in five chapters in this part of the book.

In Chap. 6, Jens Heidrich, Dieter Rombach and Michael Kläs highlight the need to manage software quality as an important part of software project management. The authors define and discuss the usage of software quality models in a project management context. They stress some challenges in relation to quality management and present solutions in relation to the challenges. The chapter starts with a discussion about the need to select a suitable quality model and emphasizes that one model may not fit in all different contexts. The authors provide an overview of quality models, and continue with a discussion about tailoring quality models for different contexts. The chapter ends with a discussion about the strategic usage of quality models.

The authors of Chap. 7 argue that project management can benefit from integrated project and system knowledge management. Barbara Paech, Alexander Delater and Tom-Michael Hesse start by presenting their vision of integrating system and project knowledge. They describe the benefits of this integrated knowledge by mapping it to the knowledge areas in the Project Management Book of Knowledge. Based on a systematic literature review, the authors discuss the capture and use of decisions and work items in relation to software artefacts. They highlight both the availability of tools and empirical evidence in relation to the links. The authors continue by describing how decisions ought to be handled as explicitly as work items. They argue that all too often decisions and rationales for decisions are insufficiently documented, although being important bearers of knowledge. The chapter concludes with a discussion on further research in relation to the integration of system and project knowledge.

In Chap. 8, Erik Jagroep, Inge van de Weerd, Sjaak Brinkkemper and Ton Dobbe introduce a framework for product portfolio management. The authors take the standpoint that a project contributes to a company's products, and hence management of software projects must be well aligned with the strategic goals of different software products. The authors use a design science approach based on a literature review and interviews to develop the framework. It is evaluated and improved through the application of the framework in industry in several steps. The chapter describes the resulting framework for product portfolio management and puts it into the context of software project management. The chapter also includes a discussion on a maturity matrix in relation to product portfolio management. Finally, the implications of the product portfolio management framework are discussed.

Christof Ebert discusses challenges and opportunities in relation to managing global software projects in Chap. 9. The chapter starts by presenting the main reasons for global software development, namely, efficiency, presence, talent and flexibility. Ebert continues with a discussion regarding the perceived challenges and benefits of global software development. The author motivates the need for matching process maturity between client and supplier based on empirical research. The chapter also includes a discussion on how global software development is impacted by the actual work organization. The author continues by discussing risks in relation to global software development. The chapter concludes with a discussion regarding trends in relation to globalization.

In Chap. 10, Sarah Beecham discusses motivational issues, in particular in relation to managing globally distributed software projects and hence for software engineers working in virtual teams. The author starts by introducing some of the theories in relation to motivational aspects in general and then continues to discuss the specifics in relation to software engineers and their characteristics. After this, Beecham presents a case study on the motivation of software engineers in a global software development context. Based on the case study, the author moves on to discuss how recommended global software development practices affect motivational issues. The chapter concludes with a mapping of software engineer characteristics and how compatible they are to working in a virtual team in a global context.

The five chapters in this part give an in-depth insight into some areas that provide a context for software project management. A project manager must be able to navigate and work in these areas to conduct their often-challenging tasks. The chapters are intended to support project managers in their daily tasks by offering practical recommendations based on grounded research and experience.

Chapter 6
Model-Based Quality Management of Software Development Projects

Jens Heidrich, Dieter Rombach, and Michael Kläs

Abstract Managing the quality of artifacts created during the development process is an integral part of software project management. Software quality models capture the knowledge and experience regarding the quality characteristics of interest, the measurement data that can help to reason about them, and the mechanisms to use for characterizing and assessing software quality. They are the foundation for managing software quality in projects in an evidence-based manner. Nowadays, coming up with suitable quality models for an organization is still a challenging endeavor. This chapter deals with the definition and usage of software quality models for managing software development projects and discusses different challenges and solutions in this area. The challenges are: (1) There is no universal model that can be applied in every environment because quality is heavily dependent on the application context. In practice and research, a variety of different quality models exists. Finding the "right" model requires a clear picture of the goals that should be obtained from using the model. (2) Quality models need to be tailored to company specifics and supported by corresponding tools. Existing standards (such as the ISO/IEC 25000 series) are often too generic and hard to fully implement in an organization. (3) Practitioners require a comprehensive set of techniques, methods, and tools for systematically specifying, adapting, and applying quality models in practice. (4) In order to create sustainable quality models, their contribution to the organizational goals must be clarified, and the models need to be integrated into the development and decision-making processes.

J. Heidrich (✉) • M. Kläs
Fraunhofer IESE, Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
e-mail: jens.heidrich@iese.fraunhofer.de; Michael.Klaes@iese.fraunhofer.de

D. Rombach
Technische Universität Kaiserslautern, Kaiserslautern, Germany

Fraunhofer Institute for Experimental Software Engineering (IESE), Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
e-mail: rombach@informatik.uni-kl.de; dieter.rombach@iese.fraunhofer.de

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_6, © Springer-Verlag Berlin Heidelberg 2014

6.1 Introduction

When it comes to managing and controlling software development projects, three dimensions are typically addressed (assuming a defined functionality and project scope): cost, time, and quality. A variety of measurement-based approaches have been developed in recent years for managing and controlling projects in terms of these three aspects. However, while well-established measures exist to quantify cost and schedule aspects (such as those described in Chaps. 3 and 4), quality is a less tangible concept. As a consequence, it is still a challenge today to objectively measure software quality in early stages of the development process. Yet, being able to manage and control software quality is an integral part of professional project management (PMI 2008).

The difficulties become obvious when we take the definition of quality as being the "totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs" (ISO 8402 1995). On the one hand, software systems are becoming ever more complex (e.g., in terms of size, algorithms, and interfaces) and heterogeneous (e.g., in terms of development platforms and languages). This makes it more difficult to systematically analyze their quality. On the other hand, quality is heavily dependent on the application domain, the stakeholders, the usage context, and the specific project environment. For example,

- **Application domain**: Safety-critical systems (such as automobiles or power plants) require far different quality characteristics than information systems (such as accounting software or web applications)
- **Stakeholders**: A top-level manager of an organization probably has a different understanding of the term "software quality" than technical software development personnel
- **Usage context**: Different mechanisms need to be considered when trying to predict the final software quality at early stages of the development process compared to measuring the actual quality of a software design or component
- **Project context**: Having contractor-subcontractor relationships in a project requires a detailed understanding of what quality of externally delivered artifacts is actually needed compared to doing in-house development only

Depending on these aspects, different characteristics are important, and different techniques, methods, and tools have to be used as part of the software development process to guarantee a certain level of quality.

Software quality models are a means for defining and operationalizing the term "software quality." The typical approach is to refine the abstract term into more concrete subconcepts down to a level of detail where concrete metrics and indicators can be assigned. Depending on the structure of such a model, mechanisms for evaluating/assessing software quality are included. They support the stakeholders of a quality model in interpreting the measurement data (e.g., by defining thresholds distinguishing between acceptable and not acceptable quality) and in aggregating the results in order to allow them to come up with an evaluation/assessment of the different quality characteristics of interest.

One of the most popular and extensive software quality models is published in the ISO/IEC standard 9126-1 (2001) and the corresponding standard for software product evaluation ISO/IEC 14598-1 (1999). It distinguishes three different views on product quality:

- *Internal Quality* describes the characteristics of intermediate artifacts of the development process required to satisfy internal quality requirements.
- *External Quality* describes externally visible quality characteristics of artifacts of the development process required to satisfy external quality requirements. They typically specify the quality of the product delivered to the customer.
- *Quality in Use* defines quality characteristics from the perspective of a user, which are of importance when the software product that was delivered to the customer is actually used.

The assumption underlying the definition of these views is that having high-quality processes helps to obtain artifacts with good internal quality; creating artifacts with high internal quality supports achieving a final product with good external quality; and a product of high external quality leads to good user experience with the product, that is, good quality in use (ISO/IEC 9126-1 2001).

For all three views on software quality, the standard provides models defining a breakdown structure of quality into subconcepts. For example, internal and external quality is broken down into functionality, reliability, usability, efficiency, maintainability, and portability. Each of these high-level quality characteristics is broken down into corresponding subcharacteristics. The models for internal and external quality essentially comprise the same quality characteristics, but use different metrics for actually measuring product quality in the end.

Recently, the successor to these two standards has been published, the ISO/IEC 25000 "Software Product Quality Requirements and Evaluation (SQuaRE)" (ISO/IEC 25000-1 2005), which comprises a whole series of standards related to quality management addressing models for product quality (external and internal quality) and quality in use (2501n series), measuring product quality (2502n series), and evaluating/assessing product quality (2504n series). Moreover, standards have been added that address how to define and systematically derive quality requirements, that is, nonfunctional requirements of software products (2503n series).

The ISO/IEC standards specify a set of quality models that try to be generic enough to be applicable for all kinds of software systems. Indeed, models proposed to be universal, such as those described in the ISO/IEC standards, stay on a high-level of abstraction, making it difficult to instantiate and apply them. In literature, a variety of more specific software product quality models can be found for different application domains, stakeholders, usage purposes, and project environments (Kläs et al. 2009). However, in practice, it is quite difficult to sustainably apply these models for managing the quality of the outcome of a software development project (Wagner et al. 2010b) (overview):

- **Challenge 1**: There is no universal quality model that can be applied in every environment. In practice and research, a variety of different quality models exists. Finding the "right" model requires a clear picture of the goals that should be obtained from using the model in a software development project. Goals may range from getting a common understanding of important quality characteristics when specifying requirements to determining the maintainability of the source code to getting early indicators for potentially failing performance or reliability targets. In order to quantitatively manage the fulfillment of the nonfunctional requirements of a product under development, having the right model is essential. Otherwise, there is a high risk of a lot of effort being spent on collecting data of limited use for the success of the project
- **Challenge 2**: Quality models need to be tailored to company specifics and supported by corresponding tools. Existing standards (such as the ISO/IEC 25000 series or the predecessor ISO/IEC 9126) are too generic and hard to fully implement in an organization. Moreover, it is difficult to come up with reliable measures and evaluation criteria. Nevertheless, quality models as proposed in respective standards may be a good starting point. However, if models are not further tailored, there is a high risk that either the models will not find all quality issues, which will in turn lead to bad product quality, or the models will find too many issues, which will be hard to prioritize and comprehensively address in the project
- **Challenge 3**: Cost-effective application of quality models requires a comprehensive framework to facilitate their specification, adaptation, and practical usage for software product evaluation/assessment. Without such a framework that can be used right out of the box, more effort will typically be spent on thinking about how to construct and implement such quality models from scratch than on actually thinking about the content of the models itself and about how product quality can effectively be managed in a development project
- **Challenge 4**: In order to create quality models that are sustainably used for decision-making, their contribution to and value for the organizational goals have to be clarified, and the models need to be integrated into development and decision-making processes (e.g., by defining appropriate quality gates). Without this integration effort, model-based quality management remains a project-specific effort largely dependent on the mechanisms the project manager wants to use for managing product quality. In order for models to be used effectively, the organization as a whole needs to implement a framework to think about software quality and how this contributes to their overall strategy across multiple development projects

This chapter deals with the definition and usage of software quality models for monitoring and controlling the quality of artifacts produced during the software development life cycle. It discusses the different challenges mentioned above in more detail and illustrates solutions based upon first-hand experience from different industrial applications.

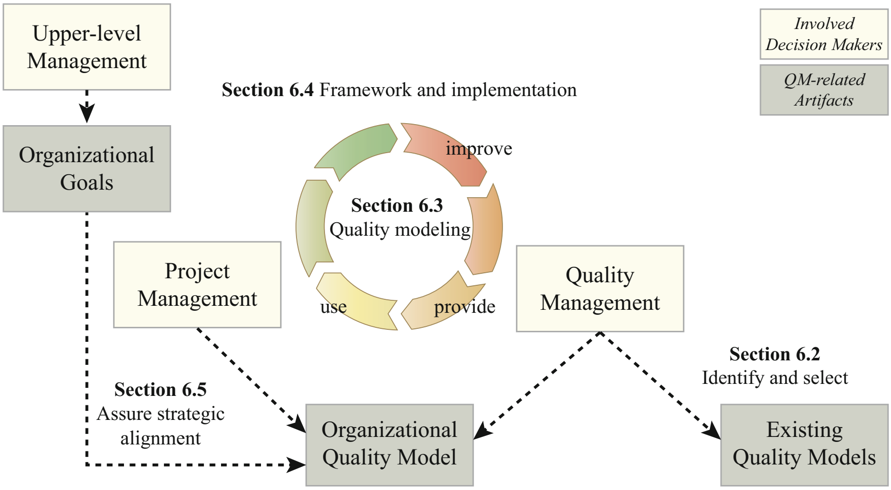


Fig. 6.1 Overview of the structure of Chap. 6

Figure 6.1 gives an overview of the content of this chapter and embeds it into the context of project and quality management. The order in which companies are typically confronted with the introduced challenges determines the order of the subsequent sections: Sect. 6.2 deals with the selection of suitable quality models for a specific organization (Challenge 1), Sect. 6.3 deals with a process for building custom-tailored models that can be used for evaluating product quality during the software and system development process (Challenge 2), Sect. 6.4 introduces a recently finalized, comprehensive framework for specifying and applying quality models in practice (Challenge 3), and Sect. 6.5 deals with integrating quality models into organizational goals and strategies (Challenge 4). Section 6.6 summarizes the chapter and gives an outlook to future work.

6.2 Selecting the Right Quality Models

In practice and research, a variety of different quality models exists that is designed for different application domains, stakeholders, usage contexts, and project environments. The difficult question is: Which quality models are relevant and can be applied in a given environment? To answer this question, a clear picture of the underlying goal to be achieved by a quality model is required. In order to state this goal more precisely, some key questions have to be answered, such as

- Which artifacts of the development process should be analyzed (e.g., requirements document, architecture, design, or code)?

- For which purpose should the model be used (e.g., measuring the product quality of some artifacts or predicting the quality of the final software product delivered to the customer)?
- Which quality characteristics are of interest (e.g., quality in general or some subcharacteristics, such as maintainability or reliability)?
- Which stakeholders want to use the analysis results (e.g., project managers, suppliers, developers)?
- In which environment and context is the model to be applied (e.g., in the development of safety-critical embedded systems or web-based services)?

A systematic literature review and state-of-the-practice survey (Kläs et al. 2009) revealed 79 product quality models[1] ranging from general standards related to software product quality, such as IEEE 1061 (1998), ISO/IEC 9126-1 (2001), ISO/IEC 25000-1 (2005), IEC 61508-1 (2010), ISO/IEC 14598-1 (1999), or ISO/IEC 15939 (2007) via domain-specific standards such as MISRA (1995), ECSS (1996), or UKMD (1997) to quality models from academic research, such as Cavano and McCall (1978), Boehm (1978), Dromey (1998), or Avizienis et al. (2001). It is beyond the scope of this chapter to discuss these models in general. However, for illustrating purposes, Fig. 6.2 gives an overview of the quality models identified and their year of creation, distinguishing between official and de facto standards, models that are actually applied in practice, and models having a more scientific background. Furthermore, it separates models addressing quality in general (dark gray boxes) from models addressing specific quality characteristics (light gray boxes).

If a suitable quality model or a set of quality models needs to be identified, a classification scheme is needed for distinguishing among the different types of existent models. The scheme presented in Kläs et al. (2009) discusses the following dimensions:

(a) **Structural components**: How is the quality model structured? What structural components are supported? For example, models can include structures for refining quality characteristics, for measuring quality characteristics, for evaluating measurement data, for aggregating results, etc.
(b) **Quality modeling goal**: What is the goal addressed by the quality model? In particular, who will use the model, for which purpose, to address which aspects of quality, and on what kinds of artifacts?
(c) **Model instantiation**: Does the quality model only prescribe a structure for specifying quality models (called metamodel) or does it also include an instantiation of that structure [i.e., a specific quality model such as those described in ISO/IEC 25010 (2011)?]

[1] A product quality model is a conceptual or mathematical model addressing one or more relevant characteristics of certain types of work products (such as requirements, design, code, documentation, or the final product) with the objective of better understanding and dealing with these characteristics (e.g., by specifying or quantifying them or correlating them with others).

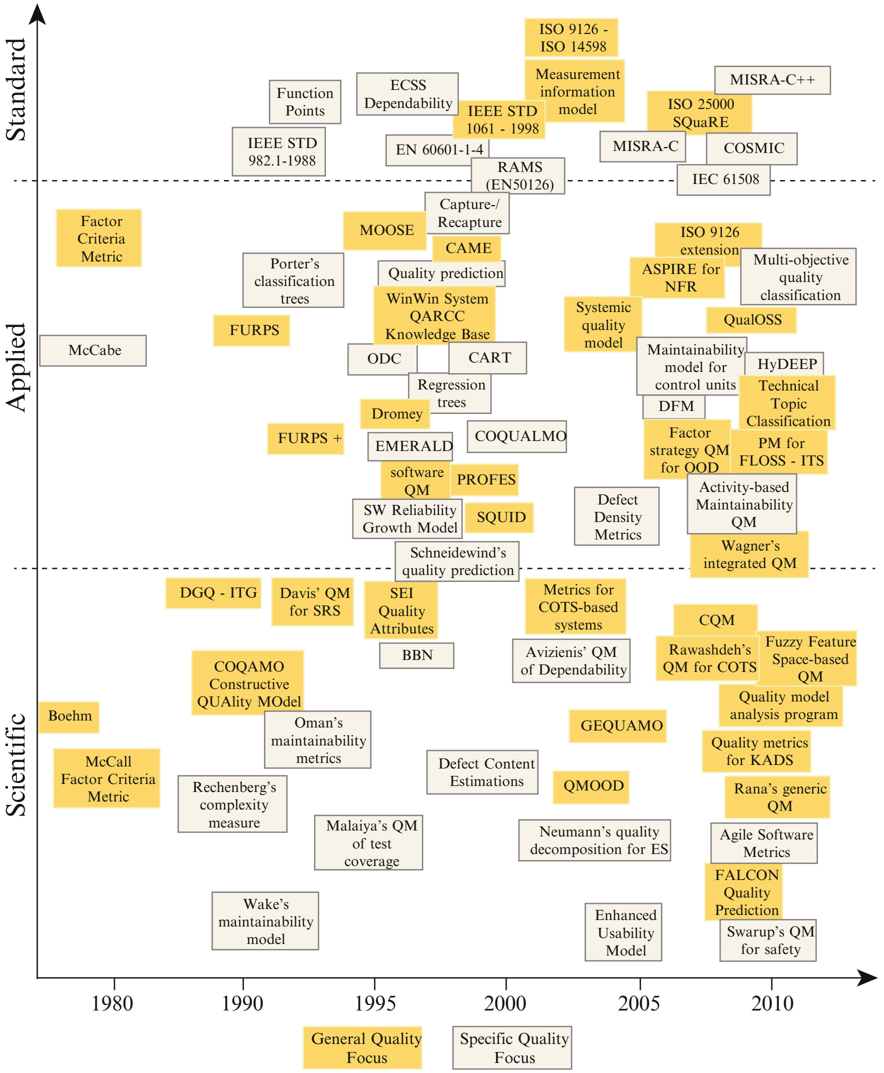


Fig. 6.2 Quality model landscapes

(d) **Method for instantiation/adaptation**: Is a procedure provided to instantiate the provided metamodel or adopt/tailor the presented quality model to specific needs?

(e) **Dissemination**: What is the degree of dissemination of the quality model? For example, is it only used in a scientific context, is there some evidence that it is actually applied in practice, or is it an official or de facto standard?

(f) **Tool support**: Is the quality model tool-supported in terms of specifying and adapting the model as well as actually applying the model to software products?

For example, there may exist commercial tool support, only prototypical tool support, or no tool support at all.

Based on the work in Kläs et al. (2009), the following subsections give further insights into what structural components of models look like (dimension a), discuss the specification of quality modeling goals (dimension b), and describe quality model landscapes as a means for systematically selecting suitable models (based on dimensions a–f).

6.2.1 Structural Components

Figure 6.3 depicts a graphical illustration of typical structural components of quality models. Mainly depending on the application purpose, different components are required by different quality models.

In general, two different parts of a model can be distinguished. The left side shows components related to the quality focus; that is, the quality characteristics of interest. The right side shows the components related to so-called variation factors, that is, factors that explain variances when analyzing quality characteristics. For example, the maintainability of a piece of software (as one characteristic in the quality focus) may be influenced by the experience of the developers or the amount of reuse that was performed (as two potential variation factors).

The relationships between quality focus characteristics and variation factors may be expressed qualitatively or quantitatively. For example, a quality model may specify that developers with higher experience produce software that is easier to maintain (quantitative expression), or it may specify that by increasing the average experience of the developers by 1year, maintenance costs can be reduced by 10 % (quantitative expression). Most quality models rely on qualitative relationships or do not specify variation factors at all. The reason for that is that in order to be able to make such statements, a very thorough understanding of the impact relationships is required. From building cost estimation (Trendowicz et al. 2006) and defect prediction models (Kläs et al. 2010b) (prediction), we know that the variation factors themselves and especially the quantitative relationships highly depend on the particular organization for which such a model is built.

The quality focus may be further refined by a breakdown structure of concepts (typically quality characteristics). For example, ISO/IEC 25010 refines maintainability into modularity, reusability, analyzability, modifiability, and testability. These subconcepts may then be quantified using concrete metrics that can be used for actually measuring the subconcepts. For example, ISO/IEC 25021 (2012) defines concrete metrics for measuring the subconcepts of maintainability. In addition, a quality model may define evaluation criteria for interpreting the measurement data and evaluating/assessing the concepts of interest. For example, ISO/IEC 25040 (2011) defines a quality evaluation reference model and guide. Finally, a quality model may specify a procedure for aggregating the evaluation

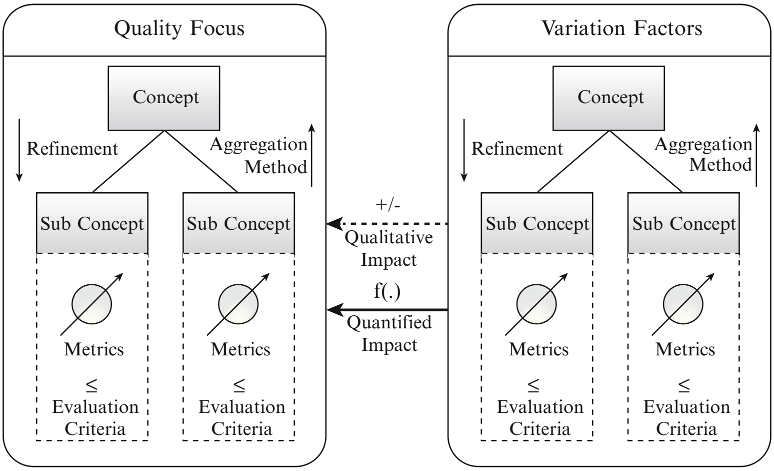


Fig. 6.3 Conceptual elements of quality models [cf. Kläs et al. (2009)]

results of the measurement data, resulting in an overall statement regarding the main concept (across the different refinement levels). For example, if a unique evaluation scale is defined for all concepts, the aggregation may be as simple as the weighted average across the results of all relevant subconcepts, or it may make use of advanced techniques from the field of multi-criteria decision analysis (MCDA) (Trendowicz et al. 2009).

Variation factors can be refined, quantified, evaluated, and aggregated in a similar way as the quality focus (e.g., the concept of developer experience can be refined into domain and programming experience and can be quantified by the respective years of professional experience).

In Sect. 6.3, we present the creation of an example quality model for assessing maintainability, which is a good illustration of the structural components discussed above (Fig. 6.2).

A quality model may be characterized according to the structural components supported by the model. This gives valuable information about what can actually be accomplished in practice when using the model. For example, if a model does not specify evaluation rules, it is possible to measure quality characteristics, but it is not possible to evaluate/access the product's quality (which can be perfectly alright depending on the goals related to the use of the model).

6.2.2 Quality Model Goal

Kläs et al. (2009) use the Goal-Question-Metric paradigm (GQM) (Basili et al. 1994a) as a schema for specifying the goals related to using a quality model. The GQM approach is a de facto standard for defining goal-oriented measurement programs. It supports organizations in clearly specifying measurement goals and systematically deriving metrics from these goals. The GQM goal template distinguishes five parameters that can easily be reused for specifying quality modeling goals:

- **Object**: Specifies the object that is analyzed by the quality model, for example, one or several types of artifacts from the development process, such as requirements specifications, design documents, or code (or parts thereof)
- **Purpose**: Specifies the usage purpose for which the quality model is built, such as characterization, understanding, evaluation, prediction, or improvement
- **Quality focus**: Specifies which quality characteristics are analyzed, for instance, product quality in general or certain quality characteristics as defined by ISO 25010, such as maintainability, reliability, or usability
- **Viewpoint**: Specifies the stakeholders interested in the results obtained by applying the quality model, for example, a quality manager (for assuring product quality at certain milestones of the development process) or a developer (for identifying quality issues and improving the quality of the affected artifacts)
- **Context**: Specifies the organizational scope of the quality model and the application domain covered by the quality model, for example, the model might focus on software developed in a certain business unit or on artifacts created for a certain class of projects; the model may be constructed for the domain of embedded software systems, management and information systems, or web applications

Figure 6.4 illustrates potential usage purposes of quality models (based upon typical GQM purposes) in connection with the structural components of a quality model needed for supporting these purposes:

1. **Specify**: The model is only used for specifying the term "product quality" by refining it into subqualities. Such models specially help to structure quality requirements or issues and define a common understanding of quality. For example, ISO/IEC 25010 defines a breakdown structure of quality characteristics
2. **Measure**: The model is used for quantifying quality characteristics by using metrics. For example, an organization wants to determine a baseline for software product quality based on the measures defined in ISO/IEC 25022
3. **Monitor**: The model is used for identifying trends in quality-related figures and making sure that a certain level is maintained by monitoring measurement data (values of metrics) over time. For example, an organization wants to make sure that the interface complexity of software components does not increase
4. **Assess**: The model is used for assessing/evaluating the quality of dedicated artifacts by checking that quality requirements are fulfilled. For example, it is checked that the application's response time is less than 2 s
5. **Control**: The model is used for assessing product quality periodically or at certain points of the development process. For example, product complexity is checked against a defined threshold every week

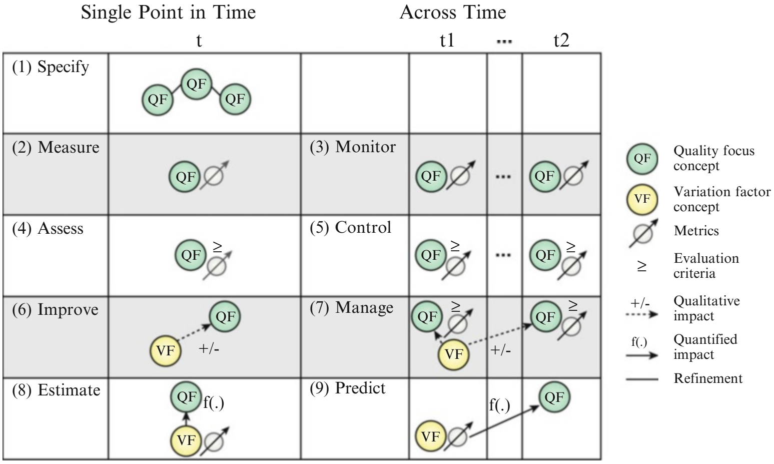


Fig. 6.4 Application purposes of quality models [cf Kläs et al. (2009)]

6. **Improve**: The model is used for improving quality characteristics by manipulating and controlling variation factors that have an impact on these characteristics. For example, coding guidelines (as a variation factor) are introduced to increase maintainability (as a quality characteristic)
7. **Manage**: The model is used for managing product quality over time by controlling product quality periodically or at certain points of the development process and making use of variation factors if quality requirements are not fulfilled. For example, involving more experienced developers (as a known variation factor) if the design of the system has some significant quality issues
8. **Estimate**: The model is used for estimating quality characteristics by making use of variation factors (e.g., because the quality characteristics cannot be measured directly). For example, Capture-Recapture models estimate the number of remaining defects by using the information about the number of joint defects found by different reviewers of the same artifact (Petersson et al. 2004)
9. **Predict**: The model is used to predict quality characteristics in the future, based on variation factors that can be measured earlier. For example, this is the principle of cost estimation models, but also of models that try to predict the number of defects in a delivered software product at early stages of the development process, such as Kläs et al. (2010b) (prediction). Where estimation models are used to determine the present but unknown state of a quantity such as the number of defects in the currently investigated artifact, prediction models make statements about the future, such as the prospective performance of the final product based on the analyzed design documentation

6.2.3 Quality Model Landscapes

Quality model landscapes make use of the classification scheme described at the beginning of this section to provide an overview of available quality models and to allow selecting those models that best fit the needs of an organization. An organization specifies its needs in terms of analyzing software product quality by filling in the classification scheme and thinking about the structural components needed, the goals related to using the quality model, and the other dimensions mentioned above. The landscapes presented in this chapter are based on the work described in Kläs et al. (2009) and combine the results of a literature review and a survey targeting practitioners and their experiences with quality models (Wagner et al. 2010a) (survey).

For example, the quality model overview diagram (Fig. 6.2), which was introduced in the beginning of this section, illustrates a simple quality model landscape visualizing three dimensions: (1) the creation date of the model (from the context information of field b of the classification scheme); (2) whether quality in general is addressed or a specific aspect (from the quality focus information of b); and (3) the dissemination of the models (from field e of the scheme).

Table 6.1 shows another quality model landscape illustrating the number of quality models identified in a systematic literature review and presented in Fig. 6.2 according to two other dimensions: (1) the usage purpose and (2) the main quality focus addressed as mentioned in the model descriptions. For example, if an organization wants to estimate the number of defects, seven potential models are of interest. As can be seen in the table, most of the models from the survey deal with product quality in general and support the specification, measurement, or assessment of product quality. Models considering the defect-proneness of products are mainly estimation or prediction models.

6.3 Building Custom-Tailored Quality Models

This section deals with how to build a custom-tailored quality model in practice. First, general strategies and the high-level construction process are shown. Afterward, the detailed steps for constructing a concrete model are illustrated.

There are two contrary strategies that can be followed: (1) An existing quality model may be selected (such as the one from ISO/IEC 25000) and applied or (2) a new custom-tailored model is built from scratch that exactly fits the needs of an organization. In this section, we will first discuss these two extremes and then explain why and how to combine these strategies to overcome their limitations.

The advantage of the first strategy is that a set of predefined quality characteristics and metrics is available that were elicited based on the knowledge and experience of experts from other organizations and institutions. The disadvantage is that existing quality models are often quite abstract and therefore hard to use out

Table 6.1 Quality models covering different usage purposes and quality focus [cf. Trendowicz et al. (2009)]

	General	Defects	Dependability	Functionality	Maintainability	Portability	Reliability	Safety	Usability
Specify	20		4		1	2		4	1
Measure	16	1		2	2				1
Monitor	1	2					1		
Assess	13				2			1	
Control		1			1				
Improve	7		2		2		1		
Manage	4	1							
Estimate	2	7							
Predict	4	5			1		3		

of the box. For example, the metrics proposed for the ISO/IEC 25000 series are very hard to implement in an organization. Another disadvantage is that the more detailed existing models were created for a certain context and cannot be transferred to a different context that easily. For example, a quality model that was designed for embedded systems cannot be directly transferred to and applied to information systems.

The advantage of the second strategy is that the model ideally fits the specific needs of an organization and can be designed in such a way that it integrates already available measurement data and data collection tools and fits into the organizational and development processes. The disadvantage is that guidance is needed for building up a meaningful quality model; therefore, expert knowledge and experience are needed. Moreover, this is a labor-intensive process and involves experts experienced in model building as well as experts of the corresponding organization the model is being built for. According to Wagner et al. (2010a) (survey), more than 70 % of 125 respondents of a survey employ company-specific models (either developed from scratch or tailored on the basis of company-specific requirements).

Kitchenham et al. (1997) propose a combination of these two strategies to overcome this gap by first choosing the most appropriate existing quality model and, then reusing this model or parts thereof for defining a custom-tailored model.

Figure 6.5 illustrates a process for developing a company-specific quality model based on the Quality Improvement Paradigm (QIP) (Basili et al. 1994b):

1. **Characterize**: Define the scope and application environment in which the organization wants to use the quality model
2. **Set goals**: Define the quality modeling goals (using the goal parameters introduced in the previous section), analyze the suitability of existing models with respect to these goals, select the most appropriate quality model if existent, and then adapt the model to the specific needs of the organization or build the model from scratch if no existing model could be found
3. **Choose process**: Define a measurement plan containing a list of metrics and additional data about who is responsible for collecting this data at which point in time of the development process. Define mechanisms/processes for data collection, processing, and visualization as well as corresponding tool support. The latter includes integrating the usage of the quality models at predefined stages of the development process
4. **Execute**: Apply the quality model to artifacts of the development process, collect measurement data, and assess/evaluate product quality based on evaluation guidelines
5. **Analyze**: Analyze and validate the assessment results and check the validity of the quality model
6. **Package**: Improve the quality model, if needed, and initiate actions for improving the software product quality, if needed

The following sections give a practical usage example of the process as described above based upon a real industrial application. For reasons of

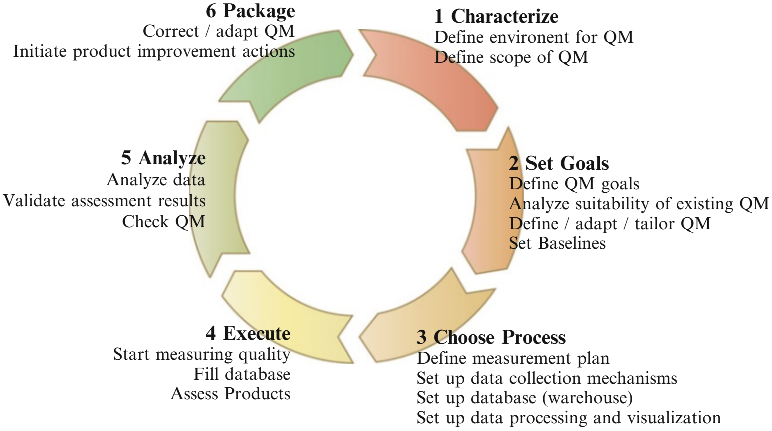


Fig. 6.5 Developing custom-tailored quality models

confidentiality, the original quality modeling goals, the quality model itself, as well as the analysis results have been carefully modified.

Additional industry-related lessons learned from building custom-tailored quality models following the process shown above can also be found in Lampasona et al. (2012).

6.3.1 Characterize: Define Environment and Scope

The first step is about defining the scope and application environment in which the organization wants to use the quality model. Our example organization develops safety-critical systems. In recent years, more and more functionality has been implemented as software. However, software is usually developed by external suppliers and delivered to the system manufacturer as an integrated unit containing the software as well as the hardware (processors, controllers, memory, etc.) the software runs on. The manufacturer has to integrate all these delivered units into an overall system and ensure that they can communicate with each other. The scope of the quality model has been defined as focusing on the software components developed by external suppliers and their integration into the overall system.

6.3.2 Set Goals: Define Goals and Build Quality Model

The second step is about defining the quality modeling goals, analyzing the suitability of existing models with respect to these goals, selecting the most appropriate quality model, and adapting the model to the specific needs of the organization. The major organizational goal behind the usage of the quality model is the reduction of maintenance costs. This should be achieved by ensuring that the software components are of high quality in order to reduce problems during integration, testing, and rework. For this reason, the main goal related to software development is to improve the maintainability of a software component delivered by external suppliers. For assessing/evaluating the quality of the software delivered, a quality model is used.

The quality modeling goal is as follows:

- **Object**: Software components delivered (mainly code and interfaces)
- **Purpose**: Evaluation/assessment
- **Quality focus**: Maintainability
- **Viewpoint**: Quality manager and external supplier
- **Context**: Construction of safety-critical embedded systems

Based on this goal specification, ISO 25010 was used as a reference quality model. However, it was decided to establish custom-tailored metrics and corresponding evaluation rules for measuring and assessing quality characteristics of interest. The tailoring was performed by external quality modeling experts based on a series of workshops and interviews. First, structured interviews with domain experts from the organization were performed, and an initial quality model based on ISO 25010 was created. After that, the model was reviewed and discussed with a broader group of stakeholders (quality managers and suppliers) for achieving a basic consensus among the experts.

An overview of the model obtained is presented in Fig. 6.6. Basically, the quality model created consisted of two parts: one part containing quality characteristics directly related to maintainability and another part containing variation factors influencing maintainability. The main focus should be on analyzability and adaptability. As a consequence, only these two quality characteristics were refined following the classical GQM approach in order to determine metrics that fit these characteristics. For each characteristic, a few metrics were assigned, grouped into different technical topics. For example, analyzability is measured based on the coupling of internal software components. Coupling is measured using standard metrics such as Fan-In (ingoing relationships), Fan-Out (outgoing relationships), and Coupling between Objects (CBO) (Dörr et al. 2004). The variation factor part of the model contains, for example, metric groups related to the logical and physical partitioning/distribution of the software functionality that cannot be influenced by the pure software component, such as a high coupling with other units and the software components of those units. In our example, this cannot be influenced by the external software supplier and becomes part of the variation factor model.

Each metric, group of metrics, quality characteristic, and variation factor has a certain weight that defines how important the respective element is with regard to the parent element. Evaluation rules (employing simple mathematical functions) provide a mapping between measurement values and an evaluation scale. Afterward, a weighted sum is calculated for the groups, quality characteristics, and variation factors considered. The top-level grades make statements about the

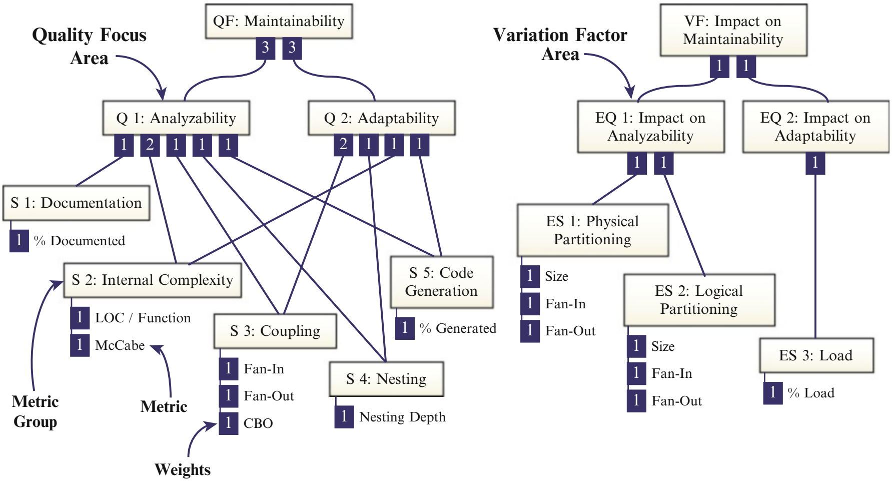


Fig. 6.6 Example quality model for maintainability

maintainability of the delivered software components and external factors that may explain variations between the quality characteristics.

There are several options regarding how to define evaluation rules, such as Trendowicz et al. (2009), and create a mapping function. The difficult part lies in identifying meaningful thresholds for distinguishing between good and bad values. An organization may retrieve values from literature. However, these values normally need to be adapted for the specific usage context. Therefore, most organizations building quality models create their own database of measurement values and perform some statistical analysis on these values in order to find realistic values and thresholds.

The assessment scale for quality characteristics was defined from 1 = very bad maintainability, 2 = bad, 3 = neither good nor bad, 4 = good, to 5 = very good maintainability. The assessment scale for variation factors was defined from 1 = very negative influences on maintainability, 2 = negative influences, 3 = no influences, 4 = positive influences, to 5 = very positive influences on maintainability. For example, if maintainability was rated with 4 and the variation factors were rated with 2, the software component provided good maintainability despite the fact that there was a negative influence.

6.3.3 Choose Process: Set Up Data Collection and Analysis

Step 3 is about defining a measurement plan as well as mechanisms/processes for data collection, processing, and visualization as well as corresponding tool support. Table 6.2 presents the measurement plan (Differding 2001) that was created for the quality model, gives a list of all metrics, and defines the range of measurement

Table 6.2 Measurement plan

ID	Range	Collection time	Data source	Resource
% Documented	%	Quality Gates	Static Code Analyzer	Supplier
LOC/Function	R	Quality Gates	Static Code Analyzer	Supplier
Average McCabe	R	Quality Gates	Static Code Analyzer	Supplier
Fan-In	N	Quality Gates	Static Code Analyzer	Supplier
Fan-Out	N	Quality Gates	Static Code Analyzer	Supplier
CBO	N	Quality Gates	Static Code Analyzer	Supplier
Nesting Depth	N	Quality Gates	Static Code Analyzer	Supplier
% Generated	%	Quality Gates	Static Code Analyzer	Supplier
Physical Size	N	Quality Gates	Interface Data	Manufacturer
Physical Fan-In	N	Quality Gates	Interface Data	Manufacturer
Physical Fan-Out	N	Quality Gates	Interface Data	Manufacturer
Logical Size	N	Quality Gates	Interface Data	Manufacturer
Logical Fan-In	N	Quality Gates	Interface Data	Manufacturer
Logical Fan-Out	N	Quality Gates	Interface Data	Manufacturer
% Load	%	Quality Gates	Questionnaire	Supplier

values, the point in time of the development process when the data should be collected (and analyzed), the data source, and who is responsible for data collection.

A questionnaire was created to get manual measurement data from external suppliers. Moreover, a static code analysis tool was used for extracting code metrics. Due to confidentiality reasons, the code was analyzed on the suppliers' side, and only the measurement values were exchanged. Moreover, interface data was extracted from the configuration database of the organization (containing an overview of all units provided by the suppliers and their interfaces). The organization decided to store all data in a relational database for further analysis and visualization. Figure 6.7 gives an overview of the general procedure for collecting and analyzing the data at certain quality gates of the organization's development process:

- Raw measurement data is collected from different artifacts of the development process (in our case, basically code and interface data) based upon the metrics defined in the quality model. This process is driven by the data collection resources mentioned in the measurement plan. The raw measurement data is stored in a database
- Afterward the measured artifacts are evaluated on the basis of the evaluation rules defined in the quality model. An assessment report is created containing the results of the evaluation. This process is driven by experts for the quality model (typically people from the quality management part of the organization and/or external measurement experts)
- The assessment results are discussed with the relevant stakeholders, in this example the suppliers and the responsible project manager of the organization. A list of countermeasures is created for coping with the weaknesses identified in the quality assessment. The succeeding quality assessment, conducted after the next quality gate of the development process, will check whether the defined

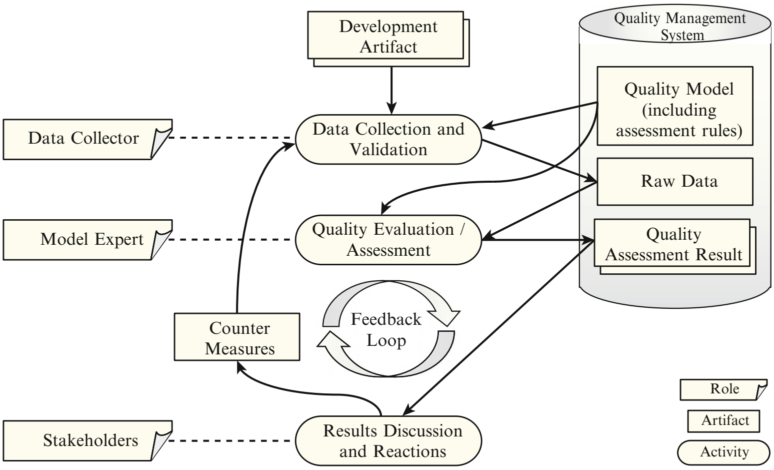

Development Artifact
Quality Management System
Quality Model (including assessment rules)
Raw Data
Quality Assessment Result
Data Collector
Data Collection and Validation
Model Expert
Quality Evaluation / Assessment
Feedback Loop
Counter Measures
Stakeholders
Results Discussion and Reactions
Role
Artifact
Activity

Fig. 6.7 Making use of models for quality assessments

countermeasures had the intended effect and probably make some adjustments, so that an overall feedback loop is created

6.3.4 *Execute: Use the Quality Model for Evaluation/ Assessment*

Step 4 is about applying the quality model to artifacts of the development process, collecting measurement data, and assessing/evaluating product quality based upon evaluation guidelines. The maintainability quality model was applied to 20 already delivered software components of different external suppliers of the organization. Measurement data was collected according to plan, the data was mapped to the evaluation scale described above, aggregated according to the defined weightings in the quality model, and finally, the quality of the delivered software components was analyzed by means of pairwise comparison. In order to avoid comparing apples and oranges, only meaningful comparisons were selected.

The main purpose of the first application of the quality model was to execute a proof of concept and calibrate the model for future usage. To define reliable baselines, more software components would have to be analyzed to allow sound statistical analyses. An example comparison of two software components can be found in Table 6.3. As can be seen from the table, SC1 was evaluated as having good maintainability (4.0 on a scale from 1 to 5) despite having a slightly negative influence on maintainability resulting from the external variation factors (2.5). In contrast, SC2 was evaluated as having poor maintainability (1.75) while having a slightly positive influence on maintainability (3.5). The same argument is generally true for both of the quality subcharacteristics analyzability and adaptability.

Table 6.3 Example results comparing software components SC1 and SC2

	SC1	SC2
Quality focus area		
QF: Maintainability	4.00	1.75
– Q1: Analyzability	3.50	2.00
– Q2: Adaptability	4.50	1.50
Variation factor area		
VF: impact on maintainability	2.50	3.50
– EQ1: impact on analyzability	2.00	4.00
– EQ2: impact on adaptability	3.00	3.00

6.3.5 Analyze: Analyze Evaluation/Assessment Results

Step 5 is about analyzing and validating the assessment results and checking the validity of the quality model. For this purpose, the assessment results were compared with expert statements from the organization regarding the actual maintainability of the software components in practice. The experts gave pairwise ratings about which software component is actually better in terms of maintainability. The quality model rated the software components in the same way as the experts in 90 % of all cases. For the 10 % of failed ratings, the organization observed a significant difference in the requirements implemented in the software component. Because of the limited number of data points, no general statement about the validity of the quality model could be made at that point in time.

6.3.6 Package: Define Improvement Actions

Step 6 is about improving the quality model if needed and initiating actions for improving software product quality. In our example case, the analysis results were discussed in a workshop, and feedback to the data collection resources was provided. The evaluation results of the software components were traced back to concrete metric results (e.g., high coupling, complexity) and corresponding improvement actions have been launched for refactoring the components. Furthermore, based on this discussion, the following improvement actions were defined for the future use of the quality model in the organization:

- Increase the number of analyzed software components and cluster them according to the different types of units to improve the interpretability of the results
- Analyze the distribution of measurement values in the database to establish more reliable baselines/thresholds for the different clusters identified
- Focus on automatic data collection to reduce data collection effort
- Set up means for dealing with missing data to increase the robustness of the quality model

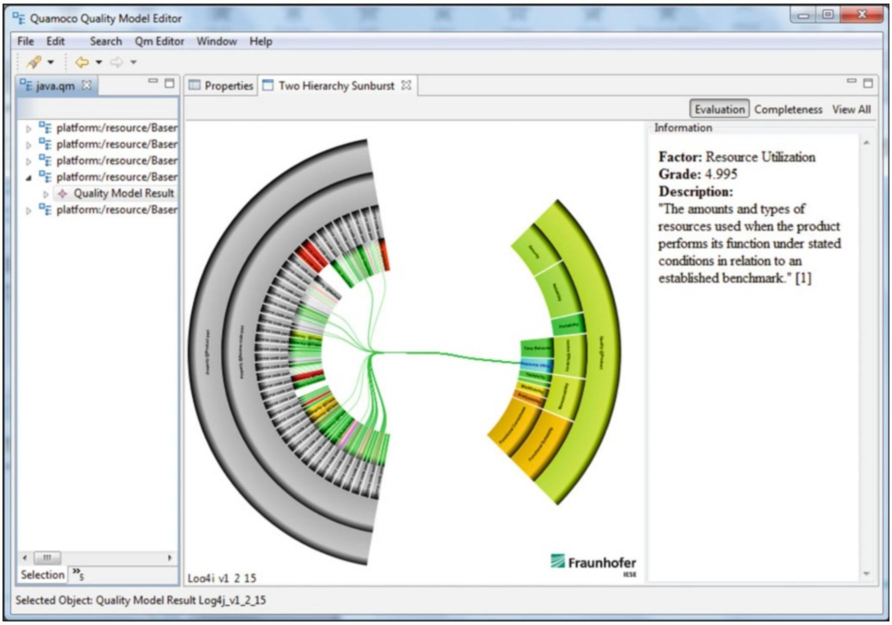
6.4 Specification and Application of Quality Models

The previous sections illustrated how to select an appropriate quality model and how to adapt and make use of such a model in practice for systematically analyzing the quality of software products. However, there is little tool support available for specifying, customizing, and applying these models. This makes it hard and very effort-consuming to develop complete and consistent quality models. A comprehensive framework is needed for the cost-efficient specification, adaptation, and practical usage of quality models. This problem was addressed in the Quamoco research project, which aimed at creating a "Quality Standard for Software-Intensive Systems" (Wagner et al. 2012).

The major motivation was the lack of a mandatory, applicable, and tool-supported quality standard for software development comparable to standards in other industries. A large number of different quality models addressing different goals exist (as seen in Sect. 6.2). However, the existing models are hard to use because the abstraction level is often too high, and it is difficult to come up with reliable and collectable measures. Moreover, adapting/tailoring the models to the specific needs is an effort-consuming process. There is also a lack of reliable evaluation criteria and little support for the meaningful aggregation of quality assessment results, which inhibits meaningful comparisons and benchmarking.

The Quamoco approach implemented the idea of balanced quality models, which means that detailed but highly adaptable core models are provided together with a fine-grained customization process (Kläs and Münch 2008). In Quamoco, the core consists of a quality base model comprising quality characteristics, metrics, and evaluation rules that are essential for all kinds of software-intensive systems. Based on this base model, different domain-specific extensions were created together with appropriate domain experts from various industries (e.g., for information systems, embedded systems, custom software development, etc.). These domain-specific quality models can be customized to the specific needs of an organization (or a part of an organization) using the Quamoco tool suite.

The tool suite supports a user in selecting an appropriate domain-specific model and tailoring this model to the specific needs of the organization by removing or modifying existing parts or adding completely new entities to the model. Furthermore, all models have default connections to measurement instruments actually collecting and analyzing the data and feeding the analysis results back into the model. Figure 6.8 shows a screenshot of the Quamoco quality model editor (downloadable from http://www.quamoco.de). As general quality models tend to become complex, visualization support is essential to get an overview of the quality characteristics addressed and the relationships modeled. The figure shows a sunburst (Stasko 2013) diagram visualizing the model structure of the Quamoco base quality model (which is available at http://www.quamoco.de), together with the evaluation results for an analyzed system. The right part of the sunburst shows the hierarchy of quality characteristics.



Factor: Resource Utilization
Grade: 4.995
Description:
"The amounts and types of resources used when the product performs its function under stated conditions in relation to an established benchmark." [1]

Selected Object: Quality Model Result Log4j_v1_2_15

Fig. 6.8 Quamoco quality modeling tool suite

The left side comprises the measured product-related factors (variation factors) in the model. Each quality characteristic is impacted by different product-related factors; the lines visible in the figure indicate all product-related factors impacting the quality characteristic Resource Utilization (the right hand-side target of all visible lines). The information pane next to the sunburst shows the results of the selected quality characteristic for the system under evaluation (Log4J, a Java-based logging mechanism). The color encoding of the sunburst diagram indicates the evaluation results of quality characteristics and technical factors (from “green” equaling no quality issues to “red” equaling many quality issues). Furthermore, the tool supports tracing down the identified quality issues to concrete findings in the analyzed artifacts.

The major outcomes of the Quamoco project can be summarized as domain-independent as well as domain-specific quality models, an approach for tailoring these models to the specific needs of an organization, a method for assessing the quality of software products, and, finally, tool support for the specification and application of quality models.

The sections below provide additional insights into the metamodel used for the specification of quality models and the approach taken for evaluating/assessing software product quality.

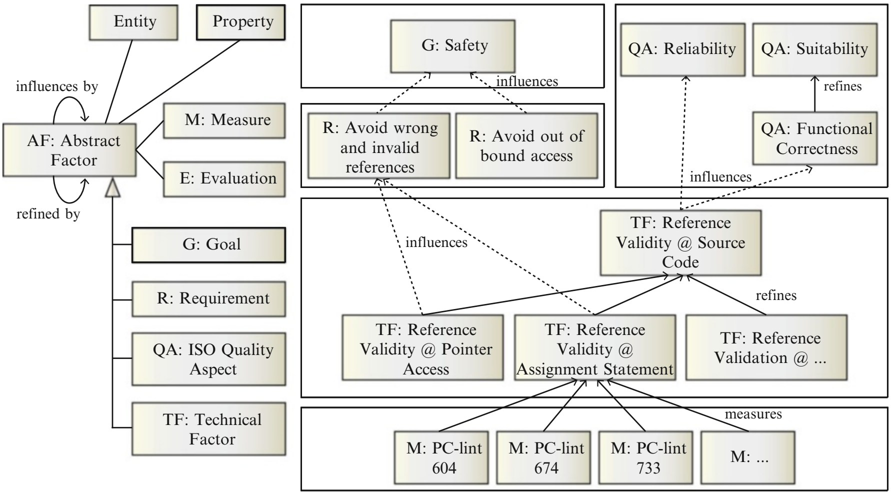


Fig. 6.9 The Quamoco metamodel for embedded systems, [cf. Mayr et al. (2012)]

6.4.1 The Quamoco Quality Metamodel

The underlying metamodel used for the specification of Quamoco quality models is quite generic, with the intent being sufficient expressiveness to model quality in diverse environments with different perceptions on quality (Kläs et al. 2010a) (meta-models). An overview exemplified by an excerpt of the domain-specific model for embedded systems (Mayr et al. 2012) can be seen in Fig. 6.9. The left side illustrates the structural components of the metamodel. The right side illustrates these structural components with excerpts from the quality model for embedded systems.

- **Abstract Factors**: An abstract factor can describe any concept (e.g., quality characteristic or variation factor) that is considered in a quality model. In the embedded systems model, four types of factors are distinguished:
  - Technical Factors define technical properties of entities that are independent of the programming language (such as having "valid pointer references").
  - ISO Quality Aspects describe ISO 25000 quality characteristics (such as "reliability" or "functional" correctness)
  - Requirements describe quality requirements impacted by technical factors (such as "avoiding wrong and invalid references")
  - Goals describe technical goals related to quality requirements (such as having "safe software systems")
- **Entity and Property**: A factor may further be described by an entity and a property it refers to. For example, the technical factor "reference validity of

assignment statements" addresses the entity "assignment statement" (as part of a source code) and the property "reference validity."

- **"Refined by" Relationship**: A factor can be refined into subfactors. However, a factor can only be refined by factors of the same type. For example, the quality aspect "suitability" is refined by the quality aspect "functional correctness." Refinement, abstract factor, entity, and property elements together implement the refinement concept (cf. Sect. 6.2.1).
- **"Influenced by" Relationship**: The relation states whether a factor positively or negatively influences another one and provides a textual justification for this relationship. However, impacts can only be modeled between factors of different types. For example, the technical factor "valid pointer references" influences the requirement "avoiding wrong and invalid references." The impact elements add the possibility to specify quality relationships as introduced in Sect. 6.2.1
- **Measure**: Factors can be quantified by measures. A measure is actually implemented by measurement instruments, which in turn are provided by measurement tools. For example, the technical factor "reference validity of assignment statements" is measured by the measure "PC-lint 604," which is provided by the "PC-lint" tool. In general, a measure may be implemented by different tools. For example, the popular measure "lines of source code" is implemented by all static code analysis tools. If subjective measurement data needs to be collected and no measurement tool is available, a manual measurement instrument (e.g., a certain questionnaire that needs to be filled in manually) can be assigned to the measure. Measure elements implement the quantification concept (cf. Sect. 6.2.1)
- **Evaluation**: Factors are evaluated by evaluation rules describing how to assess the factor on a certain evaluation scale. Factors that have no metrics assigned to them cannot be evaluated in the model. For example, the evaluation rule for the technical factor "reference validity of assignment statements" describes how to interpret and aggregate the measurement data provided by all measures assigned to the factor. Evaluation elements therefore provide implementation for both the general evaluation and the aggregation concepts as introduced in Sect. 6.2.1

The metamodel allows implementation of all conceptual structures needed to specify, measure/monitor, assess/control, and improve/manage quality (cf. Fig. 6.3): It also allows making a clear distinction between the quality characteristics of interest and all factors having an impact on them. By offering the opportunity to formulate arbitrary hierarchies of factors and relationships between different factors, the model allows the definition of different views/perspectives, such as combining a technical/developer view (requirements and technical goals) with a customer/manager (quality aspects) view in one comprehensive model. Furthermore, it allows for explicitly specifying the impact relationship between different factor hierarchies and expressing basic causal relationships.

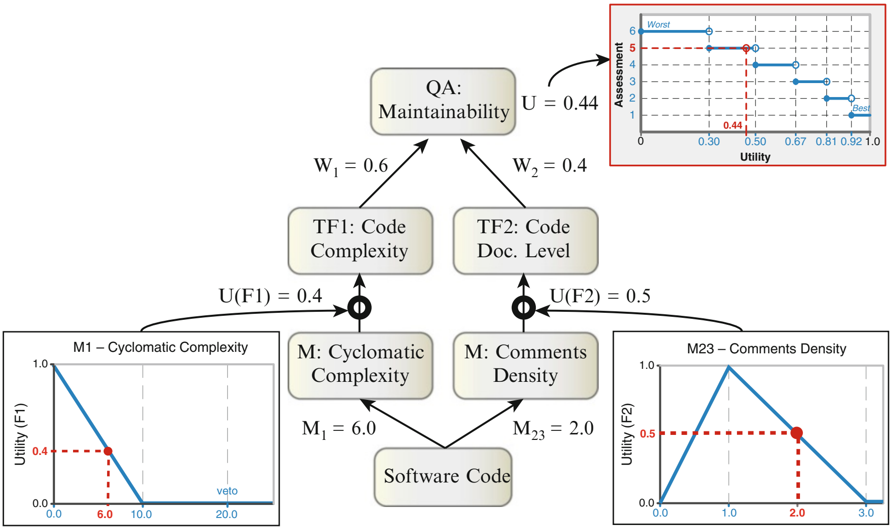


Fig. 6.10 Quality assessment example

6.4.2 *The Quamoco Quality Evaluation*

The main objective of a quantitative quality evaluation is to have an easy-to-understand, evidence-based (measurement-based), and reliable (repeatable) rating of the quality of software products. For that purpose, the measurement values need to be normalized and mapped to a uniform evaluation scale. Quamoco uses values between 0 and 1 as an internal evaluation scale, but supports different interpretation models, which can be defined based on the given context. The default interpretation model maps the evaluation results between 0 and 1 on a scale based upon the German school grade system. The mapping between evaluation results and grades was defined and checked for plausibility for Java-based systems using the evaluation results for more than 100 analyzed open source systems (Wagner et al. 2012).

The evaluation procedure employs multicriteria decision analysis (MCDA) techniques and is illustrated in Fig. 6.10. The procedure for determining evaluation rules (assuming that the remaining parts of a quality model are specified) is as follows (Trendowicz et al. 2009):

- **Weighting**: Quantify the importance of each factor relative to other factors of the same refinement and/or impact hierarchy
- **Scoring**: Collect measurement data for all leaf factors of the hierarchy
- **Evaluation**: Define so-called utility functions that map the measurement values to the evaluation scale with values between 0 and 1. Note that for most measures, some sort of normalization is required to allow defining that utility function
- **Aggregation**: Aggregate the evaluation results provided by the utility functions by computing a weighted sum (which is again a value between 0 and 1)

- **Interpretation**: Map the aggregated evaluation results to an interpretation scale (e.g., school grades)

The Quamoco models come with predefined evaluation rules calibrated on the basis of a survey on the most relevant quality characteristics, the experience of numerous software quality professionals involved in creating the model, and thresholds calculated on the basis of more than 100 software projects. Although the model results were confirmed by experts on five open source software products (Wagner et al. 2012), it is strongly recommended to perform additional calibration activities on organization-specific data before using the model.

6.5 Strategic Usage of Quality Models

The last section deals with how to support decision-making based on the outcomes from applying quality models and how this contributes to higher-level organizational goals and strategies. GQM^{+}Strategies®[2] (Basili et al. 2010) is a measurement planning and analysis approach that provides a framework and notation to help organizations develop/package their operational, measurable business goals, select strategies for implementing them, communicate these goals and strategies throughout the organization and translate these goals into lower-level goals and strategies down to the level of projects, assess the effectiveness of their strategies at all levels of the organization, and recognize the achievement of their business goals. The output of the GQM^{+}Strategies® approach is a detailed and comprehensive model that defines all the elements necessary for a measurement program. GQM^{+}Strategies® makes the business goals, strategies, and corresponding lower-level goals explicit.

In the past, a variety of approaches have been developed covering different aspects of linking activities related to IT services and software development to upper-level goals of an organization and demonstrating their business value, such as the Business Motivation Model (OMG 2010), Practical Software and Systems Measurement (USDoD 2003), Balanced Scorecards (Kaplan and Norton 1992), Information Technology Infrastructure Library (ITIL) (OGC 2002), Control Objectives for Information and Related Technology (COBIT)® (ISACA 2007), or the Sarbanes-Oxley Act (SOX 2002). The aim of GQM^{+}Strategies® is not to replace these approaches, but rather to close the existing gaps with respect to linking goals, their implementation, and the measurement data needed to evaluate goal attainment.

Figure 6.11 illustrates the basic concepts of the approach. The left side describes a hierarchy of organizational goals and strategies. Organizational goals define a target state the organization wants to achieve within a given time frame (e.g.,

[2] Registered trademark of the Fraunhofer Institute for Experimental Software Engineering, Germany and the Fraunhofer USA Center for Experimental Software Engineering, Maryland.

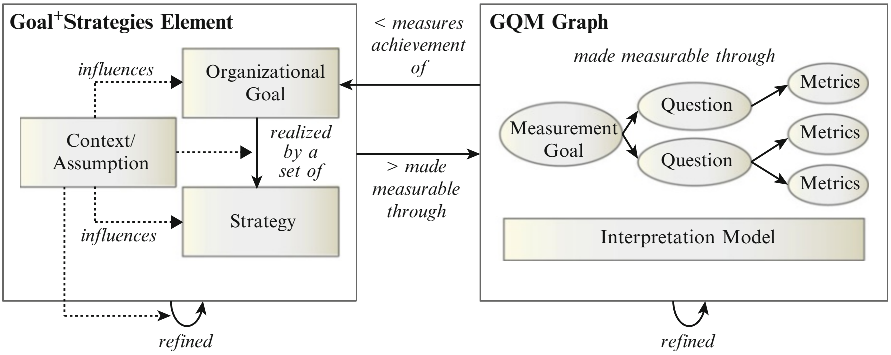


Fig. 6.11 The GQM+Strategies® grid metamodel

improved customer satisfaction or reduced rework costs). Strategies are possible approaches for achieving a goal within the environment of the organization. Context factors and assumptions provide the rationale for the refinement hierarchy. Context factors represent all kinds of factors the organization knows for sure, whereas assumptions are estimated unknowns, that is, what is believed to be true but needs to be reevaluated over time.

GQM graphs define how to measure whether a goal has been accomplished and whether a strategy has been successful. Following the classic GQM approach (Basili et al. 1994a), goals are broken down into concrete metrics. Interpretation models are used for objectively evaluating goals and strategies.

Figure 6.12 and Table 6.4 illustrate excerpts of an example GQM+Strategies® model focusing on the goals and strategies hierarchy and highlighting some measurement data that is collected for evaluating the achievement of organizational goals. On the lowest level, one strategy (DS-S) of the highlighted branch may actually be to build and introduce software product quality models for, e.g., software reliability. The use of these quality models at different quality gates of the software development process will in turn help to decrease the number of defects that slip to later stages of the process (DS-G) by finding potential reliability issues as early as possible. Less defect slippage is related to improving the quality assurance activities of organization X (PR-S), which is a strategy for improving the reliability of the IT products of organization X (PR-G). Improved IT products (NC-S1) will in turn attract more customers to use the IT-based services of company X (NC-G).

The entire model provides an organization with a mechanism for not only defining measurement consistent with larger, upper-level organizational concerns, but also for interpreting and rolling up the resulting measurement data at each level. Having this chain of arguments also supports an organization in demonstrating the values of software-related improvement initiatives, such as the systematic usage of software product quality models. The impact of these models can be evaluated directly in terms of an organization's higher-level goals and make the benefits measurable for it. More industry-related lessons learned from making strategic use

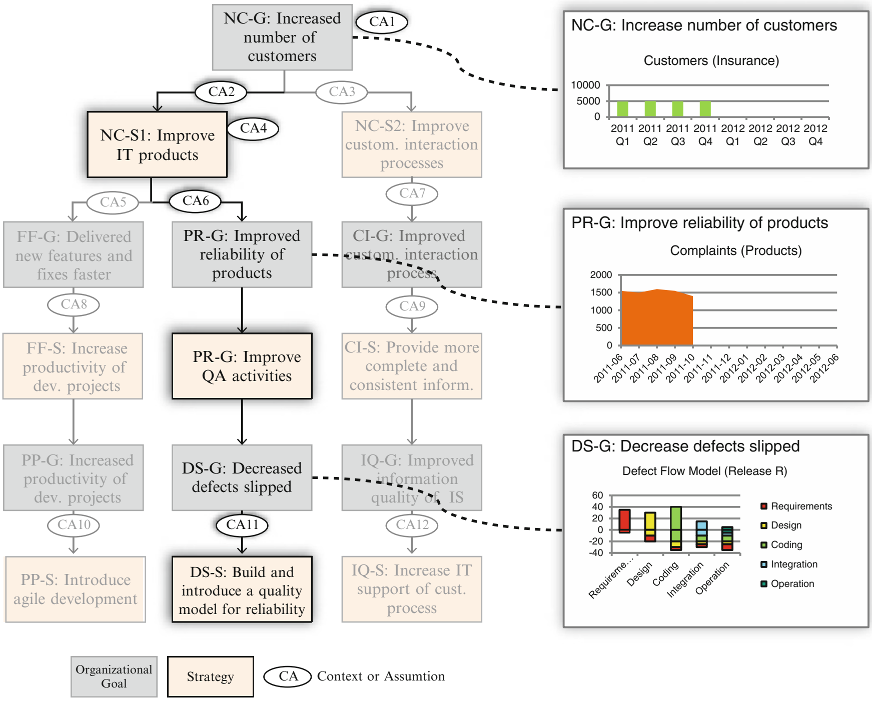

NC-G: Increased number of customers
CA1
CA2
CA3
NC-S1: Improve IT products
CA4
NC-S2: Improve custom. interaction processes
CA5
CA6
CA7
FF-G: Delivered new features and fixes faster
PR-G: Improved reliability of products
CI-G: Improved custom. interaction process
CA8
CA9
FF-S: Increase productivity of dev. projects
PR-G: Improve QA activities
CI-S: Provide more complete and consistent inform.
PP-G: Increased productivity of dev. projects
DS-G: Decreased defects slipped
IQ-G: Improved information quality of IS
CA10
CA11
CA12
PP-S: Introduce agile development
DS-S: Build and introduce a quality model for reliability
IQ-S: Increase IT support of cust. process
NC-G: Increase number of customers
Customers (Insurance)
10000
5000
0
2011 Q1
2011 Q2
2011 Q3
2011 Q4
2012 Q1
2012 Q2
2012 Q3
2012 Q4
PR-G: Improve reliability of products
Complaints (Products)
2000
1500
1000
500
0
2011-06
2011-07
2011-08
2011-09
2011-10
2011-11
2011-12
2012-01
2012-02
2012-03
2012-04
2012-05
2012-06
DS-G: Decrease defects slipped
Defect Flow Model (Release R)
60
40
20
0
-20
-40
Requireme...
Design
Coding
Integration
Operation
Requirements
Design
Coding
Integration
Operation
Organizational Goal
Strategy
CA
Context or Assumtion

Fig. 6.12 Example GQM$^+$Strategies$^®$ grid

Table 6.4 Overview of context and assumption

ID	Type	Description
CA1	Context	Company X provides banking and insurance services. X has a lot of customers in the banking area, but only few in the insurance area
CA2	Assumption	The quality of the IT products has to be improved
CA3	Assumption	The quality of the customer interaction processes has to be improved
CA4	Context	The services of X are built upon an Enterprise information system (IS) that is composed of different software components
CA5	Context	Customers complain that it takes too long to deliver new features and to fix existing bugs
CA6	Context	Customers complain that the IT products are not reliable
CA7	Context	Customers complain about issues related to customer interaction
CA8	Assumption	The delay of existing projects is mainly responsible for the inability to deliver new features and bug fixes faster
CA9	Context	Customers complain about inconsistent and incomplete information during their interaction with company X
CA10	Context	According to the experience from the recently run pilot project, agile development principles will be able to speed up software development
CA11	Context	According to the analysis of the defect data, too many defects appear in the design and coding stage
CA12	Context	Not all services of X are completely IT supported; some have to be provided manually, which decreases information quality

of quality models employing the GQM$^+$Strategies® approach can be found in Basili et al. (2013).

6.6 Conclusions and Future Work

This chapter gave an overview of the challenges and potential solutions for systematically managing the quality of software products. Controlling the quality of all artifacts created during the software development process is one crucial task of professional project management (PMI 2008). Quality models support an organization in general and project managers in particular in objectively evaluating and assessing software product quality through the use of measurement. Knowledge and experience regarding critical quality characteristics and indicators for measuring and evaluating these characteristics are captured in these models.

The chapter highlighted four challenges when dealing with quality models in practice and proposed solutions developed in recent years. First, there is no universal quality model that can be applied everywhere. A variety of quality models exists, and mechanisms such as the proposed classification scheme and quality model landscapes are needed to identify the "right" model based on a clear picture of the goals that should be obtained from using the model.

Second, it is essential to tailor quality models to company specifics. Existing standards are often too generic and hard to fully implement in an organization. A structured process, such as the six-step process proposed, is needed to develop custom-tailored quality models. This also allows for collecting the measurement data needed and to focus data analysis and interpretation on the quality characteristics of interest.

Third, in practice, it is an effort-consuming process to specify and apply quality models because no proven standard techniques, methods, and tools are available. The Quamoco approach described above provides a well-defined metamodel for the specification of quality models and comes with tool-supported, domain-specific models, which can be customized to the specific needs of an organization.

Fourth, in order to create quality models that are sustainably implemented in an organization, the link and contribution to organizational goals need to be clarified. As illustrated by the GQM$^+$Strategies® approach, the data provided from applying software product quality models can be used directly for guiding improvement actions and decision-making.

In the future, software projects will be faced with new challenges that need to be mastered from a practitioner's point of view if an organization wants to provide products with the right level of quality in order to defend and further expand its position on the market. In order to manage future projects successfully, processes and quality assurance mechanisms must handle ever shorter business and technology life cycles and must permit flexible adaptation. Introducing agile development principles is one potential approach allowing for more flexibility (as discussed in Chap. 11). Software products and systems are increasingly being developed in a distributed manner in heterogeneous environments. Chapters 9, 10, and 12 highlight some further challenges and solution approaches for managing global software and IT projects. This is particularly true for cyberphysical systems, where organizations from different domains work together on an integrated solution, each with its own special requirements regarding the integration of different processes and quality management mechanisms. As a consequence, software product quality models must be easy to adapt to new quality requirements on the one side. On the other side, they must be able to address very heterogeneous quality requirements from different domains, which probably use different development processes, and to integrate all these aspects into a comprehensive model. From a researcher's point of view, one major challenge lies in providing empirically proven quality models that can be successfully applied in dedicated domains with known effects. Future work will focus on coping with these aspects.

Acknowledgments The research leading to these results was partially supported by the ARTEMIS Joint Undertaking under grant agreement no. 269335, the research project Quamoco (grant 01IS08023), and from the German Federal Ministry of Education and Research (BMBF).

References

Avizienis A, Laprie JC, Randell B (2001) Fundamental concepts of dependability

Basili V, Caldiera G, Rombach D (1994a) Goal, question metric paradigm. Encyc Softw Eng 1:528–532 (John Wiley and Sons)

Basili V, Caldiera G, Rombach D (1994b) The experience factory. Encyc Softw Eng 1:469–476 (John Wiley and Sons)

Basili V, Heidrich J, Lindvall M, Münch J, Regardie M, Rombach D, Seaman C, Trendowicz A (2010) Linking software development and business strategy through measurement. IEEE Comput 43(4):57–65

Basili V, Lampasona C, Ocampo A (2013) Aligning corporate and IT goals and strategies in the oil and gas industry. In: Proceedings of the 14th international conference on product-focused software process improvement, lecture notes in computer science, vol 7983. Springer, New York, pp 184–198

Boehm BW (1978) Characteristics of software quality. North-Holland, Amsterdam

Cavano JP, McCall JA (1978) A framework for the measurement of software quality. In: Proceedings of the software quality assurance workshop on functional and performance issues. ACM, New York, pp 133–139

Differding C (2001) Reuse of measurement plans based on process and quality models. In: Proceeding of 3rd international workshop on advances in learning software organizations (LSO). Springer, pp 207–221

Dörr J, Trendowicz A, Kolb R, Punter T, Kerkow D, König T, Olsson T (2004) Quality models for non-functional requirements. Fraunhofer IESE Report No. 010-04/E

Dromey GR (1998) Software product quality: theory, model and practice. Griffith University, Brisbane, Australia

ECSS-Q-30A (1996) Space product assurance: dependability

IEC 61508-1 (2010) Functional safety of electrical/electronic/programmable electronic safety-related systems

IEEE 1061 (1998) Software quality metrics methodology

ISACA (2007) Control objectives for information and related technology (CoBIT®). Retrieved 04 12 2007, from www.isaca.org

ISO 8402 (1995) Quality management and quality assurance – vocabulary

ISO/IEC 14598-1 (1999) Information technology software product evaluation

ISO/IEC 15939 (2007) Systems and software engineering measurement process

ISO/IEC 25000-1 (2005) Software product quality requirements and evaluation (SQuaRE)Guide to SQuaRE

ISO/IEC 25010 (2011) SQuaRE system and software quality models

ISO/IEC 25021 (2012) SQuaRE quality measure elements

ISO/IEC 25040 (2011) SQuaRE evaluation process

ISO/IEC 9126-1 (2001) Software engineering product quality - part 1

Kaplan R, Norton D (1992) The balanced scorecard - measures that drive performance. Harv Bus Rev 71

Kitchenham BA, Linkman S, Pasquini A, Nanni V (1997) The SQUID approach to defining a quality model. Softw Qual Control 6(3):211–233

Kläs M, Münch J (2008) Balancing upfront definition and customization of quality models. In: Proceedings of the workshop on software quality modeling and assessment (SQMB 2008), Munich, Germany, pp 26–30

Kläs M, Heidrich J, Münch J, Trendowicz A (2009) CQML Scheme: a classification scheme for comprehensive quality model landscapes. In: Proceedings of the 35th EUROMICRO conference (SEAA 2009). IEEE Computer Society, pp 243–250

Kläs M, Lampasona C, Nunnenmacher S, Wagner S, Herrmannsdörfer M, Lochmann K (2010a) How to evaluate meta-models for software quality? In: Proceedings of the joint international conferences on software measurement. IWSM/MetriKon/Mensura, Shaker, pp 443–462

Kläs M, Elberzhager F, Münch J, Hartjes K, von Graevemeyer O (2010b) Transparent combination of expert and measurement data for defect prediction – an industrial case study. In: Proceedings of the 32nd international conference on software engineering (ICSE 2010), Cape Town, South Africa, pp 119–128

Lampasona C, Heidrich J, Basili V, Ocampo A (2012) Software quality modeling experiences at an oil company. In: Proceedings of the 6th international conference on empirical software engineering and measurement (ESEM), 20–21, pp 243–246

Mayr A, Plösch R, Kläs M, Lampasona C, Saft M (2012) A Comprehensive code-based quality model for embedded systems - systematic development and validation by industrial projects. In: Proceedings of the 23rd international symposium on software reliability engineering (ISSRE 2012), Dallas, TX

MISRA Report 5 (1995) Software metrics office of government commerce (2002). The IT Infrastructure Library (ITIL) Service Delivery, The Stationary Office London

Object Management Group (2010) The business motivation model (BMM) V. 1.1. Retrieved 06 08 2010, from www.omg.org

Office of Government Commerce (OGC) (2002) The IT infrastructure library (ITIL) service delivery. The Stationary Office, London

Petersson H, Thelin T, Runeson P, Wohlin C (2004) Capture–recapture in software inspections after 10 years research—theory, evaluation and application. J Syst Softw 72(2):249–264

Project Management Institute (2008) A guide to the project management body of knowledge (PMBOK® Guide), 4th edn. Project Management Institute

Sarbanes-Oxley Act (2002) Public Law No. 107-204, 116 Stat. 745, Codified in sections of 11, 15, 18, 28, and 29 in United States Code, July 30

Stasko J (2013) Sun burst. Retrieved 29 01 2013, from www.cc.gatech.edu/gvu/ii/sunburst

Trendowicz A, Heidrich J, Münch J, Ishigai Y, Yokoyama K, Kikuchi N (2006) Development of a hybrid cost estimation model in an iterative manner. In: Proceedings of the 28th international conference on software engineering (ICSE 2006), Shanghai, China, pp 331–340

Trendowicz A, Kläs M, Lampasona C, Münch J, Körner C, Saft M (2009) Model-based product quality evaluation with multi-criteria decision analysis. In: Proceedings of the joint international conferences on software measurement (IWSM/MetriKon/Mensura), Shaker, pp 3–20

United Kingdom Ministry of Defense (1997) Def Stan 00-55 requirements for safety related software in defense equipment

US Department of Defense and US Army (2003) Practical software and systems measurement: a foundation for objective project management, v. 4.0c, from www.psmsc.com

Wagner S, Lochmann K, Winter S, Göb A, Kläs M, Nunnenmacher S (2010a) Software quality in practice survey results. Retrieved 03 06 2014, from http://mediatum.ub.tum.de/doc/1110601/1110601.pdf

Wagner S, Broy M, Deißenböck F, Kläs M, Liggesmeyer P, Münch J, Streit J (2010b) Softwarequalitätsmodelle. Praxisempfehlungen und Forschungsagenda, Informatik Spektrum 33(1):37–44 (Springer)

Wagner S, Lochmann K, Heinemann L, Kläs M, Trendowicz A, Plösch R, Seidl A, Goeb A, Streit J (2012) The Quamoco product quality modeling and assessment approach. In: Proceedings of the 34th international conference on software engineering (ICSE 2012), Zurich, Switzerland, pp 1133–1142

Biography Jens Heidrich graduated from the University of Kaiserslautern, Germany, with a Diploma degree in Computer Science and received his doctoral degree from the same university in 2008. He is head of the Process Management division at the Fraunhofer Institute for Experimental Software Engineering (IESE) in Kaiserslautern, Germany, and a lecturer at the University of Kaiserslautern, Germany. He is a member of the German Informatics Society and part of the managing committee of the section “Software Measurement.”

Dieter Rombach studied mathematics and computer science at the University of Karlsruhe, Germany, and obtained his Ph.D. in Computer Science from the University of Kaiserslautern, Germany in 1984. Since 1992, he has held the Software Engineering Chair in the Department of Computer Science at the University of Kaiserslautern. In addition, he is the founding and executive director of the Fraunhofer Institute for Experimental Software Engineering IESE in Kaiserslautern, Germany.

Michael Kläs graduated from the University of Kaiserslautern, Germany, with a German Diploma degree in Computer Science in 2005 and started working at the Fraunhofer Institute for Experimental Software Engineering IESE thereafter. He works on subjects concerning goal-oriented measurement, modeling and assessing quality, as well as defect prediction and cost estimation. His current research interests focus on early quality prediction and aligning large-scale technology evaluation endeavors.

Chapter 7
Supporting Project Management Through Integrated Management of System and Project Knowledge

Barbara Paech, Alexander Delater, and Tom-Michael Hesse

Abstract Software engineering is a knowledge-intensive task. Many different kinds of knowledge are created, for example, system knowledge, such as requirements, design, or code, and project knowledge, such as project plans, decisions, and work items. In this chapter, we study two kinds of project knowledge: work items and decisions. Work items document what should or has been done by whom and when. Decisions represent the solution to a decision problem. They are important to be kept in mind so that the future development will be consistent with the past. These kinds of knowledge can be implicit or explicit. Work items are typically managed explicitly in issue trackers, while decisions are mostly hidden in informal notes or in the artifacts, which result from these decisions. Rationale management research has suggested several approaches to make decisions and their rationale explicit. However, in contrast to issue trackers, which are widespread, these approaches are considered as overhead in industry. In this chapter, we argue that work items and decisions should be managed together with the system knowledge. This has several benefits for project management processes, such as project planning or monitoring, for example, with better information for the allocation of work items or risk identification. We present a vision detailing these benefits and discuss what is known in research and practice about the realization of this vision. In particular, we review existing approaches to capture work items or decisions and their links to other knowledge and discuss the empirical evidence of their benefits for an integrated system and project knowledge management in industry.

B. Paech (✉) • A. Delater • T.-M. Hesse
Institute of Computer Science, University of Heidelberg, Heidelberg, Germany
e-mail: paech@informatik.uni-heidelberg.de; delater@informatik.uni-heidelberg.de; hesse@informatik.uni-heidelberg.de

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_7, © Springer-Verlag Berlin Heidelberg 2014

7.1 Introduction

Knowledge management (KM) is a diverse field. Many general and software engineering–specific facets are described in Bjørnson and Dingsøyr (2008), Aurum et al. (2003). Lindvall and Rus (2003) distinguish:

1. General KM activities for asset reuse, document management, collaboration, competence management, and expert networks
2. KM activities specific in software organizations, such as configuration management and version control, design rationale, traceability, defect tracking, and supporting CASE-tools
3. KM to support organizational learning such as prediction models, lessons learned, case-based systems, and data discovery

In this chapter, we focus on (1) and (2) which means, knowledge captured within a software project and used to support the team of current and future projects to better perform their work. Knowledge can be captured explicitly in artifacts such as documents or emails. Or it can be implicit in the heads of people or as part of a conversation.

We distinguish two different types of knowledge created and used in a software development project. On the one hand, *system knowledge* concerns the software system and its context such as requirements, architecture and design of the system, test cases, and the actual implementation of the design and test cases in the code. On the other hand, *project knowledge* concerns the actual development and maintenance process. In particular, all outputs of project management processes described in the PMBOK guide (Duncan 2013) and its software extension (PMI 2013) are part of the project knowledge. We distinguish the plan knowledge such as the project plan (called project management plan in the PMBOK guide if on a high level and project schedule on a detailed level) or the project participants from the execution knowledge such as work items (including work on bugs and change requests) and decisions (see Fig. 7.1). Note that we only consider the system and project knowledge on the level of an individual software product. There is also generalized knowledge crosscutting different products, for example, system knowledge such as design patterns or project knowledge such as best practices or process models.

Work items describe what should or has been done by whom and when. When captured during project work, work items typically have a completion status, a due date, and are assigned to team members. Typically, they are fine-grained capturing an aspect of the software development work relevant at a particular moment of the project. Often, work items are called issues and describe future work, change requests, or bug reports. In the PMBOK guide (Duncan 2013), work items are called activities: “a distinct, scheduled portion of work performed during the course of a project”. Sometimes (e.g., in Chap. 4, which deals with the allocation of the work to developers), work items are also called tasks.

During project work, *decisions* are made (implicitly or explicitly), which concern the project or the system. According to Ngo and Ruhe (2005), a decision problem at least comprises a set of alternatives and a set of criteria to evaluate each

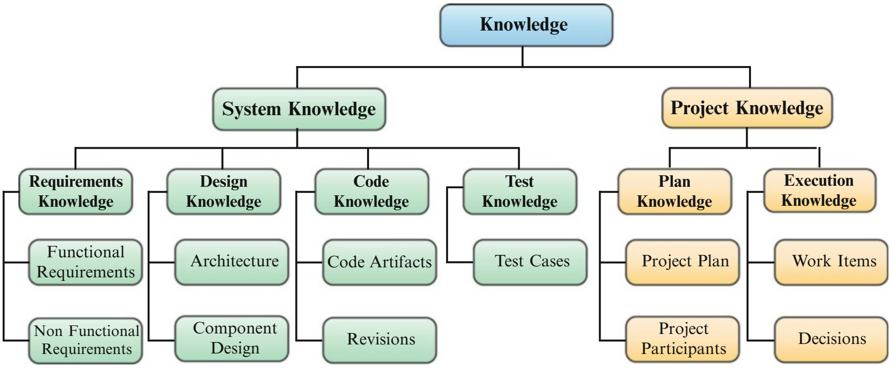


Fig. 7.1 Examples of system knowledge and project knowledge

alternative. A decision embodies a choice of an alternative. Decisions are typically complex and crosscutting, such as the argumentation for the definition of a particular architecture. In consequence, it is useful to capture decision knowledge for considering and reviewing it in upcoming project activities so that former decisions and directions of development are understood over time. Decisions are accompanied by *rationales* (called rationale or rationale knowledge) justifying the decision. Rationale Management aims to make the rationale explicit (Dutoit and Paech 2001, Dutoit et al. 2006). Thus, rationale management approaches capture decisions together with their rationale. As explained in Sect. 7.5, decision knowledge comprises the decision and many other kinds of attributes, and one of them is the rationale.

It is important to note that both work items and decisions can refer to system and project knowledge. For example, a work item can describe work to develop code (code is system knowledge) or to develop a project plan (project plan is project knowledge). Similarly, a decision can concern an architecture (architecture is system knowledge) or a milestone (milestone is project knowledge). We classify work items and decisions as project execution knowledge as they do not describe the system, but human actions (work and decisions) to produce the system. As decisions are created by all team members, the knowledge created by the project manager (e.g., the work items) is only part of the project knowledge.

In practice, system and project knowledge are typically treated separately. System knowledge is captured in all kinds of CASE-tools such as requirements management or test tools. Project knowledge is mostly captured in project management or issue tracking tools. Issue trackers (such as Bugzilla, Jira, and Trac) are a kind of project management software that is widely used in industry; for example, in the Eclipse Community Survey, 80 % of over 600 respondents use at least one issue tracking tool in their daily work (Skerrett 2011). There are tools integrating system and project knowledge. An example for such a widely used tool is MyLyn,[1]

[1] http://www.eclipse.org/mylyn/

which integrates issue tracking into Eclipse. Each time before a new revision is created, MyLyn asks the developer to select a work item from the issue tracker to be linked to the new revision. MyLyn then inserts the unique identifier of the work item into the commit message of the new revision to achieve bidirectional traceability between the new code and the work item. Tasktop[2] extends this with links to requirements or quality management tools. However, this does not provide a comprehensive integration of system and project knowledge allowing the linking of any element of system and project knowledge to another.

We argue that it is beneficial for the whole project team (including the project manager) when system knowledge and project knowledge are captured explicitly and managed together. On the one hand, the usage of project knowledge can be improved when it is linked to the affected system knowledge. For example, when a work item demanding the implementation of a requirement is linked to the requirement and later to the created code, the project manager can easily identify which code was involved in the implementation. This allows, for example, a better estimation of future implementation work. Also, if a decision is captured and linked to the affected system knowledge, the impact of changes of this decision can be analyzed more easily. On the other hand, the capture of system knowledge can be improved, as today in practice more project knowledge is captured than system knowledge. For example, from the links between work items and code as well as between work items and requirements, new links between requirements and code can be inferred. These are very helpful for maintainers.

In our work, we focus on decisions and work items for the integration of system and project knowledge as both are important parts of the project knowledge and both have obvious relations to system knowledge. As outlined in our vision in Sect. 7.2, the links between system knowledge, work items, and decisions already provide ample benefit for project management. To achieve this vision, we have conducted a literature review on the state of the art and practice and developed first ideas to realize this vision. Both are reported in this chapter.

In the following sections, we first discuss our vision of how project management can benefit from integrated management of system knowledge, decisions, and work items. Then, we analyze the state of the art and practice with respect to the integration of system knowledge and decision and work items. First, we treat work items and decisions separately. For work items, we focus on existing approaches that link work items and different kinds of system knowledge. As decisions are more complex, we first review how decisions are captured as such and then how they are linked to other kinds of knowledge. In both cases, we explore how these links are created and used, and we discuss the empirical evidence of integrated system and project knowledge management. Finally, we discuss research necessary to achieve our vision.

The remainder of this chapter is structured as follows: In Sect. 7.2, we sketch our vision of integrated system and project knowledge management and discuss how

[2] http://tasktop.com

project management can benefit from this. In Sect. 7.3, we describe how we conducted the literature review to identify relevant research on work items and decisions. The identified approaches for work items and decisions are described in Sects. 7.4 and 7.5, respectively. In Sect. 7.6, we identify issues for further research, and Sect. 7.7 provides a conclusion and an outlook.

7.2 Our Vision: Integrated System and Project Knowledge Management

In this section, we sketch our vision of *integrated system and project knowledge management through integrated management of work items and decisions and different kinds of system knowledge* (abbreviated ISPKM in the following). As we want to focus on the benefits for the project manager, we look at the knowledge areas as described in the PMBOK guide (Duncan 2013) and its software extension (PMI 2013). In the following (see Table 7.1), we describe how selected areas can benefit from ISPKM knowledge (that means work items and decisions linked to each other or to requirements, design, code, or test, respectively). In the text, we refer to the processes of the knowledge area by listing the corresponding PMBOK chapters. For some of these areas, the processes are also discussed in Chap. 1. In the following, we give examples of how the knowledge areas can benefit from ISPKM. In Sects. 7.4 and 7.5, we discuss what approaches and tools exist for ISPKM and whether empirical evidence for these benefits has already been found. The papers giving the corresponding evidence are already listed in Table 7.1.

As described in Chap. 14, there are four key project management themes, namely, process management, project planning, monitoring and taking actions, and human issues. The knowledge areas we mention below belong to project planning and monitoring and taking action.

Work items that are captured explicitly in issue trackers particularly support the knowledge area *project integration management* and the knowledge area *project time management*, as they help to capture the project management plan (PMBOK Sect. 4.2) and the project schedule (PMBOK Sects. 6.2, 6.3, and 6.6), to direct and manage project work (PMBOK Sect. 4.3), to monitor and control project work (PMBOK Sects. 4.4 and 6.6), and to perform integrated change control (PMBOK Sect. 4.5).

Links from work items to system knowledge support the project team in the project execution in general. If a work item is linked to the inputs of the work (e.g., requirements to be implemented), it is easier for the project participants to understand what to do. They also save time through direct navigation. If the work item is also linked to the output (e.g., the code created), then any other project participant (and in particular the project manager) can better understand the context of the output, for example, who created the output or the used input. Furthermore, the project management processes listed above are supported directly. When developing the project plan and project schedule, the project manager has to assign work items to the responsible project participant. Based on links between system

Table 7.1 Benefits of ISPKM

	Links between work items and system knowledge	Links between decisions and system knowledge
Develop project plan	Assign work items (Helming et al. 2010; Kagdi and Poshyvanyk 2009)	
Direct and manage project work	Understand context of work (Helming et al. 2009a, b)	Understand output of work
Monitor and control project work	Identify output from work item	
Perform integrated change control (Burge and Brown 2008)	Identify change through work items	Identify related change
Define activities, sequence activities, develop schedule	See integration management	
Collect requirements	Suggest changes during implementation (Helming et al. 2009c)	
Manage communication	Identify information recipients	Identify information recipients (Aurum et al. 2006)
Control communication	Report performance	
Control quality	Identify work on quality requirements	Enforce decision
Stakeholder engagement		Explain decision effects
Identify risks		Consider decision assumptions and rationale and decision relations (Burge and Brown 2008)[m]
Control risks		Identify changed decisions
	Improve comprehension of system and project knowledge	Improve comprehension of system and project knowledge (Falessi et al. 2006b, 2008a, b)
	Improve quality of system knowledge (Helming et al. 2009c)	Improve quality of system knowledge
	Improve capture of decisions	

knowledge (e.g., requirements) and work items, new work items can be assigned to developers who have worked on similar system knowledge in the past. This could even be done automatically. Project control is enhanced when the output of a project work (e.g., the design element or code file) is linked to the corresponding work item. As well known from change management, pre and post-traceability links of requirements support the change impact analysis (Gotel and Finkelstein 1994).

Similarly, links from work items to requirements can be used to understand the context of the requirements implementation. In particular, if code is also linked to work items, code affected by the requirements change can be identified. Clearly, this could also be achieved by direct links between requirements and code. However, it takes more effort to capture them directly, compared to the effort of linking the system knowledge to the work items (see Sect. 7.4.2).

The links between work items and system knowledge also support the knowledge area *project scope management* and *project communication management*. The collection of requirements (PMBOK Sect. 5.2) is enhanced by links between work items and requirements. For example, it is easier for project participants to identify requirements relevant to their work and thus to indicate changes necessary because of their work (e.g., if it is not possible to implement a requirement as requested). In general, information distribution (PMBOK Sect. 10.2) is supported as project participants who should receive the information can be identified from the work item. For example, a change on system knowledge by one project participant (e.g., requirements or code change) can be distributed to another project participant who is assigned a work item related to the system knowledge (improve change awareness). Links between work items and requirements are especially important for crosscutting requirements such as quality requirements. For them, enhanced comprehension, as described above, is important. Furthermore, links between work items and quality requirements support the knowledge area *project quality management* because it is easily traceable who was involved in the realization of a quality requirement and when. Similarly, performance reporting in general (PMBOK Sect. 10.3) is supported. The completion status of the output of a work item can be derived from the completion status of the work item. Furthermore, the amount of work of a work item can not only be measured in hours, but also measured in terms of output (e.g., lines of code).

As described above, links from work items to system knowledge help to identify links between system knowledge elements and support change awareness. Thus, they improve the quality of the system knowledge in terms of completeness and up-to-dateness.

Decisions provide an important background for project work. They capture the essentials of the project work and thus of the output of the work that is described in the work items. Typical decision attributes are the problem and solution description with rationale, assumptions, or constraints considered for the decision and links to related system knowledge and other decisions.

As for work items, the capture of links between decisions and system knowledge supports the project work in general (PMBOK Sect. 4.3). If a decision is linked to the outputs of the work (e.g., implemented code or components), it is easier for the project participants to understand the output. The reason is that the decision essentials are implicit in the output, for example, the assumptions or constraints for choosing a particular technology. This is missing if the decision is not linked to the output. Project participants also save time through direct navigation. Similar to work items, links from system knowledge to decisions support project control (PMBOK Sect. 4.4), change control (PMBOK Sect. 4.5), information distribution

(PMBOK Sect. 10.2), and quality management (PMBOK Chap. 8). In particular, if a decision is linked to all affected system elements, the project manager is supported in controlling whether a decision has been realized consistently (decision enforcement). If a decision is to be changed, it is easier to identify which elements will be affected by the change, and if a system element is changed, it is easier to identify whether a major decision is affected and needs to be adapted. As for work items, information distribution is supported. A change of the system knowledge by one project participant (e.g., requirements change) has to be distributed to another project participant who is responsible for a decision related to this system knowledge. Thus, project participants who should receive the information can be identified by a decision. As quality requirements are crosscutting, it is especially important to trace these decisions made with respect to quality.

Unlike work items, decisions and their links are also useful for the knowledge area *stakeholder management* and the knowledge area *risk management*. The main reason is that decisions provide the context for system knowledge through their attributes. This can be helpful for the communication with the stakeholders during the stakeholder engagement management (PMBOK Sect. 13.3), for example, to explain a decision effect that is not desired by the stakeholder when referring to the decision's problem or the constraint description. Assumptions made during decision-making can be helpful in order to create and update the risk register and assumptions log (PMBOK Sect. 11.2). For example, the assumptions of design decisions might indicate potential technical or structural risks for the system. The rationale of decisions includes the drawbacks of decisions and thus indicates the possible risks of a decision. Links from decisions to system knowledge support the identification of risks due to incorrectly realized decisions. Links between decisions help to identify decision dependencies and thus to identify risks due to decision inconsistencies. They also help in continuously controlling risks (PMBOK Sect. 11.6) when decisions change over time.

Furthermore, work items and decisions can be linked to each other and to other project knowledge. Linking work items and decisions could alleviate the capture of decisions as the capture of work items is already widespread. If a work item or a decision concerns the project management work (e.g., creating a project plan), the benefits are similar as for the system knowledge, such as enhanced understanding, control, impact analysis, or information distribution.

Altogether, the examples given above show that ISPKM supports comprehension in general and provides important information for many project management processes. Furthermore, it helps to save time in accessing information. Clearly, there is also effort involved in ISPKM, which has to be balanced with the benefits. The effort mainly arises during the capture of the work items and decisions and their links. As discussed for rationale management approaches in Dutoit et al. (2006), such an effort is particularly worthwhile for complex projects or systems such as distributed projects, product lines, safety critical systems, or COTS-based systems. The possibilities to alleviate the capture are discussed in Sects. 7.4 and 7.5. Often, they involve information retrieval methods. Thus, they are part of software analytics, which aim to gain insights into data from software development and management (Menzies and Zimmermann 2013).

In our view, ISPKM is also worthwhile for agile project management. As described in Chap. 11, agile project management focuses on communication, coordination, collaboration, and decision-making during execution. The information provided by ISPKM supports all of this. ISPKM does not require extensive up-front planning. As evidenced by the widespread use of issue trackers, work items support flexible project planning. Shared decision-making is supported by a shared understanding provided through an explicit decision. Therefore, a lightweight realization of ISPKM should also be very helpful for agile projects.

In the following sections, we review what approaches exist to realize this vision and the evidence of its benefits.

7.3 Literature Review

We conducted a literature review to identify existing research on ISPKM. We used the guidelines of Kitchenham and Charters (2007) for our search strategy and documentation. However, we did not strictly conduct a systematic literature review as the literature was only assessed by one of the authors, namely, the literature on work items by the second author and the literature on decisions by the third author. The aim was to identify *major contributions*, but not to discuss the differences between the approaches in detail. Our overall research questions were

- RQ1: How is the ISPKM knowledge captured?
- RQ2: How is the ISPKM knowledge used?
- RQ3: Which tools are used to support ISPKM?
- RQ4: What empirical evidence does exist for the benefits of ISPKM?

As discussed at the end of the last section, RQ1 and RQ3 are important to assess the effort for ISPKM. RQ2 and RQ4 are important to assess the benefits of ISPKM. The results are presented in Sect. 7.4 for work items and Sect. 7.5 for decisions. In the following, we describe the search strings and sources used and the overall outcome of the search. Each search was documented strictly: We recorded all search results per source and documented the number of hits per source and the papers identified as relevant per source.

Search for Work Item Literature. We used the following publication sources: IEEE,[3] ACM,[4] SpringerLink,[5] and ScienceDirect.[6] We did not explicitly look at approaches capturing work items, as their capture in terms of issue trackers is standard (Skerrett 2011). Thus, for RQ1 with respect to work items, we focus on the capture of the links to system knowledge. The final search string had three terms

[3] http://ieeexplore.ieee.org/Xplore/home.jsp

[4] http://dl.acm.org/

[5] http://www.springerlink.com/

[6] http://www.sciencedirect.com/

Table 7.2 Derived search terms for work items

Search terms		Restriction
Term 1	– **Requirements:** requirement OR "system specification" – **Design:** architecture OR "software design" – **Code:** code OR repository OR revision OR vcs OR "version control system" – **Test:** test*	Title, abstract, keywords
AND		
Term 2	Trace OR traceability OR link OR relation OR mining OR msr	Title, abstract, keywords
AND		
Term 3	"Work item" OR "action item" OR "bug report" OR "change request" OR ticket OR "project management"	Title, abstract, keywords

(see Table 7.2). The first term is divided into terms for the four system knowledge areas of interest, namely, requirements, design, code, and test. To reduce the number of similar terms required in the search string, we used wild cards (*), for example, test* to cover terms like testing, test case, etc. The second term ensures that traceability links between the elements are considered. Furthermore, we explicitly searched for papers in the Mining Software Repositories[7] (MSR) community, with the terms "mining" and "msr", because in this community data mining techniques are often applied to create or use links between work items and code. The third term is a collection of various synonyms for *work item*. All three terms had to appear in the title, abstract, or keywords of the papers.

The searches were executed in February 2013. We had a total of 814 hits for the four sources. Two exclusion rounds were performed to identify the relevant papers. Papers were considered relevant if they considered work items and links to one of the four knowledge areas. In the first exclusion round, we looked only at the title and abstract of each paper and identified 45 relevant papers. We aggregated all results and removed duplicates, resulting in a total of 36 papers. In the second exclusion round, we looked at the abstracts and, in those cases where the abstract did not contain enough information, at the introduction and conclusion. In the end, we identified a total of 19 papers as relevant (requirements = 5, code = 12, design = 0, test = 2).

7.3.1 Search for Decision Literature

The final search string had two terms: the first in two variants (see Table 7.3). The first term 1a addresses decision knowledge and drivers for decisions. It was used again in IEEE, ACM, SpringerLink, and ScienceDirect. For SpringerLink and ScienceDirect, we restricted the search to "Computer Science." A search for

[7] http://msrconf.org/

Table 7.3 Derived search terms for decisions

Search terms		Restriction
Term 1a	"Decision knowledge" OR "decision rationale" OR "decision motivation" OR "decision intention"	Title, abstract, keywords
Term 1b	"Decision knowledge" OR "decision rationale" OR "decision motivation" OR "decision intention" OR "design decision" OR "requirements decision" OR "implementation decision" OR "test decision" OR "maintenance decision" OR ("management decision" AND "software project")	Title, abstract, keywords
AND		
Term 2	Model OR representation OR information OR capture OR link OR benefit OR advantage	Title, abstract, keywords

"decision" or "design decision" as single terms produced far too many results. To be able to cover design decisions as well, we included typical approaches for design decision knowledge by manual search. Many other related topics, such as decision science or management, are also very comprehensive, so it was not our goal to cover them completely. However, we again considered these fields in our manual search in selected publications. In addition, a search in the *International Journal of Software Engineering and Knowledge Engineering*[8] was conducted with a broader search term (see term 1b). The second search term restricts the application of decisions as we were interested in the capture, models, representations, links, and uses of decisions. We included "information" to cover unstructured representations of decision knowledge. The usage of decisions was described by the more specific alternatives "benefit" and "advantage." All terms had to appear either in the title, the abstract, or the keywords of the papers.

The searches were executed in October 2012. In total, we had 520 hits. From the separate manual search, we had another 21 hits. Two exclusion rounds were performed on these 541 hits. Hits were considered as relevant if they explicitly addressed decision knowledge and its contents, representation, capture, linking to the other knowledge areas, or usage. Moreover, hits were not considered if the full text was not retrievable or if they were duplicated. In the first round, the titles and, if necessary, the abstracts were evaluated. This revealed 77 hits. In the second round, all abstracts were examined, and promising papers were fully read. In the end, we identified a total of 33 papers as relevant. Twenty-one of them resulted from engine search. In particular, 19 results were found by term 1a and 2 by term 1b (IJSEKE). Another ten relevant papers resulted from the manual search and are marked by the character "m" in superscript form when being referenced.

Finally, we want to emphasize that the restrictions in sources and search terms can be threats to the validity of our literature review. We included as many sources as feasible, but still even more conferences and journals could be considered. In particular, this is a possible threat to our manual search for literature on design

[8] http://www.worldscientific.com/worldscinet/ijseke

decisions. Next, the search mechanisms provided for each source are evolving over time and might vary in the quantity of results and the quality of hits. Moreover, contributions might have been classified incorrectly during the assessments as only one author per search performed them. Lastly, many empirical contributions we found have their own threats to validity, which can also impact the conclusions drawn from our review. These threats are mentioned separately in Sects. 7.4 and 7.5.

7.4 Integrating System and Project Knowledge Using Work Items

Work items represent explicit knowledge about the process executed in the project. This knowledge is gathered, updated, and detailed continuously over time. In Sect. 7.4.1, we provide an overview of the search results on work items answering RQ1 to RQ4 with respect to the links between work items and system knowledge. In Sect. 7.4.2, we present our own research on integrating system and project knowledge with work items. The following sections refer back to Sect. 7.2 to describe already available empirical evidence for certain parts of our vision of ISPKM.

7.4.1 Results of the Literature Review

Each of the following sections covers one knowledge area: requirements (see Sect. 7.4.1.1), code (see Sect. 7.4.1.2), and test (see Sect. 7.4.1.3). We did not find approaches that link work items and design.

7.4.1.1 Requirements and Work Items

Links between requirements are extensively studied in the requirements engineering community, for example, by Cleland-Huang et al. (2012). However, only very few approaches consider links between requirements and work items.

Creation of Links between Requirements and Work Items (RQ1, RQ3, RQ4). Table 7.4 summarizes the approaches for creating links between requirements and work items. Link creation, tool support, and empirical evidence are emphasized.

All approaches use textual requirements as development artifacts. In the approaches of Helming et al., the traceability links between requirements and work items are created manually. They implemented their approach in the model-based CASE-tool UNICASE, which is an application based on the Eclipse

Table 7.4 Approaches creating links between requirements and work items

Approach	Link creation	Tool support	Empirical evidence
Helming et al. (2009a, b, c, 2010)	Manual	UNICASE	–
Yadla et al. (2005)	Automatic	RETRO	Industrial

framework. UNICASE is capable of storing all kinds of system and project knowledge and the traceability links between them in a single environment.

The approach by Yadla et al. (2005) supports the automatic linking of requirements to bug reports as a special kind of work item, using information retrieval (IR) techniques. It is implemented in the tool RETRO (REquirements TRacing On-target). Basically, the approach uses the IR techniques to search for similarities of texts in requirements and in bug reports. The search is not automatically applied as soon as a bug report is created, but a team member can manually initiate the approach at any time during the project. They evaluated their approach based on two data sets for a NASA scientific instrument. For the first data set, they found a precision (fraction of retrieved instances that are relevant) of 0.69 and a recall (fraction of relevant instances that are retrieved) of 0.85. For the second data set, they found a precision of 0.99 and a recall of 0.70. Results in this range are very good and comparable to manual linkage (Maeder and Gotel 2012).

Thus, there is first evidence that the effort of link capture can be alleviated by automation.

Usages of Links between Requirements and Work Items (RQ2, RQ3, RQ4). Table 7.5 provides an overview of the usage of links between requirements and work items.

In Sect. 7.2, our vision introduced several examples of how traceability links between requirements and work items can be used. Helming et al. provide empirical evidence based on the data of the UNICASE development project that these links support direct navigation, comprehension support, and project reporting (Helming et al. 2009a, b). They observed that as long as work items and requirements stand in meaningful relation to each other (e.g., a work item is referencing a requirement in its textual description), users navigate between them, even when there are no explicit links between them. They confirmed the expected benefit that direct navigation saves time. In their analysis, developers achieved significantly lower navigation distances for linked artifacts than for non-linked artifacts.

The aggregation of links can provide further comprehension support. UNICASE provides an overview of the requirements and the number of associated open work items over time. The authors confirmed our vision that an overview of the completion status of work items that are linked to requirements realistically visualizes the team status, similar to burn-down charts in SCRUM. Helming et al. also confirmed that requirements linked to work items have a higher level of actuality, meaning that developers keep the requirements related to their work items more upto-date. They showed that the number of changes for linked artifacts is significantly higher than for non-linked artifacts.

Table 7.5 Approaches using links between requirements and work items

Approach	Usage	Tool support	Empirical evidence
Helming et al. (2009a, b)	Direct navigation, comprehension support, project reporting Up-to-date requirements specification	UNICASE	Academic
Helming et al. (2009c)	Change awareness	UNICASE	Academic
Helming et al. (2010)	Automatic assignment of work items to developers	UNICASE	Academic

Another benefit of links between requirements and work items that we introduced in Sect. 7.2 is improved change awareness. The notification strategy of Helming et al. (2009c) is based on the traceability links between requirements and work items. Based on the data of a big student project with a real customer, they showed that this traceability-based notification strategy results in a low number of notifications with a high rating of user satisfaction.

As described in Sect. 7.2, the assignment of work items to developers is one important task of a project manager. Helming et al. applied existing machine learning techniques as well as a novel approach relying on the links to assign work items to developers (Helming et al. 2010). They evaluated all approaches on three big UNICASE projects. The novel approach did outperform the other approaches whenever it was applicable.

Altogether, the work of Helming et al. provides a first evidence for the usefulness of ISPKM related to requirements.

7.4.1.2 Code and Work Items

Many commercial software development projects as well as open source projects (e.g., Eclipse, Apache, etc.) use issue trackers (Skerrett 2011) together with a centralized version control system (VCS) like Subversion or the increasingly popular Git. A major focus of the MSR community is to apply data mining techniques to analyze the vast amounts of data stored in issue tracking systems and VCSs. Our literature review found, in addition to specific approaches, the paper by Hassan (2008), which presents a brief history of MSR and discusses the achievements so far. In the following, we discuss the specific approaches to capture links between code and work items.

Creation of Links between Code and Work Items (RQ1, RQ3, RQ4). Table 7.6 provides an overview of approaches and tools creating links, sorted according to their year of publication. Although Nguyen et al. do not present an approach for the creation of links, we identified the paper as relevant and included it in the discussion in this section, because they present empirical evidence for the creation of links between code and work items.

Table 7.6 Approaches creating links between code and work items

Approach	Link creation	Tool support	Empirical evidence
Bachmann et al. (2010)	Automatic	Linkster	Open source
Sureka et al. (2011)	Automatic	Experimental tool	Open source
Bangcharoensap et al. (2012)	Automatic	–	Open source
Davies et al. (2012)	Automatic	–	Open source
Nguyen et al. (2010)	–	–	Open source

Bachmann et al. present an approach that helps to automatically link revisions and bug reports after development (Bachmann et al. 2010). They engaged an expert core developer from the Apache open source project to classify 6 full weeks of the Apache VCS history. They used this data set consisting of 493 revisions and their automatic approach to analyze the connection between the bug reports and the revision data. Bachmann et al. found (among other insights) that not all fixed bugs are stored in the issue tracker. Some are discussed (only) on the mailing list. Furthermore, developers sometimes fix bugs that are only reported in other projects' issue trackers, rather than in their own project's issue tracker, and vice versa. Thus, additional measures, such as incentives, are needed to make sure that project work is accurately reflected in the work items.

Sureka et al. also presented an approach to automatically recover traceability links between standalone bug reports and code files within a VCS (Sureka et al. 2011). Complementing existing approaches that primarily apply regular expressions (e.g., Bachmann et al. 2010), their approach uses a formal mathematical foundation (primarily based on probability theory). They could confirm the feasibility of their approach on a much larger data set of 8,470 bug reports and 10,159 revisions.

The approach of Bangcharoensap et al. (2012) showed that code files can be identified that may contain a bug based on the initial bug report description. Their approach uses three mining approaches: text mining, code mining, and change history mining. In a first step, the text mining approach measures the textual similarities between the description of a bug report and all code files to identify a ranked list of code files. In a second step, the code mining and change history mining approaches are used to further reduce the potential list of erroneous code files. They evaluated their approach using Eclipse platform project data consisting of 2,950 bug reports and 48,764 code files, achieving an accuracy of about 53 %.

Davies et al. propose an approach that measures the similarity between the text used in the bug report and the text of other already fixed bug reports together with the fixed code (Davies et al. 2012). They showed that their own approach is not very effective when used alone, but it showed statistically significant improvements when used in combination with approaches measuring the textual similarity between bug report descriptions and code files.

Nguyen et al. did not suggest a new approach, but studied linkage bias and tagging bias (Nguyen et al. 2010). Linkage bias either means that a bug report is linked to the wrong code or no code at all. Tagging bias means that not all bug reports in an issue tracking system actually represent bugs. Instead, developers often use issue tracking systems to track other issues such as work items, decisions, and enhancements. Therefore, using such data might lead to incorrect bug counts for the different parts of a software system. Nguyen et al. used a nearly ideal data set from the IBM Jazz project consisting of 13,367 fixed bug reports and examined the aforementioned biases. They found that even in this ideal setting, both types of biases do exist in the data set. They argue that linkage bias is more likely due to the software development process rather than being a side effect of the linking heuristics. They also found that, even under tagging bias, existing bug prediction models will still perform almost as if there was no bias.

Altogether, the literature so far focuses on bug reports when dealing with ISPKM related to code. The study of Bachmann et al. shows that issue trackers are not a perfect base, while Nguyen et al. show that manual link creation is not perfect. The other approaches show that automatic creation is possible, but does not yet achieve the precision of link creation between requirements and work items.

Usages of Links Between Code and Work Items (RQ2, RQ3, RQ4). Similar to links between requirements and work items, links between code and work items can be used in various ways (see Table 7.7).

Maeder and Egyed conducted a controlled experiment with 52 subjects (students of computer science) performing 315 maintenance tasks on two third-party development projects: half of the tasks with and the other half without traceability navigation (Maeder and Egyed 2011). They concluded that the mere existence of traceability links between work items and code has a profound effect on the performance (21 % faster) and quality (60 % better) of the implementation tasks. Furthermore, the existence of links fundamentally changed the way subjects navigated through the code. They found that the subjects relied predominantly on traceability navigation when it was available, displacing the manual search navigation in most cases.

The subjects adopted traceability immediately as their major way of navigation within the code, right from the first performed task, even without prior training. However, the frequent usage of traceability links could be related to the inexperience of the students with the code.

The other approaches revealed in our search do not presume links. Instead, they create temporary links between code and work items for specific usage. The empirical evidence focuses on the correctness of the approaches and not on the usage of the links.

As for requirements, Kagdi and Poshyvanyk study the use of IR techniques on code files and work items to assign developers (Kagdi and Poshyvanyk 2009). However, their evaluation is very preliminary as only one change request was analyzed.

Canfora and Cerulo study the usefulness of IR to create links between code and change requests for impact analysis, for example, for developers to identify code to work on as well as for project managers for effort estimation of the change request (Canfora and Cerulo 2005, 2006). Their method exploits IR techniques to identify code file revisions impacted by past change requests similar to the actual one. They

Table 7.7 Usages of links between code and work items

Approach	Usage	Tool support	Empirical evidence
Maeder and Egyed (2011)	Direct navigation Up-to-date requirements specification	Experimental tool	Academic
Kagdi and Poshyvanyk (2009)	Automatic assignment of work items to developers	–	Open source
Canfora and Cerulo (2005, 2006)	Impact analysis	Jimpa (Eclipse plug-in)	Open source
Gethers et al. (2011, 2012)		Experimental tool	Open source

showed with a case study consisting of four open source projects (kcalc, kpdf, kspread, Firefox) that the set of code files returned by their approach is correct in no less than 30 % of the cases and reaches a maximum of 78 %.

In a follow-up work to Kagdi and Poshyvanyk (2009), Canfora and Cerulo (2005, 2006), Gethers et al. also present an approach to perform impact analysis of a given change request to code files (Gethers et al. 2011, 2012). The approach uses a combination of information retrieval, dynamic analysis, and mining of software repository techniques. In addition, this approach uses contextual information such as the execution traces of the code and an initial code file that was verified for change, meaning that this code file definitely needs to be considered. To validate their approach, they conducted an empirical evaluation of four open source projects (jEdit, ArgoUML, muCom, JabRef). Their results indicate that their approach shows statistically significant improvements of these approaches. They only rely on the textual description of the change requests. In certain cases, an improvement of 17 % in precision and 41 % in recall was gained.

Altogether, while the explicit creation of links to code has mainly been studied for big reports, several usages have been studied that implicitly generate links between more general work items and code. Evidence exists that direct navigation provided by ISPKM related to code is useful. Furthermore, for assignment and impact analysis, a better precision and recall can be achieved than for the general creation approaches.

7.4.1.3 Test and Work Items (RQ1, RQ2)

Our literature review only identified two papers concerned with test cases and work items together (see Table 7.8).

Bettenburg et al. conducted an extensive survey among 466 developers and reporters of the Apache, Eclipse, and Mozilla projects to identify “what makes a good bug report” in these projects (Bettenburg et al. 2008). Among many other interesting insights, the developers and reporters said that it is very important to have links between tests and bug reports. This was the second most common

Table 7.8 Approaches creating or using links between tests and work items

Approach	Link creation or usage	Tool support	Empirical evidence
Bettenburg et al. (2008)	Direct navigation	–	Open source
Kaushik et al. (2011)	Automatic	–	Industrial

response, with 51 %. Thus, this survey confirms the usefulness of links between test and work items.

Kaushik et al. proposed an approach using IR techniques to create traceability links between bug reports and test cases, with the aim of recommending test cases for bugs (Kaushik et al. 2011). They evaluated their approach using data from an undisclosed industry project consisting of 9 closed bug reports and 13,380 test cases. The approach was able to find the correct test cases for each bug report, but returned a lot of irrelevant test cases as well. Therefore, the tester still has to select the appropriate test cases from the list of returned test cases.

Altogether, the focus again is on bug reports. While there is evidence of their usefulness, the work on the automatic creation of links to tests is only very preliminary.

7.4.2 A More Comprehensive Integration of System and Project Knowledge Through Work Items

In Sect. 7.2, we described our vision of how system and project knowledge could be integrated. In particular, easy capture is important. The approaches discussed above focus on linkage by IR techniques after the project work has been completed. This motivated us to present a new approach to semiautomatically capture *traceability links between requirements, work items, and code during development* (Delater et al. 2012; Delater and Paech 2013a, b). Our approach captures traceability links during development while developers work on work items. It is implemented in the tool UTC (Delater and Paech 2013c) (UNICASE Trace Client), which is an extension of UNICASE.

There are three options to create links: (A) The developer can select the work item from his/her list of assigned work items *before starting the implementation.*

Then, UTC logs all requirements the developer looks at as well as all touched code files and asks the developer to validate them before linking them to the work item and checking them into the VCS. (B) Similar to the functionality of MyLyn, the work item to be linked can be selected *after the implementation* and before the *check-in* into the VCS. (C) Furthermore, a revision can be linked manually to a work item *after check-in* into the VCS.

An evaluation in three software development projects with undergraduate students (Delater and Paech 2013b) with varying durations between 5 weeks and 6 months showed that the students mainly used options B and C, while option A was only used in 5–10 %. We think that option A will be used more often in larger projects, but there was only a small amount of requirements in our projects. Here, the subjects were very much familiar with the requirements and did not need to look at them often during development. We used the two metrics *precision* (fraction of retrieved instances that are relevant) and *recall* (fraction of relevant instances that are retrieved) to measure the quality of the creation of the traceability links. In the three projects, we achieved a precision of 0.865–0.89 and a recall of 0.90–0.92 for the links between requirements and work items and a precision of 0.83–0.88 and a recall of 0.85–0.92 for the links between work items and code. UTC also supports the exploitation of work items to improve system knowledge by inferring direct links between requirements and code from the links to the work items (a benefit mentioned already in Sect. 7.2). In the three development projects, this resulted in a precision between 0.835 and 0.88 and a recall between 0.89 and 0.92. This is comparable to manually performed linkage (Maeder and Gotel 2012). However, there are some threats to validity. Due to temporal restrictions, the sizes of the development projects were limited, for example, the number of requirements and developed code. This does not allow us to draw conclusions on larger projects. In the development projects, all undergraduate students had basic knowledge in software engineering. No undergraduate student had industrial experience. This does not allow us to draw conclusions on more experienced developers from industry.

Altogether, our approach gives a first evidence that it is feasible to capture the links between work items and system knowledge during development and that this can be used to improve the system knowledge.

7.5 Integrating System and Project Knowledge Using Decisions

Work items are gathered, updated, and detailed continuously over time. Before, while, or after a work item is executed, decisions are made. They often concern a broad variety of system or project aspects. Examples are the determination of milestone or deadline dates or the decision of how to realize a certain system component.

However, decisions and rationale often remain undocumented for most of the activities in software projects (Dutoit et al. 2006). Typically, they cannot fully be recovered at later project stages (Canfora et al. 2000). This is likely to increase the project teams' effort and expenses, for example, during the maintenance of software systems.

Hence, we argue that decisions should be treated similar to and related to work items to facilitate their capture and use. In the following, we describe the state of the art and practice concerning decision knowledge. The next section presents an overview of concepts for decision knowledge and their documentation. Then, the capture, usage, and challenges of decision knowledge are discussed in order to answer the research questions raised in Sect. 7.3.

7.5.1 Concepts for Decision Knowledge

In the following, we sketch decision-making strategies and approaches to document decision knowledge. We use another four sources in addition to our review results in order to explain typical rationale approaches. When being referenced, all those sources are marked by the character "a" in superscript form (as mentioned in Sect. 7.3, superscript "m" stands for manual search). More details on rationale management can be found in the two books on rationale management in software engineering (Burge et al. 2008; Dutoit et al. 2006)^{a}.

Decision-Making Strategies. Decision-making strategies describe how the solution of a given open question is determined.

Table 7.9 provides an overview of decision-making strategies applied in software development found by our literature review.

According to Zannier et al., "much of [the] problem solving can be viewed as problem structuring" (Zannier et al. 2007). Two kinds of problems are distinguished: the *well-structured problem*, for which criteria exist to describe the relationships between the problem and the solution, and the *ill-structured problem*, which needs structuring first to reveal such criteria. In a well-structured problem, the problem-solving process is approached in a structured and managed way, using the knowledge given by the criteria. On the contrary, ill-structured problems are closely related to wicked problems. There is no stopping rule, and solutions cannot be evaluated as true or false, but only as good or bad (Zannier et al. 2007).

According to Lipshitz et al. (2011)^{m} and Zannier et al. (2007), two major types of decision-making strategies can be distinguished: First, *rational decision-making* strategies address well-structured problems. They aim at *choosing* the optimal solution to a given problem. Rational decision-making strategies assume that there is thorough information on the set of alternatives and their outcome so that the evaluation of alternatives is *comprehensive*. This implies that the focus is on the *input and output* of the decision as the alternatives are evaluated according to their estimated performance for the given criteria. Moreover, rational decision-making relies on *formalism* in developing abstract and context-free models of the decision. A typical example for this kind of strategy is the multicriteria analysis with its implementation in the analytic hierarchy process (AHP) (Saaty 1990)^{a}.

The process of rational decision-making in teams has been evaluated by an experiment of Mentis et al. with 36 participants (20 with undergraduate and 16 with graduate degrees) in 12 teams (Mentis et al. 2009). The goal of the subjects was to find an optimal solution to a given planning issue. Mentis et al. evaluated the statements of the team discussions, classifying them as "State" for information sharing, "Analyze" for the interpretation and summarizing of information and "Argue" for positions concerning given statements. They found the beginning of

Table 7.9 Decision-making strategies from the literature review

Decision-making strategies	Empirical evidence
Naturalistic (Zannier et al. 2007; Zannier and Maurer 2006; Lipshitz et al. 2011[m])	Industrial (Zannier et al. 2007; Zannier and Maurer 2006)
Rational (Zannier et al. 2007; Zannier and Maurer 2006; Lipshitz et al. 2011[m])	Industrial (Zannier et al. 2007; Zannier and Maurer 2006) Academic (Mentis et al. 2009)

group decision processes to be dominated by state contributions, whereas argue contributions dominated the end. In newly built groups, state contributions appeared most often, whereas argue-statements were observed most often in established groups. During the whole process, information was shared continually.

Second, *naturalistic decision-making* strategies address ill-structured problems. They assume a scenario with the need for a real-time reaction under dynamically changing conditions. This shifts the determination of a solution from choosing between alternatives to *matching* the given situation with a formerly experienced one. In consequence, the goal is to find a sufficient solution that worked before. Appropriate matches of situations are based on *empirical prescriptions*. Therefore, naturalistic decision-making strategies aim at describing the *prerequisites and rules* for matching situations rather than comparing outcomes. Their decision models are *informal* and accept information to be incomplete. A typical example of this kind of strategy is the recognition-primed model (Klein 2008)[m], describing the creation process of decision patterns.

Zannier et al. evaluated the use of decision-making strategies in software projects in an interview study with 25 professionals. They found that decisions are often made by mixing these types of strategies, for example, by combining rational and naturalistic decision-making (Zannier et al. 2007). This is particularly true for agile projects, where developers tend to employ naturalistic decision-making with rational decision-making elements (Zannier and Maurer 2006). It should be noted that particular software development activities might require specialized decision-making strategies. For example, Falessi et al. present a survey on software design decision-making strategies (Falessi et al. 2011).

Documenting Decision Knowledge. The documentation of decision knowledge depends on the strategy that was chosen to make a decision. Some approaches treat knowledge from rational decision-making, whereas others fit to naturalistic decision-making.

One knowledge metamodel typically considered for documenting decisions is QOC by MacLean et al. (1991)[a]. It consists of six major elements. Questions structure the problems to be explored. Options are the considered solution alternatives, which can be evaluated and ranked with the help of criteria. Assessments link criteria to options. Arguments can be used to challenge or support all elements and particularly assessments. Decisions reflect the option finally selected. The elements of QOC are similar to those used in the decision representation language (DRL) approach by Lee (1991)[a]. Both approaches focus on the problem addressed by a decision, exploring the given issue, related criteria, and solution alternatives. In their structure, they align with rational decision-making.

On the contrary, the choice of a particular solution without extensive problem exploration can be documented via scenario-based claims analysis by Carroll and Rosson (1992)[a]. Claims advocate a certain solution. They can be supported or challenged by arguments, stating advantages and shortcomings of the claimed solution. Both claims and arguments are derived from a usage scenario of the solution. This perspective on decision knowledge aligns with naturalistic decision-making, as it focuses on a scenario and its particular needs.

The approaches to document decision knowledge revealed by our literature review propose their own metamodels for structuring decision knowledge contents. However, they refer to or are built upon QOC/DRL or Scenario-based Claims Analysis. For example, Tyree and Akerman point out that integration with the typically used rationale metamodels is desirable (Tyree and Akerman 2005)[m] and the approach of Burge and Brown (2004)[m] is directly based on DRL.

It should be noted that most of the approaches presented in the following aim at documenting design and architecture decision knowledge. In addition, we found a model for engineering decisions in general, a conceptual model for decision rationale in maintenance, and a claim-based decision documentation model for project and system knowledge. Requirements were often considered when addressing decision knowledge in design.

In the sequel, we first present the metamodel of Tyree and Akerman (2005)[m] in detail and then discuss the other approaches in comparison to it. We do so because this metamodel is very comprehensive. In addition, it comprises 13 different knowledge elements and can be employed as a simple text pattern with standard text editors (see Table 7.10).

Next, we sketch the major differences between this pattern and the other approaches (see Table 7.11, the entries are sorted in ascending order by year of publication). Except for the approach of Smith et al., which can support naturalistic decision-making and is therefore separated by a thick line, all other approaches refer to rational decision-making.

Whereas the pattern of Tyree and Akerman is already comprehensive itself, many new knowledge elements are added by other approaches. However, none covers all. Furthermore, there is a dominance of documentation that aligns with rational decision-making.

Regarding our vision described in Sect. 7.2, we observed that the documentation approaches do support our vision well. Most documentation approaches offer knowledge entities to document rationales and assumptions for decisions so that risk management will benefit. Some approaches focus on enhancing links between decisions and other artifacts.

For example, CM^2 and RATSpeak appear to be particularly useful for project control. They couple decision documentation closely to the systems' source code so that project participants are informed of decisions affecting this code. Other approaches like DGA or TREx introduce further knowledge entities, such as project goals or decision motivation as decision attributes. This helps the project manager

Table 7.10 Decision knowledge elements of the pattern by Tyree and Akerman

Knowledge element	Considered content
Issue	Open questions addressed by the decision
Decision	The alternative finally chosen in the decision
Status	A state describing the current decision condition, like pending, approved, or rejected (see Kruchten et al. 2006^{m} for a sophisticated status model)
Group	The category the decision belongs to
Assumptions	Assumptions concerning the context of the decision and their influences on the alternatives considered
Constraints	Limitations that result from the chosen alternative
Positions	Alternatives considered in the decision
Argument	Rationale supporting the selected position
Implications	The consequences which arise from the decision, like the need to adapt an artifact
Related decisions	Decisions related to the one described, e.g., due to influences or dependencies
Related requirements	Software requirements that set or influence the objectives for the described decision
Related artifacts	Other artifacts being concerned by the decision or concerning it
Related principles	Institutional principles that concerned or influenced the decision
Notes	Further notes related to the decision process

to keep the decisions aligned with the projects' requirements through additional context information. Quality management can be enhanced by the links to quality attributes in the approach of Capilla et al.

7.5.2 Decision Knowledge in Theory and Practice

Decisions often impact the development activity they originate from, but may also constrain successive activities. In this section, we answer RQ1 to RQ4 with respect to decisions and the links between decisions and system or other project knowledge. First, we discuss the abilities of the presented approaches to capture decision knowledge and to manage links between decisions and different knowledge areas. Afterward, the usage of decision knowledge is described followed by a discussion of empirical evidence and challenges.

The Capture of Decision Knowledge (RQ1, RQ3). We see three major approaches for capturing decision knowledge and its links: manual elicitation, automated extraction, or a hybrid approach. Whereas the manual elicitation approaches provide support for documenting decision knowledge manually by the users, the automated extraction approaches provide support for deriving decision knowledge from existing sources. Hybrid approaches apply both manual elicitation and automated extraction. In addition to the capture of the decision itself, the approaches differ in the kinds of links they capture or create.

Table 7.11 Additions to the pattern of Tyree and Akerman

Approach	Main characteristics	Major Additions to Tyree and Akerman Pattern
Cooperative conceptual maintenance model (CM^2) (Canfora et al. 2000)	Focus on decision rationale in the software maintenance process; comments are used to state rationales on source files	Refinement of related artifacts
RATSpeak (Burge and Brown 2004)^m	Extends DRL; emphasis on position and argument element	Background information like tradeoffs, argument ontology
Jansen and Bosch (2005)^m	Focus on architecture decisions	Refinement of implications through architectural modifications
DAMSAK (Babar et al. 2006)^m	Focus on design decision and rationale	Scenario descriptions for decisions
Decision, goal, and alternatives (DGA) (Falessi et al. 2006a)	Extends Tyree and Akerman	Project objectives
Kruchten et al. (2006)^m	Focus on design decisions; Many relationships between decisions such as "constrains", "forbids", "enables"	Enhanced state model, decision scope
Capilla et al. (2007^m, 2011)	Distinguishes optional and mandatory attributes; Focus on relationships between decisions and other knowledge	Attributes for decision evolution, Relationships to quality attributes
Zimmermann et al. (2007)	Extends QOC	Involved persons such as decision identifier, responsible, taker
Architecture rationale and elements linkage (AREL) (Tang et al. 2007)^m	Focus on design decisions and rationale	Motivational reasons for decisions
ADDRA (Jansen et al. 2008)	Focus on design decisions, identified by the delta between two architectural states	Problem causes
Jansen et al. (2009)	Focus on design decisions and knowledge domain modeling	Tailored knowledge metamodel
Könemann (2009)	Focus on design decisions	UML design models
Rockwell et al. (2009)	Focus on engineering design	Extended description of alternatives
Toeska rationale extraction (TREx) (López et al. 2012)	Focus on realization of non-functional requirements (NFR)	Ontology for architecture and NFR; Decision goals
Smith et al. (2005)	Decisions and system and project knowledge	Claims, references to project knowledge

Table 7.12 provides an overview of the approaches and the links to the knowledge areas with the entries sorted by year of publication in an ascending order. We mention every tool we discovered that supports a particular approach. For the

Table 7.12 The capture of decision knowledge in different approaches

Approach	Tool	Knowledge Elicitation/ Extraction	Links
CM^2 (Canfora et al. 2000)	COMANCHE	Manual	Decisions and code files
RATSpeak (Burge and Brown 2004)^m	SEURAT SEURATArchitecture (Wang and Burge 2010)	Manual	Decisions and requirements and artifacts
Archium (Jansen and Bosch 2005)^m	Archium	Manual	Decisions and architectural knowledge
Smith et al. (2005)	LINK-UP	Manual	Decisions and system and project knowledge
Tyree and Akerman (2005)^m	–	Manual	Decisions and other related decisions, requirements and artifacts
DAMSAK (Babar et al. 2006)^m	PAKME (Babar and Gorton 2007)^m	Manual	Decisions and requirements
DGA (Falessi et al. 2006a)	–	Manual	Decisions and goals, other related decisions, requirements and artifacts
Capilla et al. (2007^m, 2011)	ADDSS	Manual	Decisions and other related decisions and architectural knowledge
AREL (Tang et al. 2007)^m	AREL	Manual	Decisions and architectural knowledge
Zimmermann et al. (2007)	ADkwik	Hybrid	Decisions and architectural knowledge
ADDRA (Jansen et al. 2008)	–	Automated	Decisions and architectural knowledge
Jansen et al. (2009)	Knowledge architect	Hybrid	Decisions and architectural knowledge
Könemann (2009)	–	Manual	Decisions and UML models
Rockwell et al. (2009)	–	Manual	Decisions and requirements
TREx (López et al. 2012)	Plugins/Rationale Repository	Automated	Decisions and text documents

following tools, we found a working URL: (SEURAT,[9] Archium,[10] ADDSS,[11] AREL[12]). A more detailed tool evaluation can be found in two tool surveys by

[9] http://www.users.muohio.edu/burgeje/SEURAT/

[10] http://www.archium.net/

[11] http://triana.escet.urjc.es/ADDSS/

[12] http://www.ict.swin.edu.au/personal/atang/AREL-Tool.zip

Tang et al. (2010), Babar et al. (2007), which were also brought up by our literature review. The early approaches up to AREL and the approach of Rockwell et al. and Könemann record decision rationales and create links manually by user input. ADDRA and TREx use automated tool-supported extraction of decision rationales from given artifacts. ADDRA recovers architectural models from given source code and other documents and then derives architectural deltas indicating decisions. TREx gathers decision knowledge by text mining in given plaintext development documents.

Two hybrid approaches by Zimmermann et al. and Jansen et al. allow users to enter data manually according to the respective metamodels. Zimmermann et al. also use requirement descriptions to derive an initial description of the design decisions for the software. Jansen et al. manually extract a project-specific decision rationale model.

Regarding the knowledge areas, many approaches link decisions and architectural knowledge or requirements on which they are grounded or which they impact. Additionally, links to other related decisions are supported. The CM^2 approach and RATSpeak with SEURAT enable developers to link their decision rationale descriptions manually to their respective code file. The approach of Smith et al. allows annotating system or project knowledge elements with claims. In particular, project deadlines and the resource planning for projects can be linked to decision knowledge. The pattern of Tyree and Akerman, and therefore also DGA, provide the option to manually create links to any decision-related artifact from the development process. The TREx approach links a decision rationale to the documents it originates from.

Altogether, we observe that decisions are mainly linked to system knowledge, such as system components or requirements. Links focusing on project knowledge can only be found in the approach of Smith et al. Our literature review did not reveal any approach linking work items and decisions. In contrast to the capture of work items and their links, decisions and their links are mainly captured manually. One reason for this is probably that so far, no commercial tool for the decision capture does exist.

Usage of Decision Knowledge (RQ2). An overview of all usages and the corresponding approaches can be found in Table 7.13.

Many benefits presented in our vision in Sect. 7.2 are supported in these approaches. All approaches aim to increase the understanding of development by transparent decision-making and direct navigation. These usages serve to improve project execution as well as to manage quality. One approach directly aims at enhancing risk management. Two approaches support decision enforcement: Zimmermann et al. propose to inject decision contents when models are transformed or code is generated and Könemann advocates a similar idea when decisions are captured with their corresponding UML models. Four approaches support impact analysis for decision changes.

In addition to tracing the decision impact, AREL also supports tracing decisions to other knowledge along their causes. Thus, causes such as requirements or constraints can be uncovered when assessing a knowledge element. However, no

Table 7.13 Usages for decision knowledge

Usage	Approaches
Transparent decision-making	All
Direct navigation	All
Decision enforcement	Zimmermann et al. (2007), Könemann (2009)
Impact analysis	RATSpeak (Burge and Brown 2004), AREL (Tang et al. 2007)^{m}, ADDRA (Jansen et al. 2008), (Jansen et al. 2009)
Risk management	Smith et al. (2005)

approach provides particular support for analyzing the impact of system knowledge changes on decision knowledge or for information distribution. Also, no approach particularly supports quality management or stakeholder communication.

Empirical Evaluation of Documentation and Usage (RQ4). As discussed in the last section, the documentation of decision knowledge can be beneficial for development activities and for software project management in particular. Some empirical evidence for this was revealed by our literature review, but there are also challenges uncovered by empirical research. In Table 7.14, an overview of empirical studies is given, grouped by their approach or topic.

The need for documenting decisions in software projects is shown in a study conducted by Aurum et al., who interviewed key project team members in a software project with 30 team members of a large Australian insurance company (Aurum et al. 2006). It was found that many decisions were made uninformed, as the issue perception of team members is heterogeneous and the decision leader and decision executor are not the same person by default. Here, there is the need to discuss and align different understandings of issues within the project, which can be done by evaluating and reviewing the documentation of decisions.

Falessi et al. conducted two controlled experiments employing the DGA approach, including the pattern of Tyree and Akerman (2005). They involved 50 master students per study at two universities in Italy and Spain (Falessi et al. 2008a, b). The subjects had to evaluate given decision descriptions structured by the DGA approach. In detail, it was asked whether particular knowledge elements were required in order to understand the documented decision. In both studies, the decision knowledge elements "Issue," "Decision," and "Related requirements" were found to be the three elements that were required by most participants. Moreover, the evaluation of their Italian study showed that DGA documentation did not significantly affect the time needed for the individual and team decision process, but increased the effectiveness of both decision processes (Falessi et al. 2006b).

A controlled experiment for RATSpeak and SEURAT was performed with two groups of ten users. It indicated that identifying changes in the code and maintaining software was alleviated by the use of SEURAT (Burge and Brown 2008)^{m}.

However, threats to validity exist for each study. The project studied in Aurum et al. (2006) was complex and customer-specific, so results might vary for smaller

Table 7.14 Empirical evaluation of decision knowledge documentation

Topic or approach	Empirical evidence
Needs for decision knowledge	Industrial (Aurum et al. 2006)
RATSpeak with SEURAT, impact analysis	Academic (Burge and Brown 2008)[m]
DGA	Academic (Falessi et al. 2006b, 2008a, b)
Challenges in decision documentation	Industrial (Tang et al. 2006)[m]

projects and projects in different domains. Falessi et al. point out the designed environment of their study, which might affect the transfer of the results to practice (Falessi et al. 2006b). Burge and Brown particularly mention the small size of the groups in their study as a cause for individual differences (Burge and Brown 2008).

Furthermore, some challenges exist that might influence the success of establishing and using a decision documentation culture in a project. A survey of Tang et al. with 81 practitioners revealed two kinds of challenges (Tang et al. 2006)[m]. Similar challenges are described in Dutoit et al. (2006)[a].

- **Psychological challenges:** People tend to document positive information more often than negative. Especially when using decisions as a measure for uncertainty, the documented knowledge may not fully reflect the given information in a project. In addition, staff members may be interested not to disclose all their knowledge in order to strengthen their own position. So it is important to establish a knowledge sharing culture in a project and reward knowledge-sharing actively.
- **Practical challenges:** Team members may not be able or do not want to spend the effort of documenting decisions and their rationales. It is important to plan the project in a way that allows knowledge documentation and reflection. Also, the given approaches to document decision knowledge are likely not to fit all project contexts. In small and noncomplex projects, rationale documentation is not found to be necessary by team members due to the increased effort with low benefits at hand. Therefore, it is necessary to tailor the documentation technique to individual project needs. Moreover, the tool support for documenting decision knowledge is limited.

Altogether, there is a first evidence for the need of decision knowledge, as well as for the usefulness of some decision knowledge. The evidence regarding RATSpeak and SEURAT supports the benefits of links between system knowledge and decisions. However, it is likely that establishing the capture of decision knowledge is more difficult than for work items and their knowledge.

7.6 Research Issues on Integrating System and Project Knowledge

Our vision on ISPKM discussed many practical benefits. As can be seen from Table 7.1, many of them have been confirmed for work items in the first studies. Also, for the usefulness of decision knowledge, some evidence does exist. In the following, we list several research challenges that need to be solved to make ISPKM really useful in practice.

The literature review on work items has shown that only few approaches support explicit links to work items. Code is mainly linked to bug reports. Furthermore, work items have so far not been linked to design. The links are mainly used for navigation, comprehension support, assignment of work, and support of the change process in terms of change awareness, motivation to keep system knowledge up-to-date, and impact analysis.

The literature review on decision knowledge has shown that decision knowledge in itself is very complex. There are many different approaches to its documentation, mainly focusing on rational decision-making. Furthermore, most approaches rely on the project participants to create the decision knowledge manually. Thus, the capture effort is a major problem. Decisions are mostly linked to system knowledge and used for direct navigation. In the area of project management, risk management is supported by the capture of rationales. First approaches exist that support the consistency between decisions and the impacted knowledge, but their usage for information distribution and stakeholder communication is not yet supported. Also, work items and decisions are not linked.

The following research issues are derived from the gap between our vision in ISPKM and the insights above, as well as from our experiences based on the work we described in Sect. 7.4.2.

7.6.1 Project Knowledge

Improved documentation of decisions mixing naturalistic and rational decision-making: As discussed in Sect. 7.5.1, empirical evidence has shown that project participants apply both rational and naturalistic decision-making. Thus, it is important that these two kinds can be supported in parallel. However, only very few approaches support naturalistic decision-making. In our view, this is one of the reasons, why decision knowledge capture is not accepted in practice. As can be seen from the use of social media in software projects (see Chap. 16), software project participants like to share knowledge in a flexible way. Furthermore, it should be possible to transform rational decision knowledge into naturalistic knowledge and vice versa. Thus, it is necessary to study how naturalistic decision-making can be supported. For example, it should be possible to condense all rationales and decisions relevant for a specific system knowledge element into one claim that argues for this specific element. Similarly, it should be possible to start out with claims and gradually enrich these with decision knowledge and rationales. The transformation between these two kinds could also help to support the enforcement of decisions in the corresponding knowledge.

Integrating work items and decision knowledge to alleviate the decision capture: An additional way to alleviate the capture of decision knowledge is to relate it to work items. Decisions result from the work that is continually done in the project, which is described in work items. Every time a work item is performed, decisions have been made before, while, or after working. Thus, work items and decisions could be captured together and linked directly (similar to the capture of links to code described in Sect. 7.4.2) so that the context and history of decisions is understood from the project knowledge point of view. Furthermore, the system knowledge linked to the work items could provide a context for the decisions. Both benefits also work the other way round. Decisions provide project context for the work items, and system knowledge related to decisions provides system knowledge important for the work item. Care must be taken to minimize the capture effort, so that, for example, requirements need not be linked manually to both work items and decisions. As discussed below, all these links should be exploited with dedicated algorithms and aggregated for project monitoring, reporting, and risk management.

Dealing with implicit knowledge related to decisions based on links: It is well known that because of the huge amount of implicit knowledge in software projects, access to implicit knowledge (often called tacit knowledge) should also be supported (Gervasi et al. 2013). Based on our experiences with links between work items and system knowledge, we suggest that links from system and project knowledge to the persons touching it could be captured automatically. They then serve as a hint of where to find implicit knowledge.

7.6.2 System and Project Knowledge

Improved capture of links: Clearly, efficient capture of the links is of utmost importance for link usage in practice. This is a general issue, not only for links concerning work items or decisions (Cleland-Huang et al. 2012). There are many approaches using IR techniques, but precision and recall is in general not yet sufficient for practice. Therefore, it is important to use other information in the link creation process such as logs from the development process. The semiautomatic capture based on logging provided by MyLyn and our approach are first steps in this direction.

More comprehensive integration of system knowledge: The existing approaches should be extended to deal with other kinds of system knowledge. In particular, the semi-automatic approaches of capturing related system knowledge while working on a work item could be extended to also capture related design and test or project knowledge. In addition, more different kinds of work items could be supported when creating and using the links. As for work items, current decision knowledge approaches do not comprehensively relate decisions and system knowledge. In particular, decisions concerning code or tests are not documented explicitly so far. Furthermore, there are many different approaches for capturing architectural decision knowledge. Thus, research is necessary to identify the most important decision knowledge elements for the different system knowledge areas.

Intelligent exploitation and management of the links: Links can be exploited to uncover further relationships between system and project knowledge elements. As suggested in our own approach (Sect. 7.4.2), from the links between requirements and a work item as well as between the work item and the code, one can infer direct links between requirements and code. However, over time, links inferred from one work item might be made obsolete by work on other work items. Thus, intelligent algorithms are needed to discard derived links not relevant any longer. In general, intelligent algorithms are needed to manage the links over time. Very likely, this will depend on the kind of knowledge and links so that many different dedicated approaches need to be developed.

Improved support for project managers: In Sect. 7.2, we have listed many project management activities that could be supported by ISPKM. For many of them, some first approaches and evidence exist (see Table 7.1). More such approaches need to be developed. In any case, the approaches need to be incorporated into commercial tools. For issue trackers, this seems to be viable in the near future.

More empirical evidence: As there are only few approaches integrating system and project knowledge management comprehensively, empirical studies on the benefits of the integration can so far only be conducted as experiments such as Maeder and Egyed (2011). More such evidence is needed. In particular, it is important to understand the benefits of semiautomatic capture (similar to our approach) and to conduct studies in industry. The latter will only be possible if these ideas are integrated into commercial tools as mentioned above.

7.7 Conclusions and Outlook

This chapter discussed the capture and use of work items and decisions and related system and project knowledge. While work items are often captured in practice, only few approaches link them explicitly with other kinds of knowledge. In contrast, decisions are rarely captured in practice; however, there are many different academic approaches for their documentation. In both cases, some first empirical evidence exists that links between work items or decisions on the one hand and system and other project knowledge on the other hand are beneficial in general and in particular for project management activities.

Our own work focuses on two aspects: In order to alleviate the capture of system and project knowledge and their links, we are working on the semi-automatic capture and exploitation of links between work items and other kinds of knowledge. In order to support decision-making in practice, we work on tool support to integrate naturalistic and rational decision-making. Based on these two results, we hope to be able to support and study ISPKM in practice.

Acknowledgments This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design for Future—Managed Software Evolution. We thank the reviewers for their helpful comments.

References

Aurum A, Jefferey R, Wohlin C, Handzic M (eds) (2003) Managing software engineering knowledge. Springer, Berlin

Aurum A, Wohlin C, Porter A (2006) Aligning software project decisions: a case study. Int J Softw Eng Knowl Eng 16(6):795–818

Babar MA, Gorton I (2007) A tool for managing software architecture knowledge. In: SHARK/ADI'07: 2nd workshop on sharing and reusing architectural knowledge - architecture, rationale, and design. IEEE, Minneapolis, MN

Babar MA, Gorton I, Kitchenham B (2006) A framework for supporting architecture knowledge and rationale management. In: Dutoit AH, McCall R, Mistrik I, Paech B (eds) Rationale management in software engineering. Springer, Berlin

Babar MA, de Boer RC, Dingsoyr T, Farenhorst R (2007) Architectural knowledge management strategies: approaches in research and industry. In: SHARK/ADI'07: 2nd workshop on sharing and reusing architectural knowledge - architecture, rationale, and design. IEEE, Minneapolis, MN

Bachmann A, Bird C, Rahman F, Devanbu P, Bernstein A (2010) The missing links: bugs and bug-fix commits. In: FSE: 18th ACM SIGSOFT international symposium on foundations of software engineering, pp 97–106

Bangcharoensap P, Ihara A, Kamei Y, Matsumoto K (2012) Locating source code to be fixed based on initial bug reports - a case study on the eclipse project. In: International workshop on empirical software engineering in practice, pp 10–15

Bettenburg N, Just S, Schröter A, Weiss C, Premraj R, Zimmermann T (2008) What makes a good bug report? In: FSE: 16th ACM SIGSOFT international symposium on foundations of software engineering. ACM, New York, NY, pp 308–318

Bjørnson FO, Dingsøyr T (2008) Knowledge management in software engineering: a systematic review of studied concepts, findings and research methods used. Inf Softw Technol 50:1055–1068

Burge JE, Brown DC (2004) An integrated approach for software design checking using design rationale. In: First international conference of design computing and cognition, pp 557–576

Burge JE, Brown DC (2008) Software engineering using RATionale. J Syst Softw 81(3):395–413

Burge JE, Carroll JM, McCall R, Mistrik I (2008) Rationale-based software engineering. Springer, Berlin

Canfora G, Cerulo L (2005) Impact analysis by mining software and change request repositories. In: 11th IEEE international symposium software metrics, Como, pp 21–29

Canfora G, Cerulo L (2006) Jimpa: an eclipse plug-in for impact analysis. In: CSMR: 10th European conference on software maintenance and reengineering, Bari, pp 340–342

Canfora G, Casazza G, de Lucia A (2000) A design rationale based environment for cooperative maintenance. Int J Softw Eng Knowl Eng 10(5):627–645

Capilla R, Nava F, Duenas JC (2007) Modeling and documenting the evolution of architectural design decisions. In: SHARK/ADI'07: 2nd workshop on sharing and reusing architectural knowledge - architecture, rationale, and design. IEEE Computer Society, Washington, DC

Capilla R, Zimmermann O, Zdun U, Avgeriou P, Kuester JM (2011) An enhanced architectural knowledge metamodel linking architectural design decisions to other artifacts in the software engineering lifecycle. In: ECSA: 5th european conference on software architecture, Essen, Germany, pp 303–318

Carroll JM, Rosson MB (1992) Getting around the task-artifact cycle: how to make claims and design by scenario. ACM Trans Inf Syst 10(2):181–212

Cleland-Huang J, Gotel O, Zisman A (eds) (2012) Software and systems traceability. Springer, New York

Davies S, Roper M, Wood M (2012) Using bug report similarity to enhance bug localisation. In: WCRE: working conference on reverse engineering, Kingston, pp 125–134

Delater A, Paech B (2013a) Analyzing the tracing of requirements and source code during software development: a research preview. In: REFSQ'13: 19th international working conference on requirements engineering: foundation for software quality, LNCS, vol 7830. Springer, Berlin, pp 308–314

Delater A, Paech B (2013b) Tracing requirements and source code during software development: an empirical study. In: ESEM'13: 7th ACM/IEEE international symposium on empirical software engineering and measurement, Baltimore, MD, pp 24–35

Delater A, Paech B (2013c) UNICASE trace client: a CASE tool integrating requirements engineering, project management and code implementation. In: Workshop Nutzung und Nutzen von Traceability. Lecture notes in informatics, vol 215. GI, pp 459–463

Delater A, Narayan N, Paech B (2012) Tracing requirements and source code during software development. In: ICSEA'12: 7th international conference of software engineering advances, pp 274–282

Duncan WR (2013) A guide to the project management body of knowledge (PMBOK® guide), 5th edn. Project Management Institute (PMI)

Dutoit AH, Paech B (2001) Rationale management in software engineering. In: Chang SK (ed) Handbook of software engineering and knowledge engineering, vol 1. World Scientific, Singapore

Dutoit AH, McCall R, Mistrik I, Paech B (eds) (2006) Rationale management in software engineering. Springer, Berlin

Falessi D, Becker M, Cantone G (2006a) Design decision rationale: experiences and steps ahead towards systematic use. SIGSOFT Softw Eng Notes 31(5)

Falessi D, Cantone G, Becker M (2006b) Documenting design decision rationale to improve individual and team design decision making. In: ISESE'06: international symposium on empirical software engineering. ACM Press, New York, pp 134–143

Falessi D, Cantone G, Kruchten P (2008a) Value-based design decision rationale documentation: principles and empirical feasibility study. In: WICSA 08: seventh working IEEE/IFIP conference on software architecture. IEEE, Vancouver, BC, pp 189–198

Falessi D, Capilla R, Cantone G (2008b) A value-based approach for documenting design decisions rationale. In: SHARK'08: 3rd international workshop on Sharing and reusing architectural knowledge. ACM Press, New York, pp 63–70

Falessi D, Cantone G, Kazman R, Kruchten P (2011) Decision-making techniques for software architecture design: a comparative survey. ACM Comput Surv 43(4):Article 33

Gervasi V, Gacitua R, Rouncefield M, Sawyer P, Kof L, Li M, Piwek P, De Roeck A, Willis A, Hui Y, Nuseibeh B (2013) Unpacking tacit knowledge for requirements engineering. In: Maalej W, Thurimella AK (eds) Managing requirements knowledge. Springer, Heidelberg, pp 23–48

Gethers M, Kagdi H, Dit B, Poshyvanyk D (2011) An adaptive approach to impact analysis from change requests to source code. In: ASE: 26th IEEE/ACM international conference on automated software engineering, Lawrence, KS, pp 540–543

Gethers M, Dit B, Kagdi H, Pashyvanyk D (2012) Integrated impact analysis for managing software changes. In: ICSE: international conference on software engineering. IEEE, Piscataway, NJ, pp 430–440

Gotel O, Finkelstein A (1994) An analysis of the requirements traceability problem. In: First international conference on requirements engineering, Colorado Springs, CO, pp 94–101

Hassan, AE (2008) The road ahead for mining software repositories. In: FoSM'08: frontiers of software maintenance, Beijing, pp 48–57

Helming J, David J, Koegel M, Naughton H (2009a) Integrating system modeling with project management - a case study. In: COMPSAC'09: international computer software and applications conference. IEEE Computer Society, Seattle, WA, pp 571–578

Helming J, Koegel M, Naughton H (2009b) Towards traceability from project management to system models. In: TEFSE'09: ICSE workshop on traceability in emerging forms of software engineering. IEEE Computer Society, Washington, DC, pp 11–15

Helming J, Koegel M, Naughton H, David J, Shterev A, Bruegge B (2009c) Traceability-based change awareness. In: MODELS'09: 12th international conference on model driven engineering languages and systems. Springer, Berlin, pp 372–376

Helming J, Arndt H, Hodaie Z, Koegel M, Narayan N (2010) Automatic assignment of work items. In: ENASE'10: evaluation of novel approaches to software engineering. Communications in computer and information science. Springer, Berlin, pp 236–250

Jansen A, Bosch J (2005) Software architecture as a set of architectural design decisions. In: WICSA'05: 5th working conference on software architecture. IEEE, Pittsburgh, PA, pp 109–120

Jansen A, Bosch J, Avgeriou P (2008) Documenting after the fact: recovering architectural design decisions. J Syst Softw 81(4):536–557

Jansen A, Avgeriou P, van der Ven JS (2009) Enriching software architecture documentation. J Syst Softw 82(8):1232–1248

Kagdi H, Poshyvanyk D (2009) Who can help me with this change request? In: ICPC: 17th international conference on program comprehension. IEEE, Vancouver, BC, pp 273–277

Kaushik N, Tahvildari L, Moore M (2011) Reconstructing traceability between bugs and test cases: an experimental study. In: WCRE: 18th working conference on reverse engineering, Limerick, pp 411–414

Kitchenham B, Charters S (2007) Guidelines for performing systematic literature reviews in software engineering. Technical report EBSE-2007-01. School of Computer Science and Mathematics, Keele University

Klein G (2008) Naturalistic decision making. J Hum Fact Ergon Soc 50(10):456–460

Könemann P (2009) Integrating decision management with UML modeling concepts and tools. In: Joint working IEEE/IFIP conference on software architecture & European conference on software architecture. IEEE, Cambridge, pp 297–300

Kruchten P, Lago P, van Vliet H (2006) Building up and reasoning about architectural knowledge. In: Hofmeister C, Crnkovic I, Reussner R (eds) Quality of software architectures, vol 4214, Lecture notes in computer science. Springer, Berlin, pp 43–58

Lee J (1991) Extending the Potts and Bruns model for recording design rationale. In: ICSE: 13th international conference on software engineering, pp 114–125

Lindvall M, Rus I (2003) Knowledge management for software organizations. In: Aurum A, Jefferey R, Wohlin C, Handzic M (eds) Managing software engineering knowledge. Springer, Berlin, pp 73–94

Lipshitz R, Klein G, Orasanu J, Salas E (2011) Taking stock of naturalistic decision making. J Behav Decis Mak 14(5):331–352

López C, Codocedo V, Astudillo H, Cysneiros LM (2012) Bridging the gap between software architecture rationale formalisms and actual architecture documents: an ontology-driven approach. Sci Comput Program 77(1):66–80

MacLean A, Young RM, Victoria ME, Moran TP (1991) Questions, options, and criteria: elements of design space analysis. Hum Comput Interact 6(3–4):201–250

Maeder P, Egyed A (2011) Do software engineers benefit from source code navigation with traceability? - an experiment in software change management. In: ASE: 26th IEEE/ACM international conference on automated software engineering, Lawrence, KS, pp 444–447

Maeder P, Gotel O (2012) Ready-to-use traceability on evolving projects. In: Cleland-Huang J, Gotel O, Zisman A (eds) Software and systems traceability. Springer, New York, pp 173–194

Mentis HM, Bach PM, Hoffman B, Rosson MB, Carroll JM (2009) Development of decision rationale in complex group decision-making. In: CHI: 27th international conference on human factors in computing systems. ACM, New York, pp 1341–1350

Menzies T, Zimmermann TH (2013) Software analytics: so what? Guest editor introduction, special issue. IEEE Softw 30(4):31–37

Ngo T, Ruhe G (2005) Decision support in requirements engineering. In: Aurum A, Wohlin C (eds) Engineering and managing software requirements. Springer, Berlin

Nguyen THD, Adams B, Hassan AE (2010) A case study of bias in bug-fix datasets. In: WCRE: 7th working conference on reverse engineering, Beverly, MA, pp 259–268

PMI (2013) Software extension to the PMBOK guide, 5th edn. IEEE Computer Society, Project Management Institute (PMI)

Rockwell J, Grosse IR, Krishnamurty S, Wileden JC (2009) A Decision Support Ontology for collaborative decision making in engineering design. In: International symposium on collaborative technologies and systems. IEEE, Baltimore, MD, pp 1–9

Saaty T (1990) How to make a decision: the analytic hierarchy process. Eur J Oper Res 48(1):9–26

Skerrett I (2011) The eclipse foundation. The eclipse community survey 2011. http://www.eclipse.org/org/press-release/20110610_survey.php

Smith JL, Bohner SA, McCrickard DS (2005) Project management for the 21st century: supporting collaborative design through risk analysis. ACM Southe Conf 43(2):300–305

Sureka A, Lal S, Agarwal L (2011) Applying Fellegi-Sunter (FS) model for traceability link recovery between bug databases and version archives. In: APSEC: 18th Asia Pacific software engineering conference, Ho Chi Minh, pp 146–153

Tang A, Babar MA, Gorton I, Han J (2006) A survey of architecture design rationale. J Syst Softw 79(12):1792–1804

Tang A, Jin Y, Han J (2007) A rationale-based architecture model for design traceability and reasoning. J Syst Softw 80(6):918–934

Tang A, Avgeriou P, Jansen A, Capilla R, Babar MA (2010) A comparative study of architecture knowledge management tools. J Syst Softw 83(3):352–370

Tyree J, Akerman A (2005) Architecture decisions: demystifying architecture. IEEE Softw 22 (2):19–27

Wang W, Burge JE (2010) Using rationale to support pattern-based architectural design. In: SHARK'10: ICSE workshop on sharing and reusing architectural knowledge. ACM Press, New York, pp 1–8

Yadla S, Hayes JH, Dekhtyar A (2005) Tracing requirements to defect reports: an application of information retrieval techniques. Innov Syst Softw Eng 1:116–124

Zannier C, Maurer F (2006) Foundations of agile decision making from agile mentors and developers. In: Abrahamsson P, Marchesi M, Succi G (eds) Extreme programming and agile processes in software engineering. Springer, Berlin, pp 11–20

Zannier C, Chiasson M, Maurer F (2007) A model of design decision-making based on empirical results of interviews with software designers. Inf Softw Technol 49(6):637–653

Zimmermann O, Gschwind T, Küster J, Leymann F, Schuster N (2007) Reusable architectural decision models for enterprise application development. In: Overhage S, Szyperski CA, Reussner R, Stafford JA (eds) Software architectures, components, and applications. Springer, Berlin, pp 15–32

Biography Barbara Paech holds the chair "Software Engineering" at the University of Heidelberg. Since many years, her group is particularly active in the area of requirements, rational and quality engineering. Her research is often empirical and in close cooperation with industry. She has published a book on rationale management and is founding member of the International Requirements Engineering Board (IREB).

Alexander Delater is a Ph.D. candidate at the department of computer science at the University of Heidelberg, Germany. He is a member of the Software Engineering research group of Barbara Paech. His Ph.D. topic is achieving requirements-to-code traceability using work items from project management. Beyond traceability, his main research interests lie in the area of requirements engineering.

Tom-Michael Hesse is a Ph.D. candidate at the department of computer science at the University of Heidelberg, Germany. He is a member of the Software Engineering research group of Barbara Paech. His Ph.D. topic is employing documented decision knowledge to improve the software evolution process. Further research interests are knowledge representation and the integration of requirements engineering with other software engineering activities.

Chapter 8
Framework for Implementing Product Portfolio Management in Software Business

Erik Jagroep, Inge van de Weerd, Sjaak Brinkkemper, and Ton Dobbe

Abstract Whether a software product company takes up a project depends on the strategic decisions that are made with regard to an organization's products. A software project needs to fit strategic goals and enable an organization to realize a vision through its software products. Making decisions on a strategic level, however, requires information of several related topics including technological trends and the product's life cycle and surpasses the scope of an individual software project. Instead, these decisions are made on the level of the product portfolio. Product Portfolio Management (PPM) holds that an organization has to manage investment decisions over time following profit and risk criteria. Given the multitude of relevant topics and the interrelatedness between these topics, it has proven difficult to implement PPM processes in software businesses. To this end, we created the Portfolio Implementation Framework (PIF) consisting of (a) a competence model, giving an overview of the critical topics; (b) process-deliverable diagrams, which provide an implementation path for product portfolio management processes; and (c) a maturity matrix that comprises 32 capabilities, which should be realized during implementation. The maturity matrix also serves as an instrument for industry to assess, compare and improve portfolio management processes across organizations. The framework provides a holistic view on a step-by-step PPM process implementation and has proved its applicability in practice.

E. Jagroep (✉) • S. Brinkkemper
Department of lnformation and Computing Sciences, Utrecht University, Utrecht,
The Netherlands
e-mail: e.a.jagroep@uu.nl; s.brinkkemper@uu.nl

I. van de Weerd
Department of lnformation, Logistics and lnnovation, VU University Amsterdam, Amsterdam,
The Netherlands
e-mail: i.vande.weerd@vu.nl

T. Dobbe
UNIT4, Sliedrecht, The Netherlands
e-mail: ton.dobbe@unit4.com

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_8, © Springer-Verlag Berlin Heidelberg 2014

8.1 Introduction

This chapter deals with project management from the perspective of the software business. Rather than individual projects, entire product and project portfolios are considered to ensure business can continue in both the short and long term. With the introduction of the portfolio matrix by the Boston Consulting Group in the 1970s and extensions like the ones made by Higgins (1985), the topic of Product Portfolio Management (PPM) has enjoyed a prominent position on the scientific research agenda. The principle has been applied to different kinds of products, new portfolio matrices have emerged (Ward and Peppard 2006) and ongoing research has been conducted on different aspects of PPM. These include comparisons to methods for financial portfolios (Wind 1975), portfolio management methods combined with managing new product development (Cooper et al. 2001) and tools for portfolio evaluation (Cooper and Edgett 2008). Related to PPM, project portfolio management (PjPM) (Kittlaus and Clough 2006; Killen et al. 2008) and product life cycle management (Saaksvuori and Immonen 2008) are also identified as important processes.

A central discipline with regard to PPM is software product management, which is defined as the *"…discipline and role, which governs a product (or solution or service) from its inception to the market/customer delivery in order to generate the biggest possible value to the business"* (Ebert 2007). PPM has been identified as one of the four business functions with regard to software product management, next to product planning, release planning and requirements management (Bekkers et al. 2010). Though relatively less established than software project management, software product management has been confirmed as a key area within many software companies (Fricker et al. 2009). Ebert (2007) continues that a product manager should aim at having the right product mix and selecting the right projects to implement a specific strategy. The selected projects are the responsibility of the project manager, making him a key figure in helping to realize the organizational strategy. This strategic context is also discussed in terms of 'decisions' in Chap. 7.

The relation between portfolio management and software projects is elaborated upon further by Cooper et al. (2001) by identifying ailments of having ineffective portfolio management. These are

- Missing strategic criteria in project selection
- Selecting low-value projects due to deficient go/kill and project selection decisions
- Lack of focus resulting in too many active projects and thinly spread resources leading to increased time to market, poor quality of execution and decreased success rates.
- Selection of project based on politics, opinion and emotion rather than facts and objective criteria.

Given that PPM is defined as managing investment decisions over time following profit and risk criteria and concerns the strategic information gathering and decision-making across the entire product portfolio (Kittlaus and Clough 2006; Bekkers et al. 2010), PPM could help in overcoming the identified ailments. Portfolio management is about choosing which products should be introduced, developed or even discontinued and determining which are in line with the organizational strategy and remain profitable in the short and long term. In this light, it is not without reason that a product manager is projected as the 'mini-CEO' of a product (McNally et al. 2009). The project manager should be familiar with this through the 'portfolio management' perspective described in the Project Management Body of Knowledge (PMBOK).

This research aims at providing a means for implementing PPM processes within an organization. More specifically, the processes are to be structured to fit the processes of software businesses. However, a sole focus on PPM will not provide a complete solution. Given their interrelatedness, topics like market analysis, product life cycle management (PLM), partnering and contracting (Bekkers et al. 2010) and PjPM (Kittlaus and Clough 2006) should also be included. The main related topics that can be identified are PLM and PjPM and are further elaborated upon below.

PLM is defined as a business approach integrating people, processes, business systems and information to manage the complete life cycle of a product across enterprises (Lee et al. 2007). Considering the strategic scope of PPM and the scope of PPM in general, a connection can be made between the decisions that need to be made on the portfolio level and the information that is made available through PLM. Indeed, the core of product life cycle management is the preservation and storage of information relating to an organization's products and activities (Saaksvuori and Immonen 2008).

PjPM is the central management of one or several portfolios in terms of identification, prioritization, authorization, organization, realization and controlling of its associated projects (Stantchev et al. 2009). Looking back at the definition of PPM it is clear that projects surrounding a product are an example of the investment decisions that have to be made and that project managers have a key task in realising these projects in an efficient, qualitatively high and timely manner. To stress the importance, Stantchev et al. (Stantchev et al. 2009) state that in uncertain times, organizations focus even more on the effective allocation of scarce worldwide resources to their development projects.

The topics related to PPM have been researched in terms of defined capabilities (Bekkers et al. 2010), maturity models (Saaksvuori and Immonen 2008; Batenburg et al. 2006) and implementation trajectories (Batenburg et al. 2006; de Reyck et al. 2005), and in some cases, combinations of these topics are discussed (Cooper et al. 2001; Kittlaus and Clough 2006). However, no attempt has been made on creating a means of implementing PPM processes that encompasses all related topics. Note that this does not mean that the existing models are not correct. Rather, the main issue with only taking up one specific topic is that even though the results have a proven value in practice, to fully experience the benefits of PPM also, the implementation and operationalization of the other aspects have to be taken into account. Keeping this in mind, we have formulated the following research question for this research: *“How can product portfolio management be implemented in software businesses to enable the corporate strategy?”*

To answer this question, we propose the Software Product Portfolio Management Implementation Framework, or PIF in short, which consists of a competence model, process-deliverable diagrams (PDDs) and a maturity matrix. The framework provides a step-by-step process for implementing PPM processes and also addresses the interdependencies of these processes with the topics that are related to PPM. Thus, PIF provides a holistic view of PPM process implementation. The maturity comprises the capabilities associated with PPM and enables organizations to determine their maturity level within the different focus areas.

In the remainder of this chapter, we first present the followed research approach. In Sect. 8.3, we explain more the theory-building case study that has been performed. We continue with the presentation of PIF in Sects 8.4 and 8.5. This is followed by a theory-testing case study (Sect. 8.6) and the conclusions and recommendations for further research.

8.2 Research Approach

The applied research methodology can be described as design science (Hevner et al. 2004) since new artifacts are created with the purpose of fulfilling an identified business need, namely, the need for directions on implementing PPM processes. The artifact in this research is the aforementioned PIF containing a competence model, PDDs and a maturity matrix. During the individual activities in the proposed method, the five phases of design research, awareness of the problem, suggestion, development, evaluation and conclusion (Vaishnavi and Kuechler 2004; Takeda et al. 1990) have been applied. However, given their more explicit methodology, we follow the six-step methodology as defined by Pfeffers et al. (Pfeffers et al. 2007). The problem motivating our research (1) is the question on how to implement PPM within software businesses in a holistic manner. In this specific case, *holistic* means that related topics are also taken into account in the solution. Given that PIF includes a competence model, focus area maturity matrix and PDDs that provide an implementation path for this purpose, the objective of our solution (2) is to provide a well-founded framework for the step-by-step implementation of PPM processes.

The design and development (3) of PIF is drawn from both literature on PPM and practice of PPM and related topics. Literature analysis helped in defining the areas, identifying the related topics that are of importance and thereby providing a solid base, whereas the practical aspect helped to confirm, adjust or extend the findings from literature. The first steps in this stage resulted in a preliminary PIF. By performing case studies, we provided ourselves with a complete set of information to construct the final PIF. By applying PIF in case studies, the framework has been demonstrated to be able to be applied in practice (4). However, since only two case studies have been performed, further demonstration of PIF must take place by applying PIF in the field. The evaluation (5) of PIF is done through case studies and expert interviews, where during the construction of PIF the methods as described by Bekkers et al. (2010), Weerd I van de (2009) and van Steenbergen et al. (2010) have been followed. Publication in scientific conferences and journals as well as practitioner outlets covers the communication regarding PIF (6).

The activities that comprise this research are depicted in Fig. 8.1. The figure shows the research activities that have been performed in the form of rounded rectangles on the left-hand side and the products of these activities in the form of rectangles on the right-hand side. The arrows indicate the flow of the research, in this case the order in which the activities have been performed. The first activity included a literature analysis and interviews. We first performed a thorough literature analysis and conducted interviews to get a clear scope in the area of PPM. As a result of this study, an initial set of guidelines was created. These guidelines were based on best practices, lessons learned and existing methods and models and form the foundation of the framework.

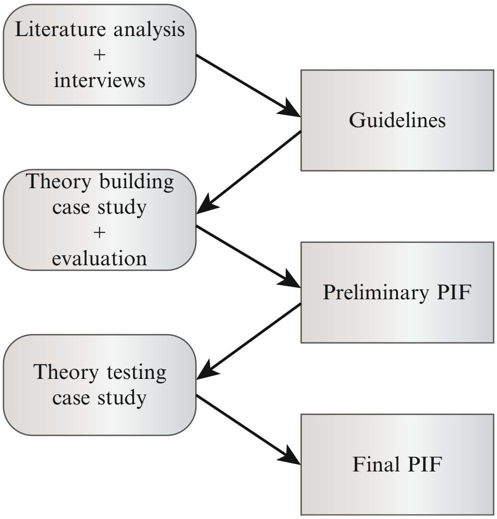
The next step was the theory-building case study at COMP1, a large Dutch software organization. The results from this case study combined with the aforementioned guidelines formed the basis for the preliminary PIF. The final activity comprised a theory-testing case study where the preliminary PIF has been applied at another large Dutch software organization (COMP2) in order to validate and refine the preliminary framework. This activity resulted in the final version of PIF. In the remainder of this section, a more thorough description of each of the research activities is presented. More details on COMP1 and COMP2 are given in the specific sections they apply to.

8.2.1 Literature Analysis and Interviews

In this stage, we used a literature study and interviews to find a direction for the solution of the described problem. During the literature study, apart from the theory on PPM itself, we also identified literature related to PPM. This resulted in a list of relevant PPM topics consisting of project portfolio management, product life cycle management, strategy, information requirements, information technology and tools. In the second step of the literature analysis, the identified topics were further investigated, and a list of key issues was created that are of influence for PPM. The key issues were identified using three criteria:

1. The issues must be mentioned by the majority of the authors.
2. Issues should (logically) be linked to other important aspects to create a solid framework.
3. Issues should be linked to matrices and models available in literature.

Fig. 8.1 Research methods



Consider, for example, 'link projects to strategic objectives' as a key issue stemming from PjPM literature. This issue holds a direct relation with the further development of a product and thereby the overall status of the product portfolio. Inspired by grounded theory research (Charmaz 2006) using the principles of qualitative analysis (Denzin and Lincoln 2011), all key issues were coded to be able to categorize them and refer back to them at a later moment in the research. For this principle, two main activities are identified: coding and continuous comparison. During the analysis, key issues for each topic have been identified, coded and categorized. Through the activity of continuous comparison, relations between the categories and literature have been identified. The codes, presented in Jagroep et al. (2011a), consist of three parts showing respectively the source of the key issue, the section and a numbering. For example, 'iv-plm-1' indicates the first key issue stemming from interviews on the topics of PLM, compared to 't-ppm-2', which indicates the second key issue stemming from the theory on the topic of PPM.

The second part of this phase consisted of interviews conducted at COMP1 to research the state of affairs of product portfolio management practice at this organization. Considering the explorative nature, a semi-structured interview was used for two reasons: (1) to question the topics that had been identified during the literature analysis and (2) to leave room for the interviewees to discuss other topics relevant to PPM. In total, 13 interviews were held with product managers situated in the Netherlands, The United Kingdom, Sweden and Spain. As with the literature analysis, the results of the interviews were transformed into key issues that in turn were coded following the scheme presented earlier.

8.2.2 Theory-Building Case Study and Evaluation

In this stage, we performed a theory-building case study and an evaluation, which resulted in the preliminary PIF. The first activity in this stage was the creation of an entity that would be used in the theory-building case study. Following the 'develop/build' and 'justify/evaluate' cycle (Hevner et al. 2004), we started with the creation of the competence model, building heavily on the continuous comparison activity as identified with the principles of qualitative analysis. In this model, an overview is presented on PPM and all topics related to PPM. To identify these topics, we built on the guidelines and categorized them into a set of topics that were to become part of the model. In most cases, this would be a one-to-one mapping of the exact topic, PLM would remain PLM, for example. However, the topic 'information requirements' was too vague to make part of the competence model. In these cases, the coded key issues were considered again to make a different, more concrete category. In this specific example, 'information requirements' resulted in 'gatekeeper introduction' and additions to the topics 'information technology' and 'process formalization'.

The second activity was to create the associated PDDs on the topics that had been identified in the competence model. In a PDD, the processes are modelled on the left-hand side and show the activities that should be performed in order to reach a desired goal. On the right-hand side, the corresponding deliverables for each activity are given. A deliverable is not limited to result from a single activity, though. For example, the activity of creating a central project administration should result in a corresponding deliverable (e.g., project database). However, this deliverable is adapted when the activity of defining separate portfolios is performed. During the process of constructing the PDDs, the main focus was to create a process that would enable an organization to cope with the identified key issues for that specific topic. The PDDs thus suggest an implementation path starting from the top of the diagram. Further information on PDDs can be found in Weerd I van de and Brinkkemper (2008).

With this initial framework, we conducted a case study where the framework was applied at COMP1. For this case study, the five major process steps as in Runeson and Höst (2009), (1) case study design, (2) preparation of data collection, (3) collecting evidence, (4) analysis of collected data and (5) reporting, were followed. The design of the case study has been sketched above; the objective is to create a framework for PPM process implementation. Given the focus on software products, the case has been selected to be a large Dutch software organization with the individual business units being the units of analysis (Yin 2009). The units of analysis have been selected based on the criteria that the persons involved had to be responsible for at least one product in the organizational portfolio.

The preparation of data collection and the analysis of the results are described in Jagroep et al. (2011a). For the actual collection of the evidence, the first-degree technique of interviews has been applied. With interviews, the researcher is in direct contact with the subjects and collects data in real time in Runeson and Höst

(2009). The interview itself is semi-structured, allowing for improvisation and exploration of the studied subjects where a pyramid model has been applied. With regard to reporting the research, the structure as proposed by in Runeson and Höst (2009) has been utilized.

Based on the results of the case study, the decision was also made to create a maturity matrix. The construction of this matrix followed after realizing that there is an increase in the maturity of the portfolio management processes when the PDDs are followed. This maturity increase, however, was not clearly visible in the competence model and PDDs. Thus, a maturity matrix was constructed to provide this clarity. In the matrix, maturity is shown in the form of capabilities, where a capability is defined as 'a demonstrable ability and capacity to perform a certain process at a certain level' (Weerd I van de 2009). A more detailed description of the maturity matrix is presented in Sect. 8.5.

The final activity performed in this phase was an evaluation of the artifacts. Specifically, an expert evaluation has been performed on respectively the PDDs and the maturity matrix. Four experts in the field of software product management (independent from the research) have participated by assessing the framework and proposing improvements through answering interview questions and discussion. These interview results have also been processed for the construction of the preliminary PIF.

8.2.3 Theory-Testing Case Study

For constructing the final version of PIF, a theory-testing case study has been performed. In this phase, PIF has been validated through a case study at COMP2, a company similar to COMP1 in terms of products and customer base, but with a different culture, philosophy and organizational traits. Given the fact that this case study is done separately from the theory-building case study, a separate plan for this case study has been developed based on Runeson and Höst (2009). Thus again, the five major process, being (1) case study design, (2) preparation of data collection, (3) collecting evidence, (4) analysis of collected data and (5) reporting, are followed for conducting this case study.

With regard to the design of the case study, the objective is to improve the theory on the existing preliminary PIF (Jagroep et al. 2011b) by confirming the theory or provide grounds for alterations. The case has been carefully selected to have an identical context as the previous work, namely, the Dutch software industry. In this sense, the case is the large Dutch software organization and the units of analysis are the individual business units that have participated in the case study (Yin 2009). The units of analysis have been selected based on the criteria that the persons involved had to be responsible for at least two products in the organizational portfolio. In comparison to the theory-building case study, these persons are active on a more strategic level, which fits the nature of product portfolio management. The theoretical background on this topic is provided by Bekkers et al. (2010) and

Jagroep et al. (2011b). The analysis of the collected data was kept identical to the theory-building case study for reasons of consistency. With regard to reporting the research, again the structure as proposed by in Runeson and Höst (2009) was applied.

8.2.4 Threats to Validity

With regard to the validity of the case studies that have been performed, we discuss the four aspects as proposed by Yin (2009). For the threats to construct validity, we deliberately made the starting point for this research a literature study. Based on the models and definitions that were found, a solid framework was created for PPM and the related topics. In the interviews for the case study, the definitions of the terms were taken up to ensure that there was a common understanding of the topics. All findings from the interviews were linked to the theoretical framework, and the chain of evidence was kept intact as presented in Jagroep et al. (2011a). Finally, the expert evaluation completed the cycle for triangulation. Also, the fact that the interviews for the theory-testing case study were conducted on the level of portfolio managers ensured that different views could be taken up in the research. The threats to the internal validity are taken up with the holistic approach that has been taken up in this research. Instead of focusing solely on PPM or a related topic, the complete set of topics has been taken into account. Therefore, the effects of factors that influence the process in one area could relatively easily be related to other areas when this seemed the case.

The external validity is purposefully limited to software products. Through multiple embedded case studies, though solely within the Dutch software industry, it became clear that the results could be applied to the organizations independent of one another. An interesting fact is that these organizations are also active on an international level; however, since this was not the main focus, no conclusions can be drawn from this. Also, since the framework is related to other literature and models that have been applied in practice, for example, Weerd I van de et al. (2006), the threats to external validity have been kept to a minimum. With regard to the reliability of the case studies, we set up a protocol (Jagroep et al. 2011a) that has been consistently applied in both case studies and encompassed the coding and analysis of the data.

8.3 Theory-Building Case Study and Evaluation

8.3.1 Theory-Building Case Study

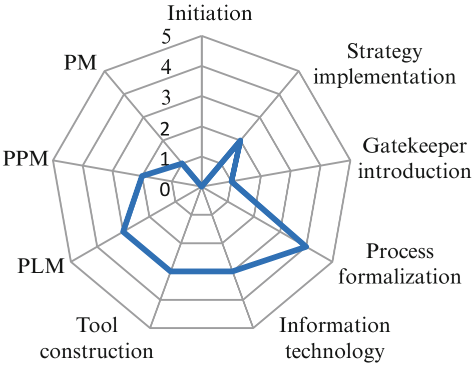
In the theory-building case study, performed at COMP1, the research artifacts have been evaluated to construct and apply the preliminary version of PIF. COMP1 is a large software company founded in the Netherlands and has over 4,000 employees in 24 countries worldwide. With a product portfolio of more than 100 products, COMP1 is considered an important player in the global software market. For this case study, 11 product managers from COMP1 were interviewed. These product managers were the units of analysis and were responsible for products in, amongst others, the ERP, bookkeeping, governmental, retail and healthcare sectors and were situated in business units in the Netherlands, Sweden, Spain and the UK.

In this stage, what has been later labeled 'preliminary PIF' consisted of an implementation model overview and the associated PDDs for the focus areas (Jagroep et al. 2011b). Based on the literature analysis, in total nine focus areas were identified, which are 'initiation', 'strategy implementation', 'gatekeeper introduction', 'PLM', 'PPM', 'PjPM', 'process formalization', 'information technology' and 'tooling'. Also, based on the literature analysis, these focus areas were in turn divided into three categories, being 'strategic processes', which guide an organization in realizing a fully operational implementation, 'core processes', the main processes concerning product portfolio management, and 'supporting processes', designed to facilitate the creation of instruments required for portfolio management.

As stated, the maturity matrix was constructed in a later stage of the case study, when it became apparent that the notion of maturity was not clearly represented in the framework. For the initial assessment, the PDDs were used and operationalized in the form of a spider graph (Fig. 8.2). Each participant described the processes, and a maturity level was determined by scoring the participant on a scale of zero (not mature) to five (most mature). This scoring was done based on the current position in the PDD. The spider graphs show each of the focus areas along with a score on each dimension based on their current activity according to the PDD. The assessments of the PPM processes resulting from the case studies have shown that the preliminary PIF could indeed be applied to software businesses. The capabilities in the matrix could be applied to both their departmental and organizational processes and provided a clear picture on the current status of their processes.

Based on the information that is contained in the spider graphs, the decision was made to construct a maturity matrix. For each of these focus areas, capabilities have been identified. These capabilities are based on the earlier-mentioned key issues, guidelines and existing maturity models. This information was also mapped on to the PDDs for each focus area to ensure that these artifacts would be compatible. In this sense, the two can be considered complementary; the capabilities represent the ability to perform an activity, and the PDDs show the actual activities and forthcoming deliverables required for implementing these activities. With regard to the dependencies between capabilities and their position in the maturity matrix, the PDDs proved useful as they indicate an order of implementation, prerequisites and other dependencies for the capabilities. A more detailed description is presented in Sect. 8.5.

Fig. 8.2 Maturity spider graph for PPM at business unit COMP1



8.3.2 *Expert Evaluation*

In this phase, the preliminary PIF artifacts (the competence model and PDDs) have again been evaluated. In total, four experts in the area of software product management have been interviewed using a semi-structured interview. The questions were aimed at evaluating the presentation of the framework, its completeness and the affordance. A question, for example, covered whether the experts found that there is a clear increase in maturity when the activities for each topic are performed. In total, six questions were formulated with the possibility of side-stepping when required. Before continuing to discuss the results, however, a remark should be made on the scope of the evaluation. Due to limited time, resources and availability of experts, the scope of the evaluation was limited to the core processes of the competence model. Though all topics are of importance, by limiting the scope the quality of the evaluation did not have to be compromised.

Based on the responses, a number of alterations were made to the framework that resulted in the form presented in this chapter. The first was on the view of the PDDs. In the earlier versions, the framework also encompassed activities that were considered as actions within the portfolio management process. The view that is taken up now is that of setting up the processes. Secondly, various alterations have been made to the framework with regard to PPM and PM. Amongst others, the activity of defining separate portfolios has been added. On the integration of these topics, an alteration was to let the project portfolio be managed by the portfolio review board instead of a separate project portfolio committee. For the full list of alterations, see Jagroep et al. (Jagroep et al. 2011a).

8.4 Software Product Portfolio Management Implementation Framework

8.4.1 Introduction

The Software Product Portfolio Management Implementation Framework, introduced as PIF, consists of a competence model (Fig. 8.3), PDDs and a maturity matrix (Sect. 8.5). For the construction of the final PIF, we followed the method as defined in van Steenbergen et al. (2010), where four process phases are identified, namely, (1) scope, (2) design model, (3) develop instrument, and (4) implement & exploit. In the scoping phase (1), performed during the literature analysis and interviews phase and the theory-building case study of the research methodology, we identified the topics that needed to be included in the framework. Going towards the design phase (2), the focus areas of the competence model were identified. These are 'Initiation', 'Strategy implementation', 'Gatekeeper introduction', 'Product Life Cycle Management', 'Product Portfolio Management', 'Project portfolio Management', 'Process formalization', 'Information technology' and 'Tooling'. Figure 8.3 shows how the focus areas are categorized according to their role within the process of implementing product portfolio management processes within an organization.

To provide more insight in the strategic value of PPM processes, Fig. 8.3 shows the focus areas in the context of an organization. For example, the board can expect information concerning the mid and long-term strategy and vision as output from the PPM processes. On a more operational level, however, the support department can expect to obtain product quality control information out of the PPM processes. This context has been added to the figure based on the results of the theory-building case study. Note, however, that this context overlaps with the stake- holders as identified in the PMBOK, again showing the interrelatedness between the subjects.

For the development of the instrument (phase 3), it is suggested to perform the activities of developing an assessment instrument and define improvement actions (van Steenbergen et al. 2010). The first is actually embedded in the capabilities that have been defined, considering that a yes/no question can be posed as to whether an organization possesses a specific capability. The improvement actions are suggested in the PDDs. At any point in time, an organization can fall back to the capabilities to determine what the next capability is and on the PDDs to determine the activities and deliverables required to reach that capability.

The final phase in this method, the implementation & exploitation phase (4), consists of implementing the framework, iteratively improving and communicating the results. Looking back at the research methodology (Fig. 8.1), it is clear that these activities are performed through applying PIF in case studies. The results of each case study contributed to finalizing PIF, making it the framework presented in this research.

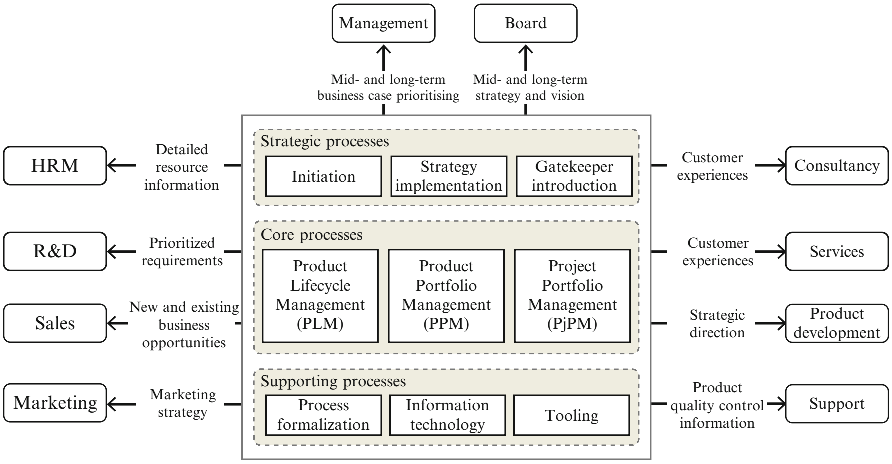


Fig. 8.3 Software product portfolio management competence model

With regard to the exploitation of the model by industry, PIF is positioned as a situational model for organizations, more specifically product software businesses, looking for improving their PPM processes. 'Situational' implies that not every organization is required to reach the highest level of maturity (Bekkers et al. 2010). Instead, each organization should decide what level of maturity best fits the organization. This activity is covered in the 'Initiation' focus area, which helps with forming the initial picture with regard to the desired PPM processes. From there on, the PDDs for the other focus areas can be followed as described above to actually implement the desired processes. Given the strategic nature of PPM, mostly high-level management is involved in deciding the strategic direction, involving both product and project managers in the process.

8.4.2 Process Descriptions for PIF

The process descriptions are modeled as PDDs. The activities are shown from the perspective of implementation and are, as explained, described on a high level to ensure their general applicability. It has to be noted that the actual usage of the framework is situational for each organization. Amongst others, differences in size, culture and organizational structure are aspects that need to be taken into account when implementing these processes.

In this section, the PDDs are explained in more detail, but due to limitations in space, only four processes are part of this chapter. These are the three core processes, of PLM, PPM and PjPM, shown respectively in Figs. 8.4, 8.5 and 8.6, and the gatekeeper introduction (Fig. 8.7). The full framework is explained in Jagroep et al. (2011a).

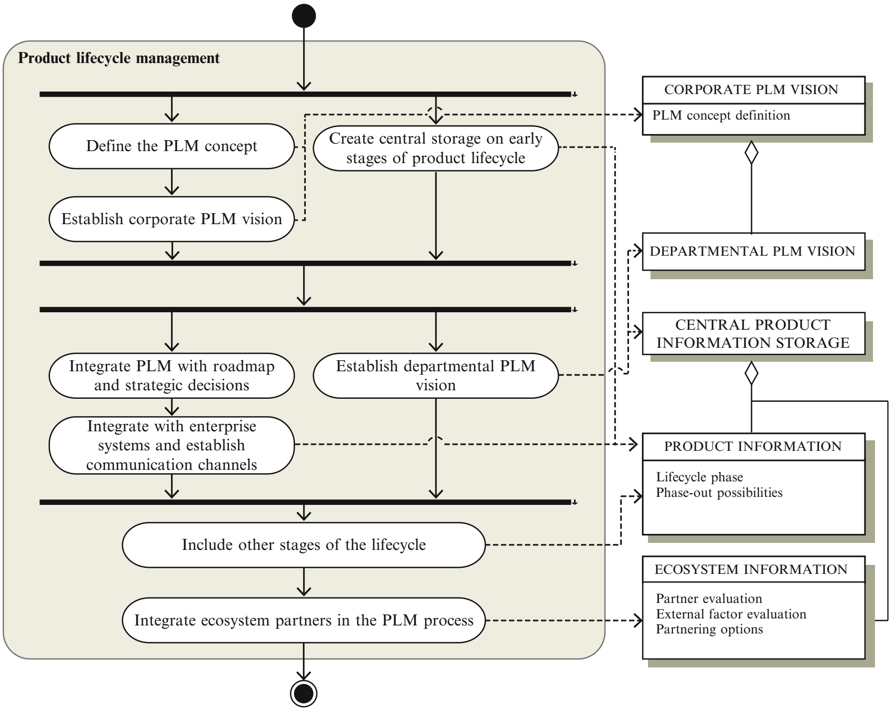

Product lifecycle management
Define the PLM concept
Create central storage on early stages of product lifecycle
Establish corporate PLM vision
Integrate PLM with roadmap and strategic decisions
Establish departmental PLM vision
Integrate with enterprise systems and establish communication channels
Include other stages of the lifecycle
Integrate ecosystem partners in the PLM process
CORPORATE PLM VISION
PLM concept definition
DEPARTMENTAL PLM VISION
CENTRAL PRODUCT INFORMATION STORAGE
PRODUCT INFORMATION
Lifecycle phase
Phase-out possibilities
ECOSYSTEM INFORMATION
Partner evaluation
External factor evaluation
Partnering options

Fig. 8.4 PDD for product life cycle management

8.4.2.1 Product Life Cycle Management

Product Life Cycle Management is the concept of preserving information on a company's products as they progress through their life cycles (Saaksvuori and Immonen 2008). As such, implementing PLM processes within an organization focuses on making this information available for decision-making. First, to ensure that the concept of PLM is familiar within an organization, a corporate vision is established, which in turn is used as a basis for departmental PLM visions. These visions define how PLM can contribute to the business processes. Parallel to defining these visions, a central product storage can be created containing information on the early life cycle phases (initiation, design, building, test & integration and release) of the product. As an organization usually has control over these early phases, the information on these phases is relatively easy to collect.

When this is established, the other phases of the life cycle (evolution and phase-out) can be included, which also encompasses the collection of information outside the organization. Also, parallel to defining the departmental visions are the activities to incorporate PLM into the strategic processes. As PLM encompasses the creation, preservation and storage of information relating to the company's products (Batenburg et al. 2006), PLM brings about critical information for effective

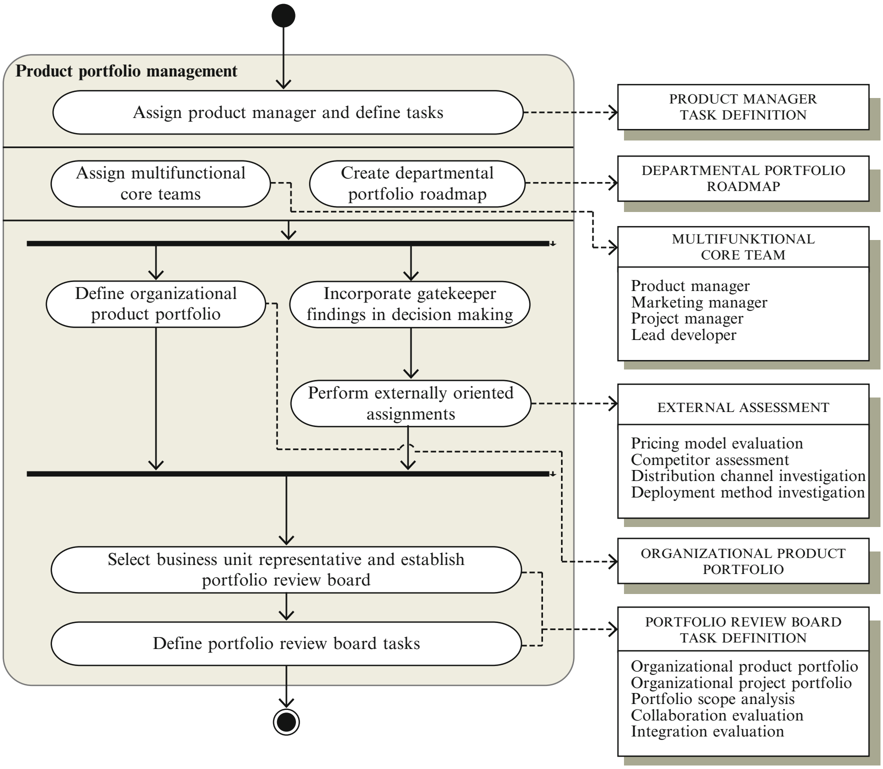

Product portfolio management
Assign product manager and define tasks
Assign multifunctional core teams
Create departmental portfolio roadmap
Define organizational product portfolio
Incorporate gatekeeper findings in decision making
Perform externally oriented assignments
Select business unit representative and establish portfolio review board
Define portfolio review board tasks
PRODUCT MANAGER TASK DEFINITION
DEPARTMENTAL PORTFOLIO ROADMAP
MULTIFUNKTIONAL CORE TEAM
Product manager
Marketing manager
Project manager
Lead developer
EXTERNAL ASSESSMENT
Pricing model evaluation
Competitor assessment
Distribution channel investigation
Deployment method investigation
ORGANIZATIONAL PRODUCT PORTFOLIO
PORTFOLIO REVIEW BOARD TASK DEFINITION
Organizational product portfolio
Organizational project portfolio
Portfolio scope analysis
Collaboration evaluation
Integration evaluation

Fig. 8.5 PDD for product portfolio management

strategic decision-making. With regard to projects, this could be information for the decision whether to delay or postpone a specific project in the portfolio.

As a final activity with regard to PLM, the integration with the ecosystem is suggested. In most cases, organizations produce and deliver their products to customers themselves. But it can also be the case that external parties are involved in these processes (e.g., development and distribution), which also possess information on the products and could influence their performance. The integration of the ecosystem is a means of getting insight into the impact a partner has on a specific product, being able to benchmark with others, enable collaboration and make the product life cycle more transparent (Batenburg et al. 2006; de Reyck et al. 2005).

8.4.2.2 Product Portfolio Management

In the PDD of Product Portfolio Management, the focus shifts from managing products on a departmental level towards managing the entire organizational portfolio. The reason for this is that product information for portfolio management

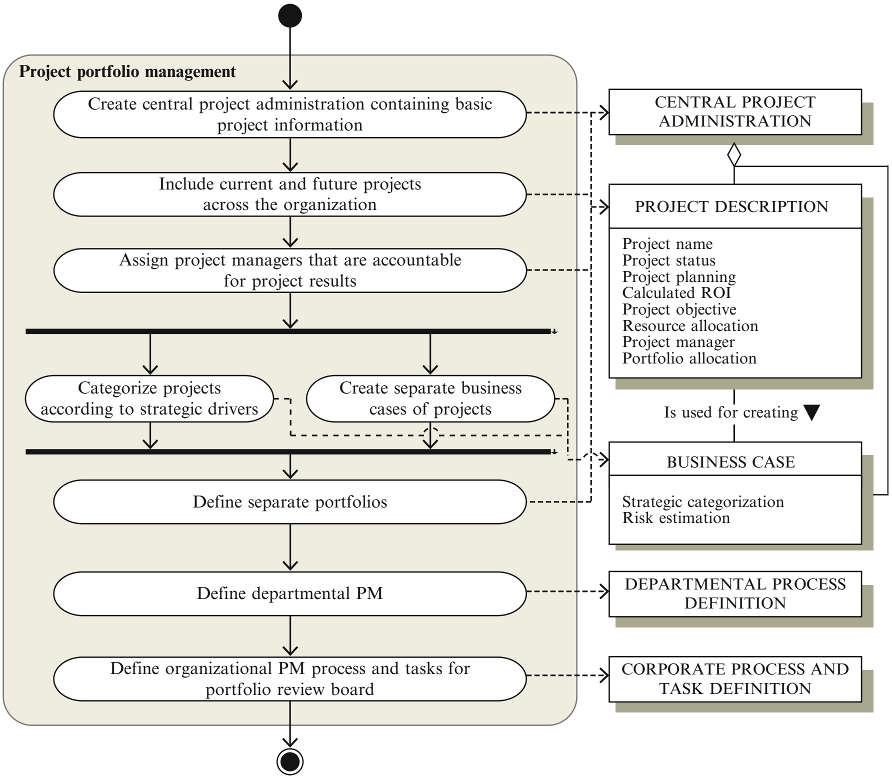

Project portfolio management
Create central project administration containing basic project information
Include current and future projects across the organization
Assign project managers that are accountable for project results
Categorize projects according to strategic drivers
Create separate business cases of projects
Define separate portfolios
Define departmental PM
Define organizational PM process and tasks for portfolio review board
CENTRAL PROJECT ADMINISTRATION
PROJECT DESCRIPTION
Project name
Project status
Project planning
Calculated ROI
Project objective
Resource allocation
Project manager
Portfolio allocation
Is used for creating ▼
BUSINESS CASE
Strategic categorization
Risk estimation
DEPARTMENTAL PROCESS DEFINITION
CORPORATE PROCESS AND TASK DEFINITION

Fig. 8.6 PDD for project portfolio management

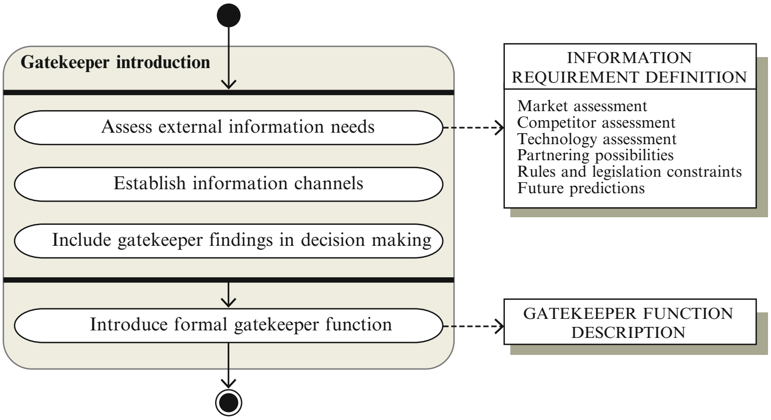

Gatekeeper introduction
Assess external information needs
Establish information channels
Include gatekeeper findings in decision making
Introduce formal gatekeeper function
INFORMATION REQUIREMENT DEFINITION
Market assessment
Competitor assessment
Technology assessment
Partnering possibilities
Rules and legislation constraints
Future predictions
GATEKEEPER FUNCTION DESCRIPTION

Fig. 8.7 PDD for gatekeeper introduction

is made available through the information that is stored on the departmental level. In other words, if the product information is not structurally stored and kept up to date on the departmental level, the information on the portfolio level will be inaccurate. Therefore, the first activities are aimed at facilitating this departmental management. This is done by defining the product manager's tasks as the 'mini-CEO' representing the business unit in strategy definition and operational execution (Ebert 2007). Also, having a clear view of the departmental portfolio and its future (in the form of a roadmap) and establishing a multi-functional team to get different perspectives on the individual product are activities to ensure the processes on the departmental level. These activities bring an organization on the third maturity level (product (line) orientation), as identified by Weerd I van de et al. (2006).

When the products are properly handled on the department level, the focus is expanded to the entire organization. More specifically, the organizational portfolio is defined, and externally oriented information is included by performing an external assessment. Defining the organizational portfolio contributes to having a common understanding of what exactly is a product, component, service and core asset (Bekkers et al. 2010) and simply having the portfolio documented in one format. The external assessment is the formalized procedure of ensuring that the individual products still fit the market. Note that gatekeeper findings (Jagroep et al. 2011a) are mentioned in the diagram as these are also externally oriented. From there on, the portfolio review board can be introduced consisting of high-level representatives of departments or business units. If proper information is provided, the organizational product and project portfolio can be truly managed across products and departments. Having proper portfolio management processes in place could also prove to have a positive influence on reaching level 4 in the success of a software project (see Chap. 2).

8.4.2.3 Project Portfolio Management

The next PDD is on the topic of Project Portfolio Management (Fig. 8.6). With regard to projects, the emphasis is put on ensuring a strategic fit and on managing projects across an organization. The suggested activities and resulting deliverables should lead to reaching the four goals of portfolio management as identified by Cooper et al. (Cooper et al. 2002). Note that in the case of PjPM, the focus again shifts from managing the projects on a departmental level, where the projects are executed under the supervision of project managers, to managing the entire project portfolio. Therefore, the first activity is to create a central project administration where basic information on current (active) and future projects is stored. As this could very rapidly become a large collection, the next activity is to bring structure in this collection by creating business cases of each project. Business cases are made on the basis of the strategic contribution to the organization and the estimated risk of each individual project. However, even with the business cases, projects with great strategic potential, for example, could end up on the bottom of the list when monetary gains are a more important criterion for prioritizing. To solve this issue, the suggestion is to create separate portfolios for each of the areas of interest. This separation could be considered as an appliance of the strategic buckets method (Cooper and Edgett 2010). For the individual project manager, this holds that a change of focus could be experienced with regard to managing individual projects; by applying this method, the strategic importance of projects will become more apparent. The risk management of a project (see Table 1.15), for example, could become a more stringent issue for the project manager to consider.

The next activity aims at defining and improving the departmental PjPM processes. When projects are taken up, resources are spent, and projects (especially in the case of software development) often take up a considerable amount of time. To get the most out of projects, the business should be enabled to make go/kill/hold/ fix decisions and 'ensure rational, accurate alignment with the business' (Rajegopal et al. 2007). When this is the case, an organization is able to quickly reallocate its resources to where they are needed the most. Put otherwise, project management would have to become more dynamic and the project manager more aware of the organizational environment and the stakeholders (see Sect. 1.3.10).

The final activity is realizing PjPM on an organizational level. First, a corporate PjPM process should be defined to ensure consistency in the manner that projects are handled. Different project managers have different management styles, which could inhibit an organization in harmonizing the organizational processes. Note that it is not implied that each project manager should operate identically compared to the others. Instead, a number of 'quality' criteria for the project management process could be established that guide the project manager on 'what' to do, without instructing 'how' to do so. Harmonization lowers the threshold for cross-departmental collaborations and is an area where the usage of social media could prove to be valuable (see Chap. 16).

Second, considering the overlap between PjPM and PPM, no companywide board is established for PjPM. Instead, the PjPM tasks are formally defined, such that they can be performed by the portfolio review board that has been established for PPM. Mind that, as with the other implementation activities, such a board should only be implemented when it fits the organization. With these activities, the organization can be placed in the portfolio optimization stage as identified by de Reyck et al. (2005). With this PDD, the management of individual projects is not elaborated upon considering the portfolio point of view of the framework. There is, however, a clear overlap with a number of knowledge areas as identified in the PMBOK. For example, 'project scope management' is aimed at ensuring that the right scope is set for both product and project. We consider the suggested strategic bucket method as a tool to categorize the scoped projects into project programs. The same holds for 'project cost management', where the proposed method could be used to better divide the available budget across projects.

8.4.2.4 Gatekeeper Introduction

The final PDD is on the topic of gatekeeper introduction (Fig. 8.7). Though gatekeeper introduction is not a core process, it is a relatively new item within the area of software product management. A gatekeeper is originally defined as 'an individual that maintains active communication with scientists at other firms, government laboratories, and universities' (Eisenhardt and Martin 2000; Vischer et al. 2010). This is the definition given in the context of dynamic capabilities, which are 'the organizational and strategic routines by which firms achieve new resource configurations as markets emerge, collide, split, evolve, and die' (Teece et al. 1997).

It is essential for the strategic decision-making process to also obtain information that resides outside of the organization. Therefore, there should be some notion on the external information needs and the information channels that are of importance. However, simply collecting information is not sufficient. The information that is collected should find its way to serve within the decision-making process. The final activity is on the actual introduction of the gatekeeper's role. The gatekeeper is the ideal person to keep track of advancements in the dynamic environment surrounding a software product. The gatekeeper can keep connections with a variety of external parties (e.g., partners, competitors, customers) and translate developments to concrete points of interest (or concern) for a software product or product portfolio.

8.5 Maturity Matrix for PPM

8.5.1 Structure

The maturity matrix (Table 8.1) is the research artifact where the element of maturity is most prominent. For reasons discussed by Weerd I van de (2009) and Bekkers et al. (2010), the choice was made to apply a focus area-oriented model. The matrix shows the nine focus areas (topics) identified in the leftmost column, their specific maturity levels in the form of capabilities, indicated by the letters A-F, and the maturity levels for the matrix, indicated by the numbers 0–10 in the top row. Each capability encompasses a measurement of maturity and can be used to determine the current situation of the PPM processes within an organization and as a reference guide towards the next stage in maturity. For example, when an organization states that capability A for PLM is implemented and B is not, it means that the maturity of this organization with regard to PLM is '1'. The next stage would be '2', which can be reached by implementing the B capability. The order of implementation is preferably from left to right where inter-process dependencies, dependencies between capabilities in other focus areas (Weerd I van de 2009), need to be taken into account.

Table 8.1 Maturity matrix for product portfolio management with example maturity profile

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Strategy processes											
Initiation	▨	▨	A	▨	B				C		
Strategy implementation	▨	▨	▨	A	▨	▨	B	▨	C		
Gatekepper introduction	▨	▨	▨	▨	A	▨	B				
Core processes											
Product portfolio management	▨	A	▨	▨	B	C		D	E		F
Product life-cycle management	▨	A	B		C		D				E
Product portfolio management	▨	▨	A	▨	B	▨	C		D	E	
Supporting processes											
Process formalization	▨	▨	▨	▨	▨	▨	A	▨	B		
Tooling	▨	▨	▨	▨	A	▨	▨	B			C
Information technology	▨	▨	▨	▨	A			B			C

Table 8.1 also shows an example of a matrix profile (grey shading), which represents the PPM maturity of a software business (van Steenbergen et al. 2010). In this particular example, most A-capabilities are implemented. However, in the focus area of PLM, none of the capabilities is implemented yet, leading to an overall maturity level of zero. In this case, the first improvement activity should be the implementation of the A and B capabilities with regard to PLM. Note that since the matrix is constructed in conjunction with the competence model and PDDs, they should be considered as complementary and referred to when a more thorough understanding is required.

8.5.2 Capabilities

The full list of capabilities that comprise the maturity matrix are presented online (see ‘Appendix J’ in Jagroep et al. (Jagroep et al. 2011a)). As an example, the capabilities for the PPM focus area are presented in Table 8.2. For each of the capabilities, a title is given, as well as a short description of the activity that needs to be performed within an organization. Consider capability B for the focus area of PPM. This capability is named ‘Constructed departmental roadmap’, and it is explained that there should be a document describing the future developments of the products in the portfolio over a certain period of time for a certain department within an organization. Since portfolio management is initiated on the product level, it should be clear what the future (development) projects (touching the topic of PjPM) on the departmental level are with regard to the products on the departmental level. This departmental roadmap could in turn be used as a means for communication of the plans with regard to a specific product.

Table 8.2 Capabilities for the PLM focus area

A	**Defined product manager tasks**	The tasks of a product manager are formally defined.
B	**Constructed departmental roadmap**	A document is created describing the future developments of the products in the portfolio over a certain period of time for a certain department within an organization.
C	**Established multi functional core teams**	A multi functional core team is responsible for the management and thereby the success of a product.
D	**Defined organizational portfolio**	The entire organizational portfolio is defined in terms of products, components and core assets, documented and communicated. All stakeholders (e.g., product managers) provide input for this mapping and validate the end result.
E	**External assessment is performed**	In conjunction with the gatekeeper function, a number of externally oriented aspects (e.g., pricing model and competitor advancements) are assessed for each product in the portfolio to improve the organizations' ability to tailor products to the markets' needs.
F	**Established portfolio review board**	A board is established that is actively managing the entire product and project portfolio of an organization, for example, through the usage of service catalogs.

With regard to the placement of the capabilities in the maturity matrix, we consider the focus area of PLM. The guidelines found for PLM suggested to first keep track of the life cycle phases that are under the control of the organizations and from there on expand to include the other life cycle phases. When the processes for an individual process are taken care of, the focus can shift to span the ecosystem of a product. As a consequence, these capabilities are placed on the left-hand side of the maturity scale. More difficult capabilities to implement, often stemming from the more mature situations in these models, are placed on the right-hand side. This process handles the intra-process, the order of the capabilities within a focus area, capability dependencies as described by Weerd I van de et al. (2008).

The inter-process capability dependencies (Weerd I van de et al. 2008), which is the order of the capabilities between focus areas, is explained using PLM capability C 'PLM integration in the product roadmap'. Looking at related focus areas, specifically PPM and PjPM, this capability, based on literature, requires that the tasks of the product manager are defined and that a central project administration is established. Therefore, the implementation of the PLM C capability depends on both A capabilities in the focus areas of PPM and PjPM and is consequently placed in a higher maturity level.

8.6 Theory-Testing Case Study

In this section, the validation of PIF is presented. First, a more thorough description is given of the case that has been studied and the results from this case study. These results confirm whether or not PIF can indeed be applied in practice and whether the framework and related theory are sufficient to assess the full scope of PPM. The section concludes with the adjustments that have been made to the research artifacts and the findings that have led to these adjustments. In this description, the maturity matrix is taken as the central object. Changes in the maturity matrix also imply changes in the other research artifacts.

As stated in the research methodology, the case study has been performed to validate the implementation framework for product portfolio management processes in the software business. To this end, the case study has been performed at a different, but nevertheless large, Dutch software organization (COMP2). The units of analysis for this case study were clusters within the organization, each with their own target markets, and their respective product portfolio's, where a cluster exists of a number of business units and multiple products. A cluster exists of a number of business units and multiple products. In total, seven interviewees have participated in the case studies, which were accountable for 32 software products. The choice to select interviewees on a higher strategic level was deliberately made as this fits the strategic nature of product portfolio management. The products that made up the portfolios of these clusters ranged from specialized software catered to a specific niche to large suites for demanding users. Examples are human resource software, administrative software and ERP systems deployed at, amongst others, local governments, healthcare institutions and retail and housing co- operatives. The average market share of these products is 47.6 %, with the highest market share being 71 % and the lowest being 20 %. With most of the products in the portfolio, the position of the market leader is claimed.

For this case study, the same research question as formulated for the entire research was the central question and, to conform to the protocol, the same semi-structured interview was used. The results of this case study have been accordingly analyzed and coded, as described in Sect. 8.2.1 and again processed to enrich the framework that was already created based on literature. Considering the fact that we already had the preliminary PIF, the interviewees got an explanation of the framework and the way their interview results would contribute to finalizing it. Note that this was done after the interview had finished and thus had no influence on the actual results of the interviews.

Table 8.3 shows the scores of the interviewees mapped onto the maturity matrix. The percentages are placed on the positions of the capabilities and indicate the part of the interviewees that have implemented a specific capability. It shows that the clusters have all implemented the A capabilities and in most cases also the B capabilities. The gray scale is a visual indication of the maturity of the organization. The dark gray indicates that the capabilities in this area are implemented within the clusters, while the shades towards lighter gray indicate that fewer clusters possess the capabilities according to the assessment. From the matrix, it is clear that the

Table 8.3 PPM maturity scores COMP 2

	0	1	2	3	4	5	6	7	8	9	10
Strategic processes											
Initiation			100%		43%				14%		
Strategic implementation				100%			86%		29%		
Gatekeeper introduction					100%		14%				
Core processes											
Product portfolio management		100%			100%	71%		43%	0%		0%
Product life-cycle management		100%	100%		14%		0%				0%
Project portfolio management			100%		100%		14%		0%	0%	
Supporting processes											
Process formalization							57%		0%		
Tooling					43%			0%			0%
Information technology					71%			0%			0%

main focus for a more mature process should be on capabilities regarding ‘Initiation’, ‘PLM’, ‘Tooling’ and ‘Information technology’. Overall, the maturity is set on level 3 since every cluster possesses the capabilities until that maturity level.

Apart from the application of PIF, also a number of changes have been made to the research artifacts based on this case study. These alterations are presented below and are structured according to the capability or focus area they apply to. Note that not every focus area required an alteration. As a general change, the core processes have been moved to the centre of the matrix for aesthetic reasons. The more detailed alterations are explained below:

Initiation—Capability C This capability has been moved from position 5 to position 8 on the maturity scale. During the interviews, all interviewees acknowledged that the portfolio plan is actually being constructed after the construction of the product roadmaps. It has been acknowledged that the product roadmaps actually form the input for the portfolio plan. Using centrally communicated themes from the high-level management, the strategic direction for the portfolio is set out.

Strategy Implementation—Capability C This capability has been moved from position 7 to 8 on the maturity scale. The portfolio strategy can be defined internally, but, as acknowledged by all seven interviewees, the external environment is a highly influential factor. Therefore, the new position for this capability is on the same level as capability E for product portfolio management.

Gatekeeper Introduction—Capability A Again, a capability has been moved, however this time, downwards on the maturity scale. All interviewees actually admitted to perform the functions of a gate keeper; they were in contact with the external environment and translated their findings into concrete directions for a product or the organization in general. With this finding, it was also indicated that tools and information technology actually depended on the information requirements that are defined by the gatekeeper process. Therefore, this capability is aligned with both A capabilities of the 'tooling' and 'information technology' process areas.

Product Portfolio Management With regard to this process area, the interviewees did not provide answers that led to alterations of the existing capabilities. However, five out of seven interviewees indicated missing the concrete activity of defining the organizational portfolio. Shifting from operational to more strategic levels requires that there is clarity on the entire product portfolio that the review board is made responsible for. Hence also the positioning of this new capability (D) before the external assessment is performed. This activity and deliverable have also been included in the PDD shown in Fig. 8.6, where the major part of the alterations have been made. The suggested order in the first PDD appeared too rigid, while all interviewees acknowledged that certain activities could indeed be performed simultaneously or that no particular order is required. More specifically, the figure shows that the activities on the departmental levels 'Create departmental portfolio roadmap', 'Assign product manager and define tasks' and 'Assign multifunctional core teams' can be performed simultaneously and in no particular order. Note also that the name 'Define product manager tasks and assign product managers' has been changed to 'Assign product manager and define tasks' for aesthetic reasons.

The final alteration is in the mid-segment of the PDD, where the focus slowly shifts towards the portfolio level. The alteration encompasses the simultaneous act of defining the organizational product portfolio and incorporating external information in the decision-making process. With regard to the order of first incorporating gatekeeper findings and then performing the externally oriented assessment, the interviewees indicated that gatekeeper findings triggered which external aspects should be further assessed.

Product Life Cycle Management For the process area of PLM, the capability of integrating PLM into the roadmap (C) has been added to the maturity matrix. Though all interviewees acknowledged that the life cycle of a product was an important factor to take into account, only one interviewee acknowledged that the information on the life cycle of a product was actually included in the roadmap for that product. Considering the importance of the information, this activity has also been made more explicit in the PDD (Fig. 8.5).

Other alterations in this process area are made in the PDD and are on the informative nature of the PLM process. The activity of actually integrating this information with the other (strategic) process in the organization ensures that the information is made available and being used within the organization. Also, a minor alteration has been made by replacing the supply chain of a product for the ecosystem surrounding a product. Based on Jansen et al. (Jansen et al. 2009) we consider an ecosystem to better fit the framework.

Project Portfolio Management Though no capabilities were altered, there are minor changes to the PDD with regard to PjPM (Fig. 8.6). The PDD has been changed to explicitly include the activity of defining the PjPM processes on a departmental level, an activity that was unclear with the previous description. Along with this alteration, the names of the corresponding deliverables were adjusted.

Tooling A process area that has been altered to a great extent is the area of tooling. First of all, the name 'tool construction' has been replaced by 'tooling'. This more generic term better suits the described processes in the PDD. Second, the capabilities have changed. The initial capability of having an established tool collection did not indicate the different strategic levels that four out of seven interviewees pointed out. To be more specific, it was acknowledged that tools were often used within the individual business units and that there was an awareness of their existence. Going one step further, two out of seven interviewees were involved in an endeavour to roll out a dedicated tool for product roadmapping. This tool can be used within a business unit, within a cluster or even throughout the organization.

The final capability goes to a more strategic level, namely, that of governance tooling. Given the fact that processes are implemented on different levels of the organization, in different environments and in different manners, there is the need to be able to keep track of their implementation and functioning. Governance tooling enables the organization to do so, but also requires significant effort to implement. With regard to the placement of the capabilities on the maturity scale, the same division as 'information technology' has been applied as three out of seven interviewees acknowledged the similarity in implementation difficulty and added value to the processes.

8.7 Implications

The portfolio implementation framework in this chapter is proposed as a contribution for the area of software product portfolio management. The importance of PPM is acknowledged within the scientific community, but no holistic method for implementing the associated process has been available. This is where PIF is suggested to fill the gap. Even though a clear distinction can be made between the suggested focus areas, there is an overlap when it comes to information requirements and process dependencies. For academic purposes, PIF is considered to be a starting point in maintaining this holistic approach. Though each focus area is interesting in its own regard, taking the interrelatedness into account is beneficial for being able to apply the results in practice. Though the focus of this research is on software, we do not argue that the framework is limited to this domain. Given the situational factors, (parts of) the framework could possibly be applied to other domains.

From a practical perspective, PIF provides a translation between scientific knowledge and practical appliance by making the topics more concrete and realizable in terms of processes. Knowledge on specific topics is available in scientific articles, but it is often the case that this knowledge cannot be applied in practice. PIF proved its applicability by case studies, and in one of these case companies, the profile of the maturity matrix has increased the awareness on this topic on the corporate level. This in turn led to the organization implementing the advice that was suggested based on PIF.

PIF is also a contribution in terms of an instrument for industry. As shown through the case studies that have been conducted, software organizations can be compared with one another using PIF. In this regard, PIF could prove to be a tool for bench-marking. The application of this framework in general could also lead to improved decision-making with regard to the product portfolio of an organization. PIF provides useful insight in the current state of affairs, which helps to raise the awareness on the subject and in turn possibly improve the portfolio as a whole. The starting point for improvement is the initiation focus area within the model and from there moving on to assess the other focus areas.

Related to software project management, PIF is positioned on a more strategic level. Though it is acknowledged that individual projects require proper management on their own, with regard to the success of a product, it is also important that the right strategic projects are taken up. This, amongst others, depends on the life cycle phase a product is in, the roadmap that is created for a product and the developments outside the organization (e.g., technological developments). Both practitioners and academics should consider a software project in the context of an entire organization and from this point of view maximize the value for business.

8.8 Conclusions and Future Research

The research question in this chapter, “How can product portfolio management be implemented in software businesses to enable the corporate strategy?” is addressed by the introduction of the software product portfolio management implementation framework (PIF). As there is a growing interest for product portfolio management with regard to software products, we believe that PIF is a good starting point for implementing and improving portfolio management processes within software organizations. Where maturity levels, methods and implementation trajectories have been defined for the individual focus areas identified in PIF, no model or framework has been available that incorporates a holistic view of these topics. Through extensive literature study and assessments of product portfolio management in practice, PIF is the first framework to incorporate a multitude of related topics within the domain of software products. With regard to PIF’s applicability in practice, two case studies were performed that indeed showed the potential of assessing and improving portfolio management processes in software businesses.

As an initial study into the implementation of product portfolio management processes within software businesses, further research is required to improve the research results. The research method that has been followed proved to be useful for exploring the research area and defining a new scope for PPM, but other aspects and relations could be added based on newer findings. Especially, considering the dynamics of business, business models and technology, PIF should be constantly adapted to fit these new situations when required. Further research could therefore be conducted on the content of PIF itself but also on the application, and forthcoming evaluation, of the framework. Practitioners should be aware of the fact that their input is essential when future frameworks and models are to better address their needs and help improve business.

With regard to the content that has been discussed, a number of directions can be thought of for future research. First, research could be conducted on the state of affairs at other software organizations. With more input from the practical perspective, more key issues, guidelines, activities, topics or even capabilities could be identified that make PIF more complete. Second, research could be conducted on the PPM processes of top performers to identify best practices. This could prove useful when the aim is to validate PIF as a benchmarking tool. And finally, experts from other domains, for example, project management could be involved in further development of the model. On the evaluation of the framework, further research could be conducted by performing more case studies where PIF is applied. Valuable insights can be gained into the completeness of the framework and its general applicability when organizations of different sizes are included. Ideally, every organization, depending on their situational factors, should be able to implement product portfolio management processes that fit the organizational needs.

Acknowledgements We would like to thank all (strategic) product managers within the two Dutch software organizations for their cooperation with the interviews and the experts who have participated in the evaluation of the framework. Furthermore, we want to thank the organizers, reviewers and participants of the 5th International Workshop on Software Product Management for their suggestions and remarks.

References

Batenburg R, Helms RW, Versendaal J (2006) PLM roadmap: stepwise implementation based on the concepts of maturity and alignment. Int J Prod Lifecycle Manage 1:333–351

Bekkers W, Weerd I van de, Spruit M, Brinkkemper S (2010) A framework for process improvement in software product management. Commun Comput Info Sci 99:1–12

Charmaz K (2006) Constructing grounded theory: a practical guide through qualitative analysis. SAGE, London

Cooper RG, Edgett SJ (2008) Maximizing productivity in product innovation. Res Technol Manage 51:47–58

Cooper RG, Edgett SJ (2010) Developing a product innovation and technology strategy for your business. Res Technol Manage 53:33–40

Cooper RG, Edgett SJ, Kleinschmidt EJ (2001) Portfolio management for new product development: results of an industry practices study. In: R&D management, Blackwell Publishers Ltd, pp 361–380

Cooper RG, Edgett SJ, Kleinschmidt EJ (2002) Portfolio management: fundamental to new product success. In: Belliveau P, Griffin A, Somermeyer S (eds) The PDMA ToolBook 1 for new product development, 1st edn. Wiley, New York, pp 331–364 de Reyck B, Grushka-Cockayne Y, Lockett M, Calderini SR, Moura M, Sloper A (2005) The impact of project portfolio management on information technology projects. Int J Project Manage 23:524–537

Denzin NK, Lincoln YS (2011) Handbook of qualitative research. Sage, Thousand Oaks, CA

Ebert C (2007) The impacts of software product management. J Syst Softw 80:850–861

Eisenhardt KM, Martin JA (2000) Dynamic capabilities: what are they? Strateg Manage J 21:1105–1121

Fricker S, Gorschek T, Byman C, Schmidle A (2009) Handshaking: negotiate to provoke the right understanding of requirements. IEEE Software 99

Hevner AR, March ST, Park J, Ram S (2004) Design science in information systems research. MIS Q 28:75–105

Higgins JM (1985) Strategy formulation implementation and control. Dryden Press, New York

Jagroep E, Weerd I van de, Brinkkemper S, Dobbe T (2011a) Software product portfolio management: towards improvement of current practice. http://www.cs.uu.nl/research/techreps/repo/CS-2011/2011-022.pdf. Accessed 25 March 2012

Jagroep E, Weerd I van de, Brinkkemper S, Dobbe T (2011b) Implementing software product portfolio management. In: Paper presented at the 5th international workshop on software product management, University of Trento, Trento, 30–30 August 2011

Jansen S, Finkelstein A, Brinkkemper S (2009) A sense of community: a research agenda for software ecosystems. Presented at the 31st international conference on software engineering, Vancouver, May 16–24

Killen CP, Hunt RA, Kleinschmidt EJ (2008) Project portfolio management for product innovation. Int J Qual Reliab Manage 25:24–38

Kittlaus HB, Clough P (2006) Software product management and pricing: key success factors for software organizations. Springer, Berlin

Lee SG, Ma YS, Thimm GL, Verstraeten J (2007) Product life-cycle management in aviation maintenance, repair and overhaul. Comput Ind 59:296–303

McNally RC, Durmusoglu SS, Calantone RJ, Harmancioglu N (2009) Exploring new product portfolio management decisions: the role of managers' dispositional traits. Ind Mark Manage 38:127–143

Pfeffers K, Tuunanen T, Rothenberger MA, Chatterjee S (2007) A design science research methodology for information systems research. J Manage Info Syst 24:45–77

Rajegopal S, McGuin P, Waller P (2007) Project portfolio management. Palgrave McMillan, Hampshire

Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empir Softw Eng 14:131–164

Saaksvuori A, Immonen A (2008) Product lifecycle management. Springer, Berlin

Stantchev V, Franke MR, Discher A (2009) Project portfolio management systems: business services and web services. In: Perry M, Sasaki H, Ehmann M. Bellot GO, Dini O (eds) Proceedings of the 4th international conference on internet and web applications and services, Venice/Mestre, 2009

Steenbergen M van, Bos R, Binkkemper S, van de Weerd I, Bekkers W (2010) The design of focus area maturity models. In: Winter R, Zhao JL, Aier S (eds) Global perspectives on design science research. Lecture Notes in Computer Science, vol 6105. Springer, Heidelberg, p 317

Takeda H, Veerkamp P, Tomiyama T, Yoshikawam H (1990) Modeling design processes. AI Magazine 11:37–48

Teece DJ, Pisano G, Shuen A (1997) Dynamic capabilities and strategic management. Strateg Manag J 18:509–533

Vaishnavi V, Kuechler W (2004) Design research in information systems. Association for information systems. http://desrist.org/design-research-in-information-systems. Accessed 31 January 2011

Vischer M, Boutellier R, Breitenmoser P (2010) Implementation of a gatekeeper structure for business and technology intelligence. Int J Technol Intell Plan 6:111–127

Ward J, Peppard J (2006) Strategic planning for information systems. Wiley, West Sussex

Weerd I van de, Brinkkemper S (2008) Meta-modeling for situational analysis and design methods. In: Syed MR, Syed SN (eds) Handbook of research on modern systems analysis and design technologies and application. Idea Group Publishing, Hersey, PA, pp 38–58

Weerd I van de (2009) Advancing in software product management: an incremental method engineering approach. Dissertation, Utrecht University

Weerd I van de, Versendaal J, Brinkkemper S (2006) A product software knowledge infrastructure for situational capability maturation: Vision and case studies in product management. http://www.cs.uu.nl/research/techreps/repo/CS-2006/2006-008.pdf. Accessed 25 March 2011

Wind Y (1975) Product portfolio analysis: a new approach to the product mix decision. Wharton University of Pennsylvania. http://marketing.wharton.upenn.edu/documents/research/7401_Product_Portfolio_A_New_Approach.pdf. Accessed 20 May 2010

Yin RK (2009) Case study research: design and methods. Sage, Thousand Oaks, CA

Biography Erik Jagroep after finishing the Business Informatics master program at Utrecht University, Erik Jagroep became a PhD candidate at Utrecht University on the topic of sustainable software development. Besides this academic position, Erik is also consultant at a large Dutch software organization where his work is aimed at improving the product management processes. This combination enables bringing research into practice fulfilling both academic and industry needs.

Inge van de Weerd is an assistant professor at the Knowledge, Information and Networks group at the VU University, Amsterdam. She holds a PhD in Information and Computing Sciences from Utrecht University and was visiting scholar at the Tokyo Institute of Technology. Her research interests focus on the intersection of IT and organizations.

Sjaak Brinkkemper is full professor of Organization and Information at Utrecht University, the Netherlands. He leads a group of about 30 researchers, with a methodology of product software development, implementation and adoption and business-economic aspects of the product software industry as the main research themes. He has published 10 books and about 150 papers on his research interests: software production, meta-modelling and method engineering.

Ton Dobbe In his career within Unit4, Ton Dobbe has carried positions in marketing, both local and international product marketing and management. In his current role as Vice President Product Marketing, Ton is responsible for both product strategy and global product marketing of Unit4's strategic international ERP solution, with a special interest in managing the UNIT4 product portfolio.

Chapter 9
Managing Global Software Projects

Christof Ebert

Abstract Projects are increasingly distributed across different sites. But distributed teams and suppliers complicate communication and create numerous frictions. Over half of all distributed projects do not achieve their intended objectives and are then canceled. Traditional labor cost-based location decisions are replaced by a systematic improvement of business processes in a distributed context. Benefits are tangible, as our clients emphasize: better multisite collaboration, clear supplier agreements, and transparent interfaces are the most often reported benefits. There is a simple rule: Only those who professionally manage their distributed projects will succeed in the future.

This chapter summarizes experiences and guidances from industry in a way to facilitate knowledge and technology transfer. It looks to processes and approaches for successfully handling global software development and outsourcing and offers many practical hints and concrete explanations to make global software engineering (GSE) a success. Starting with the necessary foundations, the chapter indicates what solutions are available to successfully implement GSE. It underlines the basic concepts and practices of GSE with broad industrial experiences and also summarizes future trends in GSE. The chapter is based on an industry perspective taking into consideration the state of the practice to ensure direct transfer and applicability to distributed projects.

Some portions of the text and pictures have been first published in the book Ebert, C.: Global Software and IT: A Guide to Distributed Development, Projects, and Outsourcing. Wiley, USA, 2012. Used with permission. The chapter is based on an industry perspective, which means that we look primarily towards industry needs in GSE to ensure applicability to distributed projects.

C. Ebert (✉)
Vector Consulting Services GmbH, Ingersheimer Strasse 24, 70499, Stuttgart, Germany
e-mail: christof.ebert@vector.com

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_9, © Springer-Verlag Berlin Heidelberg 2014

9.1 Introduction

Successfully managing global software projects has rapidly become a key need across industries. However, a majority of such global projects do not deliver according to expectations (Ebert 2012; PWC 2013). Globally distributed software development poses substantial risks to project and product management. As companies turn to globalized software, they find the process of developing and launching new products becoming increasingly complex as they attempt to integrate skills, people, and processes scattered in different places. Many companies realize after a while into global software engineering that savings are much smaller and problems are more difficult to solve than before. Disillusioned, many abandon their global software engineering (GSE) activities. What has gone wrong? GSE bears many chances and challenges, which we address in this chapter briefly from a project management perspective.

Global software engineering, IT outsourcing, and business process outsourcing during the past decade have shown growth rates of 10–20 % per year (Ebert 2012; PWC 2013; Forrester 2012). While the market proximity, cost advantages, and skill pool still look advantageous, global development and outsourcing bears a set of risks that come on top of the regular project risks. Not knowing them and not mitigating them means that soon your project belongs to the growing share of failed global endeavors. Global software engineering is as natural for the software business as is project management or requirements engineering. Going global with software is a means of effective work split and managing skills and competences. To be successful with software, we need to take advantage of continuous collaboration around the globe. This chapter looks into different business models in software.

The share of offshoring or globalization depends on the underlying business needs and on what software is being developed. While for mere IT applications or Internet services, the global development is fairly easy, embedded software still faces major challenges to distributed development. A study by embedded.com found that only 30 % of all embedded software is developed in a global or distributed context, while the vast majority is collocated (Ebert 2012; Chui et al. 2012; Forrester 2012). Similarly, the amount of quality deficiencies and call-backs across industries has increased in parallel to growing global development and sourcing.

Already in 1962, EDS started with offering IT on spare capacity (time-shared computing) as an external service (what would be today called application service provisioning). In 1976, EDS started deploying global IT services, such as financial accounting. India very early realized that this form of business could help the country leapfrog into current technologies and therefore become a major business partner for the Western world. Indian Institutes of Technology started in the 1960s with computer science curricula, thus providing the foundations for India's latter success in the IT domain. The first e-mail sent from China to a foreign country was on 20 September 1987 to the University of Karlsruhe, Germany. The text was short, yet powerful: “Across the Great Wall we can reach every corner in the world.” It was the vision of an increasingly connected world where all citizens and enterprises can do business with each other. The world was beginning to become flat. Yet, the notion of “across the wall” also shows that connected does not necessarily mean sharing the same values or being borderless and integrated.

The journey has begun, but it is far from being clear what the stable end positions will be. Clearly, some countries will come to saturation because global development essentially means that all countries and sites have their fair chance to become a player and compete on skills, labor cost, innovativeness, and quality. Software engineering is based upon a friction-free economy with any labor being moved to the site (or engineering team) that is best suitable among a set of constraints. No customer is anymore in a position to judge that a piece of software from a specific site is better or worse compared to the same software being produced somewhere else in the world. In essence, the old-economy labeling of “made in country x” has become a legacy thinking that does not relate to software industries. What counts are business impacts and performance, such as resource availability, productivity, innovativeness, quality of work performed, cost, flexibility, skills, and the like.

9.2 Foundations

Today, practically all new business plans contain offshoring as a key element to contain cost or achieve more flexibility to cope with changing demands in skills and numbers of engineers. Different business models are applied in the global context.

A first distinction is made between outsourcing and offshoring.

- Offshoring: Executing a business activity beyond sales and marketing outside the home country of an enterprise. Enterprises typically either have their local branches in low-cost countries or ask specialized companies abroad to execute the respective activity. Offshoring within the company is called captive offshoring.
- Outsourcing: A lasting and result-oriented relationship with a supplier, who executes business activities for an enterprise, which were traditionally executed inside the enterprise. Outsourcing is site-independent. The supplier can reside in the direct neighborhood of the enterprise or offshore.

Offshoring and outsourcing are two dimensions in the scope of globalized software development. They do not depend on each other and can be implemented individually. All global software development formats allow more flexibly managing operational expenses because resources are allocated at places and from regions where it is most appropriate to flexible needs and ever-changing business models. Figure 9.1 summarizes the reasons for GSE based on data from software companies in Europe and North America (Ebert 2012; PWC 2013).

Cost reduction is still the major trigger for globalization, though its relevance has been decreasing over the past years. This reasoning is simple and yet so

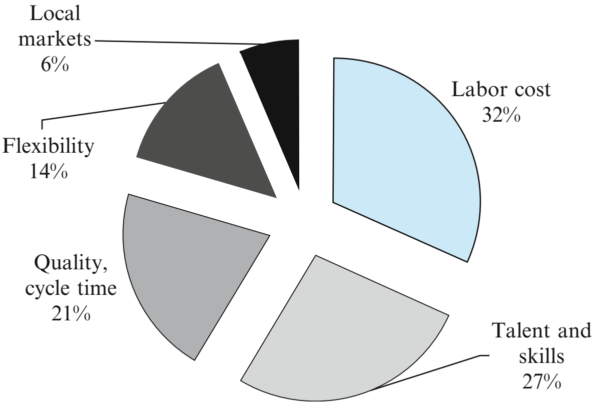


Fig. 9.1 Reasons for outsourcing and offshoring

effective that it is today mainstream for most companies and media. Labor cost varies across the globe. For similar skills and output, companies pay in different places of the world a different amount of money per working hour or per person-year. Looking to mere labor cost for the comparable skills of educated software engineers, several Asian countries offer a rate of 10–40 % of what is paid for the same working time in Western Europe or the USA. Salary differences theoretically allow reducing R&D labor cost by 40–60 % (PWC 2013; Greengard 2013; Chui et al. 2012; Forrester 2012). Obviously, insufficient competences, hidden costs, and many additional over-heads severely reduce this potential.

Other factors therefore start influencing the decision for GSE. Increasingly, global software development and offshoring is about proximity to markets, sharing the benefits of resources from different cultures and backgrounds, and flexibility in skill management. Managers want access to on-demand specialist know-how with less forecasting and provisioning, which often contains a lot of fixed cost that in today’s competitive environment is not anymore bearable. Increasingly, the target is quality improvement and innovation; both are related to blending cultures and thus achieve internal competition and a new stimulus for doing better.

The different drivers fueling the need for outsourcing and offshoring as they appear in Fig. 9.1 and in major statistics can be summarized in four categories, namely, efficiency, presence, talent, and flexibility. We will further look at these factors in the next section.

9.3 Benefits and Challenges

Working in a global context has advantages but also drawbacks. While the positive side accounts for time zone effectiveness or reduced cost in various countries, we should not close our eyes in front of risks and disadvantages. Practitioners of global software development and outsourcing clearly recognize that difficulties exist. In this chapter, we look into the risks and failures in global software projects. Only when we are aware of risks and past failures do we have a chance to do better.

Many factors cannot be quantified or made tangible initially, but will sooner or later heavily contribute to overall performance. For instance, innovation is a major positive effect that is boosted by going global. Engineers with all types of cultural backgrounds actively participate to continuously improve the product, how to innovate new products, and how to make processes more effective. Even with the slightly more complex decision-making process behind, achievements are substantial if engineers with an entirely different education and culture try to solve problems. Best practices can be shared, and sometimes small changes within the global development community can have big positive effects.

Not all companies that engage in GSE look at all four drivers with the same motivation. In several industry projects that I had been working on, we even faced a kind of trajectory where a vast majority of companies started with efficiency needs (i.e., cost savings), moved on to a presence in local markets, and, only after these two forces had been understood and digested, moved further to talent and flexibility. Also, it is clear that these four factors feed themselves. The more energy a company spends on building a regional pool of skilled software engineers, the more it also considers how to best utilize these competencies, for instance, to build a regional market or to develop new products for such markets.

It looks just rational to put stakeholders at one place, share the objectives, and execute the project. Working in one location is a major lesson learned from many failed projects that found its way into many states of the practice development methodologies, such as agile development. So, what are the strategies and tactics to survive globally dispersed projects?

In our experiences with clients around the world, 20–25 % of all outsourcing relationships fail within 2 years, and 50 % fail within 5 years (Ebert 2012). This is supported by a recent study that reports a trend towards localization and insourcing (Greengard 2013; Forrester 2012).

One fifth of the executives in a recent survey say that they are dissatisfied with the results of their outsourcing arrangements, while another fifth of the respondents see no tangible benefits (PWC 2013; Greengard 2013; Chui et al. 2012). Deloitte, considering responses from 22 primary industries in 23 countries, found that almost half of the respondents have terminated an outsourcing agreement early for cause of convenience. More importantly, one third of those who terminated a contract for these reasons chose to bring the work back in-house. In fact, insourcing has emerged as a viable option, particularly when outsourcing does not meet expectations (Greengard 2013). A Forrester study focusing on sourcing and vendor management sees "an ongoing level of dissatisfaction with outsourcing," citing Forrester's 2012 services survey of some 1,000 IT service professionals, where nearly half the respondents listed "poor service quality" as a challenge and one third stated that they were looking to bring work back in-house (Forrester 2012).

Working in a globally distributed project means overheads for planning and managing people. It means language and cultural barriers. In this chapter, we try to summarize experiences and to share the best practices from projects of different types and sizes that involve several locations in different continents and cultures.

The business case of working in a low-cost country is surely not a simple tradeoff of different costs of engineering in different regions. Many companies struggle because they just looked at the perceived differences in labor cost and not enough at risks and overhead expenses. Twenty percent of all globalization projects are cancelled within the first year (Ebert 2012; Greengard 2013). Fifty percent are terminated early (Ebert 2012; Greengard 2013). In many cases, the promise of savings has not matched the diminishing returns of unsatisfied customers.

Big savings in GSE have been reported only from (business) processes that are well defined and performed already before offshoring started and that do not need much control (Forrester 2012; Sangwan et al. 2007; Rivard and Aubert 2008). This includes maintenance projects (under the condition that the legacy software has some type of description) where some or all parts could be distributed, technical documentation (i.e., creation, knowledge management, packaging, translation, distribution, maintenance), or validation activities. Development projects have shown good results in all cases where tasks have been well separated so that distributed teams would have a direction and ownership.

In many companies around the world, I had seen global development projects failing when tasks were broken down too much, such as asking a remote engineer to do the verification for software developed concurrently in another site (Ebert 2012). In such cases, distance effects and a lack of direct communication slow down development rather than help it. The single biggest source of difficulties in outsourcing/offshoring is related to communication across sites, bad communication hindering both coordination and insufficient management processes.

For example, the continuous integration of insufficiently verified and encapsulated software components fails if done remotely to the parallel ongoing software development. Distributed teams working on exactly the same topic (e.g., the famous follow-the-sun pattern of developing a piece of software in different time zones) posed the highest challenges for coordination and often resulted in severe overheads that would be measurable or tangible only later (e.g., features misinterpreted, insufficient quality, lack of ownership and responsibility, etc.).

The challenges in global software can be summarized as follows:

- *Lack of strategy and shared values* in the parent organization resulting in insufficient collaboration and unclear work split and ownership. Roadmaps might be fragmented or insufficient visibility provided on the strategy that both contribute to an insecurity of teams and cause suboptimal results. A clear sign for the lack of strategy is the senior manager announcing, “We will work in India because it is cheaper,” or the engineering lead explaining, “Any work can be done by virtual teams.” A major underlying reason for dysfunctional global work is different values and underlying society factors. We often label this superficially as “culture issues” or, even worse, as “soft factors,” claiming that we cannot handle it with our limited management and software education. For instance, time perception in a society has a profound impact on many behaviors such as insufficient planning and monitoring, which cannot be cured only as symptoms. A culture deeply rooted in the present will always be portrayed as lazy and unfocused by a society rooted in the future and therefore demanding accurate planning. The same applies to societies that value entrepreneurship and spontaneous (re)actions to events as opposed to those that prefer clearly outlined roles and responsibilities.

Such differences must be recognized, considered, and dealt with. A shared value system and continuous team building activities will help, as well as rotating employees across these different societies.

- *Insufficient communication* due to distance, time zones, and cultural barriers. Note that distance impacts start at around 10–15 meters, that is, far earlier than what one would usually assume. People talk and share only if they are close to each other and frequently run into each other without this being planned. Lucent and others did extensive studies on communication in global teams and found that 15 % of software development is informal communication (Ebert 2012; Ebert and Dumke 2007; Carmel and Espinosa 2012). Distributed teams are less effective than a collocated team working on the same task.
- *Dispersed work organization.* The global nature of project and product work obscures a holistic view of project success factors. More sites add cost due to overhead management, separated and dysfunctional processes, tools, and teams. Tools help, but will not be enough to build a distributed team. Process immaturity is a key roadblock and a cause of inefficiencies and rework. Forrester, Gartner, BCG, and Standish report 10 % management overhead, that is, one person to synchronize for ten persons allocated an offshore task (Ebert 2012; PWC 2013; Forrester 2012; Sangwan et al. 2007). Our own experiences show that having two sites working on the same development project immediately adds 10–20 % of the cost while reducing visibility and the impacts of management. Overall effort overheads are ca. 35 % if working in two places due to interface control, management, replication, frictions, etc. (Ebert 2012; Ebert and Dumke 2007).
- *Inadequate global management* resulting in micromanaged tasks or a lack of visibility. Often, project managers would fear the lack of control and establish very small fragmented tasks to stay in control. Micromanagement creates a lack of buy-in from the teams as they expect that the manager would interfere anyway, so they do not have to pay attention. On the other end is insufficient visibility, starting with estimates and continuing with change management and progress tracking. Global team management often suffers from biased attitudes. Functional and regional rivalries exacerbate the tendency to claim the credit for success and shift the blame for failures. We experienced in several such global product lines that roadmaps and features are overly volatile because of local optimization on per regional customer basis. Our own experiences in many distributed projects over the past decade show that having two sites working on the same development project immediately adds 10–20 % of the cost while reducing visibility and the impacts of management (Ebert 2012). Alcatel-Lucent reports 30–100 % delays for multisite change requests and overall project delays

if a project is distributed across sites (Ebert and Dumke 2007). Shell underlines the relevance of strong global management for such global software projects (De Loof 2013).

- *Isolated learning*. Improvements derived from past experiences are rarely applied beyond the originating organizational silo. We found that in global software engineering individual sites – even if working in the same product line – often have their own mostly organically grown tools and processes. It is not that they don't like corporate standards; it is rather the desire of local management to optimize locally – because it is easier and faster than trying to convince headquarters in another region of the world. Different countries or regions would launch independent infrastructure optimizations in order to differentiate. This is often amplified by dysfunctional regional competition as many companies have been established to challenge "high-cost" countries with "low-cost" countries. For this reason, the parent organization would hesitate to provide all the necessary support due to fears that work would be taken away. Additional obstacles in sharing experiences arise from insufficient risk mitigation related to intellectual property or third-party access to tools and knowledge repositories. SAP reports, "Distributed development is slower and less forgiving in case of mistakes. We need to communicate more but we have less capacity to communicate. Effects of mistakes are not easily apparent and tend to be hidden by regional owners longer than possible in a centralized development" (Ebert 2012).
- *Less agility* compared to collocated teams. On one hand workflows, monitoring and engineering processes must be strengthened to assure that different stakeholders collaborate well. On the other hand, baselining reviews across sites, agreements on plans, stepwise synchronization of work products, multisite project reviews etc. all add to this very concrete overhead situation. Such tasks are normal and necessary as soon as an integrated activity is done in different sites. But they are perceived as overhead by the teams and if not well-trained they try to escape causing major trouble during development. Therefore, engineering and management workflows and tools must be as agile and seamless across interfaces and sites as possible. Modern product life-cycle management tools certainly reduce such overheads.
- *Insufficient contract management*. Contracts are absolutely crucial for managing external suppliers. They must include defined and measurable service level agreements (SLAs) to assure appropriate quality levels. For captive offshoring it might be wise, depending on organization structure, to govern by means of internal contracts and SLAs. They have the big advantage that targets and measurements are agreed on upfront and would not need continuous debates with the senior management if some delivery is late. Certainly, such internal contracts and SLAs together with a culture of accountability and clearly assigned responsibility also avoid the political game of finger-pointing at "the others" who did not do their job well.

- *Unknown legal environment.* This is a major trap for any global activity, independent of whether it is sales or engineering. To mitigate legal risks and exposure both local as well as central management must get very familiar with local laws, such as contracts, liability, intellectual property or human resource management. If you do not yet have enough experience with global development and specific regulations, we strongly recommend using consulting to ramp up your competencies and processes before you actually engage in global development. Never ever rely on the legal support from a supplier in the host country to which you want to expand your engineering teams.
- *High employee turnover rate.* Turnover rates are higher in offshore centers than in onshore ones—with the same job content (Ebert 2012). Reasons are manifold, but can be reduced to three factors, namely reduced motivation, culture clashes, and insufficient management. We see different local patterns of employee turnover rates across the world. Given the many opportunities for brilliant engineers, low motivation and insufficient rewards from their day-to-day work environment makes them search for another job. Often, it is simply the job content (e.g., doing only legacy repair, having only scattered assignments with low personal commitment and ownership) and a lack of career paths that make engineers move to another company. For instance, India's software industry is in such fast growth that many engineers are continuously approached by other companies with even more interesting offers. But, with additional effort and skilled management, turnover rates can be reduced. We have experienced that it is feasible to manage an Indian engineering team with turnover rates similar to those in Europe over many years (Ebert 2012). It all depends on management, culture, responsibilities, and, ultimately, motivation.

With these challenges, the reported cost reduction from GSE is much less than the abovementioned potential of 40–60 % savings if the same process is split across the world with changing responsibilities (Sangwan et al. 2007; Rivard and Aubert 2008). Successful companies reported from their global software projects a 10–15 % cost reduction after a 2- to 3- year learning curve. Initially, outsourcing demands up to 20 % additional efforts (Ebert 2012; PWC).

Figure 9.2 shows the average contribution of this hidden cost to the overall cost of R&D. For mere IT outsourcing, overheads are lower, specifically for management.

Externalizing insufficient engineering processes creates extra cost and learning curve-driven delays—on both sides. These additional costs sum up to 20–40 % of the regular costs of engineering (Ebert 2012; Forrester 2012; Ebert and Dumke 2007).

The learning curve for transferring an entire software package to a new team (e.g., location) takes 12 months (Ebert 2012; Sangwan et al. 2007; De Loof 2013; Ebert and Dumke 2007). The effectiveness of software design and coding grows in a learning curve with 50 % effectiveness reached after 1–3 months and 80 % after 3–5 months. This obviously depends on process maturity and technology

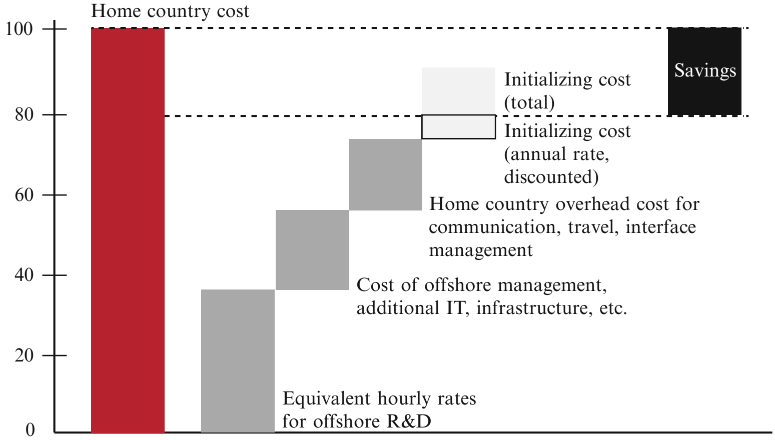


Fig. 9.2 Cost comparison including hidden cost

complexity. Each of the following bullets accounts for a 5–10 % increase in regular on-shore engineering cost in the home country (Ebert 2012; Ebert and Dumke 2007).

- Supplier and contract management
- Coordination and interface management
- Fragmented and scattered processes
- Project management and progress control
- Training, knowledge management, communication
- IT infrastructure, global tools licenses
- Liability coverage, legal support.

9.4 Global Software Development

Global software projects typically have some sort of supplier–client relationship, even if there is only one company with a captive development center. It is important for clients and suppliers to have shared processes and to maintain clear rules on collaboration regarding roles, interfaces, tools, work products, etc. Empirical studies highlight that success is higher when both the client and the supplier firms exhibit at least Capability Maturity Model Integration (CMMI) maturity level 3 (Rottman and Lacity 2006). Outsourcing insufficient engineering and management processes is a key reason for failed outsourcing projects (Ebert 2012; Sangwan et al. 2007). Insufficient processes are amplified as soon as distance and corporate boundaries add toward complexity. Figure 9.3 shows the mutual dependencies between supplier and client process maturities (Ebert and Dumke 2007).

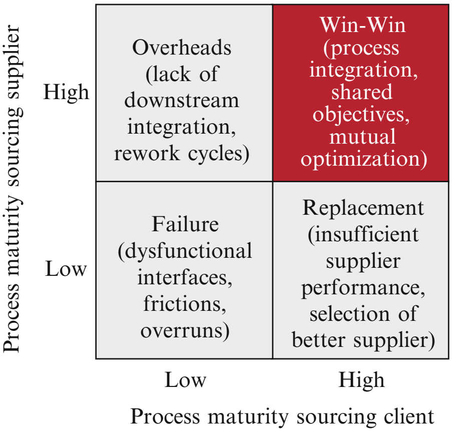


Fig. 9.3 Process maturity of suppliers and clients must match

Process maturity sourcing supplier		
High	Overheads (lack of downstream integration, rework cycles)	Win-Win (process integration, shared objectives, mutual optimization)
Low	Failure (dysfunctional interfaces, frictions, overruns)	Replacement (insufficient supplier performance, selection of better supplier)
	Low	High
	Process maturity sourcing client	

From our empirical research in different companies and GSE settings around the world, we can conclude that organizations with low organizational maturity (such as CMMI maturity level 1) should not expect that GSE would yield much benefit, except that an entire business process is given away (Ebert 2012). Instead, it will reveal major deficiencies in processes and workflow, which create all types of difficulties, such as insufficient quality, delays, additional cost, canceled offshoring contracts, demotivated workforce in both places (previous and new), and many more. The only viable alternative for such low-maturity organizations is to ramp up their own processes before proceeding with global software engineering.

Different societies—and often persons on the microscopic level—have different values and underlying driving factors. For instance, time perception varies dramatically across societies around the globe. Some focus on the past or present, while others are very future-oriented. Though this can be explained sociologically, such as the foreseeable or always surprising effects of nature on the destiny of a certain region of the world, it impacts behaviors. Therefore, the concept of urgency is different in such societies. Putting hard deadlines or considering a milestone as a deadline might work well in some societies, and fail without adequate training in others.

Administration and planning might be traditionally considered highly relevant, for example, in northern countries and in China (the first historically due to the need to plan for long winters, the latter due to thousands of years of highly sophisticated administrations) or almost irrelevant. Trust is another example. Some cultures do not care about written documents and take primarily the person and his word as the baseline. Others demand written documents and evidence in order to accept results. Knowing about such differences allows you to consider them in team building and setting a shared vision and shared values and objectives. Shared values and training on such different aspects of society is key in preparing the right development process and balancing the need for checkpoints with the level of acceptable and meaningful concrete deliverables. Needless to say, increasingly these society factors are reduced with growing globalization, as can be seen in the Indian software industry, which over the decades has adjusted extremely well to the northwestern way of planning and tracking.

Global development must balance managed processes with the flexibility to ease the work for individual engineers. Agility in a global context is of demand when engineers must act fast and don't have the time to baseline and stepwise sync around the globe's time zones. This is difficult and needs a profound understanding (driven from business rationales) of how to structure and tailor processes to avoid unnecessary overheads. Facilitate processes wherever possible, such as with standardized templates for work products or tools for workflow management. They reduce errors and improve productivity.

Global development benefits from chunking deliverables to self-contained work products that can be stepwise stabilized and integrated. It is based on the old Roman idea that self-contained pieces are easier to govern than a huge complex system (the so-called divide and conquer paradigm). This paradigm holds independent of whether you do maintenance on a big legacy system, application and service development, or the engineering of a new system. Incremental development and related life cycle models are known and applied for many years to address this "chunking" and stepwise stabilization.

Today, most life cycle models and development approaches are enhanced by agile methods. Increments toward a stable build are one of the key success factors in global development. They assure that deliveries from different teams or places in the world can be effectively integrated. Within periodic intervals, a validated baseline is made available for all team members, on which they build their enhancements or maintenance changes. Incremental development reduces delivery and quality risks because progress within the team is more continuous and can be more easily monitored. Utilize agile practices such as Scrum to build trust across sites and to ensure delivery in time, budget, and quality.

Traditionally, agile development methodologies have been seen as demanding small collocated teams (see also Chap. 11). This allows fast interaction between team members and, where necessary, immediate reaction and consideration of feature changes demanded by a customer. This seems to be in contradiction to the entire paradigm of global software engineering—at least for any distributed development. It is certainly true from a microscopic perspective: Do not split team tasks and responsibilities across sites if it can be avoided. For instance, if code is developed in one place and the unit tested in another, there is certainly the risk of inefficiency, misunderstandings, and inconsistencies. From the macroscopic point of view, distributed and global development is in line with the needs of agile development. It demands spliting the work in a way as to maximize team cohesion and minimize coupling. For instance, qualification testing or network integration in communication solutions can well be done at another place than the underlying application development. Requirements and business cases can be developed in different organizational and geographic layouts than the resulting designs and code.

In fact, hardware development since long has proven that with the right collaboration technologies, outsourced manufacturers can work with design teams at other physical places given that they have an understanding of sound and integrated engineering change management, product data management, and the like.

Within global engineering projects, it is often not so obvious how to implement an entirely incremental approach to architecture that is primarily driven by interacting classes or subsystems. Clearly, it would be advantageous to have isolated add-on functionality or independent components. In real-world systems, specifically in legacy systems that are decided to be maintained in low-cost countries, development at the top level (or architectural design) agrees not only on interfaces and impacts on various subsystems but also on a work split that is aligned with subsystems. The clash often comes when these subsystems should be integrated with all the new functionality. Such processes are characterized by extremely long integration cycles that do not show any measurable progress in terms of feature content.

9.5 Work Organization

Globally distributed software development is highly impacted by work organization and effective work split. Working in a globally distributed environment means over-heads for planning and managing people. It means language and cultural barriers. We provide some practical insights in this chapter about how to best organize work in a global setting.

Clearly, mixed teams with people from different countries, cultures, and, perhaps, companies stimulate innovation both in terms of products and technologies and in terms of more efficient collaboration. Teams that have worked together for a long time, such as departments inside a company, often struggle to identify really innovative solutions because they are captured within their traditional thinking schemes. As soon as external players are added to such a team, there is a new stimulus from outside, and it is less easy to simply wipe these ideas off the table.

Barriers of culture and people to global collaboration are not to be underestimated. They range from language barriers to time zone barriers to incompatible technology infrastructures to heterogeneous product line cultures and not-invented-here syndromes. It creates jealousy among the more expensive engineers, afraid of losing their jobs, while being forced to train their much cheaper counterparts. An obvious barrier is the individual profit and loss responsibility, which in tough times means primarily focusing on current quarter results and not investing in future infrastructures. Incumbents perceive providing visibility as a risk because they become accountable and more subject to internal competition.

Although there are no patent recipes for GSE work allocation, many experiences from previous projects indicate what we might call “typical configurations.” Such configurations are shown in Table 9.1.

The first column to the left indicates the “operational scenario” of global product development and operations. It starts with the beginning of the product (solution) life cycle and moves to installation and operation toward the bottom of the table.

Table 9.1 Global work allocation and typical configurations

Task	Business model	Supplier model	Learning curve	Break-even period	Number of partners / sites
Definition and analysis of new business models and processes; performance improvement	Preferable onshore; should be co-located	External consulting with own senior management	Long	Long	Few
Product definition; platforms and applications for resale	Onshore, close collaboration	External consulting; own senior management	Long	Long	Few
Development of internal (ICT) applications	Offshore	Typically outsourcing	Short	Middle	Few-many
Product development (generic)	On- / near- / offshore	Typically outsourcing or own development center	Middle	Middle-long	Few
Product development (embedded; complex)	On- / near- / offshore; single project should be co-located	Typically own development center	Middle	Middle-long	Few
Validation of software	On- / near- / offshore; tasks test and development should be co-located	Typically outsourcing or own test center	Middle	Middle	Few
Maintenance of internal applications	Offshore	Typically outsourcing or own development center	Middle	Middle-long	Many
Maintenance of products	Near- / offshore	Typically outsourcing or own development center	Middle	Long	Few
Selection and installation of software and infrastructure	On- / near-shore, close collaboration	Consultant; preferably own organization	Short	Short-middle	Few
Operation of infrastructure	On- / near-shore	Typically outsourcing or own IT center	Short	Short-middle	Few
Operation of internal applications	On- / near- / offshore	Typically outsourcing or own IT center	Short	Middle	Few

The second column shows the most appropriate business model for such an operational scenario. The next column indicates how external suppliers might be included. Obviously, external suppliers do not fit in to all scenarios, depending on intellectual property and dependency exposure but also related to risk management for future growth. The learning curve duration and the break-even period depend on these scenarios and are summarized in the subsequent columns. The last column finally portrays how many parties (external or internal) are most appropriate. Needless to say, most scenarios are most effectively handled with a small number of contributors—except such cases where the contribution can be well isolated and decoupled from overall project flow and risks (e.g., software components or platforms that are selected and evolve in parallel but without critical dependencies).

Effective work organization and resource allocation is key to successful global software development. There are two options for organizing global assignments, namely, virtual teams and collocated teams.

Virtual teams are set up with engineers from different parts of the world with a shared objective for the duration of the assignment (see also Chap. 10). They collaborate inside the team with high functional coherence. Virtual teams are created when skills are distributed and must cooperate toward an engineering product or design. The advantage certainly is the famous "follow the sun" approach of continuous engineering because one part of the team is almost always able to take up the work of another that just finished working hours. Evidently this works not for a setting with engineers in close time zones (e.g., North American companies working with engineers in South America, Western European companies working with engineers in Eastern Europe and India, Chinese, Japanese and Korean companies working with engineers in Vietnam).

The drawback of virtual teams is communication difficulties and the lack of team spirit because people do not know each other. Virtual teams need precisely allocated work packages and demand an overhead planning. They demand excellent collaboration tools beyond configuration management and document management. Continuous integration of the resulting code is a big advantage in virtual teams regardless of whether they work on new designs or maintenance tasks. It assures that team members in other places can continue with the same code and be sure that it is working when they start.

Collocated teams work at one place with a defined work assignment. They benefit from being together as a team and, thus, from simplified communication. Collocation means that team members should sit in the same building, perhaps the same room. From a mere people management perspective, this is of great advantage and can yield productivity gains of 30–50 % (Ebert and Dumke 2007). Being at one place, they can utilize standard engineering tools for configuration management or their shared documents, thus keeping the setup rather simple.

The difficulty in setting up such teams is that the necessary skills are not always available at one place. Often, such teams suffer from interface inconsistencies with their fellow teams working on different assignments in different places. Competition between teams could impact integration negatively. It is of benefit for collocated teams to establish clear quality gates and quality control activities (e.g., reviews, inspections, unit tests with defined exit criteria) to assure the right quality level when the resulting work products are passed on to other places in the world.

Independent of the team structure (i.e., virtual or collocated), we recommend using fully allocated team members and coherent assignments.

Coherence means that the work is split during development according to feature content, which allows assembling a team that can implement a set of related functionalities. The more coherence the work assignment has, the less dependencies and interactions occur with other teams that might work in different settings or even different places and time zones. Projects are, at their kick-off, already split into pieces of coherent functionality that will be delivered in increments to a continuous build. Coherent functional entities are allocated to development teams, which can be based in different locations. Architecture decisions, decision reviews at major milestones, and tests should be done at one place. Experts from countries with a minority contribution will be relocated for the time the team needs. This allows effective project management, independent of how the project is globally allocated.

Full allocation implies that engineers working on a project should not be distracted by different tasks in other projects. The more the allocation to a single task and shared objective within one team, the fewer engineers are distracted by disturbances and thus context switches. Full allocation does not mean 100 %, but should certainly be higher than 60 %. If tasks are too small, then related tasks should be allocated to the team. The difficulties usually start with very heterogeneous assignments such as working on two different products. In such cases, the context switching from one product to the other is highly dysfunctional and causes dramatic productivity loss.

These working principles directly impact productivity. Team members must communicate whenever necessary and without long planning and preparation to make the team efficient (Carmel and Espinosa 2012). Alcatel-Lucent, for instance, evaluated projects over 5 years and could distinguish according to the factor of collocation and the allocation degree (Ebert 2012). Collocated teams achieve an efficiency improvement of over 50 % during initial validation activities (Ebert 2012; Ebert and Dumke 2007). This means that with the same amount of defects in design and code, those teams that sit in the same place need less than half the time for defect detection. Allocation directly impacts overall project efficiency. It was found in the same long-term study that small projects with highly scattered resources would show less than half the productivity compared to projects with fully allocated staff. Cycle time is similarly impacted. People switching between tasks need time to adjust to the new job. In that same study, Alcatel-Lucent found an impact of a factor 2–3 compared to what is necessary if resources are allocated to one job during a window of 1 week upward.

Ensure that people work on few tasks or work packages—with the highest possible allocation. More tasks mean more interrupts and thus more defects and longer response times and ultimately reduced motivation. Enriching jobs in the way described above means also more training and coaching needs. We saw, however, in our own experiences over the past 10 years that coaching pays off. Looking only at the cost of nonquality, that is, the time to detect and correct defects, we found that projects with intensive coaching (ca. 1–2 % of accumulated phase effort) could reduce the cost of nonquality in the phase by over 20 % (Ebert 2012). A breakeven is typically reached at ca. 5 % coaching effort (Ebert 2012). This means that there are natural limits toward involving too many inexperienced engineers.

The higher the allocation, the more motivation and ownership you will gain from your global development teams. The major changes for a team moving to global development are concurrent engineering and teamwork. They need to be supported by the respective workflow techniques. We assemble cross-functional teams especially at the beginning of the project. Even before project kick-off, a first expert team is called to ensure a complete impact analysis that is a prerequisite to defining increments. Concurrent engineering means that, for instance, a tester is also part of the team, as experience shows that designers and testers look at the same problem very differently. Testability can only be ensured with a focus on test strategy and the potential impacts of design decisions already made during the initial phases of the project.

Based on a mapping of customer requirements to architectural units (i.e., modules, databases, subsystems, and production tools), global engineering activities can be treated according to their impact on architectural entities:

- Small independent architectural units that could be fairly well separated and left out of any customization. Typically, they are subject to moving into separate servers. Development is collocated at one place
- Big chunks that would be impacted in any project and thus need a global focus to facilitate simple customization (e.g., different signaling types can be captured

with generic protocol descriptions and translation mechanisms). Development happens in multiskilled teams. These skills are replicated in almost all locations.
- Market- or customer-specific functional clusters that would be defined based on the requirement analysis and ultimately form the project team responsible for a customer project. This type of requirement must be the exception and asks for a dedicated pricing strategy as it creates most overheads—but could be most interesting for our customers to differentiate

Such a separation of architectural units is the necessary precondition for splitting a global project into teams that can be individually collocated.

9.6 Risk Management in Global Software Projects

Globally distributed software development amplifies typical software project–and product-related risks, such as project delivery failures and insufficient quality. Worse yet, it creates new risks, such as inadequate intellectual property rights (IPR) management or lock-in situations with suppliers. These risks must be identified in due time and have to be considered together with the sourcing strategy and its operational implementation.

While the classic centralized software development approach allowed solving problems in the coffee corner or around the white board, global teams are composed of individuals who are culturally, ethnically, and functionally diverse. They work in different locations and time zones and are not easily reachable for a chat on how to design an interface or how to resolve a bug that prevents tests from progressing. This explains that, for instance, only 30 % of all embedded software is developed in a global or distributed context, while the vast majority is collocated. The reason is very simple: Embedded software poses a much higher risk on safety and reliability, and thus, companies prefer risk management in their own—known—environment, rather than adding risk through global teams. How can these risks be mitigated and be thus flexibility improved?

Risk management is the systematic application of management policies, procedures, and practices to the tasks of identifying, analyzing, evaluating, treating, and monitoring risk (see also Chap. 5). Global development projects pose specific risks on top of regular risk repositories and check lists. They relate to two major underlying risk drivers, namely, insufficient processes and inadequate management.

Not all eventualities can be buffered because in the global economy, developing and implementing products must be fast, cost effective, and adaptive to changing needs. Therefore, there is a need to utilize different techniques to effectively and efficiently mitigate risks. Governance and legal regulations play a crucial role in mitigating risks in global projects. Methods include using basic project, supplier, and quality management techniques; process frameworks such as CMMI (capability maturity model), ITIL (IT infrastructure library), COBIT® (Control Objectives for Information and Related Technology), product life cycle management, effective communication processes, SLA (service-level agreements)-based escalation, competence management, and innovation management.

Most countries today force companies with headquarters in that country or that are quoted at a local trading place to comply with rules on risk management. A good example is the Sarbanes-Oxley Act in the USA, which holds the CEO and CFO of such companies personally liable for providing correct information about the status of the company and for ensuring that the internal control and risk management system is working properly. Failing to prove compliance can result in lawsuits and severe punishment. Offshoring or outsourcing, therefore, must support these mechanisms for internal control and risk management. This can be translated to the following rules:

- Establish and maintain an efficient and effective control and risk management system that includes supplier management
- Enforce the internally applicable compliance rules also to suppliers so that full transparency can be maintained
- Provide transparency of the business processes and the resulting documentation and work products
- Document decisions with an impact on governance, compliance, and finance risks
- Ensure that industrial best practices are followed to effectively and efficiently mitigate risks, including the supply chain, as it has an immediate impact on finance performance and the legal exposure of a company.

The latter specifically applies also for mitigating software product liability risks, which in some countries can end in very costly lawsuits if, for instance, products create risks to public safety or health or even have already caused damages. From a legal perspective, best practices translate into consistently and auditably applying international standards, such as CMMI, COBIT, ITIL, etc.

Governance and compliance are the personal responsibility of a CEO or managing director and his finance deputy. It is enforced by a set of processes and checks, including periodic external audits.

Based on our project experiences, we have established a global software risk top10 list (Ebert 2012). Depending on the specific layout of global software (e.g., with or without an external supplier), the ranking list of these top 10 risks is as follows: project delivery failures; insufficient quality; distance and culture clashes; staff turnover (mostly for captive centers); poor supplier services (only for outsourced development); instability with overly high change rate; insufficient competences; wage and cost inflation; lock-in (only for outsourced development); and inadequate IPR management.

These risks can be clustered according to major drivers, which then allow selecting the most adequate mitigation strategy. Figure 9.4 shows the top 10 risks sorted on the four major drivers for global software.

Not all of these risks and suggested mitigation actions may be applicable to all organizations and scenarios. Wage and cost escalation will not be an issue for growing teams, as generally new recruits are at a lower cost than the existing

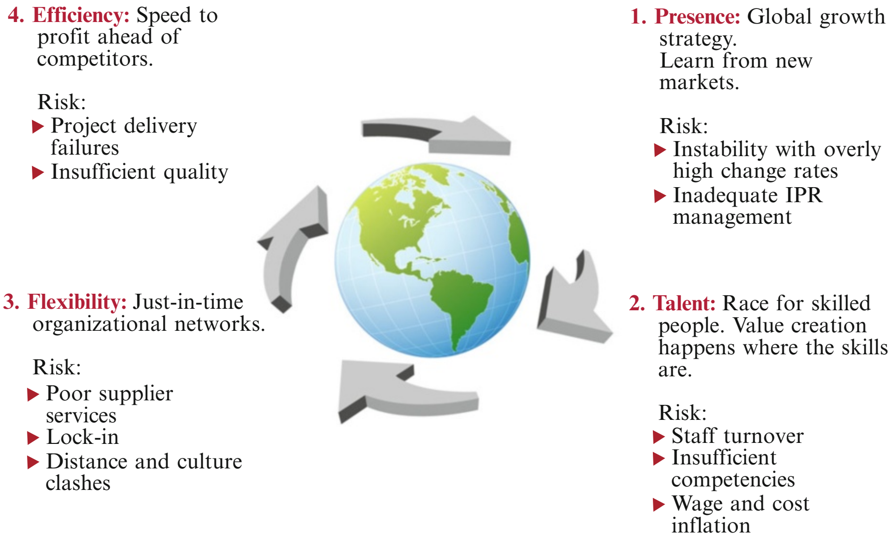
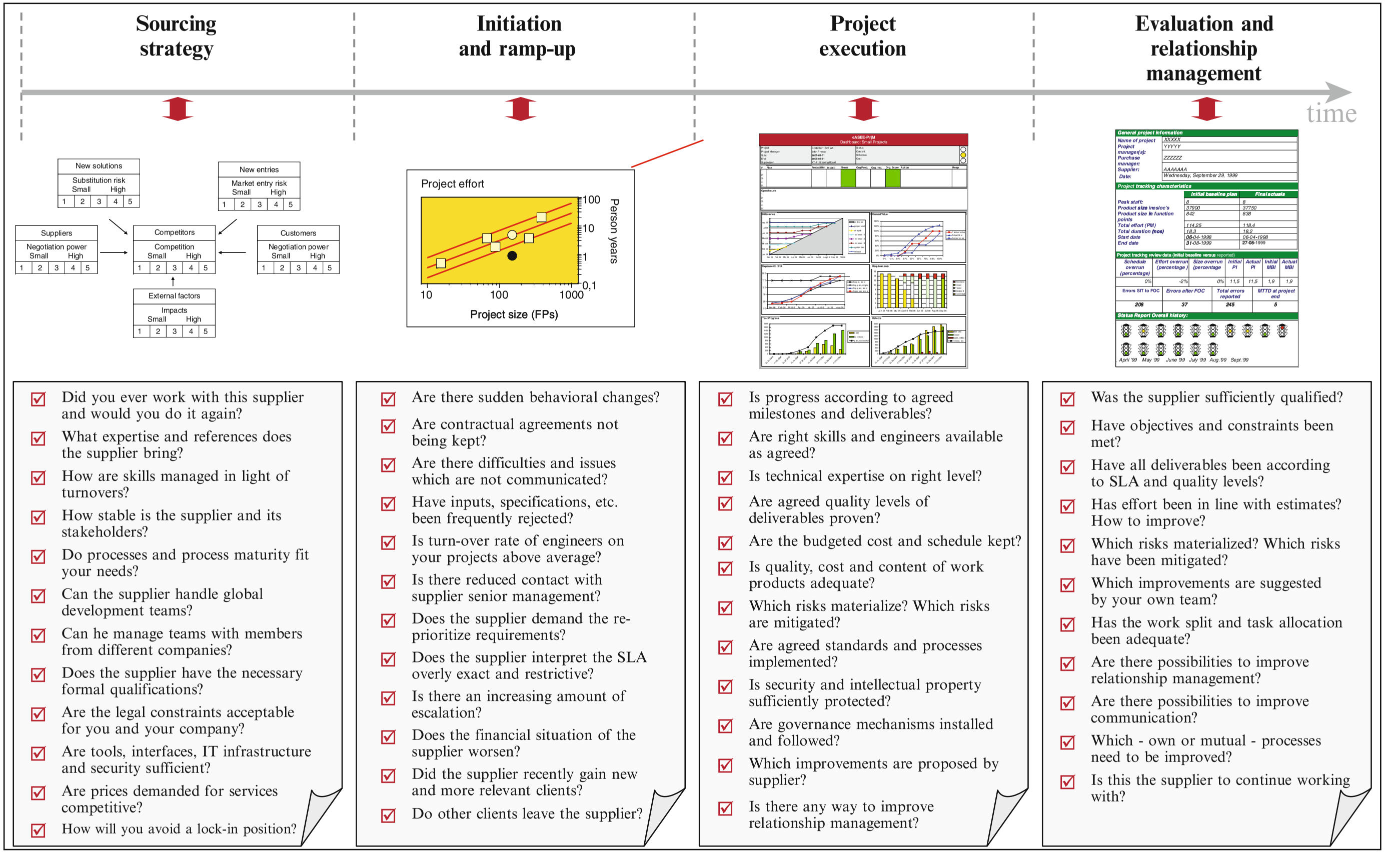

1. Presence: Global growth strategy.
Learn from new markets.
Risk:
▶ Instability with overly high change rates
▶ Inadequate IPR management
2. Talent: Race for skilled people. Value creation happens where the skills are.
Risk:
▶ Staff turnover
▶ Insufficient competencies
▶ Wage and cost inflation
3. Flexibility: Just-in-time organizational networks.
Risk:
▶ Poor supplier services
▶ Lock-in
▶ Distance and culture clashes
4. Efficiency: Speed to profit ahead of competitors.
Risk:
▶ Project delivery failures
▶ Insufficient quality

Fig. 9.4 The top 10 global software risks and their underlying drivers

average, and per head cost will come down even if wage cost is going up. Professional training like the certification of project managers increases the risk of attrition due to a better sellable skill level in the market! Long-term retention methods for attrition management will itself contribute to the other risk of wage escalation. Similarly, the strong correlation observed between skill development and attrition might not be a universal phenomenon, or there might be other overlying attributes impacting attrition more strongly. We recommend that organizations make an internal analysis to fine-tune their approach.

Risk mitigation happens all along the life cycle (see also Chap. 5). It is not enough to identify risks once and then keep a lookout at the repository. Risks are dynamic by nature, and so must be their mitigation. As a general rule for risk identification in a specific environment, we recommend setting up undesired scenarios, evaluating their probability to occur, and deciding for some 10–20 of those scenarios to take dedicated mitigation actions. A majority is mitigated inside the global development project (e.g., common tools), while only a few must be part of the corporate risk strategy (e.g., handling supplier defection). Organizations should not worry about the number of the 10–20 scenarios. They repeat in each of the organization's respective projects and will build a kind of checklist with dedicated and organization-specific mitigation strategies that are reused in each new project. Figure 9.5 shows typical checklists as they are used throughout the life cycle to reassess risks and to follow their mitigation.



☑ Did you ever work with this supplier and would you do it again?
☑ What expertise and references does the supplier bring?
☑ How are skills managed in light of turnovers?
☑ How stable is the supplier and its stakeholders?
☑ Do processes and process maturity fit your needs?
☑ Can the supplier handle global development teams?
☑ Can he manage teams with members from different companies?
☑ Does the supplier have the necessary formal qualifications?
☑ Are the legal constraints acceptable for you and your company?
☑ Are tools, interfaces, IT infrastructure and security sufficient?
☑ Are prices demanded for services competitive?
☑ How will you avoid a lock-in position?

☑ Are there sudden behavioral changes?
☑ Are contractual agreements not being kept?
☑ Are there difficulties and issues which are not communicated?
☑ Have inputs, specifications, etc. been frequently rejected?
☑ Is turn-over rate of engineers on your projects above average?
☑ Is there reduced contact with supplier senior management?
☑ Does the supplier demand the re-prioritize requirements?
☑ Does the supplier interpret the SLA overly exact and restrictive?
☑ Is there an increasing amount of escalation?
☑ Does the financial situation of the supplier worsen?
☑ Did the supplier recently gain new and more relevant clients?
☑ Do other clients leave the supplier?

☑ Is progress according to agreed milestones and deliverables?
☑ Are right skills and engineers available as agreed?
☑ Is technical expertise on right level?
☑ Are agreed quality levels of deliverables proven?
☑ Are the budgeted cost and schedule kept?
☑ Is quality, cost and content of work products adequate?
☑ Which risks materialize? Which risks are mitigated?
☑ Are agreed standards and processes implemented?
☑ Is security and intellectual property sufficiently protected?
☑ Are governance mechanisms installed and followed?
☑ Which improvements are proposed by supplier?
☑ Is there any way to improve relationship management?

☑ Was the supplier sufficiently qualified?
☑ Have objectives and constraints been met?
☑ Have all deliverables been according to SLA and quality levels?
☑ Has effort been in line with estimates? How to improve?
☑ Which risks materialized? Which risks have been mitigated?
☑ Which improvements are suggested by your own team?
☑ Has the work split and task allocation been adequate?
☑ Are there possibilities to improve relationship management?
☑ Are there possibilities to improve communication?
☑ Which - own or mutual - processes need to be improved?
☑ Is this the supplier to continue working with?

Fig. 9.5 Comparison manual versus optimized baseline staffing plan

9.7 Trends and Conclusions

Global engineering will evolve toward a standard engineering management method that must be mastered by each R&D manager. Processes and product components will increasingly be managed in a global context. Suppliers from many countries will evolve to ease the start-up and operations of GSE even for small- and mid-sized enterprises in high-cost countries. Brokers will emerge and help find partners in different parts of the world and manage the offshoring overheads.

The cost per head will stay low for a few years and, over the next few years, will steadily increase due to the rising standards of living in the emerging countries, contributing to outsourcing and offshoring (see also Chaps. 13 and 14). GSE has a strong contribution in improving the living conditions around the world. Bridging the divide is best approached by sharing values and understanding cultures. Such increasing standards of living as in China, India, and many others of today's low-cost countries will generate hundreds of millions of new middle-class people who will demand more information technology.

GSE is the consequence of the rather friction-free economic principles of the entire software industry. Basically, any code can be developed at any place in the world and made visible and accessible to any other place in the world at virtually the same time. There are not many overheads in distribution or industrialization as long as the source code is shared. Many companies start global development due to perceived cost differences. The achieved cost reductions are further delivered to customers, which means competitive pressure for those enterprises not yet embarking on global development. Further advantages appear when intensifying GSE, such as more flexible working hours of engineers and a demand-oriented provisioning of skills. Starting with smaller chunks of working, outsourcing/offshoring intensifies toward globalizing the execution of entire business processes or products. Innovative products are created due to more capacity and more efficient workflows. Product life cycles and technology growth will further accelerate due to this increasing innovation driven by global software engineering. The principle as such is amplified and will not allow any enterprise to exit.

We see four major drivers fueling the need for outsourcing and offshoring, namely, efficiency, presence, talent, and flexibility. Figure 9.6 shows these drivers: efficiency, presence, talent, and flexibility.

1. *Presence*. Global R&D and software engineering has become part of companies' growth strategies because they are closer to their markets and they better understand how to cope with regional needs, be it software development or services. Such global growth is a self-sustaining force, as it demands increasing capacities in captive or outsourced software engineering centers.
2. *Talent*. Computer science and engineering skills are scarce. Many countries do not have enough resources locally available to cope with the demand for software products and services. Fueling this trend, many younger people got nervous with media-driven perceptions about the danger of outsourcing/offshoring for the entire software field, which they decided to rather engage in

1. Presence:
Global growth strategy.
Learn from new markets.

2. Talent:
Race for skilled people.
Value creation happens where the skills are.

3. Flexibility:
JIT networks across organization.
Technology expertise that depends on context.

4. Efficiency:
Process excellence.
Speed to profit ahead of competitors.

Fig. 9.6 The way ahead: four drivers fuel future globalization

fully different fields. The consequence is a global race for excellent software engineers. Outsourcing/offshoring is the instrument to provide such skills and handle the related supplier processes.

3. *Flexibility*. Software organizations are driven by fast-changing demands on skills and the sheer numbers of engineers. With the development of a new and innovative product, many people are needed with broad experiences, while when arriving at maintenance, these skill needs look different, and manpower distributions are also changing. Such a flexible demand cannot be handled anymore inside the enterprise. Outsourcing/offshoring is the answer to provide skilled engineers just in time and thus allows building flexible ecosystems combining suppliers, customers with engineering, and service providers.
4. *Efficiency*. Software companies need to deliver fast and reliably, while at the same time, the competition is literally a mouse click away. Hardly any other business has such low entry barriers as the software business, and therefore, it stimulates an endless fight for efficiency along the dimensions of improved cost, quality, and time-to-profit. Outsourcing/offshoring clearly helps in improving efficiency due to labor cost differences across the world, better quality with many well-trained and process-minded engineers especially in Asia, and a shorter time-to-profit by following the sun and developing and maintaining software in two to three shifts in different time zones.

Global software projects need to facilitate speed, organization, and collaboration. They must leverage investments fast because of the ever-growing risk of

(IP) loss. The new product life cycle is determined by the time it takes to copy and compete. Implement processes that provide agility and efficiency. Companies need to balance the time-to-profit with the time-to-copy. They need to develop an organizational and management strategy for offshoring, along with developing an economic business case. Collaboration will further grow across disciplines, cultures, times, distance, and organizations. This demands new competences such as managing in difficult situations, teamwork across distances and cultures, sharing without losing, etc., which many students so far do not learn in their classroom projects.

Unfortunately (for the expensive western countries), these changing conditions will not have a sustainable positive impact on today's highly paid software engineers. In contrast, an increasing amount of competing software companies will evolve and further push for global alignment of engineering cost (but this time cutting down the top salaries). What looks healthy from a global perspective will have negative impacts on those of us who will not adjust fast enough to a new work split.

To be successful in a global market, a company should manage the risks of global software development and utilize the positive aspects as drivers to shape the software engineering processes in detail and the culture in general. The challenge is to continuously improve processes, innovativeness, and productivity. Software engineering has low entry barriers and a global resource pool. Engineers will have to assess their own competitive value frequently and change gears and functions opportunistically to stay employable. That is the task of all of us software engineers in the future. Those of us who stagnate will be out of business faster than we might think.

History has shown time and again that mixing genes is the best thing that can be done in the path of evolution. Or, in the words of Charles Darwin, who was one of the first truly globally acting scientists, "It is not the strongest of the species that survive, or the most intelligent, but the ones most responsive to change." Globalization is about the same concept...

References

Carmel E, Espinosa JA (2012) I'm working while they're sleeping: time zone separation challenges and solutions. Nedder Stream Press, New York

Chui M, Manyika J, Bughin J, Dobbs R, Roxburgh C, Sarrazin H, Sands G, Westergren M (2012) The social economy: unlocking value and productivity. McKinsey Global Institute

De Loof L (2013) Managing IT transformation on a global scale. http://www.mckinsey.com/insights/business_technology/managing_it_transformation_on_a_global_scale_an_interview_with_shell_cio_alan_matula. Accessed on 15 Aug 2013

Ebert C (2012) Global software and IT: a guide to distributed development, projects, and outsourcing. Wiley, New York

Ebert C, Dumke R (2007) Software measurement. Springer, Heidelberg

Forrester's Forrsights Services Survey Q2 (2012) http://www.forrester.com/Forrsights+Services+Survey+Q2+2012/-/E-SUS1391. Accessed on 15 Aug 2013

Greengard S (2013) Is outsourcing losing its appeal? Deloitte study report. 15 Apr 2013. http://www.baselinemag.com/it-services/is-outsourcing-losing-its-appeal. Accessed on 15 Aug 2013

PWC (2013) Global software 100 leaders report. http://www.pwc.com/gx/en/technology/publications/global-software-100-leaders/index.jhtml. Accessed on 15 Aug 2013

Rivard S, Aubert BA (2008) Information technology outsourcing. M. E. Sharpe, New York

Rottman J, Lacity M (2006) Proven practices for effectively offshoring IT work. Sloan Manage Rev 47(3):56–63

Sangwan R, Bass M, Mullick N, Paulish D, Kazmeier J (2007) Global software development handbook. Auerbach, Boca Raton, FL

Biography Christof Ebert is managing director at Vector Consulting Services. He supports clients around the world to improve product strategy and product development and to manage organizational changes. Prior to this, he held global management positions for 10 years at Alcatel-Lucent. A trusted advisor for companies around the world and a member of several of industry boards, he lectures at the University of Stuttgart and at the Sorbonne in Paris. He authored several books including his most recent one entitled *Global Software and IT published by Wiley*. He received the IEEE distinguished visitor award and is a member of the Alcatel Technical Academy.

Chapter 10
Motivating Software Engineers Working in Virtual Teams Across the Globe

Sarah Beecham

Abstract The motivation of software engineers affects the quality of the software they produce. Motivation can be viewed in terms of needs. The key need for a software engineer is to 'identify with their task' which requires being given a task that is challenging and understanding the purpose and significance of the task in relation to the complete system being developed. Software engineers' needs are complex – they also require regular feedback, trust, appreciation, rewards, a career path, and sustainable working hours. Furthermore, amongst other fixed environmental factors, these motivators require sensitive tuning in line with a software engineer's personality and career stage. Creating this personality-job fit is not easy in a co-located environment, so how can project managers motivate teams of individuals distributed across the globe?

This chapter reflects on some of the motivational issues that managers of virtual teams may encounter. Some background theory is presented for a deeper understanding of how to manage team motivation. Recommendations are drawn from a case study where issues raised by practitioners working in virtual teams serve to highlight and magnify known motivational issues. Project managers play an important part in software engineer motivation. If they can create a working environment that motivates individuals in the team, they will find that team members are more likely to turn up to work, are less likely to look elsewhere for employment, will work harder to meet deadlines, will take more pride in their work, and will share their knowledge, concerns, and ideas for innovation.

S. Beecham (✉)
Department of Computer Science and Information Systems, Lero – The Irish Software Engineering Centre, University of Limerick, Limerick, Ireland
e-mail: sarah.beecham@lero.ie

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_10, © Springer-Verlag Berlin Heidelberg 2014

10.1 Introduction

This chapter explores how to motivate a software engineer working in a virtual team. To answer this question, some general motivation theories are introduced that are relevant to software engineer motivation in a global setting. Since motivating practitioners is likely to lead to improved quality of the software product (McConnell 1996; Verner et al. 2014) and development of software is increasingly a global effort (Chaps. 9 and 12), examining how to motivate software engineers working in globally distributed teams should be of interest to software development practitioners.

Although motivation is a well-researched area, existing theories have not kept pace with today's software engineering climate. The twenty-first century has seen radical changes in both the working environment and the demands made on the people employed to undertake the work. The move towards developing software globally has been rapid, requiring engineers to work in teams around the clock, with mixed values and cultural styles. It is clear that global software development (GSD) is here to stay, despite the risk it poses to motivation (Frey and Osterloh 2002).

GSD can require software engineers to work on sites hundreds or even thousands of miles away from their virtual team mates, where members of the same team may never meet in person. Engineers may also have to cope with time zone differences between sites that constrain the ability to communicate in real time and can lead to delays and frustration, or to working antisocial hours. Other barriers emerge such as cultural and linguistic differences between team members who may need to discuss complex technical issues (Noll et al. 2010; Shah et al. 2012; Monasor et al. 2013). Add to this mix a backdrop of tight deadlines, centralized budgetary controls, and requirements for high-quality software. It is clear from this short summary that GSD places unique and extreme demands on the engineer. While existing *theories* of motivation do not account for the complexities introduced when working in distributed teams, fortunately we have a wealth of empirical research we can draw on to help identify how, where and what motivation means in this context. This chapter draws on theories of motivation and maps these to empirical findings of work undertaken in GSD.

The growth of agile practices (see Chap. 11), shared responsibilities, and flat organisational hierarchies have all contributed to our understanding of how to foster motivation. For example, Beecham et al. (2008) and França et al. (2012) found that agile principles generally meet software engineers' motivational needs, with a few exceptions. However, recent work also suggests that too much freedom and ad hoc arrangements can work against software engineer motivation (Fernández-Sanz and Misra 2011). This seems to contradict the open source and inner source software development paradigms that are gaining in popularity and impetus (Chaps. 13 and 14 relate). The authors of Chap. 14 discuss how software engineers, when adopting OSS practices, are likely to be the recipients of many types of rewards—shown in this chapter to be important intrinsic motivators. In OSS, self-selecting volunteers come together to create their own communities and expend effort to produce high-quality and useful software. It is not surprising that researchers look to these environments to learn about motivation since contributors appear to be motivated by some internal impetus not necessarily associated with financial reward (Ye and Kishida 2003; Roberts et al. 2004; Riehle 2007).

Having read through this chapter, the reader should come away with a basic understanding of some organisational and motivation theories (Sect. 10.2), a feel for whether software engineers are likely to have distinct personalities of their own (Sect. 10.3), and an understanding of how to motivate software engineers in a virtual team setting through a case study example presented in Sect. 10.4. Section 10.5 maps Global Teaming practices to motivational factors. Section 10.6 discusses the case study example in the context of some motivation theory. Section 10.7 concludes the chapter with a summary of software engineer motivation in GSD.

10.2 Motivation Theory

There are techniques managers can apply that will motivate employees to work harder, and increase their commitment to the organisation. Creating the right conditions can also stimulate innovation (Frey and Osterloh 2002). However, perhaps the reason that there are well over 100 theories of motivation (Petri and Govern 2012), is that no one theory truly reflects how people are motivated. At best, each theory provides new insights into what is a highly complex area (da Silva and França 2012).

Classic motivation theories can be broadly classified as either ‘content’ or ‘process’ theories. Content theories include Maslow’s hierarchy of needs (1954), Herzberg et al.’s two-factor theory (1959), and McClelland’s needs theory (1961). These content theories assume a “complex interaction between internal and external factors” and explore “the circumstances in which individuals respond to different types of internal and external stimuli” (Bassett-Jones and Lloyd 2005). Process theory, on the other hand, describes motivation as “a sequence or process of related activities” (Hall et al. 2009). Exponents of process theories include Adam’s (1963) equity theory, Vroom’s (1964) expectancy theory, Skinner’s (1976) stimulus-response theory, Locke et al.’s (1968) goal setting theory, and Hackman and Oldham (1976) and Couger and Zawacki’s (1980) task design theories. This chapter focuses on the process theories relating to work design and job characteristics (Hackman and Oldman 1976; Couger and Zawacki 1980), where task variables are explored in a GSD context. Also, Herzberg’s (1959) two-factor content theory (motivation-hygiene theory) is discussed since the external environment (a hygiene factor) is an integral part of GSD.

Motivation theories try to explain the conscious or unconscious decisions people make to expend effort or energy on a particular activity. Handy (1993) encapsulates many of these process theories in his ‘motivation calculus,’ which expresses

A software engineer with a strong need to learn is given a problem-solving task that promises to expand his or her knowledge and skill-set. The ***effort*** *the engineer is prepared to expend on the task will depend on the degree to which he or she*

*a) believes that solving the problem will lead to increased knowledge and skills (****expectancy*** *that the need will be satisfied) and*

b) having completed the task, finds that the ***result****ing increased knowledge and skills satisfy the learning needs (****instrumentality*** *of the satisfied need).*

Fig. 10.1 Example scenario of motivational elements adapted from Handy (1993)

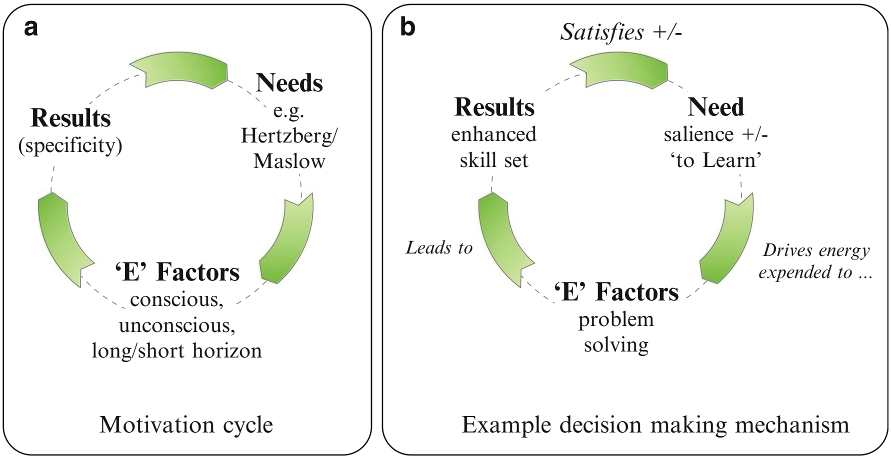


Fig. 10.2 Motivation calculus inspired by Handy (1993) (**a**) Motivation cycle (**b**) Example decision-making mechanism

motivation in terms of *needs*, *results* that satisfy needs, and *effort*[1] expended to achieve those results that satisfy the needs. The calculus demonstrates that the level of effort a person is willing to expend on a task is linked to how the person feels that effort will be rewarded; if the person values the expected reward, they will increase the effort accordingly. If the result fulfills a need and the experience is positive, it will feed into the person wanting to expend energy again on similar need-related tasks, and so the cycle continues. The re-enforcement cycle is explained in the example in Fig. 10.1 and summarised in Fig. 10.2.

Figure 10.2 indicates that it is tapping into the strength (or salience) of the need that will determine how to motivate the engineer. For example, looking at Fig. 10.2 (b),

[1] Effort is just one of the E's in the calculus; other E's include energy, excitement, enthusiasm, emotion, expenditure of time, expenditure of money, and expenditure of passion.

motivation can break down if the problem-solving task does not lead to enhancing the skill set as this will not satisfy the need to learn. Alternatively, the motivation cycle is broken if an enhanced skill set does not satisfy the need to learn. The need and how to satisfy that *need* will vary from person to person. Individual characteristics play an integral part in motivation theories.

It is the mapping of the individual need to the type of job that forms the basis for the 'job characteristics theory' (JCT) also discussed in this chapter. Section 10.3 considers the individual personality traits and characteristics of the software engineer—this is perhaps the Holy Grail, as if we can find common traits in people attracted to the software engineering profession, it will ease the task of motivating engineers.

10.2.1 Motivation as a Social Process

While Handy's approach is useful in capturing motivation, understanding can be broadened by viewing motivation as part of a *social* process. This complementary social process dimension relates strongly to the context of software engineers who must work together in teams and interact. Motivation as a social process defines how people join, remain part of, and perform adequately in a human organisation (Huczynski and Buchanan 1991). The global organisation is a social arrangement comprising members who are motivated to join, to stay, and to perform at acceptable levels. It is within a social context that teams working remotely are encouraged to work harder and more effectively. Some research suggests that social interaction itself can be motivating (Petri and Govern 2012). Self-motivation is just one factor that drives an individual to join an organisation, to stay, and to perform at acceptable levels. The other characteristics are discussed in Sect. 10.3. The increase in the use of social media in GSD (see Chap. 16) reflects the growing need for members of distributed teams to collaborate via informal channels.

10.2.2 Rational-Economic Needs

Scientific management research, conducted in the 1940s, asserted that dividing work into structural units and offering monetary incentives would motivate individuals to increase their productivity. Frederick Taylor (1947), a key proponent of the scientific management movement, introduced the rational-economic needs concepts of motivation that he believed would lead to work being more satisfying and profitable. Taylor hypothesised that workers would be motivated by the high wages that they earned by working in the most efficient and productive way. Taylor was preoccupied with finding the most efficient methods and procedures for coordination and control of work, a goal shared with today's global software development managers. Key principles of Taylor's approach include the division of labour, the sharing of responsibility between management and workers, knowledge sharing, and carrying out work in a prescribed way. It appears that Taylor anticipated the need for a work environment suited to automation. Taylor's vision can also be mapped to GSD and other product management approaches since dividing up processes into discrete, unambiguous pieces eases task allocation and sharing among developers working in virtual teams. Chapter 9 discusses how to allocate tasks in distributed and global development to maximize team cohesion and minimize coupling.

Initially, the effect of the introduced changes raised productivity and employee wages by 60 % (Huczynski and Buchanan 1991). However, despite increased output and monetary rewards, there was a strong reaction against scientific management methods from employees (Mullins 1993). Taylor's design of fragmented tasks was boring and called for a low level of skill. Requiring low levels of skill allowed the management to treat people as pure resources that could easily be replaced. In line with this, managers could reduce wages and ignore the psychological needs of the employees who had little opportunity to give feedback, experiment or make changes. These factors resulted in Taylorism in its strictest sense becoming obsolete with the term "scientific management" falling out of favour fairly soon after its introduction.

Paradoxically, even though we are aware that Taylor's methods do not work in the long term, today's managers of distributed teams seem to be reintroducing some of these practices. In terms of division of labour, one model is that testing gets outsourced to low-labour cost countries, design is undertaken in the onshore office, and coding is performed in satellite locations. Also, decision making can be the province of the centralized managers, where the needs of those working remotely are not necessarily represented in the organisational process.

In summary, Taylor's approach reflected the spirit of the times in the 1940s, a time of industrial reorganisation, new forms of technology, and the emergence of large complex organisations. We now find ourselves again in a phase of industrial reorganisation, with even more complex work structures in the form of globally distributed software development. Work patterns have changed largely due to new forms of technology, especially concerning methods of communication. Perhaps for this reason elements of Taylor's approach have not died out, a sentiment shared by academics who proclaimed in the early 1990s that "Taylorism is alive today" (Huczynski and Buchanan 1991). More recently, researchers Kennedy and Nur (2012) note that prescriptive practices associated with Taylorism continue to rise. Engineering work is now highly controlled by procedures and the increased need for senior management approval. This has implications for motivation as "so long as the Taylorist paradigm persists, the organisational aspiration to create a high commitment culture is likely to prove elusive" (Bassett-Jones and Lloyd 2005). However, it could be that it is the high employee turnover and subsequent low knowledge retention that drive the need for regimented processes (Kennedy and Nur 2012).

10.2.3 Motivation Theories for Software Engineering

Given that motivation is important, and that looking at people as machines that will do repeated work ad infinitum (even for good pay) is not a panacea, we now move away from the scientific management view and go to an approach where it is the people within the organisation that matter most.

Organisational theory records several approaches and models of motivation, many of which have been applied in software engineering research (Hall et al. 2009; Sharp et al. 2009). This chapter focusses on two theories that stand out amongst this group as being particularly relevant to motivating people who work in global software development teams: firstly, the content theory expounded by Herzberg et al. (1959)—the *two-factor theory*- and secondly, the process theory according to Hackman and Oldham's (1974) *job characteristics theory* (JCT) and adapted by Couger and Zawacki (1980) for software engineers. A brief definition of these theories is given next, along with why they might, even after 40 plus years, support the management of virtual teams.

Herzberg's two-factor theory: In 1959, Herzberg and his collaborators isolated two different factors that influenced motivation and satisfaction at work. One set of factors, classified as 'demotivators' or hygiene factors, are those that, if absent, can reduce motivation; these extrinsic factors are concerned with the work environment. However, to motivate employees to give their best, the focus must move to a different set of factors, classified as 'motivators' or intrinsic factors relating to the task itself.

The work of Herzberg is pertinent to global software development, since motivation (and demotivation) factors are viewed in terms of external influences and internal influences, even though the theory was developed over 50 years ago. However, there is some controversy as to how factors are classified, "largely because of the assertion that there was a weak correlation between financial reward and job satisfaction" (Bassett-Jones and Lloyd 2005). Herzberg classifies financial rewards as a hygiene factor, suggesting that inadequate financial reward can demotivate—and that beyond a limited threshold, money cannot motivate. Although classifying factors as either *hygiene* or *motivators* can appear contrived, it is helpful for the purpose of identifying how GSD factors may demotivate. Also, it is helpful as there might be some hygiene factors that are outside the control of the project manager. Of note is that demotivators are not the opposite of motivators; demotivators and motivators are distinct groups of factors.

Job characteristics theory: According to Beecham et al.'s review of the motivation literature (2008), Hackman and Oldham's (1974, 1976) job characteristics theory (JCT) is the most applied theory in software engineering. This process theory views the work itself as the main motivator, where given a set of personal needs, a person will only be motivated if these needs are matched by the job. The JCT model reflects the relationship between job characteristics, psychological states, and personal work outcomes. The JCT was extended by Couger and Zawacki (1980) to fit the software engineering context. The associated data collection tool, the Job

Diagnostics Survey (JDS), was developed to quantify an individual's person-job fit. Once quantified, motivation levels can be compared across individuals, across organisations, and across studies. As shown in the investigation into Herzberg's hygiene factors, global software development can involve environmental factors outside the control of the project manager. The importance, therefore, of the person-job fit seems particularly pertinent to our solution. The person-job fit for GSD is considered later in this chapter.

10.3 Characteristics of a Software Engineer

As noted in the Section on motivational theory (Sect. 10.2), there is no one-size-fits-all solution to motivating software engineers since motivation depends on individual needs and personality. Findings are derived from an in-depth systematic literature review of software engineer motivation (Beecham et al. 2008). A section of the motivation review was dedicated to studies that looked specifically at types of people attracted to the software engineering profession as opposed to what motivates them to stay in the profession, to do better work, etc. As a result, some characteristics appear similar to motivators, even though they came from a different strand of research. The original rationale for conducting the research into the characteristics was to assess whether software engineers are somehow different to professionals in other domains. Because, if there is no difference, we could argue that we do not need a separate study and model of motivation for software engineers; we could draw on the existing models and theories of motivating people in the workplace, some of which have been discussed in the previous section. On balance, 73 % of studies of software engineers indicated that software engineers do form a distinct identifiable occupational group (Beecham et al. 2008). This finding indicates that studying motivation for software engineers as a separate profession could benefit the managers of software engineers and researchers.

Figure 10.3 shows how controls and mediators will shape a software engineer's characteristics.

A systematic literature review of 92 separate studies relating to software engineer motivation (Beecham et al. 2008) observed that a software engineer's characteristics are formed by two factors: their internal make-up (termed 'control factors') and external factors (termed 'moderators'). Control factors define innate personality. Although personality will have an influence on the characteristics of a software engineer, defining personality types goes beyond the scope of this chapter. For readers interested in knowing more about this dimension, Chap. 4 gives an in-depth description of personality and how to assess different personality types. Moderators, on the other hand, are discussed briefly since they are understood to change the strength of certain characteristics.

The moderators identified in Beecham et al. (2008), listed in Table 10.1, have particular significance in a global, virtual team setting. Although they can be outside the control of the manager, they still need to be acknowledged as important

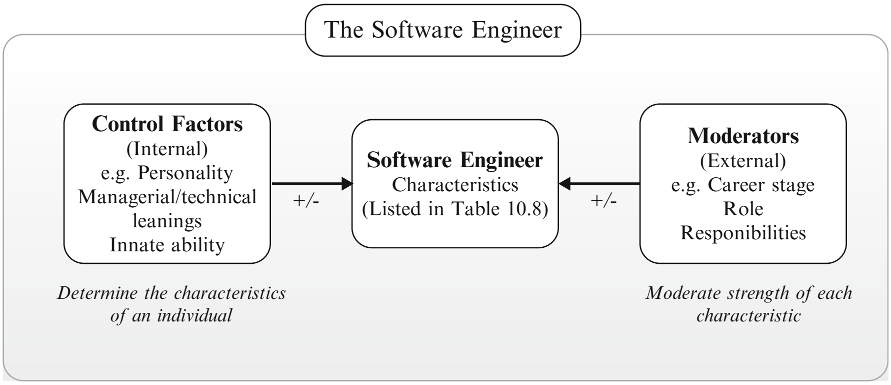


Fig. 10.3 Model of software engineer characteristics adapted from Beecham et al. (2008)

to the individual. For example, although an individual's career stage, age, and the state of the IT profession are fixed, they may all influence how to motivate the individual.

Also, the culture of the individual or given setting has been identified as problematic in many GSD studies, for example, Olson and Olson (2004), and is often labeled as a barrier to successful communication (Noll et al. 2010). However, managers can take advantage of a mixed cultural team for enhanced creativity, innovation, and holiday cover (Deshpande et al. 2010).

Promotion prospects are also seen as moderating a software engineer's characteristics. For example, Johnson et al. (2010) found a significant positive correlation between an employee's promotion focus and affective commitment. Promotion prospects are a poorly studied area in GSD research and may as a result be overlooked as an important motivator. Translating this moderator into a GSD context highlights the need for software engineers working remotely to have a clear career path and promotion opportunities. All factors listed in Table 10.1 are likely to moderate the strength of a software engineer's *characteristics*.

Of the many software engineer characteristics identified in the literature (considered in relation to GSD later in this chapter in Table 10.8), growth-oriented, introverted, and need for independence were the most cited. However, some characteristics contradict each other, such as 'introverted' with a low need for social interaction, and 'need to be sociable and identify with a group or organisation'. The view that software engineers are introverted reflects findings from the many studies coming from Couger and colleagues who measured the social needs strength of engineers (Couger and Zawacki 1980) in their Job Diagnostics Survey (for a full list of sources, see Beecham et al. 2008). This view is not universal, as seen in the body of more recent research that identified software engineers as sociable people (Beecham et al. 2008). Certainly the need for software engineers to communicate and relate to others is crucial in a GSD context. The new software engineer profile may therefore reflect the changing demands of the role.

Table 10.1 Software engineer moderators

1.	Career stage (age and experience)
2.	Culture (relating to national culture)
3.	Job type/role/occupational level
4.	State of IT profession (snapshot of evolutionary process)
5.	Type of organisation (e.g., promotion opportunities/rules)

Figure 10.4 shows the relationship between characteristics, controls and moderators (given in detail in Fig. 10.3), and motivators and outcomes. The level to which the needs (defined by a software engineer's characteristics) are met by the motivators will impact on tangible outcomes. For example, Hall et al. (2008a, b) found a positive correlation between software engineer motivation and employee turnover, and Verner et al. (2010) found a positive correlation between motivation and software engineering/management agreements on project success.

In summary, this section reflected on the characteristics of software engineers and whether they have any common characteristics. Understanding that engineers are likely to have some distinct traits should help managers to motivate their software development teams. The studies included in the motivation review (Beecham et al. 2008) did not consider the extra complexity of working in a distributed environment. The characteristics in a GSD context are considered in Sect. 10.6. The next section places all the 22 motivating factors identified in the review in a GSD context through an empirical mapping study.

10.4 Software Engineer Motivation in GSD—A Case Study

This section examines how software engineers' needs are likely to be affected by GSD. A case study conducted during 2010–2011 is used as an example of how virtual team behaviour may inhibit or strengthen software engineer motivation.

The case study is based on a GSD organisation that distributes its software development activities across several sites and countries. The organisation has its central office in Ireland and develops bespoke software for the financial services sector. It is a medium-sized organisation with approximately 200 employees. For reasons of confidentiality and anonymity, this organisation is referred to as "GSD Corp." GSD Corp offshore much of their development activities, but maintain a large team of practitioners in the central office who work mainly from 9 A.M. to 5 P.M., 5 days a week. This team develops their core product, manages the off-shore teams, and tests the bespoke software. The offshore teams are more focussed on requirements gathering and product deployment. Offshoring is undertaken for three key reasons: firstly, to extend their customer base; secondly, to work closely with their customers; and thirdly, to hire excellent technical talent in low-cost countries. All development comes under the organisational control of GSD Corp. The purpose of conducting the case study was to examine the processes used by GSD Corp to develop their software.

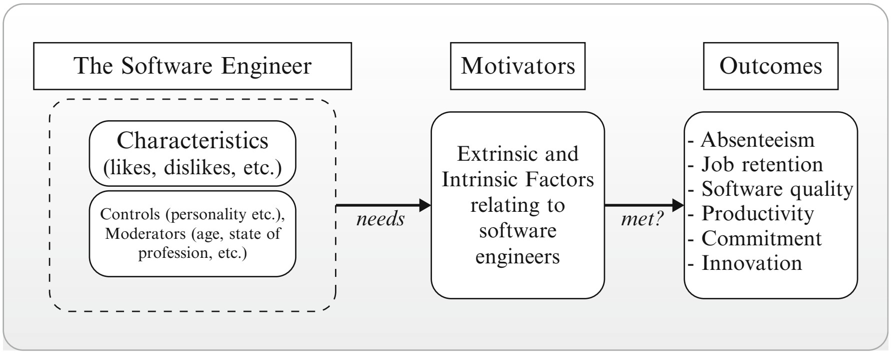

The Software Engineer
Characteristics (likes, dislikes, etc.)
Controls (personality etc.), Moderators (age, state of profession, etc.)
needs
Motivators
Extrinsic and Intrinsic Factors relating to software engineers
met?
Outcomes
- Absenteeism
- Job retention
- Software quality
- Productivity
- Commitment
- Innovation

Fig. 10.4 Model of software engineer motivation adapted from Beecham et al. (2008)

Onshore and offshore opinions and lessons learned were solicited through a series of in-depth interviews that provided insights into where GSD processes might be improved. To gain a full picture of day-to-day work patterns, two projects were studied, one of which had just been completed, and the other was still in progress. Twenty-four employees performing various roles in the development process based in four different countries (Ireland, USA, South Africa, and Australia) were interviewed using the same set of semistructured questions. Each employee was interviewed for 1–2 hours, on a one-to-one, confidential basis. All interviewees worked in a distributed environment. Detailed notes were taken during the interviews, which were also recorded and later transcribed verbatim. A cross section of roles was interviewed in the sample, including technical and business consultants, quality assurance, project managers, project leads, solution architects, technical and business stream leads, and programme manager.

The detailed notes and analysis of interview transcripts presented a full picture of how GSD Corp operates across its several sites. The findings presented in this section are drawn from a subset of responses to direct and indirect interview questions that related to motivation. Direct questions included “how motivated were you in your project?”, “was your project a success or highly challenged?”, “how do you define project success?”. Indirect questions tackled the problems and challenges the interviewees experienced in conducting their day-to-day tasks working in a virtual team, as well as discussing what excited them about their work. Using a qualitative, content analysis approach (Krippendorff 1980) similar to that used in Noll et al. (2011), responses were categorised according to whether they highlighted a challenge or a solution to a given problem, as well as advantages and disadvantages of working in virtual teams.

Table 10.2 Extrinsic motivators enhanced by virtual team practices

Extrinsic factor	Virtual team practice (drawn from case study)
Working in a successful organisation	Spreading the business into new markets was seen as a good business model. Operating in different countries was linked to organisational success.
Job security/stable environment	None of the employees interviewed felt insecure. They knew that good engineers where in short supply and did not feel that their jobs were under threat, despite the financial climate.

10.4.1 Case Study Results

The preliminary results shown in Tables 10.2, 10.3, 10.4, and 10.5 map case study findings (such as advantages and disadvantages of working in a virtual team) to motivational factors drawn from the literature (Beecham et al. 2008). Extrinsic factors are presented separately as it is likely that they will require a different management approach to the intrinsic motivators that are concerned with the job itself.

In the case study, two extrinsic factors were enabled by GSD: job security (no interviewee felt their job was under threat) and the need to work for a successful company (see Table 10.2). Some factors appear independent of whether employees work in a co-located or virtual environment, such as having a feedback mechanism in place. Factors that are likely to be equally difficult to achieve in virtual and co-located teams are not covered in these tables.

However, the case study did reveal several motivational factors that appear to be challenged as a result of working in a distributed team. Extrinsic factors (see Table 10.3) such as poor working conditions (disruptive) and a poor work-life balance (e.g., unpredictable working hours, extensive travel, and long commute) can demotivate software engineers. GSD Corp's poorly defined virtual team roles weakened practitioner empowerment and feeling of responsibility. Ambiguous roles and responsibilities can be problematic in a co-located environment, and working remotely magnifies the problem. For example, a team lead working at a remote customer site was undermined by senior management (working from a different country and time zone) who decided to discuss on-site matters directly with the customer (ignoring the team lead working in a predominantly customer-facing role). Not only was this undermining for the on-site team member, it caused confusion as to who is responsible for handling customer issues.

Considering issues directly associated with the job itself (shown in Tables 10.4 and 10.5), there were perceived differences in how employees were treated based on where they were located. For example, a practitioner working offshore reported that they sometimes missed out on training opportunities. Also, those based remotely felt they did not have the same promotion prospects as the onshore team since the remotely based senior management tended to see problems rather than when the employee was doing a good job and there were fewer options in a small organisation for internal promotion. Offshore teams felt that they worked longer

Table 10.3 Extrinsic motivation challenged by virtual team practices

Extrinsic factor	Virtual team practice (drawn from case study)
Rewards and incentives (e.g., scope for increased pay and benefits linked to performance)	Requires objective measurement, and as such is independent of location—however, making sure that rewards are given to each employee fairly across different locations may not be achievable, e.g., some remote workers were not able to take time off in lieu for working long hours and overtime.
Good management (senior management support, team-building, good communication).	Becomes even more important when working remotely—extra pressures, extra layer of complexity requires experienced and confident managers to deal with unforeseen problems. A recurring theme was that remote projects required experienced managers that can communicate well with both customers and all team.
Sense of belonging/supportive relationships	Difficult to feel supported when your counterpart might be sleeping during your core working hours. However, the organisation had a strong corporate culture, clearly communicated in all interviews.
Work/life balance (flexibility in work times, caring manager/employer, work location)	Extremely difficult to achieve, when there is a lot of travel, working away from home (and family), and keeping work hours down to core times seems impossible. It was rare to hear any reports of people working sustainable hours when working remotely.
Employee participation/involvement/working with others	Some experienced managers working remotely did not want to participate with the wider organisation; finding interference from higher management to be a negative influence. They tended to want to be left alone to sort out their customer facing issues. A fine balance needs to be struck between participation and a top-down style of management that imposes the processes.
Appropriate working conditions/environment/equipment/tools/physical space/quiet	Working conditions specially affected remote workers. For example, when working onsite with customers they often did not have any influence on where they work, or how and sometimes, when. They were not able to separate themselves from being on call to the customer: there was a tension between dealing with customer demands and their tangible deliverables.
Sufficient resources	Resources were scarce in terms of people (individuals were stretched to fill the gaps).

Table 10.4 Intrinsic motivation challenged by virtual team practices

Intrinsic factor	Virtual team practice (from case study)
Development needs addressed (e.g. training opportunities to widen skills)	Formal training, though offered centrally, would not always be extended to those employees working remotely. Many employees would have liked to attend training programmes that were only made known to them after they occurred, or when all places were filled. Also, working remotely, they were not able to take the time out for training, which was not built into the development schedule.
Technically challenging work	The technical work may be less challenging if task is highly specified, with reduced dependencies. However, balancing the many responsibilities and demands on time, and keeping both customer and management happy was very challenging.
Identify with the task (clear goals, how task fits in with whole)	Working in a distributed fashion in some cases resulted in developers not seeing the bigger picture and how their part of the work fitted in with the overall delivered product.
Employee participation/involvement/working with others	Members of the team may find it difficult to work with others if they are in different time zones. They can become disenfranchised or alienated. Difficult to make their opinion heard if not working physically together.
Career path (opportunity for advancement, promotion prospect, career planning)	There was a perceived lack of opportunity for advancement within the organisation—especially in an offshore role. Also in this SME, there were a limited number of roles available. Individual career plans to gain as much experience as possible to improve their marketability with external employers were met. Yet, even if employee work experience and skills were increased over time, they may leave the organisation to reap the benefits of increased skills.
Making a contribution/task significance (degree to which job impacts on others).	Working in a piecemeal fashion, e.g. just doing the coding, or part of the coding, prevented the developer from recognising how his/her part will make a difference. Some were uncertain as to whether software they are developing was used/implemented.
Recognition (for a high-quality, good job done	Universal recognition for a job well done is difficult to achieve when working remotely, where the main reason for making contact with head office might be to escalate a problem, or to check back when some action needs to be granted permission, or expenses need to be paid. If all is going well, then the practitioner 'doing a good job' may just be invisible.

(continued)

Table 10.4 (continued)

Intrinsic factor	Virtual team practice (from case study)
Trust/respect	Trust is a recognised problem in GSD and can cause barriers to the development. Engendering trust is difficult when teams may never have met face to face, may not share a common language, and may have different cultures. This however did not pose a problem in the case study with their strong corporate and friendly culture.
Equity	Fair treatment of all individuals working in virtual teams is difficult to achieve in GSD. For example, teams working in both the UK and the USA may feel that the employee in the other county is working less hours (e.g., a US employees developer may not be able to contact their European counterpart after 11 A.M. US time), also the US tend to take fewer days leave than their European counterparts.
Empowerment/responsibility	Responsibility by role is ambiguous—when roles are not well defined, it is difficult to know just how much authority you have to make changes and make decisions. Decision making is key to motivation (Handy 1993), therefore responsibilities need to be clear.

Table 10.5 Intrinsic motivation enhanced by virtual team practices

Intrinsic factor	Virtual team practice (from case study)
Variety of work (e.g., making good use of skills, being stretched)	The individual can find themselves fulfilling several roles, even if not trained or experienced in the role. When working remotely, there might be no-one to delegate to. Employees made excellent use of their skills. However, there became an over-reliance on the employee, who at times experienced unsustainable working hours.
Autonomy	Autonomy is usually not a problem when working remotely; a prerequisite for remote working is the ability to work independently. However, individuals can be undermined if head office is heavy handed, and interferes with communication, say with on-site customers, or if their work is monitored too stringently. For developers working under the spotlight of the customer, autonomy can be problematic.

hours and had less leave than those based in the head office. These factors threaten motivators such as career path, trust, recognition, good management, respect, rewards, and equity.

10.5 Motivational Factors and GSD Guidelines

GSD research is rich in frameworks, guidelines, and recommendations that aim to overcome challenges that arise when software is developed by teams that are separated by geographic distance, that span multiple time zones, have different first languages and represent multiple cultures. However, none of these guidelines are expressly connected to motivation. This section therefore addresses the question, “How do GSD recommended practices support software engineer motivation?” For brevity, consideration is given to guidelines developed specifically for global teams. These are presented in the global teaming model (GTM) according to Richardson et al. (2012) and encompass 20 detailed GSD practices drawn from empirical research and the GSD literature. Where possible the guideline is mapped to the motivational factors identified in Tables 10.2, 10.3, 10.4, and 10.5. Tables 10.6 and 10.7 present a mapping of extrinsic and intrinsic motivational factors to GTM practice guidelines. The motivational factors included in Beecham et al. (2008) are considered, and the case study findings have been used to explore how the Global Teaming Model (GTM) guidelines support motivation.

The mappings shown in Tables 10.6 and 10.7 indicate that if a project manager follows the guidelines offered by the Global Teaming Model, they will in turn also address many of the software engineer’s motivational needs (labeled ‘need directly addressed’). Those areas not supported by the guidelines (labeled ‘need not addressed’) tend to be environmental, and outside the scope of a process model such as the Global Teaming Model (Richardson et al. 2012). However, many of the solutions are indirectly addressed (labelled ‘implied/partial support of need’) and will require careful implementation to ensure the practice fully meets the motivational needs of the virtual engineer. Those motivators labeled ‘need directly addressed’ will also need further investigation, since as already discussed in this chapter, each guideline needs to be tailored to the individual requirements of the software engineer. (The GTM (Richardson et al. 2012) contains more detailed descriptions of the guidelines.)

Although the Global Teaming Model takes a management or organisation view, it does reflect the needs of the employee, as shown in this practice: “Ensure that the supervision, support and information needs of all team members are met regardless of location.” However, although the guidelines for global teaming reflect best practice, they are no substitute for highly experienced project managers (Hall et al. 2008a, b; Beecham et al. 2013; Monasor et al. 2013).

10.6 Theory and Practice of GSD Motivation

Šteinberga and Šmite (2011) conducted a complementary study of motivation in distributed software development teams. They mapped motivators to the GSD literature in order to assess the impact of GSD on software engineer motivation.

Table 10.6 GSD guideline support for intrinsic motivation

SW engineer intrinsic motivators	Global teaming guidelines (Richardson et al. 2012)
	NEED DIRECTLY ADDRESSED
Development needs addressed (e.g., training opportunities to widen skills)	"Training should be tailored to team member's specific needs and location." "Undertake training onsite and face-to-face so team members can be directly assessed and training provision tailored to their specific requirements."
Identify with task (how task fits in with whole)	"Project goals and objectives should be communicated, understood and agreed across all team members regardless of location."
Making a contribution/task significance	"Let team members know their input to process development and ownership is valued."
	IMPLIED/PARTIAL SUPPORT OF NEED
Variety of work (e.g., making good use of skills, being stretched)	"Gather all information relating to the technical . . . experience of potential and existing team members. When teams are in place and project details reported project managers should understand . individual's skills and knowledge."
Technically challenging work	"Gather all information relating to the technical and professional experience of potential and existing team members."
Employee participation	". . . individuals visit locations for extended periods. . ."
Autonomy	Modularisation is one approach to development where work is partitioned into modules that have a well-defined functional whole.
Recognition (for a high-quality, good job done -different to rewards/incentives)	"The global team is viewed as an entity in its own right, regardless of the location of its team members and its performance should be judged and rewarded accordingly." "Acknowledging team success may require tailoring rewards to the needs of different cultures."
Trust/respect	"Structure global team and monitor operation to minimize fear and alienation in teams." "Set up a strategy to handle, monitor and anticipate where conflict between remote locations may occur."
Equity	"Be aware of problems with unbalanced team sizes . . ." "Ensure supervision, support and information needs of all team are met regardless of location."
Empowerment/responsibility	Information of individual's role within the team and specific areas of responsibility [should be recorded].
	NEED NOT ADDRESSED
Career path (opportunity for advancement, promotion prospect) hierarchy; state of economy.	Outside scope of practice model: depends on organisation size;

Table 10.7 GSD guideline support for extrinsic motivation

SW engineer extrinsic factors	Global teaming model guidelines (Richardson et al. 2012)
	NEED DIRECTLY ADDRESSED
Rewards and incentives	"Identify common goals, objectives and rewards." Consider "cultural issues, economic situation and income tax laws when planning rewards."
Employee participation/ involvement	"Face-to-face meetings are recommended when and where possible"
	IMPLIED/PARTIAL SUPPORT OF NEED
Good management	"Ensure that the supervision, support and information needs of all team members are met regardless of location."
Feedback	"Strategies need to be put in place to encourage formal and informal reporting… Seek and encourage input from team members at all locations."
Sense of belonging/supportive relationships	"Provide training to give all team members an opportunity to learn and understand about each other's culture."
Work-life balance	"Achievable milestones should be planned and agreed." "Project manager should allocate tasks and timescales that are realistic."
Appropriate working conditions/environment	Many solutions relate to this under practice "Determine team and organisational structure between locations."
	NEED NOT ADDRESSED
Working in successful company	Environmental/not based on practice
Job security stable environment/	Environmental/not based on practice
Sufficient resources	Environmental/not based on practice

Their assessments, however, do not totally agree with the empirical results reported in this section. These differences indicate that the findings reported in this chapter are preliminary, context specific, and non-generalizable. Šteinberga and Šmite (2011) hypothesized that many motivators challenged by working in a distributed manner would be supported by agile methods. For example, feedback, recognition, and trust are likely to be promoted by iterations and small releases that enable early and frequent feedback, and recognition for a job well-done. Some of these ideas are supported by Beecham et al.'s (2007) empirical study of motivation of teams applying XP development methods. Specifically XP had a positive effect on motivation by clearly monitoring project progress, promoting knowledge sharing and learning on the job, working independently as a team, and communicating good and bad news through positive feedback without punitive repercussions. However, individual recognition was a problem (as the focus was on the team, or pairs of programmers) that could have a negative influence on promotion prospects and rewards. Also, the work tended to be repetitive and therefore did not meet the need for variety. Although agile methods were originally designed for co-located teams, research has shown that distributed teams can apply many of the practices effectively (Jalali and Wohlin 2012; Hillegersberg et al. 2011). For more information on general agile project management, see Chap. 11.

Returning to motivation theory—how can it help us understand how to manage motivation in virtual teams? The job characteristics theory emphasizes the need to match the person to the role to ensure their growth needs and social needs are met. We found in our analysis of software engineer characteristics that software engineers vary in their profiles. For example, they are not necessarily introverted or motivated by financial rewards (although some might be). Placing these ideas in the context of GSD points to the importance of global project managers identifying which practices can be adapted to meet the needs of the engineer and which practices are fixed. Where the global manager is powerless to change a practice or environmental factor associated with a given task, an option will be to select a person whose characteristics are most suited to the role required to complete that task. If the software engineer enjoys the task to the extent that environmental factors do not detract, or whose growth needs and social needs match those offered by the task, their motivation level should remain high.

An analysis of the impact of GSD on the motivation of practitioners interviewed in the GSD Corp case study listed areas that were ‘challenged’ by distributed development. Applying GSD best practice in the form of Global Teaming recommendations indicates that good management could, in many cases, counteract these vulnerable areas. Challenged areas in GSD motivation include rewards and incentives, staff development, work-life balance, and promotion opportunities. However, of more concern are those factors that, due to the environment, would be extremely difficult, or even impossible, to change or control. The only way to support practitioners involved in GSD exposed to these fixed factors is by having a clear knowledge of their characteristics. For example, an engineer with a high need to work with people face to face would be unsuited to working in a virtual team. Enjoyment of travel and ability to communicate with people from different cultures is also a prerequisite in many distributed projects.

Tables 10.8 and 10.9 list the characteristics, moderators, and controls associated with software engineers’ suitability to working in a GSD environment according to case study findings. Managers can use these tables as a starting point to identify those practitioners suited to working in virtual teams either because of their characteristics, or moderators and controls of those characteristics. For example in Table 10.8, some engineer characteristics (for the sake of the example labeled ‘low’ suitability for GSD) require high geographic stability (suggesting a dislike of travel), and an introverted personality. A manager may decide that the role demands a lot of travel and interaction with other team members, and therefore this profile could be deemed unsuited to the role. A more suitable set of characteristics for GSD (labeled high GSD compatibility) is likely to be that the software engineer is technically competent, growth oriented, independent, creative, etc.

Sharp et al.’s (2007) empirical research considered the motivation of software engineers in terms of the role they play, thus creating a more pragmatic model of motivation than considering each member of a large team individually. This view reflects Maslow’s theory of motivation (Maslow 1954), where, for example,

Table 10.8 Software Engineer Characteristics and GSD Role

	Characteristic	GSD compatibility
Ch.1	Need for stability (organisational stability)	Low
Ch.2	Technically competent	High
Ch.3	Achievement orientated (e.g., seeks promotion)	Medium
Ch.4	Growth orientated (e.g., challenge, learn new skills)	High
Ch.5	Need for competent supervising	Medium
Ch.6	Introverted (low need for social interaction)	Low
Ch.7	Need for involvement in personal goal setting	Medium
Ch.8	Need for feedback (needs recognition)	Medium
Ch.9	Need for geographic stability	Very low
Ch.10	Need to make a contribution (worthwhile/meaningful job)	High
Ch.11	Autonomous (need for independence)	High
Ch.12	Need for variety	High
Ch.13	Marketable	High
Ch.14	Need for challenge	High
Ch.15	Creative	High
Ch.16	Need to be sociable/identify with group	High

Table 10.9 Software engineer moderators and controls and GSD compatibility

	Moderators and controls	GSD compatible
Mod.1	Career stage	At stage that allows flexible working hours and travel
Mod.2	Culture	Open, interested in and tolerant of other cultures
Mod.3	Job type/role/occupational level	Applies to all development roles and levels, though inexperienced levels may not be suited to GSD
Mod.4	State of IT profession	Ideally, buoyant to support feeling of security
Mod.5	Type of organisation	Offers promotion opportunities, e.g., management, customer facing, domain specific, technical roles
Cont.1	Personality traits	Good communicator; not too introverted
Cont.2	Career paths (managerial/technical)	Fixed in a person: either type likely to be compatible with most GSD organisations
Cont.3	Competencies	Ability (both technical and managerial)

developers may be at a different stage in their careers than project managers. Sharp et al. (2007) found a difference in how software developers and project managers were motivated, and suggest that project managers should recognize these differences. Management approaches applied by project managers who assume that developers are motivated in a similar way to themselves are in danger of being ineffective or even detrimental. However, this finding does assume that roles are well defined. In GSD, this is not always the case (Richardson et al. 2012). Furthermore, the literature is divided as to whether clear role definition is a good thing. For example, Kennedy and Nur (2012) found that clear role differentiation can hinder effective project management. Motivation by *group* therefore will be difficult to achieve where group is defined by the role a practitioner plays in the development, if that role is not well defined.

While role ambiguity can be stimulating for the practitioner—and far from the Taylor approach of narrow and specialised work, it can be difficult to manage. The case study in this chapter highlights some advantages and disadvantages associated with role ambiguity: the individual may enjoy up skilling, developing a healthy CV with a broad skill set to market. However, if one member plays several roles, they can become over-stretched in terms of holding key knowledge, and having demands on their time from customers, sales force, head office, development team, etc. Also, the hours they are required to work can be unsustainable. Therefore, despite the challenging work, the individual may leave their employment if no time is allowed for their own needs and for a work-life balance.

10.6.1 Herzberg's Two-Factor Theory and the 'Crowding Out' Effect

Software engineers are motivated by internal factors such as challenging and varied work, fairness, participation, trust, respect, and social interaction (discussed earlier; see Tables 10.4 and 10.5). It is the careful management of these intrinsic factors that will result in a software engineer's increased commitment to the task. External factors such as rewards and salary also need to be managed in order to match the software engineer's expectations. However, internal and external needs must be finely balanced. For example, advantages gained from intrinsic motivation can be *crowded out* by placing too much emphasis on extrinsic motivators (Frey and Osterloh 2002). *Crowding out* can be explained as follows: An employee finds their job interesting and challenging, feels they are treated fairly, and feel that they are part of a team and have something specific to contribute. However, the motivation engendered by these intrinsic factors can be *crowded out*—"obscured by shifting the excitement connected with the job towards monetary reward." According to Frey and Osterloh (2002), "Offering extrinsic motivators, such as salary can actually switch someone's enjoyment and fulfillment from the job itself to doing the job for financial reward. That reward is often short lived." However, the authors add that extrinsic motivators cannot be ignored and that under certain circumstances they are indispensable (Frey and Osterloh 2002).

10.6.2 People, Process, and Creativity

We know from software engineer motivation research that the profession attracts people who are technical and creative with a very high need for challenging work (Beecham et al. 2008). This is reflected in, for example, the enjoyment derived from problem solving or learning a new programming language. The idea of working in the same small area, in piecemeal fashion, is anathema to the software practitioner.

As much as managers might strive to put processes in the place of dependence on people, this can never work in software development, since the task itself is so dependent on skilled people finding solutions to new problems and it is unlikely that two pieces of software will be identical when viewed as a whole.

The very job itself is difficult, hard to estimate, and challenging to get right in terms of meeting customer requirements, since each piece of software is in some way going to be different from other software written in the past. But that is the attraction, the challenge, and perhaps a key reason software engineers are attracted to this profession in the first place.

There is a tension between some management practices and software engineer creativity and job satisfaction. For example, it is difficult to work across sites operating in different time zones without structure: "While synchronous groups can often vary the degree and type of structure dynamically as needed, this is more difficult for distributed asynchronous groups that are dependent on both structure and process rules for coordination" (Ocker et al. 1995). Being dependent on structure fits Taylor's scientific management view that the way to maximize output was to discourage free thought and expect employees to follow prescribed steps (Kennedy and Nur 2012). It appears that controls and processes are orthogonal to creativity. According to Ocker et al., "too much structure, or the wrong structure, can limit the creative process." This is supported by Van de Walle et al., who note that the absence of structured task support led to greater satisfaction in their study of distributed teams (Van de Walle et al. 2007). To allow practitioners the freedom to be creative, the project manager must therefore aim for a correct balance between structure (seen as defined processes) and flexibility.

10.6.3 Returning to the Rational-Economic Model in GSD

Working remotely, managers are encouraged to keep the need for interaction among remote locations to a minimum (e.g., Richardson et al. 2012). It is, after all, these communications that can introduce difficulties such as misunderstandings, stress, and delays into the process, especially when working in different time zones, etc. Effective partitioning and allocation of work across the GSD team is something all managers need to plan for at the start of any project. There are several options to task allocation (Carmel 1999) and, according to Parnas (1972), managers can choose one or more of the following approaches: modularisation, phase-based, or integrated. Partitioning can be component based (Kotlarsky et al. 2007) or lifecycle based (Šmite 2007). Strategic partitioning of the development task can reduce the need for communication across teams.

However, when we consider the individual, the perceived productivity gains of working discretely are likely to be short-lived. Developing software is essentially a human intellectual and social activity (Ferratt and Short 1986; Burn et al. 1992; Jordan and Whiteley 1994; Garza et al. 2003; Sumner et al. 2005). If the work is viewed as repetitive, boring, and fragmented, then the individual may not feel part of the overall organization and may perceive their work to be meaningless. It is important for engineers' motivation that they perceive that their contributions matter (Ferratt and Short 1986; Crepeau et al. 1992; Garza et al. 2003). Research shows that monotony creates apathy, dissatisfaction, and carelessness (Crepeau et al. 1992; Peters 2003; Sumner et al. 2005; Ituma 2006), particularly when an individual does not develop new skills. However, another issue with task allocation by site is career advancement. For example, if a programmer desires to become a software architect, he or she needs to see a career path and be given an opportunity to learn related new skills. Working remotely can mean that the individual either does not have the scope to advance up the career ladder, or that they may be overlooked.

The concepts and theories relating motivation to a GSD context have been drawn from the literature, and the case study has been used to provide real-world examples of how these theories can be applied in practice. There are limitations to using one case study, and findings are used merely as indicators of where practices can help or hinder motivation. For example, some engineers might be highly motivated by salary, and provided they are well paid, they will continue to produce high-quality work despite many other motivators being challenged.

10.7 Summary and Conclusions

This chapter explored motivation theories and used a case study as an example of how to motivate software engineers working in virtual teams. Theories of motivation suggest that people, in whatever sphere, will have their own specific needs, and that it is the strength of those needs and the likelihood that they will be met by a given task that will determine the energy and enthusiasm the individual will expend on that task. However, despite the bespoke nature of motivation, the research does point to areas that need to be considered in every case. In every situation, the manager needs to balance three things: the *task*, the *environment*, and the software engineer's *characteristics*. If any of these are mismatched, no amount of stimulus from the job will result in sustained motivation.

Software development teams cannot be treated as a homogeneous group with similar characteristics. As it appears that engineers' needs differ according to the role they play, a way to manage engineers' motivation is by role. However, roles are an area that can be blurred, particularly in a global setting. The research and findings from the case study reported in this chapter have identified this as a problem. The demands placed on engineers working remotely can mean that they are encouraged to take on many roles to ensure project success. While keeping roles and responsibilities fluid might suit upper management, and even meet the software engineer's need for varied and challenging work, it can place unrealistic pressures on an individual's time.

This chapter listed factors known to motivate software engineers. Twenty-two different factors that motivate software engineers to produce high-quality software have been divided into intrinsic and extrinsic motivators. Case study findings, drawn from a company engaged in distributed software development, were mapped to each motivation factor. In this way, preliminary results were presented relating to areas that particularly threaten software engineers' motivation in a global setting.

Taking a needs-theory approach, managers must first consider which motivation factors they can control and which are outside their control. For example, the negative influence of extrinsic factors can be reduced by ensuring that employees are given adequate pay, a feeling of security, and good management. As global software development stems from the 'environment' it makes sense to view external factors as a separate threat or enhancement to motivation. Some environmental factors are outside the control of the manager. For example, no amount of best practice can change the culture of a country, the security of a job in a volatile economy, or a limited career path in a small organization.

The job characteristics theory (Hackman and Oldham 1974) emphasizes the importance of person-job fit. The GSD engineer needs to have certain characteristics in place that are resilient to the environment, leaving them free to be motivated by the intrinsic aspects of the work. Ideally, a software engineer recruited to work in a virtual team will be at a career stage that allows them flexibility to travel, flexibility in their place of work, and the hours they work. Also, they need to be open, tolerant, and interested in different cultures. They need a good level of confidence in their own ability, to know their own limitations, to be good and clear communicators, and on the extrovert end of the personality spectrum.

So, once the person-job fit has been matched, and extrinsic factors controlled for where possible, managers can turn their attention to the intrinsic factors that relate to the job itself. It is getting these factors right that will motivate software engineers to do the best job they can. Software engineers need technically challenging work, variety of tasks, and evidence that their efforts will result in a useful contribution. Engineers also require developmental training, to feel involved, recognition and rewards for doing a good job, and to be treated fairly, regardless of their location. Finally, they need trust and respect, responsibility, autonomy, and empowerment.

This chapter looked at how motivation theory can help to solve some GSD organisational problems such as low employee commitment and high turnover. It also examined how motivation might promote software quality and inspire innovation. However, an assumption associated with recommended management practices is that the employee stays in the organisation long enough to benefit from any motivation program. We have seen that heavy-weight processes can stifle innovation and that enforced compliance will demotivate the technical employee. A way forward could be to apply agile development methodologies and empower employees by creating an environment that allows a worker to develop a sense of ownership and pride of accomplishment (Kennedy and Nur 2012).

There is still more work required in this area. We need ways to measure the person-job fit for GSD. Also, although agile methods appear to address many software engineer motivation needs in co-located settings, we still need to know how to implement agile methods in a distributed setting so that motivation is positively affected.

Finally, when managers are allocating tasks to engineers, they would do well to heed the advice given by Herzberg:

If you want someone to do a good job, give them a good job to do.

Acknowledgments Thanks are extended to the many practitioners who gave their time and made this study possible. Also, thanks are given to the reviewers and colleagues who commented and proofread early versions of this chapter. This work was supported in part by Science Foundation Ireland grant 10/CE/I1855 to Lero – the Irish Software Engineering Research Centre (www.lero.ie).

References

Adams JS (1963) Toward an understanding of inequity. J Abnorm Social Psychol 67:422–436

Bassett-Jones N, Lloyd GC (2005) Does Herzberg's motivation theory have staying power? J Manage Develop 24(10):929–943

Beecham S, Baddoo N, Hall T, Robinson H, Sharp H (2008) Motivation in software engineering: a systematic literature review. Info Softw Technol (IST), Elsevier 50(9–10):860–878

Beecham S, O'Leary P, Baker S, Richardson I, Noll J (2013) Who are we doing global software development research for? In: 8th IEEE international conference on global software engineering (ICGSE'13), Bari, Italy

Beecham S, Sharp H, Baddoo N, Hall T, Robinson H (2007) Does the XP environment meet the motivational needs of the software developer? An empirical study. Agile 2007 conference. Washington, DC

Burn JM, Couger JD, Ma L (1992) Motivating IT professionals. The Hong Kong challenge. Info Manage 22(5):269–280

Carmel E (1999) Global software teams: collaboration across borders and time zones. Prentice Hall, Saddle River, NJ

Couger JD, Zawacki RA (1980) Motivating and managing computer personnel. Wiley, New York

Crepeau RG, Crook CW, Goslar MD, McMurtrey ME (1992) Career anchors of information systems personnel. J Manage Info Syst 9(2):145–160

da Silva FQ, França ACC (2012) Towards understanding the underlying structure of motivational factors for software engineers to guide the definition of motivational programs. J Syst Softw 85 (2):216–226

Deshpande S, Richardson I, Casey V, Beecham S (2010) Culture in global software development - a weakness or strength? In: IEEE international conferences on global software engineering (ICGSE 2010), Princeton, NJ

Fernández-Sanz L, Misra S (2011) Influence of human factors in software quality and productivity. In: Murgante B, Gervasi O, Iglesias A, Taniar D, Apduhan B (eds) Computational science and its applications - ICCSA 2011, vol 6786. Springer, Berlin, pp 257–269

Ferratt TW, Short LE (1986) Are information systems people different: an investigation of motivational differences. Manage Info Syst (MIS) Q 10(4):377–387

França ACC, Carneiro DE, da Silva FQ (2012) Towards an explanatory theory of motivation in software engineering: a qualitative case study of a small software company. 26th IEEE Brazilian Symposium on Software Engineering (SBES)

Frey BS, Osterloh M (2002) Successful management by motivation: balancing intrinsic and extrinsic incentives. Springer, Berlin

Garza AI, Lunce SE, Maniam B (2003) Career anchors of Hispanic information systems professionals. Proceedings - annual meeting of the decision sciences institute, pp 1067–1072

Hackman JR, Oldman GR (1976) Motivation through the design of work: test of a theory. Academic Press, New York

Hackman RJ, Oldham GR (1974) The job diagnostic survey: an instrument for the diagnosis of jobs and the evaluation of job redesign projects. Office of Naval research manpower administration: NCIS national technical information service, US Department of Commerce

Hall T, Beecham S, Baddoo N, Sharp H, Robinson H (2009) A systematic review of theory use in studies investigating the motivations of software engineers. ACM Transac Softw Eng Methodol (TOSEM) 18(3):10

Hall T, Beecham S, Verner J, Wilson D (2008a) The impact of staff turnover on software projects: the importance of understanding what makes software practitioners tick (Refilling the pipeline: meeting the renewed demand for information technology workers). In: ACM-SIGMIS CPR'08 Conference, Charlottesville, VA, April 3–5

Hall T, Sharp H, Beecham S, Baddoo N, Robinson H (2008b) What do we know about developer motivation? IEEE Softw 25(4):92–94

Handy C (1993) Understanding organisations, 4th edn. England Penguin Books Ltd, Middlesex

Herzberg F, Mausner B, Snyderman BB (1959) Motivation to work, 2nd edn. Wiley, New York

Hillegersberg Jv, Ligtenberg G, Aydin MN (2011) Getting agile methods to work for Cordys global software product development. In: Fifth global sourcing workshop, Courchevel 1850, France

Huczynski AA, Buchanan DA (1991) Organizational behaviour: an introductory text, 2nd edn. Prentice Hall, London

Ituma A (2006) The internal career: an explorative study of the career anchors of information technology workers in Nigeria Proceedings of the 2006 ACM SIGMIS CPR conference on computer personnel research: forty four years of computer personnel research: achievements, challenges & the future. Claremont, CA, pp 205–212

Jalali S, Wohlin C (2012) Global software engineering and agile practices: a systematic review. J Softw Evol Proces 24(6):643–659

Johnson RE, Chang C-HD, Yang L-Q (2010) Commitment and motivation at work: the relevance of employee identity and regulatory focus. Acad Manage Rev 35(2):226–245

Jordan E, Whiteley AM (1994) HRM practices in information technology management. In: Computer personnel research conference (SIGCPR) on Reinventing IS: managing information technology in changing organizations. ACM Press, Alexandria, VA

Kennedy D, Nur M (2012) The rise of taylorism in knowledge management. In: Proceedings of PICMET'12: technology management for emerging technologies (PICMET)

Kotlarsky J, Oshri I, von Hillegersberg J (2007) Globally distributed component-based software development: an exploratory study of knowledge management and work division. J Info Technol 22:161–173

Krippendorff K (1980) Content analysis an introduction to its methodology. Sage, Beverly Hills, CA

Locke EA (1968) Toward a theory of task motivation and incentives. Organ Behav Human Perfor 3:157–189

Maslow A (1954) Motivation and personality. Harper & Row, New York

McClelland DC (1961) The achieving society. Van Nostrand, Princeton, NJ

McConnell S (1996) Avoiding classic mistakes [software engineering]. IEEE Softw 13(5):111–112

Monasor MJ, Vizcaíno A, Piattini M, Noll J and Beecham S (2013) Towards a global software development community web: identifying patterns and scenarios. In: PARIS Workshop, International Conference on global software development (ICGSE), Bari, Italy

Mullins LJ (1993) Management and organisational behaviour. Pitman Publishing, London

Noll J, Beecham S, Richardson I (2010) Global software development and collaboration: barriers and solutions. ACM SIGCSE bulletin - special section on global intercultural collaboration (September)

Noll J, Beecham S, Seichter D (2011) A qualitative study of open source software development: the OpenEMR project. In: IEEE empirical software engineering and measurement conference – ESEM 2011, Banff, Canada, September, 19–23

Ocker R, Hiltz SR, Turoff M, Fjermestad J (1995) The effects of distributed group support and process structuring on software requirements development teams: results on creativity and quality. J Manage Info Syst 12(3):127–153

Olson JS, Olson GM (2004) Culture surprises in remote software development teams. ACM Q, Nova Iorque 1(9):52–59

Parnas D (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15(12):1053–1058

Peters L (2003) Managing software professionals. IEMC'03 proceedings. managing technologically driven organizations: the human side of innovation and change (IEEE Cat. No.03CH37502). IEEE, Albany, NY, pp 61–66

Petri HL, Govern JM (2012) Motivation: theory, research, and application, 6th edn. Wadsworth Publishing, Belmont, CA

Richardson I, Casey V, McCaffery F, Burton J, Beecham S (2012) A process framework for global software engineering teams. Info Softw Technol 54(11):1175–1191

Riehle D (2007) The economic motivation of open source: stakeholder perspectives. IEEE Comput 40(4):25–32

Roberts J, Hann I, Slaughter S (2004) Understanding the motivations, participation and performance of open source software developers: a longitudinal study of the Apache projects. Carnegie Mellon University Working Paper

Shah H, Nersessian NJ, Harrold MJ, Newstetter W (2012) Studying the influence of culture in global software engineering: thinking in terms of cultural models In: ACM proceedings of the 4th international conference on intercultural collaboration, Bengaluru, India

Sharp H, Baddoo N, Beecham S, Hall T, Robinson H (2009) Models of motivation in software engineering. Info Softw Technol 51(1):219–233

Sharp H, Hall T, Baddoo N, Beecham S (2007) Exploring motivational differences between software developers and project managers. In: The 6th joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE07), Dubrovnik, Croatia

Skinner BF (1976) Walden two. Macmillan, New York

Šmite D (2007) Global software development improvement. PhD Thesis, Riga Information Technology Institute, University of Latvia

Šteinberga L, Šmite D (2011) Towards a contemporary understanding of motivation in distributed software projects: solution proposal, vol 770. University of Latvia, Computer Science and Information Technologies, pp 15–26

Sumner M, Yager S, Franke D (2005) Career orientation and organizational commitment of IT personnel. ACM SIGMIS CPR conference on computer personnel research (Atlanta, Georgia, USA, April 14 16, 2005) pp 75–80

Taylor FW (1947) Scientific management. Harper & Row, New York

Van de Walle B, Campbell C, Deek FP (2007) The impact of task structure and negotiation sequence on distributed requirements negotiation activity, conflict, and satisfaction, published by LNCS

Verner J, Beecham S, Cerpa N (2010) Stakeholder dissonance: disagreements on project outcome and its impact on team motivation across three countries. In: ACM SIGMIS CPR'10, Vancouver, Canada

Verner J, Ali-Barbar M, Cerpa N, Hall T, Beecham S (2014) Factors that motivate software engineering teams: a four country empirical study. J Syst Softw 95:115–127

Vroom VH (1964) Work and motivation. Wiley, New York

Ye Y, Kishida J (2003) Toward an understanding of the motivation of open source software developers. Proceedings - International conference on software engineering, pp 419–429

Biography Sarah Beecham holds the position of research fellow in the Process Quality Group in Lero—the Irish Software Engineering Research Centre. Sarah's research interests include software fault prediction, effort estimation, evidence-based software engineering, requirements engineering, and software process improvement. She has published widely in the area of software engineer motivation.

Part III
New Paradigms

Introduction

Software development practices continuously evolve, and organizations introduce new ways of working. These evolving and new practices change the role of the project manager and the challenges experienced when performing the activities related to the role. For this part of the book, we have invited leading experts on some of these new paradigms in relation to software project management. The new paradigms include agile software development, distributed development, open source development and inner source development. This part consists of four chapters, in which the authors share their knowledge, insights and accompanying recommendations and conclusions.

In Chap. 11, Tore Dybå, Torgeir Dingsøyr and Nils Brede Moe characterize and define software project management in an agile setting. The authors start by describing traditional software project management, pinpoint its challenges and then move on to introduce project management in an agile setting. Next, the authors present several concepts that are typical in an agile setting such as self-managing teams, team leadership and feedback and learning. The chapter then describes four principles of agile software project management: minimum critical specification, autonomous teams, redundancy and feedback and learning. The authors elaborate on these four principles based on their experiences from working closely with several industry partners involved in agile software development.

Darja Šmite shares her experiences in relation to distributed software project management in Chap. 12. The author highlights ten misconceptions about project management for distributed software projects and discusses some lessons learned when working closely with industry in relation to global software development. The chapter structures a number of sourcing strategies and then goes into presenting the ten misconceptions. Each misconception is illustrated with a practical example from industry, and its implications for practice are discussed. Based on the presentation of the ten misconceptions, the chapter ends with some conclusions and reflections.

Chapter 13 discusses open source development. Ioannis Stamelos highlights the differences between traditional software development and open source development, and in particular he discusses the differences in how development and software project management are conducted. In open source, people and community organizations are emphasized. The chapter starts with a general introduction and comparison, in terms of both differences and similarities, between open source and more traditional software development. The author continues by describing management issues in relation to open source software development. Next, some challenges in the management of open source software development are highlighted. The chapter concludes with a discussion regarding future open source management techniques.

In Chap. 14, Martin Höst, Klaas-Jan Stol and Alma Oručević-Alagić discuss project management in inner source software development, i.e. where an organization uses principles from open source development for their software development. The chapter starts by positioning inner source in relation to traditional software development and presents a number of motivations for adopting inner source. Furthermore, the authors present a framework that highlights a number of key themes in project management to illustrate how traditional software project management differs from inner source project management. This is followed by two industry case studies to exemplify project management in inner source. Based on observations from these two case studies, a number of implications for using inner source software development in practice are presented. Given the nascent state of the inner source research area, the chapter concludes with several new research directions.

The four chapters in this part provide an in-depth insight into some of the new paradigms affecting software project management of today. The chapters highlight some of the trends that a project manager must be able to handle in their daily work. The chapters in this part highlight some of the new paradigms and hence challenges and opportunities that software project managers must be able to address in contemporary software development.

Chapter 11
Agile Project Management

Tore Dybå, Torgeir Dingsøyr, and Nils Brede Moe

Abstract Agile software development represents a new approach for planning and managing software projects. It puts less emphasis on up-front plans and strict control and relies more on informal collaboration, coordination, and learning. This chapter provides a characterization and definition of agile project management based on extensive studies of industrial projects. It explains the circumstances behind the change from traditional management with its focus on direct supervision and standardization of work processes, to the newer, agile focus on self-managing teams, including its opportunities and benefits, but also its complexity and challenges. The main contribution of the chapter is the four principles of agile project management: minimum critical specification, autonomous teams, redundancy, and feedback and learning.

11.1 Introduction

A software project can be seen as a collection of activities that create an identifiable outcome of value. In its simplest form, project management consists of planning, executing, and monitoring these activities (see Chap. 1). However, the high costs and failure rates of software projects continue to engage researchers and practitioners, and despite several advances, the effective management of software projects is still a critical challenge.

This challenge has led to extensive interest in agile software development in the past decade (Dingsøyr et al. 2012). A number of methods have emerged that describe practices for development phases at the team, project, and organizational

T. Dybå (✉) • T. Dingsøyr • N.B. Moe
SINTEF, Trondheim, Norway
e-mail: tore.dyba@sintef.no; torgeird@sintef.no; nilsm@sintef.no

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_11, © Springer-Verlag Berlin Heidelberg 2014

levels (Abramson et al. 2010). Scrum is the method that most clearly addresses software project management (Schwaber and Beedle 2001).

In agile software development, developers work in teams with customers that represent the system's users. The features to be implemented in each development cycle are jointly decided by the customer and the rest of the development team. Augustine et al. (2005) describes the role of the software project manager as one of facilitating and working with a team in making project-related decisions.

Our objective in this chapter is to provide software project managers with a set of principles for handling the complexity and uncertainty inherent in agile software projects. The rest of this chapter is organized as follows: Sect. 11.2 describes challenges and recent developments in software project management. Section 11.3 explains the role of self-managing software teams, while Sect. 11.4 discusses the leadership of such teams. Section 11.5 describes the importance of feedback and learning. Finally, Sect. 11.6 presents a set of principles for agile project management, while Sect. 11.7 concludes the chapter.

11.2 Software Project Management

Managing the unique and complex processes that constitute a project involves the implementation of specific management activities. In software development, as in most other businesses, there has been a tendency toward standardizing these activities by means of formalized, generic project management methodologies like, PRINCE2,[1] which was developed and championed by the UK government. Although there is a global conception of the project management phenomenon, there is no unified theory of project management (Garel 2013) or well-defined measures of project success (see Chaps. 2 and 5).

11.2.1 Traditional Project Management

Traditional project management largely derives from the linear structure and discrete, mechanical views of the systems engineering and quality disciplines of the 1950s and 1960s. Basically, traditional project management views development as a linear sequence of well-defined activities such as requirements, design, coding, and testing. It assumes that you have almost perfect information about the project's goal and expected solution. As a consequence, it does not easily accommodate for deviations in scope, schedule, or resources.

Hardware development seemed to fit well into the traditional approach. However, due to its intangible nature, software was not equally well understood and, as a

[1] www.prince-officialsite.com

consequence, software development did not fit well into the same approach. To counter this, the term “software engineering” was coined at the historic NATO conference in Garmisch-Partenkirchen in 1968 as a solution to these problems in software development, implying the need for software development to be based on the principles and practices seen in engineering.

Thus, the point of departure for most of the subsequent efforts in addressing the problems in software development has been to treat the entire task of software development as a process that can be managed through engineering methods. Hoare (1984), for example, considered the “rise of engineering” and the use of “mathematical proof” in software development as a promise to “transform the arcane and error-prone craft of computer programming to meet the highest standards of a modern engineering profession.” Likewise, Lehman (1989) focused on reducing uncertainty in the development process through the “engineering of software.”

Humphrey (1989) recognized that the key problems in software development are not technological, but managerial in nature (see also Chap. 1). Consequently, he developed a framework for managing and improving the software process, which was later known as “The Capability Maturity Model for Software” (CMM). Consistent with the views of software engineering, the CMM, and its successor CMMI, is rooted in the engineering tradition, emphasizing predictability and improvement through the use of statistical process control. Humphrey (1989) formulated his fundamental view in this way: “If the process is not under statistical control, sustained progress is not possible until it is.”

As these examples from some of the most influential academic leaders within the software community show, software development and software project management are strongly rooted in the rationalistic traditions of engineering. Indeed, most of the writings to date can be seen to have antecedents in the industrial models devised by Frederic Winslow Taylor and Henry Ford, and also in the model of bureaucracy described by Max Weber.

Contemporary project management methodologies, like PRINCE2, are standardized, process-driven project management methodologies, which build on this engineering tradition and that contrast with reactive and adaptive methods such as Scrum. What many seem to forget, is that the acronym PRINCE stands for “PRoject IN Controlled Environment.” It should not come as a surprise then that it does not fit equally well within the environment in which many, if not most, software projects operate.

11.2.2 Challenges of Software Project Management

Although there are several challenges with traditional project management principles, two are especially important for the management of software projects: complexity and uncertainty. Project complexity means that the many different actions and states of the software project and its environmental parameters interact, so the effects of actions are difficult to assess (Pich et al. 2002). In complex software projects, an adequate representation of all the technological, organizational, and environmental states that might have a significant influence on the project's outcome of value, or of the causal relationships, is simply beyond the capabilities of the project team.

Most of the classic problems of developing software derive from this essential complexity and its exponential increase with size; for example, it is estimated that for every 25 % increase in problem complexity, there is a 100 % increase in complexity of the software solution (Woodfield 1979). A further challenge is that the information needed to understand most software problems depends upon one's idea for solving them. The kind of problems that software projects deal with tend to be unique and difficult to formulate and solutions tend to evolve continually as developers gain a greater appreciation of what must be solved (Nerur and Balijepally 2007).

Adding to the complexity of the problem and its solution is the fast-changing and highly uncertain environment, for example, market turbulence and changes in customer requirements and project goals. It is necessary therefore to accept that our assumptions and predictions about future events will, by nature, be uncertain. When managing software projects, we need to be extremely cautious of extrapolating past trends or relying too heavily on past experience. Trends peter out, and the future is full of unexpected developments as well as unpredictable human behavior. The greater the uncertainty inherent in a project, the more the team have to move from traditional approaches that are based on a fixed sequence of activities to approaches that allow to redefine the activities—or even the structure of the project plan—in midcourse (De Meyer et al. 2002). Therefore, as the project complexity and uncertainty increase, managers need to go beyond traditional risk management; adopting roles and techniques oriented less toward planning and more toward flexibility and learning.

11.2.3 From Traditional to Agile Project Management

The position taken in this chapter regarding software project management is strongly influenced by socio-technical theory (Trist 1981). Its central conception is that organizations are both social and technical systems, and that the core of the software organization is represented through the interface between the technical and human (social) system. From an engineering perspective, however, the world is composed of problems whose existence is distinct from the methods, tools, and practices of software development. The technical rationality behind this worldview emphasizes "objective truths" and global "best practices" at the expense of local context and expertise. An important aspect of socio-technical theory, however, is the belief that there may be many optimal solutions—or best ways—to a specific problem, since the "joint optimization" of a particular technical and human system can be implemented in several ways that can be equally efficient. We therefore

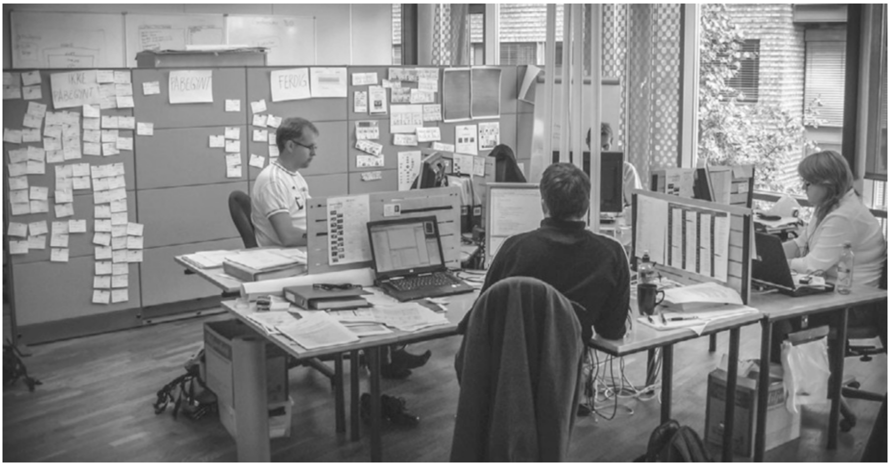


Fig. 11.1 The work environment of an agile development team

reject the assumption that complexity, uncertainty, and change can be controlled through a high degree of formalization

At its core, *agile project management* is about managing the impact of complexity and uncertainty on a project, recognizing

- The need for a dramatically shorter time frame between planning and execution
- That planning an action does not provide all the details of its implementation
- That creativity and learning are necessary to make sense of the environment

Agile project management is based on the same principles found in the Agile Manifesto.[2]

Therefore, unlike the linear sequence of well-defined activities of traditional project management, agile project management is characterized by short cycles of iterative and incremental delivery of product features and continuous integration of code changes. Agile project management introduces changes in management roles as well as in practices. Scrum, for example, defines three roles in software projects: development team members, a facilitator, and a product owner. A typical work environment for an agile team is shown in Fig. 11.1.

The task of the facilitator is to organize meetings of the development team and to make sure that the team addresses any obstacles they encounter. The task of the product owner is to prioritize what is to be developed. Apart from that, the team should be self-managed. In practice, however, many companies also appoint a project manager to assist a product owner in working on requirements and to handle other matters than those directly related to software development, such as internal and external reporting.

[2] http://agilemanifesto.org/

However, the introduction of agile development does not change the fundamental knowledge required to develop software, but it does change the nature of collaboration, coordination, and communication in software projects. Moving from traditional to agile project management implies a shift in focus from extensive *up-front planning* to the crucial decisions that are made during the execution of the project. Most importantly, moving from traditional to agile development implies dealing with complexity and unpredictability by relying on people and their creativity rather than on standard processes (Dybå 2000; Conboy et al. 2011), and thus moving from command and control to shared decision-making and self-management in software teams.

11.3 Self-Managing Software Teams

Teams are the fundamental organizational unit through which software projects are executed. A team structure brings business and process knowledge together with design and programming skills. However, three challenges characterize the effectiveness of software teams (Faraj and Sambamurthy 2006). First, the expertise required for the completion of software tasks is distributed across team members who must find effective ways of collaboration, knowledge sharing, and problem solving. Second, software projects need a combination of formal and informal modes of control with appropriate expertise in knowing how to exercise the appropriate combinations of control strategies during the execution of the project. Finally, software projects are characterized by varying levels of task uncertainty and coordination challenges. As a result, the software teams must be capable of dealing with the resulting ambiguity of their project tasks.

In accordance with contemporary perspectives, we conceptualize the software team as embedded in a multilevel system of individual-, team-, and organizational-level aspects (Kozlowski and Ilgen 2006; Moe et al. 2009). This conceptualization is important in our effort to make actionable recommendations on agile software management.

With the introduction of agile software development, self-managing software teams have become widely recommended. While organizing software development in self-managing teams has many advantages such as increased productivity, innovation, and employee satisfaction, it is not enough to put individuals together and expect that they will automatically know how to work effectively in such teams. Succeeding with managing agile teams requires a full understanding of how to create and maintain self-managing teams.

One can argue that leading self-managing teams is more challenging than leading traditional teams, because the project manager needs to enable shared leadership (the opposite of centralized leadership), shared decision-making, shared mental models, and a constant learning and improvement process. This takes time. What makes the leadership of such teams even more difficult is the fact that software development teams are typically formed anew for each project, depending on project requirements and who is available (Constantine 1993). It is extremely rare for an entire team to move from one project to another.

Self-managing teams are also known as autonomous or empowered teams. While self-managing teams represent a radically new approach to planning and managing software projects, the notion of self-management is not new; research in this area has been conducted since Eric Trist and Ken Bamforth's study of self-regulated coal miners in the 1950s (Trist and Bamforth 1951).

Self-management can be understood as a strategy for learning and improving a software team itself since it can directly influence team effectiveness, improvement work, and innovation. Self-management has also been found to result in more satisfied employees, lower turnover, and lower absenteeism (Cohen and Bailey 1997). Others also claim that self-managing teams are a prerequisite to the success of innovative projects (Takeuchi and Nonaka 1986), especially the innovative software projects (Hoegl and Parboteeah 2006). Furthermore, having team members cross-trained to do various jobs increases functional redundancy, and thus the flexibility of the team in dealing with personnel shortages. Even though there are several studies on the benefits of self-managing teams, there is substantial variance in research findings regarding the consequences of such teams on such measures as productivity, turnover, and attitudes (Guzzo and Dickson 1996).

Self-managing teams offer potential advantages over traditionally managed teams because they bring decision-making authority to the level of operational problems and uncertainties and thus increase the speed and accuracy of problem solving. Companies have implemented such teams to reduce costs and to improve productivity and quality. However, effective self-managing units cannot be created simply by exhorting democratic ideals, by tearing down organizational hierarchies, or by instituting one-person-one-vote decision-making processes (Hackman 1986). Hackman identified five general conditions that appear to foster and support self-management:

- Clear, engaging direction
- An enabling performing unit structure
- A supportive organizational context
- Available, expert coaching
- Adequate resources

To succeed with creating and maintain a self-managing agile team, the project manager must enable all these conditions.

Understanding the different levels of autonomy is also important for being able to succeed as an agile project manager. An agile team needs autonomy on both the team and the individual levels (Moe et al. 2008). The project manager must ensure that

- The team has authority to define work strategies and processes, project goals, and resource allocation
- All team members jointly share decision authority (what group tasks to perform and how to carry them out)
- A team member must have some freedom in carrying out the assigned task

The conflict between individual and team autonomy is one main reason why it is challenging to establish well-functioning agile teams. If individuals are independent and mostly focus on their own schedule and implementation of their own tasks, there will be less interaction between the group members, which will threaten the teamwork effectiveness. However the self-managing team may end up controlling group members more rigidly than they do under traditional management styles, which will reduce the motivation for the individual team member. The question is then: How can an agile project manager balance team level autonomy and individual level autonomy in agile software teams? This is especially challenging when development is done in market-driven agile projects with fixed scope and deadlines.

11.4 Team Leadership

Most models of team effectiveness recognize the critical role of team leadership. However, examining the extensive literature on leadership theories is beyond the scope of this chapter. Still, a relatively neglected issue in the current literature is what leaders should actually be doing to enhance team effectiveness (Kozlowski and Bell 2003). Many leadership theories focus on leading individuals rather than leading in a team context. In this section, we examine the functional role of team leaders and discuss leadership and decision-making in the context of self-managing teams. A recent focus group study on agile practitioners shows that planning, shielding the team from interruptions, agreeing on a work process, ensuring adequate resources, and setting up a working technical infrastructure are seen as important aspects of team leadership (Dingsøyr and Lindsjørn 2013).

In a self-managing team, members have responsibility not only for executing the task but also for monitoring, managing, and improving their own performance (Hackman 1986). Furthermore, leadership in such teams should be diffused rather than centralized (Morgan 2006). Shared leadership can be seen as a manifestation of fully developed empowerment of a team (Kirkman and Rosen 1999). When the team and the team leaders share the leadership, it is transferred to the person with the key knowledge, skills, and abilities related to the specific issues facing the team at any given moment (Pearce 2004). While the project manager maintains the leadership for project management duties, the team members lead when they possess the knowledge that needs to be shared during different phases of the project (Hewitt and Walz 2005).

11.4.1 Shared Decision Making

Product- and project-level decisions in a software company can be considered at the strategic, tactical, and operational levels (Aurum et al. 2006). In traditional development decision-making is governed by the hierarchical command and control

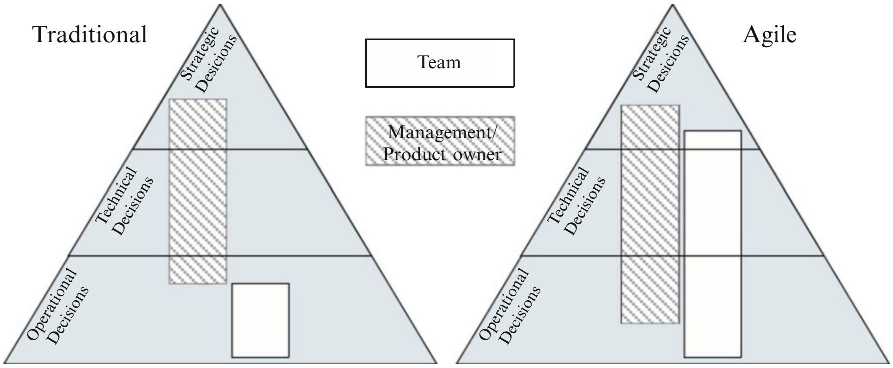


Fig. 11.2 Traditional and agile models of decision-making (Moe et al. 2012)

structure, while in agile development team empowerment and shared decision-making is encouraged at all levels (Fig. 11.2). In most organizations, you will find a mixture of these two ways of making decisions. In an agile product company, strategic decisions are primarily related to product and release plans, which may require creativity and opportunistic inputs, and should be based on an accurate understanding of the current business process and a detailed knowledge of the software product. Tactical decisions in such companies involve the project management view, where the aim is to determine the best way to implement strategic decisions, that is, to allocate the resources. On the other hand, operational decisions in an agile company are about implementation of product features and the process of assuring that specific tasks are carried out effectively and efficiently (Moe et al. 2012).

Adaptability is essential in agile teams since strategic decisions are made incrementally while important tactical and operational decisions are delayed as much as possible, in order to allow for a more flexible response to last minute feedback from the market place. Because the self-managing team is responsible for solving operational problems and uncertainties, this increases the speed and accuracy of problem solving, which is essential when developing software.

While there are several benefits with shared decision-making in agile teams, there also exist some challenges. First, the shared decision-making approach, which involves stakeholders with diverse backgrounds and goals, is more problematic than the traditional approach where the project manager is responsible for most of the decisions (Nerur et al. 2005). Also, despite the benefits of shared decision-making, cohesion has been indicated as a source of ineffective or dysfunctional decision-making; perhaps the most noted problem associated with team cohesion is groupthink. Finally, it is important to understand that not every decision must be made jointly with equal involvement by every team member, rather the team can also delegate authority to individuals or subgroups within the team. The challenge is to understand which team member is supposed to be involved in which decisions.

In agile software development, one important forum for shared decision-making is the standup meeting because it is about coordinating and planning the daily work.

The meeting is supposed to be short, and the purpose of the meeting is to improve communication, highlight and promote quick decision-making, and identify and remove impediments.

The daily meeting is a place where the agile software team members use their experience to make decisions in a complex, dynamic, and real-time environment. To understand the effect of decision making in such a complex environment with time pressure, the theory of naturalistic decision making (NDM) (Meso et al. 2002) is useful. NDM postulates that experts can make good decisions under difficult conditions, such as time pressure, uncertainty, and vague goals, without having to perform extensive analyses and compare options. The experts are able to do so by employing their experience to recognize problems that they have previously encountered and for which they already developed solutions. Experts use their experience to form mental simulations of the problem currently being encountered and use these simulations to suggest appropriate solutions.

There are certain implications of viewing daily meetings as an NDM process. First, the agile project manager needs to make sure that employees are trained to develop domain-specific expertise and collaborative teamwork skills. Second, because NDM relies on highly trained experts, the agile project manager needs to make sure the team consists of experts, not novices. If there are novices in the team, the agile project manager needs to determine how to move the novices through the stages until they become experts. Third, to make teams perform effective decision-making processes in such meetings, the project manager needs to make sure the team members have developed a shared mental model; that is, they must have a shared understanding of who is responsible for what and of the information and requirements needed to solve the tasks (Lipshitz et al. 2001).

11.4.2 Escalation of Commitment

One major source of error in decision-making is escalation of commitment (Stray et al. 2012). Escalating situations happen when decision-makers allocate resources to a failing course of action (Staw 1976). It is a general phenomenon that is particularly common in software projects due to their complex and uncertain nature. Keil et al. (2000), for example, found that 30–40 % of all software projects experience escalation of commitment.

Several studies have shown that decision-makers tend to invest additional resources in an attempt to justify their previous investments (Bazerman et al. 1984). Because groups have the capacity of employing multiple perspectives when making decisions, one might believe that escalating commitment situations should occur less frequently in agile teams than in traditional teams. However, several studies show that when having group decision-making, escalating tendencies will occur more often and will be more severe than in individual decision-making due to group polarization and conformity pressures (Whyte 1993).

To avoid situations of escalating commitment in agile projects, it is important to make sure that the team meetings do not become a place for defending decisions (Stray et al. 2012). Not only do the teams need to watch their internal process, they also need to consider which non-team members are allowed to participate or observe the team meetings. If team members start to defend their decisions or give detailed reports of what they have done because people outside the team are present in, for example, the daily meetings, we advise that these people outside the team do not participate on a regular basis.

Early signs of escalation in, for example, the daily meeting such as rationalizing continuation of a chosen course of action, and when team members start giving detailed and technical descriptions of what they have done since last meeting, must be taken seriously. Further, when the team becomes aware of the signs of escalating commitment, this needs to be addressed in the retrospective meetings (see next section).

11.5 Feedback and Learning

Agile software development is a type of knowledge work (see Chap. 7) where feedback and learning is particularly important. In order to adapt to changes in technology and customer requirements and to reduce the risk in development, agile development methods rely on frequent feedback loops both within the development team and with external stakeholders. The focus on having a “shippable product” leads to feedback on technical problems, and the demonstration of the product at the end of iterations leads to feedback on the developed functionality. The feedback represents opportunities for learning, which can lead to changes in product and personal skills, as well as changes in the development process.

With the focus in agile development on “working software” and “individuals and interactions,” knowledge is managed in a very different manner than in traditional software development. Traditional development has often focused on managing explicit knowledge in the form of written lessons learned in knowledge repositories and documented procedures in electronic process guides (see Chap. 7). Agile methods focus on managing knowledge orally, which means that dialogue is the main method of transfer (Bjørnson and Dingsøyr 2008). Von Krogh et al. write that “it is quite ironic that while executives and knowledge officers persist in focusing on expensive information-technology systems, quantifiable databases, and measurement tools, one of the best means for knowledge sharing and creating knowledge already exists within their companies. We cannot emphasize enough the important part conversations play” (von Krogh et al. 2000).

A well-known theory of learning that focuses on feedback is Argyris and Schön’s theory of single- and double-loop learning (Argyris and Schön 1996). Double-loop learning is distinguished from single-loop learning in that it concerns the underlying values. If an agile team repeatedly changes practices without solving their problems, this is a sign that they have not understood the underlying causes of their problem and is practicing single-looped learning. Lynn et al. (1999) argue that learning has a direct impact on cycle time and product success, and have identified two factors that are central for learning: capturing knowledge, and a change in behavior based on the captured knowledge. Practices that must be in place for this to happen are, among others, recording and reviewing information, have goal clarity, goal stability, and vision support.

There are, however, often challenges in agile teams to make use of opportunities for learning. A study by Stray et al. (2011) reports that many teams spend little time reflecting on how to improve how they work and they do not discuss obvious problems. Some of the teams that carry out regular retrospective meetings struggle to convert their analysis into changes in action. Among those who try to remedy identified problems actively, several give up after seeing little change.

Learning is challenging but crucial. A stream of research has established that teams who have shared mental models about product, tasks, and process work more effectively. Further, developing overlapping knowledge, sometimes referred to as knowledge redundancy, is critical in turbulent environments where people need to work on tasks assigned by priority rather than competence of team members. In the following, we discuss some of the main arenas for feedback and learning in agile development: the project kick-off and retrospectives after iteration and release.

11.5.1 Agile Project Kick off

Kick-off is one of the most used tools of project management (Besner and Hobbs 2008). Typical activities in a kick-off meeting are describing a vision for the project, establishing roles, project stakeholders, and planning the project. A vision or overall goals of the project will usually be defined by the customer, what is referred to as the product owner in Scrum. Further, in agile methods, the team is seen as a self-managing team, with one person facilitating the work of the team. Thus, the only internal roles are the team facilitator and team members. However, companies often also appoint a project manager, especially in multi team projects. The project stakeholders are usually represented by one product owner, but can also be other people from the customer who have an interest in the product to be developed, or other development projects, for example, when sharing a common technical infrastructure. A picture from a project kick-off is shown in Fig. 11.3.

As for planning the project, one important decision is the duration of an iteration. If there are frequent changes in customer requirements or technology, this calls for shorter iterations, while a more stable environment calls for longer iterations. Normally an agile team will make a rough plan for several iterations and a detailed plan for the next one. The detailed plan can be made at the kick-off by making a product owner give priorities to the set of features that is to be developed. The features are estimated, for example, using the technique planning poker, which facilitates a discussion between team members on what tasks must be performed to develop a feature. The team then commits to what they will be able to deliver in the

Fig. 11.3 Project kick off with development team, team facilitator, and customer responsible

first iteration. The "plan" for the team is then a list of prioritized features, and who is to perform the tasks of developing the features is decided on during the iteration.

What is important in the kick-off meeting to enable feedback and learning? From studies of shared mental models, we know that teams need to establish shared knowledge on a number of areas to function effectively. Shared mental models comprise knowledge of the tasks, technology, team members' skills, and interactions. The planning poker technique is one way to improve shared mental models. Estimation discussions can provide knowledge of tasks at hand, the technology used in development, as well as demonstrating team member skills (Fægri 2010).

Planning poker is carried out as follows: Every individual is given a set of playing cards with values loosely in a Fibonacci sequence, usually 0, 1, 2, 3, 5, 8, 13, 20, 40, and ∞. For each task, individuals decide on a card that represents the amount of work; this can be in number of hours or relative to a standard task. All team members show their cards, and the person with the highest and lowest estimates is asked to explain their reasoning. The process is repeated until consensus, or if consensus is unlikely a number is set based on majority vote or average of votes. If there is much divergence, it might also be necessary to decompose a task into smaller tasks that are easier to estimate. See Chap. 3 for a further discussion of estimation, and there are also some studies available on the use of planning poker as an estimation technique (Molokken-Ostvold et al. 2008).

Finally, that everyone has a clear image of team interaction is accomplished by having clear work processes. Agile methods are simple and easy to remember, which makes it easy to function as a shared mental model.

11.5.2 The Retrospective

A retrospective (or postmortem review (Birk et al. 2002) or post-iteration workshop (Outi 2006)) is a collective learning activity after an iteration or release (Dingsøyr 2005). The main motivation is to reflect on what happened in order to improve future practice—for the individuals that have participated in the project and possibly also for the organization as a whole.

Researchers in organizational learning use the term "reflective practice" (Dybå et al. 2014), which is "the practice of periodically stepping back to ponder on the meaning to self and others in one's immediate environment about what has recently transpired. It illuminates what has been experienced by both self and others, providing a basis for future action" (Raelin 2001). This involves uncovering and making explicit results of plans, observation, and achieved practice. It can lead to understanding of experience that has been overlooked in practice. Kerth argues that a retrospective can help members of a community to understand the need for improvement and motivate them to change. The retrospective helps the community to become "master of its software process" (Kerth 2001). In addition, retrospectives are claimed to foster learning, growth, and participant maturity, and provides an opportunity to celebrate success. Derby and Larsen further suggest that retrospectives lead to improved productivity, capability, quality, and capacity; the purpose is "whole-team learning" (Derby and Larsen 2006).

A typical agile retrospective will be conducted with activities to gather data, generate insight, and make decisions (ibid). To gather data, exercises such as plotting important events on a timeline or just brainstorming on "what went well" and "what could be improved" are typical. Insights are generated by analysis of the material, through use of fishbone diagrams, structuring of data and prioritization (see Fig. 11.4). Decisions about changes are made on this basis and are planned as tasks in the next iteration.

Although retrospectives today is a very common practice, there has been little research on this topic. Most works concentrate on describing approaches to conduct retrospectives, with little focus on the effects. However, in a survey on essential practices in research and development companies, "learning from post-project audits" were found to be one of the most promising practices to yield competitive advantage (Menke 1997).

Kransdorff (1996) criticizes postmortems because people participating do not have an accurate memory, which can lead to disputes. He suggests collecting data during the project, for example, through short interviews, in an effort to get more objective material.

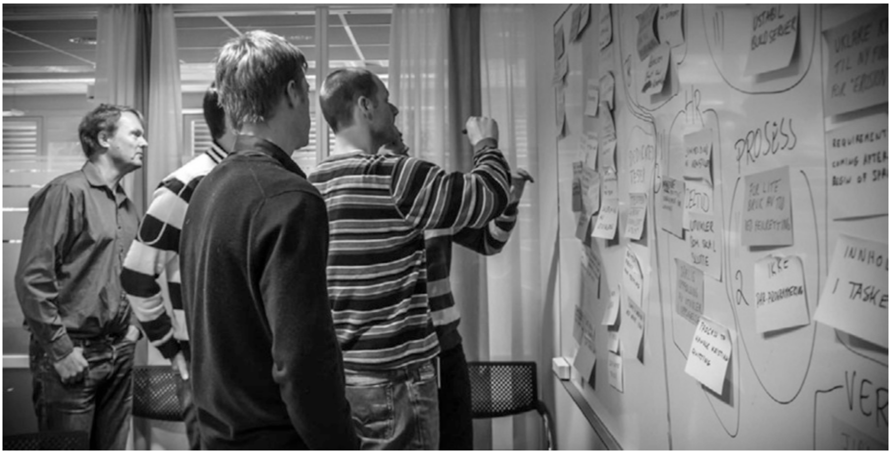


Fig. 11.4 A retrospective in a development team with a group of developers structuring the results of a brainstorming session

11.5.3 *Visualizing Project Status*

Many teams use visual boards, kanbans,[3] or "information radiators" as a central element for collaboration, coordination, and communication (Sharp and Robinson 2010). A board usually displays tasks on cards, and the placement of cards on a board shows their status. Teams have found that such boards make meetings efficient. Participants point at cards on the board to show what team members work on, and the board shows progress in the project.

Physical artifacts are easy to refer to, easy to annotate, and hard to ignore (Sharp et al. 2006). A physical board makes it easier to limit the amount of information unlike an electronic system, which is often the alternative. Such boards can help giving teams a shared mental model of the project status, importance of tasks, and how ready the product is for delivery.

A visual board can be set up quickly by placing a board in a relevant location, deciding on how to organize the board, and supplying cards to put on the board. Find a location, which is visible both for the development team and for others who have interest in the work of the team. The board could be placed with other visual information the team is using, for example, a burndown[4] chart, which shows the remaining work in this phase.

The board should show important information about status and progress of the work of the team. This can be done by dividing the board into relevant phases that

[3] A kanban is a visual card system for organizing production according to demand, central in lean production.

[4] A burndown chart shows the estimated remaining work in an iteration, and is updated daily when teams use the Scrum development process.

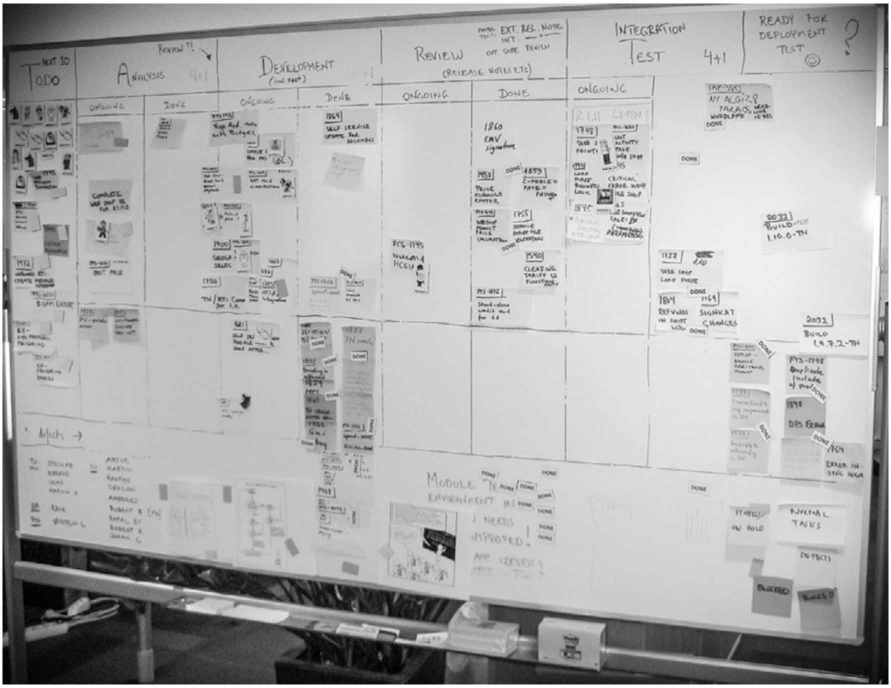


Fig. 11.5 Example visual board with areas for tasks "todo," "analysis," "development," "review," "integration test," and "ready for deployment test"

work tasks go through. Typical phases include "to do," "analysis," "development," "review," "integration test," and "ready for deployment test," as shown in Fig. 11.5. If your team has particular problems, for example, if it is unclear for developers whether a task is completed or not, you can add a phase for checking that either another developer or an external person agrees that the task is completed. Some also choose to mark problems in the project either through putting tasks in an area for problems or by marking tasks with a different color.

Physical artifacts like the card represent tokens of responsibility, and moving artifacts have been found to give more insight than electronic manipulation tools (Sharp et al. 2006), which is the alternative many teams use. A visual board makes it easy to discover common problems in a project like: tasks do not get completed, important tasks are not done, and if too many tasks are started at the same time.

11.6 Principles of Agile Project Management

A fundamental property of software is its nonphysical form; software code is essentially a large set of abstract instructions possessing unlimited complexity, flexibility, and revisability. Software exhibits nonlinear behavior and does not

Table 11.1 Principles of agile project management

Minimum critical specification	No more should be specified than is absolutely essential and critical to overall success
Autonomous teams	Autonomous teams are responsible for managing and monitoring their own processes and executing tasks
Redundancy	Team members should be skilled in more than one function
Feedback and learning	Feedback and learning are integral to project execution and the project's interaction with the environment

conform to laws of nature. One consequence is that it is inherently hard to build models of software that allow accurate reasoning about the system's qualities (Fægri et al. 2010). Agile project management addresses these basic properties of software and breaks away from the linear sequence of well-defined activities of traditional project management. It shifts focus from up-front planning to execution. In doing so, agile project management moves from traditional command and control structures to shared decision-making, self-management, and learning in software teams to deal with the complexity and unpredictability of the problem-solving activities of software projects.

Based on our extensive experience and studies of a multitude of agile projects during the last decade (Moe et al. 2009, 2010; Moe et al. 2012; Dybå and Dingsøyr 2008; Šmite et al. 2010; Dingøyr et al. 2012; Dingsøyr et al. 2010; Dybå 2011), we offer the following set of socio-technical principles of agile project management (see Table 11.1).

11.6.1 Minimum Critical Specification

This principle has two aspects; the first is that no more should be specified than is absolutely essential; the second requires that the team identify what is critical to overall success. This means that the system requirements should be precise about what has to be done, but not about how to do it, and that the use of rules, standards, and predefined procedures is kept to an absolute minimum. Focus should be on the larger system requirements and boundary conditions, leaving as many design decisions as possible to those closest to the work.

Understanding "the problem" that the system is intended to address is one of the keys to project success. Therefore, this principle is oriented toward the analysis and problem understanding that will help the project's stakeholders to focus on the nature of the overall problems and issues and come to some agreement about what these really are. It will also help the software team to understand the problems—rather than what they perceive as being the "problem"—the system is supposed to solve.

Additionally, complex and turbulent environments require software projects to be highly adaptable. Thus, specifying more than is needed closes options that should be kept open for as long as possible.

Successfully dealing with this principle requires that the project establishes shared mental models about the problem and its solution, as well as about tasks, technology, team member skills, and interactions. The project's kick-off meeting is crucial for achieving this.

11.6.2 Autonomous Team

This principle is based on the premise that autonomous, or self-managing, teams are a prerequisite for the success of innovative software projects. Such teams offer potential advantages over traditionally managed teams because they bring decision-making authority to the level of operational problems and uncertainties and thus increase the speed and accuracy of problem solving.

Members of autonomous teams are responsible for managing and monitoring their own processes and executing tasks. They typically share decision authority jointly, rather than having a centralized decision structure where one person makes all the decisions or a decentralized decision structure where team members make independent decisions.

However, there are important individual and organizational barriers and challenges to successfully applying autonomous teams in software development (Moe et al. 2009). Misalignment between team structure and organizational structure can be counterproductive, and attempts to implement autonomous teams can cause frustration for both developers and management. Shared resources, organizational control, and specialist culture are the most important barriers that need to be effectively dealt with in order to succeed.

Furthermore, autonomy at the team level may conflict with autonomy at the individual level; when a project as a whole is given a great deal of autonomy, it does not follow that the individual team members are given high levels of individual autonomy. It is a danger, therefore, that the self-managing team may end up controlling team members more rigidly than they do under traditional management styles. Thus, it is imperative to ensure that individual developers have sufficient control over their own work and over the scheduling and implementation of their own tasks.

For autonomous teams to thrive, it is thus necessary to build trust and commitment in the whole organization, avoiding any controls that would impair creativity and spontaneity. The team's need for continuous learning, not the company's need for control, should be in focus. So, make sure that both the organization and the teams know and respect the project's objective.

11.6.3 Redundancy

This principle is concerned with the overlap in individuals' knowledge and skills in order to create common references for people's creation of new knowledge; as the level of redundancy increases within the team, individuals will find it easier to share new knowledge and the project will be able to coordinate its work more effectively Therefore, this principle implies that each member of the team should be skilled in more than one function so that the project becomes more flexible and adaptive, which allows a function to be performed in many ways utilizing different people.

Having such redundancy, with team members cross-trained to do various jobs, increases the project's functional redundancy and thus the flexibility of the team in dealing with personnel shortages. Redundancy is also critical in turbulent environments where people need to work on tasks assigned by priority rather than the competence of team members.

A particular challenge, however, is that individual specialization, high levels of proficiency, and the ability to solve more complex problems are often more important motivations for people than to seek overlapping knowledge. It is essential, therefore, with a greater focus on redundancy at the organizational level surrounding the project; rather than viewing redundancy as unnecessary and inefficient, the organization must appreciate both generalists and specialists to build redundancy into its projects.

11.6.4 Feedback and Learning

Without feedback and learning, agile project management is not possible. The focus on project execution rather than on up-front planning in agile projects, leads to an intertwinement of learning and work, and of problem specification and solution. Viewing the software project as an open system that is continuously interacting with its environment also points to the importance of feedback and learning.

The complexity and unpredictability of software problems are typical of "wicked" problems (Rittel and Webber 1973; Yeh 1991), which are difficult to define until they are nearly solved. For such problems, requirements cannot be completely specified until most of the system is built and used. At the same time, the system cannot be built without specifying what is to be built. Furthermore, the problem is never really solved as improvements can always be made.

To deal with and manage software problems, therefore, the activities of requirements, design, coding, and testing have to be performed in an iterative and incremental way, which focuses on ongoing improvement of output value rather than on single delivery. The project should allow overlapping and parallel activities in a series of steps, making feedback and continual learning an internalized habit to reach a desirable result.

Together, these principles lay the foundation for successfully planning, executing, and monitoring the activities of a software project while allowing openness to define the details in each individual case according to the project's specific context.

11.7 Conclusions

The principles of agile project management have the potential to provide organizations and systems with emergent properties. However, organizations should be cautious in embracing these principles or in integrating them with existing practices. Agile management methods are ideal for projects that exhibit high variability in tasks, in the skills of people, and in the technology being used. They are also appropriate for organizations that are more conducive to innovation than those built around bureaucracy and formalization. Software organizations should, therefore, carefully assess their readiness before treading the path of agility.

The challenge in managing agile software projects is to find the balance between upfront planning and learning. Planning provides discipline and a concrete set of activities and contingencies that can be codified, executed, and monitored. Learning permits adapting to unforeseen or chaotic events. The two require different management styles and project infrastructure. Projects with low levels of complexity and uncertainty allow more planning, whereas projects with high levels of complexity and uncertainty require a greater emphasis on learning. Openness to learning is new to many software companies. But it is obvious from the many spectacular project failures that the time has come to rethink some of the traditions in software project management.

Agile project management has currently caught interest for small and co-located projects. However, in the future, agile project management might also solve some of the important challenges facing large-scale and global projects (see Chap. 12). The issues raised in this chapter are instrumental in making this move from traditional to agile project management; or in the words of Louis Pasteur: "chance favors only the prepared mind."

References

Abramson P, Oza N, Siponen MT (2010) Agile software development methods: a comparative review. In: Dingsøyr T, Dybå T, Moe NB (eds) Agile software development. Current research and future directions. Springer, Berlin, pp 31–59

Argyris C, Schön DA (1996) On organizational learning II: theory method and practise. Addison Wesley, Reading, MA

Augustine S, Payne B, Sencindiver F, Woodcock S (2005) Agile project management: steering from the edges. Commun ACM 48(12):85–89

Aurum A, Wohlin C, Porter A (2006) Aligning software project decisions: a case study. Int J Softw Eng Knowl Eng 16(6):795–818

Bazerman MH, Giuliano T, Appelman A (1984) Escalation of commitment in individual and group decision making. Organ Behav Hum Perform 33:141–152

Besner C, Hobbs B (2008) Project management practice, generic or contextual: a reality check. Proj Manage J 39:16–33

Birk A, Dingsøyr T, Stålhane T (2002) Postmortem: never leave a project without it. IEEE Softw 19(3):43–45, Special issue on knowledge management in software engineering

Bjørnson FO, Dingsøyr T (2008) Knowledge management in software engineering: a systematic review of studied concepts and research methods used. Info Softw Technol 50(11):1055–1168. doi:10.1016/j.infsof.2008.03.006

Cohen SG, Bailey DE (1997) What makes teams work: group effectiveness research from the shop floor to the executive suite. J Manage 23(3):239–290

Conboy K, Coyle S, Wang X, Pikkarainen M (2011) People over process: key challenges in agile development. IEEE Softw 28(4):48–57

Constantine LL (1993) Work organization: paradigms for project management and organization. Commun ACM 36(10):35–43

De Meyer A, Loch CH, Pich MT (2002) Managing project uncertainty: from variation to chaos. MIT Sloan Management Review Winter 2002:60–67

Derby E, Larsen D (2006) Agile retrospectives: making good teams great. The Pragmatic Bookshelf, Raleigh, NC

Dingsøyr T (2005) Postmortem reviews: purpose and approaches in software engineering. Info Softw Technol 47(5):293–303

Dingsøyr T, Lindsjørn Y (2013) Team performance in agile development teams: findings from 18 focus groups. In: Baumeister H, Weber B (eds) Agile processes in software engineering and extreme programming, vol 149. Springer, Berlin, pp 46–60

Dingsøyr T, Dybå T, Moe NB (2010) Agile software development: current research and future directions. Springer, Berlin

Dingsøyr T, Nerur S, Balijepally V, Moe NB (2012) A decade of agile methodologies: towards explaining agile software development. J Syst Softw 85(6):1213–1221. doi:10.1016/j.jss.2012.02.033

Dybå T (2000) Improvisation in small software organizations. IEEE Softw 17(5):82–87

Dybå T (2011) Special section on best papers from XP2010. Info Softw Technol 53(5):507–508

Dybå T, Dingsøyr T (2008) Empirical studies of agile software development: a systematic review. Info Softw Technol 50(9–10):833–859. doi:10.1016/j.inf-sof.2008.01.006

Dybå T, Maiden N, Glass R (2014) The reflective software engineer: reflective practice. IEEE Softw 31(4):32–36

Fægri TE (2010) Adoption of team estimation in a specialist organizational environment. In: Sillitti A, Martin A, Wang X, Whitworth E (eds) 11th international conference on agile software development, Trondheim, Norway, 1–4 June 2010. Springer, pp 28–42

Fægri TE, Dybå T, Dingsøyr T (2010) Introducing knowledge redundancy practice in software development: experiences with job rotation in support work. Info Softw Technol 52(10):1118–1132

Faraj S, Sambamurthy V (2006) Leadership of information systems development projects. IEEE Transact Eng Manage 53(2):238–249

Garel G (2013) A history of project management models: from pre-models to the standard models. Int J Proj Manage 31(5):663–669

Guzzo RA, Dickson MW (1996) Teams in organizations: recent research on performance and effectiveness. Annu Rev Psychol 47:307–338

Hackman JR (1986) The psychology of self-management in organizations. In: Pallack MS, Perloff RO (eds) Psychology and work: productivity, change, and employment. American Psychological Association, Washington, DC

Hewitt B, Walz D (2005) Using shared leadership to foster knowledge sharing in information systems development projects. In: Walz D (ed) Proceedings of the 38th Hawaii international conference on system sciences (HICCS), pp 1–5

Hoare CAR (1984) Programming: sorcery or science? IEEE Softw 1(2):5–16

Hoegl M, Parboteeah P (2006) Autonomy and teamwork in innovative projects. Hum Resour Manage 45(1):67

Humphrey WS (1989) Managing the software process. Addison-Wesley, Reading, MA

Keil M, Mann J, Rai A (2000) Why software projects escalate: An empirical analysis and test of four theoretical models. MIS Q 24(4):631–664

Kerth NL (2001) Project retrospectives: a handbook for team reviews. Dorset House Publishing, New York

Kirkman BL, Rosen B (1999) Beyond self-management: antecedents and consequences of team empowerment. Acad Manage J 42(1):58–74

Kozlowski SWJ, Bell BS (2003) Work groups and teams in organizations In: Borman WC, Ilgen DR, Klimoski RJ (ed) Handbook of psychology (vol 12): industrial and organizational psychology. Wiley-Blackwell, New York, pp 333–375

Kozlowski SWJ, Ilgen DR (2006) Enhancing the effectiveness of work groups and teams. Psychol Sci Public Inter 7:77–124

Kransdorff A (1996) Using the benefits of hindsight - the role of post-project analysis. Learn Organ 3(1):11–15

Lehman MM (1989) Uncertainty in computer applications and its control through the engineering of software. Softw Maint Res Pract 1(1):3–27

Lipshitz R, Klein G, Orasanu J, Salas E (2001) Taking stock of naturalistic decision making. J Behav Decis Mak 14(5):331–352

Lynn GS, Skov RB, Abel KD (1999) Practices that support team learning and their impact on speed to market and new product success. J Prod Innov Manag 16:439–454

Menke MM (1997) Managing R&D for competitive advantage. Res Technol Manage 40(6):40–42

Meso P, Troutt MD, Rudnicka J (2002) A review of naturalistic decision making research with some implications for knowledge management. J Knowl Manage 6(1):63–73

Moe NB, Dingsøyr T, Dybå T (2008) Understanding self-organizing teams in agile software development. In: 19th Australian conference on software engineering, pp 76–85

Moe NB, Dingsøyr T, Dybå T (2009) Overcoming barriers to self-management in software teams. IEEE Softw 26(6):20–26

Moe NB, Dingsøyr T, Dybå T (2010) A teamwork model for understanding an agile team: a case study of a Scrum project. Info Softw Technol 52(5):480–491

Moe NB, Aurum A, Dybå T (2012) Challenges of shared decision-making: a multiple case study of agile software development. Info Softw Technol 54(8):853–865

Molokken-Ostvold K, Haugen NC, Benestad HC (2008) Using planning poker for combining expert estimates in software projects. J Syst Softw 81(12):2106–2117. doi:10.1016/j.jss.2008.03.058

Morgan G (2006) Images of organizations. Sage, Thousand Oaks, CA

Nerur S, Balijepally V (2007) Theoretical reflections on agile development methodologies - the traditional goal of optimization and control is making way for learning and innovation. Commun ACM 50(3):79–83

Nerur S, Mahapatra R, Mangalaraj G (2005) Challenges of migrating to agile methodologies. Commun ACM 48(5):72–78

Outi S (2006) Enabling software process improvement in agile software development teams and organisations. VTT Publications, Espoo

Pearce CL (2004) The future of leadership: combining vertical and shared leadership to transform knowledge work. Acad Manage Exec 18(1):47–57

Pich MT, Loch CH, De Meyer A (2002) On uncertainty, ambiguity, and complexity in project management. Manage Sci 48(8):1008–1023

Raelin JA (2001) Public reflection as the basis of learning. Manage Learn 32(1):11–30

Rittel HWJ, Webber MM (1973) Dilemmas in a general theory of planning. Policy Sci 4:155–169

Schwaber K, Beedle M (2001) Agile software development with Scrum. Prentice Hall, Upper Saddle River

Sharp H, Robinson H (2010) Three 'C's of agile practice: collaboration, co-ordination and communication. In: Dingsøyr T, Dybå T, Moe NB (eds) Agile software development: current research and future directions. Springer, Berlin, p 13

Sharp H, Robinson H, Segal J, Furniss D (2006) The role of story cards and the wall in Xp teams: a distributed cognition perspective. In: Agile. Minneapolis, MN. IEEE Computer Society, pp 65–75

Šmite D, Moe NB, Ågerfalk PJ (2010) Agility across time and space: implementing agile methods in global software projects. Springer, Berlin

Staw B (1976) Knee-deep in the big muddy: a study of escalating commitment to a chosen course of action. Organ Behav Hum Perform 16(1):27–44

Stray VG, Moe NB, Dingsøyr T (2011) Challenges to teamwork: a multiple case study of two agile teams. In: Sillitti A, Hazzan O, Bache E, Albaladejo X (eds) Agile processes in software engineering and extreme programming, vol 77. Lecture Notes in Business Information Processing, pp 146–161

Stray VG, Moe NB, Dybå T (2012) Escalation of commitment: a longitudinal case study of daily meetings. In: Wohlin C (ed) Agile processes in software engineering and extreme programming. Lecture Notes in Business Information Processing. Springer, Berlin, pp 153–167. doi:10.1007/978-3-642-30350-0_11

Takeuchi H, Nonaka I (1986) The new product development game. Harv Bus Rev 64:137–146

Trist E (1981) The evolution of socio-technical systems: a conceptual framework and an action research program. Occasional paper no 2. Ontario quality of working life centre, Toronto, ON

Trist E, Bamforth KW (1951) Some social and psychological consequences of the longwall method of coal—getting. Hum Relat 4(1):3–38. doi:10.1177/001872675100400101

von Krogh G, Ichijo K, Nonaka I (2000) Enabling knowledge creation. Oxford University Press, New York

Whyte G (1993) Escalating commitment in individual and group decision making: a prospect theory approach. Organ Behav Hum Decis Process 54(3):430–455

Woodfield SN (1979) An experiment on unit increase in problem complexity. IEEE Trans Softw Eng 5(2):76–79

Yeh RT (1991) System development as a wicked problem. Int J Softw Eng Knowl Eng 1(2):117–130

Biography Tore Dybå received his MSc in Electrical Engineering and Computer Science from the Norwegian Institute of Technology and the Dr. Ing. in Computer and Information Science from the Norwegian University of Science and Technology. He is a chief scientist at SINTEF ICT and an adjunct professor at the University of Oslo. He has 8 years of industry experience from Norway and Saudi Arabia. His research interests include evidence-based software engineering, software process improvement, and agile software development. He is the author or coauthor of more than 100 refereed publications appearing in international journals, books, and conference proceedings. He is editor of the *Voice of Evidence* column in *IEEE Software and a member of the editorial boards of* Journal of Software Engineering Research and Development and Information and Software Technology.

Torgeir Dingsøyr works with software process improvement and knowledge management projects as a senior scientist at SINTEF Information and Communication Technology. In particular, he has focused on agile software development through a number of case studies, coauthored the systematic review of empirical studies, coedited the book *Agile Software Development: Current Research and Future*

Directions, and coedited the special issue on Agile Methods for the Journal of Systems and Software. He wrote his doctoral thesis on *Knowledge Management in Medium-Sized Software Consulting Companies at the Department of Computer and Information Science, Norwegian University of Science and Technology, where he is now adjunct associate professor.*

Nils Brede Moe works with software process improvement, agile software development and global software development as a senior scientist at SINTEF Information and Communication Technology. His research interests are related to organizational, socio-technical, and global/distributed aspects. His main publications include several longitudinal studies on self-management, decision-making and teamwork. He wrote his thesis for the degree of Doctor Philosophiae on *From Improving Processes to Improving Practice —Software Process Improvement in Transition from Plan-driven to Change-driven Development. Nils Brede Moe is also holding an adjunct position at Blekinge Institute of Technology.*

Chapter 12
Distributed Project Management

Ten Misconceptions That Might Kill Your Distributed Project

Darja Šmite

Abstract This chapter is dedicated to companies engaged in collaborative software projects with staff distributed across several locations. The chapter is organized around ten problem areas. Each problem area starts with a common misconception, followed by a discussion of complexities associated with distributed development as opposed to co-located development, practices known for addressing these complexities, and a short list of implications for practice. The aim is to illuminate the key complexities of managing distributed development projects. While project managers in co-located projects are equipped with tools, practices, and methods, these are often of little help when dealing with the challenges of distributed environment. Hence, inexperienced managers often fail to foresee and proactively address the common problems. The readers will learn to distinguish different types of distributed projects (including onshoring, offshoring, outsourcing, and insourcing, to name a few) and challenges, both context dependent and common for distributed projects.

12.1 Introduction

Tough competition and a lack of resources motivate many companies to look for allies that can help in delivering products more cost-effectively. Software companies are no exception. While employing a large number of experienced people with a variety of potentially necessary skills may be too costly, companies are solving immediate gaps in personnel and expertise by contracting work through various sourcing strategies.

D. Šmite (✉)
Blekinge Institute of Technology, Karlskrona, Sweden
e-mail: darja.smite@bth.se

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_12, © Springer-Verlag Berlin Heidelberg 2014

Inspired by industrial migrations and globalization of trade in services (Carmel and Tjia 2005), a large number of software companies are being to collaborate with allies in India, China, and Eastern Europe. The initial enthusiasm of the companies going global, however, soon declines as they start experiencing unique problems of working across geographic, temporal, and cultural boundaries. Global software development is thus called the crisis of the decade, and although the problems of schedule slips, cancellations, poor software, etc. are not new, in distributed projects they are exacerbated by distance (Parnas 2006).

Due to a high demand of specialized skills, sourcing of knowledge-intensive work in software projects is not as straightforward as in those disciplines that, which are more dependent on physical capital. Managerial complexity stems from the challenges associated with establishing effective communication, coordination, and control over the staff distributed across different locations. And in various sourcing strategies and project arrangements, these challenges vary.

12.1.1 Sourcing Strategies

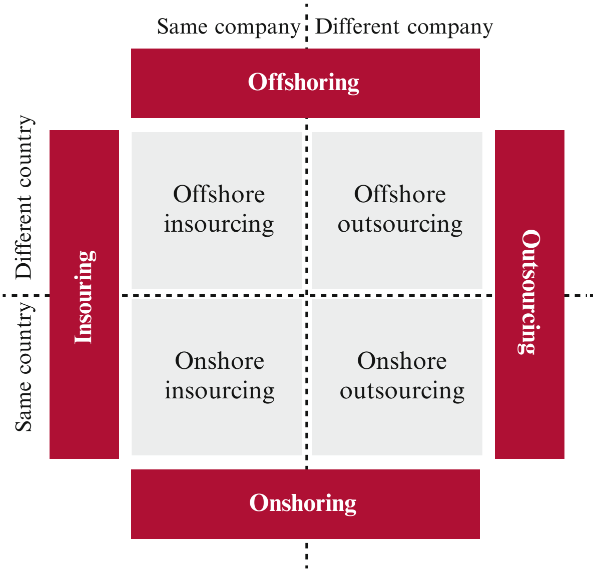
Different organizational relationships and geographical locations of collaborating partners form different sourcing strategies. Some companies contract software development activities to third-party vendors (further referred to as *outsourcing*), while others collaborate with their own subsidiaries or sites in less expensive regions (further referred to as *insourcing*). Both outsourcing and insourcing collaborations nowadays are often targeting emerging nations (further referred to as *offshoring*). Notably, as a result of rapid development and subsequent cost increase in the mature offshore destinations, offshore vendors are often moving development further to less expensive regions within the same country (the practice of contracting within the same country is further referred to as *onshoring*). The different sourcing arrangements can see in Fig. 12.1.

12.1.2 Project Arrangements

In order to better understand the difficulties associated with distributed software projects, it is essential to understand what distributed development means. In this chapter, distributed development is differentiated from global software development in general or the concept of sourcing in its wide sense. Not all sourcing projects are distributed, and even global projects might be entirely sourced for development in one location with no distribution of software engineering activities. Figure 12.2 illustrates eight examples of different project arrangements some of which are distributed and others are not.

The first arrangement (Fig. 12.2a) illustrates traditional co-located development, in which the team works under one roof. The second arrangement (Fig. 12.2b)

Fig. 12.1 Sourcing strategies



outlines an outsourcing example, in which consultants are brought to work in the same premises also known as on-site body-shopping practice. Such arrangement is a sourcing practice without distributed development. In the third arrangement (Fig. 12.2c), a company "buys" software development from the outside, while in the fourth arrangement (Fig. 12.2d) a company builds software within the company boundaries, but in a different location. The latter two practices are also known as project-shopping, in which development is not distributed.

The fifth and the sixth arrangements (Fig. 12.2e, f) outline projects, in which software development is given to two separate teams with co-located team members. Each co-located team in such setups usually requires a team lead or an on-site manager. In the e case, both teams belong to the same company (insourcing), but are situated in different locations. In the f case, the company expands development by employing an additional team's effort from a third-party vendor (outsourcing).

Similarly, the seventh and the eighth arrangements (Fig. 12.2g, h) are distributed projects. In contrast to e and f, team boundaries in these cases span two locations. Therefore, these teams are called virtual or dispersed, while the projects are called distributed. In the g case, the team comprises team members from two sites of the same company (insourcing), while, in the h case, the team comprises team members from two different companies (outsourcing). Traditionally, third-party vendors support their teams with an internal project manager, while in, case of insourcing (Fig. 12.2g) the company may decide whether this is needed or not.

Note that the given examples are not exhaustive. In practice, projects may have more than two distributed or even dispersed teams. Accordingly, the number of team leads or managers as well as the combination of outsourcing and insourcing relationships may vary. The number of team members in the distributed and

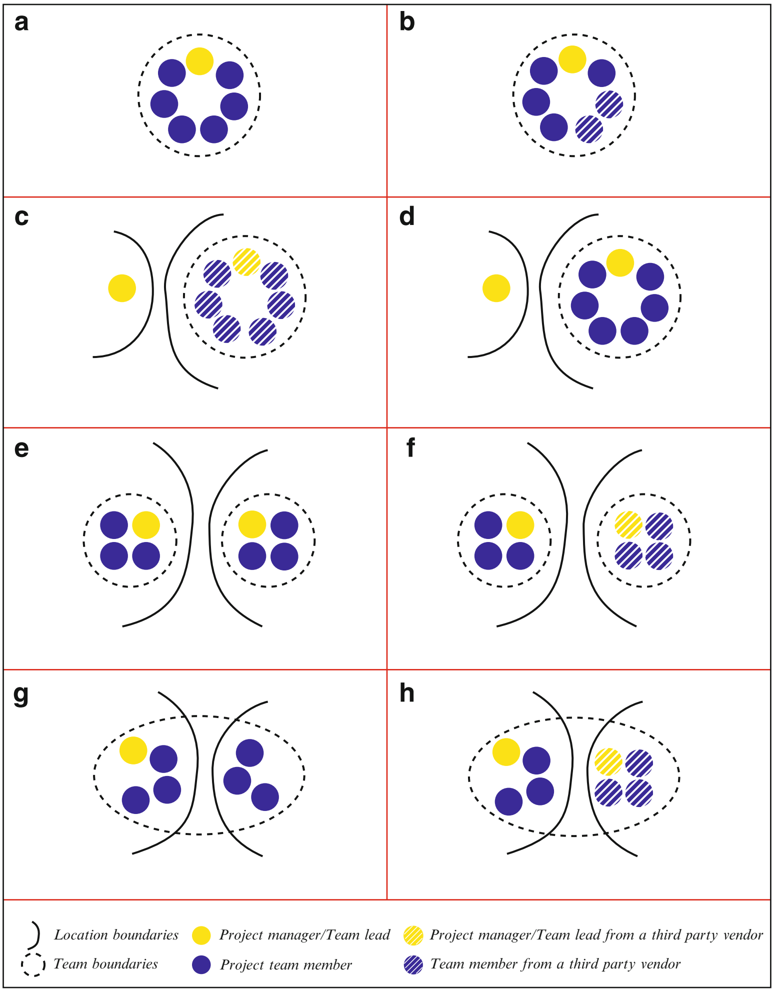

a
b
c
d
e
f
g
h
Location boundaries
Team boundaries
Project manager/Team lead
Project team member
Project manager/Team lead from a third party vendor
Team member from a third party vendor

Fig. 12.2 Nondistributed (**a–d**) and distributed (**e–h**) project arrangements. (**a**) Traditional co-located team, (**b**) co-located team with onsite consultants, (**c**) non-distributed outsourcing project, (**d**) non-distributed insourcing project, (**e**) distributed insourcing project with two distributed teams, (**f**) distributed outsourcing project with two distributed teams, (**g**) distributed insourcing project with one dispersed team, (**h**) Distributed outsourcing project with one dispersed team

dispersed teams outlined in the figure is illustrative, while in practice some distributed projects may be significantly larger than others. The managerial hierarchy may also vary.

Particular project context factors depending on the project type, number of teams, and their structures may affect the magnitude of managerial challenges. However, common challenges relevant for all (or most) distributed projects exist.

Despite the differences, all distributed projects are challenged by distance, whether employing distributed or dispersed teams, based on insourcing or outsourcing. This is associated with the loss of next-door closeness. Research shows that patterns of interaction between the team members change as the distance between their work places grows, where the breaking point is as close as 30 m (Allen 1977). While this may seem surprising, the links between distance and collaboration frequency has been known for decades. Distance on the global scale naturally has many more implications, not only on the team members, but also on the practice of project management.

The rest of the chapter is devoted to common problem areas regarding distributed project management (see Sect. 12.2). Each problem area starts with a common misconception, followed by a discussion of complexities associated with distributed development. Related industrial experiences based on the author's empirical research work and personal experiences are shared through practical illustrations. Short recommendations for practice conclude each problem area. Ten general observations conclude the chapter (see Sect 12.3).

12.2 Ten Misconceptions in Distributed Software Development

This section contains descriptions of ten misconceptions that prevail in particular among project managers with no prior experience in distributed software projects. The selection of misconceptions is based on the author's observations from conducting empirical research in a number of software companies around the world. These include small, medium, and large software companies from Sweden and Norway working with service providers and own subsidiaries from Asia and Eastern Europe, as well as several Latvian offshore companies that work for the customers around Europe and North America.

12.2.1 Experienced Project Managers Will Deal with Any Complexities: Managing Distributed Development Projects Cannot Be That Hard

The reason of failure in global and distributed projects is often not related to the abilities of project managers to manage a project or lead people. Projects fail mainly due to the lack of awareness of unique factors that make distributed projects different from co-located ones (DeLone et al. 2005; Carmel and Tjia 2005). Managers do not always know what to anticipate. Geographic, temporal, and cultural distances make even experienced managers handicapped since practices that are commonly used in co-located projects are often of little help. Distributed projects require a rich and reliable technical and communication infrastructure that can support distributed teamwork. Besides, the very basic management routines change, when managers loose the next-door closeness.

> **Practical illustration**: *A Latvian project manager with a long experience in software development was assigned responsibility for a distributed onshore insourcing project. The project employed a dispersed team with team members from two locations of the same company separated by 200 km. The manager was very traditional in his beliefs in standardizing work practices and trusting only after verifying the results. Managing the dispersed team made him uncomfortable. He lacked visibility into the process and was unable to closely supervise his employees. At one point he even thought of installing video cameras in the remote location. Remote team members felt manager's distrust and thus feared of being fired. When facing difficulties with poor IT infrastructure (slow computers and Internet connection), they decided not to complain, because they were afraid to cause any more inconveniences. The manager, however, associated these problems with poor performance, lazy work and lack of capabilities of the remote team members. Misunderstandings would have destined this collaboration to termination if only an external consultant experienced in distributed development would not have been brought into the project. After visiting the remote location the consultant revealed the true reasons behind performance problems and made a step towards initiating open communication between the dispersed team members and the project manager.*
>
> *Based on an empirical study published in* (Moe and Šmite 2008).

The style of supervision and daily managerial routines are not the only practices that change in the light of distribution. A number of specific impact factors for project management are attributed to global and distributed projects. For example, offshored projects often suffer from high turnover of the employees, which immediately affects the project performance and leads to additional costs of recruiting and training new employees. Work over distance causes significant overhead of communication and coordination since problem resolution often requires more people to be involved and computer-mediated communication is much slower than face-to-face (Matloff 2005). Task allocation also requires concern about the interdependencies between the sites to minimize the collaborative overhead; however, the nature of software development often means that many tasks are, in fact, interrelated.

All the mentioned factors and many more have important implications for various aspects of project management such as project planning, effort estimation, task allocation, and project monitoring. Experience shows that managers with no or limited prior experience in managing distributed projects often overlook or underestimate the necessity and importance of preparation early in the project (Carmel and Tjia 2005).

Implication for Practice Both planning and execution of a distributed project requires knowledge about unique factors inherited in this type of projects and a deep understanding of their impact. It is essential to adjust the methods for project estimation and monitoring to account for the specifics of distributed projects. Estimating the value of impact factors and associated overheads is not easy as they vary in different setups. Therefore, it is important to document and use historical data from within the company.

12.2.2 Experience from a Single Distributed Development Is Invaluable and Turns a Project Manager into an Expert

Experience is indeed invaluable. However, distributed projects are diverse and each of them has a different flavor stemming from the context of each individual sourcing strategy and project arrangement.

In particular, not all distributed projects experience all three distances (geographic, temporal, and cultural). Projects can be distributed nationally or internationally, from north to south or from east to west, or even have a mixture of different distances in case of involvement of more than two locations. Figure 12.3 illustrates the different arrangements of onshore and offshore projects [adapted from Šmite et al. (2014)]. For convenience, distances are expressed in concrete measures, which allow project managers to evaluate at least some of the implications of arranging projects in different locations. For example, a large temporal distance (more than 4 h) means that dispersed team members will have a small synchronous collaboration window, while geographic distance is associated with the travel time. If the flying time is more than 2 h, it means that a meeting of 2–3 h will likely require staying overnight.

The diversity of project arrangements also means that each of them shapes a different project environment. Experience from one arrangement does not necessarily equip project managers for others. This is because experience shows that techniques that work well in one context could cause failure in another (Heeks et al. 2001).

Practical illustration: *A Swedish project manager, who coordinated work in a distributed project employing several distributed teams from Swedish and Indian sites of the same company, has already had an extensive experience from working in software development projects both with Chinese developers and onsite in China. He was used to providing detailed descriptions of the tasks to avoid misunderstandings or misinterpretations, and assigning clear responsibilities. However, he failed to realize that the only authority eligible to assign responsibilities in the Indian team was their local manager. While Indian developers never refused the tasks from the Swedish project manager, they did not feel committed to fulfilling these tasks, unless the local manager approved and prioritized them.*

Based on a conversation with two project managers from a multinational software development company

Implication for Practice Project managers shall approach each distributed project with caution and search for experiences and practices that have proven their efficiency in arrangements as similar as possible to those of the current interest.

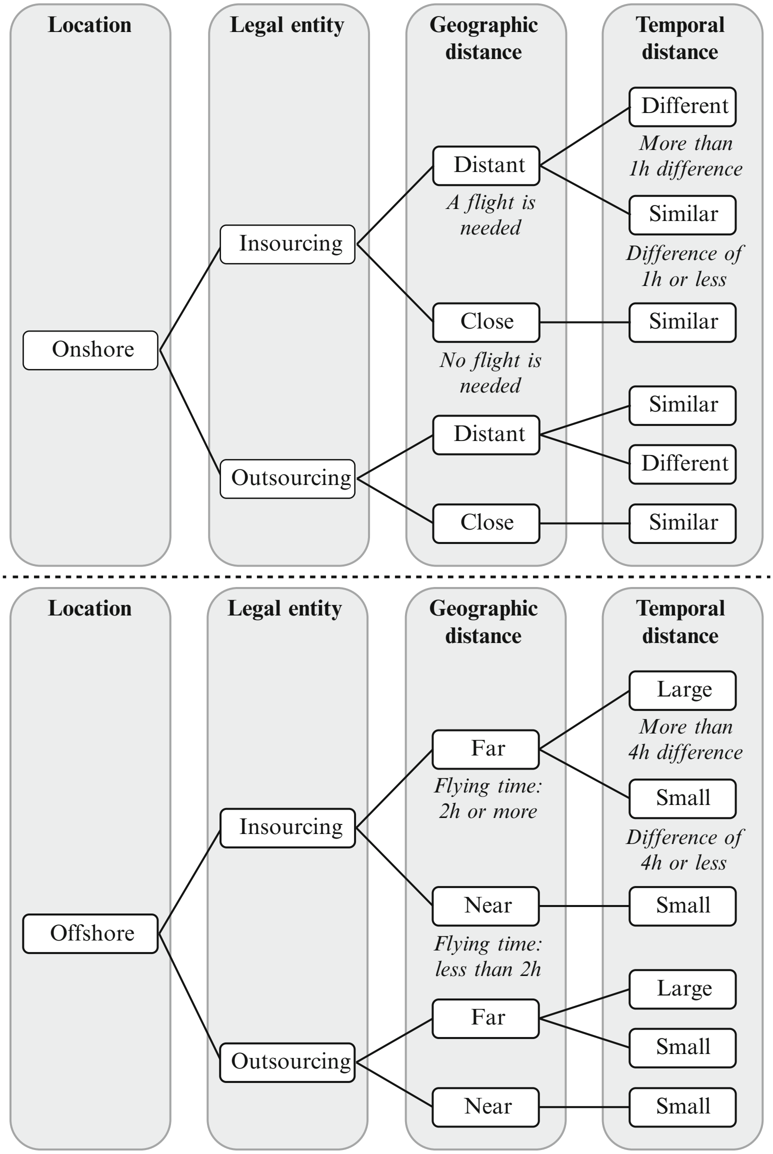


Fig. 12.3 Various project arrangements [adapted from Smite et al. (2014)]

12.2.3 Once You Are Distributed, It Does Not Matter How Many Remote Sites Are Involved

One may believe that management of a distributed development project requires no more than establishing a proper technological infrastructure that supports remote development and computer-mediated communication. However, in reality this is a misconception. Complexity of working together grows exponentially when adding additional sites (Bhat et al. 2006; van Solingen and Valkema 2010). If working across two locations is difficult, then adding the third or the forth site may increase the challenges dramatically. First of all, each site ideally requires a local team lead or a manager to supervise the development process. Secondly, coordination of daily activities and dependencies grows in terms of complexity and required effort, while the overall productivity goes down. This is mainly because decision-making and problem-solving often requires involvement of people from all locations, and also because it requires more people than if there was only one location.

A controlled study that tested different settings demonstrated that a high-quality two-site workflow may perform similarly to a four-site workflow, since the individual working speed decreases when more sites are added (van Solingen and Valkema 2010). Industrial experiences also show that work division to more than two sites is disadvantageous (Lings et al. 2007) and might make temporal distances difficult to manage (Holmström et al. 2006). Perhaps even more important is the finding from a multinational software development organization, in which an increase in the number of sites as well as geographic and temporal dispersion of those sites was associated with a negative effect on software quality (Cataldo and Nambiar 2009).

Practical illustration: *A customer from North America contracted software development to an outsourcing service provider represented by business analysts from North America and a team of developers from Latvia. The project was challenged by a number of factors – new domain, new technology, and new development methodology – many of which were underestimated. When the software was delivered the customer was not satisfied and required additional changes. The Latvian project manager was in need of adding more developers to finalize the already late project as soon as possible. Unfortunately, the company was short of staff locally and it was decided to contract work to three different sites of the same company situated in Eastern Europe. Initial optimism faded away when poorly controlled dependencies resulted in late deliveries. In fact, some of the remotely developed components needed rework that caused additional effort for the onsite team. Although the customer received what was requested and used the software at the end, the project duration exceeded the plan 4 times and the total effort exceeded the budget by 77%. It turned out to be an expensive lesson for the service provider not to distribute development late in the project.*

Based on an empirical study published in (Šmite and Gencel 2009).

Implication for Practice If you distribute development, aim for involving no more than two locations. If, however, involving more than two locations, managers shall account for nonlinear growth of complexity when adding new locations.

12.2.4 Any Problem Can Be Fixed with the Right Toolset

The basic lesson learned in distributed projects indicates that remote teamwork requires a special infrastructure. Furthermore, many offshoring enthusiasts would claim that companies overcome any geographical obstacles with the right toolset (Matloff 2005). However, technology is far from being sufficient in addressing the challenges of distributed work and even the latest advances of telecommunications tools would not help in replacing face-to-face collaboration (Matloff 2005). The truth is that most of the problems in distributed development are human and not technical. While tools are important to alleviate distributed projects, solutions for most of the problems, however, relate to managing human interaction rather than just providing the necessary equipment.

Distributed projects significantly depend on the soft skills of the team members. Language skills, cultural awareness, and good communication skills are only a few examples of what a distributed team member shall possess. These skills are especially important for project managers who are often dealing with complexities beyond administrative functions, such as unwillingness to cooperate, lack of trust among the team members, alienation due to the lack of face-to-face interaction or primarily asynchronous communication, cultural misfit, etc. (Piri et al. 2012; Moe and Šmite 2008). Many of these issues have been also related to poor motivation and thus performance (see Chap. 10). Project managers shall also understand that many of the human factors are interrelated—some of the problems create significant obstacles for overcoming other problems, and thus the exposure of the negative consequences grows.

> **Practical illustration**: *A distributed project, in which a German company offshored software development to Latvia, suffered from deficiencies in addressing the soft factors. Although the technological infrastructure was well thought through, and the remote team members used video-conferencing and instant messaging extensively, that appeared insufficient to establish the trusting relationship. The project experienced a chain of problems: a lack of language skills, too little communication, a loss of quality in the information exchanged, and misunderstandings. Diversity in the ways of working and socio-cultural differences between the collaborating partners only worsened the situation. The German team members did not fully trust the capabilities of the Latvian team, while the Latvian team did not trust the good intentions of the German managers. This generated a negative feedback loop. The Latvian team started to doubt negative feedback from the remote managers. This initiated extensive monitoring from the German partner, which affected the level of trust even more. There was a decrease in information exchange and feedback, which resulted in team members being not updated about the progress and plans. This atmosphere influenced the offshore team's morale and productivity started to decrease, driving the relationship to unavoidable conflicts.*
>
> *Based on an empirical study published in* (Moe and Šmite 2008).

Implication for Practice Project managers shall have an explicit early concern with identifying and addressing the soft factors in the project to avoid or at least substantially reduce their negative consequences. This includes recruiting employees with experience from international projects, organizing cultural communication courses, clarifying the rational for engagement in the distributed development, providing opportunities for rich communication and face-to-face interaction, if possible, as well as fostering frequent feedback, knowledge, and information exchange across the sites.

12.2.5 The Follow-the-Sun Approach Significantly Speeds Up Development

One of potential drivers of distributed development is related to the willingness to speed up development by leveraging time-zone effectiveness, also known as around-the-clock or follow-the-sun development approach. Companies may split the work across three locations to fit 3 working days in 24 h. This can be done in different ways. A project may choose to distribute sequential tasks, in which the work is forwarded from one location to the next at the end of the day. Alternatively, a project may also organize different tasks in a sequence, such as development and testing, or test case creation and test execution. Although in theory such development approach promises three times faster delivery (considering there are three sites involved), in practice, distributed development is less efficient than co-located due to the communication and coordination overhead and significant delays due to asynchronous working hours (Bhat et al. 2006; van Solingen and Valkema 2010). It is not uncommon that one site is repeatedly undoing the work of the other (Matloff 2005). This means that the gains are not as large as expected.

Interestingly, a simulation of around-the-clock development effort distribution for a different number of sites indicates that utilizing more than three sites is seen as unlikely to reduce development time at all (Taweel and Brereton 2006). Similarly, empirical findings from Intel, HP, and Fidelity suggest that the companies are not benefitting from the follow-the-sun model, striving for more synchronous work instead (Conchúir et al. 2009).

While attempts to speed up development through sequential handovers have failed, testing and defect resolution cycles have demonstrated their effectiveness (Conchúir et al. 2009; Matloff 2005).

Practical illustration: *A large truly global project employed development teams from Belarus, China, India, and Latvia with US-based project management. One of the goals of the project was to follow-the-sun and continuously develop small software components. Task specifications were developed by the project management site and then passed on to the offshore sites to be completed within a day. This, however, soon proved to be a challenging mission due to cross-site coordination problems and emerging management bottleneck. Sequential work caused frictions and conflicts. Very often in the mornings Latvian developers would look into the code and complain about their peers in India, who have re-written their code pieces, and then start re-writing the same code according to their understanding. Based on the lessons learned, the number of handovers was reduced, and more responsibilities were shared.*

Based on personal experience and experiences shared in (Carmel 1999).

Implication for Practice The follow-the-sun approach is difficult to put into practice. If applied to development tasks, this approach carries significant overhead and will not likely save the time as expected. However, it might be applied to dependent tasks, such as defect resolution and support. This will require maintenance of a central repository, automation of knowledge transfer between the sites and significant support for asynchronous communication, as well as careful progress monitoring.

12.2.6 Splitting the Work into Independent Chunks Helps to Avoid Collaboration Problems and Improves the Output

While many managers come to realize that neither follow-the-sun development nor sharing the same code base is easy, they strive to minimize the dependencies by splitting the work into independent chunks. Modularization seems like the silver bullet for distributed development, in theory, as it promises to minimize the overhead of communication and coordination, and therefore removes the necessity to deal with collaboration problems, cultural misunderstandings, and alike. In practice, however, it works only for well-defined tasks with clear interfaces. Changing requirements and complex dependencies between the modules risk to drive the project into a big bang at the end of the project, when modules are to be integrated. Integration problems emerge because software components are never truly independent and even the best designs are never error-free, while changes are never completely predictable (Herbsleb and Grinter 1999). These challenges can be addressed by implementing continuous integration strategies and incremental deliveries.

Contradictory, although too much communication and coordination is a burden, too little may be equally bad. This is because interaction is necessary to hold the remote sites and their outcomes together. Thus, managers shall be aware that strict modularization between the sites may trigger unwanted outcomes, such as alienation, ignorant implementation decisions, misplaced functionality, and redundant work.

Practical illustration: *A large and complex software development project distributed between Swedish and Chinese locations of the same company went through an evolution of ways of working by probing and adjusting different approaches. The necessity to collaborate resulted from a decision to merge two mature products, which were previously developed separately in two different locations. This decision was driven by the market demands and resulted in very coupled work on a new, shared platform across the two development locations. However, working on the same code base required significant amount of communication and coordination, which slowed down the development and made work inefficient. After struggling with developing and maintaining the shared components for a year, ways to optimize the interfaces were sought. It was therefore decided to implement a more decoupled way of working and strive for strict modularization in order to improve quality and isolate the impact of faults. The new approach improved the situation,* *but introduced new challenges. While coordination needs decreased dramatically, the importance of early decisions about allocation of common requirements increased. Because of modularization common functionality started to cause difficulties due to split responsibilities, differences in time pressure and delivery commitments. Architects believed that some functionality were misplaced or occasionally missed due to the wrong assumptions caused by the enforced modularization.*

Based on an empirical study, parts of which are published in (Šmite et al. 2013).

Implication for Practice Since modularization and joint work both have their pros and cons, managers have to make trade-off decisions. When modularization is considered, progress coordination and frequent knowledge exchange mechanisms shall be implemented to avoid later integration problems, ignorant implementation decisions, misplaced functionality, and redundant work.

12.2.7 Distributed Projects Cannot Be Agile

With the growing popularity of agile development, managers started practicing hands-on interactive methods that focus on frequent meetings and feedback. Documentation in these projects is available primarily in the office space, such as Kanban boards and burn-down charts. Success of agile development and, especially, the implementation of self-managing teams heavily rely on cohesiveness of the team members and their abilities to collaborate, share the knowledge, and solve problems effectively (see Chap. 11). This may be challenging for distributed and, in particular, dispersed teams. Therefore, agile development and distributed development are often viewed as two incompatible approaches. However, recent empirical studies show that some elements of agile development are not only possible to implement, but also appear to be useful for distributed and dispersed teams (Holmström et al. 2006; Simons 2006).

Of course, it is hard to argue that distributed projects can be truly agile, in the sense of strict adherence to particular agile methods, especially those demanding co-location and close interaction. However, agility as a philosophy and spirit is not necessarily contradicting with the distributed development environment. A collection of experiences with implementing agile methods in distributed development gathered in Šmite et al. (2010) suggest finding ways to develop a sense of teameness across locations, providing appropriate tools and technology that can facilitate this, enabling face-to-face collaboration when needed, valuing cultural differences instead of seeing them as a threat, enabling teams to deliver complete functionality independent of localization, and acquiring skilled human resources at each location. As agile "by the book" cannot work in distribute development, project managers shall be prepared to tailor their agile processes to suit the distributed environment. For example, empirical experiences suggest that pure agile development with little or no documentation beyond the code is impractical and inefficient (Simons 2006).

One important enemy worth emphasizing in those distributed projects that target offshore locations is a high turnover of the employees within the project, which can be evident only in the later stages of collaboration. Notably, managers often underestimate attrition rates, when sourcing to countries with extensive promotion-seeking behavior. While it is not a factor specific only to distributed projects, the role of attrition can be devastating in distributed agile projects, which heavily depend on high levels of cohesion, familiarity, and shared mental model of the team members. To achieve cohesiveness, agile teams shall be stable. Instability of the team members may prevent achieving the assumed benefits of agile software development.

Implication for Practice Managers who consider implementing agile methods for distributed development shall ensure opportunities for close synchronous interaction and availability of appropriate technology to enable close collaboration across the distance. Many methods and practices will require adjustments, which managers shall prepare to implement. The role of employee turnover in distributed agile projects shall not be underestimated. Managers shall foresee and minimize the risk of instability when forming agile teams.

12.2.8 Standardizing Work Processes Will Help to Control the Diversity

Managers in distributed development often rely on formalization and standardization by selecting quality certified partners, developing detailed architectural designs and plans, and establishing work common procedures and ways of working, as well as assigning the tasks to local and remote team members. This is an intuitive answer to a challenging situation, which distributed development surely is. Although many offshoring destinations have made adherence to the highest standards of the Capability Maturity Model their marketing cornerstone, it is important to remember that it is simply a project management tool and not a measure of software quality (Matloff 2005). Furthermore, in contradiction to a common view that methodological standardization helps (Carmel and Tjia 2005), in practice, management by program in distributed development often does not work as intended (Šmite et al. 2008; Matloff 2005). The reasons include disparities in work practices, limited access to remote locations, reluctance to follow enforced procedures, and clashes with the cultural and organizational habits. In fact, an industrial experience demonstrated that the benefits of process maturity diminish as the development work becomes more distributed (Cataldo and Nambiar 2009). In addition, lack of visibility into remote locations may mean that the failing course of standardization might even not be easy to reveal.

The fact that standardization may fail does not mean, however, that planning is useless. It means that project managers shall approach formalization of the ways of working carefully, considering the implications of diversity across locations. Distributed development can undoubtedly benefit to some extent from stable plans, processes, and specifications; however, spontaneous communication invoked by the developers is essential (Herbsleb and Grinter 1999). It is also essential to acknowledge the differences and not break the culture (work related or national). Managers will succeed better by integrating the differences and capitalizing on the strengths of diversity rather than forcing a single-site perspective. The changes also require more than ensuring common process descriptions and project websites; people across all location, shall be aware and committed to following the emerging ways of working.

Practical illustration: *The Latvian project manager from the on-shore in-sourcing project, who supervised a dispersed team of software developers separated by 200 km, heavily relied on standardization. The company followed a set of common processes and procedures from the ISO 9001:2000 certified quality management system. In addition, the project manager established guidelines (project quality assurance plans, communication plans, and different software development related process handling guidelines) and expected everybody to follow them. Noteworthy, the project manager also assigned the daily tasks to developers. During the course of the project, however, disparities in work practices between the central and remote locations emerged. The remote team members did not work according to the established processes. They worked cohesively as a self-managing team, shifted their tasks and acted "as a joint body". Since frequent feedback was not encouraged, they also interpreted requirements without consulting the systems analysts who worked remotely. This often resulted in making decisions without sufficient information, and led to subsequent rework. While the remote team members perceived to act in the best interest of the project, their cohesiveness and independence was not appreciated, as it ruined the standardization program set by the project manager.*

Based on an empirical study published in (Šmite et al. 2008).

Implication for Practice If common ways of working are expected, these shall originate from a dialog, between all parties involved. Alternatively, the project manager may rely on diverse approaches in each location following the principle "whatever works". Rotation of personnel across locations will help to create an understanding and awareness of the differences, as well as the necessary solutions.

12.2.9 Distributing Development to Offshore Sites Saves Costs

The vast majority of software companies start distributed development assuming that it will help to reduce development costs (often by contracting work offshore). This is usually motivated by the significant salary differences in different locations. However, attempts to understand the reliability of economic expectations lead to a conclusion that opinions about the realization of cost savings vary (Šmite and Wohlin 2011). Some studies claim achievement of significant cost savings through offshoring, while others suggest that the cost savings are much lower than expected or even hard to achieve. What companies rarely realize is that the salary differences and thus potential cost savings are often achieved by staffing projects with young, inexperienced programmers (Matloff 2005). In fact, an analysis of different staff distributions revealed that the strategies with the best potential for profitability were among the worst for both quality and productivity (Ramasubbu et al. 2011). Accordingly, it is not surprising that the achievement of cost savings is often attributed to very basic projects, such as software testing (Poikolainen and Paananen 2007), while complex software development projects are often associated with the losses [e.g., Poikolainen and Paananen (2007)].

As noted earlier, distributed development is considerably more complex than co-located projects, and making it work requires substantial additional investments. The economic benefits might thus be offset by these additional costs stemming from managerial overhead and high turnover, decreased productivity, and integration challenges. Many of the initial extra offshore costs decrease over time (Carmel and Tjia 2005). However, in case of receiving poor quality software delivered by the vendors, the calculation of cost savings becomes obsolete. These challenges suggest that distributed development shall not be done for cost-reduction reasons alone.

> **Practical illustration**: *Three small and medium sized Scandinavian companies decided to achieve cost savings through outsourcing to software vendors in Asian countries, where salary differences are significant. However, due to a chain of negative events and expensive failures all collaborations were ultimately cancelled. All Scandinavian companies started outsourcing with no prior experience with offshoring. All companies started small by outsourcing minor although complex work tasks to large software vendors. Some projects employed dispersed teams, others worked with distributed teams, but all experienced problems with the quality of the delivered software and inefficiency of the distributed work. While one of the companies terminated the contract after the first year of unsuccessful collaboration, two other companies tried to implement different improvements, including onsite training sessions and exchange visits. However, due to a high turnover of the offshore employees, the necessary level of experience and expertise were never achieved, and collaborations were terminated. Interestingly, all three companies later attempted offshoring collaborations again, but with a different strategy.*
>
> *Based on an empirical study published in* (Moe et al. 2012).

It is noteworthy that there are other than cost-related reasons for distributed development. These include search for unique experience and expertise, which is not available in-house, entrusting product customizations to developers that know the market best, and bidding for larger projects or scaling up development with the help of others when it is impossible solely in-house. These targeted benefits can compensate the inconvenience of working on a distance, as well as the additional costs necessary to making distributed development work.

Implication for Practice Managers shall not expect cost reductions that are implied by the salary differences and shall account for various extra costs of distributing software development and investments necessary to make it work.

12.2.10 Distributed Development Shall Be Avoided, as All Distributed Projects Are Complex, Inefficient, and Unsuccessful

While distributed projects are often more complex than even the most complex co-located projects, distributed development shall not necessarily lead to failure. However, to make it work, project managers shall be prepared to make the above-mentioned investments into technical infrastructure, rich communication and face-to-face interaction, knowledge and information exchange, and so on.

The research in the area of distributed development cannot undoubtedly point to particular project types that are likely to be successful or unsuccessful. However, several observations exist. For example, prior knowledge, experience, and familiarity of the team members have a strong positive effect on the cooperation (Espinosa et al. 2001). Successful distributed projects and large cost savings are associated with well-defined processes that are performed already before offshoring started, and which require little control (see Chap. 9). Complexity, size, necessity for domain knowledge, ability to define independent tasks, etc. also play an important role in determining the likelihood of success. Large, complex, and interdependent tasks are not advised for distribution (Herbsleb and Grinter 1999).

In summary, distributed development strategies shall take into account what to develop where and how. For example, an innovation project that requires frequent interaction of the software developers might not be suitable for collaboration between far off locations, but could work when being organized nearshore, where time zone overlap is not critical and traveling distance is easy to overcome. At the same time, offshoring of innovation-demanding projects to countries in which educational systems stifle creativity and independent thinking might not be the best option (Matloff 2005). Experience shows that keeping the high-level design onsite and distributing implementation to offshore teams is not a solution either since innovative ideas often come from programmers while they are working (Matloff 2005).

Practical illustration: *All three small and medium sized Scandinavian companies from the previous practical illustration have created successful collaborations after failing in their first attempts with offshore outsourcing. Instead of outsourcing software development to large software vendors in Asia, two of the three companies decided to insource to Russia by focusing on availability of the needed competence, geographic, temporal and cultural proximity. The third company decided to utilize an existing location and insourced to China by focusing on proximity to customers. All the three companies approached their new relationships differently. Managers setting up collaborations proactively invested into the integration of the remote sites into a common corporate culture, hiring people with the right competences, moving experts to the new site for knowledge exchange, and engaging developers at remote locations in challenging tasks with responsibility. All companies succeeded in maintaining control over recruitment, training and commitment, which were the problems experienced in the previously failed collaborations. This had a positive impact on the overall collaboration and even though some challenges still existed, the companies could address them more effectively.*

Based on an empirical study published in (Moe et al. 2012).

Implication for Practice Success in distributed development demands early concern and investments. Project managers shall be aware of particular factors threatening collaborations and unfavorable combinations of location-related and project-related characteristics. Mature distributed collaborations might function better than initial collaborations because accumulated experience of the project manager and the teams are important to overcome the challenges of working distributedly.

12.3 Conclusions

The ten misconceptions in distributed project management discussed in this chapter intend to highlight the main areas of concern and equip project managers with awareness about what to do and what not to do. The topics here cover a variety of common project management responsibilities, while the practices and implications illustrate that traditional approaches may not work as intended. The challenges of working across geographic, temporal, and cultural distances increase the managerial complexity related to communication, coordination, and control. More often than not these challenges cause ripple effects and snowball effects that are not always foreseen by the project managers in the beginning of the project. In this way, for example, confusion caused by delays in communication that arise from temporal distance and asynchronous work might be amplified by cultural peculiarities related to preferable communication patterns and approaches to problem resolution.

Based on the ten misconceptions, the following ten conclusions can be drawn:

1. Management of distributed development projects requires new skills and approaches, often distinct from those applied in traditional co-located development projects.
2. Distributed development projects are not the same; they have different flavors, as the distributed setups are very diverse. Therefore, experiences from a single project or a setup may not be sufficient or even applicable in a different project.
3. The number of sites participating in a distributed project matters greatly as the level of complexity has a nonlinear growth when adding new development locations.
4. Implementing better or more tools cannot fix most of the challenges in distributed development as their nature is related to managing soft factors.
5. Follow-the-sun approach is very risky and often fails to work for development tasks; therefore, increase in development speed is never as large as expected.
6. Modularization of the work decreases communication and coordination overhead however, increases the risks of integration failure, ignorant implementation decisions, misplaced functionality, and redundant work.
7. Distributed projects can never be agile in its true sense however, an agile spirit and philosophy are useful to address many challenges with distributed development.

8. Standardization of work processes rarely works as intended due to diversity and preferred ways of working.
9. Cost savings from distributed development are not as large as the salary differences suggest due to additional overheads and hidden costs.
10. Distributed development can be successful if chosen as a strategic necessity, but managers shall be prepared to invest into making it work.

References

Allen TJ (1977) Managing the flow of technology. MIT Press, Cambridge, MA

Bhat JM, Gupta M, Murthy SN (2006) Overcoming requirements engineering challenges: lessons from offshore outsourcing. IEEE Softw 23(5):38–44

Carmel E (1999) Global software teams: collaborating across borders and time zones. Prentice Hall PTR, Upper Saddle River, NJ

Carmel E, Tjia P (2005) Offshoring information technology: sourcing and outsourcing to a global workforce. Cambridge University Press, Cambridge

Cataldo M, Nambiar S (2009) On the relationship between process maturity and geographic distribution: an empirical analysis of their impact on software quality. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC/FSE), pp 101–110

Conchúir EÓ, Ågerfalk PJ, Holmström H, Fitzgerald B (2009) Global software development: where are the benefits? Commun ACM 52(8):127–131

DeLone W, Espinosa JA, Lee G, Carmel E (2005) Bridging global boundaries for IS project success. In: Proceedings of the 38th annual Hawaii international conference on systems sciences (HICSS), pp 48b

Espinosa J, Kraut R, Slaughter S, Lerch J, Herbsleb JD, Mockus A (2001) Shared mental models, familiarity and coordination: a multi-method study of distributed software teams. In: Proceedings of the international conference in information systems (ICSE), pp 425–433

Heeks R, Krishna S, Nicholson B, Sahay S (2001) Synching or sinking: global software out-sourcing relationships. IEEE Softw 18(2):54–60

Herbsleb JD, Grinter RE (1999) Splitting the organization and integrating the code: conway's law revisited. In: Proceedings of the 21st international conference on software engineering (ICSE), pp 85–95

Holmström H, Fitzgerald B, Ågerfalk PJ, Conchúir EÓ (2006) Agile practices reduce distance in global software development. Inf Syst Manag 23(3):7–18

Lings B, Lundell B, Ågerfalk PJ, Fitzgerald B (2007) A reference model for successful distribute development of software systems. In: Proceedings of the 2nd international conference on global software engineering (ICGSE), pp 130–139

Matloff N (2005) Offshoring: what can go wrong? IT Prof 7(4):39–45

Moe NB, Šmite D (2008) Understanding a lack of trust in Global Software Teams: a multiple-case study. J Softw Process Improv Pract 13(3):217–231

Moe NB, Šmite D, Hanssen GK (2012) From offshore outsourcing to offshore insourcing: three stories. In: Proceedings of the 7th international conference on Global Software Engineering (ICGSE), pp 1–10

Parnas D (2006) Agile methods and GSD: the wrong solution to an old but real problem. Commun ACM 49(10):26–34

Piri A, Niinim T, Lassenius C (2012) Fear and distrust in global software engineering projects. J Softw Maint Evol Res Pract 24(2):185–205

Poikolainen T, Paananen J (2007) Performance criteria in inter-organizational global software development projects. In: Proceedings of the 2nd international conference on global software engineering (ICGSE), pp 60–70

Ramasubbu N, Cataldo M, Balan RK, Herbsleb JD (2011) Configuring Global Software Teams: a multi-company analysis of project productivity, quality, and profits. In: Proceedings of the 33rd international conference on software engineering (ICSE), pp 261–270

Simons M (2006) Global software development: a hard problem requiring a host of solutions. Commun ACM 49(10):32–33

Šmite D, Gencel C (2009) Why a CMMI Level 5 company fails to meet the deadlines? In: Proceedings of the international conference on product-focused software development and process improvement, pp 87–95

Šmite D, Moe NB, Torkar R (2008) Pitfalls in remote team coordination: lessons learned from a case study. In: Proceedings of product-focused software development and process improvement conference (PROFES), LNCS, pp 345–359

Šmite D, Wohlin C (2011) A whisper of evidence in global software engineering. IEEE Softw 28 (4):15–18

Šmite D, Moe NB, Ågerfalk PJ (2010) Agility across time and space: making agile distributed development a success, 1st edn. Springer, Heidelberg

Šmite D, Wohlin C, Aurum A, Jabangwe R, Numminen E (2013) Offshore insourcing in software development: structuring the decision-making process. J Syst Softw 86(4):1054–1067

Šmite D, Wohlin C, Galviņa Z, Prikladnicki R (2014) An empirically based terminology and taxonomy for global software engineering. J Empir Softw Eng 19:105–153. doi:10.1007/s10664-012-9217-9

Taweel A, Brereton P (2006) Modelling software development across time zones. Inf Softw Technol 48(1):1–11

Van Solingen R, Valkema M (2010) The impact of number of sites in a follow the sun setting on the actual and perceived working speed and accuracy: a controlled experiment. In: Proceedings of the 5th IEEE international conference on global software engineering (ICGSE), pp 165–174

Biography Darja Šmite is an Associate Professor of Software Engineering at Blekinge Institute of Technology in Sweden and a part time Professor in the University of Latvia. Her major research interests are related to global software development. Experiences discussed in this book chapter are based on her research observations collected from a variety of onshoring and offshoring projects in Latvia, Sweden, Norway, Russia, India and China. Prior to her academic career, Šmite has worked in industry.

Chapter 13
Management and Coordination of Free/Open Source Projects

Ioannis Stamelos

Abstract Developing software in the free/open source software (F/OSS) way is fundamentally different from the conventional, closed, team-based, single-owner software project. As a consequence, managing a F/OSS project is done in a quite different way, emphasizing on people and community coordination and organization. Management organization may take extremely distant forms: absolute monarchies, oligarchies, or open source democracies, with community members voting to decide project evolution. On the other hand, not all F/OSS projects are based on pure voluntarism. Many such projects are sponsored by firms and are managed in their own way. In addition, certain extreme project transformations, such as forking, occur frequently in F/OSS. Human resource management (e.g., team building) and decision making (e.g., project cancellation) are also done in a completely different way. This chapter focuses on how human resource management and project organization are handled in F/OSS today. On the other hand, there are several areas in which, given enough research and experimentation, new tools may be provided to assist informed, successful F/OSS management, aiming to help both experienced and novice F/OSS coordinators. The chapter attempts to foresee how measurement, simulation, and antipattern techniques might help F/OSS managers in the future.

I. Stamelos (✉)
Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece
e-mail: stamelos@csd.auth.gr

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_13, © Springer-Verlag Berlin Heidelberg 2014

13.1 Introduction

Free/open source software development has grown to become a paradigm for producing numerous software applications in all domains, often with higher or comparable quality than the traditional, closed source way (e.g., in Coverity 2013 it is reported that at least for small projects F/OSS exhibits superior quality). F/OSS basically provides software users with three freedoms: freedom to inspect the code, freedom to redistribute the code, and freedom to modify the code (Free Software Foundation[1] Open Source Initiative).[2] Such freedoms are ruled by the distribution license under which the code is released. F/OSS has produced highly successful software items, such as the LINUX operating system, the Apache Web Server, MySQL, LibreOffice, and so forth. Today, hundreds of thousands of F/OSS projects exist, residing either on individual web sites or on forges, such as SourceForge [3] or GitHub.[4] SourceForge alone hosted over 300,000 projects in May 2013.[5] F/OSS has affected traditional software development a lot. The reader may refer to Chap. 13 for a comprehensive analysis of how F/OSS development practices may help software companies develop their own code.

On the other hand, many of the F/OSS projects are dead projects, that either have never really started or have seized attracting any interest. There is much hype and many pitfalls in the realm of F/OSS, and disciplines such as project evaluation and assessment are of paramount importance.

Code openness dictates that the development process be fundamentally different from the closed source paradigm. Moreover, F/OSS conveys both strong ideological components, such as the lack of copyright rules and free circulation of digital information (Sandred 2002), and unique business models at the same time.

As mentioned in Chap. 1, management is about planning, organizing, leading, staffing, and controlling of projects. F/OSS management is concerned with similar activities. Because of the unique F/OSS project nature, successfully managing a F/OSS project involves different and new challenges, disciplines, problems, tools, and techniques. However, F/OSS management has attracted little attention from researchers and book authors up to now compared to the extent of the closed source management literature.

The operational context of F/OSS projects is quite complicated. For example, not all F/OSS projects are initiated by individual programmers, as is commonly thought, or based on pure voluntarism. Many F/OSS projects are initiated and/or sponsored by firms and are managed in their own way (Capra et al. 2011). Hundreds of F/OSS projects are initiated or even hosted by public entities,[6] sometimes quite

[1] http://www.fsf.org/.

[2] http://opensource.org/.

[3] http://sourceforge.net/.

[4] https://github.com/.

[5] http://sourceforge.net/apps/trac/sourceforge/wiki/What%20is%20SourceForge.net.

[6] http://joinup.ec.europa.eu/, EU open source reuse repository for public administrations.

unexpectedly.[7] In addition, certain extreme project transformations, such as *forking*, occur frequently in open source. Forking in F/OSS occurs when part of a project community continues code development in an independent way, while the original project keeps on. Forking leads to a new project and a new software product. One recent event that has attracted a lot of interest is the forking of LibreOffice from the original OpenOffice project due to the acquisition of SUN, former OpenOffice owner, by ORACLE.

In F/OSS, management organization may take extremely distant forms; one may encounter monarchies, oligarchies, and democracies in F/OSS style, with community members voting to decide upon project evolution. Certain projects exhibit unique features. An example of a unique project structure in open source is the concept of project incubation, practiced by the Apache Software Foundation (ASF). Once the project is created, it goes through a phase in which the project is consolidated in several ways: a self-sustainable community is built, licensing is handled to adhere to ASF legal requirements, code is refactored, etc. When all these issues are set, the project becomes a regular ASF project.

Human resource management is also done in a completely different way. As an example, consider team building, an essential conventional software project management task. In F/OSS, team building is strongly related to attraction and recruitment of volunteers (rather than interviewing and hiring newcomers), maintaining their interest in the project and providing a motivating, rewarding, and socially pleasant development/use environment (see also Chap. 10). F/OSS coordinator fame, visibility, and recognition by the F/OSS communities are of paramount importance for attracting participants. Anecdotal evidence suggests that recovering "lost" contributors is another non-conventional human resource-related activity. Decision making in F/OSS poses problems that are unique, such as the assessment of the project status and its chances to continue successfully its trajectory in the F/OSS ecosystem.

Management needs its own computer tools and, as one might expect, F/OSS has produced several project management software tools, supporting typical management activities such as project scheduling. However, there is a lack of project management tools that are able to address the coordination and management activities and problems that are specific to F/OSS and that are discussed in the following.

The chapter briefly reviews the basic differences between the two development approaches and provides answers, such as what is a F/OSS schedule or what and how human resources are used in open source projects. It then proceeds with the review of the basic differences in the problems with which management is faced in F/OSS and of the relevant management activities. F/OSS management organizational structures, decision making, resource allocation, and management tools are some of the management issues that will be detailed and analyzed. A case study is presented, showing how F/OSS concepts may be combined with good industrial

[7] http://brlcad.org/, advertised by U.S. Army as the world's oldest open source software.

practices. Furthermore, good practices for project documentation, measurement, and reuse that might be easily adopted in F/OSS are examined. The last part of the chapter examines more challenging areas in which F/OSS management might benefit from future research outcomes. Such areas include antipatterns, project evolution forecast, and success assessment. Finally, the need for special education F/OSS management resources is anticipated.

13.1.1 Differences Between Free/Open Source Projects and Conventional Projects

We need to understand the diverse nature of F/OSS compared with closed source in order to assess their differences in terms of management. On the other hand, there are certain important similarities between F/OSS and agile projects. Some major F/OSS—closed source differences, directly affecting the way F/OSS is managed, are as follows

- F/OSS projects are not triggered by a specific customer request. They are initiated by a so-called "personal itch" (Raymond 1999). In the case of *hybrid* projects, they are initiated by a company business plan that tries to explore the benefits of software development by F/OSS communities (Sharma et al. 2002).
- F/OSS processes are fundamentally different from typical closed source life cycle models, such as the waterfall or spiral model. A typical development process is composed of an eternally perpetuated commit—debug—release cycle. Abrupt changes in the codebase may also occur because of code donations, cancellation of large code chunks, refactoring, etc.
- There is no such thing as a detailed baseline budget or schedule. Few projects have specific time schedules and task assignments (e.g., FreeBSD[8]). Informal time plans, by setting project milestones in terms of new features to be implemented, may be agreed (see the case study presented later in the chapter), but you should not expect any Gantt diagrams. Pursuing of project goals is less strict, and such time plans are mostly used to monitor progress and may be easily modified if milestones are not achieved. F/OSS projects evolve according to individual commits of code and other artifacts, such as translations, depending on the personal agendas of the community members. As such, they cannot typically be foreseen or anticipated in a systematic way. F/OSS projects appear to evolve as phenomena or events occur in natural ecosystems (Thomas and Hunt 2004).
- In most cases, there is no remuneration for participating in a F/OSS project. This is not always the case as there are many F/OSS programmers that are paid by a

[8] http://www.freebsd.org/.

company that is somehow interested in the project, or paid to add a specific functionality (Capra et al. 2011).

- As a consequence of the voluntary participation, there is no personnel hiring. F/OSS project participants are attracted by the name of the coordinator, often a software guru, or potential contributors find the project by browsing and searching the Internet, or maybe are invited to contribute (Antoniades et al. 2002).
- Formal project management practices, such as formal risk analysis and planning, or formal quality reviews, are rarely adopted. However, F/OSS projects progress and often achieve admirable results without them (Aberdour 2007). Certain informal quality assurance activities, such as code inspection or refactoring, are done on a daily basis and greatly contribute to the health of technically successful F/OSS systems.
- Project personnel roles may also differ drastically. Participant roles are not always clearly defined, and often any community member is more or less expected to contribute in any way he[9] can (Michlmayr 2004). One striking fact is that not even the roles of user and developer are distinguished; very often developers are also users of the F/OSS application. A F/OSS project often starts because of the need of a user who becomes developer (Raymond 1999). Even the role of the coordinator is blurred; many projects have a team of peer coordinators. However, in well-organized F/OSS projects, specific roles may be clearly assigned to participants.
- As a consequence of the flexible way the projects are managed, decision making is not monolithic, and often all community members may have a say on the direction the project should take. The project overall organization (i.e., the structure of authority, along with division of labor and decentralization mechanisms (Wynn 2003) dictates the exact way decisions are made.
- Project release intensity is another issue: A famous saying about F/OSS is "release early, release often" (Raymond 1999). Iteration and phased development are considered important in closed source as well[10] however, releasing in F/OSS is much more intensive.
- Unique management decisions: F/OSS managers need to make decisions that are unique in the software world. For example, they need to assess ideas for new features (see the case study later in the chapter) or assess multiple code commits that provide identical functionality to the project and choose one of them to include in the codebase (as may happen in the LifeRay project[11]). Or they have to face extreme project situations, such as project forking and community splitting.

[9] Male managers in FLOSS are an order of a magnitude more common, thus we use from now on.

[10] This is evident if we look at many widely known traditional process models, such as the Rational Unified Process or Boehm's spiral model.

[11] LifeRay Inc. chooses what features proposed by the community to develop, but encourages developers to do the same see http://www.liferay.com/community/ideas.

The above hold in most F/OSS cases; given the multitude of F/OSS projects, the reader should expect that exceptions are commonly found. Several F/OSS projects may exhibit management practices that remind a lot of those exercised by traditional project management. In addition, there are similarities with global software development projects (Chaps. 9, 10, and 12), such as distributed software development, and team communication and work split issues.

13.1.2 Similarities Between Free/Open Source Projects and Agile Projects

On the contrary, there are several similarities with agile project management (see also Chap. 11). Agility in software is trying to avoid the pitfalls of bulk, bureaucratic software development.[12] In doing so, agile practitioners propose agile project management approaches. Agile projects are closer to F/OSS projects, and there are certain similarities that are fairly easy to spot. For example, by looking at the principles of agile methods, one may observe the similarities with F/OSS development:

- Customer involvement in F/OSS is achieved by having mixed user and developer communities.
- Incremental delivery is achieved in F/OSS through frequent releasing.
- Focus in both cases is on people and not on processes.
- Change is natural and embraced in F/OSS (subject to community consensus).
- Self-organized teams are a characteristic of both agile and F/OSS paradigms.

By looking at the 12 practices of eXtreme Programming (XP, Beck 1999), we may actually spot additional F/OSS practices. XP favors *small releases* in order to allow immediate use of any new functionality added, just as Raymond suggests, in order to tighten the interaction loop between F/OSS developers and users and foster feedback from the community (Raymond 1999). XP also pursues *simple design*, avoiding complicated upfront architecture designs. This is often observed in F/OSS, where software architecture is not always documented, and development often proceeds with small ad hoc additions to the existing design. *Refactoring* is also common to both XP and F/OSS: as an example, an empirical study revealed that seven F/OSS Java programs had been subject to refactoring, although code changes were found to be rather small (Advani et al. 2005). *Collective ownership* in XP is a straightforward concept for F/OSS and *continuous integration* is a direct result of the “release early, release often” principle. The *sustainable pace* XP practice is easily achieved by F/OSS due to the volunteering nature of participation and the loose time schedules. Finally, one can draw analogies between the *on-site customer* XP practice and the availability of a community of users in F/OSS. F/OSS users are

[12] http://www.agilemanifesto.org/.

not physically present when code is shaped by the developers, but are actively involved in the whole process. Absence of physical presence is counterbalanced by their numbers and interest in the project.

As a consequence, we may come to the conjecture that agile and F/OSS project management have several things to share. Not everything, however, is different when compared to closed source management. For example, task definition, prioritization, and task allocation to resources are everyday activities for F/OSS managers as well. Task definition may stem from feature requests or *wish list* items by the community or by the F/OSS manager himself. F/OSS managers, supported by defect tracking software (e.g., Bugzilla),[13] may determine bug severity levels, and decide bug fixing prioritization and successive assignment to community members.

13.2 F/OSS Management

Taking into account the above discussion, a manager/coordinator of a F/OSS project needs to address several management challenges in new ways. In particular, F/OSS nature demands ad hoc project organization and human resource management, while there is a multitude of "everyday" issues that must be handled efficiently. F/OSS managers have an increased role in the technical aspects of the project and are more heavily involved in programming issues and in project mundane tasks. We next focus on management organization and human resource issue handling.

13.2.1 Open Source Management Organization

Several different forms of F/OSS project organization (or *governance*) are possible, ranging from single project ownership to democratic—heavily based on voting—project organizations. Intermediate, meritocratic project organizations are also common. A known example for the former type is the flagship of F/OSS, the Linux Kernel project, that is managed (and named after) by its initiator, Linus Torvalds. However, although Linus and a number of close collaborators decide which patches and code commits will be adopted in the project, the numerous contributors of the project are not directly controlled by them, and participation is voluntary or guided by interested companies. One may note that there is some similarity with the Product Owner role in SCRUM (see Chap. 11). On the other hand, one example of democracy-like organization is the Debian project.[14] This

[13] https://bugzilla.mozilla.org/.

[14] http://www.debian.org/vote/.

project has its own software for voting, and each year the project leader is elected through a complex voting system.

To obtain an idea of intermediate approaches, let us consider the way the Apache project people describe their own project organization: "The Apache projects are managed using a collaborative, consensus-based process. We do not have a hierarchical structure. Rather, different groups of contributors have different rights and responsibilities in the organization."[15] The Apache organization defines this project organization as a *meritocracy*.

The Apache project is also a good example of another interesting organizational concept, namely, *F/OSS incubation*, similar to the idea of start-up company incubators. Existing F/OSS projects that wish to become Apache projects first enter an incubation, preparatory stage, where technical and organizational issues are settled, and then become full-fledged Apache projects.[16] Projects are ensured to adhere to the Apache meritocracy and legal rules, and their communities are developed. One striking recent example is the OpenOffice project[17] donated to ASF by Oracle, after the acquisition of SUN.

Shifting from one organization model to another is not uncommon, and it is a milestone event for any F/OSS project. For example, in 2003 Mozilla moved from a totally free contribution model to a rather rigorously controlled model (Holck and Jørgensen 2005).

F/OSS managers can be individuals or project management committees. In the following, we use the term "F/OSS manager" for both cases. Nevertheless, the project organization influences all management-related issues that are discussed in the following sub-sections.

13.2.2 Open Source Human Resource Management

It is widely acknowledged that the human factor plays an ever increasing role in software development (IST 2014). F/OSS is no exception to this fact, and probably the human nature plays an even more crucial role here, because of the inherent liberalism, the ideological background, and the voluntary participation (Sandred 2002).

The first issue to resolve is to define who is the manager of the project. Typically, if one starts his own project he becomes the leader. Often, project initiators abandon their project (this is probably a bad sign itself) and someone else takes their place. As we have seen, managers may also be elected by the community or the developers alone. As in closed source, managers need not be the best developers, rather they must be the best leaders, those who believe in the project and are capable to

[15] http://www.apache.org/foundation/how-it-works.html#meritocracy.

[16] http://incubator.apache.org/.

[17] http://www.openoffice.org/.

handling and resolving community issues. Nevertheless, very often in F/OSS development the managers are very good developers.

A F/OSS manager needs to create a healthy community around his project at the beginning. In later phases, he needs to enlarge/maintain the community. Attracting/retaining people in the project is therefore of paramount importance. F/OSS managers need to leverage the salient differences between their project and other projects, either F/OSS or closed, by exploiting any communication channels available, such as the project web-site, chatting, discussion lists and forums, and the project documentation. Recruiting may involve contacting individually prospective community members and convincing them to participate. Open calls for contributors of all kinds are also common.

Once attracted and fetched into the project, new entries must be encouraged and their interest maintained. One successful mechanism, practiced by many renowned projects, such as the Apache[18] and FreeBSD,[19] is *mentoring*. Newcomers are guided by appointed experienced members to accomplish tasks that are requested by the project. This is no easy match; *mentees* are typically asked to dedicate enough time and efforts to fulfill the expectations of both the project and their mentor.

Project support mechanisms also attract people. A project with no FAQ list or discussion forum is not attractive since newcomers cannot find the information that would lead them to successfully enter the project. Another project aspect that attracts new participants is the structural quality the code and the amount of comments in it. Bad, hard to understand code will hardly generate a thriving community around it. One critical task for releasing F/OSS code on the Internet is to refactor it and improve its overall appearance.

Not all community members are active in the project. A large part of them is *lurking*, i.e., they observe project developments, read forum posts, download new versions, but do not actively participate. Motivating and mobilizing lurkers is important for managers in order to reinforce and revive their projects, by recruiting invaluable resources to ensure sustainability and project evolution.

Free participation and liberalism in F/OSS has its own cost. A frequent observation is that weird, often anonymous people, hidden behind an e-mail address or a nickname, pester serious F/OSS projects in various ways. *Flame wars* is another common phenomenon; they may take the form of hostile interaction between two community members that disagree on whatever issue has arisen, or they may spread widely and involve entire communities. A recent such event is the user reactions against Ubuntu's Unity desktop,[20] caused by rapid changes in desktop user interfaces in the past couple of years. Although "flame wars" may be an inappropriate term for all cases, fighting and lively argumentation over various F/OSS issues on the Internet is extremely common. F/OSS managers need to handle annoying

[18] http://community.apache.org/mentoringprogramme.html.

[19] http://www.freebsd.org/projects/summerofcode.html.

[20] http://www.ubuntu.com/.

interventions in their projects and mitigate disagreement among community members to ensure smooth project evolution, without altering the free expression of personal opinions that has helped so much F/OSS grow up to its current dimensions.

13.2.3 A F/OSS Project Case Study: Plone

This subsection provides a brief account of one F/OSS project, namely Plone,[21] focusing on management/governance issues. Plone is a popular content management system, built upon the Zope[22] application server. It is owned by the *Plone Foundation*, a not-for-profit organization. Plone Foundation members are community members that have contributed significantly to the project. It is run by an elected board of directors. There are committees that examine membership applications or deal with intellectual property and licensing issues.

The project is supported by more than 300 solution providers in 57 countries (data retrieved from Plone Website as of September 2013). The project started in 2001 and is particularly active, with a thematic conference organized yearly (a common characteristic of large and successful F/OSS projects).

Plone development base is organized around a group of *core developers* and a large group of "peripheral" users. Developers enter and exit this core group according to their personal commitments and careers (Aspeli 2005). Their number has grown to more than 300 currently. The user community is very lively and supported by several mailing lists. However, the bugs they report are fixed mostly by the core developers because of the complexity of the software (in other F/OSS cases, users often fix the bugs they discover).

Decision making is consensus-based. Initially, whenever necessary, decisions were forced by prominent members of the community. This was tolerated by the community because they still had the feeling that their opinion counted (Aspeli 2005). Because this governance form might not work well as the community continued to grow constantly, the project moved to a team-based structure, assigning specific responsibilities to several different members, as described later.

The release process of Plone is rather interesting: it is time-based, aiming to produce releases "of predictable size and complexity". The *PLIP* process is followed, with a 6-month fixed release cycle, a release strategy that is common in large software organizations but not in F/OSS. Within the 6-month release period, there is a "feature freeze" interval of 4 months to increase system stability. As a consequence, non-timely code commits may wait for several months before getting incorporated in a new release. Minor and major milestones and releases are planned and monitored in the Plone roadmap, including information for accepted and

[21] http://plone.org/.

[22] http://www.zope.org/.

rejected features. A ticketing system is used to keep track of the several features involved.

Two major roles in their release process are the *release manager* and the framework team. The release manager coordinates a series of releases and is appointed by the Plone Foundation board. He has the authority to make final decisions about the releases he is in charge of. The framework team supports the release manager by inspecting and reviewing candidate features. When a new feature is approved, a member of the framework team is appointed as *champion* of the feature and works with the community member(s) to complete it. A prespecified number of reviews must be successfully passed before a feature makes it to a new release. A critical component in the Plone structure is the *Quality Assurance Team*, responsible for bug handling, patch validation, etc. Plone has many more teams in the areas of *Documentation, Internationalization, Security and Continuous Integration and Testing.*

Plone is a good example of a successful project, delivering a marketable F/OSS product, that combines F/OSS liberties with solid governance structures.

13.2.4 Open Source Project Management Tools

Project management has its own software tools. Most tools help users design and maintain typical management diagrams, such as Gantt or PERT diagrams, in easy to read and modifiable ways. They also help defining resources along with their characteristics, such as cost and type, their groupings, etc. In addition, they assist managers in mundane tasks, such as calculating budgets, identifying *critical paths,* and so on. As discussed above, F/OSS management is rather different than traditional management, so only few of these functions are of real help in F/OSS. More advanced tools might assist F/OSS management, as detailed in the subsequent sections of this chapter.

As one might expect, F/OSS has its own open sourced management tools, addressing traditional management activities. There are many such tools, and a lot of them are web based. Table 13.1 lists several such projects, along with the date of their latest release, even if some of them are somewhat old or not popular. It is a good indication of the interest for project management tools by the F/OSS communities. There is everything there, from personal management tools and bug/trouble tracking tools, to multiproject organization environments. As an example, Open-Proj is a desktop tool that looks very much like Microsoft Project. Almost half of the projects have had at least one release in the past year.

Table 13.1 A list of F/OSS project management tools along with their most recent release date (as of September 2013)

Achievo 27/9/2010	eHour timesheet management 2/6/2013	ProjectPier 8/1/2012
AirTODO 9/9/2007	Kforge 5/3/2013	Redmine 14/7/2013
BORG Calendar 17/7/2013	Memoranda 6/4/2007	Retrospectiva 7/2/2010
Chandler 15/4/2009	OpenAtrium 26/7/2013	SimManTools 30/9/2007
Codendi 21/10/2010	OpenGoo (new name Feng Office) 2/9/2013	TaskJuggler 30/6/2013
Collabtive 22/8/2013	OpenProj 2/10/2008	Teambox 27/5/2011
ClockingIT 18/10/2012	Plancake 22/5/2009	Todoyu 27/3/2013
DotProject 27/7/2013	ProjectHQ 7/10/2007	Trac 7/9/2012
EgroupWare 31/8/2013	Projectivity 14/1/2012	Xplanner 24/5/2006

13.3 Current Challenges in F/OSS Management

A number of F/OSS management challenges have been pinpointed up to now. One further challenge for the F/OSS ecosystem is to overcome the factors that hinder the adoption of F/OSS by both public sector organizations and private companies. For example, in one study (Munoz-Cornejo et al. 2008), a perceived lack of security, quality, and accountability of F/OSS products were among the factors inhibiting adoption in hospitals.

Scientific journal and conference papers on F/OSS have started appearing since about 15 years ago (Querol del Amo 2007), and a significant body of empirical research on F/OSS has been produced, providing several interesting results and additional insight regarding the mechanisms and dynamics of both F/OSS code and communities.

Empirical research was the appropriate way for studying a phenomenon that did not have a formal theoretical basis. In the past 10–15 years, based on empirical findings and longitudinal observation studies, scientists have developed theories that to a large part explain what happens in F/OSS. In addition, just as closed source companies may benefit from F/OSS development practices (see Chap. 14), F/OSS may adopt certain good practices from traditional development that do not alter or otherwise affect the openness concept.

In the following, we review management areas and technical project aspects that may be decided by F/OSS management that may benefit from existing techniques and tools. We will also briefly describe research directions whenever appropriate.

13.3.1 Improved Project Documentation

Concise and clear project description and documentation have been somewhat neglected in F/OSS up to now. Managers need to produce a clear account (on the project web site) of what their project is about, what its current status is and what are the goals that are pursued. This means that detailed project goals must be discussed and decided with the help of the community.

Moreover, project documentation needs to be enhanced. F/OSS has been heavily criticized for lower usability when compared with well-supported and documented commercial software. Standard software project documents, such as plain user manuals, requirements definition, and requirements specification (addressing current and perspective user/developer queries and doubts, respectively), are still missing. Undocumented features, resulting primarily from the lack of user manuals, should be reduced as much as possible to avoid confusion and clarify project scope and tasks to be accomplished.

Improving project documentation will boost productivity and make projects more attractive to the F/OSS world, and will facilitate new entries, causing the influx of even more software people to the F/OSS ecosystems. On the other hand, project documentation requires resources and managers should carefully guide project activity in this regard. For example, it may be wise to pursue the development of a codebase that ensures project sustainability and dedicate resources to documentation in a later phase.

13.3.2 Improved Project Measurement and Qualification

Closed source software development has seen a lot of research and practical application in the area of project measurement and quality assurance. F/OSS has relied mainly on massive code inspection and fault detection and debugging ("given enough eyeballs, all bugs are shallow") (Raymond 1999). However, F/OSS managers may benefit from (open sourced) tools that produce code measurements (e.g., the CScout refactoring browser,[23] by Spinellis) or detect bad smells (Fontana et al. 2012, JDeodorant tool[24] by Chatzigewrgiou). Such tools provide useful quantitative information that may complement managers/community gut feeling and expert opinion about the project quality level, and therefore support informed refactoring. Measurements may be used to compare the codebase to known good quality levels from literature (Samoladas et al. 2008), coming both from closed and open source. Measurements allow benchmarking, that is, comparison with other projects of known good quality levels. Pieces of code that produce low-quality measurement values or that fall under some known class of bad design decisions

[23] http://www.spinellis.gr/cscout/.

[24] http://java.uom.gr/~jdeodorant/.

may become targets for refactoring by the community. Nevertheless, software metrics have their cost and are no silver bullet, and they should be used wisely and only complement expert quality judgment (see Chap. 6 for a comprehensive treatment of quality management).

F/OSS management may also obtain help from innovative tools that are becoming gradually available from social media research. A lot of research has been done concerning *Social Network Analysis or SNA* of F/OSS communities and relative projects (e.g., Madey et al. 2005). SNA may identify senior members of a large community and quantify their contribution and degree of influence, or locate modules that simultaneously change within subsequent releases, pinpointing semantic dependencies in the architecture. As another example, *sentiment* or *opinion detection* tools (Liu 2010) are useful for understanding in depth one's audience. In the future, such tools may be able to understand levels of disagreement or discontent among community members by inspecting postings related to faulty functions or new versions. Although similar information may be obtained through simple inspection of forum or discussion lists messages, such tools may process automatically large numbers of posts in extended time periods and detect trends and other interesting phenomena. F/OSS managers that are eager to use opinion detection tools may have a greater chance to understand their community better and potentially make better decisions.

Measurement of community attributes may help make informed decisions. For example, releasing of new code versions needs to be done in the right time, not necessarily too often or too early. Tools are needed to help managers decide when to produce a new release in order to maximize project usefulness and image, and monitor community fatigue effects. The sentiment analysis tools mentioned above may be used to this end.

13.3.3 Extensive and Systematic Reuse

Although reuse is practiced extensively in F/OSS, it is mainly limited to relatively small code components or utilities, such as items typically found in software libraries (e.g., the Apache Commons).[25] However, the availability of billions of open lines of code in forges or independent project sites provide, a basis that might help implement an old dream in software engineering, namely, the software component markets. Repositories of domain-specific open sourced components (Kakarontzas et al. 2012) would foster the rapid development of new projects, with programmers focusing on the innovative part of the applications.

Code openness would alleviate several problems of closed source components, such as component trustworthiness, modifiability, and qualification. For example, open component code allows identification and correction of defects of all kinds,

[25] http://commons.apache.org/.

including potential threats to security. Third parties may be hired by component consumers to qualify open source components, with the purpose of incresaing their trustworthiness. In addition, users are free to modify components as they wish, provided that they respect the potential limitations of the accompanying license. The implementation of high-quality, documented, reusable open source code would facilitate its packaging as a component for further reuse.

13.4 Future Open Source Management Techniques

In this section, we review some interesting research results that could help F/OSS management in the future. As already mentioned above, F/OSS adoption by both public sector organizations and private companies is often hindered by the ambiguity that surrounds F/OSS projects and their future development (Munoz-Cornejo et al. 2008). Therefore, F/OSS management areas of particular interest are the assessment of project evolution and assessment of project success chances, attempting to answer basic questions that every manager and F/OSS user would pose to himself about a project. In addition, problematic situations in F/OSS (i.e., antipatterns) may be systematically coded to form a base from which interested F/OSS managers may retrieve packaged knowledge about how to identify and solve recurring management problems. Eventually, the availability of advanced F/OSS management techniques will also lead to the need for special education resources.

13.4.1 Assessment of Project Evolution

Starting from a definition of the software process followed in a project, it is possible to obtain a dynamical simulation model. In closed source development, processes are already quite well defined, and many process models (waterfall, spiral, RUP, etc.) are in place since many years. Moreover, system dynamics (Abdel-Hamid and Madnick 1991) is being used occasionally to provide dynamic models, both to foresee project evolution and to understand how management-related decisions will affect the project at hand. The reader may refer to Chap. 17 for more information about software process models and simulation.

A first attempt to produce an ad hoc dynamical simulation model was made in Antoniades et al. (2002). A number of significant F/OSS determinant factors, such as code production rate, F/OSS evolution patterns, and project attractiveness, were used to model the dynamics of a F/OSS project, and a comprehensive simulation model was designed. The model was fitted to a couple of successful F/OSS projects. Accuracy results when the model was applied to new projects were good, but it became apparent that different simulation models were needed in order to take into account the multitude of different F/OSS project types and organizations. It must be stressed that simulation of FOSS projects is not based on simulation of individual

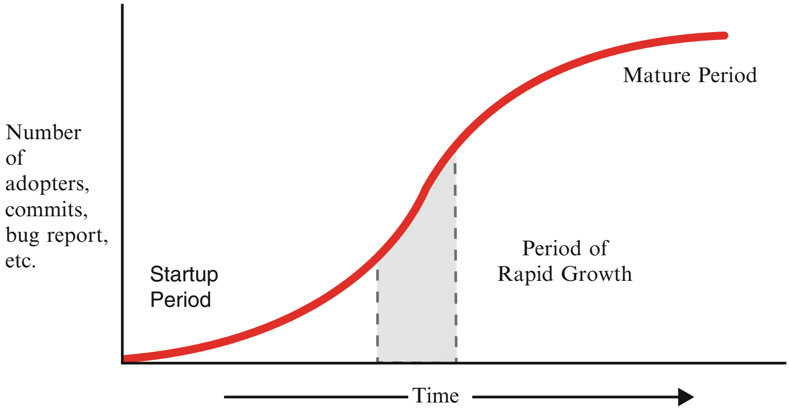


Fig. 13.1 Growth of F/OSS projects at initial phases (adapted from Midha and Palvia 2012)

participant activity. It may be based on code production functions, probability distributions of events like bug detection and removal, etc. Other works in this area have also appeared afterwards (Smith et al. 2006).

An example of useful information that may be derived through such models are the trends for project attributes. Figure 13.1 shows how a F/OSS project might grow (Midha and Palvia 2012). After an initial "incubating" phase (startup period), adopters (user and developers) start to adopt/enter gradually the project. Later at some time (e.g., 6 months after project start), the project receives attention and a period of rapid growth follows (gray area) leading to prolonged period of smooth growth (mature period). As an example, managers of hybrid, company-sponsored, projects may control the evolution of project adoption and predict future situations through what-if scenarios. The whole area appears interesting and promising.

13.4.2 Assessment of Successful Continuation Chances

It has been made clear that F/OSS projects, as all kinds of software projects, are threatened by constant risks: appearance of a strong rival F/OSS project, loss of community interest, deterioration of internal quality, and so forth. Forking, although beneficial from many points of view, is also feared by corporate users who tend to favor project stability. Managers currently use their experience and intuition or other informal means for risk assessment. However, there is one technique, namely, survival analysis, that may help managers assess the probability that their project will continue its life, or alternatively will fail and die, in the close future, that is, in the next few months.

Survival analysis (Evangelopoulos et al. 2009; Samoladas et al. 2010) comes from the field of medicine and aims at assessing the "survival" probability of patients (avoidance of physical death) through a *hazard* function. In F/OSS, such analysis may be made by considering project duration in large sets of F/OSS projects. In Samoladas et al. (2010), survival was defined as the existence of activity (posting in forums, codebase growth, etc.). By analyzing 1,147 SourceForge projects, it was found that survival chances get significantly lower for projects that last for too long time. Survival analysis produces also statistical models (*Cox regression models*) that help foreseeing such probability based on specific project factors, such as the number of project developers. For example, in Samoladas et al. (2010), it was observed that for single developer projects survival probability falls under 20 % after approximately 32 months. It must be stressed that large-scale survival analysis results must be combined with further qualitative information and evaluation of success factors to correctly assess continuation chances for a specific project.

13.4.3 F/OSS Management Antipatterns

In management science, the term *antipatterns* (Brown et al. 2000) represents practices that have negative effects on the projects they are applied on. Antipatterns are recurring, unwanted situations that may stem from misconceptions, wrong application of management practices, personnel problems, etc. Robot (a person who believes that almost anything can and should be automated), absentee manager (a manager who is physically/mentally absent during the project), and metric abuse (excessive confidence on and use of software metrics) are three examples of antipatterns. There is already a body of literature that explores project situations that may be considered as antipatterns in traditional project management. These works define and describe antipatterns, their causes, effects and symptoms, and short- and long-term solutions/remedies. Although problematic situations are reported often in F/OSS literature, there is no extensive and systematic account (see Cerone and Settas 2011 for an initial attempt).

If aware of what antipatterns may occur in his project, a F/OSS manager may learn what negative setups to avoid and know when one such antipattern occurs in his project. Moreover, he will be aware of the solutions he may adopt to mitigate the induced effects. Systematically recording F/OSS antipatterns may lead to a repository that will provide an easily accessible body of precious knowledge. Formalizing the descriptions of antipatterns may help to quantify the relationship between causes and effects, providing tools that may be of further help to future F/OSS managers. It is evident that the idea of antipattern usage is more appealing for less experienced F/OSS managers, as experienced managers of large projects typically handle such problems implicitly.

13.4.4 Integration of the Management and Business Perspective: Assessment of Business Success

Defining project success is a hard and confusing task even for closed source projects (see Chap. 2). Several authors have already looked at ways to define success and success criteria in F/OSS. Success in F/OSS is a multidimensional concept (Crowston et al. 2006). Among the criteria proposed are project continuity and release frequency (Raymond 1999) and market penetration (Feller and Fitzgerald 2002).

A frequent, natural development in successful F/OSS projects is the formation of a business entity (a company) or a not-for-profit entity (organization) that steers the evolution of the project. Sometimes, as is the case of the hybrid projects, it is a company that releases the first version of the code (e.g., opens the closed code of one of its commercial applications, in dual licensing mode), so the manager comes already from the business world. Public organizations may also initiate F/OSS projects.

Project success leads the community to seek business models specific to F/OSS. In F/OSS, in addition to his technical duties, the manager of the project is heavily involved with the business implications of his project. Such implications affect the project organization (e.g., through forking or community migration), the project code (e.g., in case of dual licensing, with part of the code being open, while the rest is commercially exploited under proprietary licenses), and miscellaneous supporting activities (e.g., translations or localizations).

We have already discussed one possible way for assessing technical success, namely, through measurement and benchmarking. If there is a business model behind the F/OSS project, the manager is expected to be able to assess the business success of the project. To do so, he may use F/OSS market success assessment models that are becoming available.

One such model (Midha and Palvia 2012) attempts to assess market success by combining internal (technical) and externally visible attributes of the project. Internal attributes affecting Market Success are License Type (whether permissive or restrictive for commercial use), Size of User Base, Size of Developer Base, Language Translations (how many times the project has been translated to different languages), and Responsibility Assignment (whether tasks are delegated explicitly to project members or not). Internal attributes affecting Technical Success are Complexity (measured through the cyclomatic complexity metric, McCabe 1976) and Modularity (measured through the number of software modules). Figure 13.2 depicts the market and technical success model. The model suggests a rather intuitive concept: Technical Success is among the factors that contribute to Market Success. However, in Midha and Palvia (2012), this hypothesis has not been supported by the empirical data they analyzed. Apparently, further research is needed to consolidate such conjectures. Once such models become sufficiently reliable, F/OSS managers will be able to quantify their overall project business success and compare with rival project performance. As an example, in case

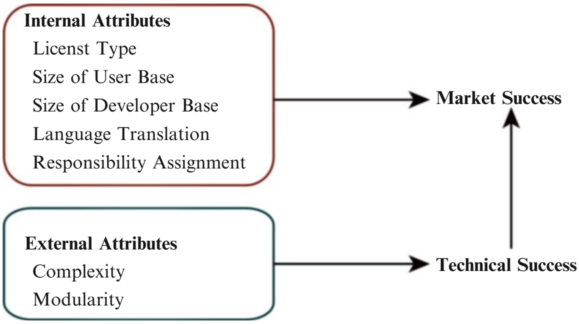


Fig. 13.2 A model for assessing F/OSS market and technical success (adapted from Midha and Palvia 2012). Internal and external attributes are used to assess both technical success and market success

coordinators feel that a project needs improvement, they may want to change the way responsibilities are assigned to community members.

13.4.5 F/OSS Manager Education

For sure, certain F/OSS managers have received some formal training in traditional software project management, including formal project planning, risk management, quality assurance, budgeting, reporting, etc. However, a course on systematic F/OSS project management is not easily found in current higher education curricula. F/OSS managers may nowadays receive training on generic F/OSS issues (e.g., through Free Technology Academy courses).[26] As F/OSS project management progresses as a discipline, it is anticipated that relevant courses will start appearing in established curricula worldwide.

Such a course should encompass standard F/OSS management approaches, as described in previous sections of this chapter, and advanced themes, such as those discussed in this section, including psychology and human aspects of F/OSS development, techniques and tools for project assessment and evolution control, etc.

On the other hand, it is widely recognized that F/OSS projects are excellent bazaars of knowledge sharing and learning (Sowe et al. 2006). Several academic software engineering courses around the world include student practical assignments that require students to participate actively in F/OSS projects. F/OSS managers need to leverage this fact, as it is one way to augment F/OSS communities with new members. Although not without problems (acceptance of students by existing communities, student level of knowledge and skills), blending formal education and F/OSS projects lead to a new dimension of the F/OSS phenomenon.

[26] http://ftacademy.org/.

13.5 Conclusions

In this chapter, after a short introduction to F/OSS, we have identified some of the basic differences between closed source and F/OSS management and similarities with agile methods. We have reviewed basic F/OSS management approaches for project organization and human resource management. We then proceeded by listing some challenges that F/OSS management is faced with and proposed potential improvement actions. Finally, we have tried to identify management areas in F/OSS that may be improved in the future. Although F/OSS managers will certainly continue applying practices that have already led to the success of numerous F/OSS projects, such as intensive personal communication with community members and mentoring, they might benefit in the future (a) from the use of tools and techniques that are successfully deployed in closed source development (related to documentation, qualification, and systematic reuse) and (b) from the use of innovative management tools and techniques that are specific to F/OSS.

Scientific and application areas of special interest for F/OSS management that may produce advanced approaches are (a) F/OSS project success and status assessment, (b) awareness of inappropriate F/OSS management practices, (c) anticipation of F/OSS project future evolution, and (d) education of F/OSS managers and involvement of F/OSS projects in formal education processes.

Most probably F/OSS managers will be skeptical about the use of more formal F/OSS management approaches than those currently exercised. It may be anticipated that managers of hybrid F/OSS projects will probably be more interested in the techniques and tools proposed in this chapter.

References

Abdel-Hamid T, Madnick S (1991) Software project dynamics: an integrated approach. Prentice Hall, Englewood Cliffs

Aberdour M (2007) Achieving quality in open source software. IEEE Softw 24(1):58–64

Advani D, Hassoun Y, Counsell S (2005) Refactoring trends across N versions of N Java open source systems: an empirical study. Technical report, University of London

Antoniades I, Stamelos I, Angelis L, Bleris G (2002) A novel simulation model for the development process of open source software projects. Softw Process Improv Pract 7(3–4):173–188

Aspeli M (2005) A model of a mature open source project. MSc Thesis, London School of Economics

Beck K (1999) Extreme programming explained: embrace change. Addison-Wesley, Boston

Brown W, Malveau R, McCormick H, Thomas S, Hudson T (eds) (2000) Anti-patterns in project management. Wiley, New York

Capra E, Francalanci C, Merlo F, Rossi-Lamastra C (2011) Firms' involvement in open source projects: a trade-off between software structural quality and popularity. J Syst Softw 84 (1):144–161

Cerone A, Settas D (2011) Using antipatterns to improve the quality of FLOSS development. Fifth international workshop on foundations and techniques for open source software certification

Coverity (2013) 2012 coverity scan report. Available from http://softwareintegrity.coverity.com/. Released 7 May 2013

Crowston K, Howison J, Annabi H (2006) Information systems success in free and open source software development: theory and measures. Softw Process Improv Pract 11(2):123–148

Evangelopoulos N, Sidorova A, Fotopoulos S, Chengalur-Smith I (2009) Determining process death based on censored activity data. Commun Stat Simul Comput 37(8):1647–1662

Feller J, Fitzgerald B (2002) Understanding Open Source Software Development. Addison-Wesley, Boston

Fontana F, Braione P, Zanoni M (2012) Automatic detection of bad smells in code: an experimental assessment. J Object Technol 11 (2):5:1–38

Holck J, Jørgensen N (2005) Do not check in on red: control meets anarchy in two open source projects. In: Koch S (ed) Free/open source software development. IGI, Hershey, pp 1–26

IST (2014) Special issue on human factors in software development. In: Amrit C, Daneva M (eds) Inf Softw Technol (Forthcoming)

Kakarontzas G. Stamelos I, Skalistis S, Naskos A (2012) Extracting components from open source: the component adaptation environment (COPE) approach. In: EUROMICRO-SEAA conference, pp 192–199

Liu B (2010) Sentiment analysis and subjectivity. In: Indurkhya N, Damerau FJ (eds) Handbook of natural language processing. Now Publishers

Madey G, Freeh V, Tynan R (2005) Modeling the free/open source software community: a quantitative investigation. In: Koch S (ed) Free/open source software development. IGI, Hershey, pp 203–221

McCabe T (1976) A complexity measure. IEEE Trans Softw Eng 2(4):308–320

Michlmayr M (2004) Managing volunteer activity in free software projects. In: Proceedings of the 2004 USENIX annual technical conference, Freenix Track, pp 93–102

Midha V, Palvia P (2012) Factors affecting the success of open source software. J Syst Softw 85 (4):895–905

Munoz-Cornejo G, Seaman C, Gunes Koru A (2008) An empirical investigation into the adoption of open source software in hospitals. Int J Healthc Inf Syst Inform 3(3):16–37

Querol del Amo M (2007) Open source software: critical review of scientific literature and other sources. MSc Thesis, Norwegian University of Science and Technology

Raymond E (1999) The cathedral and the bazaar. O'Reilly Media, Cambridge

Samoladas I, Gousios G, Spinellis D, Stamelos I (2008) 4th International conference on open source systems, pp 237–248

Samoladas I, Angelis L, Stamelos I (2010) Survival analysis on the duration of open source projects. Inf Softw Technol 52(9):902–922

Sandred J (2002) Managing open source projects. Wiley, New York

Sharma S, Sugumaran V, Rajagopalan B (2002) A framework for creating hybrid-open source software communities. Inf Syst J 12:7–25

Smith N, Capiluppi A, Fernández-Ramil J (2006) Agent-based simulation of open source evolution. Softw Process Improv Pract 11(4):423–434

Sowe S, Stamelos I, Angelis L (2006) Identifying knowledge brokers that yield software engineering knowledge in OSS projects. Inf Softw Technol 48(11):1025–1033

Thomas D, Hunt A (2004) Open source ecosystems. IEEE Softw 21(4):89–91

Wynn D (2003) Organizational structure of open source projects: a life cycle approach. In: 7th Annual conference of the southern association for information systems

Biography Ioannis Stamelos is an Associate Professor at the Department of Informatics of the Aristotle University of Thessaloniki, where he carries out research and teaching in the area of software engineering and information systems since 1996. He holds a diploma of Electrical Engineering (1983) and a PhD in Computer Science by the Aristotle University of Thessaloniki (1988). He has published approximately 150 articles in refereed international journals and conferences.

Chapter 14
Inner Source Project Management

Martin Höst, Klaas-Jan Stol, and Alma Oručević-Alagić

Abstract Software development organizations are continuously looking for better ways to manage their projects. An emerging approach to achieve this is Inner Source, which refers to the adoption of Open Source development practices within the confines of an organization. With an Inner Source approach, individuals, teams, and departments within an organization can start software projects, very similar to the Open Source model. This affects the way projects are managed in a variety of ways. Firstly, it will affect strategic aspects such as a software sourcing strategy that includes decisions on which software can be "Inner-Sourced." Secondly, at the tactical level, organizations should choose an appropriate Inner Source adoption model that suits the goals and scope of the organization. Finally, it will affect the operational aspects of a project, for example, in the way different people across a whole organization can access the source code and make improvements. Furthermore, Inner Source makes communication much more transparent. While Inner Source offers a variety of potential benefits to an organization, there are also a number of challenges to address. This chapter discusses how the introduction of Inner Source may affect conventional software developing environments and especially how it affects software project management aspects. Based on our studies and those presented in the literature, it outlines a number of benefits of Inner Source as well as a number of challenges and some suggestions as to how they can be addressed.

M. Höst (✉) • A. Oručević-Alagić
Department of Computer Science, Lund University, Lund, Sweden
e-mail: martin.host@cs.lth.se; alma.orucevic-alagic@cs.lth.se

K.J. Stol
Lero—The Irish Software Engineering Research Centre, University of Limerick, Limerick, Ireland
e-mail: klaas-jan.stol@lero.ie

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_14, © Springer-Verlag Berlin Heidelberg 2014

14.1 Introduction

The use of Open Source Software (OSS) in industry has seen an unprecedented growth during the last decade. Both the use of OSS development tools, such as Eclipse, and the inclusion of OSS components (e.g., the Apache web server[1]) and software frameworks (e.g., the Struts framework[2]) has increased. OSS refers to software that is distributed under an Open Source license,[3] which permits that the software's source code is freely available to anyone to change to his or her needs (while respecting the conditions of the license). Successful OSS projects are often developed by a large number of disparate, geographically spread, developers around the world, as outlined in Chap. 10.

The perceived success of OSS projects, and the ability to manage distributed development, has resulted in efforts to introduce Open Source development principles *inside* companies; this phenomenon is called "Inner Source" (Stol et al. 2014) though other terms have been used, such as "Corporate Open Source" (Gurbani et al. 2006) and "Progressive Open Source" (Dinkelacker et al. 2002). Stol et al. (2011) defined Inner Source as follows:

> **Definition:** *Inner Source: The leveraging of Open Source Software development practices within the confines of a corporate environment. As such, it refers to the process of developing software at an organization that has adopted OSS development practices.*

Motivations for adopting this way of working include the inherent support for distributed development and the potential to increase software reuse and quality due to an increased availability, openness and transparency of the software, and an implied invitation to anyone in an organization to join development or contribute otherwise (Gaughan et al. 2009).

There are, however, a number of key differences between conventional in-house development and Open Source development. For example, Open Source projects typically do not have the same budget constraints and lead-time limitations that are typical issues for project management in traditional projects. Therefore, Inner Source initiatives always lead to a tailoring of Open Source development practices to fit a corporate context. The objective of this chapter is to present an overview of Inner Source development, and identify implications for typical project management issues.

The remainder of this chapter proceeds as follows:

- Section 14.2 outlines what Inner Source is by positioning it in the software development landscape with respect to other strategies. This section also presents a number of motivations, outlining *why* organizations adopt Inner Source. Furthermore, this section presents different adoption models as well as a number of new management roles that may emerge as a result of adopting Inner Source.

[1] http://httpd.apache.org/.

[2] http://struts.apache.org/.

[3] http://opensource.org/licenses/index.html.

- Section 14.3 presents a framework that identifies a number of key themes related to project management in which Inner Source and conventional project management approaches can differ. This framework serves as a "lens" for the remainder of the chapter, which presents two case studies.
- Section 14.4 presents the case studies that we conducted at two organizations. The first case study presents an analysis of how well an organization's current project management approach aligns with an Open Source development approach. This case study provides insights into some of the key changes that an organization may need to make in terms of project management issues, and as such represents a scenario *prior* to adopting Inner Source. The second case study presents an analysis of an organization that has adopted Inner Source for several years and discusses a number of project management challenges as well as approaches to how those challenges were addressed. As such, this case presents a scenario *after* adopting Inner Source.
- Section 14.5 discusses the findings of the two case studies presented in Sect. 14.4, as well as a number of implications for practice. Based on this, we present a number of recommendations for Inner Source project management. In this section, we also propose a number of new directions for future research.

14.2 Inner Source

This section presents an overview of Inner Source. We outline what Inner Source is by positioning it in terms of openness of the software *product* and software *process* (Sect. 14.2.1). This is followed by a discussion of why organizations would want to adopt Inner Source (Sect. 14.2.2). Section 14.2.3 discusses Inner Source adoption models, and Sect. 14.2.4 discusses a number of Inner Source project management roles.

14.2.1 Positioning Inner Source as a Strategy

As outlined above, the term *Inner Source* refers to the adoption of OSS development practices within an organization's boundaries. It is informative to discuss the implications of this in relation to other trends in the software landscape, such as *open-sourcing* (Ågerfalk and Fitzgerald 2008; Oručević-Alagić and Höst 2010), so as to clearly define the scope of this chapter. Inner Source can be characterized by two features: (1) the development *process* is opened up to the whole developer community within an organization; and (2) the resulting *product* is proprietary, and access to its source code is limited by an organization's boundaries. Figure 14.1 presents a typology using these two characteristics as dimensions.

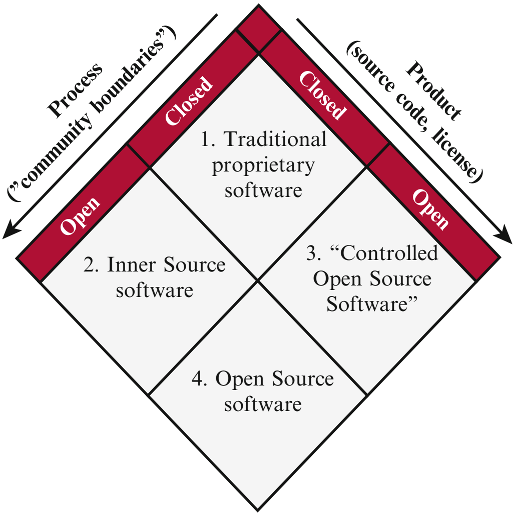


Fig. 14.1 Typology of open and closed software based on *process* and *product* dimensions

We refer to "conventional" development as proprietary software development, using a *closed process*, and a resulting *closed product*. A closed process in this context refers to the fact that there is no "open" development community in which developers can join as new members and start contributing to the development process as they see fit. Instead, in conventional software development, teams are predefined and often organized in a hierarchical fashion. The product is also closed, in that the source code is only accessible by the project team and not publicly available.

Open Source Software is at the other end in both dimensions, with an open process and an open product. A typical OSS project has an open process that welcomes new contributors (see also Chap. 13). In fact, an Open Source project's very success depends on how many developers it can attract. The *product* is open as well as the source code is necessarily available so as to comply with the requirements of an OSS license.

Products that are open but with a closed process is what we label "controlled Open Source Software." Such OSS projects comply formally with an OSI-approved Open Source license,[4] and as such are formally "Open Source." However, in practice, the development process is not open and the product's development is tightly controlled by one organization (or possibly a consortium), or a limited number of individuals. One such example is the Lua programming language,[5] which is designed, implemented, and maintained by a team of three researchers. Lua's source code is freely available under the MIT license. Ideas and suggestions are welcome in the project (and some of the project's key features originated from

[4] As of August 2013, there are 70 OSI-approved licenses.

[5] http://www.lua.org.

such suggestions), but write-access (or "commit-access") to the source code repository cannot be "earned" by others and is limited to the three original implementers (Ierusalimschy 2008). We note that controlled OSS is different from *sponsored* OSS, in which organizations contribute ("sponsor") code or other contributions or resources (Capiluppi et al. 2012). Sponsored OSS projects have an open process, where new developers are welcome to contribute, which differentiates it from "controlled" OSS.

Inner Source, then, is characterized by an *open process*, and a *closed product*. In this context, the process is only "open" to one organization (or a consortium of partners or subcontractors), and not to anyone outside the organization (or consortium). In principle, anyone in the organization is free to submit contributions. The resulting product, however, is still closed, in that it is proprietary and has no Open Source license. Licensing, while a common concern for organizations that wish to adopt OSS (Stol and Babar 2010), is therefore not a concern in Inner Source.

The typology in Fig. 14.1 is, of course, a simplification, a snapshot of reality at any one time and organizations may engage in more than one of the four scenarios. For example, there is an increasing involvement of organizations in sponsored Open Source (mentioned above), and products and services are increasingly based on such projects.

Over time, an organization may change its software sourcing strategy and move to another scenario within Fig. 14.1. For example, organizations may *open source* their product (Oručević-Alagić and Höst 2010), opening both the process *and* the product. This is, for example, what Netscape Corp. did in the late 1990s with their Netscape Navigator, from which the Mozilla web browser project emerged. Involvement of organizations may vary greatly, from being an "industry-led" project (Capiluppi et al. 2012) to total withdrawal of their involvement in the project. In a scenario where an organization is opening up its product from an Inner Source setting, we no longer speak of Inner Source, but of open-sourcing, and the resulting product becomes Open Source (Ågerfalk and Fitzgerald 2008; Oručević-Alagić and Höst 2010).

14.2.2 Motivations and Benefits of Inner Source

Organizations may adopt Inner Source for a variety of reasons and offers a number of benefits to organizations (Gaughan et al. 2009). We discuss some of these below:

- **Improved reuse**. An internal repository or "forge" that hosts Inner Source projects can provide a good starting point for projects, and as such increase reuse within the organization (Dinkelacker et al. 2002; Lindman et al. 2008; Vitharana et al. 2010).
- **Improved quality**. Inner Source projects can benefit from "Linus's Law," whereby a large community of developers peer review contributions. Since

contributions are under large-scale scrutiny, contributors may be aware of their reputation and be motivated to write "good" code (Dinkelacker et al. 2002; Riehle et al. 2009).

- **Rapid developer redeployment.** Since developers are familiar with a standard set of common tools and infrastructure used within an Inner Source setting, as well as with the available software on the internal forge, developers can be more easily transferred to other projects or products (Dinkelacker et al. 2002). This in turn will also reduce time-to-market, as project start-up time can be reduced.
- **Increased awareness.** An open environment facilitates an increased awareness of the software that is developed within an organization (Lindman et al. 2008, 2013).
- **Open innovation.** Besides an increased awareness among developers, a more open development environment may also support the concept of Open Innovation (Morgan et al. 2011; Lindman et al. 2013).
- **Large developers pool.** Open collaboration on software projects facilitates a large developer pool, thereby broadening the expertise of the developer community (Wesselius 2008; Riehle et al. 2009).
- **Increased development speed.** Given a larger development community, development of Inner Source projects may benefit from an increased development speed (Dinkelacker et al. 2002). A single team may not have enough capacity to implement all required functionality (Wesselius 2008). This in turn also supports a faster time-to-market.

The degree to which these benefits can be achieved will depend on how successful an Inner Source initiative is. While the benefits listed here have been reported in the extant literature, Inner Source as a field of research is still in its nascent phase, and precise predictions as to the extent to which benefits can be achieved cannot be given.

An important consideration for any organization with software as a product (or part thereof) is the strategic "make-or-buy" decision, or the sourcing strategy. Van der Linden et al. (2009) presented a "decision map," which distinguishes *differentiating* software (software that provides unique business value) and *commodity* software, such as operating systems, database systems and compilers. According to this decision map, outsourcing "differentiating" software is considered unwise, since this is the (usually relatively small) part of a product that offers a competitive advantage. Commodity software, on the other hand, should typically not be built in- house, as this would waste costly resources. No organization should develop its own database system or operating system; these are commodities and should be acquired elsewhere. Inner Source, then, could be one strategy toward commodification. It is a suitable approach to develop software that is considered sufficiently valuable to develop in-house, while delivering functionality needed by different stakeholders.

One important software architectural strategy within which commodification of software is common is a software product line (Van der Linden 2009). Software product lines typically consist of a common platform on the one hand and a number of derived applications that are based on that platform. Van der Linden (2009) discussed how Inner Source can help to overcome a number of bottlenecks that are common in product lines, such as the dependencies of an application on the platform. Development of the platform may lag behind development of a platform-based application, possibly delaying its delivery to the market. Van der Linden (2009) reported that Inner Source adoption at Philips Healthcare has led to a reduction of the time-to-market of at least 3 months.

14.2.3 Inner Source Adoption Models

Organizations can adopt Inner Source in different ways. In fact, each Inner Source implementation must be tailored to the specific context of an organization (Gaughan et al. 2009). Various factors may influence how an organization implements its Inner Source initiative, such as its product domain, whether or not it is subject to any legal regulations, and an organization's size. However, despite this variety, Gurbani et al. (2010) identified two main models of Inner Source adoption: the infrastructure-based model and the project-specific model. These are briefly discussed below.

14.2.3.1 Infrastructure-Based Model

The *infrastructure-based* model is the simplest model. In this setting, an organization *facilitates* Inner Source projects by providing the necessary infrastructure such as code repositories, bulletin board software, and mailing list servers. These tools enable project teams and individual developers to host an Inner Source project, very similar to how individuals can publish their software on one of the public code repositories, such as SourceForge.net. SAP (Riehle et al. 2009), Hewlett-Packard (Dinkelacker et al. 2002), and Nokia (Lindman et al. 2008) are organizations that have adopted this model.

14.2.3.2 Project-Specific Model

The *project-specific* Inner Source model is a more strategic approach and builds on the infrastructure-based model. In this setup, there is one dedicated division (project team or product group) that takes responsibility for development, maintenance, and support of an Inner Source project, referred to as a shared asset. This division is sometimes referred to as the "core team," similar to a core team in an OSS project. The project-specific model focuses on *strategic* reuse of the *shared asset*. A key responsibility of the core team is to provide ongoing support to customers of the Inner Source project. Since this requires dedicated resources to sustain the Inner Source initiative, an organization may have to choose carefully which projects

Table 14.1 Key differences between infrastructure-based and project-specific inner

Characteristic	Infrastructure-based	Project-specific
Software reuse	Opportunistic and ad hoc. Optimize for sharing maxi- mum number of projects	Strategically planned. Optimize for sharing critical assets
Support	Optional, depends on success of project	Essential for project success. Needs organizational support and funding
Owner/ maintainer	Individual or team who created the project	Central "core team." Needs organizational support
Type of inner source software	Discrete software packages, such as utilities (e.g., XML parser), compilers, shells	Critical asset that plays an important role for the organization

Source models (adapted from Stol et al. 2011)

receive such resources, in terms of budgets and person-years. Case studies of the project-specific model have been reported for Philips Healthcare (Wesselius 2008) and Alcatel-Lucent (Gurbani et al. 2006, 2010).

14.2.3.3 Comparison of Inner Source Models

Stol et al. (2011) presented a comparison of the two Inner Source models, summarized in Table 14.1. This comparison is based on observations from the current literature on Inner Source, and as such, these characteristics are not prescriptive. Organizations may have Inner Source projects that have characteristics of both models.

In the infrastructure model, all four characteristics in Table 14.1 are rather optional and casual, whereas for the project-specific model they are strategic and critical. For example, reuse in the infrastructure-based model is opportunistic and ad hoc, whereas in a project-specific model, reuse is a strategic goal. As a result, support is essential in the project-specific model, and typically one can observe a special organizational unit (i.e., a core team) that takes responsibility of the shared asset that is managed as an Inner Source project.

14.2.4 Inner Source Project Management

Open Source style development is often misinterpreted as a "chaotic" or "unmanaged" approach to software development, while it is better characterized as "self-organizing." Common Open Source governance models range from highly centralized, typically led by a "benevolent dictator for life" such as the Linux Kernel, to decentralized ones, such as GNU Linux. In between those two mentioned models sits a council-like governance model that is used, for example, in the

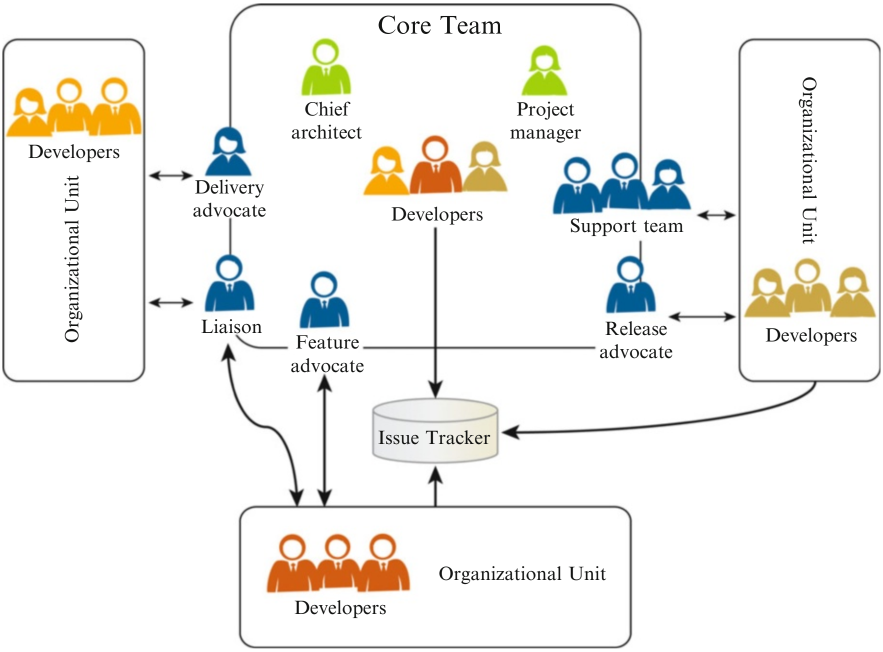


Fig. 14.2 Roles of an Inner Source core team (based on Gurbani et al. 2010)

Apache project. Hence, just as successful Open Source projects can have an extensive "management layer," so too do Inner source projects need management.

Little is written on project management in Inner Source projects. A notable exception is a study by Gurbani et al. (2010), which identifies a number of roles that emerged in Alcatel-Lucent's Inner Source project. The diagram in Fig. 14.2 presents our analysis of their presentation of these roles. In the remainder of this chapter, we assume that an organization consists of one or more *organizational units*, such as departments, teams, or business divisions.

It is important to emphasize that these roles emerged within the core team at one organization and are not prescriptive for an Inner Source initiative. They do, however, provide a useful reference model for organizational roles in Inner Source. We describe these roles briefly.

A *liaison* has the overall responsibility for the shared asset and oversees the activities performed by the core team. The liaison also interacts with the organization's internal customers (i.e., different organizational units) about, for example, feature requests. The liaison works closely with the shared asset's *chief architect*; the chief architect can be compared to a typical OSS project's benevolent dictator, who has strong technical skills and whose key responsibility is strategic road mapping and maintaining the shared asset's architectural integrity.

A *support team* (Gurbani et al. (2010) refer to them as *construction, verification and load bring-up engineers*) provides operational support to the business units.

This includes release management tasks, verification, documentation, and writing release notes. A *project manager* has overall project management responsibility, which includes release planning, project monitoring, and process compliance. A *release advocate* takes responsibility for a specific release and interacts closely with the organizational units so as to assess the potential impact of the release on the organizational units' products that integrate the shared asset. A *delivery advocate* is assigned to an organizational unit that becomes a new customer of the shared asset, so as to assist in integrating the component into a product. Gurbani et al. (2010) pointed out that an organizational unit might have specific tools, which complicates the task of integrating the shared asset. The delivery advocate also assists in ensuring that contributions from the organizational unit fit within the shared asset's overall architecture. Finally, a *feature advocate* is responsible for seeing a particular feature to completion.

14.3 A Framework for Understanding Project Management in Inner Source

Inner Source and project management are both complex and multifaceted topics with wide scopes and comprising many different aspects. One goal of this chapter is to identify key aspects of project management that are affected by adopting Inner Source. In order to define the focus of our study, we defined a framework based on a number of aspects that we consider important for understanding project management in Inner Source. The remainder of this chapter uses the framework to structure the results of our analysis. We derived this framework from a number of sources that have addressed the topic of project management in detail, which we briefly outline below.

Much has been written on project management in general, but also in a software engineering context as exemplified in this book. An important source is the Project Management Body of Knowledge (PMBOK) (Duncan 2013). This is a general guide to project management without a specific focus on software development. It includes an extensive range of factors, or topics, which are listed in the first column of Table 14.2. However, not all topics are related to Inner Source, which is why PMBOK is not an optimal instrument to focus our study.

Besides the project management field, the software engineering field also has a "body of knowledge," called SWEBOK (Abran and Moore 2004). SWEBOK is an ongoing project, currently in its third version, and "accepted knowledge" of the software engineering field. Similar to PMBOK, SWEBOK defines a number of themes, listed in the second column of Table 14.2. While all are specific to software engineering, not all themes are relevant to project management. Hence, SWEBOK would not be an optimal choice as a framework either.

In addition to these two potential sources, various books have been written on software project management (SPM), such as that by Hughes and Cotterell (2009).

Table 14.2 Themes discussed in various sources relating to project management

PMBOK	SWEBOK	SPM textbook (Hughes and Cotterell 2009)
– Integration management – Scope management – Time management – Cost management – Quality management – Human resource management – Communications management – Risk management – Procurement management – Stakeholder management	– Initiation and scope definition – Software project planning – Software project enactment – Review and evaluation – Closure – Software engineering measurement	– Project evaluation and program management – Selection of appropriate project approach – Software effort estimation – Activity planning – Risk management – Resource allocation – Monitoring and control – Managing contracts – Managing people in software environments – Working in teams – Software quality

This source includes all factors from both PMBOK and the "Software Engineering Management" part of SWEBOK; the third column in Table 14.2 lists the themes discussed by Hughes and Cotterell. However, while all themes identified by Hughes and Cotterell are relevant to project management and software engineering, not all are relevant to Inner Source. In order to define a sound framework, we identified four key themes that encompass the relevant topics from the three sources listed in Table 14.2. These are

1. **Process management**: This includes deciding in detail what development process to use during the project. In many cases, this is done by tailoring an organization-level process model to a project-specific process model that fits the constraints and requirements of a specific project.
2. **Project planning**: This includes traditional planning activities such as activity planning, effort estimation, and resource allocation.
3. **Monitoring and taking actions**: This includes activities for monitoring and control, as well as activities related to risk management.
4. **Human issues**: This includes issues related to human resources, people management, and activities that support a healthy team climate.

These themes are used to structure the remainder of this chapter. We describe the themes in further detail below. This section ends with an overview of a number of key concerns of project management in Inner Source that require attention.

14.3.1 Process Management

Process management is concerned with selecting, tailoring, and aligning a suitable software development process for a project. That is, it is a "high level" and often organization-wide project management activity. Since the use of heterogeneous processes may also be an issue in conventional software development contexts, one could argue there is little difference with an Inner Source approach. There is no single "conventional process" and no single Open Source process that is used in Inner Source. Therefore, there is no single way of, for example, adopting and tailoring processes in any of the cases. Each Open Source project has its own set of practices and customs that have emerged over time. However, as outlined in Chap. 13, there are a number of common characteristics. Likewise, an Inner Source initiative is also tailored to an individual organization. Section 14.2.3 discusses the two major Inner Source adoption models, which is the first major difference between Inner Source initiatives. Furthermore, each initiative is shaped by the context and constraints of the organization. For example, Philips Healthcare develops a Software Product Line (SPL) for their medical equipment, whereby the SPL platform is managed as an Inner Source project (van der Linden 2009). As such, the company is subject to regulations set forth by the Food and Drug Administration (FDA) in the United States. One implication of this is that the process needs to be traceable, and regulatory bodies (such as the FDA) conduct audits regularly to inspect the process.

Even though there are many different development methods, making it difficult to identify common characteristics, we discuss a number of common questions that may arise in relation to process management.

Two types of processes are of interest in this respect. First, there is the process that is selected for development of an Inner Source project. Second, there are the processes that are used by the customer teams that wish to integrate (or use) the Inner Source project, and possibly contribute to it. This potentially large variety in customer projects means that there may be a range of different processes that interact with an Inner Source project's process. Some customer projects may follow a strictly stage-gated model (e.g., waterfall), whereas others follow more iterative approaches such as Agile methods (e.g., Scrum). When customer projects wish to contribute to the shared asset, they must consider the alignment of their own process and the process used by the core team that develops the shared asset. A misalignment of such processes may result in problems when the shared asset is integrated, or worse, when a customer team misses a product release deadline (Stol et al. 2011).

There are also requirements regarding the process for an Inner Source project. This process must be formal and sufficiently rigorous to fulfill the requirements of a product and its evolution, that is, handle contributions from a variety of sources in an efficient way. It must also be aligned to other processes at the organizational level, which are not necessarily adapted to this way of working. That is, the process must resemble an Open Source process with its mechanisms for evaluating contributions, widely available and accessible information, selecting among candidate changes in a timely manner, etc., and at the same time adhere to the typical organizational level process requirements that are characterized by milestones and deadlines. Conflicts may arise between a general organizational process and an Inner Source process (Riehle et al. 2009).

Customers of an Inner Source project may use a variety of processes, which is why it is difficult to outline general guidelines as to how these should be aligned. In one case study, Lindman et al. (2008) found that they are typically agile. However, one type of required process tailoring relates to contributions to an Inner Source project. Contributors must adhere to the requirements of an Inner Source process prescribed by the core team, and the level of formality enforced by the core team must be taken into consideration.

14.3.2 Project Planning

Project planning includes traditional activities such as activity planning, effort estimation, and resource allocation, but also more general activities and tasks such as coordination of development activities and prioritization of different implementation alternatives.

There are many planning activities of a more general nature that are important for Inner Source management. For example, Gurbani et al. (2006) emphasized that the core team must have a long-term vision of the evolving shared asset. Since many projects may become dependent on the shared asset and future changes, it is important to be able to foresee the need of different projects and to be able to communicate the project vision.

It is also important to understand that contributing projects may not be as focused on general solutions as the core team must be. It is natural that contributing projects are more focused on the specific development of their project and consider the Inner Source product as only one part of their project. This means that a core team must coordinate the development schedules from different contributing teams, and as much as possible, minimize the risk of duplicate work. This may involve in some cases the prioritization and to some extent negotiation of requirements from different projects. Long-term planning also includes evolution planning and distinguishing general development from customer-specific changes (Gurbani et al. 2006). Gurbani et al. (2010) suggested that this required a “full time project manager.” Product evolution planning can be supported by keeping a list of candidate changes, and trying to find common themes among future changes.

14.3.3 Monitoring and Taking Actions

This part of project management describes how managers can follow up work and progress, and based on that take actions with the objective to correct for deviations between plans and expectations, and actual progress.

As in the previous section, this text focuses on the monitoring carried out by an Inner Source coordinator, i.e., core team, and not on the monitoring carried out by the contributing projects.

One important type of monitoring is that of quality assurance of contributions. An Inner Source coordinator may carry out this quality assurance when code is contributed, but this may prove to be a bottleneck (Gurbani et al. 2006). One suggested benefit of Inner Source is to have "truly independent" peer review as an effect from a large community who have access to the code (Linus's Law).

An Inner Source coordinator can also track features during development and compare to the vision of the evolving code. If there are differences, which in some way means that future contributions will not be accepted, they can be identified as soon as possible.

14.3.4 Human Issues

The fourth and last theme of our framework is that of human issues and is based on some of the observations that have been reported in the Inner Source literature. One aspect of human issues that has been addressed by Gurbani et al. (2006) concerns the fact that different contributing projects may have processes that are different from the way that contributions are managed by the core team. This includes, for example, how contributions are inspected.

An increased transparency of the development process that is necessary to facilitate Inner Source may introduce friction (Gaughan et al. 2009; Melian and Mähring 2008). Developers may experience an increased openness of the process as a "fish-bowl," and an increased pressure on performance. They may also see the openness as a threat to their unique competence and skill set since more people now may work on the same code with the same type of problem. It may require staff to develop new skills with respect to communication and interaction. These aspects may not be a problematic in all Inner Source initiatives, but cognizance of such potential issues may help address such issues before they arise and escalate.

There are also some positive aspects of this nature. Other developers may experience an open environment as very positive and rewarding. Others may see the open environment and the possibility to voluntarily contribute as a means for professional improvement or as a way to demonstrate their expertise, and thereby rewarding. Such rewards may positively affect developers' motivation, a topic that is discussed in more detail in Chap. 10.

14.3.5 Summary

Table 14.3 summarizes some of the tension points related to the four project management themes presented above, that may arise as a result of introducing Inner Source. For example, if there is a focus on applying an available process in conventional development, a difference in Inner Source is that there are now a number of processes in organizational units that must be aligned. In conventional

Table 14.3 Comparison of traditional project management and potential tension points in Inner Source

Element	Traditional project management	Key difference in inner-source
Process management	Enforce common processes Top-level coordination enforcement.	Alignment of different processes may be challenging
Project planning	Predefined, well-organized, use of planning tools (e.g., Gantt charts) to "predict" delivery. Project costs more easily predicted and calculated (Gaughan et al. 2007)	Planning of "chaotic" OSS style project may be unnerving. Costing of Inner Source shared asset is more difficult
Monitoring and taking actions	Standard methods for following up progress of projects. Traditional punish/reward model	Follow up of more high-level attributes like plan for future releases
Human issues	Option to deliver "good enough" code as it is not visible to most people Typically no encouragement of initiatives to contribute (maybe voluntary) to work outside own project Hierarchical business organization (Gaughan et al. 2007)	Contributions or code may be in an "embarrassing" state (Dinkelacker et al. 2002) Conflicts OSS style (ego) strong personalities; "bullying" by technological experts who take ownership Opening up code may face objections from developers who fear losing their job Employees may resist change (Gaughan et al. 2007)

development, there are a number of well-known methods for project planning, while in Inner Source there is a need to plan and synchronize different initiatives from organizational units that will encourage contributions. This also means that monitoring must concern several organizational units, for example, with respect to what each will contribute and how this is aligned to the shared asset as a whole. Finally, a number of human issues need to be considered as well.

14.4 Case Studies

In this section, two industry case studies are presented in order to illustrate project management issues in Inner Source. Organizations that wish to adopt Inner Source need to understand their current project management approach so that they can assess the extent to which this aligns with an Open Source development approach. We conducted a case study at one organization that had a strong interest in adopting Inner Source, to illustrate such an assessment of Inner Source alignment prior to adoption. This is presented in Sect. 14.4.1.

Whereas the first case study presents potential project management tension points that may arise prior to, or during adoption of Inner Source, we also conducted a case study at an organization with an established track record in Inner Source adoption. This second case study sheds light on a number of actual challenges in

Inner Source project management and how those issues have been addressed. This case study is presented in Sect. 14.4.2.

14.4.1 A Case Study of Inner Source Alignment

The first case study was conducted in a multinational software and hardware company, based in Sweden (hereafter referred to as “ToolSoft”). ToolSoft has been using Open Source software in its products and has been involved with large Open Source communities. The goal of the case study was to understand the alignment of ToolSoft’s development practices with the Open Source development practices. Therefore, a first step was to identify a set of software development practices that are typical for Open Source development and compare them to the current development practices in the company. In order to do this, practices typical for Open Source development were identified by analyzing the most important aspects of the Open Source projects hosted under the Apache project. The identified practices were also validated by studying Fogel (2005) on how to run a successful Open Source project. The identified practices are listed in Table 14.4. After the comparison was made, the data was interpreted in the light of the aspects of the framework presented in Sect. 14.3, according to which the results are also presented in this chapter.

Traditional development practices are closely related to conventional software project management (SPM) discussed in more detail in Sect. 14.3. Traditional SPM is analyzed through its main features: process management, project planning, monitoring and taking actions, and human issues. In order to understand how Open Source development practices identified in this case study can be applied within a conventional SPM context, the four features of the traditional SPM are further examined through the framework presented in Table 14.4, which outlines a number of aspects specific to Open Source development.

14.4.1.1 Process Management

The process management aspect of software project management is closely related to the Open Source infrastructure aspect. An infrastructure portal (or “forge”) hosts information about Open Source development practices including a community guide and information on source code repositories, development roadmap, etc. All individual projects developed by a community need to comply with the infrastructure aspects, just like any specific project developed within a more traditional environment needs to comply with process management defined on the organization level.

In this study, we found that ToolSoft’s development practices are mostly aligned with Open Source practices with respect to the infrastructure aspects. However, some of the content hosted under the infrastructure portal was found to be

Table 14.4 Open Source development practices considered in analyzing ToolSoft's software development practices

Aspect	Category	Element
Infrastructure	Product info	Features, documentation, FAQ, news road map, security
	Code access	Download location, binary package, release notes
	Community guide	Community overview, community roles, coding conventions, commit conventions, building and testing, debugging, mailing lists, bugs/issues, releases
Communication	Standardized	Message, channel, norm
Management	Meritocracy	Role, promotion, authority

incomplete and out of date. In an Open Source setting, up-to-date information hosted under the portal is crucial for understanding how a project operates, for example, what the project goals are, current issues and bug tracking, rules of conduct, developers guide, and documentation. In a traditional setting, much of this information is disseminated through different channels, either through interpersonal communication, unarchived electronic communication, or a project's specific documentation. What can happen in practice, is that different projects working on the same parts of software do not synchronize documentation, or a task of completing documentation is not given a sufficient priority, and resulting in outdated documentation.

14.4.1.2 Project Planning

The project planning aspect of traditional software project management deals with identification of activity, effort estimation, and resources allocation. In an Open Source community, an activity or task is identified through a transparent communication process, for example, a community participant identifies a new feature to be implemented or reports a bug through an open online forum or email list. The proposed activity is then further discussed and assessed among community participants. Any community participant can decide to work on a task. Hence, there may be multiple community participants working on the same activity. This is a very different approach to resource allocation from a traditional planning approach, where a resource is assigned to an activity, rather than a resource being able to assign him/her self to the activity. Even though effort estimates are normally provided for activities in an Open Source realm, a time constraint of an activity is not normally enforced in the same sense as in a more traditional closed source environment.

14.4.1.3 Monitoring and Taking Actions

The transparent communication aspect of Open Source development ensures that many individuals look at the way a product is being developed, and thus, the "many eyeballs" effect (Linus's Law) often found in Open Source can be related to the monitoring and taking actions aspect of the traditional software project management. In Open Source development, the monitoring aspect is steered through a transparent process where participants provide feedback on actions taken.

In this case study, we found that the communication characteristics observed in ToolSoft have a high level of misalignment with Open Source practices. As employees were seated closely to each other within the office plan, people tended to favor informal meetings and discussions over electronic communication. This way, the communication process is not transparent, or traceable, and may have to be repeated by different community members. Open source communities use electronic forms of communication that facilitate transparent and traceable discussions. Transparent discussions enable relevant participants and decision makers to get involved, which may help in resolving issue in a timely fashion. Archived discussions can be referenced in order to understand why certain decisions were taken in the past or to find out how similar issues were resolved. Implementing transparent and archived communication archives could improve efficiency in ToolSoft by ensuring that relevant resources are involved on time by creating searchable problem/resolution archives and by decreasing the amount of time spent in repetitive and less efficient ways of communication.

14.4.1.4 Human Issues

The transparent nature of communication in Open Source development helps in understanding how the project functions, the importance of contributors' roles, and thus, ensures that the participant roles are assigned in a meritocratic way. A community guide defines the rules of engagement within the community, especially in terms of online communication norms. Open Source communities have recognized the importance of friendly, standardized, and efficient communication in overall project success (Fogel 2005). Hence, the community guide and open communication aspects are related to the human issues feature of software project management.

Our assessment of the management aspect showed alignment of the defined roles in ToolSoft and Open Source development practices, but in practice the roles in ToolSoft exhibit overlapping characteristics. For example, we found that software architects sometimes took on responsibilities of project leads. Such overlapping and conflicting roles can decrease the overall process and product efficiency. One example of this is sacrificing some aspect of a product's technical maturity to meet a deadline.

14.4.1.5 Conclusion

In this case study, we found that implementing some characteristics of typical Open Source practices while retaining a more traditional software development approach in terms of communication and project management may hinder the potential efficiency of Inner Source. In order to better understand the actual effects of a transition to Inner Source and Open Source practices, more research is needed. This would increase understanding of benefits and drawbacks of Inner Source in an organization, and potentially offer ways to tailor practices to closed development environment with a limited number of resources and projects having a time constraint as a critical attribute.

14.4.2 A Case Study of Project-Specific Inner Source

The second case study was conducted at an organization that we refer to as "GlobalSoft." GlobalSoft is a multinational organization in a regulated domain. GlobalSoft has adopted a number of OSS development practices and augmented their conventional mechanism for project management. Some of this case study's findings were previously reported by Stol et al. (2011); the results presented in this section focus in particular on project management issues, taking into consideration the four themes introduced in Sect. 14.3.

The GlobalSoft organization consists of a number of *business units*. Each business unit is specialized in a specific domain for which they develop products. Each business unit has therefore highly specialized expertise and in-depth knowledge of these technologies.

Figure 14.3 shows a representation of the key elements of the Inner Source initiative at GlobalSoft. The figure shows that a core team develops and releases versions of the shared asset. The core team consists of architects and developers, as well as a support team (similar to what is depicted in Fig. 14.2). The core team can work closely with business units in so-called collaborative development projects, to develop a new (or enrich an existing) component, which is then integrated back into the shared asset. Effectively, these collaborative development projects include a *feature advocate* from the business unit. A *steering committee* defines a roadmap and decides which features are implemented. Business units integrate, and possibly customize the shared asset. As new companies are acquired, they become part of the organization as new business units, and any software that is brought in is scrutinized for potential reuse.

While Inner Source offers many potential benefits (as discussed in Sect. 14.2), it may also introduce new challenges, which are sometimes similar to those associated with using OSS in product development (Stol et al. 2011). The remainder of this section presents a number of challenges related to project management and how they were addressed by the GlobalSoft.

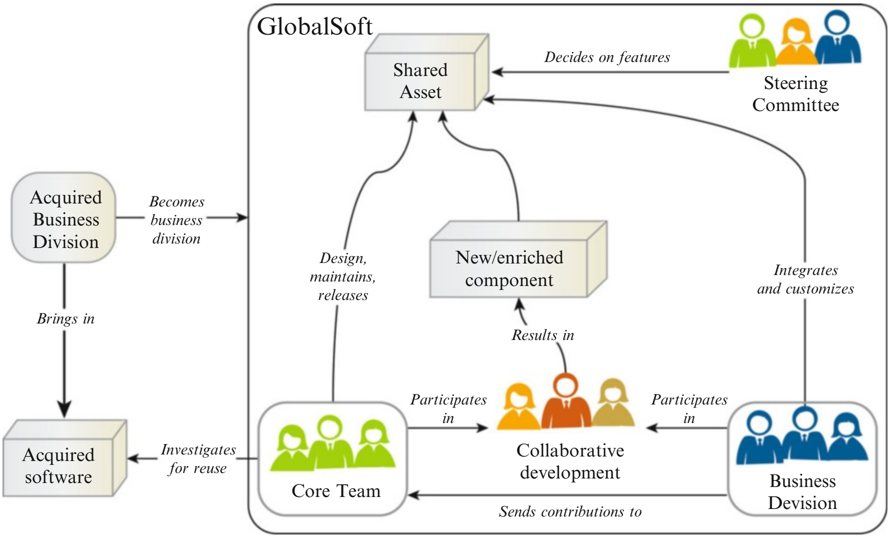


Fig. 14.3 Conceptual model of Inner Source in the GlobalSoft organization (adapted from Stol et al. 2011)

14.4.2.1 Process Management

A key concern in process management was finding agreement on the software development processes to be used. In the last two decades, GlobalSoft acquired many companies in the same domain, which have become business units. Besides each organization's unique culture, an organization may also have a set of established development processes to which they are accustomed.

One challenge that this study revealed was a misalignment of the software development life cycles at the business units and the core team. Business units want to focus exclusively on differentiating, value-adding functionality, and prefer that common functionality shared by different products be implemented by the core team. To that end, business units could send their requirements to the core team, which would then implement a certain module or component. However, the integration tests were performed by the requesting business unit; the core team did not build actual products and as a result did not do any integration tests, and as such did not find any problems in using the new functionality. By the time a business unit was ready to integrate the new component and identify problems (e.g., missing functionality or lack of attention for quality attributes such as performance), the core team had focused their attention to new tasks. As a result, little support in terms of defect fixing could be expected, which was a burden for the business unit that was trying to deliver a product to the market on time.

In order to prevent such problems, GlobalSoft adopted the concept of so-called collaborative development projects. This can be considered a hybrid solution of "pure" OSS style defect fixing by noncore developers, but with close involvement of the core team. In practice, this resulted in temporary, virtual teams that work together on either a new component or to enrich an existing component.

14.4.2.2 Project Planning

As outlined in Sect. 14.3, project planning relates to activities such as activity planning, effort estimation, resource allocation, but also includes feature prioritization. Many of these topics seem nonexistent in OSS projects, since in Open Source developers self-select tasks that they feel they can do, or they feel need to be done. This self-organizing feature of OSS style development is more difficult to combine in commercial software development, where schedules need to be met and budgets need to be respected.

At GlobalSoft, a steering committee developed a roadmap that outlined the future development plans of the shared asset. The steering committee had representatives from the business divisions as well as from the core team. This way, input was received from both the supplier side as well as from the customer side, which ensured that (1) the core team better understood what was needed by the business divisions (i.e., the shared asset's users) and (2) the business divisions could align their own roadmap of future product evolution with the shared asset on which they based their product.

One issue we found in this case study is that the development capacity of the core team was a bottleneck. This was due to the fact that the core team developed a platform that was used by many different business units, each with many feature requests. This bottleneck was in fact one of the motivations to adopt OSS development principles. Business divisions can—at their own cost—request development of certain features, but this "solution" is rather limited, given the restricted capacity of the core team.

14.4.2.3 Monitoring and Taking Actions

The case study organization used commonly used infrastructure offered by Collab. Net.[6] This includes common infrastructure such as a mailing list, a wiki, and an issue tracker, which facilitated knowledge sharing, community interaction, and tracking of issues. As developers encountered difficulties, the mailing list provided a means to ask questions and share knowledge. Architects of the core team monitor these lists regularly and provide feedback where possible.

As outlined above, the organization had grown significantly over time, as companies were acquired that became new business units. As a result of this, the scope of the shared asset (the product line platform used as a foundation for most products) was evolving. Instead of a static scope (with a fixed set of features and use

[6] www.collab.net

cases), the shared asset's scope was dynamic. As the new business units became new customers of the shared asset, new use cases had to be considered, so that the new business unit could also benefit from the platform. This could either be done by porting the existing product to GlobalSoft's platform or to rebuild the new business unit's software using the platform as a foundation.

14.4.2.4 Human Issues

There are a number of human issues to consider in Inner Source project management. A key challenge in building an internal, organization-wide and interactive community is getting developers to contribute. Without active members who answer questions of other people in the community, an Inner Source initiative may not become very successful.

A key factor that should be considered is building an environment in which all parties involved become supportive of the Inner Source program (Stol et al. 2014). In particular, incentives should be clear to each development group so that any potential tension between the core team (i.e., supplier) and the business divisions (i.e., customers) is prevented. Business divisions need to see clear benefits from actively participating in the community in order to prevent a "them versus us" atmosphere. This involves clearly communicating—and demonstrating—potential benefits. Some business divisions were more receptive of GlobalSoft's Inner Source initiative than others.

Similar to what can be observed in Open Source communities, we found that typically there was an evolution—or a "learning curve"—in involvement in GlobalSoft's Inner Source community. Usually this started with using some components of the product line's platform (similar to an OSS product's users). As more parts of a product's application were migrated to the new platform (i.e., the shared asset at GlobalSoft), a business unit would increase its interaction with the core team. At some point, a business unit could decide to use components that are in development (rather than a released snapshot version) to benefit from the latest available functionality, and a business unit could want to contribute certain functionality that they required (comparable to OSS users (or "lurkers," even) who become contributors). In GlobalSoft, this process worked better for some business units than for others. In particular, business divisions that actively followed the core team closely in their development had significantly fewer problems than those who treated the core team as a traditional "black box" software supplier.

As outlined above, GlobalSoft acquired a large number of other companies over time, each of which brought in their own software and systems. In many cases, these systems would have a different architecture than GlobalSoft's product line. One of our findings is that GlobalSoft took a conscious approach to sit together with the people who designed and developed the newly acquired systems. It was felt important to come to an understanding about the design rationale for the different systems, and to find agreement on how a new business division's software could make use of the shared asset. Respect for the acquired company's culture is important in this context, rather than prescribing and requiring conformance to GlobalSoft's existing architecture. It was felt that a close collaboration, understanding of each other's software assets, and identifying how to let the different systems grow toward each other would yield much better results (in both the technical aspect as well as in "goodwill") than enforcing GlobalSoft's architecture.

14.5 Discussion and Future Work

This section continues the discussion on differences between conventional development and Inner Source development. In particular, the objective is to discuss how introducing Inner Source affects software project management. It is important to understand that each Inner Source initiative must be tailored to the context of an organization so as to consider the various constraints an organization may have to address (e.g., regulated environments) as well as the organizational culture.

The decision to introduce a strategy such as Inner Source is always based on a trade-off. On one hand, there are a number of perceived advantages such as increased reuse and transparent communication. However, challenges may arise as well. A key issue is to recognize that the different organizational units that work with an Inner source core team are likely to use different development processes, which can result in significant coordination challenges. The case study of GlobalSoft illustrated how such process-misalignment manifested. To overcome such issues, new coordination mechanisms can be required, or existing organizational governance mechanisms can be adjusted so as to better plan development of a shared asset, and facilitate synchronization of development processes—in the case of GlobalSoft, a new collaborative development mechanism emerged.

Challenges may also arise on the operational level for developers, who work in the various organizational units and who contribute to development of a shared asset. An increased transparency of the process in general and the visibility of the contributed code may also result in tension points. For example, whereas code contributions used to be limited in visibility to a developer's project team members, all project artifacts can now be scrutinized by an organization's global developer community. Some people may also see the electronic communication that is typically used in an Open Source environment as over-formalized. This would also result in a fully transparent (and archived) communication throughout an organization. This may mean that some people would prefer to have informal meetings instead of using, for example, mailing lists.

One strategy to address several of the challenges is to start with a code asset that is already used by several projects, and then gradually introducing the concept for other and new assets. We argue that it is an advantage to start with a part of the code and the organization where the change is seen as positive, that is, in line with creating a "short-term win" (Kotter 1996).

The key observations from the case studies are summarized in Table 14.5. The results in the table suggest a few steps to take in transforming from a conventional

Table 14.5 Key findings of the two case studies

Theme	ToolSoft	GlobalSoft
Process management	Project portal that facilitates "self-management" may exist but may not be used fully	– Collaborative development projects to overcome process misalignment
Project management	Higher degree of engagement of all project participants through transparent communication process to deliver high-quality software products up to users' specifications can be expected	– Steering committee is useful to gather organization-wide input and synchronize efforts – Extra "purchasing" of critically needed software development possible but limited to capacity of core team
Monitoring and taking actions	Establish transparent and archived communication to identify and resolve issues more efficiently	– Dynamic scope of shared asset due to new required use cases
Human issues	Facilitate friendly communication atmosphere and to promote based on merit to build a healthy organization culture needed	– Need clear incentives so as to engage people – Learning curve may be steep due to novelty of approach – Respect for developers and organizations' cultures is key to successful collaboration

development approach to Inner Source. For example, an infrastructure for managing information needs to be used more extensively, and process synchronization mechanisms such as collaborative development projects can be adopted to overcome process misalignment issues.

How to carry out the introduction of Inner Source will vary across organizations and heavily depend on contextual factors. We argue that it is important to identify what gains are considered important and focus on those one at a time. It is also important to identify what challenges may arise and to prepare mitigating actions to overcome them.

Concerning process management, an important part of Inner Source management is to align different processes in different organizational units. This is also reflected in the importance of monitoring of different development initiatives in different organizational units. Conventional project management is still taking place in contributing organizational units, while an important part of Inner Source management concerns coordination of initiatives. All management levels in an organization should also be aware of the potential human issues that can affect the introduction of Inner Source. That is, introducing Inner Source will mean a number of changes at the tactical and operational levels when it comes to synchronizing development activities in different business units in the organization. This is also a basis for increasing reuse between business units, which by many is seen as an important goal of introducing Inner Source.

14.5.1 Future Work

As mentioned in this chapter, Inner Source is an emerging approach to software development. Although the first studies on this topic were published in the early 2000s, the field is still in its nascent phase, and more research is necessary to better understand how benefits (outlined in Sect. 14.2) can be achieved. We conclude this chapter by outlining a number of directions for future work.

- While different Inner Source adoption models exist (Gurbani et al. 2010), further studies of how organizations embrace Open Source development practices will be a welcome addition to the literature.
- While there exist a few reports of how Inner Source supports the development of a software product line (Van der Linden 2009), there are a few studies of reuse of Inner Source components. Such studies do exist in an Open Source context (e.g., Capiluppi et al. 2011), which can be used as a template to design studies of reuse of Inner Source components.
- In this chapter, Inner Source has been compared to traditional project management aspects. Further research can include comparison of Inner source and management in Agile projects.
- Quantitative studies to better understand how certain benefits can be achieved. Such studies typically identify dependent and independent variables, and can identify the relationship between those variables so as to be able to "predict" how a certain benefit can be achieved.

Acknowledgments The authors wish to thank Brian Fitzgerald for useful feedback on an earlier draft of this chapter. This work was conducted within the ITEA2-SCALARE project, supported by Vinnova and Enterprise Ireland, and further by Science Foundation Ireland grant 10/CE/I1855 to Lero — the Irish Software Engineering Research Centre (http://www.lero.ie), and by the Industrial Excellence Center EASE—Embedded Applications Software Engineering (http://ease.cs.lth.se).

References

Abran A, Moore JW (2004) Guide to the software engineering body of knowledge. IEEE

Ågerfalk PJ, Fitzgerald B (2008) Outsourcing to an unknown workforce: exploring open-sourcing as a global sourcing strategy. MISQ 32(2):385–409

Capiluppi A, Stol K, Boldyreff C (2011) Software reuse in open source: a case study. Int J Open Source Softw Process 3(3):10–35

Capiluppi A, Stol K, Boldyreff C (2012) Exploring the Role of Commercial Stakeholders in Open Source Software Evolution. In: Hammouda I et al (eds) OSS 2012, IFIP AICT 378, pp 178–200

Dinkelacker J, Garg PK, Miller R, Nelson D (2002) Progressive open source, 24th international conference on software engineering (ICSE), Orlando, FL, pp 177–184

Duncan WR (2013) A guide to the project management body of knowledge (PMBOK®guide), 5th edn. Project Management Institute (PMI), Newtown Square

Fogel K (2005) Producing open source software: how to run a successful free software project. O'Reilly Media, Sebastopol

Gaughan G, Fitzgerald B, Morgan L, Shaikh M (2007) An examination of the use of inner source in multinational corporations: a preliminary framework to understand inner source software development. In: Proceedings 1st OPAALS conference, pp 48–60

Gaughan G, Fitzgerald B, Shaikh M (2009) An examination of the use of Open Source software processes as a global software development solution for commercial software engineering. In: 35th Euromicro conference on software engineering advanced applications (SEAA), pp 20–27

Gurbani VK, Garvert A, Herbsleb JD (2006) A case study of a corporate open source development model. In: 28th international conference on software engineering, pp 472–481

Gurbani VK, Garvert A, Herbsleb JD (2010) Managing a corporate open source software asset. Commun ACM 53(2):155–159

Hughes B, Cotterell M (2009) Software project management. McGraw-Hill, New Delhi

Ierusalimschy R (2008) Lua Mailing List, reply of Roberto Ierusalimschy, one of the developers of Lua, Friday, 27 June. http://lua-users.org/lists/lua-l/2008-06/msg00407.html

Kotter J (1996) Leading change. Harvard Business Review Press, Boston

Lindman J, Rossi M, Marttiin P (2008) Applying open source development practices inside a company. In: Russo B, Damiani E, Hissam S, Lundell B, Succi G (eds) Open source development, communities and quality. Springer, New York

Lindman J, Riepula M, Rossi M, Marttiin P (2013) Open source technology in intra-organisational software development–private markets or local libraries. In: Ericsson Lundstrom J, Wiberg M, Hrastinski S, Edenius M, Ågerfalk PJ (eds) Managing open innovation technologies. Springer, Berlin

Melian C, Mähring M (2008) Lost and gained in translation: adoption of open source software development at Hewlett-Packard. In: Russo B, Damiani E, Hissam S, Lundell B, Succi G (eds) Open source development, communities and quality. Springer, New York

Morgan L, Feller J, Finnegan P (2011) Exploring inner source as a form of intra-organisational open innovation. In: Proceedings European conference on information systems

Oručević-Alagić A, Höst M (2010) A case study on the transformation from proprietary to open source software. In: Boldyreff C, González-Barahona JM, Madey GR, Noll J, Ågerfalk PJ (eds) Open source software: new horizons. Springer, Boston

Riehle D, Ellenberger J, Menahem T, Mikhailovski B, Natchetoi Y, Naveh B, Odenwald T (2009) Open collaboration within corporations using software forges. IEEE Softw 26(2):52–58

Stol K, Babar MA (2010) Challenges in using open source software in product development: a review of the literature. 3rd workshop on emerging trends in FLOSS research and development, co-located with international conference on software engineering, pp 17–22

Stol K, Babar MA, Avgeriou P, Fitzgerald B (2011) A comparative study of challenges in integrating open source software and inner source software. Inf Softw Technol 53(12):1319–1336

Stol K, Avgeriou P, Babar MA, Lucas Y, Fitzgerald B (2014) Key factors for adopting inner source. ACM Trans Softw Eng Methodol 23(2)

Van der Linden F (2009) Applying open source software principles in product lines. Upgrade 10 (2):32–41

Van der Linden F, Lundell B, Marttiin P (2009) Commodification of industrial software: the case for open source. IEEE Softw 26(4):77–83

Vitharana P, King J, Chapman HS (2010) Impact of internal open source development on reuse: participatory reuse in action. J Manage Inf Syst 27(2):277–304

Wesselius J (2008) The bazaar inside the cathedral: business models for internal markets. IEEE Softw 25(3):60–66

Biography Martin Höst is a Professor in Software Engineering at Lund University, Sweden. He received an M.Sc. degree from Lund University in 1992 and a Ph.D. degree in Software Engineering from the same university in 1999. His main research interests include software process improvement, software quality, risk analysis, and empirical software engineering.

Klaas-Jan Stol is a researcher with Lero—the Irish Software Engineering Research Centre. He holds a PhD in software engineering from the University of Limerick. His research interests include contemporary software development methods and strategies, including Inner Source, Open Source, crowdsourcing, and agile and lean methods, as well as research methodology and theory building in software engineering. In a previous role, he was a contributor to an Open Source project.

Alma Oručević-Alagić is a Ph.D. student at Lund University, Sweden. She received an M.Sc. degree in Software Engineering from the University of St. Thomas, St. Paul, Minnesota in 2002, and a Technical Licentiate degree in 2013 from Lund University. Her research interests include Open Source, Inner Source, and Network Analysis of software development communities.

Part IV
Emerging Techniques

Introduction

It is not only the software development practices that have changed, as discussed in Part III. New techniques have emerged that can be used in project management. Thus, in this part of the book, we have invited leading experts on different techniques useful for the project manager. The techniques are presented in a software project management context. The new techniques presented are search-based techniques, social media, software process simulation and efficient use of data. The authors share their knowledge, insights and accompanying recommendations and conclusions in four chapters in this part of the book.

In Chap. 15, Filomena Ferrucci, Mark Harman and Federica Sarro provide an overview of search-based software project management, where search-based techniques are applied to the area of software project management. The authors discuss the use of these techniques to address some of the subareas within project management such as staffing, scheduling, risk, overtime and effort estimation. The chapter starts with a general introduction to search-based software engineering before moving onto its application in software project management. A historical overview of the application of search-based techniques in some of the different subareas of software project management is provided. The authors conclude the chapter by outlining some future work in the area to further enhance software project management with the use of search-based techniques.

Rachel Harrison and Varsha Veerappa share their experiences in using social media in software project management in Chap. 16. As social media has changed the way we interact in general, it has also provided new opportunities in software projects. The authors elaborate on the implications of social media on software projects and their management. They discuss interactions and social aspects in software projects and then move onto the importance of social media in the projects. The authors have conducted a pilot study regarding the use of social media in nine companies, and they share their observations and experiences. The study is conducted in the context of global software development, and hence the authors compare the findings from co-located and distributed teams. The chapter presents some implications of the use of social media in relation to different contexts and roles, and it ends with an outlook regarding the future of social media in software projects.

In Chap. 17, Dietmar Pfahl provides a discussion of software process simulation. He highlights the perceived potential in using simulation and discusses some of the challenges in relation to its application in industrial settings. The chapter starts with a brief historical perspective and discusses the purpose and scope of software process simulation. He continues by introducing an example for illustration purposes. The example is then used in several scenarios to show the possible application of software process simulation in relation to software project management. Pfahl moves onto discussing the uptake of software process simulation by the software industry. He notes that it is quite low and stresses three main reasons for it: (1) high cost for developing a simulation model, (2) high cost for evolving the simulation model, and (3) difficulty in demonstrating the value of a simulation model. Each of these three reasons is discussed in the chapter. The chapter concludes with a presentation of five issues that need to be addressed to increase the use of simulation as a tool in software development.

Tim Menzies describes model-based software project management in Chap. 18. He stresses the need for the use of empirical data and models to support project managers in their decision-making. Menzies highlights that simplicity is needed. Software project managers need support, and in this chapter it is discussed how it is possible to identify the least a manager needs to know to improve their projects. The chapter starts by reviewing some work in relation to what the author refers to as intelligent project management. It continues by explaining why speculative project management is difficult and hence explains why support-based project management is needed. It then describes support-based project management using data mining and discusses some observations in relation to it. The chapter continues with a discussion on spectral learning, which is illustrated and discussed by using data from several case studies. A discussion on the usefulness of the ideas presented concludes the chapter.

The four chapters in this part provide an in-depth insight into some of the new techniques that may be useful for software project managers. The chapters highlight some of the new techniques that a project manager can benefit from using in their daily work. The chapters in this part highlight some of the new techniques and hence challenges and opportunities that software project managers must be able to address in their daily work.

Chapter 15
Search-Based Software Project Management

Filomena Ferrucci, Mark Harman, and Federica Sarro

Abstract Project management presents the manager with a complex set of related optimisation problems. Decisions made can more profoundly affect the outcome of a project than any other activity. In the chapter, we provide an overview of Search-Based Software Project Management, in which search-based software engineering (SBSE) is applied to problems in software project management. We show how SBSE has been used to attack the problems of staffing, scheduling, risk, and effort estimation. SBSE can help to solve the optimisation problems the manager faces, but it can also yield insight. SBSE therefore provides both decision making and decision support. We provide a comprehensive survey of search-based software project management and give directions for the development of this subfield of SBSE.

15.1 Introduction

Software Project Management includes several activities critical for the success of a project (e.g., cost estimation, project planning, quality management). These activities often involve finding a suitable balance between competing and potentially conflicting goals. For example, planning a project schedule requires to minimise the project duration and the project cost, and to maximise the product quality. Many of these problems are essentially optimisation questions characterised by competing

F. Ferrucci
DISTRA, University of Salerno, Salerno, Italy
e-mail: fferrucci@unisa.it

M. Harman • F. Sarro (✉)
CREST, Department of Computer Science, University College London, London, UK
e-mail: mark.harman@ucl.ac.uk; f.sarro@ucl.ac.uk

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_15, © Springer-Verlag Berlin Heidelberg 2014

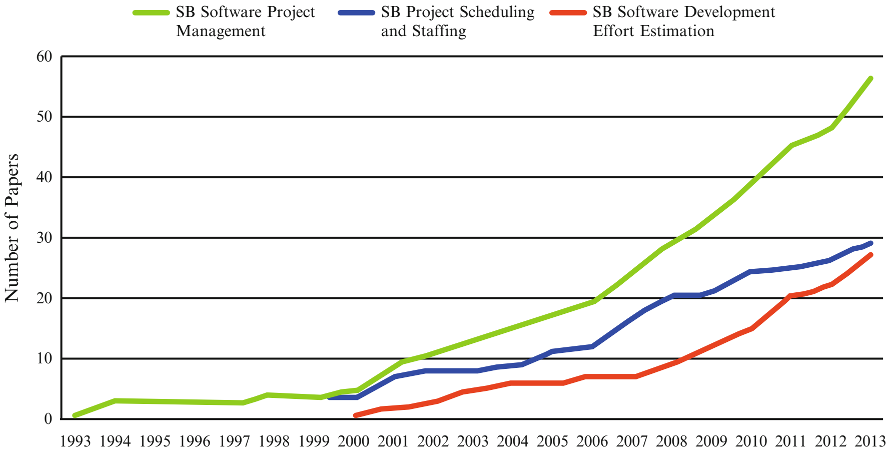


Fig. 15.1 Number of relevant publications on the use of search-based approaches for software project management from 1993 to 2013 [*source* Zhang (2013)]

goals/constraints and with a bewilderingly large set of possible choices. So finding good solutions can be hard.

Search-based software engineering seeks to reformulate software engineering problems as search-based optimisation problems and applies a variety of meta-heuristics based on local and global search to solve them (such as Hill Climbing, Tabu Search, and Genetic Algorithms). These meta-heuristics search for a suitable solution in a typically large input space guided by a fitness function that expresses the goals and leads the exploration into potentially promising areas of the search space.

Though the term Search-Based Software Engineering (SBSE) was coined by Harman and Jones in 2001 to cover the application of computational search and optimisation across the wide spectrum of software engineering activities (Harman and Jones 2001), there were already pockets of activity on several specific software engineering problems prior to the introduction of the term SBSE. One such topic was search-based software project management, the topic of this chapter. In particular, there was work on search-based project scheduling and staffing by Chang (1994), Chang et al. (1994, 1998) and Chao et al. (1993), and on search-based software development effort estimation by Dolado (2001) and Shukla (2000). Figure 15.1 shows the number of papers published on the use of search-based approaches for Software Project Management. We can note that the first work aiming at optimising project scheduling and staffing appeared in 1993, while in 2000 the SBSE community started investigating search-based approaches also for software development effort estimation.

This chapter provides a comprehensive review of techniques, results and trends published in relevant papers. We discuss the effectiveness of search-based approaches for supporting project managers in many activities and provide suggestions for future research directions.

The rest of the chapter is organised as follows. Section 15.2 reports on the main features of the most popular search-based techniques used in the context of search-based software project management. Section 15.3 introduces the problems of scheduling and staffing and building predictive models with special focus on software development effort estimation providing a description of search-based approaches proposed in the literature and the empirical studies carried out to assess their effectiveness. Future research directions are described in Sect. 15.4. Section 15.5 concludes the chapter.

15.2 Search-Based Software Engineering

Software engineering, like other engineering disciplines, is concerned with optimisation problems: we seek to build systems that are better, faster, cheaper, more reliable, flexible, scalable, responsive, adaptive, maintainable, and testable; the list of objectives for the software engineer is a long and diverse one, reflecting the breadth and diversity of applications to which software is put. The space of possible choices is enormous and the objectives many and varied. Search-based software engineering (SBSE) is an approach to software engineering in which search-based optimisation algorithms are used to identify optimal or near optimal solutions and to get insight. Thus, in SBSE an SE problem (e.g., test case generation) is treated as a search or optimization problem whose goal is to find the most appropriate solution conforming to some adequacy criteria (e.g., maximising the code coverage). Rather than constructing test cases, project schedules, requirements sets, designs, and other software engineering artifacts, SBSE simply searches for them.

The search space is the space of all possible candidate solutions. This is typically enormous, making it impossible to enumerate all solutions. Moving from conventional software engineering to SBSE basically requires choosing a representation of the problem and defining a suitable fitness function to determine how good a solution is. Typically, a software engineer will have a suitable representation for his/her problem, because one cannot do much engineering without a way to represent the problem in hand. Furthermore, many problems in software engineering have a rich and varied set of software metrics associated with them that naturally form good initial candidates for fitness functions (Harman and Clark 2004).

With these two ingredients, it becomes possible to implement search-based optimisation algorithms. These algorithms use different approaches to locate optimal or near-optimal solutions. However, they are all essentially a search through many possible candidate instances of the representation, guided by the fitness function, which allows the algorithm to compare candidate solutions according to their effectiveness at solving the problem in hand. Many techniques have been used, including local search techniques, such as Hill Climbing (HC), and global techniques, such as Genetic Algorithm (GA) and Genetic Programming (GP).

Despite the local maximum problem, HC is a simple technique that is both easy to implement and surprisingly effective (Harman et al. 2002; Mitchell and Mancoridis 2002); these aspects make it a popular first choice among search-based techniques. GA belongs to the larger class of evolutionary algorithms (EAs) (Holland 1975), which loosely model evolutionary searches for fit individuals. GP (Koza 1992) is another form of evolutionary technique that has proved very useful in SBSE for project management.

A comprehensive review of the overall field of SBSE can be found in the work of Harman et al. (2012b). In that overall SBSE survey, the reader can find a more detailed explanation of the algorithms used in SBSE. There is also a tutorial on SBSE (Harman et al. 2010), in which the reader can find a gentle introduction to the entire area. The SBSE survey (Harman et al. 2012b) and tutorial (Harman et al. 2010) cover the whole area of SBSE and, as a result, has little time and space available for each subtopic. The present survey focuses on the results, trends, techniques, and achievements in SBSE for project management. Though there have been many surveys on other subareas of SBSE, including, testing (McMinn 2004; Yoo and Harman 2012), design (Räihä 2010), and requirements (Zhang et al. 2008), there has been no previous survey on SBSE for project management.

15.3 Search-Based Software Project Management

In this section, we provide an overview of the work on search-based software project management, in which SBSE is applied to support project managers for time management (Sect. 1.3.3), cost management (Sect. 1.3.4), quality management (Sect. 1.3.5), human-resource management (Sect. 1.3.6) and risk management (Sect. 1.3.8).

The first application of SBSE to software project management has been proposed for project scheduling and resource allocation. Figure 15.2 provides a generic schematic overview of SBSE approaches to project planning. Given in input information about work packages (e.g., cost, duration, dependencies) and staff skills, the search-based approaches search for an optimal work package ordering and staff allocation guided by a single or multi-objectives fitness function. A natural goal for a search-based approach to project management is to find project plans that minimise the completion time of the project. Another goal that has been taken into account is to minimise the risks associated with the development process (e.g., delays in the project completion time, or reduced budgets available).

Figure 15.2 also highlights one of the important limitations of the approach: it relies on simulation of the likely course of the project, given the guiding project configuration parameters (effectively, the search space). Moreover, the formulation of search-based project management problem does not always model the reality of software projects (e.g., many papers do not have a realistic representation of skill and/or risk). However, there is evidence to suggest that this uptake in making the formulation of the search-based project management problem more realistic

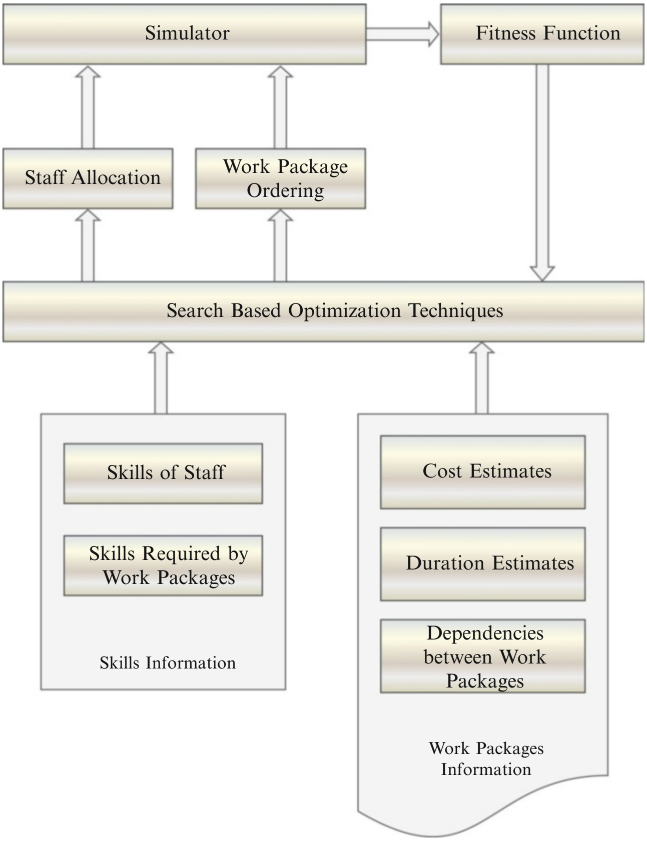

Simulator
Fitness Function
Staff Allocation
Work Package Ordering
Search Based Optimization Techniques
Skills of Staff
Skills Required by Work Packages
Skills Information
Cost Estimates
Duration Estimates
Dependencies between Work Packages
Work Packages Information

Fig. 15.2 A generic search-based project management scheme (Harman et al. 2012b)

[see e.g., Antoniol et al. (2004, 2005), Luna et al. (2012)] and oriented towards human aspects (see Chap. 4) is already taking place.

SBSE techniques also have a natural application in predictive modelling (Harman 2010a). Software development effort estimation is one of the areas in which this search-based predictive modelling approach has been most widely investigated (Ferrucci et al. 2010d). The use of search-based approaches in this context has been twofold: they can be exploited to build effort estimation models or to enhance the use of other effort estimation techniques (Sarro 2011). In the first case, the problem of building an estimation model is reformulated as an optimisation problem where the search-based method builds many possible models—exploiting past projects data—and tries to identify the best one, that is, the one providing the most accurate estimates. In the second case, search-based methods can be exploited in combination with other estimation techniques to improve critical steps of their application (e.g., features subset selection or the identification of critical parameters) aiming to obtain better estimates.

Many empirical studies were carried out in this field showing that search-based techniques are not only as effective as widely used effort estimation methods [see, e.g., Ferrucci et al. (2010a)] but also their use can significantly improve the accuracy of other data-driven effort estimation techniques [see, e.g., Corazza et al. (2013)]. Moreover, there is evidence that the use of search-based approaches can help to yield insight into open problems, such as the choice of a reliable measure to compare different estimation models [see, e.g., Ferrucci et al. (2010c), Lokan (2005)]. Furthermore, search-based approaches only have been used to obtain exact prediction (i.e., one point estimate for a project); however, they can be exploited to investigate prediction uncertainty and risk of inaccurate prediction by means of using sensitivity analysis or multi-objective optimisation, as successfully done in other fields of SBSE [see e.g., Harman et al. (2009)].

Figure 15.3 shows the key ideas developed so far for search-based project management. In Sects. 15.3.1, 15.3.2, 15.3.3 and 15.3.4, we discuss the studies that have been carried out on the use of search-based approaches for project planning and staffing, while in Sect. 15.3.5 we discuss the main studies on search-based effort estimation. Open challenges and future work are reported in Sect. 15.4.

SBSE has also been used to build predictive models to support project managers in other estimation tasks, namely, quality prediction and defect prediction. In particular, Azar (2010) considers approaches to improve predictive models of software quality using SBSE. Liu and Khoshgoftaar (2001) apply GP to quality prediction, presenting two case studies of the approach. This GP approach has been extended, refined and further explored in (Khoshgoftaar et al. 2003; Liu and Khoshgoftaar 2003, 2004; Khoshgoftaar and Liu 2007). In all these works, GP-evolved predictors are used as the basis for decision support. Bouktif et al. (2002, 2004, 2006) exploit GA and SA for quality prediction for software projects. Other authors have used a combination of GA and GP techniques for estimation as a decision support tool for software managers. Jarillo et al. (2011) and Afzal et al. (2014) apply GA and GP for predicting the number of defects and estimating the reliability of the system. Others exploit GA to search for a suitable configuration of support vector machines to be used for inter-release fault prediction (Di Martino et al. 2011; Sarro et al. 2012a).

15.3.1 Early Work on Search-Based Software Project Planning and Staffing

Chang et al. (Chang 1994; Chang et al. 1994, 1998, 2001; Chao et al. 1993) introduced the software project management net (SPM-Net) approach for project scheduling (Sect. 1.3.3) and resource allocation (Sect. 1.3.6). This was the first work on search-based software project management in the literature. The work is evaluated on simulated data, constructed synthetically to mimic the properties of

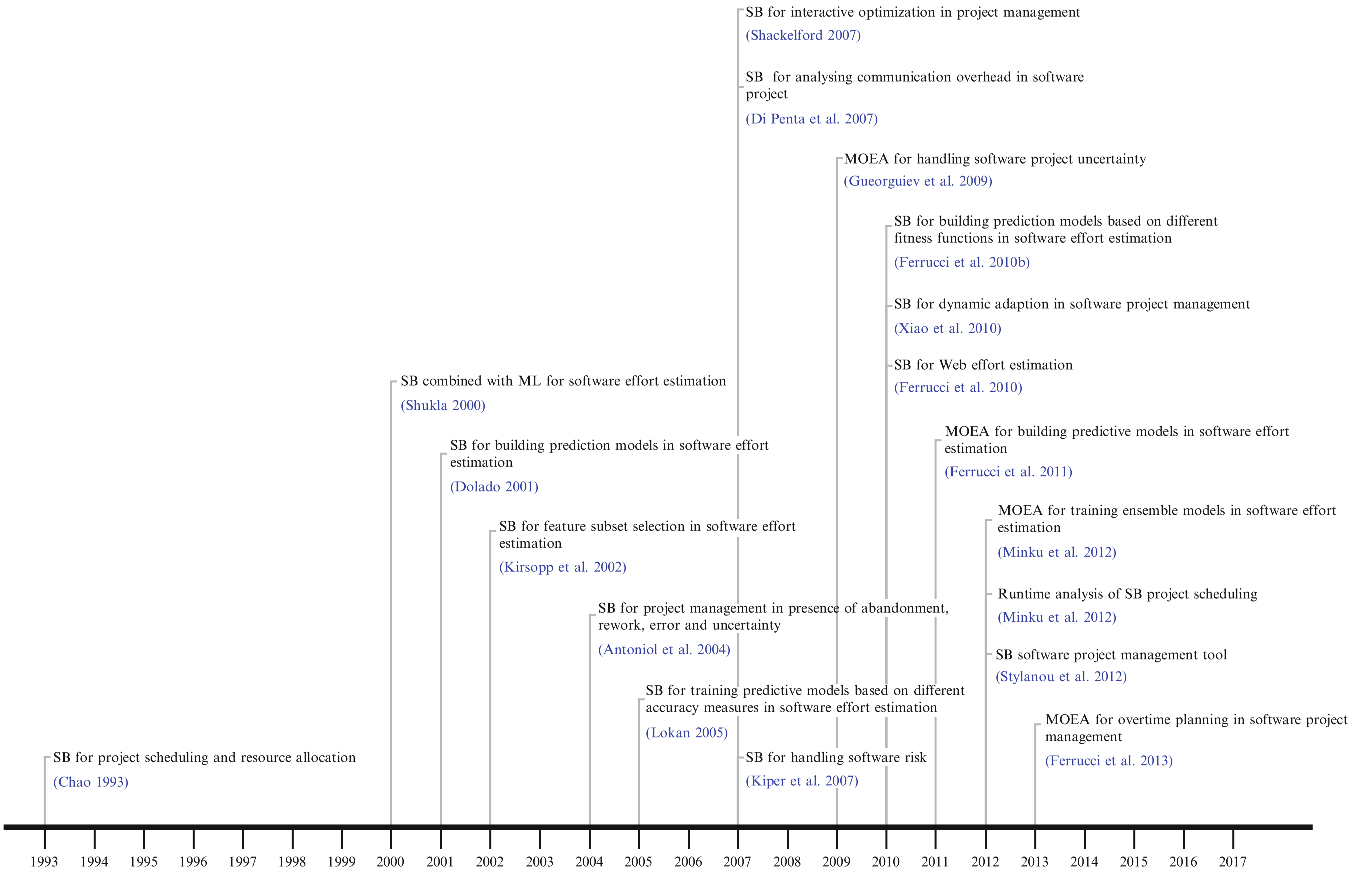

SB for project scheduling and resource allocation
(Chao 1993)
SB combined with ML for software effort estimation
(Shukla 2000)
SB for building prediction models in software effort estimation
(Dolado 2001)
SB for feature subset selection in software effort estimation
(Kirsopp et al. 2002)
SB for project management in presence of abandonment, rework, error and uncertainty
(Antoniol et al. 2004)
SB for training predictive models based on different accuracy measures in software effort estimation
(Lokan 2005)
SB for interactive optimization in project management
(Shackelford 2007)
SB for analysing communication overhead in software project
(Di Penta et al. 2007)
SB for handling software risk
(Kiper et al. 2007)
MOEA for handling software project uncertainty
(Gueorguiev et al. 2009)
SB for building prediction models based on different fitness functions in software effort estimation
(Ferrucci et al. 2010b)
SB for dynamic adaption in software project management
(Xiao et al. 2010)
SB for Web effort estimation
(Ferrucci et al. 2010)
MOEA for building predictive models in software effort estimation
(Ferrucci et al. 2011)
MOEA for training ensemble models in software effort estimation
(Minku et al. 2012)
Runtime analysis of SB project scheduling
(Minku et al. 2012)
SB software project management tool
(Stylanou et al. 2012)
MOEA for overtime planning in software project management
(Ferrucci et al. 2013)
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017

Fig. 15.3 Key ideas developed for search-based software project management

real software projects and to evaluate the properties of the algorithms used. One of the enduring problems researchers concerned with project management face is the lack of available real-world project data. It remains a common problem to this day.

At about the same time as the term SBSE was introduced to the mainstream software engineering community, Aguilar-Ruiz et al. (2001, 2002) were also experimenting with computational search as a means of managing and investigating software project management activities. The goal was to provide rules to the manager to help guide the process of project management. As with the work of Chang et al., a simulation of the project was used to evaluate the search.

This concept of a software project simulation has remained prevalent throughout the history of work on project management (Chap. 17). To evaluate a fitness for a proposed project plan, it is necessary to run a simulation of the course of the project in order to obtain an assessment of the fitness of the proposed plan. Of course, this raises additional issues as to the validity of the simulation. One of the advantages of the SBSE approach, more generally, is that it can operate directly on the engineering material (i.e., the software) in question. This is an aspect of SBSE that is unique to the software engineering domain and not shared by any other application of computational search to other engineering disciplines (Harman 2010b). However, for search-based project management, the familiar issues that arise in computational search for other engineering disciplines arise here also for software engineering (Harman 2010a). We have to be aware that our model of reality and our simulation of that model are both important players in the overall computation of fitness and thereby impact the results obtained; errors in the model or the simulation may feed through into poor quality solutions found by the search.

15.3.2 Minimising Software Project Completion Times

A natural step for a search-based approach to project management is to focus on techniques that find project plans that minimise the completion time of the project (Sect. 1.3.3). Like all project managers, software project managers are concerned with timely product delivery. In highly competitive software engineering application domains, time to market can be a key determinant of the ultimate success of a product.

Antoniol et al. (2004, 2005) applied GAs, HC and SA to the problem of staff allocation to work packages with the aim of reducing project completion time.

At the same time, Alba and Chicano (2005, 2007) also applied search algorithms to software projects. They combined several different objectives for optimisation of project management into a single weighted sum and optimise for this weighted sum. The approach was evaluated on a set of problems generated by an instance generator. Subsequently, this work was extended to handle multiple objectives using a Pareto optimisation approach (Chicano et al. 2011).

The work of Antoniol et al. (2004, 2005) targeted a massive maintenance project, in which work packages were compressible by the allocation of additional staff. This principle of compressible work packages runs contrary to Brooks' famous law (Brooks 1975). That is, adding more staff to a late project simply makes the project even later. Following Brooks' law, we may not, in general, assume that a work package of two person-months will take one month to complete should we choose to allocate two people to it. In the most extreme case, the additional communication overheads may mean that the work package takes longer to complete with two people than with one.

However, in some cases, it is realistic to assume that the duration of a work package can be derived by dividing the person-months needed for the package by the number of engineers working on it. This can only be applied within reason, even where the linearity can be assumed to hold; a two person-month work package may not be completed in under an hour by allocating one thousand engineers to it! However, linearity may apply for a reasonably useful range of values, where the tasks are highly mechanised or where they are specified in detail and prescriptive. Such work packages may be found in such massive maintenance tasks such as those studied in Antoniol et al. (2004, 2005). They may also be found in situations where software engineering activities are outsourced and therefore more highly specified for this reason.

Many authors have simplified their models of software projects by implicitly or explicitly assuming linearity (effectively denying Brooks' law). Where the linearity assumption cannot be justified and we assume that Brooks' law applies, we can still use SBSE; we simply require a richer model. We can also use SBSE to explore the impact of Brooks' law on the software project planning process. Antoniol et al. (2005) introduced an approach to investigating Brooks' law with different models of communication overhead to explore the influence of nonlinearity on project planning.

More intricate models may be required to adequately capture the true behaviour of the project once it commences. Another example of an important aspect of software engineering projects is the tendency for aspects of the project to be reworked. Software is so flexible that it is often considered easy to reassign or reimplement a component. This can lead to headaches for the project manager, who would prefer, perhaps, to schedule his or her project on the basis of known completion times.

To make the formulation of the search-based project management problem more realistic, Antoniol et al. (2004) introduced models of the project management problem that account for reworking and abandonment of work packages. In this way, we can enrich our models of the eventual software project process to cater for more real-world assumptions. This may make the overall model more realistic and the simulation, thereby, more reliable. Unfortunately, it also makes the model more complex and consequently it becomes harder to explain the outcomes to the users. Care is thus required that the model does not become such a Byzantine work of intricate beauty that its findings become simultaneously impenetrable to the decision maker; this area of SBSE is primarily concerned with decision support, rather than decision making (Harman et al. 2012b). Insights that accrue to the decision-maker rely critically on accessibility of explanations.

Much of the previous work on search-based project management (Aguilar-Ruiz et al. 2001; Alba and Chicano 2005, 2007; Chang 1994; Chang et al. 1994, 1998, 2001; Chao et al. 1993; Minku et al. 2012, 2013) has used synthetic data. This can be achieved in a disciplined and controlled manner. For example, Alba and Chicano (2007) used a systematic instance generator to create synthetic software project data concerning work package estimated effort. This approach to the construction of synthetic data allows for experimental control of the evaluation under 'laboratory conditions'. Such experimental control has been argued to be an important aspect of SBSE that complements empirical analysis on real-world case studies (Harman et al. 2012a). Antoniol et al. (2005) applied their search-based algorithms to real-world data from a large Y2K maintenance project, providing empirical evidence about search-based project management that complements (but does not replace the need for) the experimental data from other studies.

Many other approaches and formulations have been introduced for the software project management problem. For example, Alvarez-Valdes et al. (2006) applied Scatter Search to the problem of minimising project duration. Hericko et al. (2008) used a gradient-based optimisation method to optimise project team size while minimising project effort. Chen and Zhang (2013) used an Ant Colony Optimisation (ACO) approach. Kang et al. (2011) optimised the scheduling of human resource allocations by using a variant of SA, taking into account individual and team constraints based on the literature and interviews with experts in the industry, and by employing real data to validate their proposal. Rahman et al. (2010) also reported on an empirical analysis carried out exploiting real data on the use of both GA and a greedy approach that makes the locally optimal choice at each stage to assign developer to tasks and bug fixing activities. Different aspects of the management also focused on the allocation of staff (Barreto et al. 2008; Kapur et al. 2008) and the provision of decision support (Cortellessa et al. 2008). Di Penta et al. (2011) provided a recent evaluation of search-based techniques for scheduling and staffing for software project management assessed on real-world examples in the style of detailed empirical evaluations using nonsynthetic data. Their work covers single and multiple objective formulations, catering for conflicting project objectives, schedule fragmentation and developer expertise. Results are presented for HC, SA and GAs and applied to two real-world software projects.

15.3.3 Risk-Based Approach

All software projects suffer from risk (Sect. 1.3.8). Risks can be categorised as product risks and process risks. Risks to the product concern the possibility that there may be flaws in the product that make it less attractive to customers, while process risks concern the problems that may cause delays in the project completion time, or reduced budgets available forcing compromise.

Kiper et al. (2007) were concerned with the problem of technical risks, seeking to find those verification and validation activities that could be deployed to reduce risks subject to budget. This work can be categorised as a product risk; it seeks to reduce the chance that the product will exhibit a risk of faults or other low quality.

Gueorguiev et al. (2009) concerned process risk, focusing on the chances that misestimating the effort required for a work package might lead to overruns which would adversely affect the completion time of the project. The effects of overruns are not immediately obvious since they can affect the critical path, making previously less important work packages become more important for the overall project completion time.

Jiang et al. (2007) proposed an approach that extracts personnel risk information from historical data and integrates risk analysis into project scheduling performed with GA. A rescheduling mechanism is designed to detect and mitigate potential risks along with the software project development. However, the proposed approach has not been empirically validated.

Xiao et al. (2013) presented a search-based risk mitigation planning method based on GA for project portfolio management. Their results showed that with various risk mitigation actions and project objective settings, different plans can be effectively obtained, thus providing decision support for managers.

15.3.4 Overtime Planning

Effort estimation and planning of projects are hard problems that can be supported by decision support tools. Where these tools are inadequate or the project encounters unexpected ‘mission creep’, the consequences can be highly detrimental for the software engineers working on the project and the products they produce. Typically, the only remaining solution open to the project manager is to fall back on the allocation of overtime. However, unplanned overtime results in bad products, as has been repeatedly demonstrated in the literature (Akula and Cusick 2008; Nishikitani et al. 2005). It also has harmful effects on the engineers forced into such punitive working practices (see, e.g., Kleppa et al. 2008). The spectre of the ‘death march project’ (Yourdon 1997) hangs over many software engineering activities, largely as a result of the inability to plan for and manage the deployment of overtime. More thorough overtime planning is not only beneficial to the software engineers who have to undertake the work (Beckers et al. 2008), there is also evidence that it produces better products when used in agile team (Mann and Maurer 2005).

Motivated by these observations, Ferrucci et al. (2013) introduced a multi-objective formulation of the project overtime planning for software engineering management. The approach is able to balance trade-offs between project duration, overrun risk and overtime resources for three different risk assessment models. It is applicable to standard software project plans, such as those constructed using the critical path method, widely adopted by software engineers and implemented in many tools. To analyse the effectiveness of the approach, they reported an empirical study on six real-world software projects, ranging in size from a few person-weeks to roughly four person-years.

The experiments reveal that the approach was significantly better than standard multi-objective search in 76 % of experiments and was significantly better than random search in 100 % experiments. Moreover, it always significantly outperformed standard overtime planning strategies reported in the literature. Furthermore, the Pareto fronts obtained by the proposed approach can yield actionable insights into project planning trade-offs between risk, duration, and overtime using different risk assessment models. Software engineers can exploit this information when making decisions about software project overtime planning.

15.3.5 Software Development Effort Estimation

Software development effort estimation concerns with the prediction of the effort needed to develop a software project. Such an effort is usually quantified as person-hours or person-months. Development effort is considered as one of the major component of software costs, and it is usually the most challenging to predict (Sect. 3.1). In the last few decades, several methods have been proposed to support project managers in estimating software development effort (Briand and Wieczorek 2002). In particular, data-driven methods exploit data from past projects to estimate the effort for a new project under development (typical methods are linear regression and case-based reasoning). These data consist of information about some relevant factors (named cost drivers) and the effort actually spent to develop the projects. Usually a data-driven method tries to explain the relation between effort and cost drivers building an estimation model (equation) that is used to predict the effort for a new project. Also search-based methods have been used to build effort estimation models by formulating the problem as an optimisation problem that aims to identify the best model, that is, the one providing the most accurate estimates. In the following, we highlight some key problems in the use of SBSE for effort estimation and how they have been addressed and assessed.

Dolado (2001) was the first to employ GP to automatically derive equations for estimating development effort and he observed a similar or better prediction than regression equations. Based on these encouraging results, other investigations have been carried out comparing search-based approaches with other techniques proposed in the literature. Most of the studies are based on GP, and only more recently other search-based techniques such as Tabu Search (Ferrucci et al. 2010a, b) and multi-objective evolutionary approaches (Ferrucci et al. 2011; Minku and Yao 2012, 2013) have been employed.

As for the setting of these techniques, usually a trial-and-error process has been employed carrying out a validation process with different settings and selecting the one providing the best results (Ferrucci et al. 2010d). This practice is time-consuming and it has to be repeated every time new data are used, thus limiting the adoption of search-based approaches by practitioners. A heuristic approach has been instead exploited in (Ferrucci et al. 2010c) and empirically analysed in (Sarro 2013). The heuristic approach was originally suggested in (Doval et al. 1998) to set population size and number of generations of a GP for software clustering. In particular, given a project dataset containing V features, they set the number of iterations to 10 V and stop the search after 1,000 V iterations or if the fitness value of the best solution does not change in the last 100 V iterations. Thus, such heuristics adapts the search process to the size of the problem under investigation. Sarro (2013) extended the same heuristics to work also with Tabu Search (TS) (setting to V the length of Tabu List) and assessed its effectiveness by comparing it with respect to the use of five different configurations characterised by very small, small, medium, large, and very large number of solutions. The results obtained by exploiting GP and TS on seven public datasets highlighted that the considered heuristics is suitable to set both techniques since it provided comparable or superior prediction accuracy with respect to the ones obtained with the other configurations. Moreover, TS and GP configured by using the heuristics are much faster than the configurations obtained using other settings. This allowed saving time and computational resources without affecting the accuracy of the estimation models built with TS and GP, so the use of the heuristics has been revealed a cost-effective way to set these techniques on the considered datasets.

Another crucial design choice for search-based approaches is the definition of the fitness function that indicates how a solution is suitable for the problem under investigation driving the search towards optimal solutions. For the effort estimation problem, the fitness function should be able to assess the accuracy of estimation models.

It is worth noting that several different accuracy measures have been proposed in the literature for assessing the effectiveness/accuracy of effort prediction models, such as the mean of absolute error (MAE), the mean of squared error (MSE) the mean and median of magnitude of relative error (MMRE and MdMRE, respectively), the mean and median of magnitude of estimate relative error (MERE and MdEMRE, respectively) and the prediction at level k [Pred(k)] (Conte et al. 1986) (Kitchenham et al. 2001). Usually they represent a cumulative measure of the error/residual, that is, the difference between actual effort and predicted effort (e.g., MAE, MSE), or of the relative error with respect the actual (e.g., MMRE, MdMRE) or the estimated effort (e.g., MEMRE, MdEMRE); or a percentage of the cases where the considered error is less of a chosen threshold, e.g., Pred(25).

Among them, the MMRE (Conte et al. 1986) represents the most widely used measure for assessing effort estimation proposals, thus it is not surprising that it has been also the most used fitness function in the study employing search-based techniques. Nevertheless, MMRE reliability has been questioned by several researchers [e.g., Kitchenham et al. (2001), Shepperd and MacDonell (2012)] and has been shown that it does not select the best model among competing ones. Presently, there does not exist a unique measure universally accepted as the best way to assess the estimation accuracy of effort models. On the other hand, each proposed measure focuses the attention on a specific aspect. As a matter of fact, Pred(25) measures how well an effort model performs, while MMRE measures poor performance; MMRE is more sensitive to overestimates and MEMRE to underestimates. Thus, it could be argued that the choice of the criterion for assessing predictions and establishing the best model can be a managerial issue. So, a project manager could prefer to use Pred(25) as the criterion for judging the quality of a model, while another might prefer to use another criterion, just, for example, MMRE to better control overestimates, or, to get a more reliable assessment, another could jointly employ several evaluation criteria covering different aspects of model performances (e.g., underestimating or overestimating, success or poor performance).

Based on this consideration, search-based methods represent an opportunity due to their flexibility. Indeed, they allow the use as fitness function of any measure able to evaluate some properties of interest, thus allowing a project manager to select his/her preferred accuracy measure so that the search for the model is driven by such a criterion. Moreover, search-based techniques can take into account not only single evaluation criteria but also multiple ones, considering some algebraic expressions of basic measures [e.g., Pred(25)/MMRE] (Ferrucci et al. 2010c) or exploiting more sophisticated approaches based on multi-objective optimisation (Ferrucci et al. 2011; Minku and Yao 2012, 2013; Sarro et al. 2012b).

Different fitness functions have been employed in the studies carried out so far. They highlighted that such a choice can affect the performance of the obtained models: each fitness function is able to guide towards estimation models with better accuracy in terms of the selected criterion, but some of them can degrade the other summary measures (Burgess and Lefley 2001; Ferrucci et al. 2010c; Lokan 2005; Sarro 2013). Thus, project managers should be aware of this effect and should take care to select the right evaluation criterion as fitness function.

Another aspect that is important for project managers (both for trust on the solution proposed by an estimation technique and for improving the data collection process of current projects) is concerned with the transparency of the proposed solution. Search-based approaches produce transparent solutions because the prediction model is an algebraic expression that makes explicit any information about the contribution of each variable in the model (this is not always the case for other estimation techniques, e.g., neural networks).

Nevertheless, due to the variable length of the expression tree, some proposed GP approaches [e.g., Dolado (2001), Burgess and Lefley (2001), Lefley and Shepperd (2003)] produced unclear expressions that need to be simplified. To improve the transparency of solutions, an evolutionary computation method, named Grammar Guided Genetic Programming (GGGP), was proposed by Shan et al. (2002) that exploited grammars to impose syntactical constraints and incorporate background knowledge. Another approach to simplify the transparency of solutions provided by GP was exploited in (Ferrucci et al. 2010c) based on the use of trees of fixed depth and crossover and mutation operators that preserved the syntactic structure.

As for the empirical studies, industrial datasets have been widely used in effort estimation studies. They come from a single company [e.g., Desharnais (Menzies et al. 2012)] or from multiple companies [e.g., Tukutuku (Ferrucci et al. 2010a)] and are related to both software and web projects. The criteria used to evaluate the accuracy of the obtained estimates are all based on summary measures, in particular MMRE and Pred(25). To make the comparison more reliable, some studies complemented the analysis with graphical tools (boxplot of residuals) and statistical tests.

As a general result of these studies, we can conclude that search-based techniques behave consistently well obtaining estimation accuracy comparable or better than other widely used estimation techniques, such as the ones based on Manual StepWise Regression (MSWR) or Case-Based Reasoning (CBR). Recently, it has also been highlighted that TS outperformed GP since it turns out to be more efficient, while preserving the same accuracy (Sarro 2013).

Search-based methods have also been used in combination with other estimation techniques [see, e.g., Braga et al. (2008), Faheem et al. (2008)], such as some Machine Learning (ML) techniques, aiming to obtain better estimates. Indeed, as reported in several studies, ML approaches have the potential as techniques for software development effort estimation; nevertheless, their accuracy strongly depends on an accurate setting of these methods [see, e.g., Song et al. (2013)]. As an example, to use CBR we have to choose among many similarity measures, number of analogies, and analogy adaption strategies (Mendes 2009), while to employ Support Vector Regression (SVR) we have to set several parameters depending also on the employed kernel function exploited to deal with nonlinear problems (Cortes and Vapnik 1995).

There are no general guidelines on how to best configure these techniques since the appropriate setting often depends on the characteristics of the employed dataset. An examination of all possible values for configuration parameters of each technique is often not computationally affordable, as the search space is too large, also due to the interaction among parameters, which often cannot be separately optimised. Another aspect that can influence the accuracy of estimation techniques is the quality of input features, thus a Feature Subset Selection (FSS) is usually recommended to select a subset of relevant features to be used in the model construction process.

To address both the above-mentioned problems, the use of search-based approaches has been proposed and investigated. In the following, we first discuss four works conceived to select a suitable configuration for some estimation techniques, namely neural network, CBR and SVR, and then we discuss four works that exploited GAs to address the FSS problem.

Shukla (2000) was the first to propose a GA to configure Neural Network (NN) predictor in order to improve its estimation capability. In particular, GA had to find suitable weights for NN layer connections guided by a fitness function that minimises MSE values. The empirical study based on two public datasets, that is, COCOMO and Kemerer (Menzies et al. 2012), showed that GA + NN provided significantly better prediction than common used AI-oriented methods, such as CARTX and Quick Propagation trained NN. Similarly, Papatheocharous and Andreou (2009) enhanced the use of artificial neural networks by using a genetic algorithm. Their results showed that using GA to evolve the network architectures (both input and internal hidden layers) reduced the Mean Relative Error (MRE) produced by the output results of each network.

Chiu and Huang (2007) applied GA to CBR to adjust the reused effort obtained by considering different similarity distances (i.e., Euclidean, Minkowski, and Manhattan distances) between pairs of software projects. The result obtained on two industrial datasets revealed that the proposed GA improved the estimations of CBR.

To automatically select suitable SVR settings, Corazza et al. (2010, 2013) proposed and assessed an approach based on the use of TS. A total of 21 datasets were employed and several benchmarks were taken into account. The results revealed that the combination of TS and SVR significantly outperformed all the other techniques, showing that the proposed approach represents a suitable technique for software development effort estimation.

The first work that proposed the use of a search-based approach to address the FSS problem in the context of effort estimation was the one of Kirsopp et al. (2002). They employed Hill Climbing to select the best set of project features to be used with CBR. The combined approach was evaluated on an industrial dataset of 407 observations and the results showed that it performed better than random feature selection and forward sequential selection.

Li et al. (2009) proposed a GA to simultaneously optimise the selection of the feature weights and projects to be used with CBR. The empirical results employing four datasets [two industrial (Menzies et al. 2012) and two artificial (Li et al. 2009)] showed that the use of GA + CBR provided significantly better estimations than CBR.

GA was also used to improve the accuracy of an effort estimation model built by combining social choice (voting rules were used to rank projects determining similar projects) and analogy-based approaches (Koch and Mitlöhner 2009). In particular, GA was employed to find suitable weights to be associated to the project attributes. The results revealed that the proposed approach provided the best value for Pred(25), but worse MMRE values with respect to other techniques (LR, ANN, CART, COCOMO and Grey Relational Analysis).

Huang et al. (2008) integrated a GA to Grey Relational Analysis to find the best fit of weights for each software effort driver. The experimental results showed comparable (COCOMO dataset) and better accuracy (Albrecht dataset) with respect to CBR, CART, and ANN.

15.4 Possible Directions for Future Work on Search-Based Project Management

In this section, we outline several directions for future work in search-based project management, highlighting promising areas that emerge for the analysis of trends within this subfield of SBSE.

15.4.1 Iterative Optimisation

Several authors have suggested and adopted interactive evolution (Harman 2007b) design-based (Simons and Parmee 2008, 2012) and comprehension-based (Harman 2007a) software engineering tasks. However, only one attempt has been made to apply this technique, which can incorporate human expert knowledge directly into fitness computation, in project management (Shackelford and Corne 2001). Since a software project management task is inherently human-centric, it would be natural to explore the use of interactive evolution as a technique for ensuring that the project manager's expertise is accounted for in software project management (Shackelford 2007). The difficulty, as with all interactive evolution, lies in finding a way in which the manager can influence the computation of fitness without overburdening him/her with request for 'fitness assessment'.

It is also an open challenge as to how this judgment can be best incorporated into fitness. For example, the manager may be aware that certain individuals cannot work together, that certain work packages are more critical or that some dependence can be broken to make project more 'parallelisable'. This information cannot simply be requested from the manager at the outset of the optimisation process; there is too much of it, and much of the information is implicit. Rather, we need to make the whole process of using search-based project management tools more interactive, so that the manager is able to 'realise' that they know something of importance at the specific point in the optimisation process at which it applies and to introduce this domain knowledge into the overall planning process in a natural and seamless way.

15.4.2 Dynamic Adaptive Optimisation

In order to maximise the value of interactive solutions to project management, we need dynamic adaptive approaches to SBSE (Harman et al. 2012a). As an example, effective resource scheduling is complicated by different disruptions, such as requirements changes, bug fixing, or staff turnover, and dynamic resource scheduling can help address such potentially disruptive events (Xiao et al. 2010a, 2013). If solutions can be computed in real time and presented to the decision-maker in an intuitive form, then the decision-maker can ask on-the-fly 'what if' questions to help decide on key project commitments. In an ideal world, the decision-maker would interact with the tool, exploring the possible implications of the decisions, with the optimisation continuing to provide updated best-so-far solutions as the decision-maker interacts. This may require fundamentally different approaches to the algorithms and formulations that underlie search-based software project management.

15.4.3 Multi-objective Optimisation

It has been argued that in order to better match real-world scenarios, SBSE should move from a single objective paradigm to a multi-objective paradigm (Harman et al. 2007, 2012b). Indeed, more recent SBSE work has followed a more multi-objective style of approach, touching many application areas including requirements (Finkelstein et al. 2008, 2009; Zhang et al. 2007), testing (Everson and Fieldsend 2006; Harman 2011; Harman et al. 2007b), refactoring (Harman and Tratt 2007) and also, not least, project management (Ferrucci et al. 2011, 2013; Gueorguiev et al. 2009; Minku and Yao 2012, 2013; Rodriguez et al. 2011; Sarro et al. 2012b; Stylianou and Andreou 2013). Most problems in software engineering involve multiple competing objectives, and this is an observation most keenly felt in project management. Much of the future work on SBSE for project management is likely to focus on decision support in complex multi-objective problem spaces.

15.4.4 Co-evolution

In co-evolutionary computation, two or more populations of solutions evolve simultaneously with the fitness of each depending upon the current population of the other. Co-evolution can be either cooperative or competitive. In competitive evolution, two (or more) populations of candidate solutions compete with each other for supremacy, the fitness of one depending on the other, such that improvements in one population tend to lead to lower fitness in the other. This is analogue to the wellknown 'predator-prey' model of evolutionary biology, and it has found application in SBSE work on testing (Adamopoulos et al. 2004) where predators are test cases and programs and their faults are the prey on which the test cases feed.

However, co-evolution need not always follow a predator-prey model; it can also be a cooperative, symbiotic process, just as often occurs in nature. In this cooperative co-evolutionary model, several populations, all of which have distinct fitness functions, nevertheless depend on one another, without necessarily being in conflict. This is a natural model for project management, in which we seek, for example, a mutually supportive allocation of staff to teams and, simultaneously, an allocation of teams to work packages. Ren et al. (2011) explore the cooperative co-evolutionary model of search-based project management. The close fit between project management objectives and co-evolutionary models makes this a natural choice and one that deserves more attention.

15.4.5 Software Project Benchmarking

One of the enduring problems researchers concerned with project management face is the lack of available real world project data. It remains a common problem to this day, especially for project planning and staffing. Indeed, despite there existing both publicly available (Menzies et al. 2012) and private (ISBSG 2013) repositories of real project data for software project estimation, repositories containing information for project planning are not available. The promising results achieved in search-based project management may lead to the false impression that these techniques are readily available for industrial application, whereas many papers just use synthetic data, do not model skill or have a realistic representation of skill and/or risk. Previous work on software planning and staffing often relies on simulation of the likely course of the project rather than real data to assess the performance of the proposed approaches. Indeed, datasets of real-world projects are scarce; effort data are seldom ever made public not to mention skill and other kind of employee's details (e.g., percentage of overhead, abilities to self-adapt to new project, management style, team leaders or soft skill). Despite researchers making some effort to employ real data and to deal with a more human-centric problem (Chap. 4), to collect more real data on software projects and to make it publicly accessible still remains an open important challenge.

15.4.6 Confident Estimates

Obtaining exact time and effort estimates in software project management is impossible due not only to the inherently nature of estimates, but also to incomplete, uncertain and/or noisy input used as the basis of the estimates. Rather than generating exact estimates, it would be beneficial to introduce some level of uncertainty and measure its effect on the management process. As an example, sensitivity analysis can be a very useful means of determining the vulnerability of an estimate to particular assumptions and the level of confidence that can be placed in that estimate and the promising results obtained by using search-based approaches to assess software project uncertainty make us confident that these techniques can be a key instrument to support this kind of analysis [see, e.g., Harman et al. (2009)].

Elsewhere, Harman (2007a) highlighted four future directions for the use of search-based approaches to obtain more confident predictive modelling that we want herein recall:

1. Incorporation of risk into predictive models
2. More effective measurement of cost
3. More reliable models (even at the expense of predictive power; trading median accuracy for reduced variance over iterated predictions)
4. Sensitivity analysis to determine which aspects are more important

15.4.7 Decision Support Tools

Despite the promising results highlighted in the research papers, the use of search-based approaches for project management is still in the development/research stage. To the best of our knowledge, only one tool for project management based on search-based approaches has been recently proposed (Stylianou et al. 2012). We believe that to transfer the most successful methods into practice we need to develop them as freely available decision support tools. Indeed, this will allow an extensive evaluation of the interface between the technical aspects on which the research has been focused and other related socio-technical issues for implementation and exploitation, such as user interface, ease of use, human-computer interaction, and decision support. Moreover, this will allow us also to get feedback from practitioners on the usefulness and cost/effectiveness of the proposed approaches.

15.5 Conclusions

SBSE has proved widely applicable across many fields of software engineering activities. In those software engineering activities closely associated with the software product, SBSE has tended to be used as a means of finding good solutions, guided by a fitness function. By contrast, its role in the earlier phases of the software development cycle, more associated with the establishment of plans and processes, has tended to be subtler. In particular, for software project management, SBSE has tended to be used to provide decision support rather than to seek a single solution. This is a naturally exploratory and multi-objective scenario. As we have seen, search-based techniques have potential to support software project managers, with predictions, analysis of potential scenarios and the optimised configuration of process parameters. The review of trends in this chapter demonstrates that this is an active and growing field.

References

Adamopoulos K, Harman, M, Hierons RM (2004) How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In: Proceedings of the 6th conference on genetic and evolutionary computation, pp 1338–1349

Afzal W, Torkar R, Feldt R, Gorschek T (2014) Prediction of faults-slip-through in large software projects: an empirical evaluation. Software Qual J 22:51–86. doi:10.1007/s11219-013-9205-3

Aguilar-Ruiz JS, Ramos I, Riquelme JC, Toro M (2001) An evolutionary approach to estimating software development projects. Inf Softw Technol 43(14):875–882

Aguilar-Ruiz JS, Riquelme JC, Ramos I (2002) Natural evolutionary coding: an application to estimating software development projects. In: Proceedings of the 4th conference on genetic and evolutionary computation

Akula B, Cusick J (2008) Impact of overtime and stress on software quality. In: Proceedings of the 4th international symposium on management, engineering, and informatics

Alba E, Chicano F (2005) Management of software projects with GAs. In: Proceedings of the 6th metaheuristics international conference, pp 1152:1-6

Alba E, Chicano F (2007) Software project management with GAs. Inf Sci 177(11):2380–2401

Alvarez-Valdes R, Crespo E, Tamarit JM, Villa F (2006) A scatter search algorithm for project scheduling under partially renewable resources. J Heuristics 12(1–2):95–113

Antoniol G, Di Penta M, Harman M (2004) A robust search-based approach to project management in the presence of abandonment, rework, error and uncertainty. In: Proceedings of the 10th international symposium on the software metrics, pp 172–183

Antoniol G, Di Penta M, Harman M (2005) Search-based techniques applied to optimization of project planning for a massive maintenance project. In: Proceedings of the 21st IEEE international conference on software maintenance, pp 240–249

Azar D (2010) A genetic algorithm for improving accuracy of software quality predictive models: a search-based software engineering approach. Int J Comput Intell Appl 9(2):125–136

Barreto A, Barros M de O, Werner CM (2008) Staffing a software project: a constraint satisfaction and optimization-based approach. Comput Oper Res 35(10):3073–3089

Beckers DG, van der Linden D, Smulders PG, Kompier MA, Taris TW, Geurts SA (2008) Voluntary or Involuntary? control over overtime and rewards for overtime in relation to fatigue and work satisfaction. Work Stress 22(1):33–50

Bouktif S, Kégl B, Sahraoui H (2002) Combining software quality predictive models: an evolutionary approach. In: Proceedings of the international conference on software maintenance, pp 385–392

Bouktif S, Azar D, Precup D, Sahraoui H, Kégl B (2004) Improving rule set based software quality prediction: a genetic algorithm based approach. J Object Technol 3(4):227–241

Bouktif S, Sahraoui H, Antoniol G (2006) Simulated annealing for improving software quality prediction. In: Proceedings of the 8th conference on genetic and evolutionary computation, pp 1893–1900

Braga PL, Oliveira ALI, Meira SRL (2008) A GA-based feature selection and parameters optimization for support vector regression applied to software effort estimation. In: Proceedings of the ACM symposium on applied computing, pp 1788–1792

Briand L, Wieczorek I (2002) Software resource estimation. Encyclopedia Softw Eng 2:1160–1196

Brooks FP Jr (1975) The mythical man month: essays on software engineering. Addison-Wesley Publishing Company, Reading, MA

Burgess CJ, Lefley M (2001) Can genetic programming improve software effort estimation: a comparative evaluation. Inf Softw Technol 43(14):863–873

Chang CK (1994) Changing face of software engineering. IEEE Softw 11(1):4–5

Chang CK, Chao C, Hsieh S-Y, Alsalqan Y (1994) SPMNet: a formal methodology for software management. In: Proceedings of the 18th international computer software and applications conference, p 57

Chang CK, Chao C, Nguyen TT, Christensen M (1998) Software project management net: a new methodology on software management. In: Proceedings of the 22nd international computer software and applications conference, pp 534–539

Chang CK, Christensen MJ, Zhang T (2001) Genetic algorithms for project management. Ann Softw Eng 11(1):107–139

Chao C, Komada J, Liu Q, Muteja M, Alsalqan Y, Chang C (1993) An application of genetic algorithms to software project management. In: Proceedings of the 9th international advanced science and technology, pp 247–252

Chen WN, Zhang J (2013) Ant colony optimization for software project scheduling and staffing with an event-based scheduler. IEEE Trans Softw Eng 39(1):1–17

Chicano F, Luna F, Nebro AJ, Alba E (2011) Using multi objective metaheuristics to solve the software project scheduling problem. In: Proceedings of the 13th conference on genetic and evolutionary computation, pp 1915–1922

Chiu NH, Huang S (2007) The adjusted analogy-based software effort estimation based on similarity distances. J Syst Softw 80(4):628–640

Conte D, Dunsmore H, Shen V (1986) Software engineering metrics and models. The Benjamin/Cummings Publishing Company, Redwood City, CA

Corazza A, Di Martino S, Ferrucci F, Gravino C, Sarro F, Mendes E (2010) How effective is Tabu search to configure support vector regression for effort estimation?. In: Proceedings of the 6th international conference on predictive models in software engineering, pp 4:1-10

Corazza A, Di Martino S, Ferrucci F, Gravino C, Sarro F, Mendes E (2013) Using Tabu search to configure support vector regression for effort estimation. Empir Softw Eng 18(3):506–546

Cortellessa V, Marinelli F, Potena P (2008) An optimization framework for "build-or-buy" decisions in software architecture. Comput Oper Res 35(10):3090–3106

Cortes C, Vapnik V (1995) Support-vector networks. Mach Learn 20(3):273–297

Di Martino S, Ferrucci F, Gravino C, Sarro F (2011) A genetic algorithm to configure support vector machines for predicting fault-prone components. In: PROFES 2011. Lecture notes in computer science, vol 6759. Springer, Heidelberg, p 247

Di Penta M, Antoniol G, Harman M, Qureshi F (2007) The effect of communication overhead on software maintenance project staffing: a search-based approach. In: Proceedings of the 23rd IEEE international conference on software maintenance, pp 315–324

Di Penta M, Antoniol G, Harman M (2011) The use of search-based optimization techniques to schedule and staff software projects: an approach and an empirical study. Softw Pract Exp 41 (5):495–519

Dolado JJ (2001) On the problem of the software cost function. Inf Softw Technol 43(1):61–72

Doval D, Mancordis SB, Mitchell S (1998) Automatic clustering of software system using a genetic algorithm. In: Proceedings of the 9th international workshop software technology and engineering practice, pp 73–81

Everson RM, Fieldsend JE (2006) Multiobjective optimization of safety related systems: an application to short-term conflict alert. IEEE Trans Evol Comput 10(2):187–198

Faheem A, Bouktif S, Serhani A, Khalil I (2008) Integrating function point project information for improving the accuracy of effort estimation. In: Proceedings of the international conference on advanced engineering computing and applications in sciences, pp 193–219

Ferrucci F, Gravino C, Mendes E, Oliveto R, Sarro F (2010a) Investigating Tabu search for Web effort estimation. In: Proceedings of the 36th EUROMICRO conference on software engineering and advanced applications, pp 350–357

Ferrucci F, Gravino C, Oliveto R, Sarro F (2010b) Estimating software development effort using Tabu search. In: Proceedings of the 12th international conference on enterprise information systems, vol 1. pp 236–241

Ferrucci F, Gravino C, Oliveto R, Sarro F (2010c) Genetic programming for effort estimation: an analysis of the impact of different fitness functions. In: Proceedings of the 2nd international symposium on search based software engineering, pp 89–98

Ferrucci F, Gravino C, Oliveto R, Sarro F (2010d) Using evolutionary based approaches to estimate software development effort. In: Chis M (ed) Evolutionary computation and optimization algorithms in software engineering: applications and techniques. IGI Global, Hershey, PA, pp 13–28

Ferrucci F, Gravino C, Sarro F (2011) How multi-objective genetic programming is effective for software development effort estimation? In: Proceedings of the 3rd international symposium on search based software engineering. Lecture notes in computer science, vol 6956. Springer, Heidelberg, pp 274–275

Ferrucci F, Harman M, Ren J, Sarro F (2013) Not going to take this anymore: multi-objective overtime planning for software engineering projects. In: Proceedings of the 35th IEEE international conference on software engineering, pp 462–471

Finkelstein A, Harman M, Mansouri S. A, Ren J, Zhang Y (2008) “Fairness Analysis” in requirements assignments. In: Proceedings of the 16th IEEE international requirements engineering conference, pp 115–124

Finkelstein A, Harman M, Mansouri SA, Ren J, Zhang Y (2009) A search based approach to fairness analysis in requirement assignments to aid negotiation, mediation and decision making. Requir Eng 14(4):231–245

Gueorguiev S, Harman M, Antoniol G (2009) Software project planning for robustness and completion time in the presence of uncertainty using multi objective search-based software engineering. In: Proceedings of the genetic and evolutionary computation conference, pp 1673–1680

Harman M (2007a) The current state and future of search-based software engineering. In: Proceedings of the conference on future of software engineering, pp 342–357

Harman M (2007b) Search-based software engineering for program comprehension. In: Proceedings of the 15th IEEE international conference on program comprehension, pp 3–13

Harman M (2010a) The relationship between search-based software engineering and predictive modelling. In: Proceedings of the 6th international conference on predictive models in software engineering, pp 1

Harman M (2010b) Why the virtual nature of software makes it ideal for search-based optimization. In: Proceedings of the 13th international conference on fundamental approaches to software engineering, pp 1–12

Harman M (2011) Making the case for MORTO: multi objective regression test optimization. In: Proceedings of the 1st international workshop on regression testing, pp 111–114

Harman M, Clark JA (2004) Metrics are fitness functions too. In: Proceedings of the 10th international symposium on software metrics, pp 58–69

Harman M, Jones BF (2001) Search-based software engineering. Inf Softw Technol 43 (14):833–839

Harman M, Tratt L (2007) Pareto optimal search-based refactoring at the design level. In: Proceedings of the 9th conference on genetic and evolutionary computation, pp 1106–1113

Harman M, Hierons R, Proctor M (2002) A new representation and crossover operator for search-based optimization of software modularization. In: Proceedings of the 4th conference on genetic and evolutionary computation, pp 1351–1358

Harman M, Lakhotia K, McMinn P (2007) A multi-objective approach to search-based test data generation. In: Proceedings of the 9th conference on genetic and evolutionary computation, pp 1098–1105

Harman M, Krinke J, Ren J, Yoo S (2009) Search-based data sensitivity analysis applied to requirement engineering. In: Proceedings of the 11th conference on genetic and evolutionary computation, pp 1681–1688

Harman M, McMinn P, Teixeira de Souza J, Yoo S (2010) Search-based software engineering: techniques, taxonomy, tutorial. LASER Summer School 2010, pp 1–59

Harman M, Burke E, Clark JA, Yao X (2012a) Dynamic adaptive search-based software engineering. In: Proceedings of the 6th IEEE international symposium on empirical software engineering and measurement, pp 1–8

Harman M, Mansouri A, Zhang Y (2012b) Search-based software engineering: trends, techniques and applications. ACM Comput Surv 45(1):11–75

Hericko M, Zivkovic A, Rozman I (2008) An approach to optimizing software development team size. Inf Process Lett 108(3):101–106

Holland J (1975) Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor, MI

Huang SJ, Chiu NH, Chen LW (2008) Integration of the grey relational analysis with genetic algorithm for software effort estimation. Eur J Oper Res 188(3):898–909

ISBSG (2013) Data repository. Available at http://www.isbsg.org

Jarillo G, Succi G, Pedrycz W, Reformat M (2011) Analysis of software engineering data using computational intelligence techniques. In: Proceedings of the 7th international conference on object oriented information systems, pp 133–142

Jiang H, Chang CK, Xia J, Cheng S (2007) A history-based automatic scheduling model for personnel risk management. In: Proceedings of the 31st computer software and application conference, pp 361–364

Kang D, Jung J, Bae DH (2011) Constraint-based human resource allocation in software projects. Softw Pract Exp 41(5):551–577

Kapur P, Ngo-The A, Ruhe G, Smith A (2008) Optimized staffing for product releases and its application at chartwell technology. J Softw Maint Evol Res Pract 20(5):365–386

Khoshgoftaar TM, Liu Y (2007) A multi-objective software quality classification model using genetic programming. IEEE Trans Reliab 56(2):237–245

Khoshgoftaar TM, Liu Y, Seliya N (2003) Genetic programming-based decision trees for software quality classification. In: Proceedings of the 15th international conference on tools with artificial intelligence, pp 374–383

Kiper JD, Feather MS, Richardson J (2007) Optimizing the V&V process for critical systems. In: Proceedings of the 9th conference on genetic and evolutionary computation, p 1139

Kirsopp C, Shepperd MJ, Hart J (2002) Search heuristics, case-based reasoning and soft- ware project effort prediction. In Proceedings of the genetic and evolutionary computation conference, pp 1367–1374

Kitchenham B, Pickard LM, MacDonell SG, Shepperd MJ (2001) What accuracy statistics really measure. IEEE Proc Softw 148(3):81–85

Kleppa E, Sanne B, Tell GS (2008) Working overtime is associated with anxiety and depression: the Hordaland health study. J Occup Environ Med 50(6):658–666

Koch S, Mitlöhner J (2009) Software project effort estimation with voting rules. Decis Support Syst 46(4):895–901

Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge, MA

Lefley M, Shepperd MJ (2003) Using genetic programming to improve software effort estimation based on general data sets. In: Proceedings of the 5th genetic and evolutionary computation conference, pp 2477–2487

Li YF, Xie M, Goh TN (2009) A study of project selection and feature weighting for analogy based software cost estimation. J Syst Softw 82(2):241–252

Liu Y, Khoshgoftaar TM (2001) Genetic programming model for software quality classification. In: Proceedings of the 6th IEEE international symposium on high-assurance systems engineering: special topic: impact of networking, pp 127–136

Liu Y, Khoshgoftaar TM (2003) Building decision tree software quality classification models using genetic programming. In: Proceedings of the 5th genetic and evolutionary computation conference, pp 1808–1809

Liu Y, Khoshgoftaar T (2004) Reducing overfitting in genetic programming models for software quality classification. In: Proceedings of the 8th IEEE international symposium on high assurance systems engineering, pp 56–65

Lokan C (2005) What should you optimize when building an estimation model? In: Proceedings of the 11th IEEE international symposium on metrics, pp 34

Luna F, Chicano JF, Alba E (2012) Robust solutions for the software project scheduling problem: a preliminary analysis. Int J Metaheuristic 2(1):56–79

Mann C, Maurer F (2005) A case study on the impact of scrum on overtime and customer satisfaction. In: Agile development conference, pp 70–79

McMinn P (2004) Search-based software test data generation: a survey. Softw Test Verif Reliab 14 (2):105–156

Mendes E (2009) Web cost estimation and productivity benchmarking. software engineering, vol 5413, Lecture notes in computer science. Springer, Heidelberg, pp 194–222

Menzies, T, Caglayan B, Kocaguneli E, Krall J, Peters F, Turhan B (2012) The PROMISE repository of empirical software engineering data. http://promisedata.googlecode.com

Minku LL, Yao X (2012) Software effort estimation as a multi-objective learning problem. ACM Trans Softw Eng Methodol 22(4):35:1–35:32

Minku LL, Yao X (2013) An analysis of multi-objective evolutionary algorithms for training ensemble models based on different performance measures in software effort estimation. In: Proceedings of the 9th international conference on predictive models in software engineering, pp 8:1–8:10

Minku LL, Sudholt D, Yao X (2012) Evolutionary algorithms for the project scheduling problem: runtime analysis and improved design. In: Proceedings of the genetic and evolutionary computation conference, pp 1221–1228

Minku LL, Sudholt D, Yao X (2013) Improved evolutionary algorithm design for the project scheduling problem based on runtime analysis. IEEE Trans Softw Eng 40:83–102. doi:10.1109/TSE.2013.52

Mitchell BS, Mancoridis S (2002) Using heuristic search techniques to extract design abstractions from source code. In: Proceedings of the genetic and evolutionary computation conference, pp 1375–1382

Nishikitani M, Nakao M, Karita K, Nomura K, Yano E (2005) Influence of overtime work, sleep duration, and perceived job characteristics on the physical and mental status of software engineers. Ind Health 43(4):623–629

Papatheocharous E, Andreou SA (2009) Hybrid computational models for software cost prediction: an approach using artificial neural networks and genetic algorithms, vol 19, Lecture notes in business information processing. Springer, Heidelberg, pp 87–100

Rahman MM, Sohan SM, Maurer F, Ruhe G (2010) Evaluation of optimized staffing for feature development and bug fixing. In: Proceedings of the ACM-IEEE international symposium on empirical software engineering and measurement, p 42

Räihä O (2010) A survey on search-based software design. Comput Sci Rev 4(4):203–249

Ren J, Harman M, Di Penta M (2011) Cooperative co-evolutionary optimization on software project staff assignments and job scheduling. In: Proceedings of the 3rd international symposium on search based software engineering, pp 127–141

Rodriguez D, Ruiz M, Riquelme JC, Harrison R (2011) Multiobjective simulation optimisation in software project management. In: Proceedings of the 13th conference on genetic and evolutionary computation, pp 1883–1890

Sarro F (2011) Search-based approaches for software development effort estimation. In: Proceedings of the 12th international conference on product-focused software development and process improvement (doctoral symposium), pp 38–43

Sarro F (2013) Search-based approaches for software development effort estimation. Ph.D. thesis,. University of Salerno, Italy. http://www0.cs.ucl.ac.uk/staff/F.Sarro/

Sarro F, Di Martino S, Ferrucci F, Gravino C (2012a) A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction. In: Proceedings of the 27th annual ACM symposium on applied computing, pp 1215–1220

Sarro F, Ferrucci F, Gravino C (2012b) Single and multi objective genetic programming for software development effort estimation. In: Proceedings of the 27th annual ACM symposium on applied computing, pp 1221–1226

Shackelford MRN (2007) Implementation issues for an interactive evolutionary computation system. In: Proceedings of the genetic and evolutionary computation conference, pp 2933–2936

Shackelford MRN, Corne DW (2001) Collaborative evolutionary multi-project resource scheduling. In: Proceedings of the congress on evolutionary computation, vol 2. pp 1131–1138

Shan Y, McKay RI, Lokan CJ, Essam DL (2002) Software project effort estimation using genetic programming. In: Proceedings of international conference on communications circuits and systems, pp 1108–1112

Shepperd MJ, MacDonell SJ (2012) Evaluating prediction systems in software project estimation. Inf Softw Technol 54(8):820–827

Shukla KK (2000) Neurogenetic prediction of software development effort. Inf Softw Technol 42 (10):701–713

Simons CL, Parmee IC (2008) User-centered, evolutionary search in conceptual software design. In: Proceedings of the IEEE congress on evolutionary computation, pp 869–876

Simons CL, Parmee IC (2012) Elegant object-oriented software design via interactive evolutionary computation. IEEE Trans Syst Man Cybern Part C Appl Rev 42(6):1797–1805

Song L, Minku LL, Yao X (2013) The impact of parameter tuning on software effort estimation using learning machines. In: Proceedings of the 9th international conference on predictive models in software engineering

Stylianou C, Andreou AS (2013) A multi-objective genetic algorithm for intelligent software project scheduling and team staffing. Intell Decis Technol 7(1):59–80

Stylianou C, Gerasimou S, Andreou AS (2012) A novel prototype tool for intelligent software project scheduling and staffing enhanced with personality factors. In: Proceedings of the 24th international conference on tools with artificial intelligence, pp 277–284

Xiao J, Osterweil LJ, Wang Q, Li M (2010a) Dynamic resource scheduling in disruption-prone software development environments. In: Proceedings of the 13th conference on fundamental approaches to software engineering, pp 107–122

Xiao J, Osterweil LJ, Wang Q, Li M (2010b) Disruption-driven resource rescheduling in software development processes. In: New modeling concepts for today's software processes. Lecture notes in computer science, vol 6195. Springer, Heidelberg, pp 234–247

Xiao J, Osterweil LJ, Chen J, Wang Q, Li M (2013) Search-based risk mitigation planning in project portfolio management. In: Proceedings of the 2013 international conference on software and system process, pp 146–155

Yoo S, Harman M (2012) Regression testing minimization, selection and prioritization: a survey. Softw Test Verif Reliab 22(2):67–120

Yourdon E (1997) Death March: the complete software developer's guide to surviving 'mission impossible' projects. Prentice-Hall, Upper Saddle River, NJ

Zhang Y (2013) SBSE paper repository. http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/

Zhang Y, Harman M, Mansouri SA (2007) The multi-objective next release problem. In: Proceedings of the 9th conference on genetic and evolutionary computation, pp 1129–1137

Zhang Y, Finkelstein A, Harman M (2008) Search-based requirements optimisation: existing work and challenges. In Proceedings of the 14th international conference on requirements engineering: foundation for software quality, pp 88–94

Biography Filomena Ferrucci is professor of software engineering and software project management at University of Salerno (Italy). Her main research interests include software metrics, effort estimation, search-based software engineering, empirical software engineering, and human-computer interaction. She has been program co-chair of the International Summer School on software engineering.

Mark Harman is professor of software engineering in the Department of Computer Science at University College London where he directs the Centre for Renewable Energy Systems Technology (CREST). He is widely known for work on source code analysis and testing and was instrumental in the founding of the field of search-based software engineering (SBSE), the topic of this chapter. Since its inception in 2001, SBSE has rapidly grown to include over 1,000 authors, from 300 institutions spread over 40 countries.

Federica Sarro is a Research Associate working in the CREST centre, Department of Computer Science, University College London. Her main research areas are empirical software engineering and search-based software engineering with specific interest in project management, software development effort estimation and fault prediction. She has also been working on functional metrics for sizing software products and human-computer interaction. Her recent research interests include app store analysis and automatic program repair.

Chapter 16
Social Media Collaboration in Software Projects

Rachel Harrison and Varsha Veerappa

Abstract Social media has had a big impact on the way that software projects are managed and the way that stakeholders interact with each other: indeed, the nature of software projects has evolved substantially in keeping with the evolution of technology. A direct consequence of the ubiquity of the Internet is the increasing trend toward cooperation outside the boundaries of an office. The interactions involved in software projects have changed accordingly and can be broadly divided into two types: (1) interactions among stakeholders who are in a single location (e.g., people sharing the same office space) and (2) interactions among stakeholders who are in distributed locations (e.g., software projects that are partly implemented offshore). Social media has been and remains a significant facilitator to these kinds of interactions. This chapter looks at the implications of the use of social media software projects in today's changing world.

16.1 Introduction

We use the term *social media* to include all web-based platforms that allow the creation and exchange of user-generated content using Web 2.0 (McNab 2009). Social media can be used in many circumstances. These include collaborative projects like wikis where a group of individuals comes together to share knowledge with the idea that the effort of all the users together gives a better outcome than that of a single user. An example of a very popular collaborative project is Wikipedia.[1]

[1] http://www.wikipedia.org.

R. Harrison (✉) • V. Veerappa
Department of Computing and Communication Technologies, Oxford Brookes University,
Oxford OX2 9AT, UK
e-mail: Rachel.Harrison@brookes.ac.uk; vveerappa@brookes.ac.uk

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_16, © Springer-Verlag Berlin Heidelberg 2014

Another example of social media is the *blog* in which a single user contributes to content but allows other users to interact with features such as *comments* and *likes*. Users can create these blogs easily on specialized platforms like Tumblr,[2] Blogger[3] or Wordpress.[4] Microblogs are very similar to blogs and enable users to post very short entries as brief updates. An example of a very popular microblog platform is Twitter.[5]

Social networking sites such as Facebook[6] and Google+[7] enable users to create personal profile pages (with information such as photos, videos and blogs) and connect to other users to share their profile information. The users can also further interact with each other via email and instant messengers that are integrated into those social networking sites. Such sites also allow organizations to create their own pages, which can be used to promote awareness about their services and activities. Professional networking sites such as LinkedIn[8] are very similar. They enable users to post information about their careers and network with other users with similar interests. Other forms of social media include virtual games and social worlds such as SecondLife[9] and content communities such as YouTube.[10]

The popularity of social media continues to increase. In October 2012, Facebook estimated its number of users to be 1 billion while the number of Tumblr blogs is 79.3 million. Twitter, on the other hand, had more than 500 million users registered as at March 2013. People are now using social media both informally for personal social interactions and formally for business and work. Indeed, social media provide an efficient and instantaneous means of communication that can be very useful to the health services (Eckler et al. 2010; Hawn 2009; McNab 2009), marketing (Mangold and Faulds 2009), education (Martín-Blas and Serrano-Fernández 2009) and governments (Kavanaugh et al. 2012). One sector that has particularly benefited from the use of social media is the software development industry.

In recent years, software development has evolved from a mainly closed in-house activity to a more open and globally distributed one. This new context has given rise to new challenges such as communication, synchronous and asynchronous coordination and trust (Herbsleb 2007). One way to address these problems is to use social media to facilitate communication, task synchronization and knowledge sharing. This can be done at two levels: the project level (where software teams, domain experts and clients interact with each other to ensure a

[2] http://www.tumblr.com.

[3] http://www.blogger.com/.

[4] http://wordpress.com/.

[5] https://twitter.com/.

[6] http://www.facebook.com/.

[7] https://plus.google.com/.

[8] http://www.linkedin.com.

[9] http://secondlife.com/.

[10] http://www.youtube.com.

successful outcome) and the community level (where the software team interacts with peers in the broader community of software professionals to share knowledge).

In this chapter, we look at how the three main groups of people who are directly or indirectly involved in a project (internal stakeholders, external stakeholders and peers) interact. We have carried out a pilot study with practitioners in industry to understand how social media is used in their organizations for the different types of interactions we identified. The results of the study are used as evidence to support the points we make in this chapter.

16.2 Interactions in Software Projects

The three main groups of people involved in a project are internal stakeholders, external stakeholders and peers. External stakeholders are the individuals or organizations that are not part of the organization doing the actual software development activities but who have an influence on the decisions in the projects. They are not usually in the same location as the internal stakeholders. For example, an external stakeholder could be a manager, an end user in the client organization or a domain expert. Such stakeholders are usually involved in the requirements elicitation and elaboration processes as well as during testing. Figure 16.1 depicts the various types of interactions among these stakeholders during a software project with onshore and offshore teams.

Internal stakeholders are individuals or teams within the same company that interact on a daily basis during the software development process. These stakeholders can be co-located or they could be distributed geographically in the same country or across the world. In a software project, there is usually a project manager in charge of the main team where the project decisions are taken. There may also be other teams that are located in the same country as the main team. Some of the tasks may also be outsourced to offshore teams. All these stakeholders are internal stakeholders. Software engineers, project managers and testers are examples of such stakeholders. Their involvement in software projects ranges from requirements elicitation through to design, implementation and testing.

Peers are individuals who are known to the internal stakeholders either through previous acquaintance or through necessity for the project. They usually share their knowledge and experiences and even refer new peers to the internal stakeholders. Peers are usually only indirectly involved in a software project. They can provide feedback and solutions to technical problems that the internal stakeholders encounter in the software projects they are working on.

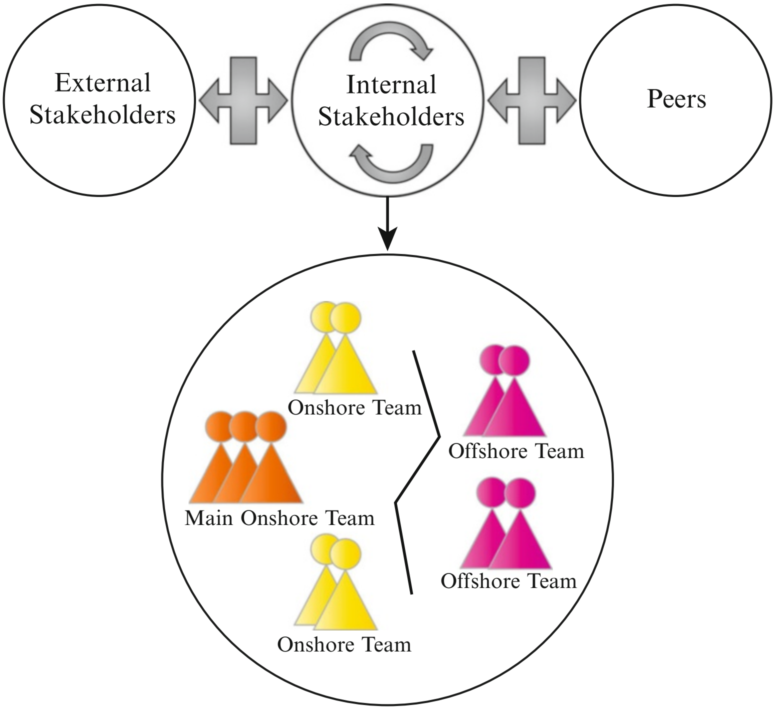


Fig. 16.1 Types of interactions in software projects

16.3 Social Aspects of Software Projects

The very nature of software projects necessitates social behaviors. According to Begel et al. (2010), the model of teaming from Tuckman and Jensen (1977) can be applied to the software life cycle to explain the social aspects of software projects. During the early phases of a software project, internal stakeholders organize into teams. This is known as *forming*. They then need to come to consensus about their goals through *storming*. They then choose and implement their software methodology and engineering processes which is known as *norming*. Stakeholders collaborate and coordinate the tasks among themselves to create a new product through *performing*. At the end of the project, *adjourning* enables them to reflect on their successes and failures in order to improve the next project's execution and outcome.

Interactions between internal and external stakeholders can involve the same social interaction and processes. These interactions may not always be directly related to coding and implementation. The external stakeholders are identified by the internal stakeholders through forming and storming during which the requirements of the external stakeholders are elicited and prioritized or milestones are agreed. During performing, external stakeholders test the deliverables to determine if they have met their expectations and the adjourning process involves any post-implementation reporting that helps to determine the success or failure of a project.

Internal stakeholders and peers interact mostly for the exchange of ideas and current practices, and these interactions usually occur continuously and are not restricted by a specific software project. However, these interactions are more likely to occur during the design and implementation phases of the software development life cycle. During these interactions, peers and internal stakeholders exchange knowledge on best practice, technical issues and latest trends in their fields of interest. This can be a great motivating factor for software projects (see Chap. 10).

These interactions that enable and facilitate the software development activities require the stakeholders to be able to find and connect with other stakeholders who have similar aims in the project or who have complementary skills.

16.4 Importance of Social Media in Software Projects

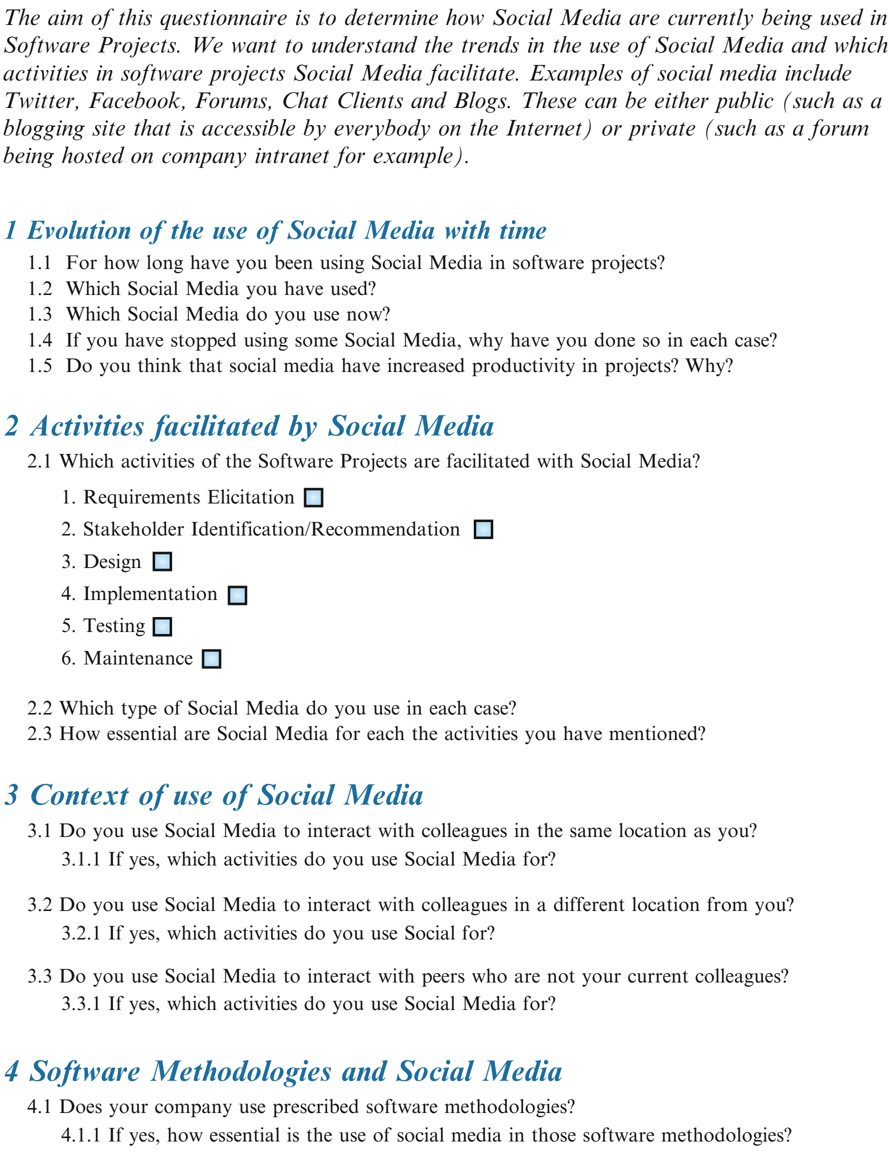
The nature of software projects has changed over time. Development is now often done in teams which are both onshore and offshore (Carmel and Agarwal 2001). In 2008, the total global IT outsourcing was estimated to be worth between $220 billion and $250 billion with a forecasted growth of 6–9 % per annum to reach around $380 billion by 2013 (Oshri and Kotlarsky 2010). This emphasizes the need for successful completion of projects as the incurred losses in case of failure can be considerable. Social media are one of the key enablers for any projects that involve distributed teams in general.

16.5 Pilot Study

The pilot study was conducted over a period of 2 months with nine companies. We contacted managers who had an overview of the software development process in their respective companies. Data collection was performed using a structured questionnaire with both open-ended and close-ended questions. Each individual had to answer 16 questions about their past and current use of social media in software projects as well as the software development methodologies that are used in their companies. The questionnaire is included in Fig. 16.2.

16.5.1 Data Collection and Analysis

Our underlying motivation was to investigate the use of social media in global software projects. We wanted to find out what kind of social media are being used and how they are being used to better understand the evolution of social media and



The aim of this questionnaire is to determine how Social Media are currently being used in Software Projects. We want to understand the trends in the use of Social Media and which activities in software projects Social Media facilitate. Examples of social media include Twitter, Facebook, Forums, Chat Clients and Blogs. These can be either public (such as a blogging site that is accessible by everybody on the Internet) or private (such as a forum being hosted on company intranet for example).

1 Evolution of the use of Social Media with time

1.1 For how long have you been using Social Media in software projects?
1.2 Which Social Media you have used?
1.3 Which Social Media do you use now?
1.4 If you have stopped using some Social Media, why have you done so in each case?
1.5 Do you think that social media have increased productivity in projects? Why?

2 Activities facilitated by Social Media

2.1 Which activities of the Software Projects are facilitated with Social Media?

1. Requirements Elicitation □
2. Stakeholder Identification/Recommendation □
3. Design □
4. Implementation □
5. Testing □
6. Maintenance □

2.2 Which type of Social Media do you use in each case?
2.3 How essential are Social Media for each the activities you have mentioned?

3 Context of use of Social Media

3.1 Do you use Social Media to interact with colleagues in the same location as you?
3.1.1 If yes, which activities do you use Social Media for?

3.2 Do you use Social Media to interact with colleagues in a different location from you?
3.2.1 If yes, which activities do you use Social for?

3.3 Do you use Social Media to interact with peers who are not your current colleagues?
3.3.1 If yes, which activities do you use Social Media for?

4 Software Methodologies and Social Media

4.1 Does your company use prescribed software methodologies?
4.1.1 If yes, how essential is the use of social media in those software methodologies?

Fig. 16.2 Pilot study questionnaire

to predict future trends. The participants were practitioners drawn from nine different organizations as shown in Table 16.1. The participants were all involved in global software development and included managing directors, project managers and software engineers.

The organizations were chosen because they were known to perform software engineering projects and to use social media. Initially, the organizations were

Table 16.1 Characteristics of participating organizations

Organization	Application domain	No. of staff on site (approx.)
A	IT standardization and certification	5
B	Automobile R&D	85
C	Software services	20
D	Financial services	40
E	Software services	9
F	Socio-technical systems development	200
G	Management consulting services	>500
H	Software services	15
I	Software services	40

contacted by email to establish their suitability and willingness to participate. The participants were sent the questionnaire by email and were given 2 months in which to complete it. The participants were all experienced in collaborating on projects and using social media.

Data thus received from the participants were anonymized and encoded to extract information that was relevant to the study. We performed a qualitative analysis of the ensuing data to look for patterns of behavior and unexpected behavior as well as confirmations of expected behavior as far as the use of social media in software projects is concerned.

16.5.2 Results: Social Media and Software Projects

Social media provide many tools to support the processes involved in executing a software project. Social media can aid software development activities from the early phases such as requirements engineering through to development, testing and documentation. They also provide a means to facilitate maintenance activities. Since social media are highly flexible, stakeholders can readily adapt them to fit their needs to help deliver successful software projects. Our survey shows that all the phases of the software development life cycle can involve the use of social media as illustrated in Fig. 16.3.

Two main characteristics of social media determine how they are used in software projects. These are *social presence* (Short et al. 1976) and *media richness* (Daft and Lengel 1986). Social presence is the acoustic, visual, and physical contact that users can achieve using social media; that is, it is the level of awareness that users have of each other during communication. Social media with high social presence support synchronous and personal interactions and increase the influence that the users have on each other. Social platforms that enable live web chats are examples of such social media. On the other hand, social media with low social presence favour asynchronous and more formal interactions (Kaplan and Haenlein 2010). Forums are a good example of such social media.

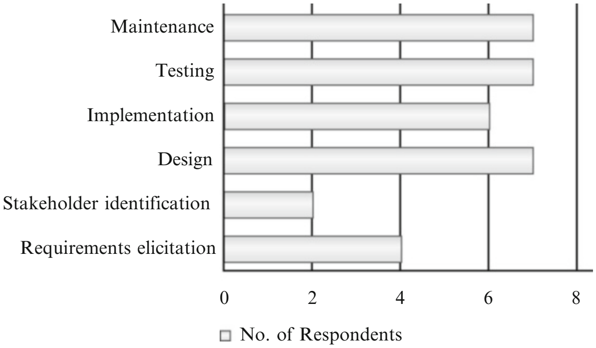


Fig. 16.3 Use of social media in the software project life cycle

Media richness is the volume of information social media can exchange in a given time interval. Thus, social media with high media richness will use a number of cues to increase clarity of the information being exchanged and actively facilitate feedback. High media richness therefore implies low ambiguity and uncertainty in the information being transmitted. For example, explaining a solution via a web chat is likely to have lower ambiguity and uncertainty than explaining it using posts on a forum or by email. Social presence and media richness determine the appropriateness of a given social media in the different stages of the software development life cycle.

From our survey, we found that although social media have been used in software projects since the introduction of web 2.0, there has been a relatively high rate of adoption of social media in the last decade, similar to that which occurred at the time of the birth of popular social media platforms such as Wikipedia, Facebook and Twitter. The most popular social media used include (unsurprisingly) Facebook, Twitter, instant messengers, LinkedIn and blogs. Figure 16.4 shows our findings on the use of different social media in the software project life cycle.

We found that currently the design, testing and maintenance phases top the list for the use of social media with the largest variety of social media being involved during design. Social media use is least popular in stakeholder identification/ recommendation and requirements elicitation activities. Software teams use Skype™ and forums in all phases of the software projects they are involved in. Twitter tends to be used in maintenance, stakeholder identification and recommendation, design and maintenance while instant messengers and blogs tends to be used during design, implementation, testing and maintenance. Facebook is used only to elicit requirements and stakeholder analysis while LinkedIn has been identified as being useful mainly in the implementation phase of software projects. An interesting observation here is that although requirements elicitation and stakeholder identification/recommendation activities involve a lot of interaction among stakeholders, they lag behind in the adoption of social media.

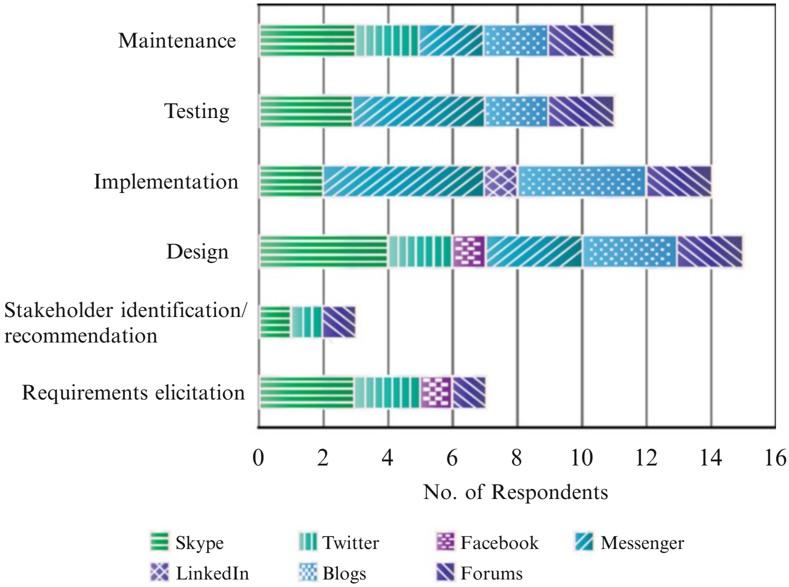


Fig. 16.4 Types of social media used in the different phases of software projects

The popularity of instant messaging in general and Skype™[11] in particular, which provide both high social presence and high media richness, can be explained by the fact that some of the tasks during software projects require very intense interactions among the stakeholders. One such example is when developers need a quick solution to a critical technical problem they encounter during implementation; they may need and want to discuss the problem urgently in detail and interact with technical experts in the team.

Forums enable stakeholders and peers to communicate asynchronously with a low level of formality and interaction. Thus stakeholders tend to use these in situations which are not urgent. For example, forums are very useful for explaining generic solutions to recurring problems which can be used by other software teams within an organization.

Twitter is actively used to exchange information between stakeholders in software projects because it provides support for lightweight coordination and communication. Although the social presence and media richness is limited, the broadcast mechanisms implemented in Twitter are very efficient in reaching a large audience.

Blogs are frequently used by stakeholders and peers to document technical information, to discuss the release of new features and to support requirements engineering. Blogs are useful in these cases because they encourage discussions and

[11] http://www.skypeTM.com/en/.

are accessible by a large number of people who can participate whenever they have time to do so.

Facebook and LinkedIn allow the creation of virtual communities in which stakeholders and peers can share views and information about specific topics related to software projects. Facebook can help stakeholders to recommend other stakeholders by inviting them to the software project pages.

16.5.3 Results: Interaction Among Co-located stakeholders

Stakeholders who are co-located may use social media to communicate with each other despite being in the same location. This is particularly likely for large organizations with many large teams. Here, stakeholders use social media to keep up-to-date with what other teams are doing and to facilitate communication during software projects. Figure 16.5 illustrates how participants in our survey use social media to interact with co-located stakeholders.

Figure 16.5 shows that social media is not always used among co-located teams. However, when it is used, it predominantly supports project management activities. Senior team members tend to use social media platforms like wikis or instant messaging for tracking the progress of tasks in a project. Wikis are excellent facilitators of software development collaboration. Their adoption in software projects is widespread. Wikis are mainly used to support defect tracking, documentation, requirements tracking, and test case management and to create project portals. Tagging is used in software project portals such as Github[12] to tag issues, project releases, work items and builds. The main advantages of using tags in software projects are their flexibility and their lightweight, bottom-up nature.

Instant messengers are extensively used among team members for coordination of tasks. For example, in our study, we found that in projects which involve many interdependent modules, team members use instant messaging to inform each other about implementation progress. Our findings on this extensive use of chat messaging are similar to those of Dittrich and Giuffrida (2011).

Social media are also widely used for knowledge sharing purposes among co-located teams. Our study has shown us that this is particularly true for large organizations where there are many teams in different parts of the same building. Members of the different teams use forums to maintain awareness of the on going work. Forums are also used to inform other teams about novel solutions to problems or new technology that has proved to be useful in projects. In more urgent situations, stakeholders also use instant messaging or IRC[13] to ask for help or clarifications on possible solutions when they encounter problems. Again our findings support those of Dittrich and Giuffrida (2011).

[12] https://github.com/.

[13] http://www.mirc.com/jarkko.html.

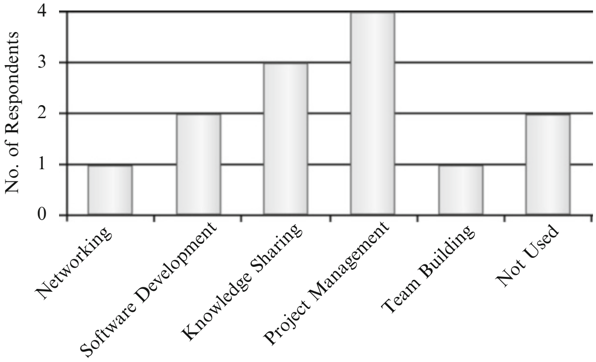


Fig. 16.5 Types of interaction among co located stakeholders

In projects which involve in-house software application development, social media are used to interact with external stakeholders to elicit requirements and for validation and testing purposes. Some of the teams in our survey used Skype™ to interact with the external stakeholders in these cases. Others used MSN messenger[14] which enables desktop sharing to help end users with live demonstrations.

Our study found that social media are also useful for networking and team building for co-located stakeholders, who tend to use social media to connect to other team members to keep in touch with them. Social media also facilitate the less formal interactions that are essential to team building. For example, team members use instant messaging and forums to joke and socialize. We found evidence for this in our survey, which revealed that co-located stakeholders interact with each other in two ways. First, they communicate in a *formal* way with each other about urgent technical issues that require quick responses, and secondly they communicate in an *informal* way to share new ideas, or (in a more social capacity) to team build or network. The importance of both formal and informal reporting is also reported in Black et al. (2010) and in Chap. 10. We found that instant messengers in general and Skype™ in particular are the preferred social media in the first case as they have high social presence while blogs and forums are widely used in the second case.

16.5.4 Results: Benefits and Limitations of Social Media in Co-located Interactions

According to our study, social media is very beneficial to software projects. The fact that social media enable stakeholders to reach a very large audience is one of the major benefits. Team members can easily interact through social media with other colleagues who are beyond their own physical "cluster," perhaps being

[14] http://windows.microsoft.com/en-GB/messenger/home.

located on a different floor of the building or in a different common area. Thus, social media has broken physical boundaries among stakeholders in the same location facilitating exchange of information among a potentially large number of co-located stakeholders.

Another benefit of social media in this context is that it enables team members who are normally less likely to interact with others to do so. This is particularly true for those team members who may find it hard to approach other team members for help or advice.

However, a concern about the use of social media is the ease with which stakeholders can shift from work-oriented interactions to more personal ones. One of our survey participants mentioned that social media were not used for co-located interactions because management feared that team members would spend time on unproductive interaction. Social media are not infallible. For example, since there is less regulation on the types of interactions permitted among co-located stakeholders, harmful activities such as trolling (in which team members harass others by publishing inappropriate personal information) can occur, resulting in a decrease in performance and a decrease in motivation for the victims (Matthews and Stephens 2010). As pointed out in Chap. 10, motivation plays an important role in the production of high-quality software.

Through the use of social media, team members who are popular may become powerful and hence have greater influence on others. This is especially true in the case of asynchronous social media such as blogs (Twitter, Facebook, etc.) and often happens when stakeholders flood these media with an excess of information. For example, if a stakeholder frequently posts irrelevant information on Twitter, other connected stakeholders may become overwhelmed and consequently inadvertently overlook other relevant information from an infrequent poster. The danger is that very active team members can persuade others to accept solutions or technologies that may not necessarily be the best solution for the project. The reverse may also occur: a stakeholder who has very good ideas and suggestions for the project may be completely ignored if their social media activity is overshadowed by very active stakeholders (Matthews and Stephens 2010).

16.5.5 Results: Interactions Among Stakeholders in Distributed Locations

Social media are particularly popular when software project teams are globally distributed. All the teams in our study use social media actively to interact with stakeholders who are not in the same location as they are. These can be internal stakeholders such as team members who are working in offshore or onshore offices and external stakeholders who are not in the same location as the main software development team. The main software development team is assumed to be located at the premises where the project is being managed. Figure 16.6 illustrates the

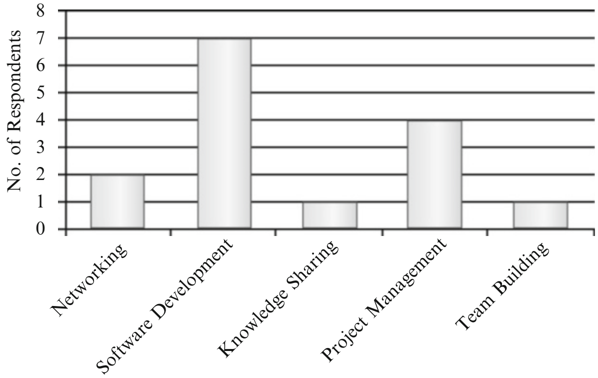


Fig. 16.6 Types of interactions among distributed stakeholders

outcome of our study concerning how social media are used among distributed stakeholders. This shows that as expected, the main purpose of social media is to facilitate software development activities. The actual activities depend on the role of the stakeholders in the projects.

In the case of internal stakeholders, social media facilitate each phase of the software development process from design through to maintenance. Instant messengers including Skype™ are used to discuss important issues regarding design and implementation. Before commissioning a release of the software, there is in-house testing. During this exercise, testers record any bugs or feature improvements on forums and blogs and discuss these with other team members using Skype™ and instant messenger clients. During maintenance, the different teams involved in the software project use social media to decide whether new features or enhancement of existing features are needed. In this phase, we once again found instant messengers (including Skype™) to be the most popular means of communication.

Project managers from onshore teams communicate with offshore teams via social media to follow progress and discuss issues during the projects. Our study found that teams which operate with this configuration have regular meetings via instant messengers including Skype™ to inform teams in other locations about how the project was progressing, discuss task allocations and negotiate schedules of delivery. We found that these meetings are held daily or weekly depending on the needs of the projects.

One of the drawbacks of working with offshore teams is the difficulty of interaction which is exacerbated by distance. People in different locations do not have the opportunity to have informal meetings or interactions as discussed in Chap. 9. This can greatly impact team building and motivation. This can be overcome (at least in part) by using social media. For example, members from some of the teams we surveyed use instant messaging and IRC to have less formal conversations and to get to know other team members better. The ability to initiate conversation among team members who have not been introduced to each other is an important success factor for global projects as it greatly facilitates communication.

Our study has shown that internal stakeholders and external stakeholders who do not share the same physical location frequently interact using social media for software development purposes. Internal stakeholders elicit requirements from external stakeholders using Skype™, Facebook, forums and blogs. Such social media also support the process of recommending stakeholders that the project team may need. During testing, external stakeholders can use forums and blogs to report bugs and request new features. This also happens during the maintenance phase. According to our study, external stakeholders are often informed about the state of the project via social media. Project managers from the software team tend to use Skype™ to communicate regularly with external stakeholders about progress, changes to milestones and delivery dates for the project.

Interactions between stakeholders at different locations can require a quick response (typically immediate solutions to design and implementation issues). This is also true when stakeholders are negotiating or providing their requirements for the project. Thus, the use of social media such as instant messengers including Skype™ ensures that the right social presence and media richness is achieved. The parties involved in these activities need to be able to exchange a large amount of information to ensure that there is no confusion or misunderstanding about what is being discussed. However, there are some interactions which can occur asynchronously without the need for media richness. Examples of such interactions in this context include bug reports and feature requests. In this case, since we require a lower social presence and media richness, social media such as forums and blogs are appropriate.

16.5.6 Results: Benefits and Limitations of Social Media in Distributed Interactions

Social media can be extremely beneficial for interactions among distributed stakeholders (Begel et al. 2010). Many of the challenges that are related to distributed software projects can be ameliorated to some extent by social media. Geographical distance whether small or large has a great impact on the ability of teams to collaborate successfully (Olson and Olson 2000).

One of the major issues that social media helps to address is the difficulty of coordination due to independencies and process non-conformities (Herbsleb 2007, Mockus and Herbsleb 2001). Our study has found that stakeholders in different locations actively use social media to coordinate software implementation. Using social media they are able to communicate any interdependencies that may exist among the components of the software being developed very effectively (Herbsleb and Grinter 1999). The fact that they use social media also helps to align processes in the project. For example, if the stakeholders use specific terminologies in a given process in the development life cycle, they can discuss this and use agreed terminology that all parties will understand. Slips in schedules (Herbsleb 2007, Mockus and Herbsleb 2001) can also be detected earlier limiting the consequences these may have on other participating teams and stakeholders. Since project managers use social media to do task allocations and follow-ups, any delays can be discovered at an early stage and dealt with as necessary.

Other challenges in distributed software development are mostly associated with offshore software development, mainly because communication difficulties are a major problem. The different cultures and languages of some stakeholders may lead to misunderstandings (Herbsleb 2007), as also discussed in Chap. 10. Furthermore, rapidly changing teams and projects in this context can mean that stakeholders do not have time to form relationships with other stakeholders, as also discussed in Chap. 12. Our study showed that social media can greatly help in these cases. For example, interacting with a colleague via a video channel can improve understanding through attention to additional subtle cues such as voice tone and facial expressions. Social media also enable stakeholders who have worked together on projects (even for a brief period) to keep in touch via blogs. These stake-holders can eventually become peers who can act as consultants on future projects.

Social media are also useful when managing unstable projects where informal communication is essential (Galbraith 1977, Kraut and Streeter 1995). For example, in a project where requirements are always changing, social media can support distributed stakeholders by providing them with a rich platform to discuss and interact with each other to resolve issues and project risks. With the help of forums and blogs, stakeholders can also become acquainted with the experts on the various subjects related to the project across the different sites and can thus reach the right expert quickly when necessary (Sengupta et al. 2006).

Distributed teams with unstable projects can be managed by using social media in a more formal way in which all interactions are *self-moderated* by stakeholders. In such a formal setting, trolling and flooding are less likely to occur (Daft and Lengel 1986). However, our study found that, as with co-located teams, there is still the risk that using these social media may give rise to less productive work. This can be especially harmful in off-shore teams as billing is done by the hour or day spent on the project. Onshore companies or clients will only pay for additional time spent on a given task if this can be justified.

As with all technology, users need to be motivated to use social media. This is the case for the use of social media in offshore teams where some stakeholders cannot see the immediate benefit of adopting social media in their day-to-day tasks and so consider it to be an overhead in their work which is hard to justify in terms of cost. Such non-conformity in the use of social media can mean that social media are regarded as a burden because they may not provide complete information to the stakeholders who are actively trying to use them.

Another issue that can arise specifically from the use of asynchronous social media is the fact that information can become out-of-date before any attention is given to it or information may be duplicated. Our study found that for the testing phase (for example) users tend to report the same bug in different ways many times.

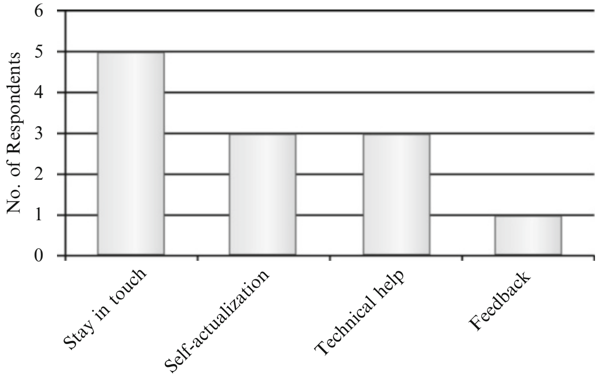


Fig. 16.7 Types of interactions among stakeholders and peers

This adds an overhead to the project as project managers need to analyse each bug report and determine whether the bug has been reported previously.

16.5.7 Results: Interaction with Peers External to the Office

Interactions with peers can have a considerable impact during software projects. The teams in our survey define these peers as being past project team members, colleagues, friends and academics. These peers provide a valuable source of knowledge and information that stakeholders can consult in order to keep in touch with best practice, technologies and trends. Figure 16.7 illustrates how our participants use social media to interact with their peers.

Networking is the main reason for using social media to interact with peers. In our study, we found that internal stakeholders use social media frequently to keep in touch with peers. These relationships have been forged by previous experiences and need to be maintained somehow when the parties are no longer interacting on a day-to-day basis. Social media such as LinkedIn and Facebook enable colleagues to keep in touch by providing updates about events that they post online.

Some of the participants in our survey mentioned using social media to communicate with peers for self-actualization purposes. Since the field of IT is so vast and new technologies abound, it can be hard to be constantly aware of new tools and techniques. Stakeholders thus use blogs maintained by peers to discuss new technologies. Our study found that stakeholders also use social media to seek technical help from their peers. Often, when they come across an issue in the project they are working on, they either post a question on their own blogs and wait for responses from peers or find forums online where experts can help. Stakeholders also use social media such as blogs and forums to provide reviews on technologies, hardware and methodologies they have used in previous software projects.

These interactions are often done in the stakeholders' spare time and are very informal. Such interactions require low social presence and a reduced level of media richness. Thus blogs and forums are the most common social media used for this. We also found that some teams use instant messaging and Skype™ at times when they need urgent technical help in projects if relevant peers can be contacted via these media.

16.5.8 Results: Benefits and Limitations of Social Media in Interactions with Peers

The main benefit of interactions between internal stakeholders and their peers is the fact that they can access a wealth of information that can be the key to the success of the software project. By staying in touch with the latest tools and techniques in the field of IT, these stakeholders are able to bring innovation into the process of software development. Furthermore, project managers and team members are able to access a larger audience when seeking help. This can lead to numerous solutions for any problems which arise. Managers can then choose the one that best suits their purpose.

However, if stakeholders spend too much time on such informal interactions, this can impact their productivity. The information available from peers may not be entirely appropriate or correct, and if stakeholders rely on it and use it unquestioningly in software projects they may introduce more problems than they solve.

16.5.9 Results: Impact of Social Media on Productivity in Software Projects

Our study found that social media led to an increase in productivity for all our survey participants. The main reason for this is the fact that social media such as Skype™ enables stakeholders to get responses almost immediately to any queries and on-going discussions. Thus, despite a distributed team, stakeholders can set up meetings almost instantaneously and resolve issues quickly. This means that the teams can make the most of the time allocated to the project and so improve productivity.

In many social media, the basic features such as blogging, instant messaging and video calls are free for use. Furthermore, most social media enable some level of customization. These features are the ones that stakeholders involved in software projects desire most frequently. The low cost and easy customization of social media imply that software engineering companies using them have a set of very powerful tools that are virtually free. Our study confirms that telephone calls and faxes among distributed software teams can be completely eliminated in most cases. This decreases the overhead costs of projects considerably.

16.5.10 Results: Social Media as a Facilitator in Software Projects

Half of the participants in our study reported that social media are useful when applying the software methodologies prescribed by their organization or the organization of their customers. The methodologies mentioned included agile methods (see Chap. 11) and CMMI.[15] These necessitate good planning, control and monitoring. Such activities can be very tedious to apply across distributed teams as project managers need to get regular updates on the progress of the projects to be able to perform these tasks. Social media provide a convenient way to achieve this. For example, in projects where teams use blogs to provide status updates and make task allocations, project managers can easily be informed about the state of projects by checking status updates and task completion progress.

16.5.10.1 Social Media in the Agile Context

Our study revealed that social media is very popular with all the agile practitioners who responded to our questionnaire. For example, for many of these practitioners, social media are useful for daily stand-up meetings or for demonstrations of prototypes in rapid prototyping.

Agile methods involve working with requirements that are often immature and ambiguous. Short development cycles and efficient communication help to manage such requirements to reduce the likelihood of defects (Abrahamsson et al. 2002; Beck and Andres 1999). Communication and feedback among the stakeholders of an agile project are particularly crucial for its success. For example, the most common agile practice, extreme programming, advocates the practice of real customer involvement where stakeholders take part in weekly or quarterly meetings for exchange of information and planning (Beck and Andres 2004). Social media are very good tools to promote these kind of activities in the agile context.

Agile methods encourage face-to-face communication to enable the stakeholders to share as much information as possible; this can be helped by facial expression or other cues as well as verbal communication. Thus, in this context, the usefulness of the social media tends to increase with the level of media richness and social presence it provides. For example, according to Wallace et al. (2002), videocon-ferencing is a very good alternative when stakeholders are not available on site for user story explanation. Korkala et al. (2006) further elaborates on this to

[15] http://cmmiinstitute.com/resources/.

include all instances where effective communication with a remote stakeholder is required.

Social media provide the additional benefit of leaving logs and traces. This can be extremely useful in the rapidly evolving agile context to provide a record of how a project has progressed over time. Such information can be hard to document methodically, especially in large projects with very short development cycles. For example, one of our participants who uses an agile process stated that keeping logs of chats has proved to be very useful when negotiating and justifying billing with customers. Social mining techniques can be used with the resulting logs and traces to get insights into the project dynamics and trends as discussed later in Sect. 16.6.

16.5.10.2 Social Media and the Project Manager

Project complexity and uncertainty are the main challenges that project managers face (Chaps. 11 and 13). This is especially true in the context of distributed software development. Social media can empower project managers to manage these complexities and uncertainties both proactively and reactively as they arise in a project. However, our understanding of the role of social media for the management of software projects is far from complete (Storey et al. 2010).

Large distributed software teams involve challenges arising from temporal, geographical and socio-cultural distance (Holmstrom et al. 2006) that project managers need to overcome. Social media help to coordinate and motivate these teams by promoting communication among the different stakeholders. Project managers can thus have better control of the progress of the individual team members as well as the project as a whole and so can better report to other stakeholders on the status of the project. For example, one of the participants of our study who works with distributed teams emphasized the usefulness of weekly meetings with offshore team members via Skype™ to plan and assign tasks to the individual team members and set milestones. These weekly meetings also enabled team members to raise any issues that they have encountered or that might arise. Such interactions enable project managers to efficiently tackle these issues to decrease their impact on the project. Our findings are again similar to those of Dittrich and Giuffrida (2011).

16.5.11 Threats to Validity

The validity of this study is limited in a number of ways (Wohlin et al. 2012). The threats to construct, internal and external validity are discussed here.

Construct validity asks whether the variables used in the study accurately measure the concepts they purport to measure. The main threat is posed by participants who may not undertake the survey with sufficient care. For example, participants who are short of time may have simply always chosen to tick the first box presented.

However, we believe that all participants were well motivated as they were not under any duress to participate. Consequently, it seems likely that the data collected have been submitted accurately and in good faith.

Internal validity is concerned with whether the relationship between the survey outcomes and the use of social media is a causal relationship rather than one which occurred simply by chance. Survey instruments are always subject to a query concerning interpretation: did the participants interpret the questions as they were intended? We had performed trials with the survey within our research group and improved the survey as a result. It is very difficult to determine whether or not every participant interpreted every question in exactly the way that we intended. However, we believe that we did constrain possible misinterpretations of the questions as far as possible.

External validity asks whether the results can be generalized. We must acknowledge that the results of our survey are limited due to the small number of companies surveyed. In the fullness of time, we hope to repeat this survey with a larger sample. It is also possible that the element of selection (via personal contact) has introduced response bias. However, the personal contacts were often only used in the first instance; the participants who completed the survey were usually unknown to the authors. Our response rate was 39 %, which is not unreasonable for such a survey.

16.6 The Future of Social Media in Software Projects

Collaborative platforms are increasingly being used by software teams. Collaborative development environments (CDEs) provide teams with the tools to program, debug, refactor and reuse code together with the collaboration tools needed for distributed software development (Lanubile et al. 2010). These collaboration tools usually include social media such as wikis, instant messenger clients and forums. Examples of CDEs include the Jazz CDE,[16] Redmine[17] and Microsoft Team Foundation Server.[18] Jazz allows developers to perform the usual tasks in software development and adds a web interface to allow developers to interact with other stakeholders via a chat client and access other collaboration information such as bug tracking and project status reports.

Microsoft Team Foundation Server is very similar and enables stakeholders to track a work item or collaborate using a project portal while Redmine provides a web interface where stakeholders can manage projects and collaborate using feeds, wiki and forums. Feeds can provide subscribers with aggregated updates from

[16] http://www-01.ibm.com/software/rational/jazz/.

[17] http://www.redmine.org/.

[18] http://msdn.microsoft.com/en-gb/vstudio/ff637362.aspx.

websites with the latest content. In software projects, feeds are an efficient way to provide awareness about workspaces, developers and processes.

With the advent of disciplines such as social media analytics (Zeng et al. 2010) and social media data mining (Sufian and Anantharaman 2011), we expect social media to play a larger and more important role in software projects. Social data mining identifies situations in which groups of people are generating electronic records of documents such as posts, instant message logs and blogs among other documents as part of their normal activity. Potentially useful information implicit in these records can be identified via specific automated computational techniques which harvest and aggregate this information. The results of these operations are then presented for use in a meaningful way (Terveen and Hill 2001).

Since social media are widely used in software development projects, mining the resulting data can provide valuable insights concerning the most important stakeholders in these projects. This can enhance the ability of social media to help other stakeholders identify experts involved or potentially involved in a project. This kind of mining has been successfully applied to code and emails for Postgres (Bird et al. 2006) where information extracted was used to demonstrate who the more active participants were in the project.

Crowdsourcing is an interesting phenomenon (O'Reilly 2007) that has great potential as it enables large numbers of stakeholders to provide requirements for new features and to provide feedback on bugs. Crowdsourcing is defined as the act of a company or institution of taking a function once performed by employees and outsourcing it to an undefined and usually large network of people by issuing an open call. The function can be performed by peer-production (in which case the job is performed collaboratively) or by sole individuals. The essential prerequisite is the use of the open call format and the large network of potential workers (Howe 2006).

Companies are using sites such as Topcoder[19] to outsource coding tasks to the constituency of developers worldwide. Similarly, some requirements elicitation and stakeholder recommendations are outsourced using tools like Stakesource (Lim et al. 2011) and OneDesk[20], respectively. In this new context, social media will become critical for the success of the software projects. Companies who will be crowdsourcing tasks in the software development life cycle will have a more compelling need to communicate with the successful bidders for the tasks. These bidders can be located in any part of the world. Thus, project managers will need to be able to communicate both synchronously (in urgent situations) and asynchronously to make sure that process alignments are performed and the bidders understand what the requirements are. They also need to be able to properly monitor and coordinate projects as such models of operation have many risks associated with them. Social media are excellent as facilitators in this case as they can provide all the required tools for these project management activities.

[19] http://www.topcoder.com/.

[20] http://www.onedesk.com/.

16.7 Conclusions

Social media have already proved their usefulness in software projects in the today's changing world. They help project managers to overcome project management difficulties that arise from the relatively new context of distributed software development while providing other project stakeholders with the tools they need to interact successfully with each other. Social media have also brought about a new way of working where knowledge acquired can be shared with other teams or peers and help can be obtained not only from individuals within the organizations involved in the projects but also from external peers who are members of the wider networks of the team members, effectively crowdsourcing new knowledge. New applications for social media are being discovered on a daily basis. These include social media data mining which can be used to find implicit useful information from the data exchanged using social media. As the very nature of software projects evolves with time, social media will also find new applications in these new project configurations. However, despite their numerous advantages as project management facilitators, social media can still present challenges to software projects if they are not used within a properly regulated framework.

Acknowledgments The authors would like to thank all of the participants in the survey reported here. This research is funded by the Faculty of Technology, Design and Environment at Oxford Brookes University. The authors are very grateful for this support.

References

Abrahamsson P, Salo O, Ronkainen J, Warsta J (2002) Agile software development methods—review and analysis. VTT Publications, p 107. http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf. Last accessed Aug 2013

Beck K, Andres C (1999) Extreme programming explained: embrace change, 1st edn. Addison Wesley, Boston, p 224

Beck K, Andres C (2004) Extreme programming explained: embrace change, 2nd edn. Addison Wesley, Boston

Begel A, DeLine R, Zimmermann T (2010) Social media for software engineering. In: Proceedings of the FSE/SDP workshop on future of software engineering research—FoSER'10, 2010, p 33

Bird C, Gourley A, Devanbu P, Gertz M, Swaminathan A (2006) Mining email social networks. In: Proceedings of the 2006 international workshop on mining software repositories—MSR'06, p 137

Black SE, Harrison R, Baldwin M (2010) A survey of social media use in global systems development. In: Proceedings of the 1st workshop on Web 2.0 for software engineering, Web2SE, ACM/IEEE ICSE 2010, pp 1–5

Carmel E, Agarwal R (2001) Tactical approaches for alleviating distance in global software development. IEEE Softw 18(2):22–29

Daft RL, Lengel RH (1986) Organizational information requirements, media richness and structural design. Manage Sci 32(5):554–571

Dittrich Y, Giuffrida R (2011) Exploring the role of instant messaging in a global software development project. In: 6th IEEE international conference on global software engineering (ICGSE). IEEE

Eckler P, Worsowicz G, Rayburn JW (2010) Social media and health care: an overview. PM R 2(11):1046–1050

Galbraith J (1977) Organization design. Addison Wesley, Reading, p 426

Hawn C (2009) Take two aspirin and tweet me in the morning: how Twitter, Facebook, and other social media are reshaping health care. Health Aff (Millwood) 28(2):361–368

Herbsleb JD (2007) Global software engineering: the future of socio-technical coordination. In: Future of software engineering (FOSE'07), pp 188–198

Herbsleb JD, Grinter RE (1999) Splitting the organization and integrating the code. In: Proceedings of the 21st international conference on Software engineering—ICSE'99, pp 85–95

Holmstrom H, Conchuir E, Agerfalk P, Fitzgerald B (2006) Global software development challenges: a case study on temporal, geographical and socio-cultural distance. In: 2006 I.E. international conference on global software engineering (ICGSE'06), pp 3–11

Howe J (2006) The rise of crowdsourcing. Wired Mag 14(6):1–4

Kaplan AM, Haenlein M (2010) Users of the world, unite! The challenges and opportunities of Social Media. Bus Horiz 53(1):59–68

Kavanaugh AL, Fox EA, Sheetz SD, Yang S, Li LT, Shoemaker DJ, Natsev A, Xie L (2012) Social media use by government: from the routine to the critical. Gov Inf Q 29(4):480–491

Korkala M, Abrahamsson P, Kyllonen P (2006) A case study on the impact of customer communication on defects in agile software development. In: AGILE (AGILE'06), pp 76–88

Kraut RE, Streeter LA (1995) Coordination in software development. Commun ACM 38(3):69–81

Lanubile F, Ebert C, Prikladnicki R, Vizcaino A (2010) Collaboration tools for global software engineering. IEEE Softw 27(2):52–55

Lim SL, Damian D, Finkelstein A (2011) StakeSource2.0: using social networks of stake- holders to identify and prioritize requirements. In: Proceeding of the 33rd international conference on Software engineering—ICSE'11, p 1022

Mangold WG, Faulds DJ (2009) Social media: the new hybrid element of the promotion mix. Bus Horiz 52(4):357–365

Martín-Blas T, Serrano-Fernández A (2009) The role of new technologies in the learning process: Moodle as a teaching tool in physics. Comput Educ 52(1):35–44

Matthews P, Stephens R (2010) Sociable knowledge sharing online: philosophy, patterns and intervention. In: Aslib proceedings. Emerald

McNab C (2009) What social media offers to health professionals and citizens. Bull World Health Organ 87(8):566

Mockus A, Herbsleb J (2001) Challenges of global software development. In: Proceedings seventh international software metrics symposium, pp 182–184

O'Reilly T (2007) What is Web 2.0: design patterns and business models for the next generation of software. Commun Strat First Quarter(1):17

Olson G, Olson J (2000) Distance matters. Hum Comput Interact 15(2):139–178

Oshri I, Kotlarsky, J (2010) Realising the real benefits of outsourcing: measurement excellence and its importance in achieving long term value. In: Global sourcing of information technology and business processes, vol 55, pp 250–270

Sengupta B, Chandra S, Sinha V (2006) A research agenda for distributed software development. In: Proceeding of the 28th international conference on software engineering—ICSE'06, p 731

Short J, Williams E, Christie B (1976) The social psychology of telecommunications. Wiley, Hoboken, p 206

Storey MA, Treude C, van Deursen A, Cheng LT (2010) The impact of social media on software engineering practices and tools. In: Proceedings of the FSE/SDP workshop on future of software engineering research—FoSER'10. ACM Press, New York, p 359

Sufian A, Anantharaman R (2011) Social media data mining and inference system based on sentiment analysis, Chalmers University of Technology

Terveen L, Hill W (2001) Beyond recommender systems: helping people help each other, HCI in the new millennium. Addison-Wesley, New York, pp 487–509

Tuckman BW, Jensen MAC (1977) Stages of small-group development revisited. Group Organ Manage 2(4):419–427

Wallace N, Bailey P, Ashworth N (2002) Managing xp with multiple or remote customers, Proceedings of the 3rd international conference on extreme programming and agile processes in software engineering (XP2002), pp 134–137

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering. Springer, Berlin

Zeng D, Chen H, Lusch R, Li SH (2010) Social media analytics and intelligence. IEEE Intell Syst 25:13–16

Biography Rachel Harrison is Professor of Computer Science in the Department of Computing and Communication Technologies at Oxford Brookes University. Previously she was Professor and Head of Department at the University of Reading, and before that she was a member of Faculty in the Department of Electronics and Computer Science at the University of Southampton where she gained her PhD. Her research is concerned with empirical and automated software engineering and includes work on metrics, evolution, requirements, and search-based software engineering.

Varsha Veerappa is a postdoctoral research assistant in the Department of Computing and Communication Technologies at Oxford Brookes University. She completed her PhD in Computer Science at University College London (2008–2012). Prior to this she worked in industry in various roles from project manager to systems analyst and programmer. Her research interests include requirements engineering, search-based software engineering, data mining, natural language processing and software metrics.

Chapter 17
Process Simulation: A Tool for Software Project Managers?

Dietmar Pfahl

Abstract Process simulation has been introduced as a tool in support of software project management more than 25 years ago. Since then, it has been considered an approach with high potential for making software project managers' work more effective and successful. Unfortunately, despite high expectations and many reports on prototypical process simulation applications in industrial contexts, little evidence exists that process simulation has become an accepted and regularly used tool of software project managers. This chapter investigates the reasons for lacking impact of process simulation in the software industry. This is done with the help of an in-depth description of a software process simulation application example. The application example focuses on the effects of various workforce allocation strategies on project performance, expressed in terms of project duration, effort consumption, and product quality. With the help of the application example and based on existing literature, the gap between the current state of the art of software process simulation and the actual state of practice is described and its root-causes are discussed. The chapter concludes with a list of issues that need to be addressed in order to close the gap between the state of the art and the state of practice. Most of the issues relate to the difficulty of demonstrating a positive cost-benefit ratio when applying process simulation as a tool in support of software project management tasks.

D. Pfahl (✉)
Institute of Computer Science, University of Tartu, Tartu, Estonia
e-mail: dietmar.pfahl@ut.ee

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_17, © Springer-Verlag Berlin Heidelberg 2014

17.1 Purpose and Scope of Software Process Simulation

Software process simulation (SPS) is a method that aims at calculating and visualizing the behavior of software development process parameters over time on a computer. During the past 25 years, beginning with the pioneering work of Kellner and Hansen (1989), Abdel-Hamid and Madnick (1991), and several others in the USA and Europe, much research has been conducted on finding the best way of developing and using SPS models. For example, in Pfahl et al. (2006), the authors report the existence of more than 250 papers and articles related to the topic of SPS, published between 1987 and 2004. The systematic literature reviews by Zhang et al. (2008, 2010) and Bai et al. (2011) give insights into the content and quality of SPS-related publications including more recent years.

This chapter cannot give an introduction into the wide range of SPS modeling methods and tools nor can it give a comprehensive overview of the various types of SPS techniques. Regarding the diversity of SPS modeling approaches, the recent literature survey of Bin Ali and Petersen (2012) can serve as an entry point for the interested reader and a detailed description of a specific SPS modeling approach, that is, integrated measurement, modeling and simulation (IMMoS), can be found in Pfahl and Ruhe (2002). Regarding the various types of SPS techniques, the book chapter by Müller and Pfahl (2008) can be used as an introductory overview, while the excellent book by Madachy (2007) gives a comprehensive introduction into one of the most popular SPS techniques, that is, system dynamics (SD).

It should be noted, though, that the reader does not need detailed knowledge about the specific types of SPS techniques and the methods and tools available for developing and evolving SPS models in order to understand the examples and the following discussion presented in this chapter.

Figure 17.1 visualizes the main elements that play a role in SPS model development and application: input, executable SPS model, and output. Input for both model development and application consist of data and expert knowledge. During SPS model development, data are needed (1) to create static models, derived with the help of statistical analyses, which may help establish causal relationships between the various variables representing the internal structure of the SPS model, and (2) to calibrate the SPS model. During SPS model application, data are needed to feed the model with the requested values for its input parameters. Expertise is required during SPS model development in many ways. For example, to define model purpose and scope, to select model parameters, to design the internal model structure, to collect (or estimate) the required data, to specify the input and output parameters as well as the user interface, and to implement, verify, and validate the SPS model. Once the internal structure together with the input and output parameters of the SPS model have been designed and the model has been implemented and tested, simulation runs can be conducted, resulting in the production of output data that are represented in suitable visualizations, for example, tables, Gantt charts, and graphs showing the values of output parameters along the time axis.

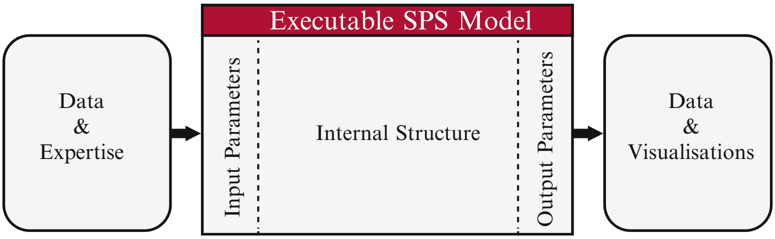


Fig. 17.1 Schematic visualization of SPS model development and application

The starting point of any SPS modeling project is the identification and explicit formulation of a problem statement. The problem statement defines the modeling goal and helps to focus the modeling activities. In particular, it determines the model purpose and scope. For SPS models, Kellner et al. (1999) propose the following categories for model purpose and scope.

Purpose:

- Strategic management
- Planning, control, and operational management
- Process improvement and technology adoption
- Understanding
- Training and learning

Scope:

- A portion of the product development life cycle, for example, requirements allocation and specification activities, design activities, coding activities, verification and validation activities, and planning activities
- A development project, for example, single product development life cycle
- Multiple, concurrent projects, for example, across teams, departments or divisions
- Long-term product evolution, for example, multiple, successive releases of a single product
- Long-term organization, for example, strategic organizational considerations spanning successive releases of multiple products over a longer time period

Purpose-scope combinations can be used to classify SPS applications in a domain-independent manner. In Fig. 17.2, we use Kellner et al.'s scheme (Kellner et al. 1999) to indicate primary and secondary SPS applications relevant for software project management (PM). Primary SPS applications (marked with an "x" in Fig. 17.2) aim at supporting project managers and engineers in their planning and control tasks, that is, in their value-generating activities. Secondary SPS applications (marked with an "o" in Fig. 17.2) aim at helping project managers and engineers in understanding, learning, training, and improvement tasks, that is, in activities that are only indirectly value-generating.

Scope Purpose	Portion of lifecycle	Development project	Multiple, concurrent projects	Long-term product evolution	Long-term organization
Strategic management					
Planning	x	x	x		
Control and operational management	x	x	x		
Process improvement and technology adoption	o	o	o		
Understanding	o	o	o		
Training and learning	o	o	o		

Fig. 17.2 Classification of SPS applications based on the scheme by Kellner et al. (1999)

The aim of this chapter is to discuss the question whether there exists evidence that SPS—despite the large number of related publications—actually has had an impact in industry, as claimed in Zhang et al. (2011), or at least has a realistic potential to have an impact, as claimed, for example, in Müller and Pfahl (2008). In order to prepare for a discussion of this question, which we conduct in Sect. 17.3, it is helpful to present an example of a software PM-related SPS application. The example will help non-experts grasp the potential of SPS and it also will make the discussion in Sect. 17.3 more concrete and tangible. Since most of the published PM-related SPS applications in industry are primary (x-type) SPS applications, which the interested reader can easily find examples of in the existing literature, we will give an example of a secondary (o-type) SPS application in Sect. 17.2. Readers interested in examples of primary SPS applications may refer to the available literature surveys, for example, (Zhang et al. 2010), for pointers to related publications.

17.2 An Illustrative Application Example

For the example of an SPS application related to software PM, we use the SPS model GENSIM 2.0 (GENeric SIMulator, Version 2.0), which was developed at the University of Calgary, Canada (Garousi et al. 2009). The choice of the SPS model is purely of convenience. For the discussion in Sect. 17.3, we could have used any other existing SPS model that matches the purpose and scope needed for the application, no matter what modeling technique and tool has been used to develop this model. In our example, the scope of the SPS model is “development project” and the purpose “understanding”. More specifically, in the application example, we use the SPS model GENSIM 2.0 to analyze the impact of certain characteristics of the software development team (workforce) on project performance, that is, project duration, effort consumption, and quality of the software produced.

In the following subsections, we first briefly characterize the SPS model GEN-SIM 2.0 and then present two complementary scenarios of the application example.

17.2.1 SPS Model GENSIM 2.0

GENSIM 2.0 is a customizable and reusable SPS model. Inspired by the idea of frameworks in software systems development, GENSIM 2.0 consists of a small set of generic reusable components that can be plugged together and extended to model a wide range of different software development processes. A detailed description of GENSIM 2.0 and its implementation can be found in Khosrovian (2008).

The components of GENSIM 2.0 capture key attributes of the entities involved in different building blocks of the software development processes that affect the project performance measures. What makes the model results interesting and hard to precisely predict, are the numerous complex relationships and influences between each pair of these attributes. The current implementation of GENSIM 2.0 consists of three constructive phases (requirements, design, and code). Each constructive phase consists of a related development activity type (e.g., requirements specification) and a related verification activity type (e.g., requirements inspection). In addition, there are three analytic phases, that is, unit, integration, and system test. Similar to the constructive phases, each analytic phase has two related activity types, a development activity type and a validation activity type (e.g., unit test case development, and unit testing). This separation of activity types allows for accommodating agile development techniques such as test-first, if desired.

When a project is conducted, engineers of the development team are assigned to activities of a certain type, for example, a requirements inspection activity, a coding activity or a unit test activity. Typically, activities of the same type can be performed concurrently. Whether this actually happens depends among other things on the number and skill of the available engineers.

It should be noted that GENSIM 2.0, unlike many other SPS models, has not been custom-built to target a specific issue only. Rather, it is intended to be reused, customized, and applied to tackle emerging software development-related problems of any kind. Thus, GENSIM 2.0 is independent of a specific application domain.

GENSIM 2.0 can assist software development process management in many different ways. However, in the following subsections, we focus on the impact of changes in workforce characteristics (i.e., quantity and quality) on project performance using two scenarios:

- *Scenario 1:* Analyzing the effect of workforce headcount change on project performance
- *Scenario 2:* Analyzing the effect of workforce skill change on project performance.

17.2.2 Scenario 1: Effect of Workforce Headcount Change on Project Performance

The headcount of the workforce available for a project and their capabilities in carrying out different activities in the project have a significant impact on the project's performance. GENSIM 2.0 enables the PM to analyze this impact, taking into account all the mutual influences between the characteristics of the staffing profile, the sequence of activities, organizational policies for workforce allocation, and other factors involved in the overall development process. The scenario presented in this sub-section shows how GENSIM 2.0 could assist the PM by providing estimates of the potential effects of changes to a project's staffing profile.

To achieve increased reusability, in the implementation of GENSIM 2.0, organization-specific policies are extracted from the SD model and incorporated into an external Dynamic Link Libraries (DLL), which allows for easy modification of these heuristics and algorithm. The workforce allocation is an example of such an algorithm. The current workforce allocation algorithm in GENSIM 2.0, which is also used for the purpose of the scenarios represented in this subsection, is explained in detail in Khosrovian (2008).

Characteristics of the available workforce in GENSIM 2.0 are represented by an $n \times m$ matrix S, as shown in Eq. (17.1). In this matrix, n is the headcount of the available workforce, m is the number of types of activities that are carried out ij in the development life cycle, and s represents the skill level of the ith engineer in carrying out the jth activity. Skill level of 1 means that the engineer is fully skilled in carrying out the activity type and skill level of 0 means that he/she is not able to carry out the activity type at all. In all runs of this scenario, since we are only concerned with the number of engineers that can carry out activities of a certain type, we make the assumption that an engineer is either fully skilled (level = 1) or not skilled at all (level = 0).

$$S_{n\times m} = \begin{bmatrix} s_{11} & \cdots & s_{1m} \\ \vdots & \ddots & \vdots \\ s_{n1} & \cdots & s_{nm} \end{bmatrix}, s_{ij} \in [0,1]$$

Equation 17.1 Staffing profile representation in GENSIM 2.0

In this scenario, we use GENSIM 2.0 to evaluate how doubling the headcount of a project's available workforce affects the project's performance measures, that is, effort, quality, and duration. The analysis is performed on two extreme cases. In the first case, we start out with the assumption that each activity type can be carried out only by one engineer. In the second case, we start our analysis with the assumption that all activity types can be carried out by all available engineers. Any other case between these extremes could be investigated in a similar fashion as shown in the following.

17.2.2.1 Scenario 1—Case 1

In this case, the initial workforce consists of six engineers and each activity type can be carried out by exactly one of them. Hence, the staffing profile matrix could look like the example given in Eq. (17.2). As can be seen, in this example each engineer is capable of carrying out instances of two consecutive activity types. The simulation run with this staffing profile is referred to as the baseline run.

The effect of doubling the headcount of the workforce is analyzed in two different ways. Firstly, each type of activity can be carried out by only one of the engineers and each engineer can carry out only one type of activity. Therefore, the staffing profile matrix is defined as shown in Eq. (17.3). The simulation run with this staffing profile is referred to as run A. Secondly, each activity can be carried out by two of the engineers and each of those two can carry out the same two activity types. Hence, the staffing profile matrix is specified as shown in Eq. (17.4).

$$S_{6\times12}=\begin{bmatrix}1&1&0&0&0&0&0&0&0&0&0&0\\0&0&1&1&0&0&0&0&0&0&0&0\\0&0&0&0&1&1&0&0&0&0&0&0\\0&0&0&0&0&0&1&1&0&0&0&0\\0&0&0&0&0&0&0&0&1&1&0&0\\0&0&0&0&0&0&0&0&0&0&1&1\end{bmatrix}$$

Equation 17.2 Initial staffing profile matrix for scenario 1—case 1

$$S_{12\times12}=\begin{bmatrix}1&0&0&0&0&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0&0&0&0&0\\0&0&1&0&0&0&0&0&0&0&0&0\\0&0&0&1&0&0&0&0&0&0&0&0\\0&0&0&0&1&0&0&0&0&0&0&0\\0&0&0&0&0&1&0&0&0&0&0&0\\0&0&0&0&0&0&1&0&0&0&0&0\\0&0&0&0&0&0&0&1&0&0&0&0\\0&0&0&0&0&0&0&0&1&0&0&0\\0&0&0&0&0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&0&0&0&0&1\end{bmatrix}$$

Equation 17.3 Staffing profile matrix for run A of scenario 1—case 1

The simulation run with this staffing profile is referred to as run B. The underlying rationale for choosing the staffing profiles as specified in Eqs. (17.3) and (17.4) is that in each of the new staffing profiles one of the properties of the baseline staffing profile is preserved when doubling the number of engineers. In Eq. (17.3), the property that an activity type can only be performed by one engineer is preserved. In Eq. (17.4), the property that one engineer can perform two consecutive activity types is preserved.

Simulation results of run A, run B, and the baseline run are shown in Table 17.1. Because in run B each engineer can carry out two activities and could be potentially allocated to any of them, run B yields a much greater improvement than run A with regard to the duration of the project. These simulation results might help a project manager understand to what degree increased flexibility of the engineers in the development team is beneficial.

The difference between the estimated project durations of simulation runs A and B can be explained by the constraints inherent to the process structure. For example, a requirements inspection activity can only begin when the related requirements specification activity is finished. A potential overlap between specification and inspection exists since defects found during an inspection must be corrected, thus creating (little) additional work to be added (as rework) to the specification activity. As a result of the comparatively small overlap between inspection and specification rework, in run B, most of the times when there is more than one requirement specification activity to be done, two engineers can be assigned to this type of activity inparallel and there is no competition between the specification and inspection activity. Since similar mechanisms apply to design and coding activities, this explains the time gain (i.e., duration reduction) in run B as compared to run A.

$$S_{12\times 12} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Equation 17.4 Staffing profile matrix for run B of scenario 1—case 1

Quality remains the same in all three runs because the skill levels of all the engineers remain constant across different runs. The difference in the effort estimations is explained by the fact that the time step chosen for the simulation runs is one whole day and therefore engineers can only be reallocated to new activities on a daily basis, that is, only once a day. In case there is little work left to be done in any

Table 17.1 Simulation results for scenario 1—case 1

Run	Duration [days]	Difference in duration from baseline (%)	Effort [PD]	Difference in effort from baseline (%)	Quality [UD]	Difference in quality from baseline (%)
Baseline	1,088	0	901	0	2	0
A	1,080	−0.73	1,061	+17.75	2	0
B	610	−43.93	1,093	+21.30	2	0

PD person-days, *UD* # of undetected defects in the code document

of the activities, the model still allocates workforce to that activity for the whole day, which in turn causes the resulting effort estimations to be slightly different from the actual effort that has to be spent for that activity.

17.2.2.2 Scenario 1—Case 2

In this case, the initial workforce consists of six engineers and each activity type can be carried out by any of the engineers, that is, each of them is capable of carrying out all types of activities. Hence, the staffing profile matrix is defined as shown in Eq. (17.5). The simulation run with this staffing profile is referred to as the baseline run for case 2.

$$S_{6\times12} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Equation 17.5 Initial staffing profile matrix for example 1—case 2

The doubling effect is analyzed by running a simulation using a team of 12 engineers with the same pattern of capabilities as the baseline run, that is, each engineer is capable of carrying out any type of activity. The simulation run with this staffing profile is referred to as run C. As shown in Table 17.2, in run C, the estimated duration of the project is reduced by 45 % percent as compared to the baseline run of case 2. The reason why there is not a reduction by 50 %—as one might expect—is that some activities can be finished within 1 day no matter whether 6 or 12 engineers are allocated. As a result, since the simulation time step in the example was set to 1 day, the duration of any activity requiring less time than 1 day will remain equal (i.e., 1 day) for both runs. The difference in the effort estimates between baseline and run C are also due to the choice of the simulation time step, as explained in case 1.

Table 17.2 Simulation results for example 1—case 2

Run	Duration [days]	Difference in duration from baseline (%)	Effort [PD]	Difference in effort from baseline (%)	Quality [UD]	Difference in quality from baseline (%)
Baseline	280	0	1,005	0	2	0
C	154	−45.00	1,025	+1.99	2	0
D	232	−17.14	1,135	+12.94	2	0

PD person-day, *UD* # of undetected defects in the code document

Any other staffing profile in-between the extreme profiles used in run A (case 1) and run C (case 2), that is, involving arbitrary settings of the staffing profile matrix, could be investigated in the same manner. For example, in simulation run D, using the staffing profile shown in Eq. (17.6), the duration of the project is estimated to be 232 days, the effort spent on the project is estimated to be 1,135 person-days, and the number of undetected defects in the code is estimated to be two defects (cf. last row of Table 17.2). In other words, if the doubling of the workforce goes not hand in hand with a preservation of the property that all engineers can do any type of activity, then the reduction in time is much smaller than in run C of Table 17.2.

$$S_{12\times12}=\begin{bmatrix}1&1&0&1&0&0&1&0&0&1&0&0\\0&1&1&0&1&0&0&1&0&0&1&0\\0&0&1&0&1&1&1&0&1&1&0&1\\0&1&0&0&0&1&0&0&0&1&0&0\\0&1&0&1&1&1&1&1&0&0&1&0\\1&0&1&0&1&1&0&1&0&1&1&1\\1&1&0&0&1&0&1&0&1&0&1&1\\0&1&0&1&1&1&0&1&0&1&0&0\\0&0&1&0&1&0&1&1&0&1&0&0\\1&0&1&0&0&1&0&1&0&0&0&1\\0&1&0&1&1&0&0&1&0&0&0&0\\0&0&0&0&0&0&0&1&0&1&0&1\end{bmatrix}$$

Equation 17.6 Staffing profile matrix with arbitrary settings (run D of scenario 1—case 2)

17.2.3 Scenario 2: Effect of Workforce Skill Change on Project Performance

The second scenario illustrates how GENSIM 2.0 can be used to analyze the effects of training the available workforce or hiring better skilled workforce, and to understand which types of skills are more worth the investment than others. GENSIM 2.0 assumes that for each engineer, a skill level, defined as a number s in the interval [0, 1], can be specified for any of the activity types required in a project. If providing the skill level with such accuracy is not possible and the engineers' skill levels could only be specified on an ordinal scale a mapping from the ordinal scale, onto interval [0, 1] would resolve the issue. For example, if the

Table 17.3 Example of mapping skill levels from ordinal to ratio scale

Value on ordinal scale	Value on ratio scale
Unable to do	0
Weak	0.25
Medium	0.5
Good	0.75
Excellent	1

engineers' skill levels are provided on an ordinal scale with five values including excellent, good, medium, and weak, we could map them onto interval [0,1] using the mapping scheme of Table 17.3.

In the internal model structure of GENSIM 2.0, the skills of the workforce influence performance parameters in two different ways. Whenever the skill level of an engineer is increased, the speed with which he/she performs the related activity is increased while his/her chances of introducing defects decrease for development activities, and his/her chances of detecting defects increases for verification and validation activities. For example, if the skill level of engineers that perform testing activities is increased, the speed with which they test artifacts increases and the effectiveness of the testing technique increases, too. If the skill level of engineers increases, the speed with which they develop/rework artifacts increases and at the same time the number of defects they inject into the artifacts decreases. Because of the lack of reliable data on the magnitude of the effects of workforce skill levels on other model parameters, it is assumed that all the parameters affected by the workforce skill levels increase/decrease proportional to the assumed optimal performance of a directly affected parameter, for example, if the skill level of an engineer is 0.5 for system test activities, then the effectiveness of the system testing technique used by this engineer will drop from the modeled optimal value by 50 %.

The concrete question that scenario 2 answers is: What are the effects on project duration, effort consumption, and quality of the final product if we invest in training engineers or hiring better skilled engineers? Like in scenario 1, the question is analyzed for two example cases. In case 1, we start out with the situation where all engineers in the team can perform all types of activities with a uniform skill level of 0.5 for all types of activities. Then we analyze how a 50 % skill increase (i.e., from 0.5 to 0.75) for a single activity cluster, that is, development, verification, or validation, affects project performance. In case 2, we start out with a large set of engineers where each engineer is specialized and can carry out only exactly one type of activities. In the baseline situation, all engineers have a skill level of 0.5 for the type of activity they are specialized on, then—similar to what we do in case 1—the skill levels for certain activity type clusters are increased to 0.75. Again, we analyze the impact of the skill change on project performance.

17.2.3.1 Scenario 2—Case 1

In case 1 of scenario 2, the staffing profile matrix has the same size and structure as in run C (scenario 1, case 2), that is, the workforce consists of 12 engineers and each

Table 17.4 Inputs for the simulation runs of scenario 2—case 1

Run	Uniform skill levels for requirements, design, and code development activities (D)	Uniform skill levels for inspection activities (I)	Uniform skill levels for testing activities, incl test case development (T)
Baseline	0.5	0.5	0.5
E	*0.75*	0.5	0.5
F	0.5	*0.75*	0.5
G	0.5	0.5	*0.75*

of them can potentially carry out any of the activities, though—different to run C—with a skill level that is smaller than 1.

The scenario includes four different simulation runs with differences in the engineer's skill levels as shown in Table 17.4. In simulation run E, the engineers' skill levels for requirements specification, design, and coding activities are increased to 0.75, assuming that such an increase can be achieved by hiring new engineers or training the current engineers. In simulation run F, the engineers' skill levels for all types of inspection activities are increased to 0.75, and in simulation run G the engineers' skill levels for all types of test-related activities are set to 0.75.

Table 17.5 shows for each simulation run the results of those project parameters that correspond to the most important project performance measures, that is, project duration, effort consumption, and quality (in terms of undetected defects in the code of the end product).

If the main concern of PM is the quality of the final product, increasing the engineers' skill levels for test-related activities (run G) result in more quality improvement than increasing the engineers' skill levels for other types of activities (runs E and F). However, if effort or the duration of the project is considered as well, simulation run G yields the smallest improvement with regards to these factors. Thus, in order to decide on the kind of skills that shall be invested in, priorities of the PM have to be taken into account and trade-offs have to be analyzed.

17.2.3.2 Scenario 2—Case 2

In case 2 of scenario 2, the workforce comprises 70 engineers and each engineer can be assigned to only one specific type of activity. The number of engineers that can conduct a specific type of activity is shown in Table 17.6. These numbers were picked arbitrarily but with realistic project sizes in mind.

We performed four different simulation runs with differences in skill levels set in the same way as in case 1. The results of the simulation runs baseline, H, I, and J are shown in Table 17.7.

As the data in Table 17.7 show, similar to the results of case 1, if the major concern is quality of the final product, investing in the skill of engineers in charge of test-related activities is the best choice. However, if duration and effort are

Table 17.5 Results of the four simulation runs of scenario 2—case 1

Run	Duration [days]	Difference in duration from baseline (%)	Effort [PD]	Difference in effort from baseline	Quality [UD]	Difference in quality from baseline (%)
Baseline	956	0	4,796	0	502	0
E	786	−17.78	3,924	−18.18	418	−16.73
F	500	−47.69	2,879	−39.97	230	−52.3
G	917	−4.07	4,645	−3.14	158	−68.52

PD person-day, *UD* # of undetected defects in the code document

Table 17.6 Workforce information for scenario 2—case 2

Activity type	Number of engineers
Requirements specification development (D)	6
Requirements specification inspection (I)	1
Design development (D)	12
Design inspection (I)	3
Code development (D)	23
Code inspection (I)	5
Unit test case development (T)	5
Unit testing (T)	5
Integration test case development (T)	5
Integration testing (T)	5
System test case development (T)	10
System testing (T)	10

D, I, and T correspond to the definitions given in the top row of Table 17.4

Table 17.7 Simulation results for scenario 2—case 2

Run	Duration [days]	Difference in duration from baseline (%)	Effort [PD]	Difference in effort from baseline (%)	Quality [UD]	Difference in quality from baseline (%)
Baseline	545	0	4,872	0	502	0
H	446	–18.17	3,986	–18.19	418	–16.67
I	329	–39.63	2,984	–38.74	230	–54.12
J	524	–3.85	4,729	–2.92	158	–68.60

PD person-day, *UD* # of undetected defects in the code document

important as well, investing in the training of engineers conducting inspections is the best choice.

As in other engineering disciplines, scenarios like those presented in this and the previous subsections can be automated and fed with many different sets of input data. The simulation can even be coupled with optimization techniques such that—in this example—the best combination of skill levels for an available set of engineers can be found under both project and business constraints. Thousand other sorts of interesting software PM problems could be tackled by a simulator like GENIM 2.0—or any other suitable SPS model. The question is however: Why is this not done in practice? In the next section, we try to give a few answers to this question.

17.3 The Gap Between State of the Art and State of Practice

Although literature surveys (e.g., Zhang et al. (2010)) suggest that the number of published SPS applications is constantly growing, there is no evidence that the software industry has actually started adopting SPS in the context of commercial software development for any of the purposes listed in Fig. 17.2 (cf. Sect. 17.1). The claim by Zhang et al. (2011) that there are indications signaling that SPS research is starting to have an impact on software development practice is relying exclusively on the published work of a small number of experienced researchers in the SPS community who have a vested interest in creating the impression that their research efforts have indeed impact. In their published work, these researchers present examples of specific SPS applications in the software industry. However, apart from claims that positive feedback from engineers and managers has been received in response to those specific SPS applications, there exists no substantial empirical evidence that SPS had an impact beyond a single-case application in any software development organization. Also, there is no evidence that SPS has been repeatedly applied in a software development organization or even has become a common practice in the aftermath of the one-case application.

If one takes a look at the hundreds of publications reporting SPS research and application, one finds the same result: Whenever researchers (either from academia or industry) apply SPS for a specific purpose in a specific context at a company, other than anecdotal evidence about the positive effects of the SPS application in the specific context is hardly ever reported. Sometimes, anecdotal evidence is backed- up by questionnaire-based or interview-based surveys but those have usually only a handful of respondents.

Why is there so little convincing evidence of successful and sustained impact of SPS applications in the software industry? The simple and most likely answer is that there has not been yet any significant positive impact that could have been reported. Based on our research and experience in the field, we believe that there are—at least—three reasons for the lack of (sustained) success of SPS applications in software industry:

1. High cost of developing an SPS model for a professional software development environment
2. High cost of SPS model evolution and ownership within a professional software development environment
3. Difficulty of demonstrating the benefit of applying an SPS model in a professional software development environment

In the following subsections, we take a closer look at each of these three points.

17.3.1 High Cost of SPS Model Development

The first reason for little acceptance of SPS in the software industry is the high development cost (e.g., in terms of effort) that needs to be invested to come up with a trustworthy SPS model. Related to software PM, this is particularly true for SPS models that address purpose/scope combinations, which are marked with an “x” in Fig. 17.2, that is, SPS models for the purpose of planning, control, and operational management with scope “portion of lifecycle”, “development project”, and “multiple, concurrent projects”. In order to become trustworthy, these kinds of SPS models need to represent the complex internal dynamics of software projects accurately and require sufficient data, either based on past measurement or based on (trustworthy and reliable) expert opinion for sufficiently accurate model calibration. In order to fulfill these requirements, much efforts needs to be invested for (1) analyzing the development processes that are actually used in an organization, (2) collecting the required data, and (3) eliciting and checking the required expert knowledge.

How demanding this task is can be illustrated with the help of the GENSIM 2.0 model, which we used in the application example of Sect. 17.2. This model contains a minimum set of 28 model parameters that have to be calibrated based on empirical data or expert estimates (Khosrovian 2008), as shown in Table 17.8. Other SPS models of similar complexity, like for example, the SPS model presented in Raffo et al. (2004), have similarly long lists of calibration parameters. Very few software organizations—if any at all—have data like that shown in Table 17.8 readily available.

Besides the calibration of parameters, one must also consider the cost for capturing the actual development process and transforming it into a SPS model structure. For models like GENSIM 2.0 or the one used by Raffo et al. (2004), this effort is comparatively low as they are using generic model structures like that of the V-Model and the ISO 12207 reference process model. If the SPS model structure is supposed to capture the proprietary processes of a software development organization, process modeling effort needs to be invested. In mature organizations with rich processes that are documented and where the process documentation is continuously maintained, the process modeling effort might be affordable. However, in less mature organizations with little process standardization and documentation or in agile organizations that have less rigidity in the processes that are followed in a project and have more flexibility due to enhanced self-management and mutual adjustment (Chap. 11), the process modeling effort will be high as the model will have to capture many variants and exceptions. What typical effort numbers are is difficult to tell since data on the cost of SPS model development are hardly ever reported. The fact that many SPS models that have been published were developed in the course of a master or PhD thesis lets us assume that the cost is typically more than several person-months of effort.

It should be noted, though, that the cost for the development of SPS models for the purpose of training and learning (i.e., the last row of the SPS model categories marked with an “o” in Fig. 17.2) can be assumed to be lower, because the

Table 17.8 Calibration parameters of GENSIM 2.0

Calibration parameter name	Unit	Value
Initial requ. dev. rate per person per day	Page/person-day	0.07
Initial design dev. rate per person per day	Page/person-day	0.829
Initial code dev. rate per person per day	KLOC/person-day	0.048
Code rework effort for code faults detected in CI	Person-day/defect	0.3387
Code rework effort for code faults detected in UT	Person-day/defect	0.4325
Code rework effort for code faults detected in IT	Person-day/defect	1.0815
Code rework effort for code faults detected in ST	Person-day/defect	5.6225
Design rework effort per fault	Person-day/defect	0.29
Requ. spec. rework effort per fault	Person-day/defect	0.125
Average requ. spec. to design conversion factor	Page/page	31
Average design to code conversion factor	KLOC/page	0.2
Average requ. spec. to design fault multiplier	N/A	3
Average design to code fault multiplier	N/A	3
Maximum requ. spec. ver. rate per person per day	Page/person-day	8
Maximum design ver. rate per person per day	Page/person-day	30
Maximum code ver. rate per person per day	KLOC/person-day	0.6
Minimum requ. spec. fault injection rate per size unit	Defect/Page	40.14
Minimum design fault injection rate per size unit	Defect/Page	1.362
Minimum code fault injection rate per size unit	Defect/KLOC	14.52
Maximum requ. spec. ver. effectiveness	N/A	0.75
Maximum design ver. effectiveness	N/A	0.76
Maximum code ver. effectiveness	N/A	0.53
Average UT productivity per person per day	KLOC/person-day	0.3093
Average IT productivity per person per day	KLOC/person-day	0.1856
Average ST productivity per person per day	KLOC/person-day	0.1546
Maximum UT effectiveness	N/A	0.66
Maximum IT effectiveness	N/A	0.69
Maximum ST effectiveness	N/A	0.93

CI code inspection, *UT* unit test, *IT* integration test, *ST* system test, *KLOC* kilo line of code

requirements for predictive accuracy are lower. This is due to the fact that these kinds of SPS models are only needed to illustrate certain phenomena of project dynamic and therefore do not require real-world data but only “plausible” data suitable to show clearly the phenomena that are of educational interest.

17.3.2 High Cost of SPS Model Evolution and Ownership

In most—if not all—published SPS applications in industry, the model was developed by a modeling expert (the academic or industrial researcher) who is not a member of the productive software development team. This works for the development of the initial SPS model. However, if a model shall be used on a continuing basis—and thus needs to evolve in order to keep up with the evolution of the software development processes it is supposed to capture—the software organization using the SPS model needs to adopt the SPS model as a part of its development environment, and thus take over SPS evolution under its own responsibility. It might indeed be the case that there are software organizations that take the ownership of SPS models; however, no publications seem to exist in support of such an assumption. With the further emergence of lean and agile development practices, chances that this will ever happen are supposedly not increasing.

17.3.3 Difficulty of Demonstrating Benefits of SPS Models

From a business point of view, in order to demonstrate a positive return on investment (ROI) of SPS modeling and application, it is necessary to compare costs and benefits in quantitative terms. Usually, cost (i.e., effort) is easier to quantify than benefit. Thus, since even SPS modeling and application costs are hardly ever published, it is no surprise that quantified benefits of SPS applications in the context of software PM cannot be found in the software engineering literature. Typically, presentations of benefits of SPS applications consist in reporting satisfaction and positive feedback of stakeholders, for example, project managers, who were involved in the SPS application as potential beneficiaries.

Of course, it is much more difficult to assess the benefits of an SPS application in tangible terms than to assess the development and application costs. Looking at the example presented in Sect. 17.2, one would have to translate the performance improvements due to change of workforce characteristics (i.e., team size and skill levels) into some cost unit that can be compared to the investment. This is not a trivial task.

17.4 Issues that need to be Addressed

We intend in this chapter to address the question whether there exists evidence that SPS actually has had an impact in industry or at least has a realistic potential to have an impact. Given the lack of published evidence for any kind of impact in the software industry, the first part of the question must be answered with a "no". Whether there is a chance that SPS at some point in time in the future will begin to have an impact and become an accepted and regularly used PM tools depends on progress in three areas: (1) adequate SPS model validity, (2) lower SPS model development and evolution cost, and (3) measurable benefits of SPS model application.

Addressing the first area, that is, adequate SPS model validity, requires that methods, techniques, and tools be provided that help define and achieve validity levels of SPS models appropriate for the type and context of application. Addressing the second area, that is, lower SPS model development and evolution cost, requires that modeling costs are measured and published, and that methods techniques and tools be provided that help reduce SPS modeling cost. Addressing the third area, that is, measurable benefits of SPS model application, requires that appropriate scenarios and measurement systems be provided that help demonstrate the benefits of SPS model applications in an objective manner. In the following subsections, we list issues that need to be addressed for each of the three areas.

17.4.1 Issues Related to SPS Model Validity

17.4.1.1 Issue 1: Defining Appropriate SPS Model Validity Levels

When looking at the different purpose/scope categories defined in Fig. 17.2 (cf. Sect. 17.1), it seems to be obvious that those SPS models falling in the categories marked with "x" (x-type SPS models) require stricter validity levels, for example, in terms of predictive accuracy, than those SPS models falling in the categories marked with "o" (o-type SPS models), if project managers shall accept simulation results as trust-worthy input to their regular decision making. However, while SPS models that have been developed for the purpose of understanding and training might require less predictive accuracy, they need to be able to illustrate certain phenomena of interest in a clear and easy-to-understand way. Thus, they might have higher demands with respect to usability and expressive power. Also, SPS models that have been developed for the purpose of improvement and technology adoption might also need less predictive accuracy than SPS models for the purpose of planning and control, but at least they must be able to rank the project performance when comparing process alternatives correctly. In any case, different types of models have different types and levels of validity that they need to fulfill in order to gain acceptance in industrial practice. Research needs to be done to define how appropriate validity levels should be defined for the various types of SPS model applications.

17.4.1.2 Issue 2: Providing Methods, Tools and Techniques for Assessing SPS Model Validity Levels

Once ways have been found to define adequate validities, methods, tools, and techniques need to be developed that help assess the degree of validity of an SPS model. This includes both quantitative and qualitative methods. The general simulation and SD literature has made proposals on how to validate models (e.g., Sargent 2011). These proposals need to be tailored to the specific needs of software organizations and software project managers. A first attempt in this regard made by Pfahl and Ruhe (2002) was insufficient and had no consequences. Thus, a more comprehensive, fresh attempt should be undertaken.

17.4.2 Issues Related to SPS Model Development and Evolution Cost

17.4.2.1 Issue 3: Collection and Reporting of SPS Modeling Cost

The fact that hardly ever SPS modeling cost is reported in the literature may be due to two reasons: either cost (effort) is indeed not collected, or it is collected but consider too unimportant or too sensitive information to be reported. Whatever the reason is, in order to develop a common understanding of the cost of developing and evolving typical SPS models for use in professional software development contexts, both cost data collection and cost reporting (at least for research publications) must become a standard practice.

17.4.2.2 Issue 4: Reduction of SPS Modeling Cost

Based on our own SPS modeling projects with partners in the software industry, we learned that SPS model development and evolution costs are too high to make it an attractive endeavor for software organizations—at least as long as no tangible benefits and a positive ROI can be demonstrated based on trustworthy empirical evidence.

It is probable that there exist many ways to reduce SPS modeling cost. One visionary approach would be to automate the modeling process based on the data and knowledge that is available (for free or at low cost) in a software organization.

This vision, however, is far-fetched and currently there has not been very promising work done in this direction—although the new research paradigm of search- based software engineering (SBSE) might open up new opportunities in this regard (Chap. 15).

Two options for SPS modeling cost reduction that might be more realistic for the near future are (1) systematic re-use via SPS model patterns and (2) lean/agile SPS modeling processes. Usually, SPS models are developed from scratch each time. Reuse of SPS model elements is not yet common practice. The tendency to develop SPS models each time from scratch is mainly due to the lack of practical guidance for planning and performing comprehensive reuse of process simulation modeling artifacts (or parts of them). In order to facilitate reuse and speeding up the developing process, we have started to investigate possibilities for exploiting the principles of (process) design patterns and agile software development for SPS modeling (Garousi et al. 2009; Angkasaputra and Pfahl 2004).

17.4.3 Issues Related to Assessing Benefits of SPS Model Application

17.4.3.1 Issue 5: Assessing and Reporting Benefits of SPS Applications

In order to convince software professionals of the value of using SPS, convincing SPS application scenarios that demonstrate benefits in a tangible way must be developed. For different SPS applications (i.e., having different scope and purpose), typical scenarios are needed, with clearly defined benefit criteria and related measures. In order to be convincing, benefit measurement must go beyond asking stakeholders whether they "liked" the application or "are satisfied with the results". The benefits must be demonstrated in such a convincing way that the question whether a software organization would invest into SPS and use it on a regular basis as part of their productive development activities is answered positively.

How difficult this can be has been demonstrated by an endeavor that was as ambitious as SPS, that is, the Experience Factory at NASA (Basili et al. 2002). Since SPS models are integrating all types of models that typically are expected to be included in an Experience Factory organization, the complexity of the SPS development and evolution task is comparable.

17.5 Conclusions

The goal of this chapter has been to find an answer to the question whether SPS is—or could become—a beneficial and regularly used tool for software project managers. As we have argued, there is a belief in the SPS research community that the answer to this question should be "yes". This belief is partly nurtured by the belief that process simulation has shown to be beneficial in the management of manufacturing, production, and business processes. This belief is, however, not substantiated by empirical evidence. For example, a study Melão and Pidd reports a "low usage of simulation in the design, modification and improvement of business processes" (Melão and Pidd 2003). This finding is corroborated by a more recent study, which found that among over 500 respondents who say that they do business process modeling only 28 % use simulation (Harmon and Wolf 2011). To make things worse, there exists one important difference between manufacturing, production, and business processes as compared to software development processes. Software development processes tend to be less standardized and more flexible, and thus typically evolve quickly. Lack of standardization and higher flexibility make it more difficult to construct and maintain sufficiently valid SPS models.

Based on our analysis of the gap between the claims made in academia about the potential of SPS and the impact it has on professional software development, we suggest that several issues need to be satisfactorily addressed before the potentialities of SPS regarding software PM materialize and a measurable impact will be achieved. Given the huge task ahead, we are not optimistic that SPS modeling and application will become an accepted and regular practice in the foreseeable future for "x"-type SPS models, that is, SPS models in support of PM planning and operational control tasks. Such tasks might be more effectively dealt with by simpler and more direct techniques, such as those outlined in Chaps. 3 and 4. Due to the different nature of their validity requirements, we are more optimistic about "o"-type SPS models, in particular models in support of learning and training tasks, if the issues listed in the previous section are properly addressed.

Acknowledgments Keyvan Khosrovian helped with the preparation of the application examples that were part of his master thesis. His work was supported by NSERC (Natural Science and Engineering Research Council, Canada). Dietmar Pfahl is partly supported by the "Estonian higher education information and communications technology and research and development activities state program 2011–2015 (ICT program)"—EU Regional Development Fund.

References

Abdel-Hamid TK, Madnick SE (1991) Software projects dynamics – an integrated approach. Prentice-Hall, Upper Saddle River, NJ

Angkasaputra N, Pfahl D (2004) Making software process simulation modeling agile and pattern-based. In: Pfahl D, Raffo D, Rus I, Wernick P (eds) Fifth international workshop on software process simulation and modeling, ProSim 2004, Edinburgh, Scotland – Proceedings. IEE Publishing, Stevenage, pp 222–227

Bai X, Zhang H, Huang L (2011) Empirical research in software process modeling: a systematic literature review. In: Proceedings of the 2011 international symposium on empirical software engineering and measurement (ESEM'11), IEEE Computer Society, Washington, DC, pp 339–342

Basili VR, McGarry FE, Pajerski R, Zelkowitz MV (2002) Lessons learned from 25 years of process improvement: the rise and fall of the NASA software engineering laboratory. In: Proceedings of the 24th international conference on software engineering (ICSE'02), ACM, New York, NY, pp 69–79

Bin Ali N, Petersen K (2012) A consolidated process for software process simulation: state of the art and industry experience. In: Proceedings of the 38th EUROMICRO conference on software engineering and advanced applications (SEAA), pp 327–336

Garousi V, Khosrovian K, Pfahl D (2009) A Customizable pattern-based software process simulation model: design, calibration and application. Softw Proc Improv Pract 14:165–180

Harmon P, Wolf C (2011) Business process modeling survey. BPTrends Report

Kellner MI, Hansen GA (1989) Software process modeling: a case study. In: Proceedings of the 22nd annual Hawaii international conference on system science, vol II. Software Track, pp 175–188

Kellner MI, Madachy RJ, Raffo DM (1999) Software process simulation modeling: why? what? how? J Syst Softw 46(2/3):91–105

Khosrovian K (2008) Software process evaluation using a customizable pattern-based simulator. Master Thesis, University of Calgary, Canada

Madachy R (2007) Software process dynamics. Wiley, Chichester

Melão N, Pidd M (2003) Use of business process simulation: a survey of practitioners. J Operat Res Soc 54:2–10

Müller M, Pfahl D (2008) Simulation Methods. In: Singer J, Shull F, Sjøberg D (eds) Advanced topics in empirical software engineering: a handbook. Springer, Berlin, pp 117–152

Pfahl D, Ruhe G (2002) IMMoS - a methodology for integrated measurement, modelling, and simulation. Softw Proc Improv Pract 7(3/4):189–210

Pfahl D, Ruhe G, Lebsanft K, Stupperich M (2006) Software process simulation with system dynamics - a tool for learning and decision support. In: Acuña ST, Sánchez-Segura MI (eds) New trends in software process modelling. Series on software engineering and knowledge engineering, vol 18. World Scientific, Singapore. ISBN 981-256-619-8, pp 57–90

Raffo DM, Nayak U, Setamanit S, Sullivan P, Wakeland W (2004) Using software process simulation to assess the impact of IV&V activities. In: Proceedings of software process simulation modeling workshop, Edinburgh, Scotland, pp 197–205

Sargent RG (2011) Verification and validation of simulation models. In: Proceedings of the 2011 winter simulation conference, pp 183–198

Zhang H, Kitchenham B, Pfahl D (2008) Software process simulation modeling: facts, trends and directions. In: Lin et al (ed) Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC 2008). IEEE Computer Society, pp 59–66

Zhang H, Kitchenham B, Pfahl D (2010) Software process simulation modeling: an extended systematic review. In: Münch J, Yang Y, and Schäfer W (eds) International conference on software process, ICSP 2010 – Proceedings. Springer, Berlin (Lecture Notes in Computer Science 6195), pp 309–320

Zhang H, Jeffrey R, Houston D, Huang L, Zhu L (2011) Impact of process simulation on software practice: an initial report. In: Proceedings of ICSE 2011, ACM, pp 1046–1056

Biography Dietmar Pfahl is a member of the Software Engineering Group in the Institute of Computer Science, University of Tartu, Estonia. In addition, he is an Adjunct Professor with the Schulich School of Engineering at the University of Calgary, Canada. His research interests include software project and product management, software process improvement, software testing, and empirical software engineering. He is a senior member of the ACM and a member of the IEEE and the IEEE Computer Society.

Chapter 18
Occam's Razor and Simple Software Project Management

Tim Menzies

Abstract Occam's Razor is a principle of parsimony for problem solving. It states that among competing hypotheses, the one with the fewest assumptions should be selected. This chapter applies Occam's Razor to model-based project management. In this style of management, a manager uses models to guide their decisions. Ideally, such models should be supported by empirical data.

This chapter explores the limits to building models from data. Results from AI and data mining show that most data sets support only very simple models. For such data, some minimal modeling (supported by automatic tools) will produce models as good as anything else.

Automatic tools can exploit this "minimal models" effect. Such tools can automatically find very simple and very succinct recommendations about how to change and improve software projects.

18.1 Introduction

> *(Models) are almost the quintessential artifacts, for adaptivity to the environment is their whole raison d'etre.*
>
> Herbert Simon "The Science of the Artificial" (Simon 1996)

The chapter is about simplicity. Specifically, it discusses how to help managers find the least they need to change in order to most improve their projects.

This chapter assumes that managers learn what to change by reflecting on previous projects. That is, management decisions are based on patterns in the historical log of prior projects. What is argued here is that those patterns are either

T. Menzies (✉)
West Virginia University, Morgantown, WV 26506, USA
e-mail: tim.menzies@gmail.com

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*,
DOI 10.1007/978-3-642-55035-5_18, © Springer-Verlag Berlin Heidelberg 2014

very simple or nonexistent. This means in turn that it is easy to learn what to change, and the proposed changes are very succinct. Such succinct recommendations can be read and understood very quickly. Further, since they are short, they need not be cumbersome to implement. Hence, we recommend this approach for model-based project management.

A key concept in this chapter is "support-based" reasoning. In this approach, managers make decisions using models that are supported by domain data. The approach has two phases: (1) Prune away irrelevances from any collected project data. In this phase, we heavily prune the data using Occams's Razor,[1] implemented as a set of data mining tools. (2) In the reduced space, build models in order to better facilitate their usage in software project management.

Why explore model-based project management? The answer is simple: much of software project management can be characterized as the construction and exploration of models of software projects. For example, suppose we have a model of which actions lead to what results. Using that model a software manager can tackle many of the management processes listed in Chap. 1:

- *Integration management* (coordinate areas to work together throughout a project): Given a community trying to integrate their work, a project manager could reflect on the tasks already completed to decide what tasks should be implemented next in order to maximize the odds of project completion within time and budget. These odds could be computed by running "what-if" scenarios across a model.
- *Scope management* (ensure that the project includes all of the requirements and no new requirements are added in a way that could harm the project): Our project manager could run more "what-if" scenarios over their model of a project to find what actions are best or worst to do next.
- *Time, cost and quality management* (processes to ensure that the project is completed on schedule and in budget): Using more "what-if" scenarios, our manager could explore different ways to increase the odds that a project completes on time and on budget (all the while, maintaining project quality).
- *Risk management* (involves identifying, managing and controlling risk of a project): Suppose we ran more what-if scenarios on the model to find the worst possible outcomes. The set of between our current situation and these worst cases would be a list of monitors to implement to ensure a project does not slip into those worse cases.

Note the precondition of all the above: *managers have access to some model about their projects*. The goal of this chapter is to answer the following question:

> Is it possible to reduce the effort associated with building valid and usable models to support project management?

[1] "*Entia non sunt multiplicanda praeter necessitate,*" which translates to "entities must not be multiplied beyond necessity."

This is an important issue since the limits to that modeling process are also the limit to supporting model-based project management. To explore this issue, we distinguish two different ways to built models: (1) *speculation-based modeling* and (2) *support-based modeling*.

In *support-based modeling*, real project data is converted into models, perhaps with the help of some data mining algorithms (Witten and Frank 1999). On the other hand, models built via *speculation* do not use any form of supporting evidence. Experts can write such speculative models just by recording their reflections and impressions.

Speculative modeling takes many forms, some of which as discussed below (and include (1) Delphi sessions where business users meet to brainstorm defect and effect prediction models; and (2) the AI modeling work discussed in Sect. 18.3). While speculative modeling sounds somewhat unfounded (perhaps even reckless and dangerous) it can lead to useful results. Norman Fenton builds such models with his clients. His models are intricate Bayesian nets describing the factors that lead to defects in his client's software project. Given months of rigorous and highly structured meetings, these speculation, can mature into reasonable accurate defect predictors (Fenton et al. 2007).

However, there are two problems with speculation-based modeling. Firstly, doing it properly requires extensive and prolonged interaction with experts:

- In a personnel communication, Fenton reports that one of his models took 2 years to build.
- Valerdi reports the effort associated to commission a software cost model within an organization. Using the *wideband Delphi method* (consecutive sessions with dozens of experts) takes weeks of time to complete (Valerdi 2011).
- Later in the chapter, a review of the history of AI shows that speculation-based modeling was a contributing factor to one of the great commercial disasters in the history of AI (the 1987 AI winter).

The second problem is that unless the speculation-based models are validated (using real project data), it is unclear if the models built via speculation are invalid or not. Early last century, the physicist Wolfgang Pauli lamented such models, commenting that without supporting evidence then all we can say about such models is that "it is not only not right, it *is not even wrong*".

This chapter explores speculative vs. support-based modeling (and we will favor the latter since we hope our models will never be "not even wrong"). The good news is that if we focus on models that are *supported by data*, then new results offer much optimism for modeling. Specifically, if we apply certain data pruning operators, then we can generate very succinct results. There are three kinds of data pruning operators discussed in this chapter:

- Instance selectors to prune irrelevant examples (Olvera-López et al. 2010)
- Feature selectors to prune irrelevant variables (Hall and Holmes 2003)
- Range selectors to prune variable ranges that do not contribute to decision-making (Menzies and Hu 2003; Menzies et al. 2010, 2012)

For more details on all these data pruning operators, see later in this chapter. But, in short, data pruning works as follows. Consider a table of data where each row is data from one project and each column is one feature we can measure about that project:

- Instance selectors remove the rows that we do not need
- Feature selectors remove the columns that we do not need
- Range selectors remove the column ranges that are not needed for decision-making

The result of data pruning is a much smaller table with fewer rows and columns (as well as fewer ranges marked as "try this to improve a project"). Once the data is pruned in this way, then modeling is easy. Or, to be more precise, it is simple to search the pruned space to uncover project recommendations. As shown below, such a minimal model will generate very succinct recommendations on how to improve a project. Such suggestions are therefore quick to read, audit, critique, and apply.

Our general conclusion is that model-based software management is not limited by our ability to build models from data (but it may be limited by our ability to collect data from projects). The caveat for this is that *before* we reason about data, we need to first prune irrelevancies. As discussed below, this pruning can be implemented using data mining tools.

The rest of this chapter is structured as follows: Firstly, in Sect. 18.2, we review decades of research on intelligent project management that concludes (1) less is more and (2) more is less. More specifically, that section argues that (1) succinct recommendations are more useful than verbose ones since (2) larger and more intricate recommendations are harder and more intricate to understand and apply.

An opposite approach to support-based modeling is discussed in Sect. 18.3. It will be seen that while there is much value in the speculations of experts, such speculation-based modeling can be very slow to perform and hard to maintain. In Sect. 18.4, it is shown how to use some automatic data mining tools to implement the data pruning. This section introduces the core technologies used in this chapter. Those technologies include one called *spectral learning*, which is a general method for finding features that matter while ignoring features that are irrelevant. Finally, in Sect. 18.5, this chapter puts it all together and presents an example of Occam's Razor with data pruning. The examples in this section will relate to cost, effort, and defect of some NASA software projects.

18.2 Occam's Razor and Project Management

This chapter argues that modeling is simple—if we prune away superfluous details. It turns out that this is a very old idea (even though, until very recently, there was little tool support for this approach). William of Ockham (c. 1287–1347) proposed *Occam's Razor,* which is a principle of parsimony for problem-solving. It states that among competing hypotheses the one with the fewest assumptions should be selected. To put that another way: (1) Prune the unnecessary and (2) focus on what remains.

It turns out that Occam's Razor is a core principle of cognitive psychology. In the early 1980s, Jill Larkin and her colleagues explained human expertise in terms of a long-term memory of possibilities and a short-term memory for the things we are currently considering (Larkin et al. 1980):

- Novices confuse themselves by overloading their short-term memory. They run so many "what-if" queries that their short-term memory chokes and dies.
- Experts, on the other hand, only reason about the things that matter so their short-term memories have space for conclusions.

The exact details of how this is done are quite technical, but the main lesson is clear: *Experts are experts since they know what to ignore*. Larkin et al. offered numerous examples of this effect. Novices performed badly when they confused themselves with too many options. Also, experts performed better since they could clear their short-term memory of all but essential features (Larkin et al. 1980).

Occam's Razor has obvious business implications. For one thing, we can build very simple decision-support systems:

- In 1916, Henri said that managers plan, organize, coordinate, and control. In that view, managers systematically assess all relevant factors to generate some optimum plan (Fayol 1916)
- Then in the 1960s, computers were used to automatically and routinely generate the information needed for the Fayol model. This was the era of the management information system (MIS), where thick wads of paper were routinely filed in the trash can (Ackoff 1967)
- In 1975, Mintzberg's classic empirical fly-on-the-wall tracking of managers in the day-to-day work demonstrated that the Fayol model was normative, rather than descriptive. For example, Mintzberg found 56 US foreman who averaged 583 activities in an 8-h shift (one every 48 s). Another study of 160 British middle and top managers found that they worked for half an hour or more without interruption only once every 2 days (Mintzberg 1975)

The lesson of MIS is that management decision-making is not inhibited by a lack of information. Rather, according to Ackoff, it is confused by an excess of irrelevant information (Ackoff 1967). This was true in the 1960s and now, in the age of the Internet, this problem has become particularly acute. As Mitch Kapor said in his famous quote, "Getting information off the Internet is like taking a drink from a fire hydrant."

Further, the lesson of Mintzberg's study is that it is vitally important to give managers succinct summaries of their available actions since, given their work pressures, they just cannot digest long and overly complex ideas. Too much data can overload a decision maker with too many irrelevancies. Data must be condensed before it is useful for supporting decisions. Modern decision-support systems evolved to filter useless information to deliver small amounts of relevant information (a subset of all information) to the manager. Hence, Occam's Razor.

18.3 Speculation-Based Modeling (Is Difficult)

This chapter advocates support-based modeling. One way to understand that style of modeling is to contrast it with another approach. Accordingly, this section discusses *speculation-based modeling*.

In 1978, Herbert Simon won a Nobel Prize for his methods of handling incomplete and imperfect knowledge while making decisions (Simon 1960, 1978, 1982). He wrote of *bounded rationality*; that is, the rationality of individuals is limited by the information they have, the cognitive limitations of their minds, and the finite amount of time they have to make a decision.

One of Herbert Simon's colleague was Allan Newell. Newell offered a rich engineering framework for implementing Simon's vision. In his seminal *knowledge-level* keynote address to the 1980 American Association of Artificial Intelligence, Allen Newell asked the following question: "What is knowledge?" (Newell 1982). Newell's answer was to define a knowledge level of goals, actions, and a *principle of rationality*:

> If an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action.

In keeping with Simon's concept of bounded rationality, Newell's knowledge level agents do not need to reason optimally. Rather, the principle of rationality characterizes intelligence as an opportunistic approach where an agent makes trade-offs and decisions on just their immediate issues. Note that this is very different to Fayol's view or that of normative economics [which, according to Simon, makes unrealistic assumptions that human decisions can be modeled mathematically as a kind of optimizer that seeks a competitive equilibrium under competing constraints (Simon 1978)].

This intelligence is the *generation and assessment of options*. This, in turn, can be modeled as a set of nested tasks such as the following (Brooks 1975):

- Find problems
  - Search to detect problems
  - Search to find a diagnosis (what has changed since before?)
- Solving problems
  - Search to generate alternative ways to change and improve the current situation
  - Search through the alternatives to find the better options
  - Search the better options to select the best action

- Resolution
  - Monitor the proposed action

Newell and Simons implemented their view of intelligence using rule-based languages like SOAR (Rosenbloom et al. 1993). A rule-based language is expressed as a set of condition-action pairs. If the contents of working memory matched to some condition, then its associated action was fired, which, in turn, could update working memory (thus triggering more rules). SOAR's rules were divided into tasks where each corresponded to a task like those listed above.

Rule-based, or heuristic, programming is the quintessential speculation-based modeling technique. In the 1970s and 1980s, generations of graduate students (including the author of this chapter) were trained the art of *knowledge acquisition,* which is the art of interviewing experts to extract rules. Such rules were viewed as the "diamonds in the head of the experts" (Feigenbaum and McCorduck 1983). As such, they were rarely questioned since the ethos of the time was that if we combined enough partial insights from the experts, the resulting *expert system* would achieve competency.

For a time, rule-based systems such as SOAR were wildly successful. Large companies such as DEC claimed they were saving millions of dollars annually by using rule-based systems to automatically configure complex hardware systems (e.g., computers and elevators) (McDermott 1981; Marcus and McDermott 1989). The American government made heavy investments in AI (partially in response to a high-profile Japanese AI initiative where government and industry jointly spent half a billion dollars on fifth generation computer systems). Graduates from top universities flocked to AI start-up companies. Rule-based expert shells (and their associated consultancy services) were sold for millions of dollars to industry and government clients.

Many technologies suffer from a "technology hype" curve where initial promising results lead to an exploding market share (due to wildly optimistic and excessive commercial expectation). This is usually followed by large prominent commercial failures and a collapse in the market. This pattern was seen in the Internet boom bust of 2002 and, in the United States, the AI winter of the late 1980s.

That AI winter was caused by a variety of factors:

- Rule-based systems were marketed as being easier to maintain than conventional programs (since all these rules run in small localized regions, independent of each other). In practice, this was not the case. Real-world rule bases often contained large groups of rules with significant and complex interactions (Bachant and McDermott 1984; Brug et al. 1986). Hence, the rules proved hard to write and hard to maintain. Note that this result is consistent with the experience of Valerdi and Fenton, mentioned above.
- The next generation of computer microchips arrived, showing that fast computation did not need specialized parallelized AI hardware and software.
- There was a management change in the American DARPA organization and the new management was less enamored of AI than before.

As a result, US government funding for AI was greatly reduced, which led to a decrease in private venture capital for AI. The AI bubble burst and, in the United States, the AI winter set in. A similar pattern (of disillusionment with AI) was seen around the globe. In the United Kingdom, the infamous Lighthill Report convinced the British government to end support for AI research in nearly all British Universities—a policy that was persisted for almost a decade before being revoked (McCarthy 1973). Also, when the Japanese finally ended their fifth generation project, all they had to show was some interesting hardware and not the breakthrough in artificial intelligence that was initially promised.

18.4 Support-Based Modeling (Can Be Simplified with Data Mining)

To summarize the above, researchers such as Newell and Simon proposed a different way of looking at project management. Their technology was certainly better (and less naive) than what was seen before (e.g., Fayol's model). It also had some initial successes. However, the fatal flaw in that technology was that it was hard to maintain and hard to scale.

If a technology has anything to offer, then after the bubble and after the bubble bursting and after a trough of disillusionment, it rises again to a steadily sustainable level. Just like the Internet market returned from its collapse in 2002, there was a revival in the AI market. In the 1990s, data mining technology matured to the point where large-scale search and generalization became possible. In the twenty-first century, many companies now use AI and data mining as part of their core business strategies (e.g., Google, NetFlix, Facebook, Yahoo, and Microsoft). Any graduate with even minimal data mining skills can command respectable salaries in many organizations.

This chapter revisits Newell and Simon's approach, but from the perspective of data mining. Like the classical AI approach, project management will still be viewed as the opportunistic search for actions that can take an agent to some improved place. However, instead of requiring humans to manually model all possible actions, we find those actions using data mining.

An interesting feature of this approach is that, in the usual case, the space of actions is very small. That is, data mining can usher in a new era of simpler and more intelligent project management tools.

18.4.1 Occam's Razor and Data Mining

As mentioned above, one lesson from the 1960s obsession with management information system (MIS) was that project managers can be overloaded with too much information (Ackoff 1967). Therefore, the wise decision analyst seeks the *smallest amount of the most important* information. The lesson of this chapter is that, using some data mining tools, this smallest amount may be very small indeed (Chang 1974; Olvera-López et al. 2010; Kohavi and John 1997; Hall and Holmes 2003; Menzies and Hu 2003, 2007; Geletko and Menzies 2003; Menzies et al. 2012; Gay et al. 2010).

Consider a table of data with rows and columns where each row is one example instance about something (e.g., a software project) and each column is one thing we can measure about that instance. A repeated result is that:

- After instance selection, *most of the rows can be ignored*: If a model can be learned from rows of data, it follows that those rows are multiple copies of some underlying effect. Hence, rows of data can usually be reduced to just a few *prototypes* that most exemplify the underlying model. Such row selection algorithms also serve to remove very similar (or repeated) rows, or noisy rows that can confuse model generation (Chang 1974; Olvera-López et al. 2010; Menzies et al. 2012). Experiments on the reduced row set show that predictive performance can be just as good as with the full set [especially when row pruning avoids outliers and other noisy examples (Menzies et al. 2012)].
- After column selection, *most of the columns can be ignored*: Feature subset selection algorithms test what columns can be removed without damaging the signal in the data. In the usual case, dozens of columns can be reduced to just a few. This process removes (a) noisy columns; or (b) columns that are irrelevant to the prediction task at hand; or (c) columns that are redundant (since they are similar to other columns in the data) (Kohavi and John 1997; Hall and Holmes 2003).
- After range selection, *most variable ranges can be ignored*: A "contrast set" is the delta between two populations. Minimal contrast sets can be formed by listing the ranges that are more common in some preferred population. Such minimal contrast sets can be very small, even between complex concepts. For example, it may require gigabytes of gene sequencing to list all the properties of human and ape DNA. But one of those two species can talk and the other cannot. Hence, a contrast set between these two complex entities might be one simple test (e.g., can they say their name?). Any variable range not in a contrast set is not something that can select one population over another; hence, in terms of offering actionable analytics, they can be ignored (Menzies and Hu 2003, 2007; Geletko and Menzies 2003; Gay et al. 2010).

  These three pruning rules can dramatically simplify a model.
- As an example of the impact of *range pruning*, previously we have discussed how contrast sets can simplify seemingly very complex models. In one case, we explored a data set that had generated a 6,000-node decision tree. The same data yielded four contrast sets with 2–5 attributes—which were small enough to show on one-eighth of a page (whereas the original decision tree filled 100 pages) (Menzies and Sinsel 2000).

- As an example of the impact of *column and row pruning*, with Kocaguenli, we have explored tables of software effort data. In experiments with 681 projects from 19 different data sets, we found that in each data set, we could ignore most of that data (Kocaguneli et al. 2013a, b). Specifically, within the rows, 36–84 % of them can be pruned and within the columns, 17–83 % of them can be pruned. After pruning we were usually left with just 11 % of the cells in the original data tables. Further, when we used *all* of the *pruned* data, there was very little difference in our ability to predict the effort required to build software (Kocaguneli et al. 2013a, b).

An interesting feature of the second result is that it did not need all the data to do its pruning. Kocaguenli's method understands the cost curve associated with collecting different features. To exploit that knowledge, they first run queries on the cheaper features. Next, for the more expensive features, they only ask for least number of values that are most informative. For example:

- The feature "have we done this kind of project before?" is an inexpensive query
- While "how long did each subcontractor spend on these tasks?" is a far more expensive query. In fact, this query many never get answered accurately (since subcontractors tend to zealously guard their productivity data)

Hence, Kocaguenli's rig was an *active* learner that sorted the data by the "cheap" features, and then only asked for the smallest number of "costly" features. But even if Kocaguenli had to access all the costly features, his approach would still have project management implications. Data collection cost is one issue and reasoning about models generated from the data is another. This chapter explores the latter issue; that is, given some data (however it is collected), how hard is it to produce succinct recommendations from that data.

The above pruning results are interesting, for many reasons:

1. It offers support to the general lesson of MIS systems in the 1960s—managers do not need to see all the data. Rather, they only need a small percentage (in the above experiments, just 11 %)
2. These results lend support to Simon and Newell's approach to AI and project management. Recall that their premise was that humans have limitations to the time they have to make decisions, as well as the cognitive processing they can apply to any task. If we could focus managers on just the essential data, then even with time and cognitive limitations, it might be possible for managers to make decisions that are just as good as if they took the time to reflect over all the data
3. Recalling Mintzberg's work, it shows that managers do not need to reflect on all data to make a decision. Rather, if they have access to data pruning tools, they need only quickly reflect on small sections of the data

But are these results a fluke? Is it just a function of the particular data sets we studied with Kocaguenli? Perhaps not. Other empirical studies report similar conclusions (that most rows and columns can be pruned without losing the signal in a table of data (Kohavi and John 1997; Hall and Holmes 2003; Chang 1974; Olvera-López et al. 2010).

More generally, there are mathematical results showing that these empirical studies are actually a reflection of some underlying mathematical process. The rest of this section describes those results, which are (1) the curse (and blessing) of dimensionality (Miller 2002), see Sect. 18.4.2, (2) intrinsic dimensionality (Levina and Bickel 2004), see Sect. 18.4.3, and (3) spectral learning (Kamvar et al., 2003), see Sect. 18.4.4.

Take together, these three results promise that we should expect, in the routine case, that quickly peeking at seemingly complex data is just as effective as a more extensive exploration of that data.

18.4.2 The "Curse" of Dimensionality (Is a Blessing in Disguise)

Our first mathematical result relates to the connection of data to the model that can be learned from that data. It will be argued that, for support-based modeling, the only learnable models are models that use just a few features. The business implication here is that, for any data set, the lessons learnable from that data are either succinct or inaccessible.

A problem with building models is, as they get more complex, it becomes harder to find data that supports any specific part of that model. This is because of a strange relationship between the number of columns in a table of data (the dimensions) and the volume of that data. To understand this *curse of dimensionality*, we need only reflect on the volume of data containing similar examples. For a table of data, we say we have one dimension per column. As the number of dimensions increases, what happens to that volume of data:

- The volume of a $n=2$, for example, the dimensional circle of radius r is $\mathrm{V}_2 = \pi r^2$.
- If we add a dimension, the circle is a sphere with volume $\mathrm{V}_3 = 4/3\ \pi r^3$.
- More generally, for $n > 3$, the volume of an n-dimensional sphere is $\mathrm{V}_n = \mathrm{V}_{n-2} * 2\pi r^2/n$. That is, to compute a volume of an n-dimensional sphere, go back two dimensions, then multiply that by the factor $2\pi r^2/n$.

Now here is the strange relationship. Note that in the last equation, when $n > 2\pi r^2$, the volume starts decreasing. For example, for a unit sphere (with radius $r = 1$), the volume maximizes at $n = 6$ dimensions, then shrinks to effectively zero after $n = 15$ dimensions.

To understand the significance of the above, consider what it means to have a well-supported model: *to be convincing, a model should be supported by enough data to be believable*. A corollary of this is that where there is *insufficient data, we cannot* build a model. This is where the curse of dimensionality comes in—it says that if we try to build a model from too many dimensions, then that model will not be well supported.

If the above math does not convince the reader, the same point can be made with (slightly) less algebra as follows: Consider the standard Euclidean distance measure from the origin to some point in n dimensions

$$X = \{x_1, x_2, \ldots x_n\} : d = \sqrt{\sum\nolimits_1^n x_i^2}$$

where d is the radius of some sphere. For the simple case where $x_i = x_j$, we can re-write the above equation as

$$d = \sqrt{nx_i^2}$$

(and for the more complex case, we refer the reader to the above math).

At first glance, this equation seems to say that more dimensions means more volume (since new dimension $n' > n$ adds to the sum of differences between the features). But that first glance is misleading. For modeling, we must generalize from related examples. If *related* means *nearby*, then it is important to check what happens if we increase the number of dimensions n, while keeping d constant.

To see how to change x_i for some constant d, we rearrange our equation to

$$x_i = \sqrt{d^2/n}.$$

Note how, for constant d, x_i must decrease as n increases; that is, we must decrease the gap x_i between any two instances (i.e., their distance along any single dimension). And as this gap shrinks, it becomes less likely to find new examples in that reduced space.

In summary, what this math shows is that the repeatable effects that we can summarize into a model are either low-dimensional or so rare that they are no longer repeatable. To say that another way, models are either simple or unsupportable since more increasingly complex models are increasingly difficult to support by the available data. This means that the curse of dimensionality can also be a blessing:

- Since it is impossible to find the data to support bigger models, then all we need ever do is build small ones.
- Which, in turn, means that we might be able to build those small models with just a little data.

18.4.3 Intrinsic Dimensionality

This section offers another mathematical argument that, in the expected case, the models we learn from data should be very simple. The math of the previous section is interesting, but a little abstract. A more convincing case (that our data has a very simple structure) would need to look at actual data and report the complexity of the structures they contain.

One tool for reporting those structures is to explore the geometry of the data using the *correlation dimension* explored discussed by Levina and Bickel (2004). To find the underlying dimensionality of a data set with n items, we plot the radius r against the number of items found at distance within r. Then we normalize this by the number of connections between n items. The result is *C(r)*, which is the expected number of neighbors at distance r:

$$C(r) = \frac{2}{n(n-1)} \sum\nolimits_{i=1}^{n} \sum\nolimits_{j=i+1}^{n} 1 \left\| x_i, x_j \right\| < r \Big\}$$

In the above, some distance measure is required to find items within radius r. We use the standard Euclidean distance described above.

To understand the intuition behind this math, consider two rooms:

- One room has no gravity, so data can float around.
- In the other room, gravity has been turned on and all the examples have fallen down to the floor.

If we lie on the floor of both rooms and blow a bigger and bigger bubble, then in the gravity-less room, we will encounter at most r^3 more examples (since the examples are floating in 3-d). However, in the other room, we will only ever find at most r^2 more examples. That is, the room with gravity contains examples in a two-dimensional space that just happens to be floating in a three-dimensional space.

Levina and Bicket report that in astrophysics it is standard to report the intrinsic dimension as the maximum slope of the a plot where *log(C(r)) is on the y-axis* and vs. *log(r) is on the x-axis*. We have computed this slope for effort and defect prediction data sets from the PROMISE repository.[2] The following are the data sets describing software project data:

- Defect data sets report how many bugs have been seen in functions or classes (described in terms of static code measures such as lines of code or depth of inheritance trees)
- Effort data sets report the total staff months needed to complete a project (described in terms of the process, personnel, and project attributes)

The results are shown in Fig. 18.1. Note that the median intrinsic dimension is usually below four. This is an interesting result since these data sets have up to

[2] http://promisedata.googlecode.com

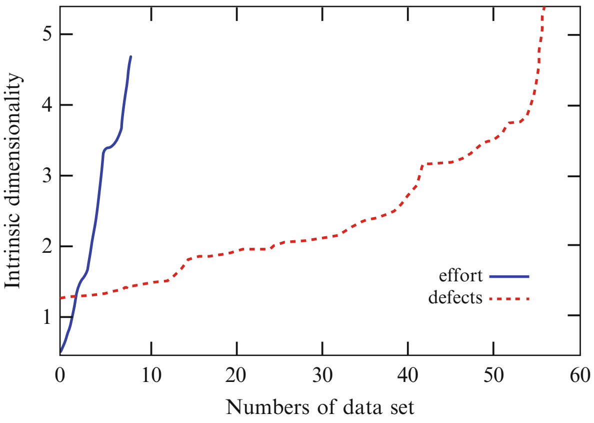


Fig. 18.1 Results from 58 defect data sets and 8 effort estimation data sets (data sets ordered by size of the slope)

21 columns; that is, the intrinsic dimensions of this software engineering data is much less than the raw number of columns.

Such small intrinsic dimensionality values suggest that managers might waste much time reflecting over all the dimensions. In fact, their time could be better spent reflecting over a much smaller number of underlying dimensions. This is very good news for Newell and Simon since it suggests that the space of operators needed to navigate the dimensions is very small (since there are so few places to go). It is also very good news for Mintzburg's managers since it suggests they may be able to make faster decisions.

Of course, for the method in the above paragraph to be practical, it must be possible to identify these crucial dimensions. As discussed below, this can be done automatically with data mining tools.

18.4.4 Spectral Learning

The maths of the last two sections say that it is possible to reduce even complex data sets to something much simpler and comprehensible. This section discusses *spectral learning*, which is one way to implement that simplification.

Levina and Bickel report that it is possible to simplify seemingly complex data:

> ... the only reason any methods work in very high dimensions is that, in fact, the data are not truly high-dimensional. Rather, they are embedded in a high-dimensional space, but can be *efficiently summarized in a space of a much lower dimension.*
>
> Levina and Bickel (2004)

We say above that software engineering data can have very low underlying dimensions. One method for exploring these underlying dimensions is *spectral learning* (Kamvar et al. 2003). Spectral learners reason about the *eigenvectors* of the project data. These vectors have the interesting property that they combine related features and ignore irrelevant features. For example, in the following diagram, the big arrow through the middle of the cloud of dots shows an eigenvector (and the little arrow shows an *orthogonal dimension*, discussed below). Observe how the big arrow tends to point in the general direction of the data; that is, the eigenvector is one way to summarize overall trends in the data.

The value of the angle between a raw feature (something measured from a project) and an eigenvector shows the impact of the former on the latter: If the eigenvector runs alongside to the feature, then that feature is oriented in the same direction as the eigenvector. Such aligned features are most powerful in predicting trends in the data. Similarly, if the eigenvector runs at right angles to the feature, then that feature is orthogonal (i.e., irrelevant) to the overall trend in the data.

For example, consider Fig. 18.2, where the principal eigenvector is shown as the larger arrow in the data. In Fig. 18.2, the x-axis is somewhat aligned with the first eigenvector. The y-axis is less aligned: that is, changes in the y values have less impact on the position of the data than changes in x. That is, the larger arrow in the above figure combines both x and y, but gives more credence to the former.

18.5 Spectral Learning and Project Management

The above discussion claims that we can simplify reasoning in project management by just reflecting on the intrinsic dimensions, found via spectral learning. If we believe the above math, then when we discuss data sets with many dimensions, we only need to dwell on a small subset of those dimensions. Also, we can support Newell and Simon's goals (the hunt for operators that can lead to some desired objectives) since we only have to explore a few dimensions. Better yet, since this is based on data mining, the effort required to set all this up is minimal.

18.5.1 Sample Data

To illustrate that technique, we need some sample data. Table 18.1 shows nine projects from NASA (Kocaguneli et al. 2012). This data was collected in the period 1987–1990 as an attempt by NASA to understand the historical records of all their software in support of the planning activities for the International Space Station. In that study, five NASA analysts worked halftime to fully document all records describing NASA's software development experience. NASA analysts traveled around the country to interview NASA employees and contractors collecting meta-knowledge about a spreadsheet of 100+ rows and less than 30 columns.

The data in Table 18.1 is described in terms of the COCOMO ontology. This ontology offer sets of features for project, personnel, product, and platform information about each project (Boehm et al. 2000). Each project has associated values for *months* of development (measured in calendar time) as well as *effort* (and

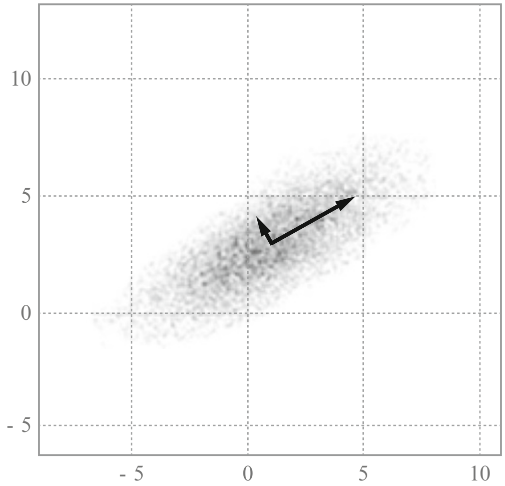


Fig. 18.2 The first two components of a cloud of data

Table 18.1 Nine projects described using the COCOMO ontology

	A	B	C	D	E	F	G	H	I
Months	38	28	29	30	25	25	23	19	14
Effort	2514	813	474	390	557	397	418	186	232
Defects	5688	4546	5614	5262	3546	3251	3311	1846	1109
Kloc	136	119	134	150	74	84	80	58	37
Acap	h	h	h	h	n	h	h	vh	n
apex	h	h	h	h	n	h	n	h	n
Cplx	vh	n	n	h	vh	h	h	h	h
Data	h	n	n	h	n	n	h	l	l
docu	n	n	n	n	n	n	n	n	n
Flex	h	h	h	h	h	h	h	h	h
Ltex	h	h	h	h	l	h	h	h	h
pcap	h	vh	h	h	n	h	h	n	n
pcon	n	n	n	n	n	n	n	n	n
Plex	h	n	h	n	l	n	n	n	n
pmat	h	h	l	l	n	h	h	n	h
Prec	h	h	h	h	h	h	h	h	h
pvol	n	l	l	n	h	l	l	l	l
Rely	vh	h	n	n	h	h	n	n	h
Resl	h	h	h	h	h	h	h	h	h
Ruse	n	n	n	n	n	n	n	n	n
Sced	n	n	n	n	n	n	n	n	l
Site	n	n	n	n	n	n	n	n	n
Stor	xh	n	vh	n	vh	vh	n	n	n
Team	vh	vh	vh	vh	vh	vh	vh	vh	vh
Time	xh	h	n	vh	vh	n	n	n	n
Tool	n	n	n	n	n	n	n	n	n

effort = staff * months) and the number of *defects* found during inspections of project artifacts.

Note that the data has 26 dimensions (three objectives: months, effort, and defects) and 23 others. The columns use the symbols *vl*, *l*, *n*, *h*, *vh*, and *xh* to denote *very low*, *low*, *nominal*, *high*, *very high*, *extremely high*. The row names come from the COCOMO project (Boehm 2000) and include

- *data:* data base size
- *acap:* analysts capability
- *aexp:* application experience
- *cplx:* process complexity
- *lexp:* language experience
- *modp:* modern programming practices
- *pcap:* programmers capability
- *rely:* required software reliability
- *sced:* schedule pressure
- *stor:* main memory constraint
- *time:* time constraint for cpu
- *tool:* use of software tools
- *turn:* turnaround time
- *vexp:* virtual machine experience
- *virt:* machine volatility

This is part of a larger data set with 93 projects, available online at http://goo.gl/iA0tlq.

18.5.2 Cluster + Contrast

If we project that data onto the first two eigenvectors of the data, we can generate the visualization of the data shown in Fig. 18.3.

This data can be clustered into 9 groups. If we replace each cluster with its middle centroid, then we get Fig. 18.4.

The above example used spectral learning to project the 23 dimensions down to two, and then cluster the data. For technical details of that process, see (Papakroni 2013; Menzies et al. 2012).

With that knowledge, we can now map out the space of management actions supported by this data. For every centroid, we find the *nearest* neighbors that have *different* and *better* class values to our *asIs* centroid. The *asIs* centroid is the one nearest the task at hand. For example, given a particular project, the *asIs* point would be centroid closest to that project. We reason about centroids near to *asIs* since recent results show that software engineering is a highly heterogeneous set of tasks and the best results come from reasoning from similar projects (since different projects can be very different indeed) (Menzies et al. 2012).

Fig. 18.3 Data placed onto a 2d grid

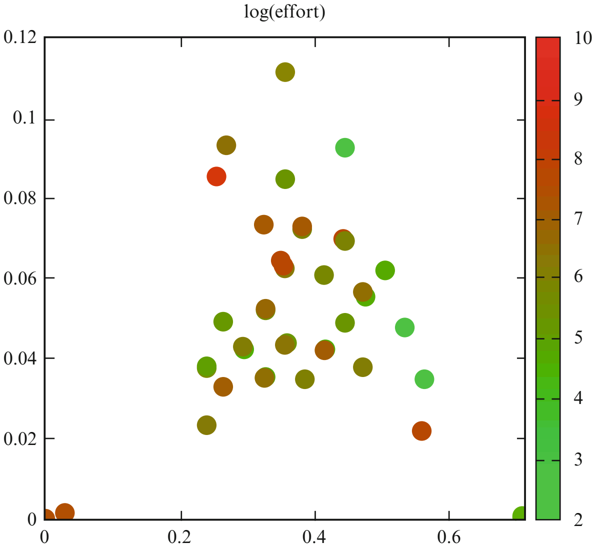
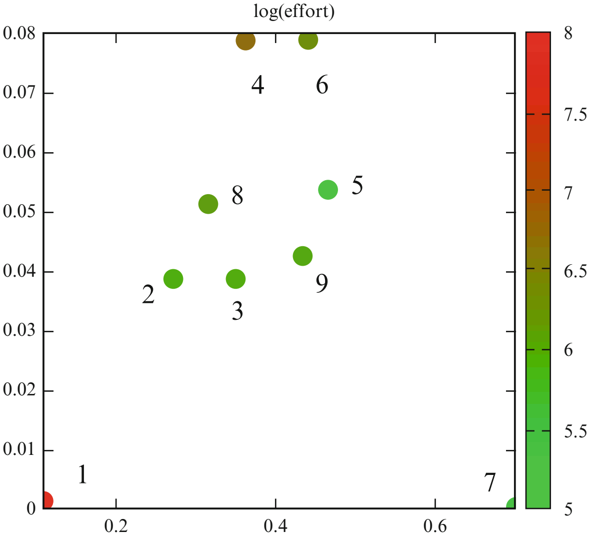


Fig. 18.4 Cluster centroids found in Fig. 18.3



For this example, we will *use Cohen's rule* (Kocaguneli et al. 2013a, b), which says that two values are *different* if they differ by more than the 30 % of the standard deviation of the entire population. For a more sophisticated analysis, we might use, say, ANOVA with a post hoc Hedges test (Kampenes et al. 2007).

As to *better*, that is a domain-specific predicate. In the case of effort estimation, "better" would mean something like "lower average effort" than at *asIs*. For some agile company rushing some product to market, it might be "least effort without too many defects". Alternatively, to explore a more complex case, it might be "same or less analyst capability, while fewer defects".

Let the nearest different better centroid be part of the *toBe* cluster. The task now is to learn a rule that reports how to change *asIs* into *toBe*. If all the data is discretized (Dougherty et al. 1995), we can learn that rule as follows:

- Let m and n be the number of projects in the *asIs* and *toBe* clusters
- Ignore features that management cannot change
- For every other feature, count the number of times f and g each range of that feature appears in *asIs* and *toBe*
- Let b (for best) and r (for rest) be $b = f/m$ and $r = g/n$ (respectively)
- Sort its feature range by $b*b/(b+r)$. There is some math about how we use this particular fraction: for details, see Menzies et al. (2007)
- Build rules by taking the x highest values in that sort, then selecting all the projects in *asIs* and *toBe* that match those values
- Keep increasing the size of x (starting at one). Stop when the projects selected by ranges $1, 2, \ldots x, x+1$ are not different and better to those found by ranges $1, 2, \ldots x$
- Report the following rule: if your project falls into *asIs*, then apply ranges $1, 2 \ldots x$ in order to drive your project into the better land of *toBe*

The above procedure implements the instance, feature, and range pruning processes described in the introduction.

- Instance pruning: Many instances are mapped into a much smaller set of centroids
- Feature pruning: Spectral learning inputs the raw features and outputs a much smaller number of features that ignore irrelevant features while combining the influences of the most relevant features
- Range pruning: The above procedure only reports the ranges from the rules.[3] If a range is not included in those rules, then it is not useful for moving a project from one region of the data to another

18.5.3 Technical Aside

The reader familiar with the AI search literature will recognize the rule generation algorithm as a greedy search,[4] ordered by a Bayesian weighting measure. This is a very simple search procedure. Previously, we have explored a more complex search in the context of spectral learning on software engineering data. We found that that

[3] For example, if programming language experience (plex) takes the range (vl,l,n,h,vh), then range pruning might ignore all but, for example, h, vh.

[4] A greedy search takes the next best idea and applies it. This process stops when the next idea does not improve on everything that has been seen before.

the search only returned tiny rules—suggesting that the extra sophistication of that search was somewhat unnecessary (Menzies et al. 2012). Note that this result—that a simple search does just as well as a more complex one—is to be expected for data with low intrinsic dimensionality.

Also, the reader familiar with the data mining literature will know that there are many ways to cluster data and then extract rules that highlight the differences between two regions of that data [see, e.g., (Novak et al. 2009; Gupta and Grossman 2004; Farnstrom et al. 2000)]. While we prefer the above (since it runs in near-linear time), it would be a worthwhile experiment to reimplement the above process using other tools for clustering and rule-learning.

18.5.4 Results

When applied to the 93 projects that generated the centroids shown above, the above methods learned the rules of Table 18.2. One thing to note here is that of the nine clusters, rules were learned for only six (*1, 2, 3, 4, 5,* and *6*). This is because 7, 8, and 9 contained the lowest effort examples. For those clusters, the management advice must be "for goodness sake, don't change anything". Note that not all clusters are mentioned since some of the differences between neighbors were statistically insignificantly different.

To use this table of rules, a project manager needs to know which cluster represents their projects; that is, where is "their" *asIs* cluster. This can be accomplished by another data mining technique—nearest neighbor reasoning. In this approach, the manager's project is compared to each of the clusters in Fig. 18.4. The manager's *asIs* cluster is the one with the closest centroid. There are two important observations to make from these results:

- In the pruned data, *recommendations are very succinct*. All the rules in the above table are very small. These rules use at most three features (but often use much less). Also, of the nearly two-dozen features in this table, only seven were ever indicated to be important for improving a project. Note that this is consistent with the intrinsic dimensionality results discussed above. When learning from a space with low intrinsic dimensionality, we should not expect to see complicated effects that use many features. In the pruned data, *context changes recommendations*. Note that no feature is found in the majority of rules and nearly all the used features only appear in one rule (exception rely appears in three rules)
- Across the data, *what is true everywhere may be false somewhere*. The above results echo recent comments about ecological inference (which is the conceit that if something holds across a population, then it also holds for individuals within that population). For example, just because process maturity (*pmat*) is a good general technique to improve software quality, it does not mean for specific projects that it is the most important factor. In the above rules, *pmat* was only found to be most useful for improving cluster 4

Table 18.2 Recommendations learned by comparing neighboring clusters from Fig. 18.4

Cluster ID		
asIs	toBe	Recommendation (on how to move from "asIs" to "toBe")
1	2	plex = n
2	3	pvol = n
3	8	rely = n
4	6	pmat = n and reply = n
5	9	acap = n and pcap = n and tool = n
6	8	rely = h and acap = vh

Elsewhere, we have conducted a much larger study with this method (Menzies et al. 2012), where the authors applied the above process to numerous data sets:

- Two effort estimation data sets: NasaCoc and China (containing information on United States and Chinese software systems)
- Seven other defect estimation data sets that describe thousands of projects in terms of object-oriented class features

The results are shown in Fig. 18.3. Each data set generated between 2 (SYNAPSE) and 8 (XALAN) clusters. One cluster always had the best score (lowest effort or defects), and this cluster was labeled C_0 . No rules were learned for this cluster since our recommendation for projects in C_0 . C_0 is to "maintain the status quo"—that is, we do not know how to improve the median value of the dependent variable (defect or effort) for this cluster, given the current data.

In Table 18.3, lines $C_i \in C_1 \ldots C_7$ show the cluster rules learned from C_i's best neighbor. Also, the gray row in Table 18.3 shows the rules learned if we ignore the clusters and generate recommendations from all data. In a comparative analysis of these *global rules* versus the *intracluster* rules, it was found that the cluster-based rules were *better* for controlling defects and reducing effort (Menzies et al. 2012). Here, *better* means that when rows were selected using either the global or the intracluster rules, then the latter resulted in lower mean and variance in the effort and defects of the selected rules. Certain effects were observed:

- In the pruned data, *recommendations are very succinct*: in Table 18.3, all the recommendations only reference one feature
- In the pruned data, *context changes recommendations*: in Table 18.3, recommendations were rarely repeated in different clusters
- In the pruned data, *what is true everywhere may be false somewhere*: The underlined cluster rules in Table 18.3 are those that are same as the global rules (these appear in the results for XALAN and XERCES). Note that there are very few cluster rule sets that are the same as the global rules

Table 18.3 Rules found in different clusters

	Effort		Defect						
Cluster	NASACOC	CHINA	LUCENE	XALAN	JEDIT	VELOCITY	SYNAPSE	TOMCAT	XERCES
global	kloc = 1	afp = 1	rfc = 2	loc = 1	rfc = 2	cam = 7	amc = 1	loc = 2	cbo = 1
C0									
C1	rely = n	added =4	amc =7	amc =1	ic =7	noc =1	dit =4	cbm =1	dit =1
C2	prec = h	delete d = 1	ca =1	cam =2	noc =1	dam =1 or 5		dam =1	dam =1
C3		delete d = 1	dam =5	cam =3	amc = 6	avg_cc = 4		noc = 1	ca = 1 or 7
C4			mfa = 1	dit =2 or 4	noc =1	moa =1		rfc =5	cbo =1
C5			moa = 1	loc =1				lcom3 = 5	
C6				loc =1 or 2				max_cc = 4	
C7				moa =1				cbm =1	

18.6 General Applications to Project Management

This section shows how the above data mining process can directly implement AI modeling, without requiring speculative modeling. Returning to Newell and Simon's view of AI, it was said above that their view of intelligence was modeled as the principle of rationality: If an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action.

More specifically, for Newell and Simon, rationality was a search through a nested set of tasks. Those tasks were listed above and included the following:

- Finding problems by searching to detect problems or searching to find a diagnosis
- Solving problems by searching to generate alternatives, or searching alternatives to find better options, or searching the options to select the best action
- Resolution by monitoring the proposed action

It turns out that the above lists of tasks can be directly implemented by the data mining methods discussed above.

- Search to detect problems:
  - Apply the above clustering algorithm to find clusters of *problem* projects that have poor scores.
- Search to find a diagnosis (what has changed since before?):
  - For the problem clusters, find nearly by clusters that are *better* (have higher scores).
  - These rules can be interpreted as the set of reasons why better projects can get worse.
- Search to generate alternative ways to change and improve the current situation:
  - Learn rules to find ranges that can drive *problem* to better.
- Search through the alternatives to find the better options:
  - As discussed above, rank those rules using some criteria that selects for the better projects (in the above, we used $b*b/(b+r)$).
- Search the better options to select the best action:
  - Show the business users the ranked list generated in the last point. Discuss with them what actions are practical in the local domain.
- Monitor the proposed action:
  - Once the user has selected their preferred action, then let $new = problem + action$ has been reached.
  - For the new context, find nearby *worrying* clusters (such *worrying* clusters have scores worse than the *new* context).

– Learn rules to find ranges that can drive *new* to *worrying*. These are the ranges that can damage the *new* situation.
– Take action to implement *new* while, at the same time, monitoring for the conditions that would drive a new situation into being *worrying*.

18.7 Discussion

This chapter has explored the following issue: *Is it possible to reduce the effort associated with building valid and usable models to support project management?*

It has proposed a recipe for the effective generation of recommendations on how to change a project. The core of the method is Occam's Razor: Prune the unnecessary and focus on what is remaining.

Prior AI-based approaches to model-based project management used speculative-models, that is, interview methods to uncover rules from business experts. If conducted in a well-structured manner, speculative modeling can achieve impressive results. However, they take considerable time, and without validation data, they may suffer from the problem stated by Wolfgang Pauli: "it is not only not right, *it is not even wrong*".

An alternative to speculation-based modeling is support-based modeling in which recommendations are generated from data. It was shown above that if the data is culled with instance, feature and range pruning, then the resulting recommendations can be very succinct.

Some mathematics was presented that suggested that this is not some quirk of the data explored in this chapter. Rather, it is a fundamental property of the structure of data. Due to intrinsic dimensionality and spectral learning, models can be very simple; models can be very simple indeed. Better yet, due to the curse of dimensionality, if we make models too complex, there will be no data to support it. The curse of dimensionality imposes an upper bound on the complexity of models supported by data.

While the results of the last paragraph may not hold for all data sets, the intrinsic dimensionality results we have calculated from known effort and defect data sets suggest that these results could hold widely in software engineering. That said, it is important to ask: "when can we be simple?". That is, when should the techniques of this chapter be applied, or be avoided? One answer is to compute the correlation dimension of the data, then plot *log(C(r)) on the y-axis and log(r) on the x-axis*. If the median slope on that line is very small (say, less than five), then the methods discussed here are appropriate.

It should be noted that most software engineering data we have checked so far has such a low dimensionality. Hence, our conclusion must be that managers can take control of their projects since if they look at their project data "the right way," then they will only ever find a small set of succinct actions supported by that data.

References

Ackoff RL (1967) Management misinformation systems. Manag Sci (December):319–331

Bachant J, McDermott J (1984) R1 revisited: four years in the trenches. AI Magazine (Fall):21–32

Boehm B (2000) Safe and simple software cost analysis. IEEE Softw (September):14–17

Boehm B, Abts C, Chulani S (2000) Software development cost estimation approaches - a survey. Ann Softw Eng 10:177–205

Brooks FP (1975) The mythical man-month, Anniversary edn. Addison-Wesley, Boston, MA

Brug A, Van de Bachant J, McDermott J (1986) The taming of R1. IEEE Exp (Fall):33–39

Chang CL (1974) Finding prototypes for nearest neighbor classifiers. IEEE Trans Comput C-23:1179–1185

Dougherty J, Kohavi R, Sahami M (1995) Supervised and unsupervised discretization of continuous features. In: International conference on machine learning, San Francisco, CA, pp 194–202

Farnstrom F, Lewis J, Elkan C (2000) Scalability for clustering algorithms revisited. SIGKDD Explor 2:51–57

Fayol H (1916) Administration industrielle et générale; prévoyance, organisation, commandement, coordination, controle. H. Dunod et E. Pinat, Paris, OCLC 40204128

Feigenbaum E, McCorduck P (1983) The fifth generation. Addison-Wesley, Reading, MA

Fenton N, Neil M, Marsh W, Hearty P, Radlinski L, Krause P (2007) Project data incorporating qualitative factors for improved software defect prediction. In: PROMISE'09

Gay G, Menzies T, Davies M, Gundy-Burlet K (2010) Automatically finding the control variables for complex system behavior. Autom Softw Eng 17(4):439–468

Geletko D, Menzies T (2003) Model-based software testing via treatment learning. In: IEEE NASE SEW 2003

Gupta C, Grossman R (2004) Genic: a single pass generalized incremental algorithm for clustering. In: 2004 SIAM international conference on data mining

Hall MA, Holmes G (2003) Benchmarking attribute selection techniques for discrete class data mining. IEEE Trans Knowl Data Eng 15(6):1437–1447

Kampenes VB, Dybå T, Hannay JE, Sjøberg D (2007) A systematic review of effect size in software engineering experiments. Inf Softw Technol 49(11–12):1073–1086

Kamvar SD, Klein D, Manning C (2003) Spectral learning. In: IJCAI'03, pp 561–566

Kocaguneli E, Menzies T, Bener A, Keung J (2012) Exploiting the essential assumptions of analogy-based effort estimation. IEEE Trans Softw Eng 28(2):425–438

Kocaguneli E, Menzies T, Keung J, Cok D, Madachy R (2013a) Active learning and effort estimation: finding the essential content of software effort estimation data. IEEE Trans Softw Eng 39(8):1040–1053

Kocaguneli E, Zimmermann T, Bird C, Nagappan N, Menzies T (2013b) Distributed development considered harmful? ICSE 2013:882–890

Kohavi R, John GH (1997) Wrappers for feature subset selection. Artif Intell 97(1–2):273–324. http://citeseer.nj.nec.com/kohavi96wrappers.html

Larkin J, McDermott J, Simon DP, Simon H (1980) Expert and novice performance in solving physics problems. Science 208:1335–1342

Levina E, Bickel PJ (2004) Maximum likelihood estimation of intrinsic dimension. In NIPS

Marcus S, McDermott J (1989) SALT: a knowledge acquisition language for propose-and-revise systems. Artif Intell 39(January):1–37

McCarthy J (1973) Lessons from the lighthill flap. http://www.aiai.ed.ac.uk/events/light-hill1973/1973-BBC-Lighthill-Controversy.mov

McDermott J (1981) R1's formative years. AI Mag 2(2):21–29

Menzies T, Hu Y (2003) Data mining for very busy people (November)

Menzies T, Hu Y (2007) Just enough learning (of Association Rules): the TAR2 treatment learner. Artif Intell Rev 25:211–229

Menzies T, Sinsel E (2000) Practical large scale what-if queries: case studies with software risk assessment. In: Proceedings ASE 2000

Menzies T, El-Rawas O, Hihn J, Feather M, Boehm B, Madachy R (2007) The business case for automated software engineering. In: ASE'07: proceedings of the twenty-second IEEE/ACM international conference on automated software engineering. ACM, New York, NY, pp 303–312

Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Bener A (2010) Defect prediction from static code features: current results, limitations, new approaches. Autom Softw Eng 17(4):375–407

Menzies T, Butcher A, Cok D, Marcus A, Layman L, Shull F, Turhan B, Zimmermann T (2012) Local vs. global lessons for defect prediction and effort estimation. IEEE Trans Softw Eng 39:822–834

Miller A (2002) Subset selection in regression, 2nd edn. Chapman & Hall, New York

Mintzberg H (1975) The manager's job: folklore and fact. Harv Bus Rev (July–August):29–61

Newell A (1982) The knowledge level. Artif Intell 18:87–127

Novak PK, Lavrač N, Webb GI (2009) Supervised descriptive rule discovery: a unifying survey of contrast set emerging pattern and subgroup mining. J Mach Learn Res 10(June):377–403

Olvera-López J, Arturo J, Ariel Carrasco-Ochoa J, Martínez-Trinidad F, Kittler J (2010) A review of instance selection methods. Artif Intell Rev 34(2):133–143

Papakroni V (2013) Data carving: identifying and removing irrelevancies in the data. Lane Department of Computer Science and Electrical Engineering, West Virginia University

Rosenbloom PS, Laird JE, Newell A (1993) The SOAR papers. The MIT Press, Cambridge, MA

Simon H (1960) The new science of management decision. Prentice Hall, Englewood Cliffs, NJ

Simon H (1978) Rational decision-making in business organizations- a Nobel memorial lecture, Dec 8. http://goo.gl/E80Nyy

Simon H (1982) Models of bounded rationality, vol 2. MIT Press, Cambridge, MA

Simon H (1996) The science of the artificial, 3rd edn. MIT Press, Cambridge, MA

Valerdi R (2011) Convergence of expert opinion via the wideband Delphi method: an application in cost estimation models. In: Incose international symposium, Denver, CO

Witten IH, Frank E (1999) Data mining: practical machine learning tools and techniques with java implementations. Morgan Kaufmann, San Francisco, CA

Biography Tim Menzies (P.h.D., UNSW) is a Professor in CS at WVU and the author of over 230 referred publication. He teaches data mining and artificial intelligence and programming languages. He is an associate editor of IEEE Transactions on Software Engineering, Empirical Software Engineering and the Automated Software Engineering Journal. In 2012, he served as co-chair of the program committee for the IEEE Automated Software Engineering conference. In 2015, he will serve as co-chair for the ICSE'15 NIER track.

Index

G. Ruhe and C. Wohlin (eds.), *Software Project Management in a Changing World*, DOI 10.1007/978-3-642-55035-5, © Springer-Verlag Berlin Heidelberg 2014