Tom Mens
Alexander Serebrenik
Anthony Cleve
Editors

Evolving Software Systems

Springer

Evolving Software Systems

Tom Mens • Alexander Serebrenik • Anthony Cleve
Editors

Evolving Software Systems

Springer

Editors
Tom Mens
Computer Science Department
University of Mons
Mons, Belgium

Alexander Serebrenik
Department of Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands

Anthony Cleve
PReCISE Research Center
University of Namur
Namur, Belgium

The cover figure is modified from:
GNU/Linux distro timeline Authors: A. Lundqvist, D. Rodic - futurist.se/gldt
Published under the GNU Free Documentation License

ISBN 978-3-642-45397-7 ISBN 978-3-642-45398-4 (eBook)
DOI 10.1007/978-3-642-45398-4
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013956637

Printed on acid-free paper

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

This book is dedicated to Manny Lehman (1925-2010) and David Notkin (1955-2013), for the important contributions they made to the field of software evolution.

Foreword

There are three things in life that are immensely gratifying when you are an academic with a number of books on your publication list. One of them is arriving in someone's office and browsing through the bookshelf to spot a copy of your book. Although that copy of the book is technically owned by the host you are visiting, it still feels very much like you are holding a copy of *your* book. A second is arriving in Schloss Dagstuhl after a long and arduous journey to see all of your books on display. For those of you who never attended a Dagstuhl seminar: the centre holds a library filled to the brim with computer science literature. For each registered guest they verify whether he or she authored a book, instantly ordering a copy when lacking one. Finally, a third is when you are at a conference and during one of the coffee breaks a shy PhD student finally has the nerve to step up to you, explaining how a part of the book really helped during the early stages of her PhD. In an academic world where citations and h-indices are used as superficial indicators of impact, influencing a PhD student is the one thing that truly matters.

Looking back, it was sometime in summer 2006 when the plans for a book on the state-of-the-art in software evolution with chapters contributed by experts in the field became real. Originally, the book was targeted towards practitioners, teachers and—last but not least—PhD students. In 2008, Springer released the book under the title “Software Evolution” and we actively reached out towards our intended audience. Over the years, we received a considerable amount of feedback from both teachers and PhD students who praised us with the material covered in the book and how they used it in graduate level academic teaching and research. As time went by it became clear that the book was steadily loosing appeal because the material was slowly becoming outdated. Indeed, the software engineering research discipline is moving rapidly, justifying a new book in order to stay abreast of the state-of-the-art. This is why, in summer 2012, Tom Mens decided to edit the book “Evolving Software Systems” that you are currently holding in your hands (or perhaps reading on some screen), with Alexander Serebrenik and Anthony Cleve as co-editors, and covering new relevant software evolution topics.

Having reviewed an early draft of this book, it became apparent how fast the field of software evolution itself is evolving. Some topics have vanished from the radar: the migration towards aspects and the modernization of databases to name but a few. Some topics remain popular but heavily shifted because they have been adopted into practice: models (with a new emphasis on domain-specific modeling); refactoring (with attention towards both small-scale and large-scale refactoring); clone detection (with a recent interest in model-based cloning) and mining software repositories (where infrastructures such as Git, Jira and Stack Overflow are shaping the kind of information available for mining). Some topics belong to the core of the field, resurfacing in one way or another: empirical research, metrics, run-time evolution, architecture. Some topics are emerging: software development as a social activity, software ecosystems, the semantic web. Finally, some topics are on the verge of breaking through, hence not yet covered in the book: exploiting cloud computing for heavy-duty analysis, migrating existing programs towards multiprocessor system-on-chip, frequent release cycles as witnessed in continuous deployment and mobile app development. So while the field of software evolution itself is evolving rapidly, the community needs books like these to keep pace with all what's happening.

Like it or not: we live in interesting times !

Antwerp, August 2013 *Serge Demeyer*

Preface

In 2008 the predecessor of this book, entitled “Software Evolution” [592] was published by Springer, presenting the research results of a number of researchers working on different aspects of software evolution. Since then, the software evolution research has explored new domains such as the study of socio-technical aspects and collaboration between different individuals contributing to a software system, the use of search-based techniques and metaheuristics, the mining of unstructured software repositories, techniques to cope with the evolution of software requirements, and dealing with the dynamic adaptation of software systems at runtime. Moreover, while the research covered in the book pertained largely to evolution of individual software projects, more and more attention is currently being paid to the evolution of collections of inter-related and inter-dependent software projects, be it under the form of web systems, software product families, software ecosystems or systems of systems. We therefore felt that it was time to “release” a new book that complements its predecessor by addressing a new series of highly relevant and active research topics in the field of software evolution. As can be seen in Table 1, both books together aim to cover most of the active topics in software evolution research today. We are very grateful to the authors of contributed chapters for sharing this feeling and having accepted to join us in this endeavour.

Where does software evolution fit in a computing curriculum?

The Computer Science Curricula body of knowledge aims to provide international guidelines for computing curricula in undergraduate programs. The CS2013 version [446] has been created by a joint task force of the ACM and the IEEE Computer Society, and constitutes a major revision of the CS2008 and CS2001 versions. This body of knowledge is organized into 19 knowledge areas, software engineering being one of the essential ones (it accounts for 8.8% of the total core hours, namely 27 out of 307 hours).

Table 1: Comparison of, and partial overlap between, the software evolution topics addressed in chapters of [592] (second column) and this book (third column), respectively. Chapters written in **boldface** have the indicated topic as their principal theme.

Topic addressed	Software Evolution 2008 [592]	Evolving Software Systems 2014
software cloning, code duplication	**Ch. 2** [475]	
defect, failure and bug prediction	**Ch. 4** [954]	
re-engineering, restructuring and refactoring	**Ch. 5** [237], Ch. 1 [590], Ch. 2 [475], Ch. 8 [617]	
database evolution	**Ch. 6** [358]	
migration, legacy systems	**Ch. 6** [358], Ch. 7 [378]	
service-oriented architectures	**Ch. 7** [378]	
testing	**Ch. 8** [617]	
aspect-oriented software development	**Ch. 9** [589], Ch. 10 [70]	
software architecture evolution	**Ch. 10** [70], Ch. 7 [378]	
reverse engineering, program understanding, program comprehension	Ch. 1 [590], Ch. 8 [617]	
incremental change, impact analysis, change propagation	Ch. 1 [590]	
evolution process	Ch. 1 [590]	
software visualisation	Ch. 3 [216]	
component-based software development	Ch. 10 [70]	
open source software	**Ch. 11** [291]	Chapter 10
software repository mining	**Ch. 3** [216], Ch. 4 [954], Ch. 11 [291]	**Chapter 5**
software transformation, model transformation, graph transformation	Ch. 6 [358], Ch. 7 [378], Ch. 10 [70]	Chapter 2
model evolution, metamodel evolution	Ch. 1 [590]	**Chapter 2**
software measurement, software quality	Ch. 4 [954]	**Chapter 3**
software requirements		**Chapter 1**
search-based software engineering		**Chapter 4**
socio-technical networks, Web 2.0		**Chapter 6**
web-based systems		**Chapter 7**
dynamic adaptation, runtime evolution		**Chapter 7.6**
software product line engineering		**Chapter 9**
software ecosystems, biological evolution		**Chapter 10**

Within the software engineering knowledge area, 10 themes have been identified as being the most important to be taught, and software evolution is one of these themes. In particular, the CS2013 body of knowledge recommends the following learning outcomes for this theme:

- Identify the principal issues associated with software evolution and explain their impact on the software life cycle.
- Discuss the challenges of evolving systems in a changing environment.
- Discuss the advantages and disadvantages of software reuse.
- Estimate the impact of a change request to an existing product of medium size.
- Identify weaknesses in a given design, and remove them through refactoring.

- Outline the process of regression testing and its role in release management.

To achieve these outcomes, it is suggested that at least the following topics be taught: software development in the context of large, pre-existing code bases (including software change, concerns and concern location, refactoring), software evolution, software reuse, software reengineering, and software maintainability characteristics.

What is this book about?

This book goes well beyond what is expected to be part of an undergraduate software evolution course. Instead, the book is a coherent collection of chapters written by renowned researchers in the field of software evolution, each chapter presenting the state of the art in a particular topic, as well as the current research, available tool support and remaining challenges. All chapters have been peer reviewed by at least five experts in the field.

We did not aim to cover all research topics in software evolution. Given the wealth of information in the field that would be an impossible task. Moreover, a number of important and actively studied software evolution topics have been profoundly covered by the predecessor of this book [592]. Therefore, we have primarily focused on new domains currently being explored most actively by software evolution researchers.

This book is divided into four parts. Part I, Evolving Software Artefacts, focuses on specific types of artefacts that are used during software evolution. Chapter 1 focuses on the evolution of software requirements, and its impact on the architecture and implementation of a software system. Chapter 2 focuses on the coupled evolution of software models and their metamodels. Chapter 3 explores the use of maintainability models for assessing the quality of evolving software.

Part II of the book focuses on techniques used by software evolution researchers. Chapter 4 explores the use of search-based techniques and metaheuristics to address untractable software evolution activities. Chapter 5 explores how to mine unstructured data stored in repositories containing historical information that may be relevant to understand the evolution of a software system. Chapter 6 discusses how socio-technical information, available thanks to Web 2.0 technology, can be leveraged to help developers in performing evolution activities.

Part III focuses on how specific types of software systems, or collections of software systems, evolve. Chapter 7 looks at the evolution of web systems. Chapter 7.6 explores the runtime evolution of adaptive software systems. Chapter 9 overviews the product line evolution of families of software products. Chapter 10 focuses on the evolution of software ecosystems and how we can learn from the evolution from natural, biological ecosystems.

A large appendix complements this work. It starts by outlining the emerging and future directions in software evolution research. It also contains a list of acronyms used in the different chapters, a glossary of important terms used in each chapter, pointers to additional resources that may be useful to the reader (books, journals, standards and major scientific events in the domain of software evolution), and datasets.

Who Should Read this Book?

This book is intended for all those interested in software engineering, and more particularly, software maintenance and evolution. Researchers as well as software practitioners will find in the contributed chapters an overview of the most recent research results covering a broad spectrum of software evolution topics.

While this book has not been written as a classical textbook, we believe that it can be used in teaching advanced (graduate or postgraduate) software engineering courses on e.g., software evolution, requirements engineering, model-driven software development or social informatics. This is exactly what we, book editors, intend to do in our own advanced software engineering and evolution lectures as well.

For researchers, the book and its contributed chapters can be used as a starting point for exploring the addressed topics in more depth, by gaining an understanding in the remaining research challenges and by diving into the wealth of literature referenced in each chapter. For ease of reference, and to avoid duplication, all bibliographic references are bundled together at the end of the book rather than on a per chapter basis, with back references to the chapters where the references have been cited. There is also an index of terms that makes it easy to find back the particular sections or chapters in which specific topics have been addressed.

Practitioners might be interested in numerous tools discussed in the forthcoming chapters. Such tools as Software QUALity Enhancement (SQUALE, Chapter 3) have been successfully applied in the industry, some others are still waiting to be discovered and used by practitioners. We hope that our book can help practitioners in this endeavour.

Happy reading!

Mons, Eindhoven, Namur,
November
2013

Tom Mens
Alexander Serebrenik
Anthony Cleve

List of Contributors

Gabriele Bavota
Department of Engineering, University of Sannio, Benevento, Italy
e-mail: gbavota@unisannio.it

Dorothea Blostein
School of Computing, Queen's University, Kingston, Ontario, Canada
e-mail: blostein@cs.queensu.ca

Alexander Borgida
Department of Computer Science, Rutgers University, New Jersey, USA
e-mail: borgida@cs.rutgers.edu

Goetz Botterweck
Lero – The Irish Software Engineering Research Centre, University of Limerick, Ireland
e-mail: goetz.botterweck@lero.ie

Maëlick Claes
Department of Computer Science, University of Mons, Belgium
e-mail: maelick.claes@umons.ac.be

Anthony Cleve
PReCISE Research Center, University of Namur, Belgium
e-mail: anthony.cleve@unamur.be

Massimiliano Di Penta
Department of Engineering, University of Sannio, Benevento, Italy
e-mail: dipenta@unisannio.it

Damiano Distante
Unitelma Sapienza University, Rome, Italy
e-mail: damiano.distante@unitelma.it

Neil A. Ernst
Department of Computer Science, University of British Columbia, Canada
e-mail: nernst@cs.ubc.ca

Rudolf Ferenc
Software Engineering Department, University of Szeged, Hungary
e-mail: ferenc@inf.u-szeged.hu

Philippe Grosjean
Department of Biology, University of Mons, Belgium
e-mail: philippe.grosjean@umons.ac.be

Tibor Gyimóthy
Software Engineering Department, University of Szeged, Hungary
e-mail: gyimothy@inf.u-szeged.hu

Ahmed E. Hassan
School of Computing, Queen's University, Kingston, Ontario, Canada
e-mail: ahmed@cs.queensu.ca

Péter Hegedűs
MTA-SZTE Research Group on Artificial Intelligence, Hungary
e-mail: hpeter@inf.u-szeged.hu

Markus Herrmannsdörfer
Technische Universität München, Germany
e-mail: markus.herrmannsdoerfer@tum.de

Ivan J. Jureta
FNRS & Louvain School of Management, University of Namur, Belgium
e-mail: ivan.jureta@unamur.be

Holger M. Kienle
Freier Informatiker, Germany
e-mail: hkienle@acm.org

David Lo
School of Information Systems, Singapore Management University, Singapore
e-mail: davidlo@smu.edu.sg

Tom Mens
Department of Computer Science, University of Mons, Belgium
e-mail: tom.mens@umons.ac.be

Hausi A. Müller
Department of Computer Science, University of Victoria, British Columbia, Canada
e-mail: hausi@cs.uvic.ca

John Mylopoulos
Dipartimento di Ingegneria e Scienza dell'Informazione, University of Trento, Italy
e-mail: jm@disi.unitn.it

Rocco Oliveto
Dipartimento Bioscienze et Territorio, University of Molise, Italy
e-mail: rocco.oliveto@unimol.it

Andreas Pleuss
Lero – The Irish Software Engineering Research Centre, University of Limerick, Ireland
e-mail: andreas.pleuss@lero.ie

Alexander Serebrenik
Software Engineering and Technology Group, Eindhoven University of Technology, The Netherlands
e-mail: a.serebrenik@tue.nl

Stephen W. Thomas
School of Computing, Queen's University, Kingston, Ontario, Canada
e-mail: sthomas@cs.queensu.ca

Yuan Tian
School of Information Systems, Singapore Management University, Singapore
e-mail: yuan.tian.2012@phdis.smu.edu.sg

Norha M. Villegas
Department of Information and Communications Technologies, Icesi University, Colombia
e-mail: nvillega@icesi.edu.co

Guido Wachsmuth
Software Engineering Research Group, Delft University of Technology, The Netherlands
e-mail: g.h.wachsmuth@tudelft.nl

Contents

Part II Techniques

Part III Evolution of specific types of software systems

Appendices

Part I
Evolving Software Artefacts

Chapter 1
An Overview of Requirements Evolution

Neil Ernst, Alexander Borgida, Ivan J. Jureta and John Mylopoulos

Summary. Changing requirements are widely regarded as one of the most significant risks for software systems development. However, in today's business landscape, these changing requirements also represent opportunities to exploit new and evolving business conditions. In consonance with other agile methods, we advocate requirements engineering techniques that embrace and accommodate requirements change. This agile approach to requirements must nonetheless be systematic and incorporate some degree of planning, especially with respect to accommodating quality attributes such as safety and security. This chapter examines the nature of requirements evolution, and the two main problems that it entails. The first is to correctly understand what is changing in the requirements, that is, the elicitation problem. The other is to act on that new information using models and other representations of the requirements, influencing the architecture and implementation of the software system. This chapter first motivates the importance of considering changing requirements in evolving software systems. It then surveys historical and existing approaches to requirements evolution with respect to the elicitation and taking action problems. Finally, the chapter describes a framework for supporting requirements evolution, defining the Requirements Evolution Problem as finding new specifications to satisfy changed requirements and domain assumptions. To motivate this, we discuss a real-life case study of the payment card industry.[1]

[1] Portions of this chapter are adapted from [276].

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_1, © Springer-Verlag Berlin Heidelberg 2014

1.1 Introduction

Most software systems are now expected to run in changing environments. Software developed using an agile methodology often focuses on releasing versions that are only partially completed, at least with respect to the full set of customer requirements. Software must operate in a context where the world is only partially understood and where the implementation is only partially completed. What drives the implementation in this scenario is the requirements, whether represented as user stories, use cases or formal specifications.

Focusing on software evolution only makes sense if one understands what the objectives are for that software. These objectives are themselves frequently changing. Indeed, as we discuss in this chapter, expecting a system's requirements to be constant and unchanging is a recipe for disaster. The 'big design up front' approach is no longer defensible, particularly in a business environment that emphasizes speed and resilience to change [436]. And yet, focusing on implementation issues to the exclusion of system objectives and business needs is equally unsatisfactory.

How does our view of the importance of the system's requirements fit with the historical literature on software evolution? Requirements have long been seen as important, as we shall describe below, in Section 1.2. In 2005, a paper on "challenges in software evolution" [598] highlighted the need to research the "...evolution of higher-level artifacts such as analysis and design models, software architectures, requirement specifications, and so on." More recently, Mens [591] listed several key challenges for software evolution including "How to ensure that the resulting system has the desired quality and functionality?" This question is the motivation for our work in requirements evolution, as we firmly believe that understanding the evolution of non-functional requirements, in particular, will help answer this.

There are three major objections to making requirements more prominent in the study of software evolution. For one, the tangible is easier to study. In many cases, particularly short-term or small-scope projects, requirements are either not used explicitly or stale the moment they are 'finished'. However, this is seldom true of high-value software products, and where it is, typically is symptomatic of a larger software process or organizational pathology. Secondly, in terms of quantity, many change tasks involve low-level corrective maintenance concerns, rather than high-level evolutionary ones. While the numbers of corrective change tasks might be greater, our position is that evolutionary requirements changes are more complex and more costly, and therefore more important, than coping with bug fixes. Finally, requirements and associated changes to the requirements are seen as part of the problem domain and therefore untouchable, much like understanding the organizational objectives might be. We believe that revisiting the problem domain and re-transitioning from problem to solution is of paramount importance in software development.

It is our view that requirements artifacts should drive implementation decisions. In other words, requirements must be tangible, and requirements must be relevant. While they often take the form of work item lists, as is the case in most industrial tools, it is preferable that they be well-structured graphs that represent all aspects of the requirements problem, capturing stakeholder objectives, domain assumptions, and implementation options. Such models allow for lightweight reasoning (e.g., [277]) where the key challenge is ‘requirements repair’: re-evaluating available solutions to solve the changed requirements, adding (minimal) new implementations where necessary [627]. We will explain this with reference to the Requirements Evolution Problem, which defines how software should respond to changes in its constituent parts: elements in the specification (the implementation), the system requirements (in the form of goals) and domain knowledge and constraints. We argue this is distinct from the Self-Adaptation Problem (cf. Chapter 7.6), which is concerned with building systems that are self-adaptive, and do not require outside intervention. The Requirements Evolution Problem explicitly supports this guided intervention. This distinction is crucial; while adaptivity is important, at some point systems will need to be managed, such as when they lack implementation to support a particular change—in operating environment, requirements, or capabilities.

In this chapter, we focus on requirements evolution. We begin by introducing the context for considering requirements in the broader field of software evolution. We then turn to the history of research into requirements evolution, including empirical studies. Next, we look at current approaches, focusing first on how industry, and industry tools, have dealt with requirements evolution. We then survey the state of the art in requirements evolution research. To conclude this chapter, we elaborate on one approach to managing changing requirements, with examples drawn from the payment card industry.

The Requirements Problem

As a reference framework, we introduce an ontology for requirements. This work is based on [450] and [449], both of which derive from the fundamental requirements problem of Zave and Jackson [945]. In modern requirements engineering, it is often the case that one distinguishes different kinds of sentences encountered in stating a “requirements problem”, according to the “ontology” of the requirements modeling language. In Zave and Jackson’s original formulation, the requirements problem is

Definition 1.1. Requirements Problem: Given requirements R (optative statements of desire), a space of possible solution specifications S_P, domain world knowledge W_D, find a subset S of S_P, and software solution finding knowledge W_S such that $W_D, W_S, S \vdash R$.

Domain world knowledge reflects properties of the external world the software operates in, e.g., constraints such as room capacity. Solution finding knowledge reflects how our requirements problem is constructed, so refinement relationships are elements of W_S, that is, the expression “ψ refines ϕ” ($\phi, \psi \in R$) is part of W_S. The Requirements Evolution Problem extends this requirements problem definition to introduce change over one increment of time.

Definition 1.2. Requirements Evolution Problem: Given (i) requirements R, domain knowledge W_D, and (ii) some chosen *existing* specification S_0 (i.e., such that $W_D, W_S, S_0 \vdash R$), as well as (iii) modified requirements problem (δ(R),δ(W_D),δ(S)) that include modified requirements, domain knowledge and possible tasks, produce a subset of possible specifications $\hat{S}$ to the changed requirements problem (i.e., $\delta(W_D), \hat{S} \vdash \delta(R)$) which satisfy some desired property Π, relating $\hat{S}$ to S_0 and possibly other aspects of the changes.

As an example, consider Figure 1.1. Here we represent a simplified version of a system for managing payments at soccer stadiums (taken from [277]), which must comply with the Payment Card Industry Data Security Standard (PCI-DSS). Payment card issuers, including Visa and Mastercard, developed the PCI-DSS for securing cardholder data. It takes effect whenever a merchant processes or stores cardholder data. We represent this as a set of high-level requirements (ovals) refined by increasingly more concrete requirements, eventually operationalized by tasks (diamond shapes).

In our proposed solution, S_0, the existing approach, consists of tasks “Buy Strongbox”, “Use Verifone POS”, and “Virtualize server instances”, shown in grey shading. In order to be PCI compliant, the requirements evolve to add requirement “Use Secure Hash on Credit Cards” (double-lined oval). This conflicts with our solution (shown as line with crosses), as Verifone terminals do not support this (hypothetically). Instead, we must evolve our implementation to include the task “Use Moneris POS” terminals, i.e., add that to $\hat{S}$ (and retract “Use Verifone POS”), which does not conflict, since it does support secure hashes.

In what follows we use this framework to characterize the challenge of managing evolving requirements. In particular, while software evolution tends to focus on managing largely changes in S, in the field of requirements we are faced with changes in any or all of S, W, R. Furthermore, since these three components are related ($W \cup S \vdash R$), changes in one impact the validity of the inferential relation. For example, changes in requirements R, e.g., from R_0 to R_1, will force a re-evaluation of whether $W \cup S$ still classically entails the satisfaction of R_1.

1.2 Historical Overview of Requirements Evolution

In this section, we survey past treatments of evolving requirements. We begin by exploring how software evolution research dealt with changing requirements. The importance of evolving requirements is directly connected to the wider issue of evolving software systems. While the majority of the literature focused on issues with maintaining and evolving software, a minority tries to understand how changes in requirements impact software maintenance.

The study of software evolution began when IBM researchers Belady and Lehman used their experiences with OS/360 to formulate several theories of software evolution, which they labeled the ‘laws’ of software evolution. This work was summarized in [510]. These papers characterize the nature of software evolution as an

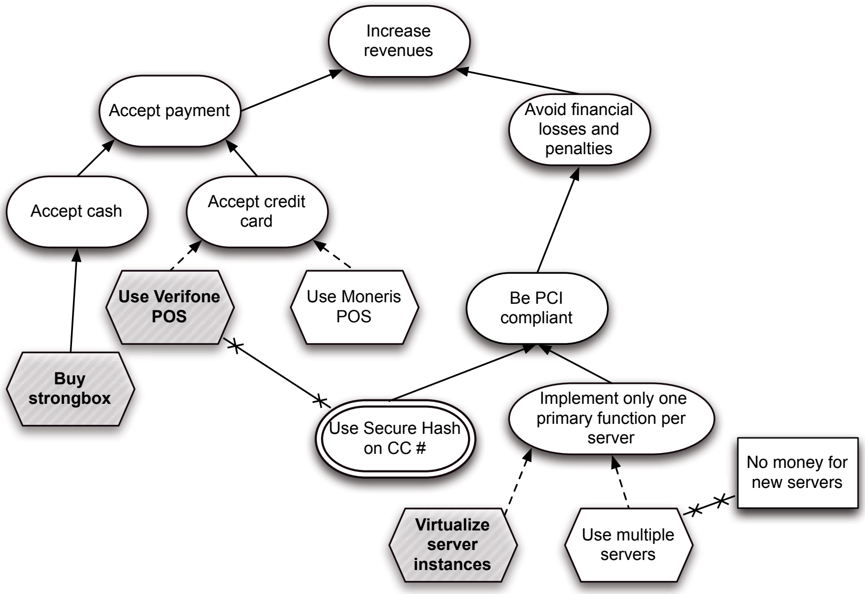


Fig. 1.1: An example of a Requirements Evolution Problem. Shaded grey nodes represent the original solution; the double outlined requirement represents a new requirement. Dashed lines represent alternative refinements, double-crossed arrows conflicts, and regular arrows refinement.

inevitable part of the software lifecycle. 'Inevitability' implies that programs must continually be maintained in order to accommodate discrepancies with their continuously evolving operational environment. One law states that software quality will decline unless regular maintenance activity occurs, and another implies that a system's complexity increases over time. While their work largely focused on implementation artifacts, it clearly acknowledged requirements as a driving force for the corrective action necessary to reconcile actual with anticipated behavior: "Computing requirements may be redefined to serve new uses [91, p. 239]."

Early in the history of software development it became clear that building software systems was nothing like engineering physical artifacts. An obvious difference was that software systems were malleable. Reports suggested a great deal of effort was being spent on maintenance tasks. Basili, writing in 1984, lists 40% [76], and the U.S. National Institute of Standards and Technology report in 2002 claimed industry data show that 70% of errors are introduced during requirements and architecture design, with a rework cost that is 300 or more times the cost of discovering and correcting the errors earlier [797]. Swanson [812] focused on post-release maintenance issues, and looked beyond low-level error fixing (which he termed *corrective* maintenance) to address the issues that Lehman and Belady raised. His work identified "changes in data and processing environments" as a major cause of *adap-*

tive maintenance activity. Swanson's paper marks one of the first times researchers realized that it was not possible to 'get it right the first time'. In some projects, anticipating everything was essential (safety-critical systems, for example); Swanson's insight was that in other projects this was not cost-effective (although it remained desirable).

Development processes still reflected the engineering mindset of the time, with heavy emphasis on up-front analysis and design. US military standards reflected this, since the idea of interchangeable parts was particularly important for military logistics, and the military had experienced enormous software cost overruns. These pressures were eventually realized as the US government's MIL-STD–498, criticized for insisting on a waterfall approach to software development. Afterwards came the slightly more flexible software process standards IEEE/ISO–12207, and IEEE–830, perhaps the best known standard for software requirements to date. But David Parnas's paper on the "Star Wars" missile defence scheme [679] illustrated the problems with this standard's philosophy, many of which come down to an inability to anticipate future requirements and capabilities, e.g. that "the characteristics of weapons and sensors are not yet known and are likely to remain fluid for many years after deployment" [679, p. 1329]. This demonstrated the massive impact unanticipated change can have on software systems, a central concern of this chapter. Indeed, the US military no longer insists that software be developed according to any particular standard [577, p. 42].

In response to the problems with the waterfall approach, iterative models, such as Boehm's 'spiral' model of development [121] called for iterations over system design, so that requirements were assessed at multiple points. However, such process-oriented models can do little to address unanticipated changes if they do not insist on releasing the product to stakeholders. As Fred Brooks notes, "Where a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time. Hence plan to throw one away; you will, anyhow [144]." The point of Brooks's quote is to emphasize how little one can anticipate the real concerns in designing software systems, particularly novel (for its time) systems like OS/360. Instead, development should be iterative and incremental, where iterative means "re-do" (read 'improve') and increment means "add onto", as defined in [498].

1.2.1 From Software Evolution to Requirements Evolution

This section focuses on that part of software evolution that is concerned with changing requirements or assumptions (i.e., the components of the requirements problem which are in R or W). Historically, some researchers have turned to focus in detail on this relationship between requirements and evolution of software. Not all maintenance activities can be said to result in 'software evolution': for instance, when designers are correcting a fault in the implementation *(S)* to bring it (back) into line with the original requirements (which Swanson called 'corrective maintenance').

Chapin [169, p. 17] concludes that evolution only occurs when maintenance impacts business rules or changes properties visible to the customer.

Harker et al. [363] extended Swanson's work to focus on change drivers with respect to system requirements (summarized in Table 1.1), because "changing requirements, rather than stable ones, are the norm in systems development [363, p. 266]." He characterized changes according to their origins. At this point, requirements engineering as a distinct research discipline was but a few years old, and an understanding was emerging that the importance of requirements permeated the entire development process, rather than being a strictly 'up-front' endeavour.

Table 1.1: Types of requirements change [363]

Type of requirement		*Origins*
Stable	Enduring	Technical core of business
Changing	Mutable	Environmental Turbulence
	Emergent	Stakeholder Engagement in Requirements Elicitation
	Consequential	System Use and User Development
	Adaptive	Situated Action and Task Variation
	Migration	Constraints of Planned Organisational Development

As an aside, it is interesting to ponder whether there is in fact such a thing as an enduring requirement, as defined by Harker et al. A useful analogy can be derived from Stuart Brand's book on architectural change in buildings [138]. He introduces the notion of shearing layers for buildings, which distinguish change frequency. For example, the base layer is *Site*, which changes very little (absent major disasters); *Skin* describes the building facade, which changes every few decades, and at the fastest layer, *Stuff*, the contents of a building, which changes every few days or weeks. The implication for requirements is that good design ought to identify which requirements are more change-prone than others, and structure a solution based on that assumption. There probably are enduring requirements, but only in the sense that changing them fundamentally alters the nature of the system. For example, if we have the requirement for a credit card processing software to connect to the customer's bank, such a requirement is sufficiently abstract as to defy most changes. On the other hand, we can easily foresee a requirement "Connect to other bank using SSL" changing, such as when someone manages to break the security model. We posit that the enduring/changing distinction originates in the abstractness of the requirement, rather than any particular characteristic.

The above taxonomy was expanded by the EVE project [487]. Lam and Loomes emphasized that requirements evolution is inevitable and must be managed by paying attention to three areas: monitoring the operating environment; analysing the impact of the system on stakeholders, or on itself; and conducting risk management exercises. They proposed a process model for systematizing this analysis.

Changes to requirements have long been identified as a concern for software development, as in Basili [76]. Somerville and Sawyer's requirements textbook [787] explicitly mentions ‘volatile’ requirements as a risk, and cautions that processes should define a way to deal with them. Their categorization closely follows that of Harker et al.

Several historical research projects in the area of information systems modeling have touched on evolution. CIM [149] labeled model instances with the time period during which the information was valid. Furthermore, CIM “should make incremental introduction and integration of new requirements easy and natural in the sense that new requirements should require as few changes in an existing model as possible [149, p.401].” Little guidance was given on how to do this, however. In a similar vein, RML [350], ERAE [264] and Telos [628] gave validity intervals for model instances using logic augmented with time arguments. These modeling languages were oriented to a one-off requirements model that can then be used to design the system (rather than allowing on-the-fly updates and inconsistencies during run-time). In other words, these methodologies assume complete knowledge of the system, e.g., the precise periods for which a concept is applicable.

Research has also considered the issue of maintaining consistency in requirements models. Models can be inconsistent when different users define different models, as in viewpoints research. The importance of permitting inconsistency in order to derive a more useful requirements model was first characterized by Easterbrook and Nuseibeh [269]. We return to the use of formal logic for managing evolving requirements in Section 1.4. Zowghi and Gervasi explain that “Increasing the completeness of a requirements specification can decrease its consistency and hence affect the correctness of the final product. Conversely, improving the consistency of the requirements can reduce the completeness, thereby again diminishing correctness [955].” With respect to changes in R, then, there seems to be a tradeoff between making R as detailed as possible and making R as consistent as possible. In early requirements analysis where we suspect W will change (for example, in mobile applications) incompleteness should be acceptable if it supports flexibility - we would rather have high-level consistency with incomplete requirements.

Finally, one could consider the elaboration (i.e., increasing the completeness) of the initial requirements model, from high-level objectives to lower-level technical requirements, as ‘evolving’ requirements (as in [35]); we focus on requirements models for which the elicitation necessary for a first release is assumed to be completed, and then changes, rather than the process of requirements elicitation at an intermediate point in time.

1.2.2 Empirical Studies of Requirements Evolution

The focus of this section is on research projects which conducted empirical studies using industrial Requirements Evolution Problems. Many industrial case studies focus on source code evolution, and little attention is paid to the requirements themselves (which presumably are driving many of the changes to source code). This is typically because requirements are often not available explicitly, unlike source code. This is particularly true in the case of open-source software. Nonetheless, the following studies do show that, when available, the problem of requirements change is important and not particularly well understood.

The SCR (Software Cost Reduction) project at the U.S. Naval Research Laboratory was based on a project to effectively deliver software requirements for a fighter jet, the A–7E. In a retrospective report on the project [23], which includes updates since the initial release in 1972, Chapter 9 of the report lists some anticipated changes. Of interest is that these changes, while anticipated, are not very detailed, and some invariants are assumed (which we would term domain assumptions, such as "weapon delivery cannot be accurate if location data is not accurate"). This early work identified the criticality of understanding how R could and did change, and that such changes needed to be anticipated.

Chung et al. [187] looked at the problem of (non-functional) requirements evolution at Barclays, a large bank. After introducing a modeling notation, they considered the change of adding more detailed reporting on accounts to customers. The paper showed how this change can be managed using their modeling notation, leading to new system designs. In our parlance, they considered changes in all of R, W, S. In particular, the paper focused on tracking the impact of a change on other non-functional properties of the system, such as accuracy and timeliness (i.e., quality attribute requirements in R). Their notation allows analysts to model priorities and perform semi-automated analysis of the updated model. With respect to our property Π, this study used the degree to which a solution $\hat{S}$ satisfied quality attributes as the property over which to evaluate solution optimality. The paper concludes with some principles for accommodating change.

In [36], Anton and Potts looked at the evolution of the features offered by a customer-centric telephony provider. The paper traced, using historical documents, which features were available and how this changed over time. In particular, the paper focused on the consequences of introducing certain features, with the objective of reducing the effort of providing a new service to customers. This survey was end-user oriented as it focused on how features appeared to users of telephone services, not other businesses or the internal feature requirements. Changes in W are related to subsequent changes in S (features are properly parts of the solution), but there is little or no role for explicit members of R, except as reverse-engineered. One can reverse engineer changes in R by inference: if S changes as new features are added, and the authors show it was in response to some initiating change from the domain knowledge W (such as customer usage), then we can infer a change in R (since $W \cup S \vdash R$).

The Congruence Evaluation System experience report of Nanda and Madhavji [632], while not conducted on industrial data, did shed some useful light on the Requirements Evolution Problem. This was an academic-built proof of concept that ultimately failed. In their analysis, Nanda and Madhavji explicitly note that changes in W, which they term "environmental evolution" was a major factor. They particularly note how difficult it was to communicate these changes into direct impacts on the requirements, as they were typically not monitored explicitly.

Anderson and colleagues conducted a series of case studies of changing requirements [31, 32], focusing explicitly on changes in R. Their experiences led to the development of a taxonomy for requirements evolution. The case studies focused on smart cards and air traffic control, and spurred the development of the Requirements Maturity Index, in order to measure how frequently a particular requirement changed. However, the index did not provide for different measures of requirements value or importance, i.e., there was no explicit notion of comparison with respect to Π.

Tun et al. [860] used automated extraction techniques to obtain problem structures from a large-scale open source project (the Vim text editor). They concede that requirements may not be easily extracted, but contend that these problem structures shed some useful light on how one might implement new requirements. These problem structures are essentially triples of W, S, R, with particular focus on looking at S in order to attempt to derive the other two. The challenge with all studies of purely source code (i.e., S) is that one must to some extent speculate about whether changes are coming from the domain knowledge W or from the requirements changing.

There is relevant work in the Mining Software Repositories community on extraction of requirements from project corpora, most recently the work of Hindle et al. at Microsoft [404]. They correlated project specifications to source code commits and were able to identify related changes. In this context, Hindle et al. used the project specification as the representation of R and the code commits as insight into the implementation S. The chief problem with open-source project is that requirements are rarely made explicit. Instead, they occur as user stories or prototyped features. In [279] we looked at techniques for extracting a set R from these issue trackers. See also Chapter 5 later in this book, on repository mining.

Many studies of changing requirements have focused on software product lines. We do not discuss them here, since Chapter 9 goes into them extensively. Herrmann et al [392] used an industrial case study to identify the need for "delta requirements", requirements which must be added subsequent to software delivery, and then took an existing methodology and extended it to incorporate modeling techniques for delta requirements. The advantage of defining delta requirements is that it permits baselining one's core functionality (similarly to product lines), and then extending these as necessary.

Ideally, of course, one would minimize changes before they occur, rather than needing to manage changes afterwards. The issue of changing requirements in the highly formal requirements environment of spacecraft design was considered in [636], with the aim of minimizing the number of downstream changes required, as each change is potentially very costly to re-implement. The authors proposed a technique, semantic decoupling, for modeling requirements R to minimize the dependencies between abstraction levels. In semantic decoupling, one tries to minimize the dependencies between related software modules (S), so that changes in R will not be as wide-ranging. This of course requires a reasonably complete definition of a traceability matrix in order to identify the relationships (which does typically exist in the domains they study).

This sampling of academic case studies of changing requirements in industrial settings has provided some clear examples of the importance of requirements evolution. In domains as varied as spacecraft, smart cards, and phone features, changing requirements are clearly a major concern for business, and the source of much cost and effort.

1.3 A Survey of Industry Approaches

It is useful to consider the treatment of changing requirements in industry settings, as a way to understand the current practices, and how these might inform research proposals. Industrial tools have a strong focus on interoperability with office software like Microsoft Word, because a common use-case for these tools is generating documentation. Furthermore, these tools are not the whole story, as many industry consultants (e.g., [504, 922]) focus as much on managing change through methodology as through tools. This means creating a change process which might incorporate reviews, change tracking, prioritization meetings, and so on.

1.3.1 Standards and Industry

IEEE Standard 830 [421], which describes a standard for "Software Requirements Specification" (SRS), is the culmination of the strict specification approach, what some have derisively called "Big Requirements Up Front". It lays out in great detail the standard way for describing "what" must be built. Section 4.5 of the standard addresses evolution, which it recommends managing using notation (marking requirements as "incomplete") and processes for updating the requirements. As with most standards, this is reasonable in mature organizations, but prone to problems if these key ideas are not followed. Furthermore, completeness and stability are often orthogonal concerns. The standard acknowledges that evolutionary revisions may be inevitable.

1.3.2 Requirements Management Tools

Commercial tools have generally done a poor job supporting change. IBM DOORS[2] and IBM Requisite Pro are document-centric tools whose main interface consists of hierarchical lists of requirements (e.g., "R4.2.4 the system shall . . . "). Traceability is a big feature of such tools, and requirements can be linked (to one another and to other artifacts, such as UML diagrams). Multiple users are supported, and changes

[2] `http://www-01.ibm.com/software/awdtools/doors/`

prompt notification that the requirement has changed. Version control is important: each requirement is an object, and the history of that object is stored, e.g., "modified attribute text" on DATE by USER. In DOORS, one can create requirements baselines which are similar to feature models. One can extend the baseline to create new products or changes to existing projects. It is not clear what the methodology for defining a baseline is.

The tool focus of Blueprint Requirements Center[3] is agile, with strong support for simulation and prototyping. Workbenching requirements scenarios is important in Blueprint. Workbenching or simulation helps analysts understand all the potential variations, as well as giving something concrete to the business user before costly implementation. Blueprint also focuses on short-cycle development, allowing requirements to be broken into sprint-specific stories or features. What both Blueprint and the IBM suite miss, however, is a way to combine requirements management with workbenching integrated into a framework for evaluating change impacts.

1.3.3 Task Managers

An increasingly popular strategy in industry is to forego IEEE specification conformance in favour of lightweight task management tools. This might be described as the agile approach to requirements: treating requirements as tasks that must be carried out. Jira, from Atlassian Software[4], is a commonly-used tool in large-scale projects. Jira allows one to manage what is essentially a complex to-do list, including effort estimation, assignment, and some type of workflow management (e.g., open issue, assign issue, close issue). Similar tools include Bugzilla, Trac, and IBM's Rational Team Concert. More recently, Kanban [30] has made popular visual work-in-progress displays, the most basic of which are whiteboards with lifecycle phases as swimlanes. These tools are well-suited to the deliberate reduction of documentation and adaptive product management that agile methodologies such as Scrum or XP recommend. Leffingwell [503] gives a more structured approach to agile requirements engineering, managing changes using time-boxed iterations (e.g., 2 week cycles) at the boundaries of which the team re-prioritizes the user stories to work on for the next cycle. In this fashion, changes in the domain knowledge W and new requirements R can be accommodated on a shorter time-frame than a model with change requests. This constant iteration only works well with a robust set of tests to verify the requirements were correctly implemented, e.g., using unit and system tests, but as important is some automated acceptance tests using, e.g., Behavior-Driven Development (BDD).

[3] http://www.blueprintsys.com/products/

[4] http://www.atlassian.com/software/jira/

1.3.4 Summary

Particularly for smaller organizations, requirements are not treated at a high level, often existing as an Excel spreadsheet or maintained implicitly [46]. Furthermore, the transition to agile software development has made one of its chief priorities the reduction of unnecessary documentation ("working software over comprehensive documentation"[5]). It is an open and important research question whether omitting at least some form of explicit requirements model is sustainable in the long-term.

The tools we have described work well for managing low-level tasks, such as fixing specific bugs. However, connecting the design and roadmapping component of product management with the specifics of task management is more difficult. While some might use tools like Confluence or other wikis for this knowledge-management task, spreadsheets are still very popular for tracking lists of possible features. What is missing is a higher-level view of "why" changes are being made, and what impact those changes might have on satisfying the requirements. A tool which can preserve the overall requirements model throughout the lifecycle is necessary. That is not to say such an approach could not be integrated into a tool like IBM DOORS. Indeed, there is a lot of work on integrating requirements tools, task managers, code repositories and so on using product lifecycle management (PLM) or application lifecycle management (ALM). The emerging standard for Open Services for Collaboration (OSLC)[6] is one initiative that looks to overcome the traditional silos.

1.4 Recent Research

We now survey some of the latest research in requirements evolution. In many cases, research has focused most on eliciting requirements and potential changes, and less on how such models/representations would be used to adapt software post-implementation. Interest in the notion of requirements at run-time has greatly increased recently, however, and we touch on this below. There are overlaps with work on adaptive software (see Chapter 7.6 later in this book) and model-driven evolution (Chapter 2). To conclude, we introduce two summary tables showing how the individual work addresses elements of the Requirements Problem, as well as an explanation of where gaps exist between research and practice.

[5] http://agilemanifesto.org/

[6] http://open-services.net/

1.4.1 Problem Frames Approach

We mentioned the empirical study of Tun et al. [860] earlier. This work builds on the seminal problem frames approach of Michael Jackson [431] to extract problem frames from existing software systems in order to recover the original requirements. A problem frame captures a particular set of elements involved in the Zave and Jackson approach to the requirements problem: $W,S \vdash R$. For example, the text editor Vim has a feature “Spell Completion”. From the requirement description, Tun et al. reconstruct the problem diagram using problem frame syntax: the requirement is on the right, “complete word automatically”, linked with shared phenomena including “keyboard user” and “buffer”, and finally, to the machine element implementing “Spell Completion” (the feature). Matching related problem diagrams can show feature interaction problems, in this case, where two features both use the shared phenomena of “buffer”. These feature interactions are difficult to manage and can be a large source of problems.

Another project by Tun et al.[859] uses problem frames to identify common problematic design patterns, and to then transform that feature using a catalog. The idea is to support evolution of features using well-known patterns to avoid feature interaction problems. For example, if I know that my buffer is shared by two features, I can apply a pattern like “Blackboard” to solve the potential problems. Similarly, Yu et al. [942] use problem frames in the context of security requirements. Their tool, OpenArgue, supports argumentation about requirements satisfaction that can be updated as more information arrives. As with the requirements evolution problem we defined, this approach seeks to reason about what the implications of this new information are.

1.4.2 Extensions of the NFR Framework

The NFR model, introduced in [186], represented a qualitative approach to modeling system requirements as refinements of high level objectives, called goals. This has been extended to reason about partial goal satisfaction in a number of ways. To begin, Giorgini et al. [327] and Sebastiani et al. [760] formalized a variant of the NFR framework’s qualitative approach, the idea being that qualitative reasoning is better suited to up-front problem exploration. Their tools (e.g., GR-Tool[7]) can reason over qualitative models and generate satisfying alternatives. One can leverage this approach to incrementally explore evolving requirements problems.

What was not well understood was how to turn these into specifications. From the evolution point of view, work on alternatives and variability in goal modeling (e.g., [520], [497]) allows these qualitative models to capture context-driven variability, a point also made in Ali et al. [18], who make the case that requirements variability necessitates the monitoring of the operating contexts. This monitoring

[7] http://troposproject.org/tools/grtool/

information is then used to inform a designer about possible revision options. In all these cases the main contribution to requirements evolution is in eliciting alternative solutions and extending the system specification with annotations for monitoring for violations of these models. Dalpiaz et al. [215] also introduced qualitative requirements variability, but in the area of dynamic reconfiguration. This proposal goes from modeling and elicitation to system specification *over time*, i.e., not just for the initial design but also once the system has been released.

1.4.3 Run-time Adaptive Requirements

Work in the area of adaptive requirements focuses on understanding how to build requirements-based systems that are responsive at run-time to system changes. In particular, the notion of "requirements at runtime", explored in a series of workshops at the Requirements Engineering conference (`requirements-engineering.org`), introduced the notion of using requirements models to drive system changes. See also the chapter on adaptive software later in this book (Chapter 7.6). One thing that is necessary for run-time evolution is the ability to understand what is changing. Qureshi et al. [705] define a set of ontological concepts for managing run-time adaptation to the changes in the requirements problem. The main achievement is the addition of context to requirements problems, in order to suggest variations when contexts change. Another approach is to loosen the formal representation: In the RELAX framework [920], a language is designed to specifically manage "the explicit expression of environmental uncertainty in requirements". When something changes in *W* (the world), for example, a new device appears on a mobile ad-hoc network, the RELAX language can define service levels which satisfy higher level requirements (e.g., "connect to AS MANY devices as possible" as opposed to "connect to ALL devices"). In similar fashion, Epifani et al. [275] use a formal model and a Bayesian estimator to adapt numeric parameters in the specification at run-time. This allows them to set an initial model of the system and then fine-tune various parameters as more information is collected. This is a little like learning the true system requirements, rather than specifying them all at once.

1.4.4 KAOS-based Approaches

A major contribution to the RE literature is the KAOS goal modeling framework, first introduced in [217]. The original focus was on a methodology and tool for goal-based decomposition of requirements problems. The original work has been extended in a number of ways. One direction has considered the importance of alternatives in system design. From an evolution perspective, variability and alternatives support resiliency in two ways when change is encountered. First, the upfront analysis supports enumeration of possible scenarios that need to be handled (for example, the obstacles encountered in [872]). Second, variants can be managed as a form of product line, and called upon if and when things change (see Chapter 9 for more on product lines and requirements). Later work [515] introduced probabilistic techniques for monitoring the partial satisfaction of goals in KAOS models. As designers explore the solution space, numeric measures are used to evaluate the value of a given configuration. Not covered in detail in the paper is how this model would be adjusted at run-time and used to evolve the specification *S*, but some of the KAOS concepts, and in particular its formalism, have found their way into problem frames work. In [871], van Lamsweerde discusses how one might compare alternative models of requirements and systems to be designed therefrom.

1.4.5 Paraconsistent and Default Logics

Several requirements modeling approaches rely on formal logic explicitly (KAOS also uses a formal temporal logic, but it is not the focus of the KAOS-based approaches described above). Here we review two approaches.

Default logic approaches, appearing in [956] and [325], rely on David Poole's Theorist system [691] to define what *must* be (typically the World knowledge) and what *might* change, represented by initial defaults. The connection to the requirements model is two-fold: the selection of the order in which requirements are considered for revision, and the ability to 'downgrade' requirements to default (preferred) status rather than 'mandatory' status. Default logic is non-monotonic in that asserted (TOLD) facts can later be contradicted and no longer concluded; for example, the sentence "requirement R is refined by task T" can be over-ruled if new information is discovered that says, for example, that "requirement R has no refinements". In classical logic, as long as the original sentence remains in the theory, it can be deduced.

Closely aligned with this perspective is the REFORM framework of Ghose [325], which identifies three main properties for a system managing evolution:

1. distinguish between what are called *essential* and *tentative* requirements;
2. make explicit the rationale for satisfying a requirement (refinements);
3. make explicit the tradeoffs for discarding a requirement when the requirements model changes.

Ghose [325] also defines some useful principles for handling changes to the requirements:

1. make *minimal* changes to the solution when the problem changes;
2. make it possible to ignore the change if the change would be more costly than ignoring it;
3. support *deferred commitment* so that choosing a solution is not premature.
4. maintain discarded requirements to support requirements re-use.

They go on to implement these ideas in a proof-of concept system for managing requirements. One issue to consider in such non-monotonic systems for requirements is that reasoning from events to explanations is abductive, and therefore in the NP-hard class of problems. Abductive reasoning is to reason ‘backward’, using observations and a background theory to derive explanations, as opposed to deductive reasoning, which uses a background theory and an explanation to derive possible observations.

Another approach to managing change is *to support paraconsistent reasoning*, that is, reasoning in the presence of contradictory evidence without trivially concluding everything, as in classical logic. This is vital in handling evolving requirements since one common occurrence is that a fact previously asserted as true is then found to be false. For example, stakeholders might indicate initially that requirement R_x must be satisfied, but at a later time, perhaps the stakeholders realized they did not need the requirement. In a formal model we would have $\{R_x, \neg R_x\}$, a classical inconsistency.

In the RE domain, tolerating inconsistency is essential, for reasons listed by Nuseibeh et al. [648]:

1. to facilitate distributed collaborative working;
2. to prevent premature commitment to design decisions;
3. to ensure that all stakeholder views are taken into account;
4. to focus attention on problem areas [of the specification] .

Hunter and Nuseibeh [413] use Quasi-Classical Logic (QCL), an approach to reasoning in the presence of inconsistency which labels the formulas involved. This also permits one to identify the sources of the inconsistency and then, using their principle of “inconsistency implies action”, choose to act on that information, by, for example, removing the last asserted fact, perhaps using principles such as Ghose’s, above. An example from the London Ambulance case has a scenario where, based on the information gathered, one can derive both “dispatch Ambulance 1” and “do not dispatch Ambulance 1”. Two useful capabilities emerge from labeled QCL: one can continue to derive inferences not related to this inconsistency, for example, that Ambulance 2 needs a safety check; and secondly, to understand the chain of reasoning that led to the inconsistency, and resolve it according to meta-level rules.

Our work [278] used paraconsistent reasoning to manage evolving requirements problems. We defined a special consequence relation $\mid\!\sim$ which used a form of maximally consistent subset reasoning to draw credulous conclusions about whether a given high-level requirement could be satisfied, or alternately, which specification to implement in order to satisfy those requirements.

Paraconsistent reasoning (whether using defaults, QCL, or other approaches such as multi-valued logics) supports evolving requirements by mitigating the challenge of conflicting and possibly inconsistent specifications, whether in the World, Requirements, or Specification. While there is a computational complexity concern, practically speaking this is less of an issue as processing speeds have increased.

1.4.6 Traceability Approaches

Traceability refers to the linkages between (in our case) requirements and downstream artifacts such as code, models and tests. The importance of traceability with respect to software evolution is to support the identification and impact of the changes to the requirements. A number of current approaches use traceability to manage requirements evolution.

In the work of Charrada and Glinz [171], outdated requirements are identified based on changes in the source which are believed to impact requirements. Such changes might, for example, include the addition of a new class or method body, which is likely a new feature being added. This addresses the issue of requirements drifting from the actual state of the source code. This approach relies on machine learning techniques to identify statistically likely candidates, which in general falls into the area of mining software repositories. The basic notion is to gather a large body of data, including source code, requirements documents (if any), tests, emails, etc. Machine learning techniques such as Latent Dirichlet Allocation (see Chapter 5) can then be used to extract interesting labels for that data, including which quality requirements are affected, as in Hindle's work [404]. There is some promise in these approaches, but the major stumbling block is to gather useful data in large enough volumes (typically in the millions of records) that the statistical techniques will be sufficiently accurate. As one might imagine, identifying requirements in source code is tricky without a good set of requirements documents to go from.

Should sufficient data not be available, one is forced to leverage human knowledge to a greater extent. Herrmann et al. [392] use the information about the previous incarnation of the system to derive "delta requirements", which specify only those changes to the requirements necessary to implement the system (we might think of this as the set represented by $\delta R \setminus R$). The challenge with this approach is to correctly characterize the existing system's initial requirements.

Welsh and Sawyer [913] use traceability to identify changes that affect dynamically adaptive systems (DAS). They include five possible changes to a DAS that might need to be accommodated:

- environmental change (a change to W_D)
- broken assumption (an incorrect fact in W_D)
- new technology (new elements in S)
- consequential change (changes to the inferences drawn from $W \cup S$)
- user requirements change (changes to R)

Traceability techniques should somehow identify which type of change is occurring and what implications that change has for the other elements of the system. Welsh and Sawyer extend the i* strategic rationale framework [938] to annotate the models with possible changes and impacts. The primary contribution is to support elicitation and modelling.

1.4.7 Feature Models

Feature models are covered in greater detail in Chapter 9. Techniques for dealing with changes to feature models, including the product lines which are typically derived from the feature models, overlap with the management of requirements evolution. Requirements researchers typically consider feature models to focus on the user-oriented aspects of a system, i.e., be designed with marketable units in mind. Requirements as we define them, however, would also consider non-functional properties of the system (which are not necessarily user-oriented, such as maintainability) and features which may not be relevant to product lines.

That being said, the techniques for managing evolution in feature models are relevant to requirements evolution as well.

1.4.8 Summary

Table 1.2 is a summary of the focus of the approaches discussed, based on whether the approach emphasizes changes in domain knowledge W_D, specification/implementation S, requirements R, or some property Π that can be used to compare approaches. The most glaring omission are techniques for quantifying the difference between various solutions (that is, defining properties Π), although this work has been the subject of work in search-based software engineering (Chapter 4). Applying optimization techniques like genetic algorithms to the Requirements Evolution Problem seems a fruitful area of research.

We said at the beginning of this chapter that managing evolving requirements could be broken down into elicitation and modeling and turning those representations into software. Most of the approaches we discussed focus on the modeling and analysis aspects of the problem. There is unfortunately little work on taking the frameworks and applying them to industrial requirements problems. Part of the challenge is that a lot of industries simply do not manage requirements in a manner which would permit, for example, delta requirements to be generated. Another is that academic tools for the most part ignore the vastly different scale of industrial challenges (Daimler, for example, has DOORS models with hundreds of thousands of objects).

An emerging trend in requirements evolution is the linkage to dynamic, self-adaptive systems (cf. Chapter 7.6). Researchers are increasingly looking beyond the traditional staged lifecycle model where requirements are used to derive a specification and then ignored. Instead, requirements, and other models, are believed to be useful beyond the initial release of the software. Most of the research to date has identified challenges and future work that must be dealt with before we can realize the vision of “requirements at run-time”. For example, Welsh and Sawyer [913], Ghose [325], and several others focus on understanding the nature of the problem using classification techniques.

Table 1.2: Research approaches compared with respect to the Requirements Evolution Problem components. (●: covered; ○: partial coverage; –: not covered, na: tool mentioned but not available.)

Approach	**Paper**	Domain changes δW	Specification changes δS	Requirements changes δR	Solution comparison Π	Tool support	Empirical evidence
Problem Frames	Tun et al. [859]	●	●	●	–	–	–
	Tun et al. [860]	○	●	●	–	[a]	●
	Yu et al. [942]	○	●	●	–	[b]	○
NFR extensions	Sebastiani et al. [760]	●	○	●	●	[c]	○
	Lapouchnian et al. [497]	●	–	●	●	–	○
	Ali et al. [18]	●	–	●	●	–	–
	Dalpiaz et al. [215]	●	–	●	●	–	●
Adaptive Requirements	Qureshi et al. [705]	○	○	●	○	–	–
	Whittle et al. [920]	●	●	●	●	–	●
	Epifani et al. [275]	●	●	●	●	–	●
KAOS-based	van Lamsweerde and Letier [872]	●	–	●	○	[d]	●
	Letier and van Lamsweerde [515]	●	○	●	●	[d]	●
	van Lamsweerde [871]	○	–	●	●	[d]	○
Paraconsistent	Zowghi and Offen [956]	○	–	●	●	na	–
	Ghose [325]	○	–	●	○	–	–
	Hunter and Nuseibeh [413]	●	○	○	–	na	○
	Ernst et al. [278]	●	–	●	○	[e]	●
Traceabiity	Charrada and Glinz [171]	○	●	●	○	–	○
	Hindle et al. [404]	○	●	●	–	–	●
	Herrmann et al. [392]	○	○	●	●	-	●
	Welsh and Sawyer [913]	●	○	●	●	[f]	●

[a] http://mcs.open.ac.uk/yy66/vim-analysis.html

[b] http://sead1.open.ac.uk/pf

[c] http://troposproject.org/tools/grtool/

[d] http://www.objectiver.com/index.php?id=25

[e] http://github.com/neilernst/Techne-TMS

[f] http://is.gd/7wP7Sj

One of the seminal papers in characterizing evolutionary systems is that of Berry et al. [100]. In it, the authors argue that for dynamically adaptive systems requirements engineering is continuous. Systems must understand what objectives are paramount at a given point in time, what constraints exist, and how to act to achieve their objectives. They therefore argue for four levels of adaptivity:

Level 1 Humans model the domain, W and possible inputs to a system S.

Level 2 The given system S monitors W for its inputs and then determines the response. This is the domain of self-adaptive software research, such as Epifani et al. [275] or Whittle et al. [920].

Level 3 Humans identify adaptation elements for the set of systems. This is what variability modeling for goal models does, for example Dalpiaz et al. [215].

Level 4 Humans elicit mechanisms for self-adaptation in general.

Berry et al.'s classification allows us to understand the general trajectory for research into requirements evolution. It moves beyond level 1, which is interested in inputs and outputs for a specific system, increasingly focusing instead on adapting and evolving the software *in situ*, based on a set of observations and a formalism for responding to the inputs. Unknown unknowns, the inputs not modeled for reasons of either cost or ignorance, will still bring us back to level 1.

While this is encouraging in terms of software that will be more resilient, one question that is commonly left unanswered in research is the issue of responsiveness. Particularly in formal analysis, we can quickly run up against fundamental complexity results that render complete optimal solutions infeasible, since exponential algorithms seem to be the only alternative. While the performance of such algorithms has improved with advances in inference engines, processing speed and parallelization, it is very much an open question as to how much analysis is possible, particularly in the 'online' scenario. Hence, a number of researchers focus on incremental or partial approaches to analysis. It is important to keep in mind how well the proposed evolved design will work under realistic scenarios.

Table 1.3 presents a matrix of where research is needed in requirements evolution. These are common RE research themes (see Nuseibeh and Easterbrook [647], Cheng and Atlee [178] or van Lamsweerde [870] for an overview of requirements engineering research in general) but in this case specifically to do with evolving requirements. We rank the challenges according to industry adoption and interest, existing research interest, and finally, challenges and obstacles to further research. One particularly important area, emerging from industry, is a general need for more applied and empirical studies of requirements evolution, in particular in agile projects using lightweight tools. Adaptivity is another emerging research area; in this table, the research area most under-served is understanding how Requirements Evolution Problems can be elicited and analyzed, i.e., what future capabilities will a system require, and how to monitor and understand the possible changes that might occur.

Table 1.3: Research opportunities in requirements evolution (**H** - High, **M** - Moderate, **L** - Low)

Research area (*Evolution and...*)	**Industry adoption**	**Research interest**	**Challenges**
Elicitation	**M** numerous approaches	**L** Most approaches focus on stable systems.	Longitudinal case studies required.
Analysis	**L** conducted by business personnel	**L** most work focuses on adaptation - what to do next.	Understanding system context.
Modeling	**L** models mostly informal	**M** numerous studies of formalization of change, e.g. [277]	Scalability; ease of use by industry.
Management	**M** most tools support some form of impact analysis, but at a simple level.	**H** Numerous frameworks and techniques.	Study mainly on greenfield systems. Little empirical validation.
Traceability	**H** widely seen as important.	**H** see traceability workshops e.g. [693]	Trace recovery; scalability
Empirical research	n/a	**M** - increasing amount of empirical validation in research papers	Reality is messy; Industry reluctance to share data

1.5 A Framework for Requirements Evolution

This section considers strategies for managing Requirements Evolution Problems (REP) at the next stage in the software lifecycle: implementation. In our work, we represent requirements as goals G according to goal-oriented requirements engineering [870]. Recall the definition of the Requirements Evolution Problem needed to relate changes in requirements R, domain knowledge W_D, and solutions S to the existing solution (or find such a solution, if there isn't one). In any event we supported solution comparison by the use of a desired property Π, relating $\hat{S}$ to S_0 and possibly other aspects of the changes. Π, in other words, allows us to define a partial order over potential $\hat{S}$.

We will store instances of these elements (i.e., a specific goal instance such as "system will allow user registration") in a knowledge base called REKB. The REKB can answer questions about the items stored in it. In [277] we discuss this in more detail, but the essential operations (ASK questions in knowledge base parlance) include:

ARE-GOALS-ACHIEVED-FROM Answers True if a given set of tasks in REKB can satisfy a given set of goals. This is the "forward reasoning" problem of Giorgini et al. [327].

MINIMAL-GOAL-ACHIEVEMENT REKB discovers Σ, the set of sets S of tasks which minimally satisfy desired goals. This is the (abductive) backward reasoning problem of Sebastiani et al. [760].

GET-MIN-CHANGE-TASK-ENTAILING This operation produces solutions to the Requirements Evolution Problem. Given an initial solution S and a set of desired goals, find a set of minimal sets of tasks which are not dominated for some criterion of minimality and satisfy the new requirements problem. The key issue is to define what the minimality criterion might be for a new solution, which we discuss below.

We concentrate on those changes which are unanticipated. By ‘unanticipated’ we mean that there exists no mechanism in the implementation specification *S* to accommodate these changes. This clearly positions the Requirements Evolution Problem as quite distinct from the Self Adaptation Problem (SAP). With respect to the aspects of *G*, *W*, and *S*, the SAP is to accommodate changes in *W,G* by creating a suitably adaptive $\hat{S}$ *ab initio*. In other words, unlike the Requirements Evolution Problem, self-adaptation approaches do not modify requirements themselves, but rather choose predefined tasks in *S* to accommodate the changes. This is what RELAX [920] is doing: using a fuzzy logic to define criteria for satisfying requirements that may be accomplished by different tasks/monitors.

It is also becoming clear that with a suitably flexible framework and a wide pool of services, an adaptive specification can be used to select services that satisfy our changed goals or domain assumptions. Since these services can be quite heterogeneous, there is a continuum between *adapting* and *evolving*. The essential distinction is the extent to which the changes are anticipated in the implementation.

There are two key concerns in the Requirements Evolution Problem:

1. What do we do when new information contradicts earlier information? This is the problem of **requirements problem revision**.
2. What solutions (sets of tasks) should we pick when the REKB has changed and been revised? This is the problem of **minimal solution selection**.

We discuss these below, after introducing our motivating case study.

1.5.1 The Payment Card Industry Example

As we discussed in Section 1.1, the Payment Card Industry Security Standards Council is an industry consortium of payment card issuers, including Visa and Mastercard. This body has responsibility for developing industry standards for securing cardholder data. The data security standard (DSS) takes effect whenever a merchant processes or stores cardholder data. The standard is updated every three years, and there are three versions extant, which we have modeled for our case study. Among the high level PCI-DSS requirements are goals of protecting cardholder data, securing one’s network, and using access control measures.

We modeled a scenario where a football stadium was upgrading its payment card infrastructure. The stadium model captured had 44 nodes and 30 relations; the PCI-DSS had 246 goals, 297 tasks/tests, and 261 implications (connections between goals and tasks). Our case study captured three general circumstances of change: expansion, contraction, and revision, which we focus on here.

Most of the changes to the PCI-DSS, particularly those smaller in scope, are to clarify previous requirements or to amend erroneous or irrelevant requirements. This is exactly why requirements evolution is a problem: as the standard is implemented, it becomes clear which parts are unsuited to the real-world problems of payment card security. Often, unfortunately, this evolution occurs because a hacker discovered an unanticipated hole in the security. In some sense, the standard is always one step behind these attacks. The following examples show how the standard was revised:

1. Version 1.2 of the standard required organizations to change security keys (that is, electronic keys) annually. However, in some cases it became clear that this was either too onerous, or too infrequent. Version 2.0 of the standard therefore revised this requirement to ask that organizations change keys according to best practices. Note the ambiguity in this last phrase.
2. Similarly, previous versions of the standard asked organizations to use cryptography. However, cryptography means many things, so this was updated to demand the use of strong cryptography. Consider the situation in which we (as stakeholders) had decided to use a 56-bit encryption protocol (which is easily broken). We now have to update this to a newer protocol, such as Triple-DES. This switch may conflict with our choice of technology from before, and requires us to drop support for a particular task (which would otherwise lead to an inconsistency).
3. In previous iterations of the standard, secure coding guidelines had to be followed, but only for web applications such as web pages. In the latest version, this has been revised to demand these guidelines apply to all applications. Again, this might require our IT system to change coding practices, implement testing frameworks, or hire consultants to be consistent with this revision.

We then applied the methodology described below to find solutions to these revisions in a reasonable amount of time. This is what might occur if, for example, an organization needed to understand what testing to carry out to ensure compliance: the tasks the REKB identified using GET-MIN-CHANGE-TASK-ENTAILING would correspond to the validation tests identified in the PCI-DSS standard. More information on the case study is available in [277].

1.5.2 Methodological Guidance for Solving Unanticipated Changes

Since the focus of the Requirements Evolution Problem is changing systems, it behooves us to outline the process by which these changes occur, as well as the impact the changes have on the requirements problem. Figure 1.2 outlines these steps in graphical form.

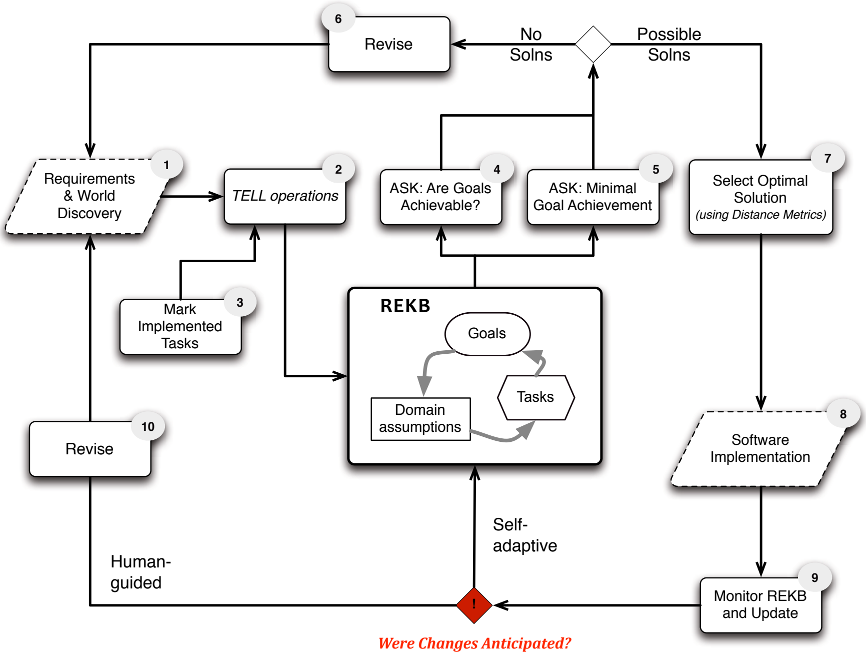


Fig. 1.2: A methodology for Requirements Evolution Problems

Step 1. Elicit requirements from stakeholders and map the speech acts into domain assumptions W, goals in R, and solution tasks in S. Define domain assumptions that are relevant to the context of the particular company. For instance, if one is working with a payment processor (like Verifone) for a 1,200 terminal soccer stadium, one will want to add the details of the Verifone-specific constraints. At the same time, identify relevant problem modules. In the case study this is the set of applicable standards and regulations: the PCI-DSS, Sarbanes-Oxley, etc. For example, requirements 1 and 1.1 of the PCI DSS could be represented as the goal G_1: "Install and maintain a firewall configuration to protect cardholder data" and goal $G_{1.1}$: "Establish firewall and router configuration standards", along with the domain assumption $W_1 : G_{1.1} \rightarrow G_1$.

Step 2. 'TELL' this requirements problem to the REKB, introducing the goals and tasks as atoms and asserting the domain assumptions.

Step 3. Identify existing implemented tasks and add to the REKB, marking them as "implemented". Rather than defining future tasks to be performed, we need to check whether the requirements problem can already be satisfied. In the first iteration, this is clearly unlikely, but in future iterations it may be possible.

Step 4. These previously implemented tasks will be the initial input for the ARE-GOALS-ACHIEVED-FROM operator. This step is essential to prevent over-analysis: if we have a set of tasks that already solve the (new) problem, just use those. This is where the difference between adaptation (existing implementation solves the new problem) and evolution begins.

Step 5. If no candidate solutions were discovered in Step 4, then we must analyze the REKB using MINIMAL-GOAL-ACHIEVEMENT. That operation returns Σ sets of S. In the case of the PCI-DSS, this means finding (zero or more sets of) some set of tests which will satisfy the goals captured in the standard, and in particular, the goal "comply with the PCI DSS".

Step 6. If the model is not satisfiable, repeat the elicitation steps to revise the REKB. This is the process of refining our REKB in order to solve the Requirements Problem.

Step 7. Once we have Σ, which is a set of 'candidate solutions', decide on a member of Σ using decision criteria (Π). There are several possibilities, including one, maximize the number of preferred goals contained. Two, minimize a distance function between existing tasks and new tasks. Here we would make use of the previously implemented tasks for earlier versions of the system implementation.

Step 8. Implement the initial solution to the Requirements Problem as RP_1.

Step 9. Monitor the implementation, domain, and goals for changes. This can be done using e.g., awareness requirements [788].

Step 10. Something has changed (i.e. in W or G) and the system (S) can no longer satisfy our goals. We must re-evaluate the Requirements Problem to find a solution that will. We update the REKB with new information and repeat from Step 2.

The diamond with exclamation mark reflects the key distinction between a Requirements Evolution Problem and a Self-Adaptation Problem. If the detected change (step 9) was anticipated, then we can look to the current version of the REKB. Assuming the design was properly instantiated, this ought to provide a new solution from within the initial REKB. However, as in Berry et al. [100], if there is no solution in the initial REKB, we must intervene as humans and revise the REKB accordingly.

This is a high-level methodology: variants are possible, of course. For one, we could select more than one solution in Step 7 in order to maximize flexibility. Step 7 might also be expanded to reflect software product line development.

1.5.3 Revising Requirements

We mentioned that one of the key concerns in the Requirements Evolution Problem is how to manage new information which contradicts existing information. Steps 6 and 10 of the Requirements Evolution Problem methodology is predicated on revising the REKB when new information is found (assuming the REKB and its revision are consistent). We can draw on the research into belief revision in knowledge representation with a few caveats. Most importantly, it has long been argued that the context of revision is important. For example, the difference between bringing a knowledge base up to date when the world changes (update) and revising a knowledge base with new information about a static world was outlined in Katsuno and Mendelsohn [453]. This is an important distinction in RE as well.

According to one seminal approach to belief revision, the AGM postulates [16], there are three key principles:

1. the use of "epistemic entrenchment" to partially order the formulae, in order to decide which to give up when revising the belief set;
2. the principle that the "new information" φ ought to be retained in the revised belief set;
3. information should be discarded only if necessary (minimal mutilation or information economy), because obtaining it is expensive.

The problem with these principles for the REKB is that **a)** we are dealing with three distinct sorts of well-formed formulas (wffs) (namely, goals R, specifications S and domain assumptions W_D) and **b)** our central concern is solving the requirements problem. This last point distinguishes the REKB version of revision: the concern of classical revision is the state of an agent's beliefs (e.g., that it is raining rather than sunny); the concern of requirements revision is how best to incorporate the new information in order to solve the modified requirements problem. In this formulation, the new information may in fact be rejected, whereas in AGM revision, this is never the case.

For example, consider the case where we are told that the stakeholders have a new goal: to support VISA's touchless card readers[8]. The AGM postulates would have us accept this new fact on the principle that recent information is dominant over older information. In the Requirements Evolution Problem, before accepting new information we must understand its implications with respect to solving the requirements problem. Consider the case where our soccer stadium already supports the goal of "accept touchless payment cards". If the designers are told a new customer goal is to "require signatures on payments", we can see there is a conflict, which implies the REKB must be revised (absent paraconsistent reasoning). The AGM postulates would say that the new goal is paramount, and that the old goal be rejected (or a workaround devised). In a design situation, however, this new goal may be illogical, and should itself be rejected. In this situation the best we can do is ask for preferences between these conflicting goals. We reject it not because it imperils the current solution, but because it conflicts with other goals in the sense that we cannot solve them simultaneously.

This leads to a new definition of revision in the REKB formulation of the requirements problem. When domain assumptions change, since these are invariant by definition, we apply standard belief revision operators to those wffs. For example, if previously we had believed that "50% of the clientele possess touchless credit cards", and after monitoring sales for a few months, our statistics inform us that the figure is closer to "90%", it seems intuitive to accept the new information. In this case, our domain assumptions are ordered using an epistemic entrenchment relation.

[8] A touchless card reader is referred to as PayPass or PayWave, and does not require a swipe or insertion for low-value transactions.

For goals and specifications, we have broad freedom to reject changes. Our motivation for deciding on the particular revision to accept is with respect to the requirements problem. We prefer a new state of the REKB that brings us better solutions. The definition of 'better' solution will be defined with respect to the distance function we use in GET-MIN-CHANGE-TASK-ENTAILING, i.e. Π. This means that even if stakeholders inform us of new tasks that have been implemented, or new goals they desire, we may reject these revisions if they do not lead to a better solution. This might be the case if, as with the previous example, stakeholders tell us that they have upgraded the payment terminals to accept touchless payments. It may be that this is now possible, but still does not satisfy other goals in our REKB. This ability to reject the most current revision, unlike classical belief revision, means that revising requirements problems is properly aligned with our definition of the REKB as a support mechanism for design decisions.

1.5.4 Selecting Non-Dominated Solutions

The second challenge in requirements evolution was deciding what solutions to select when the REKB has changed and been revised. Recall the GET-MIN-CHANGE-TASK-ENTAILING operator takes a set of goals and a set S_0 of tasks, the old implementation, and returns a set of sets of tasks Σ which are equally desirable (non-dominated) solutions to the requirements problem with respect to a distance function. The important consideration in choosing new solutions is the choice of a distance function (i.e., Π above), so let us examine some possible choices.

Requirements re-use is important, so we do not want to completely ignore previous implementations in selecting a new solution. That suggests there are properties of a new solution with respect to the old one that might be useful heuristics for the decision. We defined several properties Π in [277], together with illustrative examples based on a case where: $S_0 = \{a,b,c,d,e\}$ was the initial solution (the set of tasks that were implemented); and $S_1 = \{f,g,h\}, S_2 = \{a,c,d,f\}$ and $S_3 = \{a,b,c,d,f\}$ are minimal sets of tasks identified as solutions to the new requirements:

1. *The standard solutions*: this option ignores the fact that the new problem was obtained by evolution, and looks for solutions in the standard way. In the example, one might return all the possible new solutions $\{S_1,S_2,S_3\}$, or just the minimum size one, S_1.
2. *Minimal change effort solutions*: These approaches look for solutions $\hat{S}$ that minimize the extra effort $\hat{S} - S_0$ required to implement the new "machine" (specification). In our view of solutions as sets of tasks, $\hat{S} - S_0$ may be taken as "set subtraction", in which case one might look for (i) the smallest difference cardinality $\mid \hat{S} - S_0 \mid$ (S_2 or S_3 each requires only one new task to be added/implemented on top of what is in S_0); or (ii) smallest difference cardinality *and* least size $\mid \hat{S} \mid$ (S_2 in this case).
3. *Maximal familiarity solutions*: These approaches look for solutions $\hat{S}$ that maximize the set of tasks used in the current solution, $\hat{S} \cap S_0$. One might prefer such

an approach because it preserves most of the structure of the current solution, and hence maximizes familiarity to users and maintainers alike. In the above example, S_3 would be the choice here.

4. *Solution reuse over history of changes*: Since the software has probably undergone a series of changes, each resulting in newly implemented task sets $S_0^1, S_0^2, \ldots, S_0^n$, one can try to maximize reuse of these (and thereby even further minimize current extra effort) by using $\bigcup_j S_0^j$ instead of S_0 in the earlier proposals.

The above list makes it clear that there is unlikely to be a single optimal answer, and that once again the best to expect is to support the analyst in exploring alternatives.

1.5.5 Summary

This framework supports the iterative 'exploration' of one's requirements, domain knowledge, and solution. As analysts, one can ASK questions of the REKB and understand how complete or accurate the solution will be. Furthermore, using GET-MIN-CHANGE-TASK-ENTAILING, iterating and incrementing the solution itself, particularly in response to change, happens continuously, as new information is added to the REKB. It focuses on requirements as first-class citizens of software evolution and tries to reconcile the satisfaction of those requirements by a suitable software specification, respecting domain knowledge.

In addition to the methodology, we also need to track and version our artifacts using metaphors from version control (e.g., diff, checkin, etc.). We would also like our REKB to be scalable to models of reasonable size: in a related paper [277], we showed incremental reasoning was possible 'online', i.e., in less than 10 seconds.

Related work includes:

- The 'cone of uncertainty', which captures the idea that before the project begins uncertainty about exactly what the requirements are is quite broad. The cone narrows as the project progresses and we have more information about the relevant requirements.
- Levels of knowledge, including 'known knowns', 'known unknowns' (changes we anticipate), 'unknown knowns', or tacit knowledge, and 'unknown unknowns', possible changes we are not aware of. These illustrate the major challenge in requirements evolution. However, being aware of these levels of knowledge is already a major achievement in most projects.
- The Architecture Tradeoff Analysis Method (ATAM), which incorporates analysis of sensitivity points and trade-offs against non-functional quality attributes of the software. This is a scenario exploration technique designed to test a possible design against changes.
- The V-model (and all other related testing approaches) which insists that requirements are reviewed early by the test team in order to ensure testability.

1.6 Conclusions

In this chapter, we have made the point that focusing solely on implementation artifacts is insufficient. It is the requirements which are providing the guidance for maximizing stakeholder value, and so understanding, modeling, and reasoning about evolving requirements is extremely important. We discussed how research in software evolution led to research in requirements evolution, and showcased some of the industrial and academic approaches to managing requirements evolution. The previous section on the REKB defined our approach to the requirements evolution problem: that of incremental exploration of the problem space using the REKB as a form of workbench. Since we expect our system to be subject to change pressures, and constantly evolving, supporting exploration allows both an initial problem exploration, as well as a revision when something previously established changes.

The vision for managing the Requirements Evolution Problem is growing closer to the vision of adaptive software systems (Chapter 7.6). In both cases, we would like to support rapid and assured changes to a software system, and in the adaptive case, without human intervention. To date, the primary difference is in which artifacts are in focus. For requirements at run-time, the requirements model is viewed as the driver of system understanding. At runtime we need to monitor how the system is doing with respect to its requirements. This is best done by comparing the execution (trace) to a runtime version of requirements, rather than a runtime model of the implementation. The implementation is responsible for the actual system. However, in answering questions about how the system is performing, e.g. with respect to quality attributes or high-level goals, one can only answer these questions (directly anyway) by understanding the state of execution of requirements. Requirements evolution is a complex problem, but supporting incremental and iterative analysis of the requirements model will help us in making software in general more adaptable and efficient.

Chapter 2
Coupled Evolution of Software Metamodels and Models

Markus Herrmannsdörfer and Guido Wachsmuth

Summary. In model-driven software engineering, software models are the primary engineering artifacts which are built using dedicated modeling languages. In response, modeling languages are receiving increased adoption to improve software engineering in industry, and thus their maintenance is gaining importance. Like software, modeling languages and thus their metamodels are subject to evolution due to changing requirements. When a metamodel evolves, models may no longer conform to it. To be able to use these models with the new modeling language, they need to be migrated. Metamodel evolution and the corresponding model migration are coupled. In this chapter, we introduce the problem, classify it, and discuss how it can be addressed. To get a feeling about the extent of the problem in practice, we give an overview of the empirical results that are available in the literature. We then present different approaches to the problem and classify them according to a number of features. Finally, we give an overview of the available tools and compare them to each other, before we conclude the chapter with directions for future work.

The following people influenced the presented work on the coupled evolution of metamodels and models: Sebastian Benz, Elmar Jürgens, Maximilian Kögel, Daniel Ratiu, Louis M. Rose, Sander Vermolen, and James Williams. We owe our gratitude to the following institutions for supporting this line of research: BMW, the German Federal Ministry of Education and Research (BMBF), NWO/JACQUARD, and Oracle.

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_2, © Springer-Verlag Berlin Heidelberg 2014

2.1 Introduction

In software engineering, various artificial languages are employed. This includes general-purpose programming languages such as Java [342], domain-specific languages such as HTML [929], general-purpose modeling languages such as the UML [652], domain-specific modeling languages such as BPMN [653], data formats such as SVG [930], and ontologies such as the Web Service Modeling Ontology [725]. *Software language* has been established as the overall term for such languages [471].

Software languages evolve [286]. A software language, as any other software artifact, is designed, developed, tested, and maintained. Requirements, purpose, and scope of software languages change, and they have to be adapted to these changes. This applies particularly to domain-specific languages [302] that are specialized to a specific problem domain, as their specialization causes them to be vulnerable with respect to changes of the domain. But general-purpose languages like Java or the UML evolve, too. Typically, their evolution is quite slow and driven by heavy-weighted community processes. For example, the last formally released version 2.4.1 of UML dates back to August 2011, while the upcoming version 2.5, a minor revision to the 2.4.1 version, is in its finalization phase since October 2012.

Software language evolution implies a threat of language erosion [285]. Typically, language processors and tools no longer comply with a changing language. But we do not want to build language processors and tools from scratch every time a language changes. Thus, appropriate co-evolution strategies are required. In a similar way, language utterances like programs or models might become inconsistent with a changing language. But these utterances are valuable assets for software developers, making their co-evolution a serious issue.

Software language engineering [471] evolves as a discipline to the application of a systematic approach to the design, development, maintenance, and evolution of languages. It concerns various technical spaces [485]. Software language evolution affects all these spaces: Grammars evolve [492], metamodels evolve [285], XML schemas evolve [491], database schemas evolve [605], ontologies evolve [296], and APIs evolve [249], too. In this chapter, we focus on the evolution of metamodels.

2.1.1 Metamodels and Models

In model-driven software engineering, software models are the primary engineering artifacts. From these software models, other artifacts like e. g. the code implementing the software are generated. Software models are expressed by means of modeling languages. This includes general-purpose languages like the UML as well as domain-specific modeling languages [205, 471]. In modelware [106], the technical space of modeling languages, technologies are organized around *metamodels*. A metamodel is an *intensional definition* of a modeling language. It specifies the abstract syntax of the language. Models expressed in a modeling language need to

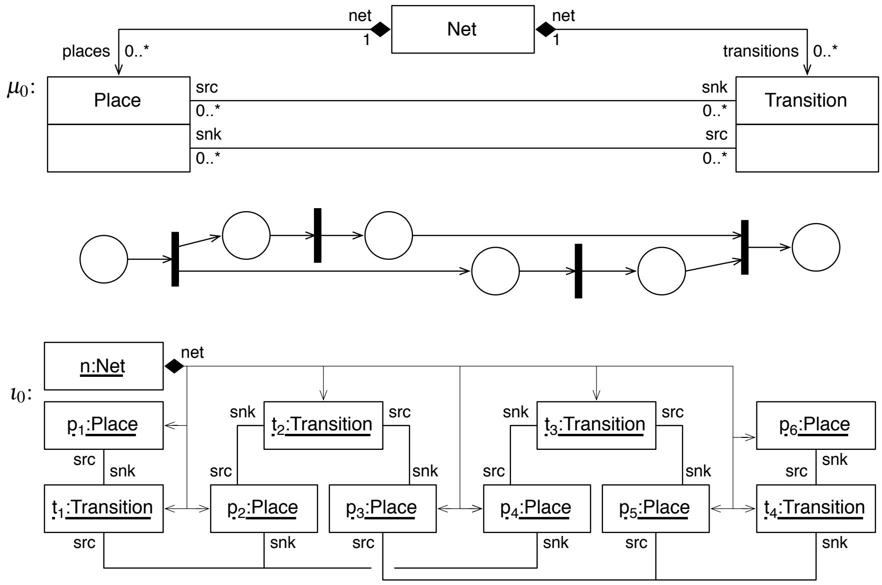


Fig. 2.1: Metamodel μ_0 for Petri net models (top) and compliant instance model ι_0 in concrete syntax (mid) and abstract syntax (bottom).

conform to the metamodel of this language, that is, they need to obey the syntactic rules defined by the metamodel. The *extension* of a metamodel is the set of models which conform to it.

With its MetaObject Facility (MOF) [654], the OMG provides two standard description means for metamodels: Essential MOF (EMOF) as a minimal specification and Complete MOF (CMOF) as an extension to the former one. EMOF is rooted in the UML and reuses its basic concepts like packages, classes, properties, and associations. Additionally, CMOF offers sophisticated features like property redefinition, union sets, and package merge.

Example 2.1 (Petri net metamodel and compliant instance). The class diagram in Figure 2.1 shows an EMOF compliant metamodel μ_0 for Petri net models. A Petri `Net` consists of `places` and `transitions`. Connections between places and transitions are modeled as two associations from `Place.src` to `Transition.snk` and from `Transition.src` to `Place.snk`. Figure 2.1 also shows an example Petri net model ι_0 in a common graphical concrete syntax and a corresponding object diagram which captures its abstract syntax. The abstract syntax model complies with metamodel μ_0. Every object instantiates a concrete class in μ_0. Similarly, every link instantiates an association while obeying cardinalities.

2.1.2 Metamodel Evolution

Modeling languages evolve frequently to meet the requirements of their users. In this chapter, we are particularly interested in changes, where the abstract syntax of a language, thus its metamodel, evolves. This happens whenever new features are added to a language, obsolete features are removed, or the internal structure of models is changed.

Example 2.2 (Petri net metamodel evolution). Figure 2.2 shows evolved versions of the original metamodel μ_0 from Figure 2.1. Changed metamodel elements are formatted in bold text with a gray background. Since a Petri net without any places and transitions is of no avail, we restrict `Net` to comprise at least one place and one transition. This results in a new metamodel μ_1. In a next step, we make arcs between places and transitions explicit. This step might be useful if we want to annotate metaclasses with a means for graphical or textual description in order to assist automatic tool generation. The extraction of `PTArc` and `TPArc` yields μ_2. As `PTArc` and `TPArc` both represent arcs, we state this in μ_3 with a generalisation `Arc`. In an extended Petri net formalism, arcs might be annotated with weights. We can easily reproduce this extension by introducing a new attribute `weight` in μ_4. Until now, we cover only static aspects of Petri nets. To model dynamic aspects, places need to be marked with tokens as captured in μ_5.

2.1.3 Model Migration

When the metamodel of a modeling language evolves, existing models might no longer conform to this metamodel. Model migration is needed to keep models conform to their metamodel. Just as models need to conform to their metamodel, a model migration needs to conform to its corresponding metamodel evolution. Model migration *co-evolves* models with their evolving metamodel [285], resulting in a *coupled evolution* of metamodels and models. Formally, a coupled evolution can be defined as a triple (μ, μ', m) of the original metamodel μ, the evolved metamodel μ', and migration m, a partial function from the extension of μ to the extension of μ'.

Example 2.3 (Petri net model migration). Reconsider the evolution of the Petri net metamodel from the previous example. In the first step, Petri net models without any places or transitions no longer conform to μ_1. This was actually intended and such models will not be migrated. All other models (including ι_0 from Figure 2.1) still conform to μ_1. In the second step, links between places and transitions no longer conform to μ_2. All models can be migrated by replacing such links with a link between the place and a newly created instance of either `PTArc` or `TPArc` and a link between the transition and the new object. For example, ι_0 does no longer comply with μ_2 and needs to be migrated. Figure 2.3 shows the resulting compliant model ι_2. In the third step, all models (including ι_2) still conform to μ_3. In the fourth step,

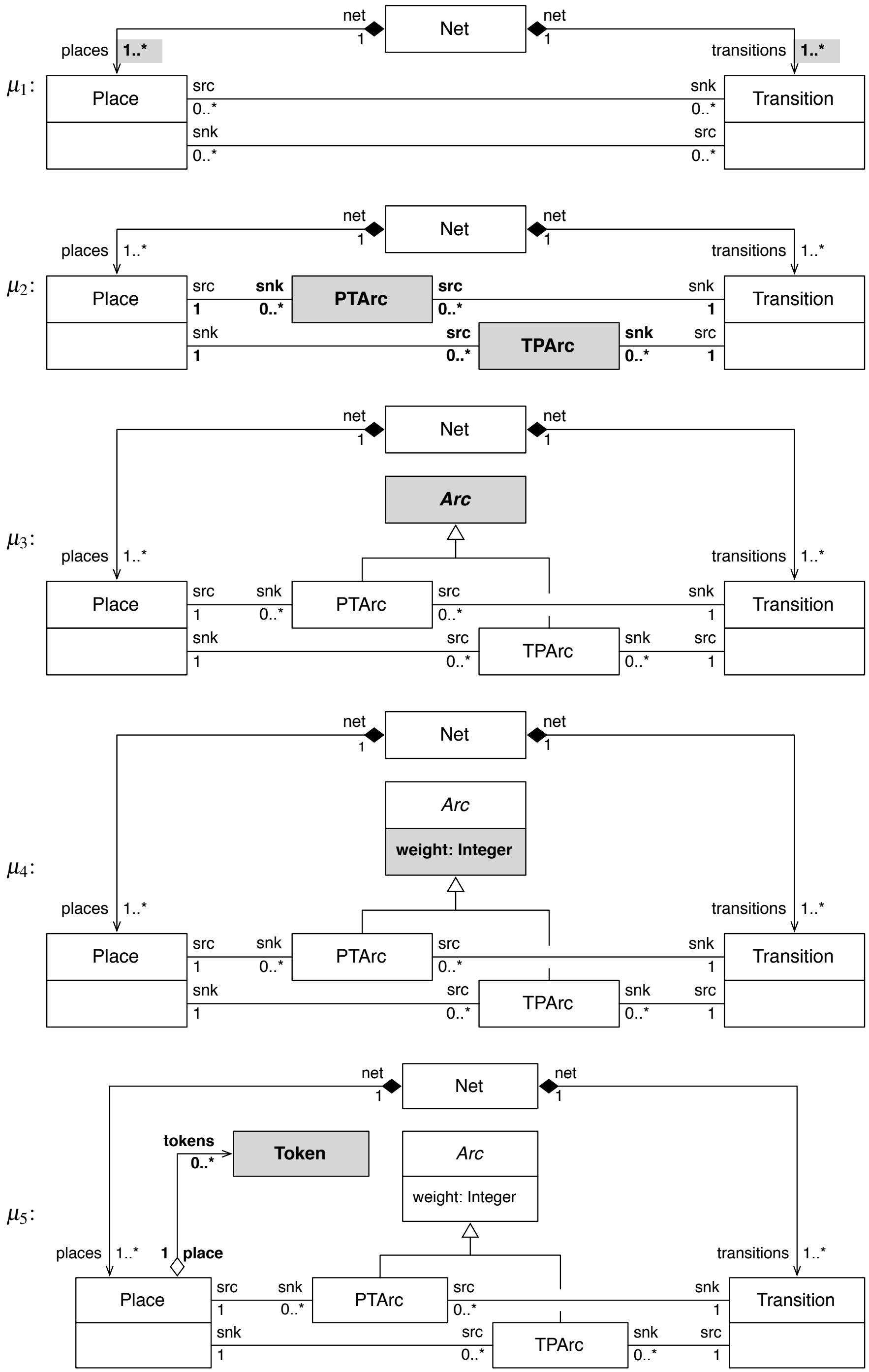


Fig. 2.2: Steps in the evolution of a metamodel for Petri net models.

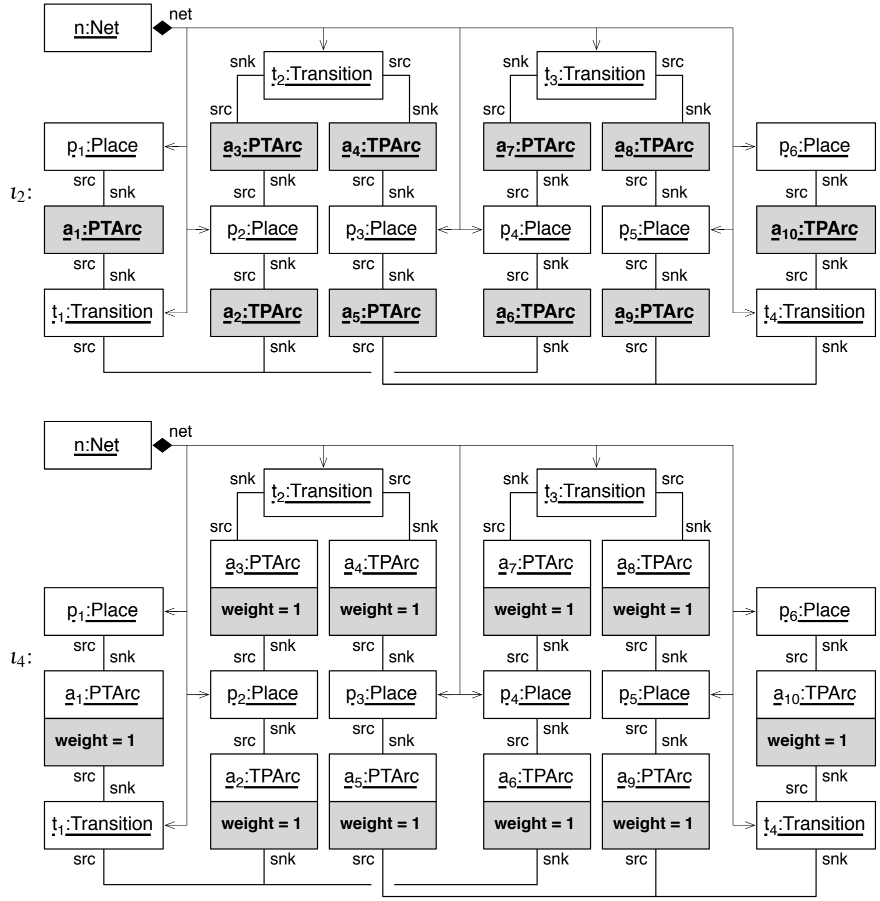


Fig. 2.3: Steps in the migration of a Petri net model.

models lack weights for arcs. For migration, we can add a default weight of 1 to all `Arc` objects. Figure 2.3 shows a model ι_4, the result of migrating ι_2 accordingly. In the last step, models lack tokens for places. For migration, we associate zero tokens to each place.

2.2 Analysis: Classification of Coupled Evolution

Coupled evolution of metamodels and models can be classified according to metamodel aspect, granularity, language preservation, model preservation, reversibility, and automatability. We stick to a simplified, unified version of the terminology from [394, 399, 904].

2.2.1 Metamodel Aspect

A metamodel can not only specify the structure of models, but can also define constraints on models or an API to access models. Furthermore, it can provide documentation of metamodel elements. We can classify evolution according to its effect on these different aspects, distinguishing *structural* evolution, *constraints* evolution, *API* evolution, and *documentation* evolution. For example, the evolution of the Petri net metamodel in Example 2.2 is purely structural.

2.2.2 Granularity

Metamodel evolution can be of different granularity. *Composite* evolution can be decomposed into smaller evolution steps, while *primitive* evolution is atomic and can not be decomposed. We further distinguish two kinds of primitive evolution. *Structural primitive* evolution modifies the structure of a metamodel, i. e. creates or deletes a metamodel element. *Non-structural primitive* evolution modifies an existing metamodel element, i. e. changes a feature of a metamodel element.

Example 2.4 (Granularity of Petri net metamodel evolution). In the evolution of the Petri net metamodel (ref. Example 2.2), all steps except the fourth are composite, since they can be decomposed into smaller evolution steps. For example, the first step can be decomposed into two non-structural primitive evolution steps, each changing the cardinality of a relation. The fourth step adds a single new attribute to the metamodel. Thus, it can be classified as structural primitive.

2.2.3 Language Preservation

A metamodel is an intensional definition of a language. Its extension is a set of conforming models. When a metamodel evolves, this has an impact on its extension and thus on the expressiveness of the language it defines. We distinguish different classes of coupled evolution according to this impact [904]: Coupled evolution is a *refactoring* if its migration is a bijective mapping between extensions of the original and the evolved metamodel. Coupled evolution is a *construction* if its migration is an injective mapping from the extension of the original metamodel to the extension of the evolved metamodel. Coupled evolution is a *destruction* if its migration is a potentially partial, surjective mapping from the extension of the original metamodel to the extension of the evolved metamodel.

Notably, this classification is typically only applicable to smaller evolution steps. Composite evolution is often neither a pure refactoring nor a construction nor a destruction, but a mixture of refactoring, construction, and destruction steps.

Example 2.5 (Language preservation in Petri net metamodel evolution). We revisit the coupled evolution of the Petri net metamodel (ref. Example 2.2) and Petri net models (ref. Example 2.3). The first evolution step is a destruction, since the migration is a partial, surjective mapping, keeping only Petri net models with places and transitions. The second evolution step is a refactoring, since the migration is a bijective mapping between models with direct links between places and transitions, and models with `PTArc` or `TPArc` objects in-between. The third evolution step is also a refactoring, since no migration is needed (identity is a bijective mapping). The last two evolution steps are constructions, since both the introduction of default weights and the introduction of empty token sets are injective mappings. The overall evolution is neither a refactoring nor a construction nor a destruction, since the overall migration is partial but not surjective.

2.2.4 Model Preservation

Model preservation properties indicate when migration is needed. Coupled evolution is *model-preserving*[1] if all models conforming to an original metamodel also conform to the evolved metamodel. Thus, model-preserving evolution does not require migration. Coupled evolution is *model-migrating*[2] if models conforming to an original metamodel might need to be migrated in order to conform to the evolved metamodel. A migration is *safe* if the migration preserves distinguishability, that is different models (conforming to the original metamodel) are migrated to different models (conforming to the evolved metamodel). In contrast, an *unsafe* migration might either migrate only some models or yield the same model when migrating two different models.

In contrast to the classification according to language preservation, this classification is complete with unsafe migration as its default. Both classifications are related. Refactorings and constructions are by definition either model-preserving or safely model-migrating, since their migration is either a bijective or injective mapping. Destructions are unsafely model-migrating.

Example 2.6 (Model preservation in Petri net metamodel evolution). Again, we revisit the coupled evolution of the Petri net metamodel (see Example 2.2) and Petri net models (see Example 2.3). The first evolution step is unsafely model-migrating, since Petri net models without places or transitions are not migrated. The third evolution step is model-preserving, since no migration is required. The second, fourth, and fifth evolution steps are safely model-migrating, since the introduction of additional objects, weights, and empty token sets keeps models distinguishable. The overall evolution is unsafely model-migrating, since some models are not migrated.

[1] In [89, 352], such evolution is called a *non-breaking change*.

[2] In [89, 352], such evolution is called a *breaking, resolvable change*.

2.2.5 Reversibility

Reversibility properties indicate whether a coupled evolution can be undone. Coupled evolution is *reversible* if and only if the sequential composition with another coupled evolution step is a refactoring. Coupled evolution is *safely reversible* if and only if either itself or a sequential composition with another coupled evolution step is model-preserving. All other evolutions are *irreversible*.

The definition points out relations to the classification according to language and model preservation. Distinguishability ensures reversibility. Thus, any safely model-migrating evolution, including refactorings and constructions, is also safely reversible. Destructions are irreversible.

Example 2.7 (Reversibility in Petri net metamodel evolution). Once more we revisit the coupled evolution of the Petri net metamodel (see Example 2.2) and Petri net models (see Example 2.3). The first evolution step is irreversible, since Petri net models without places or transitions are not migrated. All other steps are safely reversible: The second evolution step can be undone by reverting μ_2 to μ_1 and migrating links with `PTArc` or `TPArc` objects to direct links. The third evolution step is model-preserving itself. The fourth evolution step can be undone by reverting μ_4 to μ_3 and removing weights in the corresponding migration. Similarly, the fifth step can be undone. The overall evolution is irreversible, since Petri net models without places or transitions are not migrated.

For the last two steps, safe reversibility is not so obvious, since the inverse migration is unsafe. For example, the fourth step requires the addition of weights, as we did in the migration from ι_2 to ι_4. We can obtain ι_2 back from ι_4 by removing the weights. Since added information is removed again, the composition is model-preserving. But this is not the case the other way around. When μ_4 would evolve into μ_3, ι_4 could be migrated to ι_2. However, any other model with the same places, transitions and arcs, but different weights, would also be migrated to ι_2. This migration would be unsafe, since we cannot restore the original weights.

2.2.6 Automatability

When a metamodel evolves, automatability of the corresponding model migration is crucial. In the worst case, the evolution requires a *model-specific* migration, making automation impossible. Instead, software engineers have to revisit their models and migrate each of them individually.

Example 2.8 (Model-specific Petri net model migration). We reconsider the last migration step from Example 2.3, which migrated models conformant to μ_4 by adding an empty set of tokens to each place. However, imagine a Petri net model which models resource dependencies in a software system. Models conformant to μ_4 can only model dependencies, but not the actual resources. With μ_5, resources can be modeled as tokens. After migration, models perfectly conform to μ_5, stating the absence of resources. This is typically inaccurate with respect to the system under study. Instead, software engineers should migrate each model individually, modeling the resources of systems under study accordingly.

In contrast to model-specific migration, *language-specific* migration depends only on the evolving metamodel, but not on particular models. This enables partial automation. Language engineers can develop a migration transformation and ship this with the evolved metamodel. Language users can then migrate their existing models by applying the migration transformation to them.

Example 2.9 (Language-specific Petri net model migration). In the migration of Petri net models as discussed in Example 2.3, the creation of objects in links between places and transitions, the introduction of a default weight for arcs, and the introduction of an empty token set for places is model-independent. These migration steps can be specified as a transformation from models conformant to μ_0 into models conformant to μ_5. The transformation specification will be *language-specific*, since it relies on the metamodels μ_0 and μ_5. We will discuss such a manually specified migration in Example 2.12 in Section 2.4.2.

Language-independent migrations free language engineers from specifying migrations manually. Instead, they specify only the metamodel evolution, either explicitly or implicitly. The corresponding migration specification is automatically derived from the metamodel evolution.

Example 2.10 (Language-independent Petri net model migration). In the previous example, we discussed a language-specific specification of the migration of Petri net models as discussed in Example 2.3. However, each step in the evolution of the Petri net metamodel from Example 2.2 is an instance of a recurring evolution pattern: In the first step, the cardinality of an association is restricted. In the second step, associations are turned into association classes. In the third step, a common super class is extracted. In the last two steps, new properties with default values are added. Each of these patterns is coupled with a generic, corresponding migration pattern. These migration patterns are *language-independent*. The concrete evolution of the Petri net metamodel instantiates the evolution patterns, and by this the migration patterns to a concrete migration for Petri net models. We will discuss such a coupling of evolution and migration steps in Example 2.14 in Section 2.4.4.

2.3 Empirical Results: Metamodel Evolution in Practice

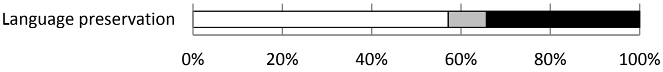
Software language evolution in general and metamodel evolution in particular are not only interesting theoretical problems, but highly relevant for language developers as well as for language users. To illustrate this relevance, we summarise three different case studies in this section. These case studies investigated metamodel evolution in different real-world modeling languages. The first study addresses the evolution of the general-purpose modeling language UML, which is standardized by the OMG. The second study covers the evolution of two proprietary domain-specific modeling languages used in the automotive industry. The third study examines the evolution of modeling languages in GMF, an open-source project for the generation of graphical editors in Eclipse.

2.3.1 *Evolution of the Unified Modeling Language*

The Unified Modeling Language (UML) is a general-purpose modeling language managed by the OMG. It is widely used to model software-intensive systems. UML provides a metamodel which captures important concepts in object-oriented software modeling. It also provides different diagram types to visualize partial views on UML models.

Since the adoption of UML 1.1 by the OMG in 1997, UML has evolved significantly. The minor revisions UML 1.3 and 1.4 fixed shortcomings and bugs. Minor revision UML 1.5 integrated action semantics into UML 1.4. In the major revision UML 2.0 in 2005, the specification was split into a core infrastructure [651] capturing the architectural foundations and a superstructure [652] specifying user-level constructs. Some changes made in this revision were not model-preserving. thus requiring model migration.

Street et al. [799] analyzed the metamodel evolution of UML from version 1.4 to 2.0. They focused particularly on changes at the user-level, examined them and analyzed their impact on migrating legacy UML 1.4 models to UML 2.0. They classified changes into constructions, refactorings, and destructions. As can be seen in Figure 2.4, most of the changes were constructions which allow to improve existing models. Required migrations for refactorings and destructions could be mostly automated.



	Language preservation
□ Construction	20 (57.1%)
▨ Destruction	3 (8.6%)
■ Refactoring	12 (34.3%)

Fig. 2.4: Classification of UML metamodel changes with respect to language preservation

Particularly interesting is the evolution of activity diagrams. While their semantics was based on state machines in UML 1.4, it is based on Petri nets since UML 2.0. A significant number of changes were needed to achieve this switch. Additionally, several new features were added to enhance control flow modeling. In [729], Rose et al. provide a specification for the automatic migration of UML 1.4 activity diagrams to UML 2.0. We discuss the corresponding coupled evolution approach in Section 2.4.2 and its realization in the Epsilon Flock tool in Section 2.5.2.

2.3.2 Evolution of Automotive Metamodels

To better understand the nature of coupled evolutions of metamodels and models in practice, we presented a study on the evolution of two industrial metamodels from the automotive domain in earlier work [394]:

- Flexible User Interface Development (FLUID) for the specification of automotive user interfaces, and
- Test Automation Framework - Generator (TAF-Gen) for the generation of test cases for these user interfaces.

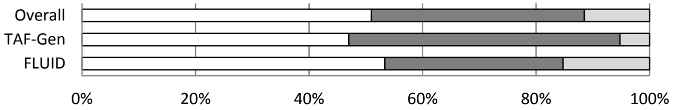
During the evolution of both metamodels, the impact on existing models was not taken into account.

The study investigated whether reuse of migration knowledge can significantly reduce migration effort. Its main goal was to determine substantiated requirements for tool support that is adequate for model migration in practice. The study analyzed the evolution of both metamodels based on revisions in a version control system[3]. First, all revisions of the metamodels were extracted from the version control system. Next, subsequent metamodel revisions were compared using a model differencing tool resulting in a set of changes for each evolution step. For each metamodel change, a corresponding model migration was defined. Finally, each metamodel change and its corresponding model migration was classified according to the classification presented in Section 2.2.6.

The result of the study for both metamodel evolutions is shown in Figure 2.5 as a bar chart. The figure shows the fraction and the accumulated numbers of metamodel changes that fall into each class. Half of the metamodel changes require migration, but no change needs model-specific migration. Of the model-migrating changes, over 75% can be covered by a language-independent migration. Moreover, the language-specific migrations in the study always resulted from a group of metamodel changes which could not be treated separately in terms of migration. As a consequence, these coupled evolutions could not be composed of reusable migration migration patterns.

The results show that in practice model migration is required for a significant number of metamodel changes. They also indicate that the migration of all models usually can be completely automated by a model transformation. Moreover, the study showed that even more effort can be saved by reusing recurring migration patterns. However, it also occurs in practice that some migrations are too complex to be captured by such patterns.

[3] More information about approaches to mine software repositories can be found in Chapter 5.



	FLUID	TAF-Gen	Overall
□ Model-preserving	119 (53.4%)	63 (47.0%)	182 (51.0%)
▨ Language-independent migration	70 (31.4%)	64 (47.8%)	134 (37.5%)
□ Language-specific migration	34 (15.2%)	7 (5.2%)	41 (11.5%)
■ Model-specific migration	0 (0.0%)	0 (0.0%)	0 (0.0%)

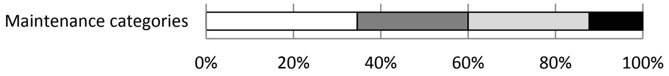
Fig. 2.5: Classification of FLUID and TAF-Gen metamodel changes with respect to migration automatability

2.3.3 Evolution of the Graphical Modeling Framework

In [398], we performed a follow-up study to investigate couplings of metamodel evolution and model migration in more detail. We studied the two year evolution of modeling languages provided by the Graphical Modeling Framework (GMF), an open source project for developing graphical editors in Eclipse. These modeling languages allow to define the graphical syntax of a language in a platform-independent model, from which GMF generates a platform-specific model by executing a model-to-model transformation. GMF then transforms the platform-specific model into code for a graphical editor by means of a model-to-text transformation.

The goal of this study was to identify the reasons for metamodel evolution, to understand the impact of metamodel evolution on related language artifacts, and to detect patterns of metamodel evolution steps and corresponding model migrations. Like in the previous study, all revisions of the languages' metamodels were extracted from a version control system. Additionally, corresponding changes of related artifacts were taken into account as well, in order to analyze the impact of metamodel evolution on other artifacts. Each commit was classified manually according to standard maintenance categories based on the commit message and related change requests [424]: perfective, adaptive, preventive, and corrective. Next, metamodel evolutions within a commit were split into evolution steps, and their migration was classified according to the classification presented in Section 2.2. Finally, the correctness of the migration was validated by comparing it to the handwritten migrator provided by GMF.

The study yielded several interesting insights into metamodel evolution in practice. First, it shows that metamodel evolution is similar to the evolution of object-oriented code. The classification according to standard maintenance categories as illustrated in Figure 2.6 shows that these categories apply to metamodel evolution as well. The classification according to metamodel aspects as shown in Figure 2.7 indicates user requests and technological change as the main reasons for metamodel evolution. Furthermore, the splitting of metamodel evolution steps into small changes revealed patterns that turned out to be similar to object-oriented refactorings [301].



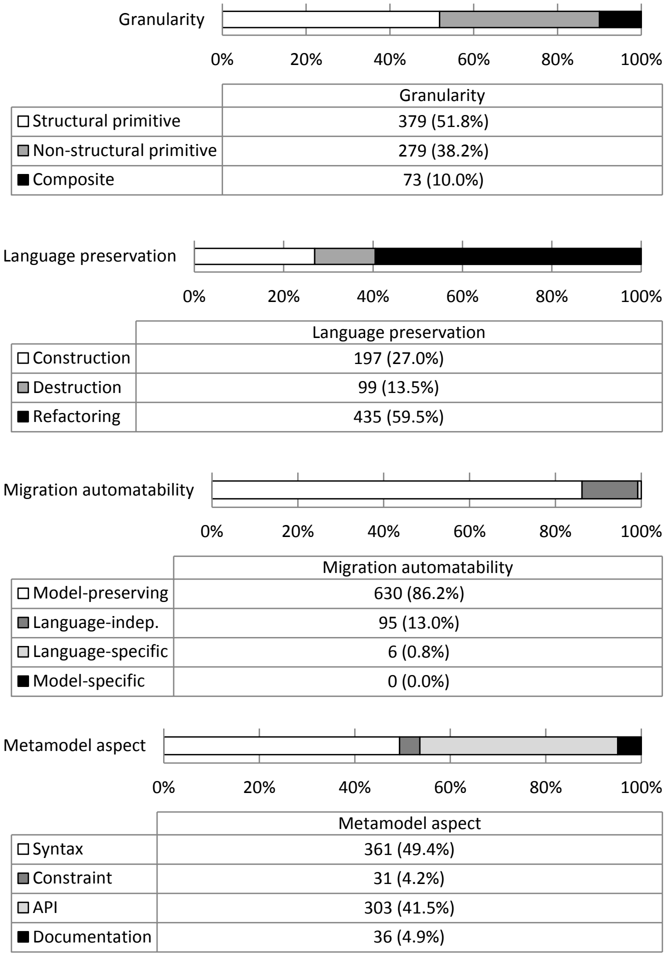
	Maintenance categories
□ Perfective	45 (34.6%)
▩ Adaptive	33 (25.4%)
▨ Preventive	36 (27.7%)
■ Corrective	16 (12.3%)

Fig. 2.6: Classification of GMF metamodel commits along standard maintenance categories [424].

Second, a large majority of these changes are either primitive structural changes, such as the addition of an attribute, or primitive non-structural changes, such as a renaming. Also, most of these changes are either constructions or refactorings, and all corresponding migrations are fully automatable. This is illustrated by the classifications in Figure 2.7. Third, the same figure shows, that metamodel evolution applies to other aspects than abstract syntax definition, such as static constraints, APIs, and documentation, which are also provided by a language's metamodel. Finally, Figure 2.8 shows that not only models but also other artifacts such as model-to-model and model-to-text transformations need to be migrated in order to stay conforming with evolving metamodels.

2.3.4 Discussion of the Empirical Results

The empirical results from the different studies have several commonalities. First, the number of constructions is quite high and the number of destructions is rather low. This holds for all case studies, even for the GMF case study, since there many of the refactorings are metamodel changes that do not change the syntax of the modeling language. This means that there is a tendency to add new constructs to a metamodel and not to remove existing constructs from it. Second, a metamodel evolution can be divided into many small changes where the changes can be regarded separately from each other in terms of migration. These metamodel changes are only getting bigger in case of language-specific migrations where several changes have to be treated together. However, all the case studies indicate that such language-specific migrations are rather rare in practice, but nevertheless occurring. Third, the migrations that occur in practice can be automated to a high degree. In all three case studies, the migration could be implemented as a model transformation to automatically migrate models. Moreover, the model transformation can be mostly built from



	Granularity
□ Structural primitive	379 (51.8%)
▨ Non-structural primitive	279 (38.2%)
■ Composite	73 (10.0%)

	Language preservation
□ Construction	197 (27.0%)
▨ Destruction	99 (13.5%)
■ Refactoring	435 (59.5%)

	Migration automatability
□ Model-preserving	630 (86.2%)
▨ Language-indep.	95 (13.0%)
░ Language-specific	6 (0.8%)
■ Model-specific	0 (0.0%)

	Metamodel aspect
□ Syntax	361 (49.4%)
▨ Constraint	31 (4.2%)
░ API	303 (41.5%)
■ Documentation	36 (4.9%)

Fig. 2.7: Classification of changes in GMF metamodels with respect to granularity, language preservation, migration automatibility, and metamodel aspect.

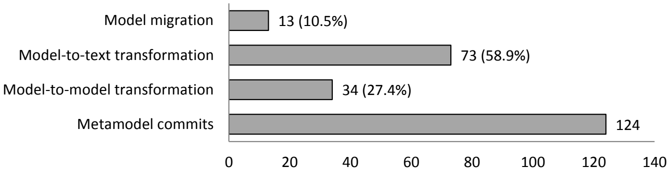


Fig. 2.8: Impact of GMF metamodel commits on related artifacts.

recurring migration patterns, providing further automation. These empirical results are the basis for many of the approaches that have been developed to tackle the problem of model migration.

2.4 State-of-the-Art: Approaches and their Classification

Over the last decade, coupled evolution of metamodels and models has attracted much attention from the scientific community. According to Rose et al. [730], we can distinguish three categories of approaches: manual specification approaches, metamodel matching approaches, and operator-based approaches. Table 2.1 lists the different approaches and groups them according to these three categories. The table also marks the characteristic features for each category with asterisks. In this section, we will first present the table's underlying classification scheme, before we discuss each category and its approaches.

Table 2.1: Classification of different approaches to the coupled evolution of metamodels and models (The asterisks denote the characteristic features of the approach categories).

	feature	evolution spec.						migration specification										eval.		
		style		source				coupling			language			target		exec.				
						detec.														
approach	notation	imperative	declarative	user-defined	recorded	simple	complex	fixed	overwritable	extendable	custom	TL	GPL	in-place	out-of-place	online	offline	preliminary	regular	comparison
manual specification									*		*									
Sprinkle	GME		•	•					•		•				•		•	•		
MCL	GME		•	•					•		•				•		•			
Flock	Ecore								•		•				•		•			•
metamodel matching			*			*	*													
Gruschko	Ecore		•			•			•			•^{1}			•		•			
Geest	MS DSL		•			•			•				•^{2}		•		•		•	
Cicchetti	Ecore		•				•	•				•^{3}			•		•			
AML	Ecore		•				•			•		•^{3}			•		•		•	
operator-based		*																		
Hößler	MOF	•		•				•							•		•			
Wachsmuth	MOF	•		•				•				•^{4}			•		•			
COPE	Ecore	•			•					•			•^{5}	•			•		•	

^{1} ETL ^{2} C# ^{3} ATL ^{4} QVT ^{5} Groovy

2.4.1 Classification Scheme

To be able to compare existing approaches, a scheme is required according to which all approaches can be classified and compared. Table 2.1 includes a classification scheme for metamodel-model co-evolution approaches. However, classification schemes are best represented as feature models, as they allow to define the features of the different approaches as well as how they can be composed. Figure 2.9 shows the corresponding feature model using the FODA notation [452]. An introduction to the FODA notation can be found in Section 9.2. We now explain the various features of this model.

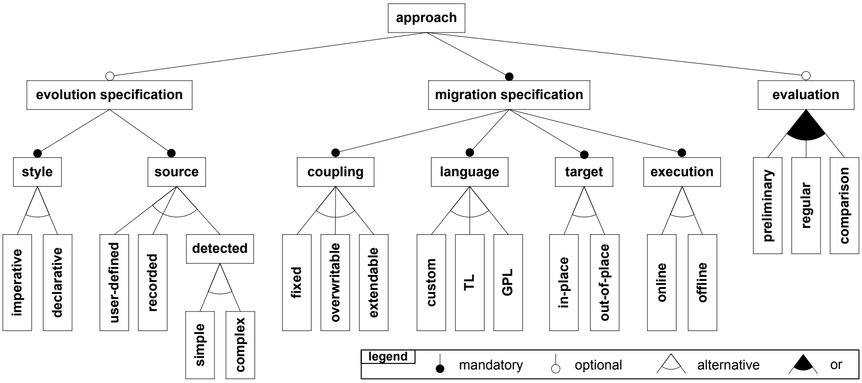


Fig. 2.9: Feature model for the classification of approaches

2.4.1.1 Metamodel Evolution Specification

The evolution of a metamodel is implicitly specified by the original and the evolved version of the metamodel. However, many approaches are based on explicit *evolution specification*s. There are two *style*s of such specifications: *Imperative* specifications describe the evolution by a sequence of applications of change operators. In contrast, *declarative* specifications model the evolution by a set of differences between the original and evolved version of the metamodel.

Explicit evolution specifications can have different *source*s. One prominent source is the automated *detection* of the evolution based on the original and evolved version of the metamodel. Two kinds of detections can be distinguished: First, detections which are only able to detect *simple* changes like additions and deletions. For some approaches, this includes the detection of moves as well. Second, detections which can also detect more *complex* changes, for example folding and unfolding of abstractions. As an alternative to detection, the evolution can be *recorded* while the user edits the metamodel, or *user-defined* where the user specifies the evolution manually.

2.4.1.2 Model Migration Specification

In contrast to evolution, the model migration always needs to be specified explicitly. The dependency of migration on evolution is reflected by coupling evolution specifications with *migration specification*s. There are three kinds of *coupling*s: With a *fixed* coupling, the migration is completely defined by the evolution. Only the developer of a coupled evolution tool can add new couplings. With an *overwritable* coupling, the user can overwrite migrations for single evolution steps with custom migration specifications. With an *extendable* coupling, the user can define completely new, reusable couplings between evolution and migration specifications.

Approaches with overwritable coupling need to provide a *language* to specify the custom migration. Such a language might be *custom* defined as a domain-specific migration language. Alternatively, an existing model *transformation language* (TL) can be reused. Another way is to add migration support to a *general-purpose programming language* (GPL) in form of an API or an internal domain-specific language (DSL).

Depending how the *target* model is derived from the original model, migration might be performed either *in-place* or *out-of-place*. In the first case, the target of the migration is the original model itself which is modified during migration. In the second case, the target is a new migrated model which is created during migration. The original model is preserved.

Furthermore, the *execution* of the migration might be *offline* where applications cannot use some of the models during the migration, or *online* where applications can still use all models and where the access of a model by an application triggers its migration lazily.

2.4.1.3 Approach Evaluation

Evaluation is crucial for the validation of approaches and thus an important quality of an approach. Approaches might provide no evaluation at all. They might provide only evaluation of *preliminary* nature, for example by toy examples. Often, the need for further evaluation is stated explicitly in corresponding publications. Other approaches include *regular* evaluation on industrial or open-source systems of medium to large scale. Some authors provide a *comparison* of their approach with existing approaches.

2.4.2 Manual Specification Approaches

Manual specification approaches provide custom model transformation languages with specific model migration constructs which reduce the effort for building a migration specification.

Model transformation is a well-established research area in model-driven software engineering. In general, we can distinguish *endogenous transformations* between models expressed in the same language and *exogenous transformations* between models expressed using different languages [597]. Neither kind of transformation is well-suited for model migration. Endogenous transformations require the same metamodel for source and target models, which is not the case in the presence of metamodel evolution. Exogenous transformations can handle different source and target metamodels, but require complete mapping specifications, which leads to a lot of identity rules. Manual specification approaches overcome these difficulties by providing constructs particularly intended for model migration.

Example 2.11 (Migrating transformation specification for Petri Net metamodel evolution). Let us illustrate this using the evolution of Petri net metamodels from Example 2.2. The migration from μ_1 to μ_2 can be specified in the exogenous, out-of-place model transformation language ATL [448] as follows:

```
module migrate_mu1_to_mu2;
create OUT : mu2 from IN : mu1;

rule Nets {
  from o : mu1!Net
  to   m : mu2!Net (
    places <- o.places, transitions <- o.transitions
  )
}

rule Places {
  from o : mu1!Place
  to   m : mu2!Place
}

rule Transitions {
  from o : mu1!Transition
  to   m : mu2!Transition (
    src <- o.src->collect(p | thisModule.PTArcs(p,o)),
    snk <- o.snk->collect(p | thisModule.TPArcs(o,p))
  )
}

lazy rule PTArcs {
  from place : mu1!Place, destination : mu1!Transition
  to   arc : mu2!PTArc (
    src <- place, snk <- destination, net <- destination.net
  )
}
```

```
lazy rule TPArcs {
  from transition : mu1!Transition, destination : mu1!Place
  to   arc : mu2!TPArc (
    src <- transition, snk <- destination, net <- transition.net
  )
}
```

Note that the first two rules constitute just an identity transformation without migration and thus could be easily spared. The last three rules replace links from `Places` to `Transitions` by `PTArcs` and links from `Transitions` to `Places` by `TPArcs`.

Sprinkle et al. [792, 793] introduce a visual language to declaratively specify the differences between two versions of a GME-based metamodel. The Model Change Language (MCL) [65, 633] is another visual migration language targeting GME. With both languages, the user not only specifies the metamodel differences, but defines a model migration based on them. This overwrites the default copying behavior. The migration is performed out-of-place and offline. MCL permits a number of idioms that—according to the authors' experience—cover most common migration cases. Migration algorithms not covered by MCL can be specified imperatively using a C++ API. Sprinkle's approach is evaluated by an experience report that demonstrates the modeling of a complex migration taken from the application of the Embedded Systems Modeling Language (ESML) [148] in the avionics industry.

Flock is a textual migration language for EMF-based models [729]. Here, only the model migration is specified. Differences between metamodel versions are not made explicit. Instead, Flock automatically copies only those model elements which conform to the evolved metamodel. The user then iteratively redefines the migration specification to migrate non-conforming elements. Using the Petri net example from [904], Flock has been compared to migration specifications in model transformation languages ATL and Ecore2Ecore as well as to an operator-based approach with COPE which is introduced later.

Example 2.12 (Manual migration specification for Petri Net metamodel evolution). Let us reconsider the migration of Petri net models as discussed in Example 2.3. Migration from μ_1 to μ_2 can be specified in Flock as follows:

```
migrate Transition {
  for (source in original.src) {
    var arc = new Migrated!PTArc;
    arc.src = source.equivalent();
    arc.snk = migrated;
  }

  for (sink in original.snk) {
    var arc = new Migrated!TPArc;
    arc.src = migrated;
    arc.snk = sink.equivalent();
    arc.net = migrated.net;
  }
}
```

The migration considers instances of `Transition`. For each `src`, it creates a `PTArc` and connects it to the `src` and to the `migrated` transition. Similarly, it creates a `TPArc` for each `snk` and connects it to the `snk` and to the migrated transition.

2.4.3 Metamodel Matching Approaches

Metamodel matching approaches automatically detect the differences between two metamodel versions. These are stored in a declarative difference model from which a migration specification is automatically generated.

Gruschko et al. [89, 352] support the automatic detection of simple changes in Ecore metamodels. They propose automatic migration steps for resolvable changes and envision to support the user in overwriting the migration specification for unresolvable changes. The approach is only prototypically implemented and has not been evaluated.

Geest et al. [318] apply a similar approach in the context of Microsoft DSL Tools. The difference model is obtained by a possibly human-aided comparison of the metamodel versions. Only simple changes can be detected and the generated migration specification can be overwritten. The approach has been evaluated on evolving metamodels from the Web Service Software Factory (WSSF)[4].

Cichetti et al. [188] also detect complex metamodel changes. Here, the difference model consists of simple changes which are interpreted in terms of complex changes. The migration specification consists of a set of model transformations to be executed consecutively. Since this is prevented by interdependent changes, they characterize dependencies between complex changes [189].

The Atlas Matching Language (AML) allows the user to parameterize the detection of complex changes [311]. Therefore, the user combines existing or user-defined heuristics to a thus extendable matching algorithm. From a difference model obtained by such an algorithm, an ATL [448] transformation specifying the migration is automatically generated. The approach was evaluated on the Petri net example from [904], and on the Java metamodel from NetBeans.

Example 2.13 (Metamodel matching in Petri Net metamodel evolution). We revisit the evolution of Petri net metamodels from Example 2.2. Metamodel matching for μ_0 and μ_1 is performed as follows:

1. Each class is matched with its counterpart in the evolved metamodel.
2. No changes are detected on classes.
3. Relations are matched with their counterpart in the evolved metamodel.
4. Changes in the lower bounds of `Net.places` and `Net.transitions` are detected.
5. `ChangedReference` entries are created and added to the difference model.

[4] WSSF community: http://codeplex.com/servicefactory

The changes in the difference model are classified as unresolvable, since strengthened lower bounds require manual migration. Thus, no migration is generated.

Metamodel matching for μ_1 and μ_2 yields the following changes in the difference model:

1. `AddedClass` entries for `PTArc` and `TPArc`.
2. `ChangedReference` entries for `Place.src`, `Place.snk`, `Transition.src`, and `Transition.snk`.
3. `AddedReference` entries for `PTArc.src`, `PTArc.snk`, `TPArc.src`, and `TPArc.snk`.

These changes are classified as breaking, resolvable changes, which corresponds to model-preserving according to the classification in Section 2.2. This classification depends on detecting a pattern of an added class, a changed reference which now points to the added class, and an added reference between added class and original target. This pattern can be detected twice, for `PTArc` and `TPArc`. It corresponds to a conversion of an association into a class with two associations, which is a well-known object-oriented refactoring [301]. The generated model migration in the form of an ATL transformation is similar to the one shown in Example 2.11. It transforms links of the original association into an object of the added class and corresponding links of the changed and added reference.

2.4.4 Operator-based Approaches

Operator-based approaches specify coupled evolution as a sequence of *coupled operators*, which encapsulate common metamodel evolution steps and their corresponding model migration.

Hößler et al. [406] formalize a fixed suite of reusable coupled operators. The completely theoretical approach is based on a generic instance model supporting versioning and was neither implemented nor evaluated.

Wachsmuth [904] presents an operator suite for the MOF metamodeling formalism. Based on ideas from grammar evolution [490], operators are classified according to language and model preservation properties. For migration, the evolution specification is translated into a QVT Relations model transformation.

COPE [395] adds tool support for the evolution of Ecore-based metamodels to EMF. It provides an extensive suite of coupled operators [399], which can be extended with user-defined reusable operators. In addition to user-defined evolution specifications, COPE supports recording of operator applications. The operators are specified in a DSL embedded into a general-purpose language. COPE is the only model migration approach performing in-place migration, since in-place transformation is not very common for exogenous transformations. Its evaluation by reverse engineering the evolution of the Palladio Component Model [395] and the evolution of GMF (see Section 2.3.3) proved the applicability of operator-based approaches in model-driven software engineering.

Example 2.14 (Applying coupled operators to model Petri Net metamodel evolution). The coupled evolution of the Petri net metamodel (ref. Example 2.2) and Petri net models (ref. Example 2.3) can be modelled as a sequence of applications of coupled operators from [399]. First, we specialize two references in μ_0:

```
specialize composite reference Net.places: Place {1..*}
specialize composite reference Net.transitions: Transition {1..*}
```

This yields μ_1, where we replace two associations with classes:

```
association Place.snk: Transition to class PTArc
association Transition.snk: Place to class TPArc
```

This results in μ_2, where we extract a common super class:

```
extract abstract class Arc from PTArc, TPArc
```

This gives us μ_3, where we introduce weights for arcs:

```
create attribute Arc.weight: int {1} = 1
```

This leaves us with μ_4, where we add tokens to places:

```
create class Token
create composite reference Place.tokens: Token {0..*} = []
create opposite reference Place.tokens Token.place {1}
```

This finally results in μ_5.

2.4.5 Discussion of State-of-the-Art

Out of the 10 compared approaches, 7 target either MOF, which is a well-recognized standard in model-driven software engineering, or its implementations. Approaches targeting the same modeling framework can be easily compared with each other, leading to evaluations by comparison. 6 out of 10 approaches use declarative evolution specifications, since they either define new or use existing declarative model transformation languages. There is only one recording approach which is probably more complex to implement, i. e. most of the compared approaches focus on specifying the model migration after the metamodel evolution took place. To be able to specify language-specific model migrations, 7 out of 10 approaches allow at least to overwrite the migration specification, but only two of them can be extended by reusable couplings. 7 out of the 10 compared approaches reuse or refine existing model transformation languages: Manual specification approaches tailor model transformation languages to migration (3), metamodel matching approaches synthesize transformation specifications (3), and Wachsmuth's operator-based approach specifies operators in QVT. Only one approach performs in-place migration, since exogenous transformation languages that are required for metamodel evolution do not support in-place migration. No approach can perform migration online—probably since models are design-time artifacts, thus being stored and not in use most of the time. Finally, half of the 10 compared approaches are evaluated, 4 out of these 5 at least regularly on industrial or open-source systems.

2.5 Tool support: Available Tools and their Comparison

Tool support is crucial to bring scientific approaches into practice. Tools also allow to compare different approaches with each other using common case studies. Therefore, a number of tools have been built for the approaches discussed in Section 2.4. Unfortunately, only two tools can still be obtained from the internet at the time of writing this chapter: Edapt which is the successor of COPE, and Epsilon Flock. In this section, we present these two tools and summarize results from two case studies, in which formerly available tools were compared with general-purpose model transformation tools as well as with each other.

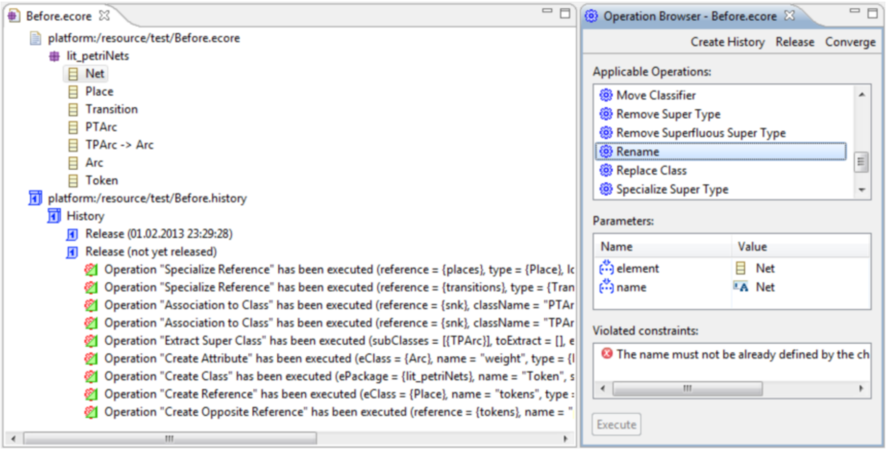
2.5.1 COPE / Edapt

Edapt[5] is the official Eclipse tool for migrating EMF models in response to the adaptation of their metamodel. Like its predecessor COPE, it records the metamodel evolution as a sequence of *coupled operators* in a history model [393]. Each coupled operator performs an in-place transformation of both the metamodel and the model. Edapt provides two kinds of coupled operators—reusable and custom coupled operators [393].

Reusable coupled operators enable reuse of migration specifications across metamodels by making transformations for metamodel evolution and model migration independent of the specific metamodel through parameters. Currently, Edapt comes with a library of over 60 available reusable coupled operators, which proved to be useful in a number of real-life case studies [399]. *Custom coupled operators* allow to attach a custom migration to a recorded metamodel adaptation. The custom migrations are implemented in Java based on the API provided by Edapt to navigate and modify models. In the Example 2.14, all the history can be covered by reusable coupled operators. Figure 2.10 shows the corresponding history model in Edapt's user interface.

Edapt's user interface—depicted in Figure 2.10—is directly integrated into the existing EMF *metamodel editor*. The user interface provides access to the *history model* in which Edapt records the sequence of coupled operators. The user can adapt the metamodel by applying reusable coupled operators through the *operation browser*. When a reusable coupled operator is executed, its application is recorded in the history model. A custom coupled operator is performed by first modifying the metamodel in the editor, and then attaching a *custom migration* to the recorded metamodel changes. Figure 2.10 shows the reusable coupled operators that need to be executed to evolve the Petri net metamodel. For instance, the operator `Association to Class` is used to replace the links between `Places` and `Transitions` by instances of `PTArc` and `TPArc`, respectively.

[5] http://www.eclipse.org/edapt



Release (01.02.2013 23:29:28)
Release (not yet released)
Operation "Specialize Reference" has been executed (reference = {places}, type = {Place}, lc
Operation "Specialize Reference" has been executed (reference = {transitions}, type = {Tran
Operation "Association to Class" has been executed (reference = {snk}, className = "PTAr
Operation "Association to Class" has been executed (reference = {snk}, className = "TPAr
Operation "Extract Super Class" has been executed (subClasses = [{TPArc}], toExtract = [], e
Operation "Create Attribute" has been executed (eClass = {Arc}, name = "weight", type = {I
Operation "Create Class" has been executed (ePackage = {lit_petriNets}, name = "Token", s
Operation "Create Reference" has been executed (eClass = {Place}, name = "tokens", type :
Operation "Create Opposite Reference" has been executed (reference = {tokens}, name = "

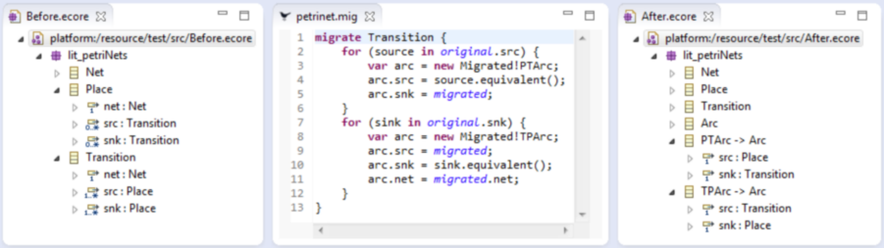
Name	Value
element	Net
name	Net

Violated constraints:
The name must not be already defined by the ch

Fig. 2.10: Instantiations of reusable coupled operators in a history model in Edapt.

2.5.2 Epsilon Flock

Epsilon Flock[6] (subsequently referred to as Flock) is a tool supporting *manual specification* of migrations. It provides a textual transformation language tailored to model migration. In particular, Flock automatically copies all model elements which are not affected by metamodel evolution from original to migrated models. Flock is built on top of Epsilon [473], an extensible platform providing inter-operable languages and tools for model-driven development.



```
migrate Transition {
    for (source in original.src) {
        var arc = new Migrated!PTArc;
        arc.src = source.equivalent();
        arc.snk = migrated;
    }
    for (sink in original.snk) {
        var arc = new Migrated!TPArc;
        arc.src = migrated;
        arc.snk = sink.equivalent();
        arc.net = migrated.net;
    }
}
```

Fig. 2.11: Manual migration specification in Epsilon Flock.

Flock employs a so-called conservative copying algorithm during model migration [729]. This algorithm copies a model element only from the original to the migrated model if there is a class in the evolved metamodel having the same name as the class of the model element. For each copied model element, the algorithm

[6] http://www.eclipse.org/epsilon/doc/flock/

iterates over all its feature values, copying again only those which have a conforming feature in the evolved metamodel. This conforming feature needs to have the same name and the original value needs to conform to it. In our running example, instances of class `Net` and `Place` are automatically copied from original to migrated model.

To migrate model elements or feature values that no longer conform to the evolved metamodel, Flock allows to overwrite the conservative copying algorithm. Therefore, Flock provides two kinds of rules: a rule to migrate instances of a type, and a rule to delete instances of a type. The migration we discussed in Example 2.12 uses only rules of the first kind to migrate instances of `Transition`. In the migration rules, instances can be retyped and feature values can be migrated. In Example 2.12, the values of the new features `src` and `snk` need to be set during migration. Figure 2.11 shows the same migration in the Flock editor, integrated into the Eclipse IDE.

2.5.3 Comparison of Migration and Transformation Tools

The Transformation Tool Contest is a workshop series where participants submit solutions for transformation cases. In 2010, one of the three cases was about migration. In this section, we only give a brief overview of the procedure and the results, but more details can be found in [727]. The goal of the migration case was to answer three research questions:

1. What are the pros and cons of the different transformation tools considering model migration?
2. Which classes of transformation tools are particularly well-suited for realizing a model migration?
3. How do graph transformation tools compare to model transformation and migration tools in a model migration scenario?

The case required to specify the migration of activity diagrams from UML 1.4 to UML 2.2 (see Section 2.3.1). Besides the migration itself, the case defined a number of extensions, covering a different migration semantics, migration of concrete syntax and different model serializations. The study subjects were nine tools: two model migration tools (Flock, COPE), four model transformation tools (ATL/Java, PETE, UML-RSDS, MOLA), and three graph transformation tools (GrGen.NET, Fujaba, GreTL). Participants submitted solutions to the case consisting of an installation on a remote virtual machine, an accompanying description, and a full listing of the solution. Using the artifacts provided, the solutions were reviewed by model and graph transformation experts and by the other participants. At the workshop, the participants presented their solutions and had to face the judgment from an opponent. Finally, the solutions were evaluated by all participants using an evaluation sheet according to the criteria in Table 2.2.

Table 2.2: Criteria for evaluation by participants at the Transformation Tool Contest 2010.

Name	**Description**
Correctness	Does the transformation produce a model equivalent to the migrated UML 2.2 model included in the case resources? Furthermore, does the transformation specification lend itself to reasoning about the correctness of the migration process? *-5 (probably does not work at all), 0 (cannot judge), 5 (works for one model), 10 (works for more than one model)*
Conciseness	How much code is required to specify the transformation? Sprinkle et al. [793] proposed that the amount of effort required to codify migration should be directly proportional to the number of changes between original and evolved metamodel. *-6 (very verbose), -3 (quite verbose), 0 (cannot judge), 3 (quite concise), 6 (very concise)*
Understandability	How easy is it to read and understand the transformation? *-3 (no idea how it works), 0 (some idea how it works), 3 (fully understand how it works)*
Appropriateness	How suitable is the tool for the specific application defined by the case? *-6 (totally inappropriate), -3 (inappropriate), 0 (neutral), 3 (somewhat appropriate), 6 (perfect fit)*
Tool Maturity	How mature is the tool? *-4 (prototype), 0 (average), 4 (good)*
Extensions	To what extent have the extensions defined by the case been solved? *3 points for each completed extension (for a theoretical maximum of 9 points)*

Table 2.3: Per-criterion and overall rank and score for each solution, determined by the participants of Transformation Tool Contest 2010.

Criterion (weight)	**Flock**	**COPE**	**GrGen.NET**	**Fujaba**	**MOLA**	**PETE**	**ATL/Java**	**GReTL**	**UML-RSDS**
Correctness (5)	7	2	2	2	6	1	5	8	9
	3.5	*5.0*	*5.0*	*5.0*	*3.9*	*5.5*	*4.2*	*2.2*	*0.9*
Conciseness (3)	1	2	2	4	5	7	6	8	9
	3.6	*2.5*	*2.5*	*0.3*	*0.0*	*-0.9*	*-0.8*	*-1*	*-1.6*
Understandability (3)	1	2	3	5	4	8	6	7	9
	2.7	*2.2*	*1.4*	*1.0*	*1.3*	*0.3*	*0.8*	*0.7*	*-0.5*
Appropriateness (3)	1	2	3	5	4	7	8	6	9
	4.8	*4.6*	*2.2*	*1.0*	*2.0*	*0.6*	*0.5*	*1.3*	*0.3*
Extensions (3)	1	4	4	2	8	7	3	4	9
	5.4	*3.0*	*3.0*	*4.7*	*0.3*	*2.4*	*4.5*	*3.0*	*0.0*
Tool Maturity (4)	3	5	2	1	4	6	8	7	9
	1.6	*0.7*	*2.2*	*3.6*	*1.3*	*0.0*	*-2.3*	*-0.9*	*-3.3*
Overall ranking	1	2	3	4	5	6	7	8	9
	21.6	*18*	*16.5*	*15.6*	*9.2*	*7.9*	*7.1*	*7.3*	*-4.2*

Table 2.3 shows the ranking and scores for each solution and each criterion as well as the overall ranking and score of each solution. For example, PETE was ranked first for correctness, seventh for appropriateness, and overall sixth. Flock was ranked first and COPE second in front of the model and graph transformation tools. Flock was ranked better than COPE for conciseness, understandability and appropriateness, since the migration is specified directly between source and target metamodel and not as a sequence of operators. However, COPE was ranked significantly better than Flock for correctness, since the migration is recorded together with the evolution, thereby not losing the intention behind the evolution. The statistical analysis on 12 tool evaluations revealed that tools tailored for migration perform significantly better on the criteria conciseness, understandability and appropriateness (the point biserial correlation shows strong impact, and the unpaired t-test confirms significance). A striking similarity between the solutions with the highest three overall rankings (Flock, COPE and GrGen.NET) is that they rely on a retype operation for changing the type of an input element into a different type of the output element. However, the statistical analysis identified no significant differences between imperative and declarative tools. Moreover, in-place execution of migrations performs better on the criteria conciseness and appropriateness (the point biserial correlation shows weak impact). Finally, the statistical analysis identified no significant differences between graph and model transformation tools.

2.5.4 Comparison of Model Migration Tools

The authors of the three migration tools AML, COPE and Flock together performed a comparison of their migration tools. The comparison was done by applying a number of model migration tools to two common migration scenarios. Again, we only give a brief overview of the results, but details can be found in [728]. The goal of the comparison was to answer two research questions:

1. What are the strengths and weaknesses of the different model migration tools?
2. What is the most appropriate model migration tool for a certain situation?

The study objects were two migration scenarios: the small, artificial Petri net evolution known from the literature and discussed in Section 2.1, and a real-life evolution from the GMF open-source project as discussed in Section 2.3.3. The study subjects were four tools from different categories of approaches: two manual specification tools (Ecore2Ecore [415], Flock), one metamodel matching tool (AML), and one operator-based tool (COPE). Each tool was assigned to a person different from its author and the evaluation criteria were identified. Table 2.4 shows the resulting evaluation criteria. The study participants had to familiarize themselves with the tools using the small example, before the tools were applied to the larger example and experiences were recorded with the application. The study participants compiled the experiences by criterion and synthesized a guide for selecting a tool.

Table 2.4: Summary of criteria and results for the comparison of model migration tools.

Name	**Description**	**AML**	**COPE**	**Ecore2Ecore**	**Flock**
Construction	Ways in which tool supports developing migration strategies	○	○	−	+
Change	Ways in which tool supports change to migration strategies	+	+	○	+
Extensibility	Extent to which user-defined extensions are supported	+	+	−	−
Reuse	Mechanisms for reusing migration patterns and logic	+	+	−	○
Conciseness	Size of migration strategies produced with tool	−	+	−	+
Clarity	Understandability of migration strategies produced with tool	−	+	−	+
Expressiveness	Extent to which migration problems can be codified with tool	−	+	○	+
Interoperability	Technical dependencies and procedural assumptions of tool	○	−	○	+
Performance	Time taken to execute migration	+	−	+	○

Table 2.4 also shows the resulting assessment of the different tools according to the evaluation criteria. + means that the tool was strong in the criterion, − that it was weak, and ○ that it was neither strong nor weak. Each tool has its strengths and weaknesses. Consequently, there is no tool that can be recommended for all situations. Table 2.5 summarizes the recommendations and guidelines in choosing a migration tool that were synthesized from the presented results.

Table 2.5: Summary of model migration tool selection advice.

	Recommended Tools			
Requirement	**AML**	**COPE**	**Ecore2Ecore**	**Flock**
Frequent, incremental co-evolution		●		
Reverse-engineering	●		●	●
Modeling technology diversity				●
Quicker migration for larger models	●		●	
Minimal dependencies			●	
Minimal hand-written code	●	●		
Minimal guidance from user	●			
Support for language-specific migrations		●		●

COPE is ideal for dealing with frequent, incremental co-evolution and requires hand-written code only to express language-specific migrations. However, COPE is not a perfect match when performance of the migration is important and when reverse engineering a migration. In contrast, the other three tools are perfect for reverse engineering the migration after the metamodel evolution has been performed. Generating the migration, ATL requires minimal hand-written code and minimal guidance from the user, but does not support language-specific migrations. The low-level Ecore2Ecore is optimized towards high performance and depends only on few other libraries and tools, but is not very expressive and user-friendly. Based on a transformation language and framework, Flock supports language-specific migrations and many modeling technologies, but requires some effort and guidance to manually specify the migration.

2.5.5 Discussion of Tool Support

Even though quite a number of prototypical tools have been built for the approaches, only two tools are still available today. There is Flock for a manual specification approach and Edapt for an operator-based approach. These two tools have been compared to the metamodel matching tool AML which is no longer available today. In any case, AML's implementation used in the comparison was not mature enough to really compete with the other tools. Thus, building a tool following a metamodel matching approach and showing that it really works in practice, is still an open issue. For the other two tools, the comparison revealed that Flock is better suited for reverse engineering the model migration after the metamodel evolution, while Edapt is better suited for forward engineering the model migration together with the metamodel evolution. These two model migration tools have been also compared to a number of general-purpose model and graph transformation tools. This comparison showed that model and graph transformation tools can be used to specify the migration, but are significantly less suited than special-purpose model migration tools.

2.6 Conclusions

In model-driven software engineering, software models are the primary engineering artifacts. These models are expressed by means of modeling languages. Like software, modeling languages and thus their metamodels are subject to evolution due to changing requirements. When a metamodel is adapted to the new requirements, existing models may no longer conform to it and need to be migrated. This chapter discussed the problem of coupled evolution of metamodels and models and how it can be addressed.

First, the problem was classified according to a number of dimensions: the impact on the modeling language, the impact on existing models, the reversibility of the coupled evolution, as well as the potential for automating the migration. If the coupled evolution is reversible, then at least no information is lost in the model. However, reversibility does not ensure that the meaning of the model is preserved. Even though there are some definitions of meaning preservation [396, 397, 792], further work is needed to be able to prove that a migration preserves the meaning of all models. An important part of this work is to provide explicit semantics specifications for modeling languages.

Second, the chapter gave an overview of the available empirical results for the coupled evolution of metamodels and models. In practice, model migration can be automated to a large extent, and modeling language evolution turned out to be quite similar to general software evolution. However, additional studies are necessary to further substantiate the results of these empirical studies. With the increasing adoption of model-based software development in practice, more and more metamodel histories should be available for analysis.

Third, the existing approaches addressing the problem of coupled evolution were presented and classified according to a feature model. There are three main categories of approaches: manual specification of the migration, operator-based specification of the coupled evolution, as well as the generation of the migration from a metamodel matching. While the first two categories have been extensively studied, there is not yet a mature approach for the third category. However, there are some promising directions that generate operator sequences from a metamodel matching [493, 890].

Fourth, the chapter presented the available model migration tools and a comparison with each other as well as to general-purpose model transformation tools. Unfortunately, from the many proposed approaches, there are only a few tools available that can still be obtained from the internet. The comparison of the model migration showed that the different categories of approaches favor different usage scenarios: Manual specification and metamodel matching approaches favor reverse engineering of the model migration, while operator-based approaches favor the forward engineering of the coupled evolution. Moreover, special-purpose model migration tools outrank general-purpose model transformation tools when focusing on model migration scenarios.

Finally, not only the models need to be migrated, when the metamodel changes, but also other artifacts, like concrete syntax definitions or transformation specifications. There are already a number of results for the migration of graphical syntax defined with GMF [248] or the migration of model transformations [588, 726]. However, more work is necessary to unify these approaches with the approaches for model migration.

Chapter 3
Software Product Quality Models

Rudolf Ferenc, Péter Hegedűs and Tibor Gyimóthy

Summary. Both for software developers and managers it is crucial to have information about different aspects of the quality of their systems. This chapter gives a brief overview about the history of software product quality measurement, focusing on software maintainability, and the existing approaches and high-level models for characterizing software product quality. The most widely accepted and used practical maintainability models and the state-of-the-art works in the subject are introduced. These models play a very important role in software evolution by allowing to estimate future development costs, assess risks, or support management decisions. Based on objective aspects, the implementations of the most popular software maintainability models are compared and evaluated. The evaluation includes the Quality Index, SQALE, SQUALE, SIG, QUAMOCO, and Columbus Quality Model. The chapter presents the result of comparing the features and stability of the tools and the different models on a large number of open-source Java projects.

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_3, © Springer-Verlag Berlin Heidelberg 2014

3.1 Introduction

The need for measuring the quality of software products is almost as old as software engineering itself. Software product quality monitoring has become one of the central issues of software development and evolution. Both for software developers and managers it is crucial to have clues about different aspects of the quality of their systems. The information is mainly used in making decisions during software evolution (e. g., to start a refactoring phase or reimplement a system because of wear out), backing up intuition, estimating future costs and assessing risks.

A large number of quality models, measures, and approaches have been introduced in the past. These software quality assessment models belong to one of the following types:

1. *Software Process Quality Models* – the idea behind these models is that they measure and improve the software development process. These models are based on the assumption that better development processes lead to better quality software products. These models make their estimation based on different process metrics (e. g., defect removal efficiency, percentage of management effort for a given project size, average age of unresolved issues). Some of the well-known process quality models are SPICE [259], ISO/IEC 9001 (Quality management systems – Requirements) [427], and Capability Maturity Model Integration (CMMI) [184].

2. *Software Product Quality Models* – these models measure the software product itself. They measure different kinds of source code metrics (e. g., Lines of Code, McCabe's cyclomatic complexity, coupling) and combine them somehow to assess the quality of the product. Early quality models are McCall's [574] and Boehm's [122] models followed by the standard ISO/IEC 9126 [422] and its successor ISO/IEC 25000 (SQuaRE) [425]. Many practical product quality models have been derived from these standards since then (e. g., ColumbusQM [63], SIG [387], SQALE [516], SQUALE [619], QUAMOCO [905]).
3. *Hybrid Software Quality Models* – these models combine the previous approaches: they calculate both product- and process-based metrics to assess the quality of software, like in the work of Nagappan et al. [630]. Particularly, they added line changes, code churn and other process metrics to software product metrics and built a hybrid model for post-release failure prediction.

This book chapter deals only with the second type of models that assess the software quality based on software product metrics.

Even though early product quality models have appeared in 1977, right after the introduction of the first source code metrics, the explosion of new practical quality models has started after 1991 with the appearance of the ISO/IEC 9126 software product quality standard (see Figure 3.1). This standard defines six high-level product quality characteristics: *functionality*, *reliability*, *usability*, *efficiency*, *maintainability* and *portability*. The characteristics are affected by low-level quality properties, that can either be *internal* (measured by looking inside the product, e. g., by analyzing the source code) or *external* (measured by execution of the product, e. g., by performing testing).

In the context of software evolution, which is the focus of this book, maintainability is probably the most attractive, observed and evaluated quality characteristic of all (discussed in more details later on in Section 3.2). The importance of maintainability lies in its very obvious and direct connection with the costs of altering the behavior of the software [62]. Although, the quality of source code unquestionably affects maintainability, the standard does not provide a common set of source code measures as internal quality properties. The standard also does not specify the way how the aggregation of quality attributes should be performed. Thus it offers a kind of freedom to adapt the model to specific needs.

Many researchers took the advantage of this freedom and a number of practical quality models have been proposed so far [1, 51, 63, 68, 387, 516, 619, 905]. Most of the models discussed in this chapter share some basic common principles:

- They extract information from the source code, therefore they assess quality properties related to software maintainability. However, we often refer to these models as quality models as they define quality to be the maintainability of the code.
- Each of them uses a hierarchical model (e. g., Figure 3.2) for estimating quality with some kinds of metrics at the lowest level. In the case of each considered source code metric, its distribution over the source code elements is taken. Either the whole distribution, or a number (e. g., average), or a category (based on threshold values) is used for representation.
- The number or category is aggregated “upwards” in the model by using some kind of aggregation mechanism (weighting or linear combination, etc).

Many of these practical quality models have been implemented and integrated into modern tools supporting software evolution. They allow a continuous insight into the quality of the software product under development. Moreover, many other direct applications of these models exists. Besides system level qualification some of them provide a list of critical elements that programmers should fix in order to improve the overall maintainability of the source code. As an example, Section 3.2.4.5 presents a drill-down approach demonstrating a sophisticated technique for deriving maintainability values at source code element level. Another popular field of application of these models is in the cost estimation of future development efforts [62].

This chapter is organized as follows. Section 3.2 gives a historical overview of software product quality measurement starting from the first software metrics through simple metrics-based prediction models and early theoretical quality models to the state-of-the-art practical quality models. In Section 3.3, an application of practical quality models during software evolution is introduced. Section 3.4 collects and describes some of the available tools implementing the modern practical quality models. Then, in Section 3.5 we evaluate the introduced practical models and their implementing tools. First, we compare the models behind the tools based on a set of evaluation criteria, and afterwards we present our experiences of using the tools by analyzing different open-source Java projects. Finally, we conclude the chapter in Section 3.6 and list some of the future research directions in the field.

3.2 Evolution of Software Product Quality Models

The need for measuring the quality of the software products has almost the same age as software engineering itself. The measuring approaches have gone through a rigorous evolution during the past 50 years. The history of software product quality measurement is presented on the timeline in Figure 3.1.

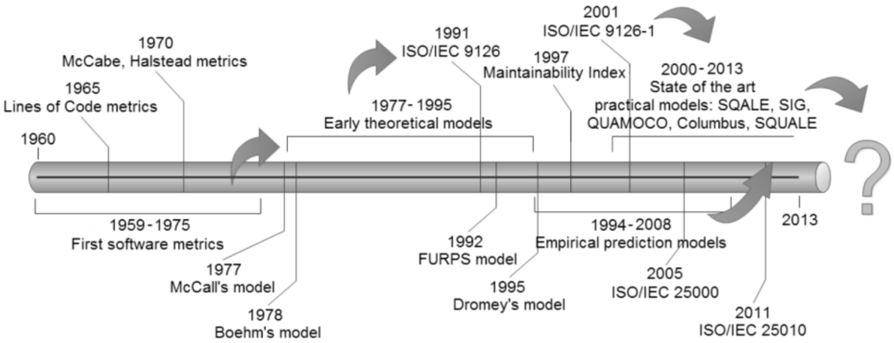


Fig. 3.1: The history of software quality measurement

The first tools for assessing product quality were simple metrics like Lines Of Code, McCabe complexity or Halstead's metrics. They started to appear from the mid 1960's. The growing number of metrics has inspired the appearance of the early theoretical quality models like McCall's [574] or Boehm's model [122] at the end of the 1970's. They all tried to capture high-level quality properties based on a hierarchical model. In 1990's all these theoretical models have been merged into the robust ISO/IEC 9126 [422] software product quality standard that had a huge impact on further quality models. The standard has been revised resulting in a new edition in 2005, marked as ISO/IEC 25000 (Systems and software Quality Requirements and Evaluation – SQuaRE) [425].

Another branch of quality assessment approach that started from the mid 1990's is the development of empirical prediction models using software metrics as predictors. These approaches try to predict software quality by using different techniques like regression [663], neural networks [952], or Naive-Bayes classifiers [869] based on empirical studies. One well-known such model is the Maintainability Index.

To overcome the complexity and lack of application details of the ISO standards as well as the hard interpretation and explicability of the empirical prediction models, a whole set of new practical quality models have been introduced in the past few years (e. g., ColumbusQM [63], SIG [387], SQALE [516], SQUALE [619], QUAMOCO [905]). Most of these models follow the structure of the ISO standards but also define concrete source code metrics and algorithms for aggregating them to higher levels of the hierarchical model. The problem of the hard interpretation of the results has been addressed by utilizing so-called reference systems (benchmarks) that serve as the basis of the qualification. As another possible solution the concept of *technical debt* [146] has been introduced. This term was coined by Ward Cunningham to describe the obligation that a software organization incurs when it chooses a design or construction approach that is expedient in the short term but that increases complexity and is more costly in the long term.

Although the models share a lot of properties, they also differ in many aspects. It is a very interesting open question if these practical models can be unified and merged into a common standard like it was done with the early theoretical models. Our vision is that these practical models can be merged into a common standard in the future to form a whole new direction of software quality assessment.

This section gives an overview of the evolution of software quality measurements and approaches starting from the first software metrics through simple metrics-based prediction models and early theoretical quality models to the state-of-the-art practical quality models. At the end of the section we also present some of the current applications of the existing practical quality models.

3.2.1 Software Metrics

Although the first dedicated book on software metrics was not published until 1976 [326], according to the software metrics roadmap of Fenton and Neil [290], the history of active software metrics dates back to the mid 1960's when the *Lines of Code* (LOC) metric was used as the basis for measuring programming productivity and effort. In the late 1960's LOC was also used as the basis for measuring program quality (normally measured indirectly as defects per KLOC). One of the first prediction models was presented in 1971 by Akiyama [10] when he proposed a regression-based model for module defect density prediction in terms of the module size measured in KLOC.

Starting from the mid 1970's an explosion of interest arose in the measures of software complexity (pioneered by Halstead [361] and McCabe [573]) and measures of functional size (such as function points pioneered by Albrecht [5]), which were intended to be independent of the programming language of choice. The Halstead complexity and McCabe cyclomatic complexity became the main predictors of different quality aspects and effort estimation. Early theoretical quality models (McCall's, Boehm's, etc.) have started to appear also in the mid 1970's and were based on software metrics.

With the appearance of new programming paradigms such as object-orientation a whole new set of metrics have been developed. The most well-known metric suite for OO systems was introduced by Chidamber and Kemerer [180]. Since their appearance, these OO metrics have been used to characterize, evaluate and improve the design of large applications [495]. This variety of software metrics also inspired works on new prediction models for software quality and effort estimation.

3.2.2 Early Theoretical Quality Models

The approaches for modeling software quality appeared right after the introduction of the first software metrics. One of the earliest documented quality model [574] was created by McCall et al. in 1977. McCall produced this model for the US Air Force and he attempted to bridge the gap between users and developers by focusing on a number of software quality factors that reflect both the users' views and the developers' priorities. The structure of McCall's quality model consists of three major perspectives (types of quality characteristics) for defining and identifying the quality of a software product, and each of these major perspectives consists of a number of quality factors. Each of these quality factors have a set of quality criteria, and each quality criterion could be reflected by one or more metrics. The perspectives are:

1. **Product revision**
   The product revision perspective identifies quality factors that influence the ability to change the software product, these factors are:
   - *Maintainability*, the ability to find and fix a defect.
   - *Flexibility*, the ability to make changes required as dictated by the business.
   - *Testability*, the ability to validate the software requirements.
2. **Product transition**
   The product transition perspective identifies quality factors that influence the ability to adapt the software to new environments:
   - *Portability*, the ability to transfer the software from one environment to another.
   - *Reusability*, the ease of using existing software components in a different context.
   - *Interoperability*, the extent, or ease, to which software components work together.
3. **Product operations**
   The product operations perspective identifies quality factors that influence the extent to which the software fulfills its specification:
   - *Correctness*, the functionality matches the specification.
   - *Reliability*, the extent to which the system fails.
   - *Efficiency*, system resource (including CPU, disk, memory, network) usage.
   - *Integrity*, protection from unauthorized access.
   - *Usability*, ease of use.

In total, McCall identified 11 quality factors broken down by 3 perspectives, as listed above.

In 1978, Boehm et al. [122] also defined a hierarchical model of software quality characteristics, trying to qualitatively define software quality as a set of attributes and metrics. It consists of high-level characteristics, intermediate-level characteristics and lowest level (primitive) characteristics which contribute to the overall quality level. At the highest level of his model, Boehm defined three primary uses (or basic software requirements), which are the following:

- **As-is utility**, the extent to which the as-is software can be used (i. e., ease of use, reliability and efficiency).
- **Maintainability**, ease of identifying what needs to be changed as well as ease of modification and retesting.
- **Portability**, ease of changing software to accommodate a new environment.

In the intermediate level, there are seven quality characteristics that represent the qualities expected from a software system:

- *Portability*, the extent to which the software will work under different computer configurations (i. e., operating systems, databases etc.).
- *Reliability*, the extent to which the software performs as required, i. e., the absence of defects.
- *Efficiency*, optimum use of system resources during correct execution.
- *Usability*, ease of use.
- *Testability*, ease of validation, that the software meets the requirements.
- *Understandability*, the extent to which the software is easily comprehended with regard to purpose and structure.
- *Flexibility*, the ease of changing the software to meet revised requirements.

The primitive characteristics can be used to provide the foundation for defining quality characteristics; this use is one of the most important goals established by Boehm when he constructed his quality model.

In 1995, Dromey [262] presented a product based quality model that recognizes that quality evaluation differs for each product. He realized that a more dynamic idea for modeling the evaluation process is needed to be general enough to be successfully applied for different systems. Dromey was focusing on the relationship between the quality attributes and the sub-attributes, as well as attempting to connect software product properties with software quality attributes. Dromey's quality model is structured around a 5 step process:

1. Choose a set of high-level quality attributes necessary for the evaluation.
2. List components/modules in your system.
3. Identify quality-carrying properties for the components/modules (qualities of the components that have the biggest impact on the product properties from the list).
4. Determine how each property affects the quality attributes.
5. Evaluate the model and identify weaknesses.

The FURPS [344] model was originally presented by Robert Grady at Hewlett Packard in 1992, then it has been extended by IBM Rational Software into FURPS+, where the '+' indicates such requirements as design constraints, implementation requirements, interface requirements and physical requirements. Under the FURPS model, the following characteristics are used:

- **Functionality** - it may include feature sets, capabilities, and security.
- **Usability** - it may include human factors, aesthetics, consistency in the user interface, online and context sensitive help, wizards and agents, user documentation, and training materials.
- **Reliability** - it may include frequency and severity of failure, recoverability, predictability, accuracy, and mean time between failures.
- **Performance** - it imposes conditions on functional requirements such as speed, efficiency, availability, accuracy, throughput, response time, recovery time, and resource usage.
- **Supportability** - it may include testability, extensibility, adaptability, maintainability, compatibility, configurability, serviceability, installability, and localizability.

ISO/IEC 9126 [422] is an international standard for the evaluation of software products. The standard is divided into four parts which address, respectively, the following subjects: quality model; external metrics; internal metrics; and quality in use metrics. ISO/IEC 9126 Part one, referred to as ISO/IEC 9126-1 is an extension of the work done by McCall, Boehm, Grady and others in defining a set of software quality characteristics. The standard defines six high-level product quality characteristics which are widely accepted both by industrial experts and academic researchers. These characteristics are: *functionality*, *reliability*, *usability*, *efficiency*, *maintainability* and *portability*. Table 3.1 shows the characteristics defined by the standard together with their sub-characteristics.

In the context of software evolution, maintainability is one of the most observed and evaluated quality characteristics (see Table 3.3). The importance of maintainability lies in its direct connection with the costs of changing the software, either by performing bug-fixes, refactoring it or adding new features. Although the source code quality directly affects maintainability, the standard does not provide a common set of source code measures as internal quality properties. The standard also does not specify how the aggregation of quality attributes should be performed. These are not deficiencies of the standard, but it offers a kind of freedom to adapt the model to specific needs.

The successor of the ISO/IEC 9126 standard family is the ISO/IEC 25000 (Systems and software Quality Requirements and Evaluation – SQuaRE) [425] family. It introduces slight modifications to the previous standard which are mainly terminology changes. Table 3.2 lists the quality characteristics and subcharacteristics of the most recent standard.

Table 3.3 provides an overview of the described theoretical quality models. The first four rows show some basic characteristics of the models based on the work of Fahmy et al. [282]. The first row contains the number of levels of the hierarchical

Table 3.1: The ISO/IEC 9126 characteristics and subcharacteristics

Characteristics	Subcharacteristics	Characteristics	Subcharacteristics
Functionality	Suitability Accuracy Interoperability Security Functionality Compliance	Maintainability	Analyzability Changeability Stability Testability Maintainability Compliance
Reliability	Maturity Fault Tolerance Recoverability Reliability Compliance	Efficiency	Time Behavior Resource Utilization Efficiency Compliance
Usability	Understandability Learnability Operability Attractiveness Usability Compliance	Portability	Adaptability Installability Co-Existence Replaceability Portability Compliance

Table 3.2: The ISO/IEC 25000 (SQuaRE) characteristics and subcharacteristics

Characteristics	Subcharacteristics	Characteristics	Subcharacteristics
Functional suitability	Functional Appropriateness Functional Correctness Functional Completeness	Portability	Adaptability Installability Replaceability
Security	Confidentalility Integrity Non-repudiation Accountability Authenticity	Usability	Appropriateness Recognisability Learnability Operability User error protection User interface aesthetics Accessibility
Maintainability	Modularity Reusability Analysability Modifiability Testability	Reliability	Availability Fault tolerance Recoverability Maturity
Performance efficiency	Time-bahaviour Resource utilisation Capability	Compatibility	Co-existence Interoperability

model. Second row shows the relation types between the quality attributes in the model. After that rows three and four highlight the main advantages and disadvantages of the particular models. In the following rows the quality attributes of the different models are presented. Only quality attributes at the highest level are considered. It can be seen that there are a lot of common properties among the models. However, only Reliability appears at the highest level in each model. Maintainability, Efficiency, Usability and Portability are also very common attributes, they

Table 3.3: Comparison of theoretical quality models [13, 282]

Characteristics	**McCall**	**Boehm**	**Dromey**	**FURPS**	**ISO 9126**	**ISO 25000**
Nr. of levels	2	3	2	2	3	3
Relationship	Many-Many	Many-Many	One-Many	One-Many	One-Many	One-Many
Main advantage	Evaluation Criteria	Hardware Factors Included	Different Systems	Separation of FR & NFR	Evaluation Criteria	Evaluation Criteria
Main disadvantage	Components Overlapping	Lack of Criteria	Comprehen-siveness	Portability not Considered	Generality	Generality
Quality Attributes						
Maintainability	✓	✓	✓		✓	✓
Flexibility	✓					
Testability	✓	✓				
Correctness	✓					
Efficiency	✓	✓	✓		✓	✓
Reliability	✓	✓	✓	✓	✓	✓
Integrity	✓					
Usability[1]	✓	✓	✓	✓	✓	✓
Portability	✓	✓	✓		✓	✓
Reusability	✓		✓			
Interoperability	✓					
Understandability		✓				
Modifiability		✓				
Functionality			✓	✓	✓	✓
Performance				✓		✓
Supportability				✓		
Security						✓
Compatibility						✓

[1] Also referred to as Human Engineering in some models

appear in 5 out of the 6 models. Moreover, these properties are contained in each model just not at the highest level everywhere (e. g., in FURPS Maintainability is included in the Supportability characteristic). On the other hand, there are also attributes that are specific to one model e. g., Supportability, Security, Compatibility, etc. Further reading about these theoretical quality models can be found in the work of Al-Qutaish [13].

3.2.3 *Metrics-based Empirical Prediction Models*

The first step towards applying quality models in practice was the development of different empirical models. All these models apply software metrics as quality predictors.

One of the most widely known empirical maintainability prediction models is the *Maintainability Index* (MI) [662] introduced in 1997 by the Carnegie Mellon Software Engineering Institute (SEI). The formula has many derivatives, but the original form is given in Equation 3.1. A common variation, shown in Equation 3.2, adds the comment lines to the model.

$$MI = 171 - 5.2 * \ln V - 0.23 * G - 16.2 * \ln LOC \tag{3.1}$$

$$MI = 171 - 5.2 * \log_2 V - 0.23 * G - 16.2 * \log_2 LOC + 50 * \sin\left(\sqrt{2.46 * CM}\right) \tag{3.2}$$

The applied measures are the following:

- V - Halstead Volume.
- G - Cyclomatic Complexity.
- LOC - count of source Lines Of Code (SLOC).
- CM - percent of lines of Comments.

The CM percentage in the maintainability index formula has been interpreted in two different ways. Liso [530] assumed CM to range between 0 and 100 and discussed the appropriateness of the value 2.46 leading to strange peaks in $\sin\left(\sqrt{2.46 * CM}\right)$. Thomas [831] has assumed CM to range between 0 and 1.

Later on, many variations of the formula have been introduced, e. g., one of its derivatives is built into Microsoft Visual Studio as well. However, the effectiveness and usefulness of the maintainability index is and has been a subject of debate [142, 289, 387, 483].

Therefore, a wide variety of new approaches has been introduced for improving the MI. The applied methods are ranging from regression models to fuzzy aggregation and Bayes classifiers (see Table 3.4). These empirical studies also differ in which metrics have found to be the most effective maintainability predictors. Riaz et al. presented a detailed comparative study [714] on existing empirical maintainability prediction models. The work collects many important features of the different empirical models, e. g., the definition of quality the authors used, the applied validation methodology, or the accuracy of the model. Table 3.4 contains a summary of some well-known works in the field and their important properties.

Table 3.4: A summary of empirical maintainability prediction models [714]

Authors	**Year**	**Approach**	**Maintainability Metrics**
Oman, and Hagemeister [663]	1994	3 regression based models: 1. Single metric model based on Halstead's Effort 2. A four-metric polynomial model 3. A five-metric linear regression model	Subjective assessment (ordinal scale metric) by using the US Air Force Operational Test and Evaluation Center's software maintainability evaluation instrument, which provides a rating as well as categorizes maintainability as low, medium or high
Coleman, Ash, Lowther, and Oman. [196, 197]	1994 1995	1. HPMAS (Hewlett Packard's software Maintainability Assessment System) 2. Polynomial maintainability assessment model	Same as first but they call it HPMAS Maintainability Index
Welker, Oman [912]	1997	1. Improved, three-metric MI model 2. Improved, four-metric MI model	Same as first but they call it HPMAS Maintainability Index
Genero, Olivas, Piattini, and Romero, F. [319]	2001	Fuzzy Prototypical Knowledge Discovery used for prediction based on Fuzzy Deformable Prototypes	Expert opinion using an ordinal scale
Misra [609]	2005	6 models based on multivariate regression analysis	Maintainability Index (MI)
Van Koten, Gray [869]	2006	1. Bayesian Network - Naive-Bayes Classifier 2. Regression Models (Regression Tree, Multiple linear regression models)	CHANGE metric: count of LOC changed during a 3-year maintenance period
Shibata, Rinsaka, Dohi, and Okamura [774]	2007	3 Non-Homogeneous Poisson Process based Software Reliability Models (NHPP-based SRMs): 1. Exponential SRM 2. S-Shaped SRM 3. Rayleigh SRM	Their own queueing model with an infinite number of servers, which is related to the software fault-detection/correction profiles
Zhou, and Leung [952]	2007	Multivariate Linear Regression, Artificial Neural Network, Regression Tree, Support Vector Regression, Multivariate Adaptive Regression Splines	CHANGE metric: count of LOC changed during a 3-year maintenance period
Zhou, and Xu [953]	2008	1. Univariate Linear Regression Analysis 2. Multivariate Linear Regression Model	Maintainability Index (MI)

3.2.4 State-of-the-art Practical Quality Models

The appearance of the widely accepted ISO/IEC 9126 and related standards [422] has pushed forward the research in the field of quality models. Numerous papers, ranging from highly theoretical to purely practical ones, are dealing with this important research area. Some of the research has focused on developing a methodology for adapting the ISO/IEC 9126 model in practice [119, 807]. They provide guidelines or a framework for constructing effective quality models.

This section focuses more on practical models that are directly applicable for assessing the quality of software systems. Using the results of static source code analysis is one of the most widespread solutions to calculate an external quality attribute from internal quality attributes [71]. There are several case studies examining if metrics are appropriate indicators for external quality attributes such as code fault proneness [357, 660], maintainability [59] and attractiveness of the user interface [584].

The majority of these practical models consider the maintainability aspect of quality only, because it is the easiest characteristic to assess based on pure source code analysis. Some of the models consider other quality attributes as well, like usability (often requiring manual input for the qualification). Regarding the terminology, we use quality model and maintainability model as synonyms throughout this section.

3.2.4.1 Software QUALity Enhancement project (SQUALE)

The SQUALE model presented by Mordal et al. [619] introduces so-called practices to connect the ISO/IEC 9126 characteristics to metrics. A practice in a source code element expresses a low-level rule and the reparation cost of violating this rule. The reparation cost of a source code element is calculated by the sum of the reparation costs of its rule violations. The practices can use multiple source code measures like complexity, lines of code, coding rule violations, etc. (e. g., the comment rate practice uses the measures cyclomatic complexity v(G) and source code lines, SLOC). Based on the measures, a practice rating in the [0;3] interval can be calculated, where 3 means the fully achieved goal, 0 means not achieved goal, 1 and 2 means partly achieved goal. In the case of the comment rate practice, the rating can be determined according to the following rule:

Comment rate practice

```
if $v(G) < 5$ and $SLOC < 30$ then
    $rating = 3$
else
    $rating = \frac{\%\_comments\_per\_loc}{1-10^{(-v(G)/15)}}$
end if
```

A criterion assesses one principle of software quality (e. g., safety, simplicity, or modularity) and it aggregates a set of practices. A criterion mark is computed as the weighted average of the composed practice marks. There are different weighting profiles, e. g., hard, medium, soft.

A factor represents the highest quality assessment to provide an overview of project health (e. g., functional capacity or reliability). A factor aggregates a set of criteria and its mark is computed as the average of the composed criteria marks.

The model also defines a so-called improvement plan that gives the order in which the elements should be improved. The plan is based on how to achieve the biggest improvement in the rating with the lowest invested effort.

3.2.4.2 Software Quality Assessment based on Lifecycle Expectations (SQALE)

The SQALE quality model introduced by Letouzey and Coq [516] is basically a requirements model. Assessing software source code is therefore similar to measuring the distance which separates it from its quality target.

The model consists of quality characteristics built on top of development activities following one another. The characteristics are taken from the ISO/IEC 9126 standard; however, they are grouped differently and their subcharacteristics are changed entirely. Each subcharacteristic is measured by a number of different control points. The control points are base measures (indicators) that measure different non-compliance aspects of the source code. e. g., an understandability (a subcharacteristic of maintainability) indicator is the file comment ratio. If it is below SQALE's default threshold of 25%, a violation is counted.

Every rule violation has a remediation effort (which depends on the rule). The model calculates an index for every characteristic which is the sum of all the remediation efforts of its rule violations. The index represents the remediation effort which would be necessary to correct the non-compliances detected in the component, versus the model requirements. Since the remediation index represents a work effort, the consolidation of the indices is a simple addition of uniform information. In this way coding rule violation non-compliances, threshold violations for a metric or the presence of an antipattern non-compliance can be compared using their relative impact on the index.

Besides these remediation indices the model presents a five level rating for the different components or the system as a whole. The ratings are A, B, C, D, E (A being the best, E the worst) and can be calculated by summing the remediation costs of the rule violations for a component divided by the average development cost of reimplementing the same component (estimated from LOC). Based on preset thresholds for this ratio a rating can be derived (e. g., if the ratio is less than 0.1% then the rating is A).

3.2.4.3 Quamoco Quality Model

The Quamoco quality framework [905] is the result of a German national research project carried out between 2009 and 2011. The Quamoco Consortium – consisting of research institutions and companies – has developed a quality standard applicable in practice that makes the performance and efficiency of software products made in Germany assessable and accountable.

Quamoco is based on practical experiences learnt from existing quality models. The high-level of detail of this approach for the qualified certification of software projects also takes into account the diversity of different software products. This means that Quamoco contains a basic standard of quality that is complemented by domain-specific quality standards. The quality of software products can thus be modeled flexibly. At the same time, Quamoco ensures that all identified quality requirements are fully integrated.

The Quamoco approach uses the following definitions:

- Quality Model: A model with the objective to describe, assess and/or predict quality.
- Quality Meta Model: A model of the constructs and rules needed to build specific quality models.
- Quality Modeling Framework: A framework to define, evaluate and improve quality. This usually includes a quality metamodel as well as a methodology that describes how to instantiate the metamodel and use the model instances for defining, assessing, predicting and improving quality.

The main concepts of the quality model are *Factors*. A factor expresses a property of an entity. Entities are the things that are important for quality. Properties describe the attributes of the entities. This concept of a factor is rather general. Thus, the Quamoco model uses it on two levels of abstraction:

- Quality Aspects describe abstract quality goals defined for the whole product. The quality model uses the "-ilities" of ISO/IEC 25000 as quality aspects. Typical examples for such quality aspects are Maintainability, Analysability, and Modifiability.
- Product Factors describe concrete, measurable properties of concrete entities. An example for a factor is the Complexity of a method, which can be measured by the cyclomatic complexity number, or by the nesting depth of the method.

To close the gap between abstract quality aspects and measurable product factors, the product factors need to be set in relation to the quality aspects. This is done via *Impacts*. An impact is either positive or negative and describes how the degree of presence or absence of a product factor influences a quality aspect.

A third layer in the levels of abstraction are *Measures*, which describe how a specific product factor can be quantified. To realize the connection to concrete tools in a quality assessment, the approach further introduces *Instruments*. An instrument describes a concrete implementation of a measure. For the example of the nesting depth, an instrument is the corresponding metric as implemented in the quality analysis framework ConQAT [234]. This way, different tools can be used for a single measure.

In order to fully utilize the quality model, aggregation formulas need to be specified. They are called *Evaluations* and they are assigned to the factors in the quality model.

3.2.4.4 SIG Maintainability Model

Kuipers and Visser introduced a maintainability model [483] as a replacement of the Maintainability Index by Oman and Hagemeister [662]. Based on this work Heitlager et al. [387], members of the Software Improvement Group (SIG) company proposed an extension of the ISO/IEC 9126 model that uses source code metrics at low-level. Metric values are split into five categories, from poor (−−) to excellent (++). The evaluation in their model means summing the values for each attribute (having the values between -2 and +2) and then aggregating the values for characteristics using the mapping presented in Table 3.5. The model was recently adapted to the ISO/IEC 25000 standard.

Table 3.5: The SIG quality characteristic mapping

	Volume	**Complexity**	**Duplications**	**Unit size**	**Unit tests**
Analysability	✓		✓	✓	✓
Changeability		✓	✓		
Stability					✓
Testability		✓		✓	✓

Correia and Visser [202] presented a benchmark that collects measurements of a wide selection of systems. This benchmark enables systematic comparison of technical quality of (groups of) software products. Alves et al. presented a technique for deriving metric thresholds from benchmark data [24]. This method is used to derive more reasonable thresholds for the SIG model as well.

Correia and Visser [203] introduced a certification method that is based on the SIG quality model. The method makes it possible to certify technical quality of software systems. Each system can get a rating from one to five stars (−− corresponds to one star, ++ to five stars). Baggen et al. [58] refined this certification process by doing regular re-calibration of the thresholds based on the benchmark.

The SIG model uses binary relation between system properties and characteristics. Correia et al. created a survey [201] to elicit weights for their model. The survey was filled out by IT professionals, but the authors finally concluded that using weights does not improve their quality model because of the lack of consensus among developers.

The validation of the model has been done through an empirical case study. Luijten and Visser [543] showed that the metrics of the SIG quality model correlate with the time needed for resolving a defect in a software.

3.2.4.5 Columbus Quality Model

This subsection describes the Columbus Quality Model (ColumbusQM) in full technical details to give an insight for the reader about the complexity of a modern maintainability model.

The Columbus approach [63] to compute ISO/IEC 9126 quality characteristics uses a so-called benchmark (i. e., a source code metric repository database consisting of source code metrics of open-source and industrial software systems) and it is based on a directed acyclic graph (see Figure 3.2), whose nodes correspond to quality properties that can either be internal or external. The nodes representing internal quality properties are called *sensor nodes* (white nodes in Figure 3.2) as they measure internal quality directly. The other nodes are called *aggregate nodes* as they acquire their measures through aggregation. The approach uses aggregate nodes defined by the ISO/IEC 9126 standard (dark gray nodes in Figure 3.2) as well as newly defined ones (light gray nodes in Figure 3.2).

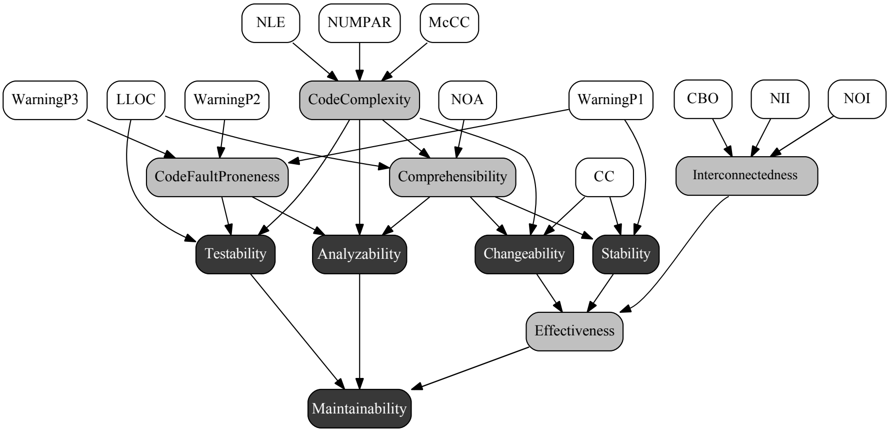


Fig. 3.2: Java Attribute Dependency Graph of ColumbusQM

The edges of the graph represent dependencies between an internal and an external or two external properties. Internal properties are not dependent on any other attribute, they “sense” internal quality directly. The aim is to evaluate all the external quality properties (attributes) by performing an aggregation along the edges of the graph. In the following we will refer to this graph as *Attribute Dependency Graph (ADG)*.

Let $G = (S \cup A, E)$ stand for the *ADG*, where S, A, and E denote the sensor nodes, aggregate nodes, and edges, respectively, and $S \cap A = \emptyset$. We want to measure how good or bad an attribute is. *Goodness* is the term that is used to express this measure of an attribute. For the sake of simplicity we will write goodness of a node instead of goodness of an attribute represented by a node. Goodness is measured on the $[0,1]$ interval for each node, where 0 and 1 mean the worst and best, respectively. The goodness of each sensor node u is not known precisely, hence it is represented by a random variable X_u with a probability density function $g_u : [0,1] \to \mathbb{R}$. g_u is called the *goodness function* of node u.

Constructing a goodness function. The currently presented way of constructing goodness functions is specific to source code metrics. For other sensor types, different approaches may be needed. The model makes use of the metric histogram over the source code elements, as it characterizes the whole system from the aspect of one metric. The aim is to give a measure for the goodness of a histogram. As the notion of goodness is relative, it is expected to be measured by means of comparison with other histograms in the benchmark. Let us suppose that H_1 and H_2 are the histograms of two systems for the same metric, and $h_1(t)$ and $h_2(t)$ are the corresponding normalized histograms (i. e., density functions, see Figure 3.3). By using Equation 3.3 we obtain a distance function (not in the mathematical sense) defined on the set of probability functions. Fig. 3.3 helps understanding the meaning of the formula: it computes the signed area between the two functions weighted by the function $\omega(t)$.

$$\mathscr{D}(h_1, h_2) = \int_{-\infty}^{\infty} (h_1(t) - h_2(t))\,\omega(t)\,dt \tag{3.3}$$

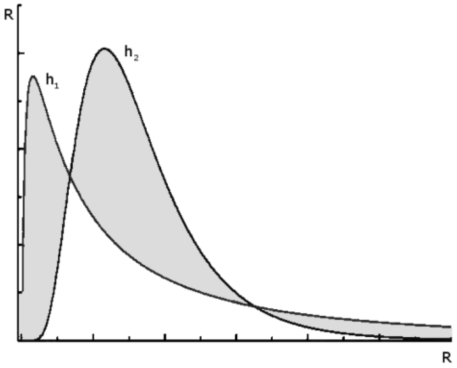


Fig. 3.3: Comparison of probability density functions

The weight-function plays a crucial role: it determines the notion of goodness, i. e., where on the horizontal axis the differences matter more. If one wants to express that all metric values matter in the same amount, she would set $\omega(t) = c$, where c is a constant, and in that case $\mathscr{D}(h_1, h_2)$ will be zero (as h_1 and h_2 integrate to 1). On the other hand, if one would like to express that higher metric values are worse, one could set $\omega(t) = t$. Non-linear functions for $\omega(t)$ are also possible. As in case of most source code metrics, higher values are considered to be worse (e. g., McCabe's complexity), we use the $\omega(t) = t$ weight function for these metrics (linearity is implicitly subsumed by the choice).

The choice leads to a very simple formula, given in Equation 3.4, where $H_1^{'}$ and $H_2^{'}$ are the random variables corresponding to the h_1 and h_2 density functions, $E\left(H_1^{'}\right)$ and $E\left(H_2^{'}\right)$ are the expected values of these (the equality is based on the definition of the expected value of a random variable). Lastly, $\tilde{H}_1$ and $\tilde{H}_2$ are the averages of the histograms H_1 and H_2, respectively. The last approximation is based on the Law of Large Numbers (the averages of a sample of a random variable converge to the expected value of the same). By this comparison we get one goodness value for the subject histogram (this value is relative to the other histogram).

$$\begin{aligned}\mathscr{D}(h_1,h_2) &= \int_{-\infty}^{\infty}(h_1(t)-h_2(t))\,t\,dt = \int_{-\infty}^{\infty}h_1(t)\,t\,dt - \int_{-\infty}^{\infty}h_2(t)\,t\,dt \\ &= E\left(H_1^{'}\right) - E\left(H_2^{'}\right) \approx \tilde{H}_1 - \tilde{H}_2\end{aligned} \tag{3.4}$$

In order to obtain a proper goodness function, this comparison needs to be repeated with histograms of many different systems independently. In each case we get a goodness value which can basically be regarded as sample of a random variable from the range $[-\infty,\infty]$. A linear transformation of the values changes the range to the $[0,1]$ interval. The transformed sample is considered to be the sample of the random variable X_u. Interpolation of the empirical density function leads to the goodness function of the sensor node.

There is a theoretical beauty of the approach. Let us assume that one disposes histograms of N different systems for one particular metric. Each histogram can be considered to be sampled by different random variables $Y_i, (i = 1,\ldots,N)$. Furthermore, one would like to assess the goodness of another histogram corresponding to the random variable X. The goodness is by definition described by the series of random variables in Equation 3.5. The random variable for goodness (before the transformation) is then described by random variable Z in Equation 3.6.

$$Z_1 := E(Y_1) - E(X), \ldots, Z_N := E(Y_N) - E(X). \tag{3.5}$$

$$Z := \frac{1}{N}\sum_{i=1}^{N} Z_i \rightarrow \Phi_{\nu,\sigma}, \text{ if } N \rightarrow \infty. \tag{3.6}$$

According to the Central Limit Theorem [266] for independent (not necessarily identically distributed) random variables, Z tends to a normal distribution which is independent of the benchmark histograms. This is naturally a theoretical result, and it states that when having a large number of systems in the benchmark, the constructed goodness functions are (almost) independent of the particular systems in the benchmark. Actually, $\Phi_{\nu,\sigma}$ is a benchmark-independent goodness function (on $[-\infty,\infty]$) for X, just that it can be approximated by having a benchmark with a sufficient number of systems.

To be able to perform the construction of goodness functions in practice, a source code metric repository database has also been built that consists of source code metrics of more than 100 open-source and industrial software systems.

Aggregation. After being able to construct goodness functions for sensor nodes, there is a need for a way to aggregate them along the edges of the ADG. Recall that the edges represent only dependencies, we have not yet assigned any weights to them. Assigning a simple weight would lead to the classic approach. In models that use a single weight or threshold in aggregation, the particular values are usually backed up with various reasonings and cause debates among experts. The Columbus model is able to handle this ambiguity. Many experts were asked in an online survey (both industrial and academic people) for their opinion about the weights. For every aggregate node, they were asked to assign scalars to incoming edges such that the sum of these would be 1. The number assigned to an edge is considered to be the amount of contribution of source goodness to target goodness. This way, for each aggregate node v a multi-dimensional random variable $\mathbf{Y}_v = \left(Y_v^1, Y_v^2, \ldots, Y_v^n\right)$ exists (n is the number of incoming edges). The components are dependent random variables, as

$$\sum_{i=1}^{n} Y_v^i = 1, \tag{3.7}$$

holds, that is, the range of $\mathbf{Y}_v$ is the standard $(n-1)$-simplex in $\mathbb{R}^n$. It is important that one cannot simply decompose $\mathbf{Y}_v$ to its components because of the existing dependencies among them.

Having an aggregate node with a composed random variable $\mathbf{Y}_v$ for aggregation ($\mathbf{f}_{\mathbf{Y}_v}$ will denote its composed density function), and also having n source nodes along the edges, with goodness functions $g_1, g_2, \ldots g_n$, the aggregated goodness for the aggregated node is defined by $g_v(t)$ in Equation 3.8 where Δ^{n-1} is the $(n-1)$-standard simplex in $\mathbb{R}^n$ and C^n is the standard unit n-cube in $\mathbb{R}^n$.

$$g_v(t) = \int_{\substack{t=\mathbf{qr} \\ \mathbf{q}=(q_1,\ldots,q_n)\,\in\,\Delta^{n-1} \\ \mathbf{r}=(r_1,\ldots,r_n)\,\in\,C^n}} \mathbf{f}_{\mathbf{Y}_v}(\mathbf{q})\, g_1(r_1)\ldots g_n(r_n)\, d\mathbf{r} d\mathbf{q}, \tag{3.8}$$

It is the generalization of how aggregation is performed in classic approaches. Classically, a linear combination of goodness values and weights is taken, and it is assigned to the target node. When dealing with probabilities, one needs to take every possible combination of goodness values and weights, and also the probabilities of their outcome into account. In the formula, the components of the vector $\mathbf{r}$ traverse the domains of source goodness functions independently, while vector $\mathbf{q}$ traverses the simplex where each point represents a probable vote for the weights. For fixed $\mathbf{r}$ and $\mathbf{q}$ vectors their scalar product ($t = \mathbf{qr} = \sum_{i=1}^{n} r_i q_i \in [0,1]$) is the goodness of the target node. To compute the probability for this particular goodness value, one needs to multiply the probabilities of goodness values of source nodes (these are independent) and also the composed probability of the vote ($\mathbf{f}_{\mathbf{Y}_v}(\mathbf{q})$). This product is integrated over all the possible $\mathbf{r}$ and $\mathbf{q}$ vectors (please note that t is not uniquely decomposed to vectors $\mathbf{r}$ and $\mathbf{q}$). $g_v(t)$ is indeed a probability distribution function on $[0,1]$ interval, i. e., its integral is equal to 1, because both $\mathbf{f}_{\mathbf{Y}_v}(\mathbf{q})$ and the goodness functions integrate to 1 on Δ^{n-1} and C^n respectively.

With this method it is now possible to compute goodness functions for every aggregate node. The way the aggregation is performed is mathematically correct, meaning that the goodness functions of aggregate nodes are really expressing the probabilities of their goodness (by combining other goodness functions with weight probabilities).

Although this approach provides goodness functions for every aggregate node, managers are usually only interested in having one number that represents an external quality attribute of the software. Goodness functions carry much more information than that, but an average of the function may satisfy even the managers. The resulting goodness function at every node has a meaning: it is the probability distribution which describes how good a system is from the aspect represented by the node. Therefore, the approach leads to *interpretable* results. Provided that the goodness functions are computed for every node of the *ADG*, and that the dependencies in the *ADG* are known, it is easy to see the root causes of the quality score.

Drill-down. Additionally to system level maintainability, ColumbusQM implements an algorithm [379] to drill down to lower levels in the source code and to get a similar measure for the building blocks of the codebase (e. g., classes or methods). For this, the model defines the **relative maintainability index** for the source code elements, which measures the extent to which they affect the system level goodness values. The approach is related to the aggregation and decomposition techniques introduced by Posnett et al. [699] and Serebrenik et al. [769].

The basic idea is to calculate the system level goodness values by ColumbusQM, leaving out the source code elements one by one. After a particular source code element is left out, the system level goodness values will change slightly for each node in the ADG. The difference between the original goodness value computed for the system, and the goodness value computed without the particular source code element, will be called the *relative maintainability index* of the source code element itself. The relative maintainability index is a small number that is either positive when it improves the overall rating or negative when it decreases the system level maintainability. The absolute value of the index measures the extent of the influence to the overall system level maintainability. A relative index can be computed for each node of the ADG, meaning that source code elements can affect various quality aspects in different ways and to different extents.

It is important to notice that this measure determines an ordering among the source code elements of the system, i. e., they become comparable to each other. And what is more, the system level maintainability being an absolute measure of maintainability, the relative index values become absolute measures of all the source code elements in the benchmark. Therefore, the ordering can be used by programmers to rank source code elements based on their criticality for improving the overall maintainability.

3.2.4.6 Other Approaches

The CAST (http://www.castsoftware.com) company has its own solution for software quality analysis, the Application Intelligence Platform (AIP), that uses an application quality benchmarking repository called Appmarq. Being a closed source proprietary tool, we were unable to try it out and evaluate it in detail as we did this with other solutions.

The Laboratory for Quality Software (LaQuSo – http://www.laquso.com) is a joint initiative of Eindhoven University of Technology and Radboud University Nijmegen. Since the starting of LaQuSo one of its focus areas has been the development of a product certification methodology. This has resulted in LSPCM (LaQuSo Software Product Certification Model). LaQuSo offers product certification as a service, which is a check that the artifact fulfills a well-defined set of requirements. These requirements are defined by the customer or a third party; LaQuSo as an independent evaluator will do the check. Serebrenik et al. [766] have analyzed requirements of three off-shoring projects using LSPCM. Application of LSPCM revealed severe flaws in one of the projects. The responsible project leader confirmed later that the development significantly exceeded time and budget. In the other projects no major flaws were detected by LSPCM and it was confirmed that the implementation was delivered within time and budget.

VizzMaintenance (http://arisa.se/products.php) is an Eclipse plug-in which brings detailed information about the maintainability of a software system. It uses static analysis to calculate 17 well-known software quality metrics. It then combines these values in a software quality model [928]. It supports the decisions which classes should be refactored first to improve their maintainability.

Vanderose et al. introduce a Model-Centric Quality Assessment (MoCQA) framework [593, 875, 877] which is a theoretical framework designed to help plan and support a focused quality assessment all along the software lifecycle. They aim at assessing other quality characteristics than maintainability, such as completeness [876], that are arguably useful to assist the software maintenance process.

There are other works that deal with software design quality and quality from the end user's point of view. For example, Ozkaya et al. [674] emphasize the importance of using quality models like ISO/IEC 9126 in practice right from the beginning of the design phase. The approach presented in their paper is general enough for evaluating design or end user quality, but not the product quality itself. Research of Bansiya and Davis [68] focus on the software design phase. They adapted the ISO/IEC 9126 model for supporting quality assessment of system design.

The work of Marinescu and Lanza [495] introduces a metrics-based approach for detecting design problems. It allows the software engineer to define metrics-based rules that "quantify" design principles, rules and heuristics related to the quality of a design. The work introduces an important suite of detection strategies for the identification of different well-known design flaws found in the literature. Additionally, the work presents a new type of quality model, called Factor-Strategy, allowing the quality to be expressed explicitly in terms of compliance with principles, rules and heuristics of good object-oriented design.

3.3 Application of Practical Quality Models in Software Evolution

It might be difficult to see the role of the presented maintainability models in software evolution at first glance. But whether one likes it or not, today, software industry is a giant business driven by business needs and profit. Thus keeping the costs of software evolution as low as possible is a central issue. As maintainability is in direct connection with the changing of software systems, measuring and controlling it is of vital importance for software evolution.

On the other hand, as applying techniques that improve the maintainability of the code or avoiding structures that deteriorate systems has an additional cost without having a short term financial benefit, they are often neglected by the business stakeholders. Hence maintainability of the systems is often overshadowed by feature developments whose business value is more evident at least in short terms. Although the developers are usually aware of its long term benefits, they do not have strong enough arguments to convince stakeholders for investing extra effort to improve maintainability. By better understanding the relationship of maintainability and the long term development costs, it would be possible to show the return on investment of keeping maintainability of systems at a high-level. It would make the extra investment more appealing to the business stakeholders as well, thus reaching higher quality software and cheaper evolution in general.

In this section we introduce a cost model that is based on source code maintainability and proves its direct connection with development costs. It is a possible application of practical quality models during software evolution in modern industrial environments.

3.3.1 A Cost Model Based on Software Maintainability

The approach [62] adopts the concept of entropy in thermodynamics, which is used to measure the disorder of a system. According to the second law of thermodynamics, the entropy of a closed system cannot be reduced; it can only remain unchanged or increase. The only way to decrease entropy (disorder) of a system is to apply external forces, i.e. to put energy into making order.

The notion of entropy is applied in a very similar way for software systems [432]. Maintainability of a source code is usually defined as a measure of the effort required to perform specific modifications in it. Assuming that the higher the disorder is, the more effort is needed to perform the modifications, maintainability can be interpreted as a measure of the disorder, i.e. entropy of the source code.

The approach lays on two basic assumptions:

1. Making changes in a source code does not decrease the disorder of it, provided that one does not work actively against this. In other words, when making

changes to a software system without explicitly aiming to improve it, its maintainability will decrease, or at least it will remain unchanged.

2. The amount of changes applied to the source code is proportional to the effort invested, and to the maintainability of the code. In other words, if one applies more effort, the code will change faster. Additionally, a more maintainable code will change faster, even if the applied effort is the same. Another interpretation is that the effort aiming on code change is inversely proportional to the maintainability at time t.

Before formalizing these assumptions, the following notions are introduced:

- $\mathscr{S}(t)$ - the size of the source code at time t, measured in lines of code.
- $\lambda(t)$ - the change rate of the source code at time t, i.e. the probability of changing any line independently (for the sake of simplicity we assume that it is the same for all lines of code). $\mathscr{S}(t)\lambda(t)$ equals the number of lines changed at time t.
- k - a constant for the conversion between different units of measure. The approach deals with two scalar measures: maintainability and cost. Instead of fix particular units of measure for each, a conversion constant k is introduced. In the sequel, it can be assumed without the loss of generality, that cost is expressed by any measure of effort, e.g. salary, person month, time, etc., while maintainability may have any other scalar measure. In practice, after fixing the measures of unit for each, k can be estimated from historical project data.
- $\mathscr{C}(t)$ - the cost invested into changing the system until time t, measured from an initial time $t = 0$. Obviously, $\mathscr{C}(0) = 0$.
- $\mathscr{M}(t)$ - maintainability (i.e. disorder) of the system at time t.

In the following, it is assumed that modifications do not explicitly aim on code improvement, meaning that only new functionality is being added to the system and no refactoring or other explicit improvements are done. In this case, the first assumption above can be formalized as in Equation 3.9, meaning that the decrease rate of maintainability is proportional to the number of lines changed at time t. The constant factor q is called the *erosion factor* which represents the amount of "damage" (decrease in maintainability) caused by changing one line of code.

$$\frac{d\mathscr{M}(t)}{dt} = -q\mathscr{S}(t)\lambda(t) \qquad (q \geq 0), \tag{3.9}$$

The erosion factor depends on many internal and external factors like the experience and knowledge of the developers, maturity of development processes, quality assurance processes used, tools and development environments, the programming language, and the application domain. The $q \geq 0$ assumption makes it impossible for the code to improve by itself just by adding new functionality. The assumption is in accordance with Lehman's laws [511] of software evolution, which state that the complexity of evolving software is increasing, while its quality is decreasing at the same time.

Formalizing the second assumption leads to Equation 3.10. The numerator represents the amount of change introduced at time t. The formula states that the utilization of the cost invested at time t for changing the code is inversely proportional to maintainability.

$$\frac{d\mathscr{C}(t)}{dt} = k\frac{\mathscr{S}(t)\lambda(t)}{\mathscr{M}(t)} \tag{3.10}$$

Solving the above system of ordinary differential equations, yields the following result:

$$\begin{aligned}\mathscr{C}(t_1) - \mathscr{C}(t_0) &= \int_{t_0}^{t_1} k\frac{\mathscr{S}(t)\lambda(t)}{\mathscr{M}(t)}dt = -\frac{k}{q}\int_{t_0}^{t_1} \frac{\dot{\mathscr{M}}(t)}{\mathscr{M}(t)}dt = \\ &= -\frac{k}{q}\left[\ln\mathscr{M}(t_1) - \ln\mathscr{M}(t_0)\right] = -\frac{k}{q}\ln\frac{\mathscr{M}(t_1)}{\mathscr{M}(t_0)}.\end{aligned} \tag{3.11}$$

By expressing $\mathscr{M}(t)$ from the above equation, we get to the main result:

$$\mathscr{M}(t_1) = \mathscr{M}(t_0)\, e^{-\frac{q}{k}(\mathscr{C}(t_1)-\mathscr{C}(t_0))}, \tag{3.12}$$

which suggests that the maintainability of a system decreases exponentially with the invested cost to change the system. The erosion factor q determines the decrease rate of maintainability. It is obvious that for a higher erosion factor the decrease rate will be higher as well. It is crucial for software development companies to push the erosion factor as low as possible, for instance by training the employees, improving processes, utilizing sophisticated quality assurance technologies.

Although, the formula does not provide a way of having an absolute measure for maintainability, one can easily define a *relative maintainability* for the system. Indeed, by letting $t_0 = 0$, and defining $\mathscr{M}(0) = 1$, we get to the following function for maintainability:

$$\mathscr{M}(t) = e^{-\frac{q}{k}\mathscr{C}(t)} \tag{3.13}$$

For the interpretation, let us consider two artificial scenarios. Figure 3.4 shows the case, when the invested effort is constant over the time. In this case, both the maintainability $\mathscr{M}(t)$ and the change rate $\lambda(t)$ decrease exponentially.

In the other case, let us suppose that one intentionally wants to keep the change rate of the system constant. Figure 3.5 shows how the maintainability $\mathscr{M}(t)$ and the overall cost $\mathscr{C}(t)$ change over time. Now, the maintainability decreases linearly until it reaches zero, while the cost is increasing faster than an exponential rate. The cost will reach infinity in finite time, exactly when maintainability reaches zero, meaning that any further change would require infinite amount of effort. This is, of course, just a theoretical possibility, as no one disposes an infinite amount of resources required to degrade the maintainability of a system to absolute zero.

The problem with applying the model to real-world software systems lies in the erosion factor q. While the other model parameters (k and $\mathscr{C}(t)$) can be computed easily, the erosion factor, which measures the "damage" caused by changing one

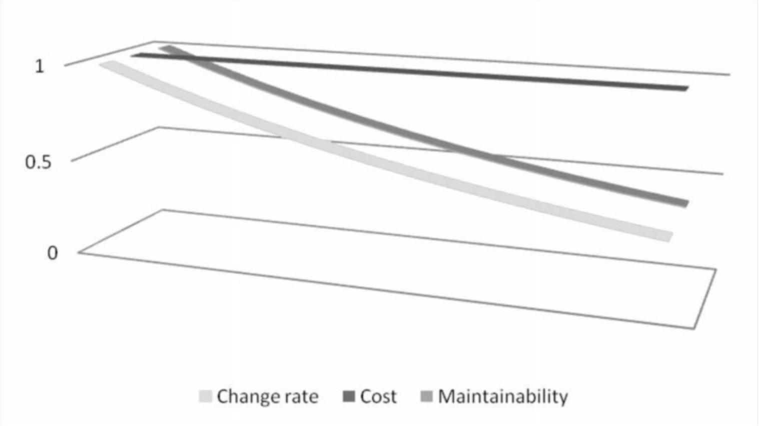


Fig. 3.4: Changes of *Change rate* ($\lambda(t)$) and *Maintainability* ($\mathscr{M}(t)$) during evolution when the cost of the development ($\mathscr{C}(t)$) is constant over time.

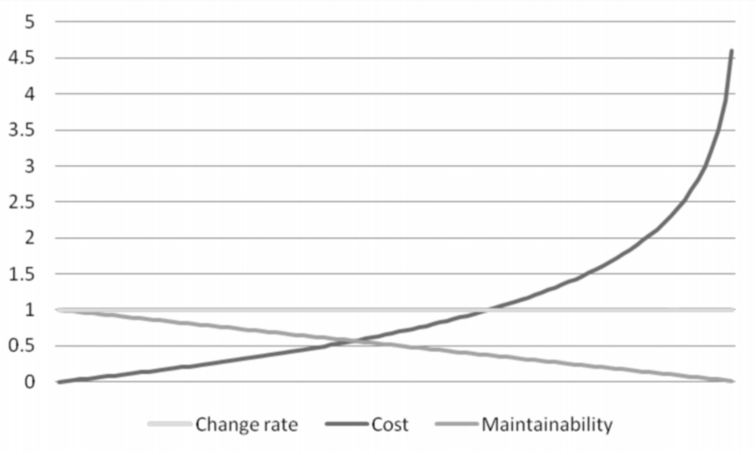


Fig. 3.5: Changes of *Cost* ($\mathscr{C}(t)$) and *Maintainability* ($\mathscr{M}(t)$) during evolution when the *Change rate* ($\lambda(t)$) is constant over time.

line, is challenging. Contrarily, if there was an absolute measure of maintainability, the constant, project-specific erosion factor q could easily be computed by expressing it from Equation 3.13. Furthermore, by having an absolute measure for q as well, the erosion factors of different projects, organizations could be compared. The analysis of the causes of the differences would make it possible to lower the erosion factor, e. g., by improving the processes, and training people.

In addition, the overall cost of development could also be expressed explicitly from the model, according to Equation 3.14. For computing future development costs, it would just be required to have an estimate for the change rate $\lambda(t)$ over a time period.

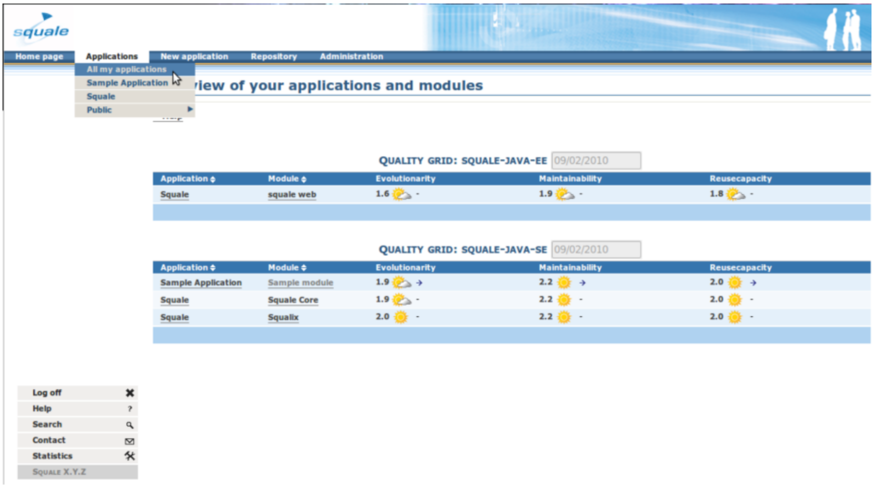
$$\mathscr{C}(t) = -\frac{k}{q} \ln \left| 1 - \frac{q}{\mathscr{M}(0)} \int_0^t \mathscr{S}(s)\,\lambda(s)\,ds \right|. \tag{3.14}$$

The introduced practical quality models are good candidates for obtaining an absolute measure of maintainability for software systems. Using the absolute maintainability calculated by one of these models would allow to obtain an absolute erosion factor q, which can be used to estimate further development costs and to compare the erosion factors of different projects and organizations.

3.4 Tools Supporting Software Quality Estimation

3.4.1 Software QUALity Enhancement project (SQUALE)

The implementation of the SQUALE model (see Section 3.2.4.1) is available as an open-source tool (`http://www.squale.org`). The project officially started in June 2008, funded by the French Government. The first official open-source version was released in January 2009.



view of your applications and modules

QUALITY GRID: SQUALE-JAVA-EE 09/02/2010

Application ⇕	Module ⇕	Evolutionarity	Maintainability	Reusecapacity
Squale	squale web	1.6 ⛅ -	1.9 ⛅ -	1.8 ⛅ -

QUALITY GRID: SQUALE-JAVA-SE 09/02/2010

Application ⇕	Module ⇕	Evolutionarity	Maintainability	Reusecapacity
Sample Application	Sample module	1.9 ⛅ →	2.2 ☀ →	2.0 ☀ →
Squale	Squale Core	1.9 ⛅ -	2.2 ☀ -	2.0 ☀ -
Squale	Squalix	2.0 ☀ -	2.2 ☀ -	2.0 ☀ -

Fig. 3.6: SQUALE tool

The Software QUALity Enhancement project – SQUALE focused on two main aspects. First, it works on enhanced quality models inspired by existing approaches

(GQM [874], McCall et al. [574]) and standards (ISO/IEC 9126 [422]), validated and improved by researchers, dealing with both technical and economical aspects of quality. Second, the development of an open-source application that helps assessing software quality and improving it over time based on third party technologies (commercial or open-source) that produce raw quality information (like metrics), using the quality models to aggregate this raw information into high-level quality factors, all this targeting different languages.

The tool provides a web-based interface for configuring the qualifications of new applications. The qualification process is run as the part of a scheduled audit of the source code. The quality results are presented in the same web application. Figure 3.6 shows the overview page of a quality audit result of SQUALE.

3.4.2 Software Quality Assessment based on Lifecycle Expectations (SQALE)

According to the official site (`http://www.sqale.org/tools`), the following tools implement the SQALE model (see Section 3.2.4.2):

- Insite SaaS by Metrixware (`http://www.metrixware.com`)
- Sonar by SonarSource (`http://www.sonarsource.com`)
- SQuORE by SQuORING (`http://www.squoring.com`)
- Mia-Quality by Mia-Software (`http://www.mia-software.com`)

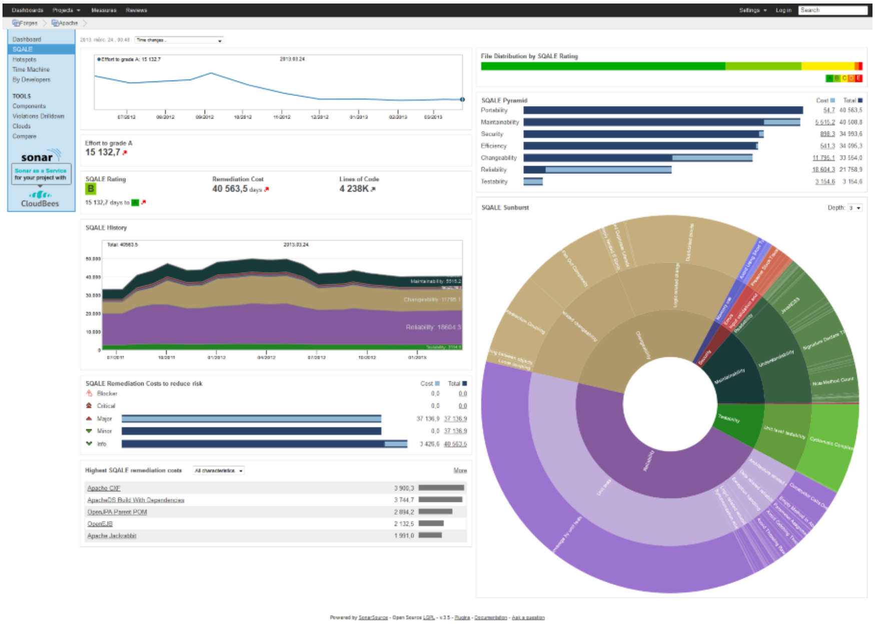
The results of the tool evaluation in the following section refers to the Sonar implementation of the model. Sonar is an open platform to manage code quality (`http://www.sonarsource.org`). Using an extensive plug-in mechanism it is fairly easy to extend the basic functionality of the framework (e. g., support analysis for new languages, add new metrics).

The Technical Debt Evaluation (SQALE) Sonar plug-in is a full implementation of the SQALE methodology. This method contains both a Quality Model and an Analysis Model. The Technical Debt Evaluation (SQALE) plug-in comes with a number of features, including custom widgets, visualizations, rules and drill-downs. Figure 3.7 shows the summary page of the tool.

3.4.3 QUAMOCO Quality Model

The QUAMOCO framework (see Section 3.2.4.3) is available as an open-source Eclipse extension (`https://quamoco.in.tum.de`). The Quamoco Consortium provides a toolchain [233] for the creation/editing of quality models and for the automatic analysis of software products:

- Quality Model Editor: This editor enables the comfortable creation of quality models.



Effort to grade A
15 132,7

SQALE Rating	Remediation Cost	Lines of Code
B 15 132,7 days to A	40 563,5 days	4 238K

SQALE Pyramid	Cost	Total
Portability	54,7	40 563,5
Maintainability	5 515,2	40 508,8
Security	898,3	34 993,6
Efficiency	541,3	34 095,3
Changeability	11 795,1	33 554,0
Reliability	18 604,3	21 758,9
Testability	3 154,6	3 154,6

SQALE Remediation Costs to reduce risk	Cost	Total
Blocker	0,0	0,0
Critical	0,0	0,0
Major	37 136,9	37 136,9
Minor	0,0	37 136,9
Info	3 426,6	40 563,5

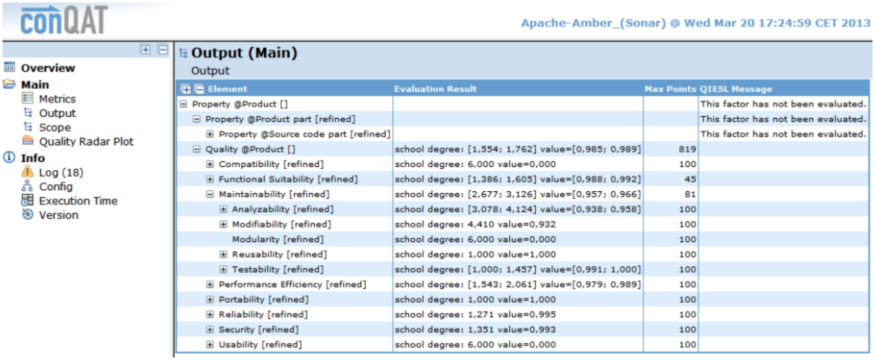
Highest SQALE remediation costs All characteristics More

Apache CXF	3 900,3
ApacheDS Build With Dependencies	3 744,7
OpenJPA Parent POM	2 894,2
OpenEJB	2 132,5
Apache Jackrabbit	1 991,0

Powered by SonarSource - Open Source LGPL - v.3.5 - Plugins - Documentation - Ask a question

Fig. 3.7: Sonar SQALE Maintainability Model plug-in

- ConQAT-Integration: By integrating the quality model into the analysis framework ConQAT [234], automatic quality assessments for the programming languages Java, C#, and C/C++ can be conducted.



Apache-Amber_(Sonar) @ Wed Mar 20 17:24:59 CET 2013

Output (Main)
Output

Element	Evaluation Result	Max Points	QIESL Message
Property @Product []			This factor has not been evaluated.
Property @Product part [refined]			This factor has not been evaluated.
Property @Source code part [refined]			This factor has not been evaluated.
Quality @Product []	school degree: [1,554; 1,762] value=[0,985; 0,989]	819	
Compatibility [refined]	school degree: 6,000 value=0,000	100	
Functional Suitability [refined]	school degree: [1,386; 1,605] value=[0,988; 0,992]	45	
Maintainability [refined]	school degree: [2,677; 3,126] value=[0,957; 0,966]	81	
Analyzability [refined]	school degree: [3,078; 4,124] value=[0,938; 0,958]	100	
Modifiability [refined]	school degree: 4,410 value=0,932	100	
Modularity [refined]	school degree: 6,000 value=0,000	100	
Reusability [refined]	school degree: 1,000 value=1,000	100	
Testability [refined]	school degree: [1,000; 1,457] value=[0,991; 1,000]	100	
Performance Efficiency [refined]	school degree: [1,543; 2,061] value=[0,979; 0,989]	100	
Portability [refined]	school degree: 1,000 value=1,000	100	
Reliability [refined]	school degree: 1,271 value=0,995	100	
Security [refined]	school degree: 1,351 value=0,993	100	
Usability [refined]	school degree: 6,000 value=0,000	100	

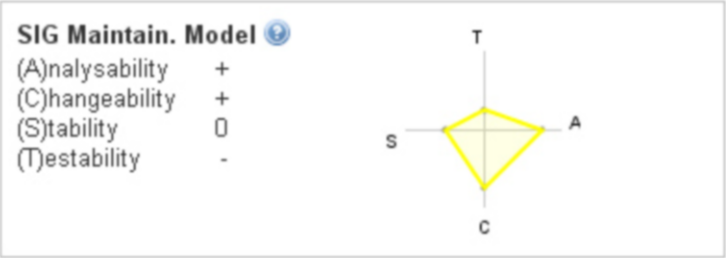
Fig. 3.8: QUAMOCO quality report

The quality analysis with the prepared quality models can be started interactively from Eclipse or run from command line allowing to be integrated into the build processes. The tool presents its results in Eclipse and also creates a detailed HTML quality report (see Figure 3.8).

3.4.4 SIG Maintainability Model

The Software Improvement Group (`http://www.sig.eu`) offers software product certification based on the implementation of their maintainability model (see Section 3.2.4.4) as a commercial service. No official downloadable version of the tool exists on their homepage.

However, the SIG Maintainability Model is implemented as a free and downloadable Sonar plug-in. The results of the tool evaluation in the following section refers to this Sonar plug-in implementation of the model. The SIG plug-in provides a high-level overview about the following ISO/IEC 9126 maintainability subcharacteristics: Analysability, Changeability, Stability and Testability. The values range from −− (very bad) to ++ (very good). Figure 3.9 shows a screenshot about the results of the plug-in.



SIG Maintain. Model

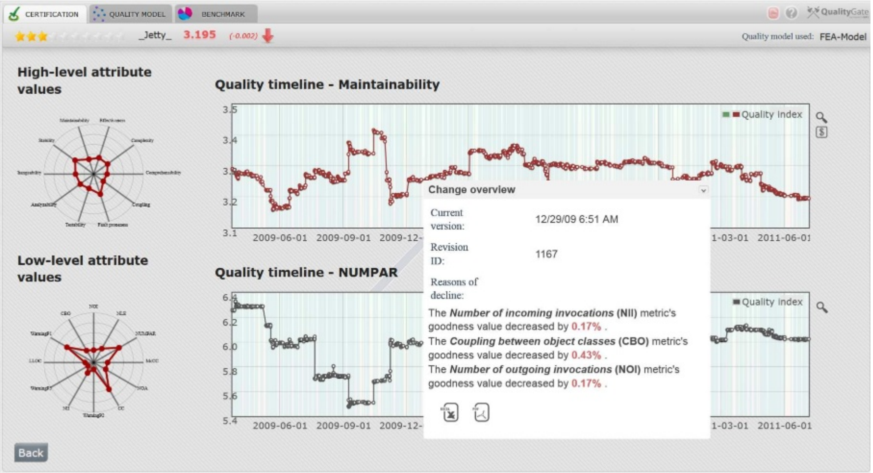
(A)nalysability	+
(C)hangeability	+
(S)tability	0
(T)estability	-

Fig. 3.9: Sonar SIG Maintainability Model Plug-in

3.4.5 Columbus Quality Model

The Columbus Quality Model (see Section 3.2.4.5) is implemented by a proprietary tool called SourceAudit, member of the QualityGate product family. The QualityGate source code quality assurance platform developed by FrontEndART (`http://www.frontendart.com`) is based on research conducted at the Department of Software Engineering of University of Szeged and on the ISO/IEC 9126 standard.

The tool is able to continuously monitor the maintainability of software products. It can be integrated into the common build processes or manage individual



Change overview

Current version: 12/29/09 6:51 AM

Revision ID: 1167

Reasons of decline:

The *Number of incoming invocations* (NII) metric's goodness value decreased by 0.17% .

The *Coupling between object classes* (CBO) metric's goodness value decreased by 0.43% .

The *Number of outgoing invocations* (NOI) metric's goodness value decreased by 0.17% .

Fig. 3.10: QualityGate implementation of the Columbus Quality Model

qualifications with the help of a Jenkins continuous integration system[1] based administration page. The results of the qualification is presented in a sophisticated web application but many types of different reports can also be generated. Figure 3.10 shows a screenshot of the tool containing a one year long period of quality analysis results.

3.5 Comparing the Features of the Quality Models and Tools

To evaluate and compare the different models and their implementing tools, we installed and ran them on several projects. As two of the tools were available as Sonar plugins, we decided to perform a maintainability assessment on the open-source projects presented in Sonar's Nemo demo application.[2] The benefit of it was twofold: the data in Nemo already contained the quality analysis results of the SQALE model commercial plug-in; and we could easily identify the exact source code locations and versions from Sonar to be able to run the other tools on the same source code.

As the SIG model is not part of Nemo, we also installed and configured our own local version of Sonar. Besides the SQALE and SIG models we decided to include the Sonar Quality Index plug-in[3] in the evaluation as well. It is a Maintainability Index style combination of different metrics and not a hierarchical quality model. Nonetheless, we were interested in the relation of QI to other sophisticated models.

[1] `http://jenkins-ci.org/`

[2] `http://nemo.sonarsource.org/`

[3] `http://docs.codehaus.org/display/SONAR/Quality+Index+Plugin`

Altogether 97 open-source Java projects have been analyzed with six tools. Although Nemo contains almost 200 systems, 50 of them do not have any version control information, therefore we could not locate their source. For around another 50 projects the version control information was changed since the Sonar analysis, so we also left them out from the experiment. Except SQUALE, all the analyses have been run in an automated way with default models and configurations. In the case of SQUALE, we found no way of automating the qualification process, therefore all the projects have been configured and analyzed manually through its web interface. When a qualification analysis failed, we tried to manually fix the causing problem and re-run the analysis. If a more complex error occurred – which we could not fix easily – we marked the analysis as failed. The typical causes of errors are listed in Section 3.5.2.

3.5.1 Comparing the Properties of Different Practical Models

The comparison of the models was done using the following evaluation criteria:

1. *Interpretability* – applying the model should provide information for high-level quality characteristics which is meaningful, i. e., conclusions can be drawn.
2. *Explicability* – there should be a way to efficiently evaluate the root causes, i. e., a simple way to decompose information obtained for high-level characteristics to attributes or even to properties.
3. *Consistency* – the information obtained for higher level characteristics should not contradict lower level information.
4. *Scalability* – the model should provide valuable information even for large systems in reasonable time.
5. *Extendibility* – there should be an easy way to extend the model with new characteristics and its attributes.
6. *Reproducibility* – applying the model on the same system twice should result in the same information.
7. *Comparability* – information obtained for quality characteristics of two different systems should be comparable and should correlate with the intuitive meaning of the characteristics.
8. *Aggregation type* – the way of acquiring quality values for high-level characteristics based on low-level values. The possible values are:
   - Linear combination (LC) – a simple linear combination of the values.
   - General function (GF) – combination of the values with an arbitrary (not necessarily linear) function.
   - Fixed threshold (FT) – the values are categorized based on fixed thresholds.
   - Benchmark-based threshold (BT) – the values are categorized based on thresholds derived from a benchmark.
   - Benchmark based (B) – the aggregation is done in some sophisticated way based on a repository of other systems (benchmark).

9. *Input measures* – what type of source code measures are considered in the model. The possible values are:
    - Metrics (M)
    - Rule violations (R)
    - Code clones (C)
    - Unit tests (T)
10. *Base model* – which theoretical model serves as the base concept of the practical model.
11. *Rating* – what kind of qualification or rating does the model provide for expressing the level of maintainability. The possible values are:
    - Ordinal – discrete quality categories (like 1 to 5 stars, etc.)
    - Scale – a continuous value from an interval (e. g., a real number between 0 and 10)

Table 3.6 presents the summary of the model evaluations against the criteria above. We can note that the most popular base model is the one defined in the ISO/IEC 9126 standard. Despite the fact that it already has a successor – ISO/IEC 25000 – only one model supports it in some extent. Probably most of the models will adapt to the new standard in the future.

Table 3.6: The properties of the different practical quality models

	SQALE	**ColumbusQM**	**SIG**	**QI**	**SQUALE**	**QUAMOCO**
Interpretable	✓	✓	✓	✓	✓	✓
Explicable	✓	✓	–[4]	–	✓	✓
Consistent	✓	✓	✓	✓	✓	✓
Scalable	N/A[5]	✓	✓	✓	✓[6]	✓
Extendible	–	✓	–	–	–	✓
Reproducible	✓	✓	✓	✓	✓	✓
Comparable	✓	✓	✓	✓	✓	✓
Aggregation type	FT	B	BT	LC	FT+GF	FT
Input measures	M, R	M, R, C	M, C, T	M, R, C, T	M, R	M, R
Base model	ISO 9126	ISO 9126	ISO 9126		McCall, ISO 9126	partly ISO 9126, ISO 25000
Rating	Ordinal A, B, C, D, E	Scale [0..10]	Ordinal [-2..2]	Scale [0..10]	Scale [0..3]	Scale [1..6]

Regarding the rating of the models, the scale type appears to be in majority which is able to express the maintainability in a more precise, continuous way. Another

[4] Refers to the Sonar plug-in which does not allow to drill-down the qualifications

[5] SQALE qualifications were already available in Sonar Nemo

[6] We found performance issues with the default embedded database, but we have not tried with other suggested database servers

advantage of the scale type ratings is that it is easy to convert the rating of one model to the rating of the other. On the contrary, ordinal ratings are harder to convert due to the different number of rates.

One would expect that a model uses all the possible static source code information: metrics, rule violations, code clones and unit tests as its input measures. However, only the Quality Index seams to use all this information. Metrics are considered by all the examined models and rule violations are taken into account by all models except SIG.

The models vary in the way they aggregate the source code measures. The most common approach is to use a fixed threshold to categorize metric values. However, a constant improvement is shown in this area by introducing complex aggregation formulas and deriving dynamic thresholds based on a benchmark. The ColumbusQM uses the benchmark in even a more sophisticated way to aggregate quality properties.

Almost all of our requirements are met by the examined models. Most of the models failed to fulfill the Extendibility requirements as they provide no easy way to extend the base model. Another requirement that two models could not met is Explicability. The results of the models that do not fulfil this requirement is hard to be traced back to the root causes in the source code.

3.5.2 Evaluating the Properties of the Different Tools

The evaluation of the tools was made according to the following aspects:

1. *Supported languages* – the languages supported by the tool (or it is language independent).
2. *Stability* – the number of projects successfully analyzed from all projects (in total 97 projects have been analyzed).
3. *Input type* – the input of the tool i. e., requires only sources or binaries too.
4. *Type* – the type of the application (e. g., a plug-in to an existing framework, a web application).
5. *Supported build processes* – the type of common build frameworks into which the qualification can be integrated.
6. *OS platform* – the supported OS platforms.
7. *Proprietary* – is the evaluated tool free or proprietary?
8. *Presentation of the results* – the way of the presentation of qualification results (e. g., in a web application, HTML)

Table 3.7 presents the summary of the evaluation of the tools against the aspects above. The stability line needs some further explanation. In case of SQALE all the projects were successfully analyzed because it was already in the Sonar Nemo system. The other tool that was able to parse all the systems is QualityGate SourceAudit, because it is able to analyze projects without having to compile the code. In the case of the two Sonar plugins, the SIG model and Quality Index, the cause of

Table 3.7: The properties of the different evaluated tools

	SQALE	**QualityGate SourceAudit**	**SIG**
Supported languages	Lang. independent	Lang. independent	Lang. independent
Stability	100% (97/97)	100% (97/97)	77% (75/97)
Input type	Sources, binaries are optional	Sources only	Sources, binaries are optional
Type	Sonar plug-in	Web application and web service	Sonar plug-in
Supported build processes	ant, maven, batch	ant, maven, batch	ant, maven, batch
OS platform	Windows & Linux	Windows & Linux	Windows & Linux
Proprietary	Yes	Yes	Yes (free Sonar plugin)
Presentation of the results	Web application	Web application, Excel, PDF reports	Web application

	QI	**SQUALE**	**QUAMOCO**
Supported languages	Java	Java	Java, C#, and C/C++
Stability	77% (75/97)	31% (30/97)	63% (61/97)
Input type	Sources, binaries are optional	Sources and binaries	Sources and binaries
Type	Sonar plug-in	Web application	Eclipse plug-in
Supported build processes	ant, maven, batch	ant	batch
OS platform	Windows & Linux	Windows & Linux	Windows & Linux
Proprietary	No	No	No
Presentation of the results	Web application	Web application, PDF reports	Eclipse GUI, HTML report

unsuccessful qualification was that some of the projects could not be compiled and not the failure of the models. As we used the maven wrapper to upload the results into Sonar, it caused the failure of the qualification also. The other two tools, QUAMOCO and SQUALE are also affected by the compilation errors as they require the binaries for the qualification. Additionally to the build errors, QUAMOCO failed with a non-trivial parser error for about 10 projects. The most unstable tool was SQUALE according to our experiences; however, it must be noted that we used the program with default settings only.

To summarize, most tools were able to analyze the majority of the projects with minimal invested effort. Therefore, they can be a great help both for managers and developers in software evolution activities.

3.6 Conclusions

For software developers and managers alike it is crucial to be able to measure different aspects of the quality of their systems. The information can mainly be used for making decisions, backing up intuition, estimating future costs and assessing risks during software evolution.

There are three main approaches for measuring software quality: process-based, product-based and hybrid. This chapter focused on the history, evolution, state-of-the-art and supporting tools of the product based software quality assessment. The introduction of the ISO/IEC 9126 standard as the joint model of the early theoretical software product quality models caused an explosion in the number of new practical quality models. All these models adapt the standard and use a hierarchical model for estimating quality with some kind of metrics at the lowest level. Section 3.2.4 gives an overview about the evolution of software quality measurements and approaches starting from the first software metrics through simple metrics-based prediction models and early theoretical quality models to focus on the currently available state-of-the-art approaches for software product qualification.

Each of the tools implementing these models have been evaluated on almost 100 open-source Java systems. The tools and underlying quality models were compared according to a set of predefined criteria. Most tools were able to analyze the majority of the projects with minimal invested effort. Therefore, we conclude that they can be a great help both for managers and developers in software evolution activities. However, we note that the correctness of the models has not been evaluated. In the end irrespective of how easy it is to use them or what features they have, we expect that models that are more accurate will be more frequently used. However, comparing the correctness of the existing models requires a huge effort that should be addressed by the joint work of the community.

It is also a very interesting open question if the state-of-the-art practical models can be unified and merged into a common standard, like it was done with the early theoretical models. To be able to assess this possibility, a very deep analysis of model results would be needed. It should be examined how well the results of the current practical models correlate with each other. Our vision is that these practical models can be merged into a common standard in the future which will lead to a more exact and objective product quality assessment.

Part II
Techniques

Chapter 4
Search Based Software Maintenance: Methods and Tools

Gabriele Bavota, Massimiliano Di Penta and Rocco Oliveto

Summary. Software evolution is an effort-prone activity, and requires developers to make complex and difficult decisions. This entails the development of automated approaches to support various software evolution-related tasks, for example aimed at suggesting refactoring or remodularization actions. Finding a solution to these problems is intrinsically NP-hard, and exhaustive approaches are not viable due to the size and complexity of many software projects. Therefore, during recent years, several software-evolution problems have been formulated as optimization problems, and resolved with meta-heuristics.

This chapter overviews how search-based optimization techniques can support software engineers in a number of software evolution tasks. For each task, we illustrate how the problem can be encoded as a search-based optimization problem, and how meta-heuristics can be used to solve it. Where possible, we refer to some tools that can be used to deal with such tasks.

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_4, © Springer-Verlag Berlin Heidelberg 2014

4.1 Introduction

Software evolution activities require developers to take complex decisions, often making choices among several possible solutions. For example, let us consider a scenario where, because of maintenance and evolution tasks, the system architecture is deteriorated, resulting in lowly-cohesive and strongly-coupled modules. To mitigate such a problem, developers can reorganize the system decomposition. However, even for a relatively small system, the number of possible choices to improve the architecture can be very high, and such a number exponentially increases with the system size. Similar considerations apply to other evolution-related activities, such as selecting and performing a refactoring, or fixing a bug. In general, as other activities like testing, software evolution requires software engineers to solve problems for which finding a solution is NP-hard [312].

For such reasons, the use of search-based optimization techniques can be a very promising and effective way to deal with many software evolution activities. All that is required is to provide:

- a problem *representation*, i.e., to encode the activity through an appropriate data structure allowing the heuristics to (i) evaluate the quality of a possible problem solution, and (ii) produce new solutions;
- a way to quantitatively evaluate the quality of a given solution, often referred as *fitness* or *objective* function; and
- a set of *operators* to produce new solutions starting from existing ones.

The idea of solving software engineering problems using search-based optimization techniques has been named "Search-Based Software Engineering" [190]. The potentials and challenges of applying search-based optimization techniques to various kinds of software engineering problems have been largely discussed by Harman [365]. Also, Harman has outlined how SBSE can support various software maintenance [364] and program comprehension [366] tasks. Among others, there are two aspects that make the application of SBSE to software evolution special. First, many evolution decisions imply balancing across conflicting objectives. To this aim, it can be desirable to use multi-objective optimization, which instead of producing solutions (near) optimizing a single objective, produce a set of solutions—that, as will be explained in Section 4.2—are "Pareto-optimal" i.e., there is no solution among the found ones that is better than others with respect to all objectives. Second, many software evolution activities are highly human intensive, i.e., (i) automatic approaches must be able to account for developers' rationale, and (ii) it is hardly possible to find a completely automatic way to evaluate the quality of a solution. To this aim, it is necessary to find ways to encode rationale in the meta-heuristic fitness functions, as well as to use "interactive" optimization techniques [818] for which the fitness function is (partially) evaluated by humans.

This chapter describes the main achievements of SBSE techniques in the field of software maintenance and evolution. Specifically, we describe work related to:

- *(Re)modularization approaches*, i.e., approaches aimed at identifying and creating modules that achieve certain properties, such as high cohesion and low coupling, or aimed at reducing the application footprint.
- *Software analysis and transformation approaches*, aimed at automatically modifying source code for various specific purposes, for example to improve maintainability or fixing bugs.
- *Refactoring approaches*, aimed at automatically suggesting and applying refactoring activities, e.g., those proposed by Fowler [301]. This is a special case of code transformation, which does not alter the semantic of the source code, but improves its maintainability. Given the amount of work in this area, we discuss it in a separate section rather than together with other transformation approaches.

It is important to note that this chapter is not a systematic literature review on search-based software maintenance and evolution. There are also other pieces of work related to the use of optimization techniques in the area of software evolution. Due to space limitation, it is not possible to describe all of them. We chose to describe the aforementioned dimension because these are the one that have been investigated the most in past and recent years, also according to the SEBASE repository[1], which collects a large set of references for SBSE papers.

Instead, the chapter describes the available SBSE techniques to be used for various kinds of problems, explaining how the problem needs to be represented, how solutions can be evaluated through a fitness function, which are the operators to be used, and the meta-heuristics that work better. Also, wherever possible, the chapter points out available tools for each specific problem. In summary, the chapter aims to be a guideline for practitioners interested to apply SBSE techniques in the context of software evolution, as well as for PhD students interested to work on such a research topics, and for instructors that need to introduce such techniques in the context of software engineering or software evolution courses, especially in graduate curricula.

The chapter is organized as follows. Section 4.2 provides basic background notions about the optimization techniques used to solve software evolution problems. Section 4.3, Section 4.4, and Section 4.5 describe search-based approaches for software modularization, source code transformation, and refactoring, respectively. Section 4.6 concludes the chapter.

4.2 An Overview of Search-Based Optimization Techniques

This section provides some background on search-based optimization techniques that have been used to solve the various software maintenance problems described in this chapter. Further details can be found in the books by Goldberg [337] and by Michalewicz and Fogel [607].

For the techniques described below, the problem is encoded in a representation (named *chromosome*), an instance (solution) of which (named *genotype*) represents

[1] http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/

an instance of the real world problem to be solved (referred to as *phenotype*). The quality of a solution (i.e., of a *genotype*) is evaluated by means of a *fitness function*.

2 Pseudo-code of iterated Hill Climbing (from [607]).

```
 1: $t \leftarrow 0$
 2: initialize $best$
 3: while $t < MAX$ do
 4:     $local \leftarrow FALSE$
 5:     select a current point $v_c$ at random
 6:     evaluate $v_c$
 7:     while not local do
 8:         select all new points in the neighborhood of $v_c$
 9:         select the point $v_n$ from the new points with the best value of evaluation function $eval$
10:         if $eval(v_n)$ is better than $eval(v_c)$ then
11:             $v_c \leftarrow v_n$
12:         else
13:             $local \leftarrow TRUE$
14:         end if
15:     end while
16:     $t \leftarrow t+1$
17:     if $v_c$ is better than $best$ then
18:         $best \leftarrow v_c$
19:     end if
20: end while
```

4.2.1 Hill Climbing

Hill Climbing (HC)—see Algorithm 2—is a “local” search method, where the search proceeds from a randomly chosen point (solution) v_c in the search space (line 5) by considering the neighbors of the point. Different families of HC exist based on how neighbors are explored. For example, stochastic HC identifies a neighbor by randomly mutating genes of the individual, i.e., by producing a slightly different solution. An iterated HC, as the one shown in Algorithm 2, iterates across all possible neighbors of a given solution (line 9). Once a fitter neighbor (v_n) is found (lines 10-11), this becomes the current point in the search space and the process is repeated. If no fitter neighbor is found (line 13), then the search terminates and a maximum has been found (by definition). To avoid local maxima, the HC algorithm is restarted multiple (t) times from a random point (lines 7-20).

Multiple ascent HC is a variant of the standard HC algorithm designed to escape from local optima. In particular, when a local optimum is reached, a set of random changes are performed in order to move away from that point and continue to explore the solution space. This procedure is repeated n times, depending on a parameter called *number of descents*, while the number of random changes applied is set through the descent depth parameter.

3 Pseudo-code of Simulated Annealing (from [607]).

```
1:  $t \leftarrow 0$
2:  $T \leftarrow T_{max}$
3:  randomly select a current point $v_c$
4:  **while** halting-criterion not met **do**
5:      **while** termination-condition not met **do**
6:          select a new point $v_n$ in the neighborhood of $v_c$
7:          **if** $eval(v_c) < eval(v_n)$ **then**
8:              $v_c \leftarrow v_n$
9:          **else**
10:             **if** $random[0,1) < e^{\frac{eval(v_n)-eval(v_c)}{T}}$ **then**
11:                 $v_c \leftarrow v_n$
12:             **end if**
13:         **end if**
14:     **end while**
15:     $T \leftarrow g(T,t)$
16:     $t \leftarrow t+1$
17: **end while**
```

4.2.2 Simulated Annealing

Simulated Annealing (SA) [604], like HC, is a local search method. As it can be seen from Algorithm 3, the algorithm is pretty similar to HC. However, one can move from a solution v_c to a neighbor v_n if (i) vc has a better fitness value than v_n (lines 7-8) or (ii) one can move from v_n to a less fit solution v_c (lines 10-11) if $p < m$, where p is a random number in the range $[0\ldots1]$ and $m = e^{\Delta fitness/T}$. The parameter T (temperature) regulates the likelihood to move to a less fit solution and it decreases ("cools") over time according to a function $g(T,t)$ (line 15). A typical *cooling mechanism* is given by $T = T_{max} \cdot e^{-t \cdot r}$ (T_{max} is the starting temperature (line 2), r is the *cooling factor*, t the number of iterations), and $\Delta fitness$ is the difference between the fitness values of the two neighbor individuals being compared. The effect of cooling in SA is that the probability of following an unfavorable move is reduced. This (initially) allows the search to move away from local optima in which the search might be trapped. As the simulation "cools", the search becomes more and more equivalent to a simple hill climb.

4 Pseudo-code of Particle Swarm Optimization.

```
 1: for $i = 1 \rightarrow n$ do
 2:     initialize the particle's position $x_i$
 3:     set the particle's best known position $p_i \leftarrow x_i$
 4:     initialize the particle's velocity $v_i$
 5:     if $eval(p_i) < eval(g)$ then
 6:         $g \leftarrow p_i$
 7:     end if
 8: end for
 9: while termination-condition not met do
10:     for $i = 1 \rightarrow n$ do
11:         update particle's velocity $v_i$
12:         $x_i \leftarrow x_i + v_i$
13:         if $eval(x_i) < eval(p_i)$ then
14:             $p_i \leftarrow x_i$
15:             if $eval(p_i) < eval(g)$ then
16:                 $g \leftarrow p_i$
17:             end if
18:         end if
19:     end for
20: end while
```

4.2.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) was introduced by Kennedy and Ebhart in 1995 [456]. The basic concept of the algorithm is to create a swarm of particles which move in the space around them (the problem space) searching for their goal, the place which best suits their needs given by a fitness function. A nature analogy with birds is the following: a bird flock flies in its environment looking for the best place to rest. The best place can be a combination of characteristics like space for all the flock, food access, water access or any other relevant characteristics.

PSO is described in the pseudocode of Algorithm 4. First, an initial population (named *swarm*) of n random solutions (named *particles*) is created. Every particle in the swarm is described by its position and velocity. A particle position represents a possible solution to the optimization problem, and velocity represents the search distances and directions that guide particle flying. At each particle is assigned an initial position x_i (line 2), which is also the best known position p_i known so far (line 3), and an initial velocity v_i (line 4). Then, each particle flies in the problem space with a velocity that is regularly adjusted according to the composite flying experience of the particle and some, or all, the other particles (line 12). Given the new velocity, the position is updated accordingly (line 12). The fitness of each particle (that depends on the position x_i) is evaluated and, if needed, the best position p_i is updated (lines 13-14). Similarly, the overall best position among all particles g is updated (lines 15-16). The process of updating particles' velocity and position (lines 9-20) is repeated until a termination criterion (e.g., maximum number of iterations) is met.

5 Pseudo-code of a Genetic Algorithm (from [607]).

```
1:  $t \leftarrow 0$
2:  initialize a population $P(t)$ of $n$ individuals
3:  evaluate $P(t)$
4:  **while** termination-condition not met **do**
5:      $t \leftarrow t+1$
6:      select a subset $P'(t-1)$ of individuals from $P(t-1)$ to reproduce
7:      apply crossover to $P'(t-1)$ and introduce offspring in $P(t)$
8:      mutate individuals in $P(t)$
9:      evaluate $P(t)$
10: **end while**
```

4.2.4 Genetic Algorithms

Genetic Algorithms (GAs) [337] belong to the family of evolutionary algorithms that, inspired by the theory of natural evolution, simulate the evolution of species emphasizing the law of survival of the strongest to solve, or approximately solve, optimization problems. Thus, these algorithms create consecutive populations of individuals, considered as feasible solutions for a given problem (phenotype) to search for a solution which gives the best approximation of the optimum for the problem under investigation. To this end, a fitness function is used to evaluate the goodness (i.e., fitness) of the solutions represented by the individuals, and genetic operators based on selection and reproduction are employed to create new populations (i.e., generations).

As shown in Algorithm 5, the elementary evolutionary process of these algorithms is composed of the following steps:

1. a random initial population $P(0)$ is generated (line 1) and a fitness function is used to assign a fitness value to each individual (line 2);
2. given t the current generation (line 3), some individuals of a population $P'(t-1)$ are selected to form the parents (line 6) and new individuals are created by applying genetic operators (i.e., crossover and mutation). The crossover operator (line 7) combines two individuals (i.e., parents) to form one or two new individuals (i.e., offspring), while the mutation operator (line 8) is used to randomly modify an individual. Then, to determine the individual that will survive among the offspring and their parents a survivor selection is applied according to the individuals' fitness values (line 9);
3. step 2 is repeated until stopping criteria hold.

When designing a GA, the crossover and mutation operators play a crucial role. Different crossover operators can be used. Among the most used, there are:

- *One-point crossover*. A point in the chromosome of the two parents is selected, and all the genes beyond that point in either chromosome are swapped between the two parents.

- *Two-point crossover*. Two points in the chromosome of the two parents are selected, and everything between the two points is swapped between the parents, generating the offspring.
- *Uniform crossover*. A fixed mixing ratio between two parents is used. Unlike one- and two-point crossover, the uniform crossover enables the parent chromosomes to contribute the gene level rather than the segment level.

As for the mutation, the selection of the operator depends on the representation of the solution. For integer and float genes, a widely-used operator is the uniform mutation. Using such an operator, the value of a chosen gene is replaced with a uniform random value selected between the user-specified upper and lower bounds for that gene. If the solution is represented by a binary string a common mutation operator is the *bit flip*, where the bit of the chosen gene is inverted (i.e., if the value is 1, it is changed to 0 and vice versa).

It is worth noting that during each generation these operators are applied with a certain probability, named *crossover rate* and *mutation rate*. In addition, at each generation parents have to be selected for crossover and mutation. Thus, also the selection operator plays an important role. The most used selectors are *roulette wheel*, in which each individual's probability of selection is directly proportional to its relative fitness, and *tournament selection*, where small subsets of the population are selected randomly (a tournament) and the most fit member of the subset is selected for the next generation.

Finally, the stopping criterion for the evolutionary process is usually based on a maximum number of generations. This stopping criterion can be combined with other criteria to reduce the computation time. For example, the search process can be stopped when there is no improvement in the fitness value for a given number of generations.

A variant of GAs is Genetic Programming (GP) [477], where the aim is to generate programs (that can be also prediction models, or expressions, etc.) having certain properties. The representation is often (but not necessary) a program Abstract Syntax Tree (AST) and the fitness is evaluated by executing the program.

4.2.5 Multi-Objective Optimization

An optimization problem can have one objective, but also more than one objective (multi-objective optimization). In a multi-objective optimization problem, a solution is described in terms of a decision vector $(x_1, x_2, ..., x_n)$ in the decision space X. Then the fitness function $f : X \rightarrow Y$ evaluates the quality of a specific solution by assigning it an objective vector $(y_1, y_2, ..., y_k)$ in the objective space Y, where k is the number of objectives.

Comparing solutions in multi-objective optimization is not as trivial as in the case of single-objective optimization problems. Specifically, in multi-objective optimization problems it is necessary to exploit the concept of Pareto dominance: an objective vector y_1 is said to dominate another objective vector y_2 $(y_1 \succ y_2)$ if no

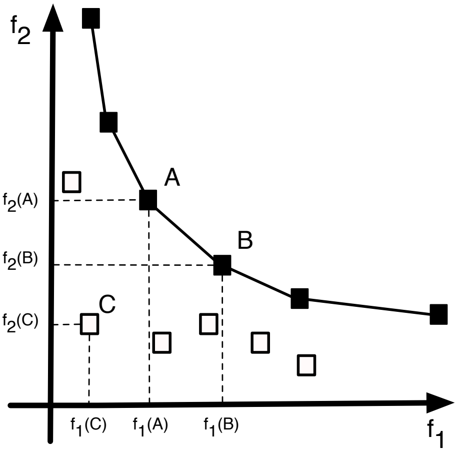


Fig. 4.1: Pareto dominance: A and B are non-dominating solutions, while C is dominated by both A and B.

component of y_1 is smaller than the corresponding component of y_2 and at least one component is greater. The Pareto dominance allows to say that a solution x_1 is better than another solution x_2, i.e., x_1 dominates x_2 ($x_1 \succ x_2$), if $f(x_1)$ dominates $f(x_2)$. For example, in Figure 4.1, point C is dominated by A and B since $f_1(A) > f_1(C)$, $f_2(A) > f_2(C)$, $f_1(B) > f_1(C)$, and $f_2(B) > f_2(C)$. Instead, A and B represent non-dominating solutions: if we consider A, there is at least another solution (B in our case) such that $f_1(B) > f_1(A)$. Similarly, if we consider B, there is at least another solution (A) such that $f_2(A) > f_2(B)$. It is worth noting that using such a definition it is possible to define a *set* of optimal solutions, i.e., solutions not dominated by any other solution. Such solutions may be mapped to different objective vectors. In other words, there may exist several optimal objective vectors representing different trade-offs between the objectives. This set of optimal solutions is generally denoted as the *Pareto set* $X^* \subseteq X$, while the fitness values achieved by such solutions represent the Pareto front $Y^* \subseteq Y$.

In principle, a multi-objective optimization problem can be reduced to a single-objective optimization problem. For instance, the different objectives can be aggregated into a single one. However, the analysis of the Pareto front can help the decision maker in (i) selecting the most suitable solution, i.e., the solution that provides the best compromise in a particular scenario; and (ii) analyzing the trade-off provided by each solution.

The concept of Pareto dominance is also used to rank solutions and to apply selection strategies based on non-domination ranks. Generally, such algorithms are elitist: the best solutions—i.e., the non-dominated solutions—are either kept in the population itself or are stored separately to be reused. In the first case, they participate to the reproduction process. However, the number of non-dominated solutions might greatly increase with the number of objectives, which limits the number of places reserved for new individuals. Therefore, such algorithms generally use a specific operator to preserve diversity. The elitist Non-dominated Sorting Genetic Algorithm (NSGA-II) [227] is certainly one of the most popular algorithms belonging to this category.

A naive multi-objective optimization algorithm would require $\mathcal{O}(M\,N)$ comparisons to identify each solution of the first nondominated front in a population of size N and with M objectives, and a total of $\mathcal{O}(M\,N^2)$ comparisons to build first nondominated front. This is because each solution needs to be compared with all other solutions. Since the above step has to be repeated for all possible fronts—which can be at most N, if each front is composed of one solution—the overall complexity for building all fronts is $\mathcal{O}(M\,N^3)$.

NSGA-II uses a faster algorithm for nondominated sorting, which has a complexity $\mathcal{O}(M\,N^2)$:

1. for each solution p in the population, the algorithm finds the set of solutions S_p dominated by p and the number of solutions n_p that dominate p. The set of solutions with $n_p = 0$ are placed in the set first front F_1.
2. $\forall p \in F_1$, solutions $q \in S_p$ are visited and, if $n_q - 1 = 0$, then solution q is placed in the second front F_2. This step is repeated $\forall p \in F_1$ to generate F_3, etc.

To compare solutions, NSGA-II uses the "crowded comparison operator". That is, given two solutions x_1 and x_2, x_1 is preferred over x_2 if it belongs to a different (better) front. Otherwise, if x_1 and x_2 belong to the same front, the solution located in the less crowded region of the front is preferred.

Then, NSGA-II produces the generation $t+1$ from generation t as follows:

1. generating the child population Q_t from the parent population P_t using the binary tournament selection and the crossover and mutation operators defined for the specific problem;
2. creating a set of $2N$ solutions $R_t \equiv P_t \bigcup Q_t$;
3. sorting R_t using the nondomination mechanism above described, and forming the new population P_{t+1} by selecting the N best solutions using the crowded comparison operator.

4.3 Search-based Software Modularization

Software (re)modularization is probably one of the software evolution tasks where SBSE techniques have been applied most. Given a set of artifacts—for example classes or source code files—the aim of software modularization is to identify groups of artifacts that, according to given criteria, are cohesive enough and exhibit low coupling with other groups. During software evolution, this can be useful to support system restructuring, but also—without restructuring the system—to support program comprehension by highlighting groups of cohesive components.

4.3.1 The Bunch approach for software modularization

Bunch [610] is a software modularization tool that relies on search-based optimization techniques.

Problem definition. Generally speaking, software modularization can be seen as a graph partitioning problem, whose solution is known to be NP-hard [312]. In the past, various authors have tackled this problem with clustering techniques [559, 701, 923]. In the following, we illustrate the problem as it has been formalized and solved—using search-based optimization techniques—by Mitchell and Mancoridis in their *Bunch* tool. Bunch operates on a system representation called Module Dependency Graph (*MDG*), a graph $G = (V, E)$ where nodes V are system artifacts and edges E are relations between such artifacts (e.g., function or method calls). The goal of modularization is to partition G into n clusters $Pi_G = \{G_1, G_2, \ldots, G_n\}$. Each cluster G_i is composed of a set of (non-overlapping) artifacts from V, i.e., $G_i \cap G_j = \emptyset \; \forall i, j \; \in 1 \ldots n$.

Solution representation. To find solutions of the modularization problem using search-based heuristics, the problem needs to be encoded in a chromosome. Given a software system composed of n software components (e.g., classes), the chromosome is represented as a n-sized integer array, where the value $0 < v \leq n$ of the i^{th} element indicates the cluster to which the i^{th} component is assigned. A solution with the same value (whatever it is) for all elements means that all software components are placed in the same cluster, while a solution with all possible values (from 1 to n) means that each cluster is composed of one component only.

Fitness function. Starting from the *MDG* (weighted or unweighted), the output of a software module clustering algorithm is represented by a partition of this graph. A good partition of an *MDG* should be composed of clusters of nodes having (i) high dependencies among nodes belonging to the same cluster (i.e., high cohesion), and (ii) few dependencies among nodes belonging to different clusters (i.e., low coupling). To capture these two desirable properties of the system decompositions (and thus, to evaluate the modularizations generated by Bunch), Mancoridis et al. [555] define the Modularization Quality (MQ) metric as in Equation 4.1, where k is the number of modules, A_i is the Intra-Connectivity (i.e., cohesion) of the i^{th} cluster and $E_{i,j}$ is the Inter-Connectivity (i.e., coupling) between the i^{th} and the j^{th} clusters.

$$MQ = \begin{cases} \left(\frac{1}{k}\sum_{i=1}^{k} A_i\right) - \left(\frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} E_{i,j}\right) & \text{if } k > 1 \\ A_1 & \text{if } k = 1 \end{cases} \tag{4.1}$$

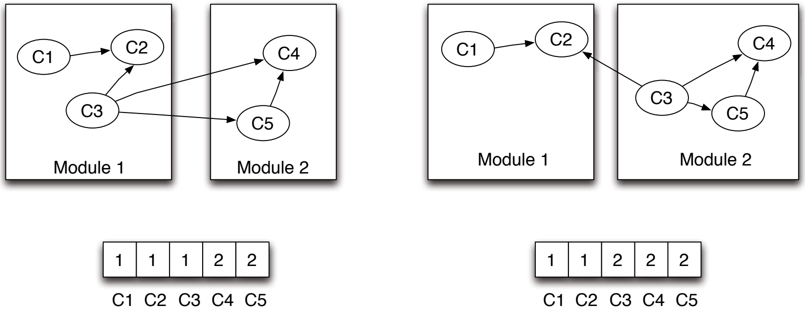
The Intra-Connectivity of a cluster i is given by Equation 4.2, where μ_i is the number of intra-cluster edges, N_i is the number of nodes of cluster i, and consequently N_i^2 is the maximum number of such intra edges.

$$\frac{\mu_i}{N_i^2} \tag{4.2}$$

The Inter-Connectivity between two clusters i and j is given by Equation 4.3, where $\varepsilon_{i,j}$ is the the number of edges between i and j, while N_i and N_j are the number of nodes in i and j respectively.

$$\frac{\varepsilon_{i,j}}{2 \cdot N_i \cdot N_j} \tag{4.3}$$

Figure 4.2 shows an example of *MDG* and of its representation. For the *MDG* in Figure 4.2-a, the *MQ* is equal to $1/2 \cdot (2/3^2 + 1/2^2) - (1/((2 \cdot 1)/2)) \cdot (2/(2 \cdot 3 \cdot 2)) = 0.07$. Moving Component $C2$ to Module 2 (Figure 4.2-b), the *MQ* becomes $1/2 \cdot (1/2^2 + 3/3^2) - (1/((2 \cdot 1)/2)) \cdot 1/(2 \cdot 3 \cdot 2) = 0.2$. Hence, as expected, the modularization quality increases.



(a) MQ=0.5

(b) MQ=1.5

Fig. 4.2: Module Dependency Graph (MDG) used by Bunch [610], its chromosome representation, and resulting *MQ* value.

Supported search-based techniques and change operators. Bunch allows to solve the software modularization problem using different search-based optimization heuristics, namely HC, SA, and GAs. In principle Bunch also allows to solve the problem exhaustively, however—as reported by Mitchell and Mancoridis [610]—the number of possible partitions exponentially increases with the number of nodes.

The HC approach works as follows. It starts with a randomly generated modularization i.e., a chromosome filled with random numbers varying between 1 and *n*. Then, neighbor solutions are created by moving one artifact from a cluster to another, i.e., by randomly changing the value in a gene. After that, the fitness function—i.e., the *MQ*—of the new produced solution is evaluated, and if its fitness is better than the previous one, then the solution is accepted and the evolution continues.

The above approach has two weaknesses. The first one is that HC algorithms can converge to local optima; the second one is that the algorithm may tend to create isolated clusters, i.e., clusters composed of one artifact only. The local optima problem is mitigated through multiple restarts of the HC, using initial solutions belonging to a population of randomly generated ones, and specifically from a subset of it having the highest MQ. An alternative is to use SA instead of HC. Since SA does not always proceed towards (locally) better solutions, this can mitigate the local optima problem.

Also, the problem can be solved using GAs, which evolve multiple solutions—i.e., a population of individuals—rather than single one. The GAs-based approach of Bunch [610]—named *Gadget*—has been described by Doval et al. [260]. A GA evolves the population using a *selection operator*, which selects individuals to reproduce based on the fitness function, a one-point *crossover* operator, and a *mutation* operator which is the same used for HC.

The problem of isolated clusters is dealt by assigning such isolated clusters to another, randomly chosen, cluster.

Empirical evaluation. There are different ways to evaluate the quality of solutions obtained using a modularization technique. When a reference (ideal) solution is available for a given system, it can be compared with the solution produced by the modularization technique. Such a comparison can be made using the MoJoFM eFfectiveness Measure (MoJoFM) [914], defined in Equation 4.4, where $mno(A,B)$ is the minimum number of *Move* or *Join* operations one needs to perform in order to transform the partition A into B, and $max(mno(\forall A,B)$ is the maximum possible distance of any partition A from the gold standard partition B.

$$MoJoFM(A,B) = 100 - \left(\frac{mno(A,B)}{max(mno(\forall A,B))} \times 100 \right) \tag{4.4}$$

When no reference solution is available, one can qualitatively evaluate a modularization solution (e.g., by relying on experts), and also evaluate the *stability* of the technique being used. A clustering technique is stable if it produces similar results over multiple runs. From a qualitative point of view, Mitchell and Mancoridis [610] applied Bunch on a 50 KLOC C++ program that implements a file system service. Bunch created two main clusters, related to two different file systems being accessed, and this was confirmed by the system expert. Also, Bunch created a hierarchical decomposition, which allowed experts to review the proposed modularization at different levels of granularity. To evaluate the clustering stability, Mitchell and Mancoridis [610] used (i) the *EdgeSim* similarity measurement, that normalizes the number of intra and inter cluster edges that are in agreement between two different modularizations, and (ii) the *MeCl* similarity that determines the distance between two modularizations. A study performed on the Java Swing library reported an average *EdgeSim* of 93.1% and an average *MeCl* of 96.5%.

4.3.2 Multi-Objective Modularization

The modularization approach described in Section 4.3.1 produces solutions that are (near) optimal with respect to a single objective, i.e., the MQ. To this extent, MQ achieves a compromise between having a good cohesion and low coupling.

The alternative to single-objective optimization is to use multi-objective optimization where—as explained in Section 4.2.5—the found solutions are Pareto-optimal, i.e., each solution is better than another with respect to a particular objective, while it may not be better with respect to other objectives.

Fitness function. Praditwong et al. [701] have proposed an approach for multi-objective optimization, where the considered objectives are the following:

- maximizing the number of intra-module edges: that is, a high number of intra-module edges denotes a high cohesion;
- minimizing the number of inter-module edges: that is, a low number of inter module edges denotes a low coupling;
- maximizing the number of clusters, which generally favors high cohesion;
- minimizing the number of isolated clusters: such objective is an alternative to the repairing solution described in Section 4.3.1 to handle isolated clusters;
- maximizing MQ which, as explained in Section 4.3.1 favors solutions achieving a compromise between cohesion and coupling.

Supported search-based techniques and change operators. Praditwong et al. [701] have implemented the multi-objective modularization using the same operators of Mitchell et al. [260], however using a NSGA-II [226] multi-objective GA instead of a simple GA.

Empirical evaluation. Praditwong et al. [701] have evaluated their multi-objective approach to modularize 17 MDG extracted from various C programs (e.g., Unix utilities such as *bison*, *ispell*, *lynx*, *ncurses*, and *rcs*), and compared it with the single-objective, hill-climbing based approach of Mitchell and Mancoridis [610]. Other than exhibiting the advantages outlined above, i.e., the capability of the software engineer to select solutions that are particularly good for specific objectives than for others, the multi-objective GA was also able to outperform single-objective modularization for each specific modularization objective (e.g., MQ increase between 15% and 50% for 10 out of 17 programs, and decrease within 5% for the other 7). In summary, besides the usual advantages of multi-objectives, it can be preferred to the single-objective alternative also for what concerns the quality of the obtained solutions [701]. However, the drawback is that multi-objective optimization is more expensive from a computational point-of-view. That is, the number of evaluations required is two orders of magnitude higher.

4.3.3 Achieving different software modularization goals

The above described approaches deal with software modularization from a structural point of view, and with the aim of obtaining cohesive and decoupled clusters. The existing literature also reports approaches where search-based remodularization was used for different purposes.

Di Penta et al. [247] and Antoniol et al. [37] deal with remodularizing software libraries with the aim of minimizing the footprint of an application in the program memory. This is particularly useful when porting applications towards devices with a limited memory. Years ago, this was particularly true for many mobile devices; nowadays most mobile devices (tablets and smartphones) have enough memory. However, memory occupation is still a concern for some specific devices such as embedded systems or active sensors.

To deal with the software miniaturization problems, Di Penta *et al.* and Antoniol et al. [37] start from a graph highlighting dependencies between applications and libraries, and between objects composing libraries. Given this graph, the goal to be achieved is to minimize the footprint of applications, considering the set of libraries they should be linked to. Since libraries can be partitioned in different ways such that the overall size of the libraries used by each application is minimized, this is still a modularization problem that can be solved using search-based optimization techniques. The miniaturization problem is then solved by using a GA similar to the one used by Doval et al. [260] for remodularization purposes. However, instead of using the MQ as fitness function, a mono-objective one (to be minimized) consisting of a weighted sum of the four factors keeping into account: (i) the total number (or size) of objects linked to each application, (ii) the number of inter-library dependencies (to avoid linking a library every time another is linked), (iii) the difference with the initial libraries (to avoid scrambling the libraries completely), and (iv) feedbacks provided by experts/original developers.

Di Penta et al. [247] and Antoniol et al. [37] applied their approach on various C programs, such as Grass, QT, MySQL, and Samba. The application footprint size was reduced of over 60% for MySQL and Samba, and between 5% and 25% for Grass and QT

A different miniaturization approach has been proposed by Ali et al. [17]. In their work, they aim at determining the set of features to be included in an application when porting it towards a mobile device with the aim of (i) maximizing customers' satisfaction and (ii) keeping the devices' battery consumption low. For each feature, they also measure the estimated battery consumption, using a framework by Binder and Hulaas [107], based on bytecode analysis. Finally, they use a NSGA-II [226] multi-objective optimization to determine the set of features to include in the ported application. Ali *et al.* [17] experimented their approach to miniaturize an email client (Pooka) and an instant messenger (SIP). The minuaturization was experimented by considering some user requirements collected through a survey, and some hypothetical constraints in terms of user satisfaction and consumption. Compared with a manual minuaturization, the proposed approach allowed to save about 77% of the effort.

4.3.4 Putting the developer in the loop: interactive software modularization

The approaches for software modularization described above have the advantage of being completely automatic, i.e., they produce a possible modularization without requiring manual intervention.

While automatic re-modularization approaches proved to be very effective in increasing cohesiveness and reducing coupling of software modules, they do not take into account developers' knowledge when deciding to group together (or not) certain components. For example, a developer may decide to place a function in a given module even if, in its current implementation, the function does not communicate a lot with other functions in the same module. This is because the developer may be aware that, in future releases, such a function will strongly interact with the rest of the module. Similarly, a developer may decide that two functions must be placed in two different modules even if they communicate. This is because the two functions have different responsibilities and are used to manage semantically different parts of the system.

To deal with this problem, different authors have proposed methods to incorporate developers' feedback in search-based remodularization algorithms.

Hall et al. [360] proposed a supervised remodularization approach, named SUMO (**Su**pervised Re**mo**dularization), that integrates existing modularization approches—such as Bunch [610]—with corrections provided by the software engineer. The idea of the approach is the following:

1. First, a solution of the modularization problem is created using automatic modularization (Bunch, for example).
2. After that, the user provides corrections through a user interface. Such corrections consist of two sets of relations, Rel^+, defined as pairs of artifacts that should belong to the same cluster (i.e., go together), and Rel^-, defined as pairs of artifacts that should not go together.
3. After that, a constraint satisfaction approach is used to modify the initially produced clusters with the aim of satisfying constraints expressed by sets of relations Rel^+ and Rel^-. Steps 2 and 3 are repeated until the software engineer finds no further corrections.

Bavota et al. [82] proposed the use of Interactive GA [818] (IGA) to solve the modularization problem, considering it as both single-objective and multi-objective optimization problem. The problem is encoded as done in Gadget [260]. The single-objective GA uses as fitness function the *MQ* metric, while the multi-objective GA considers the five different objectives of the approach by Praditawong et al. [701] described in Section 4.3.2.

The basic idea of the IGA is to periodically add a constraint to the GA such that some specific components shall be put in a given cluster among those created so far. Thus, the IGA evolves exactly as the non-interactive GA. Every *nGens* generations, the best individual is selected and shown to the software engineer. After that, the software engineer analyzes the proposed solutions and provides feedback, indicating that certain components must be placed in a specific cluster.

In principle, the IGA can ask feedback for every pair of components. However, this would be too much work for the software engineer. To limit the amount of feedback, the Algorithm 6—takes the best solution produced by the GA, randomly selects two components (from the same cluster or from different clusters), and then asks the software engineer whether, in the new solutions to be generated, such components must be placed in the same cluster (i.e., stay together) or whether they should be kept separated. In total, every *nGens* generations the software engineer is asked to provide feedback about a number *nFeedback* of component pairs from the best solution (in terms of *MQ*) contained in the current population.

6 Pseudocode of the Interactive GA for software modularization.

```
for $i = 1 \ldots nInteractions$ do
    Evolve GA for $nGens$ generations
    Select the solution having the highest MQ
    for $j = 1 \ldots nFeedback$ do
        Randomly select two components $c_i$ and $c_j$
        Ask the developer whether $c_i$ and $c_j$ must go together or kept separate
    end for
    Repair the solution to meet the feedback
    Create a new GA population using the repaired solution as starting point
end for
Continue (non-interactive) GA evolution until it converges or it reaches $maxGens$
```

After feedback is provided, the solution is repaired by enforcing the constraints, e.g., by randomly moving one of c_i and c_j away if the constraint tells that they shall be kept separated. After all *nFeedback* have been provided, a new population is created by randomly mutating such a repaired solution. Then, the GA starts again.

During the GA evolution, to ensure constraints specified by the software engineers are satisfied, a penalty factor is added to the fitness function (as proposed by Coello Coello [195]), to penalize solutions violating the constraints imposed by the developers. Given $CS \equiv cs_1, \ldots cs_m$ the set of feedback collected by the users, the fitness $F(s)$ for a solution s is computed by Equation 4.5, where $k > 0$ is an integer constant weighting the importance of the feedback penalty, and $vcs_{i,s}$ is equal to one if solution s violates cs_i, zero otherwise. After *nInteractions* have been performed, the GA continues its evolution in a non-interactive way until it reaches stability or the maximum number of generations.

$$F(s) = \frac{MQ(s)}{1 + k \cdot \sum_{i=1}^{m} vcs_{i,s}} \tag{4.5}$$

A variant of the interactive GA described above specifically aims at avoiding isolated clusters, by asking the developers where the isolated component must be placed. For non-isolated clusters, the developer is asked to specify for each pair of components whether they must stay together or not.

Finally, the multi-objective variants of IGA are quite similar to the single-objective ones. Also in this case, two variants have been proposed, one—referred to as R-IMGA (Random Interactive Modularization Genetic Algorithm)—where feedback is provided on randomly selected pairs of components, and one—referred to as IC-IMGA (Isolated Clusters Interactive Modularization Genetic Algorithm)—where feedback is provided on components belonging to isolated (or smallest) clusters.

In summary, interactive approaches to software modularization help software engineers to incorporate their rationale in automatic modularization approaches. The challenge of such approaches, however, is to limit the amount of feedback the software engineers have to provide. On the one hand, a limited amount of feedback can result in a modularization that is scarcely usable. On the other hand, too much feedback may become expensive and make the semi-automatic approach no longer worthwhile.

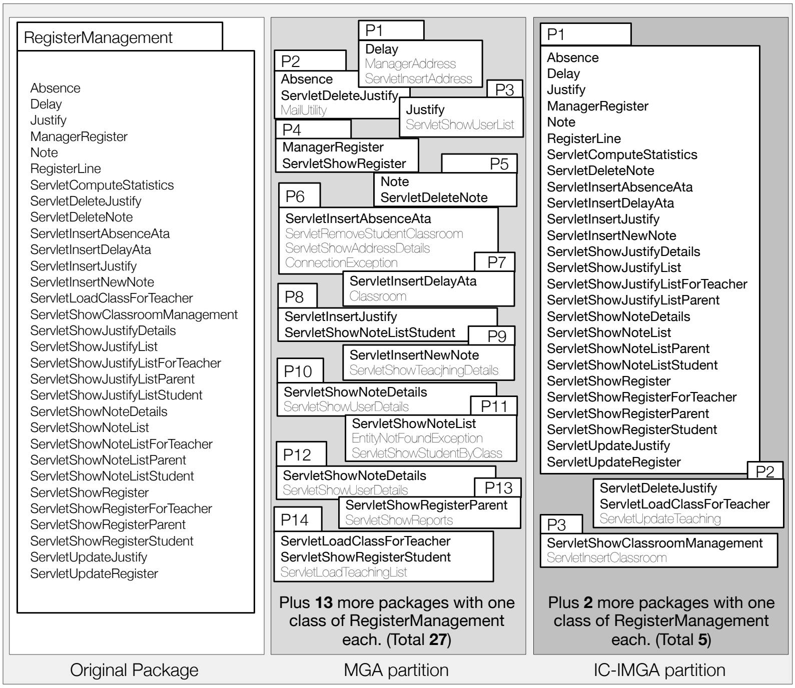


Fig. 4.3: MGA *vs* IC-IMGA in reconstructing the *RegisterManagement* package of SMOS [82].

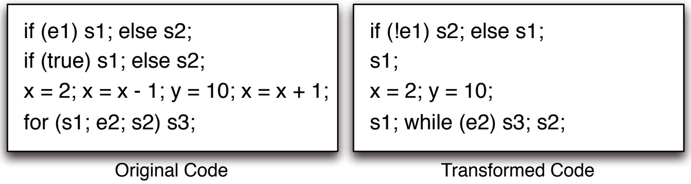
In the evaluation reported in the paper by Bavota et al. [82], the authors compare the different variants of IGAs with their non-interactive counterparts in the context of software re-modularization. The experimentation has been carried out on two software systems, namely GESA and SMOS, by comparing the ability of

GAs and IGAs to reach a fair trade-off between the optimization of some quality metrics (that is the main objective of GAs applied to software re-modularization) and the closeness of the proposed partitions to an authoritative one (and thus, their meaningfulness). The achieved results show that the IGAs are able to propose re-modularizations (i) more meaningful from a developer's point-of-view, and (ii) not worse, and often even better in terms of modularization quality, with respect to those proposed by the non-interactive GAs.

To understand in a practical way what is the difference between the performances of interactive and non-interactive GAs, Figure 4.3 shows an example extracted from the re-modularization of the SMOS software system. The figure is organized in three parts. The first part (left side) shows how the subsystem `RegisterManagement` appears in the original package decomposition (i.e., which classes it contains) made by the SMOS's developers. This subsystem groups together all the classes in charge to manage information related to the scholar register (e.g., the students' delay, justifications for their absences and so on). The second part (middle) reports the decomposition of the classes contained in `RegisterManagement` proposed by the MGA (Modularization Genetic Algorithm). Note that some classes not belonging to the `RegisterManagement` were mixed to the original set of classes. These classes are reported in light gray. Finally, the third part (right side) shows the decomposition of the classes contained in `RegisterManagement` proposed by the IC-IMGA. Also in this case, classes not belonging to the original `RegisterManagement` package are reported in light gray. As we can notice, the original package decomposition groups 31 classes in the `RegisterManagement` package. When applying MGA, these 31 classes are spread into 27 packages, 13 of which are singleton packages. As for the remaining 14 they usually contain some classes of the `RegisterManagement` package mixed with other classes coming from different packages (light gray in Figure 4.3). The solution provided by IC-IMGA is quite different. In fact, IC-IMGA spreads the 31 classes in only 5 packages. Moreover, it groups together in one package 26 out of the 31 classes originally belonging to the `RegisterManagement` package. It is striking how much the partition proposed by IC-IMGA is closer to the original one resulting in a higher MoJoFM achieved by IC-IMGA with respect to MGA and thus, a more meaningful partitioning from a developer's point of view.

4.4 Software Analysis and Transformation Approaches

In this section we describe how to instantiate a search-based approach to automatically modify source code (and models) for various specific purposes, for example improving maintainability or fixing bugs.



```
if (e1) s1; else s2;
if (true) s1; else s2;
x = 2; x = x - 1; y = 10; x = x + 1;
for (s1; e2; s2) s3;
```

Original Code

```
if (!e1) s2; else s1;
s1;
x = 2; y = 10;
s1; while (e2) s3; s2;
```

Transformed Code

Fig. 4.4: Example of program transformation [283].

4.4.1 Program transformation

Program transformation can be described as the act of changing one program to another. Such a transformation is achieved by converting each construct in some programming language into a different form, in the same language or, possibly, in a different language. Further details about software transformation—and in particular model transformation—can be found in a paper by Mens and Van Gorp [597].

Program transformation has been recognized as an important activity to facilitate the evolution of software systems. The hypothesis is that the original source code can be progressively transformed into alternative forms. The output of the process is a program possibly easier to be understood and maintained, or without bugs (according to a given test suite).

Figure 4.4 reports an example of transformation for an original program to another program. In this case the program semantics is preserved and the transformation aims at improving the comprehensibility of the program. Such a transformation is achieved by applying a set of transformation axioms. The choice of the axioms to be applied is guided by the final goal of program transformation, that is making the code more comprehensible.

Problem definition. The program transformation problem can be considered as an optimization problem, where an optimal sequence of transformation axioms (*transformation tactic*) is required in order to make an input program easier to comprehend [283]. Transformations can be applied at different points—e.g., nodes of the Control Flow Graph (CFG)—of the program. The set of transformation rules and their corresponding application point is therefore large. In addition, many rules may need to be applied to achieve an effective overall program transformation tactic, and each will have to be applied in the correct order to achieve the desired result. All these considerations suggest that the problem is hard to solve and it represents a rich soil for search-based approaches. Specifically, search-based approaches can be used to identify a sub-optimal sequence of transformations in a search space that contains all the possible allowable transformation rules.

Solution representation. A solution is represented by a sequence of transformations that have to be applied on an input program. Fatiregun et al. [283] use a very simple representation, where each solution has a fixed sequence length of 20 possible transformations. Thus, each solution contains the identifier of the specific transformation in the considered catalogue. The FermaT[2] [908] transformation tool is used to apply the transformation encoded in each solution. FermaT has a number of built-in transformations that could be applied directly to any point within the program. Examples of such transformations are: `@merge-right`, that merges a selected statement into the statement that follows it, or `@remove-redundant-vars`, that will remove any redundant variables in the source program.

Fitness function. A program is transformed in order to achieve a specific goal, e.g., reduce its complexity. In this case, the fitness function could be based on existing metrics for measuring software complexity, such as Lines of Code (LoC) or cyclomatic complexity. Fatiregun *et al.* [283] measure the fitness of a potential solution as the difference in the lines of code between the source program and the new transformed program created by that particular sequence of transformations. Specifically, they first compute the LOC of the original program. Then, they apply on the original program the sequence of transformations identified by the search-based approach obtaining a new version of the program. Using such a fitness function, an optimum solution would be the sequence of transformations that results in an equivalent program with the fewest possible number of statements.

Supported search-based techniques and change operators. The search-based techniques used to support program transformation are HC and GA [283]. In the HC implementation the neighbor has been defined as the mutation of a single gene from the original sequence leaving the rest unchanged. As for GA, a single point crossover has been used. Note that the solution proposed by Fatiregun et al. makes the implementation of crossover and mutation operators quite simple. Specifically, for the crossover a random point is chosen and genes are then swapped, creating two new children. As for the mutation, a single gene is chosen and it is changed arbitrarily.

Empirical evaluation. The effectiveness (measured in terms of size reduction) of search-based approaches for program transformation has been only preliminary evaluated on small synthetic program transformation problems [283]. Specifically, the transformations achieved with both GA and HC were compared with those returned by a purely random search of the search space. The comparison was based on two different aspects: the most desirable sequence of transformations that the algorithm finds and the number of fitness evaluations that it takes to arrive at that solution. Results indicated that GA outperforms both the random search and the HC as the source program size increases. In addition, the random search outperforms HC in some specific case. Unfortunately, until now only a preliminary analysis of the benefits provided by search-based approaches for program transformation has been performed. Studies with users are highly desirable to evaluate to what extent the achieved transformation are meaningful for developers.

[2] http://www.cse.dmu.ac.uk/ mward/fermat.html

4.4.2 Automatic Software Repair

Repairing defects in software systems is usually costly and time-consuming, due to the amount of software defects in a software system. Because of this very often—due to the lack of available resources—software projects are released with both known and unknown defects [521]. As example, in 2005, a Mozilla's developer claimed that, "*everyday, almost 300 bugs appear [...] far too much for only the Mozilla programmers to handle*" [41]. These considerations have prompted researchers in the definition of methods for automatic repair of defects. Specifically, Le Goues et al. [502] have formulated such a problem as an optimization problem and have proposed Genetic Program Repair (GenProg), a technique that uses existing test cases to automatically generate repairs for bugs in software systems.

Problem definition. The basic idea is inspired by the definition of a repair, that is a patch consisting of one or more code changes that, when applied to a program, cause it to pass a set of test cases [502]. Thus, given a buggy source code component and a test suite (where there is at least one test case that did not pass due to the presence of the bug), GP is used to automatically mutate the buggy code aiming at pass all the tests in a given test suite.

Solution representation. In GenProg each solution is represented by an abstract syntax tree that includes all the statements (i.e., assignments, function calls, conditionals, blocks, and looping constructs) in the program. In addition, to each solution is associated a weighted path computed by executing all the test cases in the given test suite. Specifically, a statement in the AST is weighted with 1 if the statement is covered by a test case that does not pass, 0 otherwise. The weighted path is used to localize buggy statements to be mutated (i.e., statements covered by test cases that do not pass). In addition, the weighted path is used to avoid mutating correct statements (i.e., statements covered by test cases that pass). The conjecture is that a program that contains an error in one area of source code likely implements the correct behavior elsewhere [273].

Fitness function. The fitness function is used to evaluate the goodness of a program variant obtained by GP. A variant that does not compile has fitness zero. The other variants are evaluated taking into account whether or not the variant passes test cases. Specifically, the fitness function for a generic variant v is a weighted sum:

$$f(v) = W_{Pos_{test}} \cdot |\{t \in Pos_{test} : v \ passed \ t\}| + W_{Neg_{test}} \cdot |\{t \in Neg_{test} : v \ passed \ t\}| \tag{4.6}$$

where Pos_{test} is the set of positive test cases that encode functionality that cannot be sacrificed and Neg_{test} is the set of negative test cases that encode the fault to be repaired. The weights $W_{Pos_{test}}$ and $W_{Neg_{test}}$ have been empirically determined using a trial and error process. Specifically, the best results have been achieved setting $W_{Pos_{test}} = 1$ and $W_{Neg_{test}} = 10$.

Selection and genetic operators. As for the selection operator, in GenProg two different operators have been implemented, i.e., roulette wheel and tournament selection. The results achieved in a case study indicated that the two operators provided almost the same performances. Regarding the crossover operator, in GenProg a single point crossover is implemented. It is worth noting that only statements along the weighted paths are crossed over. Finally, the mutation operator is used to mutate a given variant. With such an operator is possible to:

- *Insert new statement*. Another statement is inserted after the statement selected for mutation. To reduce the complexity, GenProg uses only statements from the program itself to repair errors and does not invent new code. Thus, the new statement is randomly selected from anywhere in the program (not just along the weighted path). In addition, the statement's weight does not influence the probability that it is selected as a candidate repair.
- *Swap two statements*. The selected statement is swapped with another statement randomly selected, following the same approach used for inserting a new statement.
- *Delete a statement*. The selected statement is transformed into an empty block statement. This means that a deleted statement may therefore be modified in a later mutation operation.

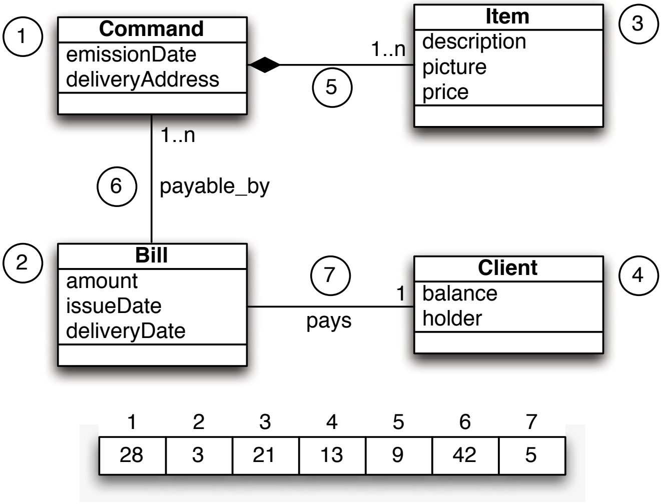
In all cases, the weight of the mutated statement does not change.

Refinement of the solution. The output of GP is a variant of the original program that passes all the test cases. However, due to the randomness of GP, the obtained solution contains more changes than what necessary to repair the program. This increases the complexity of the original program by negatively affecting its comprehensibility. For this reason, a refinement step is required to remove unnecessary changes. Specifically, in GenProg a minimization process is performed to find a subset of the initial repair edits from which no further elements can be dropped without causing the program to fail a test case (a 1-minimal subset). To deal with the complexity of finding a 1-minimal subset, delta debugging [947] is used. The minimized set of changes is the final repair, that can be inspected for correctness.

Empirical evaluation. GenProg has been used to repair 16 programs for a total of over 1.25 million lines of code [502]. The considered programs contain eight different kinds of defects, i.e., infinite loop, segmentation fault, remote heap buffer overflow to inject code, remote heap buffer overflow to overwrite variables, non-overflow denial of service, local stack buffer overflow, integer overflow, and format string vulnerability. In order to fix such defects, 120K lines of module or program code need to be modified. The results achieved indicated that GenProg was able to fix all these defects in 357 seconds. In addition, GenProg was able to provide repairs that do not appear to introduce new vulnerabilities, nor do they leave the program susceptible to variants of the original exploit.

4.4.3 Model transformation

Similar to program transformation, model transformation aims to derive a target model from a source model by following some rules or principles. Defining transformations for domain-specific or complex languages is a time consuming and dif-



1	2	3	4	5	6	7
28	3	21	13	9	42	5

Fig. 4.5: Solution representation for search-based model transformation [459].

ficult task, that requires knowledge of the underlying meta-models and knowledge of the semantic equivalence between the meta-models' concepts[3]. In addition, for an expert it is much easier to show transformation examples than to express complete and consistent transformation rules. This has pushed researchers to define a new approach for model transformation, namely Model Transformation by Examples (MTBE).

Problem definition. Using MTBE it is possible to exploit knowledge from previously solved transformation cases (examples) to transform new models by using combinations of known solutions to a given problem. This means that the target model can be obtained through an optimization process that exploits the available examples. The high number of examples as well as the high number of sequences of application of such transformations make MTBE very expensive to be performed manually. For this reason, a search-based approach, called MOdel Transformation as Optimization by Examples (MOTOE) has been proposed to identify a sub-optimal solution automatically [459]. Specifically, the approach takes as inputs a set of transformation examples and a source model to be transformed, and then it generates as output a target model. The target model is obtained by applying a subset of transformation fragments (code snippets) in the set of examples that best matches the constructs of the source model (using a similarity function).

Solution representation. In MOTOE [459] is represented by a n-dimensional vector, where n is the number of constructs in the model. This means that each construct in the model is represented by an element of such a vector. Each construct is then transformed according to a finite set of m code snippets extracted from

[3] The interested reader can find details on the evolution between models and meta-models in Chapter 2

the transformation examples (for instance change an inheritance relationship to an association). Each code snippet has a unique value ranging from 1 to m. Thus, a particular solution is defined by a vector, where the i^{th} element contains the snippet id (i.e., the transformation id) that has to be used to transform the i^{th} construct in the model. In the example depicted in Figure 4.5 there is a class diagram with 7 constructs, i.e., 4 classes and 3 relationships. The solution is represented by a vector with 7 elements that contains the transformation to be applied to each construct. For instance, the transformation 28 is applied to the class `Command`.

Fitness function. The fitness function quantifies the quality of a transformation solution. In MOTOE, the transformation is basically a 1-to-1 assignment of snippets from the set of examples to the constructs of the source model. Thus, the authors proposed a fitness function that is represented by the sum of the quality of each transformation:

$$f = \sum_{i=1}^{n} a_i \cdot (ic_i + ec_i) \tag{4.7}$$

In order to estimate the quality of each transformation, the authors analyze three different aspects (captured by $a_i \cdot (ic_i + ec_i)$ in the formula above):

- *Adequacy* (a_i) of the assigned snippet to the i^{th} construct. Adequacy is 1 if the snippet associated to i^{th} construct contains at least one construct of the same type, and value 0 otherwise. Adequacy aims at penalizing the assignment of irrelevant snippets.
- *Internal coherence* (ic_i) of the individual construct transformation. The intern coherence measures the similarity, in terms of properties, between the i^{th} construct to transform and the construct of the same type in the assigned snippet.
- *External coherence* (ec_i) with the other construct transformations. Since a snippet assigned to the i^{th} construct contains more constructs than the one that is adequate with the i^{th} construct, the external coherence factor evaluates to which extent these constructs match the constructs that are linked to i^{th} construct in the source model.

The fitness function depends on the number of constructs in the model. To make the values comparable across models with different numbers of constructs, a normalized version of the fitness function can be used [459]. Since the quality of each transformation varies between 0 and 2 (ic_i and ec_i can be both equal to 1), the normalized fitness function is $f_{norm} = \frac{f}{2 \cdot n}$.

Supported search-based techniques and change operators. The search-based techniques used to support model transformation in MOTOE are PSO and a hybrid heuristic search that combines PSO with SA [459]. In the hybrid approach, the SA algorithm starts with an initial solution generated by a quick run of PSO. In both the heuristic the change operator assigns new snippet ids to one or more constructors. Thus, it creates a new transformation solution vector starting from the previous one.

Empirical evaluation. The performance of MOTOE has been evaluated when transforming 12 class diagrams provided by a software industry [459]. The size of the these diagrams varied from 28 to 92 constructs, with an average of 58. Altogether, the 12 examples defined 257 mapping blocks. Such diagrams have been used to build an example base. Then a 12-fold cross validation procedure was used to evaluate the quality of transformations produced by MOTOE, i.e., one class diagram is transformed by using the remaining 11 transformation examples. The achieved transformations have been compared—construct by construct—with the known transformations in order to measure they correctness (automatic correctness). In addition, a manual analysis of the achieved transformation was performed to identify alternative but still valid transformations (manual correctness). The achieved results indicated that when using only PSO the automatic correctness measure had an average value of 73.3%, while the manual correctness measure had an average value of 93.2%. Instead, when using the hybrid search, correctness is even higher, i.e., 93.4% and 94.8% for the automatic and manual correctness, respectively. This means that the proposed transformations were almost as correct as the ones given by experts.

4.5 Search-based Software Refactoring

Refactoring has been defined as "*the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*" [301, 664]. Different refactoring operations[4] might improve different quality aspects of a system. Typical advantages of refactoring include improved readability and reduced complexity of source code, a more expressive internal architecture and better software extensibility [301]. For these reasons, refactoring is advocated as a good programming practice to be continuously performed during software development and maintenance [88, 301, 458, 596].

Despite its advantages, to perform refactoring in non-trivial software systems might be very challenging. First, the identification of refactoring opportunities in large systems is very difficult, due to the fact that the design flaws are not always easy to identify [301]. Second, when a design problem has been identified, it is not always easy to apply the correct refactoring operation to solve it. For example, splitting a non-cohesive class into different classes with strongly related responsibilities (i.e., Extract Class refactoring) requires the analysis of all the methods of the original class to identify groups of methods implementing similar responsibilities and that should be clustered together in new classes to be extracted. This task becomes even more difficult when the size of the class to split increases. Moreover, even when the refactoring solution has been defined, the software engineer must apply it without changing the external behavior of the system.

All these observations highlight the need for (semi)automatic approaches supporting the software engineer in (i) identifying refactoring opportunities (i.e., design flaws) and (ii) designing and applying a refactoring solution. To this aim, several different approaches have been proposed in the literature to automate (at least in part) software refactoring [83, 84, 618, 649, 659, 765, 858]. Among them, of interest for

[4] A complete refactoring catalog can be found at http://refactoring.com/catalog/.

this chapter are those formulating the task of refactoring as a search problem in the space of alternative designs, generated by applying a set of refactoring operations. This idea has been firstly presented by O'Keeffe and Cinnéide [656] that propose to treat object-oriented design as a combinatorial optimization problem, where the goal is the maximization of a set of design metrics capturing design quality (e.g., cohesion and coupling). In short, the idea is to represent a software system in an easily manipulable way (*solution representation*), in order to apply a set of refactoring operations to it (*change operators*). This can be done by selecting, during the population evolution, the solutions (i.e., refactored version of the system) maximizing a set of metrics capturing different aspects of design quality (*fitness function*).

Starting from the work by O'Keeffe and Cinnéide, search-based refactoring approaches have been applied to improve several different quality aspects of source code maintainability [659, 765], testability [649], and security [323]. Moreover, search-based refactoring techniques have also been proposed with the aim of introducing design patterns [438], and improving the alignment of code to high-level documentation [615].

The main advantages of search based refactoring techniques as compared to non search based ones are:

- *Higher flexibility.* They are suited to support a wide range of refactoring operations, also allowing to apply several of them in combination with the aim of maximizing the gain in terms of the defined fitness function.
- *Wider exploration of the solution space.* Generally, refactoring approaches are based on heuristics suggesting when a refactoring should be applied. For example, if a method has more dependencies toward a class other than the one it is implemented in, it can be a good candidate for move method refactoring. On the one side, while these heuristics can help in solving very specific design problems, they do not allow a wide exploration of the solution space (i.e., the alternative designs). On the other side, search-based algorithms explore several alternative designs by applying many thousands of refactorings in different orders, with the aim of finding a (sub)optimal solution for the defined fitness function.

We will mainly focus our discussion of search-based software refactoring on the CODe-Imp (Combinatorial Optimisation Design-Improvement) tool [614], used for the first time by O'Keeffe and O'Cinnéide [656] to test the conjecture that the maintainability of object-oriented programs can be improved by automatically refactoring them to adhere more closely to a pre-defined quality model. Then, we will briefly overview other applications of search-based refactoring presented in literature.

4.5.1 The CODe-Imp tool

CODe-Imp [614] is a tool designed to support search-based refactoring and used by several works in this field [323, 615, 649, 658, 659]. In its current implementation, the tool can be applied to software systems written in Java.

Solution Representation. In CODe-Imp the solution representation is the program itself and, in particular, its AST. Having such a representation allows to:

1. Easily evaluate the fitness of a refactoring solution. In fact, through the program AST it is possible to easily extract information exploited by almost all the fitness functions used in the search-based refactoring field, such as the number of methods in a class, the list of attributes, methods, and constructors accessed/called by a particular method, and so on. Thus, the evaluation of a generated solution—i.e., an AST representing a refactored version of the original program—is quite straightforward.
2. Determine which refactorings can legally be applied by the change operator. By "legally" we mean refactorings that do not alter the external behavior of a software system.
3. Easily apply the selected refactorings to a program. Once the search-based algorithm has found the list of refactorings to apply on the object system, it is important that these refactorings can be mapped and applied to the system source code. The mapping with source code is performed through its AST representation.

Change Operators. In the context of search-based refactoring, the change operator is a transformation of the solution representation that corresponds to a refactoring that can be carried out on the source code [659]. The current implementation of CODe-Imp supports the 14 design-level refactorings reported in Table 4.1.

Table 4.1 shows how each refactoring operation supported in CODe-Imp also has its complement (i.e., a refactoring operation undoing it). For example, a Push Down Method refactoring can be undone by applying a Pull Up Method refactoring, as well as a Replace Inheritance with Delegation can be undone by the inverse refactoring operation, i.e., Replace Delegation with Inheritance. This choice is not random, but it is driven by the fact that some search techniques (e.g., SA) must be able to freely move in the solution space. Thus, it must be possible to undo each performed refactoring operation.

Also, before applying a refactoring operation, a set of pre- and post-conditions is verified, to allow the preservation of the system's external behavior. For example, before performing a Push Down Method refactoring it is important to verify that the subclasses involved in this refactoring do not override the method inherited from their superclass. Only refactoring operations satisfying the set of defined pre- and post-conditions are considered as legal change operators in the search algorithm. CODe-Imp adopts a conservative static program analysis to verify pre- and post-conditions.

Note that, the set of change operators reported in Table 4.1 is the one adopted in all work involving the CODe-Imp despite the different final goals of the presented refactoring process, like maintainability [657–659], testability [649], and security [323].

Fitness Function. In search-based refactoring the employed fitness functions are composed of a set of metrics capturing different aspects of source code design quality. The set of adopted metrics strongly depends on the objective of the refactoring process. For example, given a hypothetic search-based approach designed to support

Table 4.1: Refactoring operations supported by CODe-Imp [614].

Refactoring Operation	Description
Push Down Method	Moves a method from a superclass to the subclasses using it
Pull Up Method	Moves a method from some subclasses to their superclass
Decrease/Increase Method Visibility	Changes the visibility of a method by one level (e.g., from private to protected)
Push Down Field	Moves a field from a superclass to the subclasses using it
Pull Up Field	Moves a field from some subclasses to their superclass
Decrease/Increase Field Visibility	Changes the visibility of a field by one level (e.g., from private to protected)
Extract Hierarchy	Adds a new subclass to a non-leaf class *C* in an inheritance hierarchy. A subset of the subclasses of *C* will inherit from the new class.
Collapse Hierarchy	Removes a non-leaf class from an inheritance hierarchy.
Make Superclass Abstract	Declares a constructorless class explicitly abstract.
Make Superclass Concrete	Removes the explicit abstract declaration of an abstract class without abstract methods.
Replace Inheritance with Delegation	Replaces an inheritance relationship between two classes with a delegation relationship
Replace Delegation with Inheritance	Replaces a delegation relationship between two classes with an inheritance relationship

Extract Class refactoring—i.e., the decomposition of a complex low-cohesive class in smaller more cohesive classes—it would be necessary to verify that the extracted classes are (i) strongly cohesive, and (ii) lowly coupled between them. These two characteristics would indicate a good decomposition of the refactored class. Thus, in such a case cohesion and metrics should be part of the defined fitness function.

In the studies conducted with CODe-Imp, several different fitness functions have been used and tuned to reach different final goals in source code design. O'Keeffe and Ó Cinnéide [657] try to maximize the understandability of source code by adopting as fitness function an implementation of the *Understandability* function from the Quality Model for Object-Oriented Design (QMOOD) defined by Bansiya and Davis [69][5]. QMOOD relates design properties such as encapsulation, modularity, coupling, and cohesion to high-level quality attributes such as reusability, flexibility, and understandability using empirical and anecdotal information [69]. In the work by O'Keeffe and Ó Cinnéide [659] the employed *Understandability* function is defined by Equation 4.8 where software characteristics having a positive coefficient (e.g., *Encapsulation*) positively impact source code understandability, while those having a negative coefficient (e.g., *Abstraction*) negatively impact code understandability. A detailed description of these metrics can be found in [657].

$$\begin{aligned} Understandability = -0.33 \times (&Abstraction + Encapsulation - Coupling \\ &+ Cohesion + Polymorphism - Complexity - DesignSize) \end{aligned} \tag{4.8}$$

O'Keeffe and Ó Cinnéide [659] perform a broader experimentation using, besides the *Understandability* function, also QMOOD's *Flexibility* and *Reusability* functions as evaluation functions. The definition of the *Flexibility* function is given in Equation 4.9 while the *Reusability* function is defined in Equation 4.10.

$$\begin{aligned} Flexibility = {} & 0.25 \times Encapsulation - 0.25 \times Coupling \\ & + 0.5 \times Composition + 0.5 \times Polymorphism \end{aligned} \tag{4.9}$$

$$\begin{aligned} Reusability = {} & -0.25 \times Coupling + 0.25 \times Cohesion \\ & + 0.5 \times Messaging + 0.5 \times DesignSize \end{aligned} \tag{4.10}$$

It is worth noting that there are very interesting differences across the three above presented fitness functions. For example, in the flexibility fitness function the *Polymorphism* is considered a good factor (i.e., a design quality increasing the flexibility) and thus is multiplied by a positive coefficient (0.5). On the contrary, in the understandability one the *Polymorphism* plays a negative role (-0.33 as coefficient), decreasing the fitness function. Also, the *Design Size* represents a positive factor for the reusability of a software system (+0.5 of coefficient) while it is considered a negative factor for the code understandability (-0.33). These observations highlight that *the fitness function is strongly dependent on the goal of the refactoring process*.

As further support to this claim, the fitness function used in [649] and aimed at increasing program testability is, as expected, totally different from those described above: it is represented by just one metric, the Low-level design Similarity-based Class Cohesion (LSCC) defined by Al Dallal and Briand [11].

Finally, a customized fitness function has been used in CODe-Imp to improve the security of software systems [323]. In this case, the fitness function has been defined as a combination of 16 quality metrics, including cohesion and coupling metrics,

[5] The interested reader can find quality models useful to define alternative fitness functions in Chapter 3

design size metrics, encapsulation metrics, composition metrics, extensibility and inheritance metrics [323].

Supported search-based techniques and change operators. In CODe-Imp a variety of local and meta-heuristic search techniques are implemented [658]. O'Keeffe and O'Cinnéide [658] evaluate the performances of four search techniques, namely HC, Multiple ascent HC, SA, and GAs in maximizing the QMOOD understandability previously described. As for GAs, the solution representation previously described (i.e., based on the AST) is considered as the phenotype, while the sequence of refactorings performed to reach that solution is considered as the genotype. The mutation operator simply add one random refactoring to a genotype, while the crossover operator consists of "cut and splice" crossover of two genotypes, resulting in a change in length of the children strings [658].

Empirical Evaluation. The results of the study performed by O'Keeffe and O'Cinnéide [658] indicated that HC and its variant produce the best results. A similar study has also been performed in [659], where the authors compared HC, Multiple ascent HC, and SA in maximizing all three QMOOD functions described above (i.e., *Understandability*, *Flexibility* and *Reusability*). Also this study highlighted the superiority of HC and its variants against the other techniques, with a quality gain in the values of the fitness function of about 7% for *Flexibility*, 10% for *Reusability*, and 20% for *Understandability* as compared to the original design. For this reason the current version of CODe-Imp just supports the HC algorithm and some of its variants.

4.5.2 Other search-based refactoring approaches

Seng et al. [765] proposed a refactoring approach based on GA aimed at improving the class structure of a system. The phenotype consists of a high-level abstraction of the source code and of several model refactorings simulating the actual source code refactorings. The source code model represents classes, methods, attributes, parameters, and local variables together with their interactions, e.g., a method that invokes another method. The goal of this abstraction is simply to avoid the complexity of the source code, allowing (i) an easier application of the refactoring operations and (ii) a simpler verification of the refactoring pre- and post- conditions needed to preserve the system external behavior. The refactorings supported are Move Method, Pull Up Attribute, Push Down Attribute, Pull Up Method, Push Down Method. Note that also in this case each refactoring operation can be undone by another refactoring operation, allowing a complete exploration of the search space.

Concerning the genotype, it consists of an ordered list of executed model refactorings needed to convert the initial source code model into a phenotype. As for the mutation operator, it is very similar to that discussed for the work of O'Keeffe and O'Cinnéide [658], and simply extends a current genome with an additional model refactoring. As for the crossover operator, it combines two genomes by selecting the first *n* model refactorings from parent one and adding the model refactorings of parent two to the genome; *n* is randomly chosen [765].

Given the goal of the refactoring process proposed by Seng et al. [765]—i.e., to improve the class structure of a system—the fitness function is defined, as expected, as a set of quality metrics all capturing class quality aspects, and in particular by two coupling metrics (Response for class and the Information-flow-based-coupling), three cohesion metrics (Tight class cohesion, Information-flow-based-cohesion, and the Lack of cohesion of methods), and a complexity metric (a variant of the Weighted method count).

The evaluation reported in [765] shows that the above described approach executed on an open source software system is able to improve the value of several quality metrics measuring class cohesion and coupling. In particular, the improvements in terms of cohesion go from 31% up to 81%, while the reduction of coupling is between 3% and 87%. Moreover, since the approach is fully automated and does not require any developer interaction, the authors manually inspected the proposed refactoring operations to verify their meaningfulness. They found all of them justifiable.

Jensen and Cheng [438] use GP to identify a set of refactoring operations aimed at improving software design by also promoting the introduction of design patterns. As for the previously discussed approaches their solution representation is a high-level representation of the refactored software design and the set of steps (i.e., refactorings) needed to transform the original design into the refactored design.

The change operators defined in the approach by Jensen and Cheng have been conceived for creating instances of design patterns in the source code. An example of these operators is the *Abstraction*, that constructs a new interface containing all public methods of an existing class, thus enabling other classes to take a more abstract view of the original class and any future classes to implement the interface [438].

As for the fitness function, it awards individuals in the generated population exhibiting (i) a good design quality as indicated by the QMOOD metrics [69] previously described in Section 4.5.1, (ii) a high number of design patterns retrieved through a Prolog query executed on the solution representation, and (iii) a low number of refactorings needed to obtain them. The evaluation reported by Jensen and Cheng [438] shows as the proposed approach, applied on a Web-based system, is able to introduce on average 12 new design pattern instances.

4.6 Conclusions

This chapter described how search-based optimization techniques can support software evolution tasks. Table 4.2 summarizes the works we discussed, reporting for each of them (i) the maintenance activity it is related to, (ii) the objectives it aims at maximizing/minimizing, (iii) the exploited search-based techniques, (iv) a reference to the work. We have identified three main kinds of activities for which search-based techniques can be particularly useful. The first area concerns the identification of modules in software projects, with the aim of keeping an evolving system maintainable, of restructuring existing systems, or even of restructuring applications for particular objectives such as the porting towards limited-resource devices. The second area concerns source code (or model) analysis transformation, aimed at achieving different tasks, e.g., finding a patch for a bug. The third area concerns software refactoring, where on the one hand different kinds of refactoring actions are possible [301] on different artifacts belonging to a software system and, on the other hand, there could be different refactoring objectives, such as improving maintainability, testability, or security.

In summary, software engineers might have different possible alternatives to solve software evolution problems, and very often choosing the most suitable one can be effort prone and even not feasible given the size and complexity of the system under analysis, and in general given the number of possible choices for the software engineers. Therefore, it turns out that finding a solution for many software evolution problems is NP-hard [312] and, even if an automatic search-based approach is not able to identify a unique, exact solution, at least it can provide software engineers with a limited set of recommendations, hence reducing information overload [623]. All software engineers need to do in order to solve a software evolution problem using search-based optimization techniques is to (i) encode the problem using a proper representation, (ii) identify a way (fitness function) to quantitatively evaluate how good is a solution for the problem, (iii) define operators to create new solutions from existing ones (e.g., Genetic Algorithms (GAs) selection, crossover, and mutation operators, or hill climbing neighbor operator), and (iv) finally, apply a search-based optimization techniques, such as GAs, hill climbing, simulated annealing, or others. In some cases, there might not be a single fitness function; instead, the problem might be multi-objective and hence sets of Pareto-optimal solutions are expected rather than single solutions (near) optimizing a given fitness function.

Despite the noticeable achievements, software engineers need to be aware of a number of issues that might limit the effectiveness of automatic techniques—including search-based optimization techniques—when being applied to software evolution problems. First, software development—and therefore software evolution—is still an extremely human-centric activity, in which many decisions concerning design or implementation are really triggered by personal experience, that is unlikely to be encoded in heuristics of automated tools. Automatically-generated solutions to software evolution problems tend very often to be meaningless and difficult to be applied in practice. For this reason, researchers should focus their effort in developing optimization algorithms—for example Interactive GAs [818]—where human

Table 4.2: Search-based approaches discussed in this chapter.

Activity	Objectives	Techniques	Reference
Software Modularization	Maximize Modularization Quality (MQ) [555]	HC, SA, GA	[610]
Software Modularization	Multi-objective for maximizing cohesion, minimizing coupling and number of isolated clusters	NSGA-II	[701]
Software Modularization	Maximize MQ and User Constraints	Interactive GA	[82, 360]
Software Miniaturization	Minimize the footprint of an application	GA	[247]
Software Miniaturization	Select features to include in an application when porting it towards a mobile device maximizing customers' satisfaction and minimizing battery consumption	GA	[17]
Program Transformation	Minimize code complexity	HC, GA	[283]
Model transformation	Derive a target model from a source model by following some rules or principles maiming adequacy, internal coherence, and external coherence	PSO, PSO+SA	[459]
Automatic Software Repair	Maximize the tests passed in a given test suite	GP	[502]
Refactoring	Maximize understandability of source code	GA, HC	[657]
Refactoring	Maximize understandability, flexibility, and reusability of source code	HC, SA, GA	[659]
Refactoring	Maximize program testability	HC	[649]
Refactoring	Maximize software security	HC, SA	[323]
Refactoring	Maximize class cohesion, minimize class coupling, minimize class complexity	GA	[765]
Refactoring	Maximize the presence of design patterns	GP	[438]

evaluations (partially) drive the production of problem solutions. For example, in Section 4.3.4 we have described how such techniques can be applied in the context of software modularization. This, however, requires to deal with difficulties occurring when involving humans in the optimization process: human decisions may be inconsistent and, in general, the process tend to be fairly expensive in terms of required effort. To limit such effort, either the feedback can be asked periodically (see Section 4.3.4), or it could be possible to develop approaches that, after a while, are able to learn from feedback using machine learning techniques [535].

Second, especially when the search space of solutions is particularly large, search-based optimization techniques might require time to converge. This may be considered acceptable for tasks having a batch nature. If, instead, one wants to integrate such heuristics in IDEs—e.g., to continuously provide suggestions to developers [158]—then performance becomes an issue. In such a case, it is necessary to carefully consider the most appropriate heuristic to be used or, whenever possible, to exploit parallelization (which is often possible when using GAs). Last but not least, it is worthwhile to point out that such performance optimization can be particularly desirable when, rather than traditional off-line evolution, one expects automatic run-time system reconfiguration, e.g., in a service-oriented architecture [159] or in scenarios of dynamic evolution such as those described in Chapter 7.6 of this book.

Chapter 5
Mining Unstructured Software Repositories

Stephen W. Thomas, Ahmed E. Hassan and Dorothea Blostein

Summary. Mining software repositories, which is the process of analyzing the data related to software development practices, is an emerging field of research which aims to improve software evolutionary tasks. The data in many software repositories is unstructured (for example, the natural language text in bug reports), making it particularly difficult to mine and analyze. In this chapter, we survey tools and techniques for mining unstructured software repositories, with a focus on information retrieval models. In addition, we discuss several software engineering tasks that can be enhanced by leveraging unstructured data, including bug prediction, clone detection, bug triage, feature location, code search engines, traceability link recovery, evolution and trend analysis, bug localization, and more. Finally, we provide a hands-on tutorial for using an IR model on an unstructured repository to perform a software engineering task.

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_5, © Springer-Verlag Berlin Heidelberg 2014

5.1 Introduction

Researchers in software engineering have attempted to improve software development by mining and analyzing software repositories, such as source code changes, email archives, bug databases, and execution logs [329, 371]. Research shows that interesting and practical results can be obtained from mining these repositories, allowing developers and managers to better understand their systems and ultimately increase the quality of their products in a cost effective manner [847]. Particular success has been experienced with *structured* repositories, such as source code, execution traces, and change logs.

Software repositories also contain *unstructured* data, such as the natural language text in bug reports, mailing list archives, requirements documents, and source code comments and identifier names. In fact, some researchers estimate that between 80% and 85% of the data in software repositories is unstructured [118, 351].

Unstructured data presents many challenges because the data is often unlabeled, vague, and noisy [371]. For example, the Eclipse bug database contains the following bug report titles:

- "`NPE caused by no spashscreen handler service available`" (#112600)
- "`Provide unittests for link creation constraints`" (#118800)
- "`jaxws unit tests fail in standalone build`" (#300951)

This data is *unlabeled* and *vague* because it contains no explicit links to the source code entity to which it refers, or even to a topic or task from some pre-defined ontology. Phrases such as "link creation constraints," with no additional information or pointers, are ambiguous at best. The data is *noisy* due to misspellings and typographical errors ("spashscreen"), unconventional acronyms ("NPE"), and multiple phrases used for the same concept ("unittests", "unit tests"). The sheer size of a typical unstructured repository (for example, Eclipse receives an average of 115 new bug reports per day), coupled with its lack of structure, makes manual analysis extremely challenging and in many cases impossible. Thus, there is a real need for automated or semi-automated support for analyzing unstructured data.

Over the last decade, researchers in software engineering have developed many tools and techniques for handling unstructured data, often borrowing from the natural language processing and information retrieval communities. In fact, this problem has led to the creation of many new academic workshops and conferences, including NaturaLiSE (International Workshop on Natural Language Analysis in Software Engineering), TEFSE (International Workshop on Traceability in Emerging Forms of Software Engineering), and MUD (Mining Unstructured Data). In addition, premier venues such as ICSE (International Conference on Software Engineering), FSE (Foundations of Software Engineering), ICSM (International Conference on Software Maintenance), and MSR (Working Conference on Mining Software Repositories), have shown increasing interest in techniques for mining unstructured software repositories.

In this chapter, we examine how to best use unstructured software repositories to improve software evolution. We first introduce and describe common unstructured software repositories in Section 5.2. Next, we present common tools and techniques for handling unstructured data in Section 5.3. We then explore the state-of-the-art in software engineering research for combining the tools and unstructured data to perform some meaningful software engineering tasks in Section 5.4. To make our presentation concrete, we present a hands-on tutorial for using an advanced IR model to perform the task of bug localization in Section 5.5. We offer concluding remarks in Section 5.6.

5.2 Unstructured Software Repositories

The term "unstructured data" is difficult to define and its usage varies in the literature [105, 557]. For the purposes of this chapter, we adopt the definition given by Manning [557]:

> *"Unstructured data is data which does not have clear, semantically overt, easy-for-a-computer structure. It is the opposite of structured data, the canonical example of which is a relational database, of the sort companies usually use to maintain product inventories and personnel records."*

Unstructured data usually refers to natural language text, since such text has no explicit data model. Most natural language text indeed has latent structure, such as parts-of-speech, named entities, relationships between words, and word sense, that can be inferred by humans or advanced machine learning algorithms. However, in its raw, unparsed form, the text is simply a collection of characters with no structure and no meaning to a data mining algorithm.

Structured data, on the other hand, has a data model and a known form. Examples of structured data in software repository include: source code parse trees, call graphs, inheritance graphs; execution logs and traces; bug report metadata (e.g., author, severity, date); source control database commit metadata (e.g., author, date, list of changed files); and mailing list and chat log metadata (e.g., author, date, recipient list).

We now describe in some detail the most popular types of unstructured software repositories. These repositories contain a vast array of information about different facets of software development, from human communication to source code evolution.

5.2.1 Source Code

Source code is the executable specification of a software system's behavior [514]. The source code repository consists of a number of *documents* or *files* written in one or more programming languages. Source code documents are generally grouped into logical entities called *packages* or *modules*.

While source code contains structured data (e.g., syntax, program semantics, control flow), it also contains rich unstructured data, collectively known as its *linguistic data*:

- Comments in the form of developer messages and descriptions and Javadoc-type comments.
- Identifier names, including class names, method names, and local and global variable names.
- String literals found in print commands and functions.

This unstructured portion of source code, even without the aid of the structured portion, has been shown to help determine the high-level functionality of the source code [482].

5.2.2 Bug Databases

A *bug database* (or *bug-tracking system*) maintains information about the creation and resolution of bugs, feature enhancements, and other software maintenance tasks [770]. Typically, when developers or users experience a bug in a software system, they make a note of the bug in the bug database in the form of a *bug report* (or *issue*), which includes such information as what task they were performing when the bug occurred, how to reproduce the bug, and how critical the bug is to the functionality of the system. Then, one or more maintainers of the system investigate the bug report, and if they resolve the issue, they close the bug report. All of these tasks are captured in the bug database. Popular bug database systems include Bugzilla[1] and Trac[2], although many exist [770].

The main unstructured data of interest in a bug report is:

- *title* (or *short description*): a short message summarizing the contents of the bug report, written by the creator of the bug report;
- *description* (or *long description*): a longer message describing the details about the bug;
- *comments*: short messages left by other users and developers about the bug

5.2.3 Mailing Lists and Chat Logs

Mailing lists (or *discussion archives*), along with the *chat logs* (or *chat archives*) are archives of the textual communication between developers, managers, and other project stakeholders [775]. The mailing list is usually comprised of a set of time-stamped email messages, which contain a *header* (containing the sender, receiver(s),

[1] www.bugzilla.org

[2] trac.edgewall.org

and time stamp), a *message body* (containing the unstructured textual content of the email), and a set of *attachments* (additional documents sent with the email). The chat logs contain the record of the instant-messaging conversations between project stakeholders, and typically contain a series of time-stamped, author-stamped text message bodies [103, 776, 777]. The main unstructured data of interest here are the message bodies.

5.2.4 Revision Control System

A *revision control system* maintains and records the history of changes (or edits) to a repository of documents. Developers typically use revision control systems to maintain the edits to source code. Most revision control systems (including CVS [95], Subversion (SVN) [685]), and Git [110]) allow developers to enter a *commit message* when they commit a change into the repository, describing the change at a high level. These unstructured commit messages are of interest to researchers because taken at an aggregate level, they describe how the source code is evolving over time.

5.2.5 Requirements and Design Documents

Requirements documents, usually written in conjunction with (or with approval from) the customer, are documents that list the required behavior of the software system [514]. The requirements can be categorized as either *functional*, which specify the "*what*" of the behavior of the program, or *non-functional*, which describe the qualities of the software (e.g., reliability or accessibility). Most often, requirements documents take the form of natural language text.

Design documents are documents that describe the overall design of the software system, including architectural descriptions, important algorithms, and use cases. Design documents can take the form of diagrams, such as UML diagrams [303], or natural language text.

5.2.6 Software System Repositories

A *software system repository* is a collection of (usually open source) software systems. These collections often contain hundreds or thousands of systems whose source code can easily be searched and downloaded for use by interested third parties. Popular repositories include SourceForge[3] and Google Code[4]. Each software

[3] `sourceforge.net`

[4] `code.google.com`

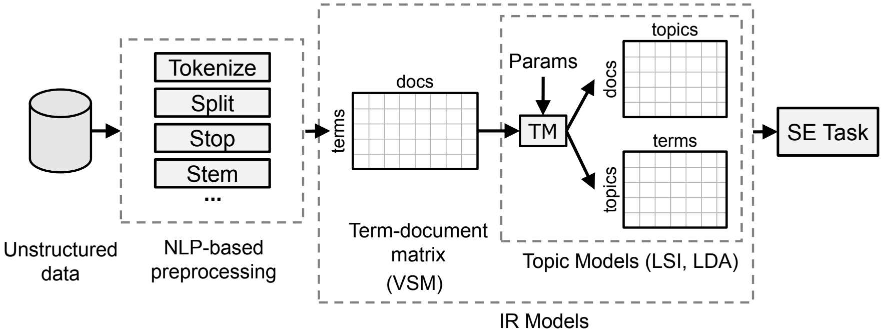


Fig. 5.1: The process of mining unstructured software repositories. First, the unstructured data is preprocessed using one or more NLP techniques. (See Figure 5.2 for an example.) Then, various IR models are built. Finally, the software engineering (SE) task can be performed.

system in the repository may contain any of the above unstructured repositories: source code, bug databases, mailing lists, revision control databases, requirements documents, or design documents.

5.3 Tools and Techniques for Mining Unstructured Data

To help combat the difficulties inherent to unstructured software repositories, researchers have used and developed many tools and techniques. No matter the software repository in question, the typical technique follows the process depicted in Figure 5.1. First, the data is preprocessed using one or more Natural Language Processing (NLP) techniques. Next, an information retrieval (IR) model or other text mining technique is applied to the preprocessed data, allowing the software engineering task to be performed. In this section, we outline techniques for preprocessing data, and introduce common IR models.

5.3.1 NLP Techniques for Data Preprocessing

Preprocessing unstructured data plays an important role in the analysis process. Left unprocessed, the noise inherent to the data will confuse and distract IR models and other text mining techniques. As such, researchers typically use NLP techniques to perform one or more preprocessing steps before applying IR models to the data. We describe *general* preprocessing steps that can be applied to any source of unstruc-

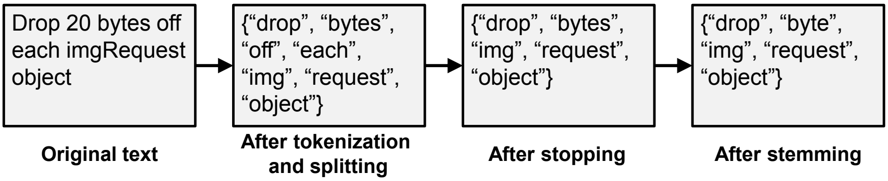

Drop 20 bytes off each imgRequest object
{“drop”, “bytes”, “off”, “each”, “img”, “request”, “object”}
{“drop”, “bytes”, “img”, “request”, “object”}
{“drop”, “byte”, “img”, “request”, “object”}
Original text
After tokenization and splitting
After stopping
After stemming

Fig. 5.2: An illustration of common NLP preprocessing steps. All steps are optional, and other steps exist.

tured data, and then outline more specific preprocessing steps for source code and email.

5.3.1.1 General Preprocessing Steps

Several general preprocessing steps can be taken in an effort to reduce noise and improve the resulting text models built on the input corpus. These steps are depicted in Figure 5.2.

- **Tokenization.** The original stream of text is turned into tokens. During this step, punctuation and numeric characters are removed.
- **Identifier splitting.** Identifier names (if present) are split into multiple parts based on common naming conventions, such as camel case (`oneTwo`), underscores (`one_two`), dot separators (`one.two`), and capitalization changes (`ONETwo`). We note that identifier splitting is an active area of research, with new approaches being proposed based on speech recognition [551], automatic expansion [501], mining source code [274], and more.
- **Stop word removal.** Common English-language stop words (e.g., “the”, “it”, “on”) are removed. In addition to common words, custom stop words such as domain-specific jargon, can be removed.
- **Stemming.** Word stemming is applied to find the morphological root of each word (e.g., “changing” and “changes” both become “chang”), typically using the Porter algorithm [692], although other algorithms exist.
- **Pruning.** The vocabulary of the resulting corpus is pruned by removing words that occur in, for example, over 80% or under 2% of the documents [554].

5.3.1.2 Source Code Preprocessing

If the input data is source code, the characters related to the syntax of the programming language (e.g., “`&&`”, “`->`”) are often removed, and programming language keywords (e.g., “`if`”, “`while`”) are removed. These steps result in a dataset containing only the comments, identifiers, and string literals. Some research also takes steps to remove certain comments, such as copyright comments or unused snippets of code. The main idea behind these steps is to capture the semantics of the developers' intentions, which are thought to be encoded within the identifier names, comments, and string literals in the source code [695].

5.3.1.3 Email Preprocessing

Preprocessing email is an ongoing research challenge [103, 775], due to the complex nature of emails. The most common preprocessing steps include:

- Detecting and removing noise: header information in replies or forwards; previous email messages; and signature lines [825].
- Isolating source code snippets or stack traces [53, 102], so that they can either be removed or treated specially. We discuss this step in more detail in Section 5.4.7.

5.3.1.4 Tools

Many researchers create their own preprocessing tools, based on commonly available toolkits such as the NLTK module in Python [111] and TXL [198] for parsing source code. The authors of this chapter have released their own tool, called *lscp*[5], that implements many of the steps described above.

5.3.2 Information Retrieval

"Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)."

— Manning [557, p. 1]

IR is used to find specific documents of interest in a large collection of documents. Usually, a user enters a query (i.e., a text snippet) into the IR system, and the system returns a list of documents related to the query. For example, when a user enters the query "software engineering" into the Google IR system, it searches every web page ever indexed and returns those that are somehow related to software engineering.

IR *models*—the internal workings of IR systems—come in many forms, from basic keyword-matching models to statistical models that take into account the location of the text in the document, the size of the document, the uniqueness of the matched term, and even whether the query and document contain shared topics of interest [948]. Here, we briefly describe three popular IR models: the Vector Space

[5] github.com/doofuslarge/lscp

Model, Latent Semantic Indexing, and latent Dirichlet allocation. A full treatment of each of these models is beyond the scope of this chapter. Instead, we aim to capture the most essential aspects of the models.

5.3.2.1 The Vector Space Model

The Vector Space Model (VSM) is a simple algebraic model based on the *term-document matrix* A of a corpus [741]. A is an $m \times n$ matrix, where m is the number of unique terms, or words, across n documents. The i^{th}, j^{th} entry of A is the weight of term w_i in document d_j (according to some weighting function, such as term-frequency). VSM represents documents by their column vector in A: a vector containing the weights of the words present in the document, and 0s otherwise. The similarity between two documents (or between a query and a document) is calculated by comparing the similarity of the two column vectors of A. Example vector similarity measures include Euclidean distance, cosine distance, Hellinger distance, or KL divergence. In VSM, two documents will only be deemed similar if they contain at least one shared term; the more shared terms they have, and the higher the weight of those shared terms, the higher the similarity score will be.

VSM improves over its predecessor, the Boolean model, in two important ways. First, VSM allows the use of term weighting schemes, such as tf-idf (term frequency, inverse document frequency) weighting. Weighting schemes help to downplay the influence of common terms in the query and provide a boost to documents that match rare terms in the query. Another improvement is that the relevance between the query and a document is based on vector similarity measures, which is more flexible than the strict Boolean model [741].

Tools. Popular tools implementing VSM include Apache Lucene[6] and gensim[7].

5.3.2.2 Latent Semantic Indexing

Latent Semantic Indexing (LSI) (or *Latent Semantic Analysis* (LSA)) is an information retrieval model that extends VSM by reducing the dimensionality of the term-document matrix by means of *Singular Value Decomposition* (SVD) [232]. During the dimensionality reduction phase, terms that are related (i.e., by co-occurrence) are grouped together into topics. This noise-reduction technique has been shown to provide increased performance over VSM for dealing with polysemy and synonymy [57], two common issues in natural language.

SVD is a factorization of the original term-document matrix A that reduces its dimensionality by isolating its singular values [740]. Since A is likely to be sparse, SVD is a critical step of the LSI approach. SVD decomposes A into three matrices:

[6] lucene.apache.org

[7] radimrehurek.com/gensim

$A = TSD^T$, where T is an m by $r = rank(A)$ term-topic matrix, S is the r by r singular value matrix, and D is the n by r document-topic matrix.

LSI augments the reduction step of SVD by choosing a reduction factor, K, which is typically much smaller than the r, the rank of the original term-document matrix. Instead of reducing the input matrix to r dimensions, LSI reduces the input matrix to K dimensions. There is no perfect choice for K, as it is highly data- and task-dependent. In the literature, typical values range between 50–300.

As in VSM, terms and documents are represented by row and column vectors, respectively, in the term-document matrix. Thus, two terms (or two documents) can be compared by some distance measure between their vectors (e.g., cosine similarity) and queries can by formulated and evaluated against the matrix. However, because of the reduced dimensionality of the term-document matrix after SVD, these measures are well equipped to deal with noise in the data.

Tools. For LSI, popular implementations include *gensim* and R's LSA package.

5.3.2.3 Latent Dirichlet Allocation

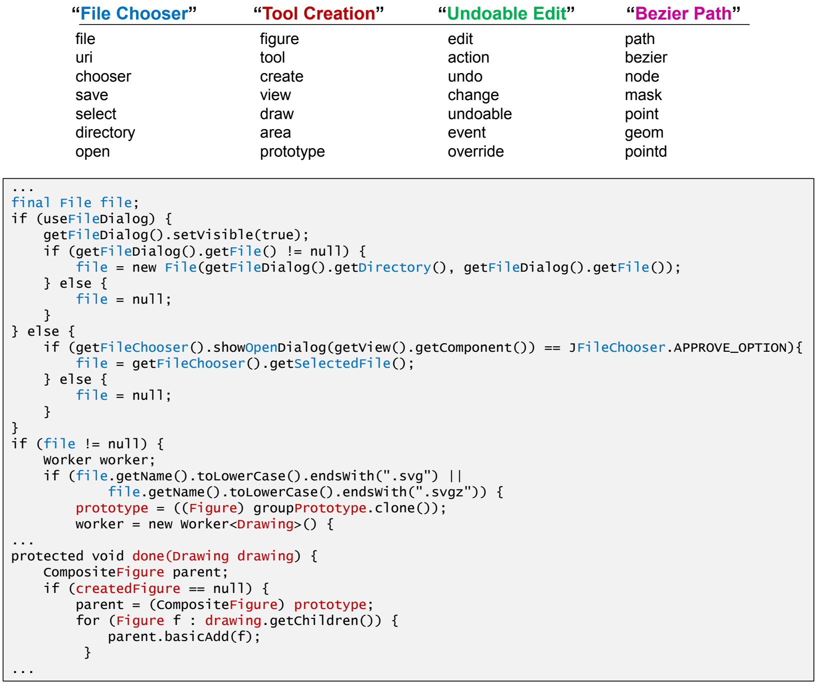
Latent Dirichlet allocation (LDA) is a popular probabilistic topic model [117] which takes a different approach to representing documents than previous models. The main idea behind LDA is that it models each document as a multi-membership mixture of K corpus-wide topics, and each topic as a multi-membership mixture of the terms in the corpus vocabulary. This means that there is a set of topics that describe the entire corpus. Each document is composed of one or more than one of these. Each term in the vocabulary can be contained in more than one of these topics. Hence, LDA is able to discover a set of ideas or themes that well describe the entire corpus [116].

LDA is based on a fully generative model that describes how documents are created. Informally stated, this generative model makes the assumption that the corpus contains a set of K corpus-wide topics, and that each document is comprised of various combinations of these topics. Each term in each document comes from one of the topics in the document. This generative model is formulated as follows:

- Choose a topic vector $\theta_d \sim$ Dirichlet(α) for document d.
- For each of the N terms w_i in d:
  - Choose a topic $z_k \sim$ Multinomial(θ_d).
  - Choose a term w_i from p($w_i|z_k, \beta$).

Here, p($w_i|z_k, \beta$) is a multinomial probability function, α is a smoothing parameter for document-topic distributions, and β is a smoothing parameter for topic-term distributions.

Like any generative model, the task of LDA is that of *inference*: given the terms in the documents, which topics did they come from, and what are the topics themselves? LDA performs inference with *latent variable models* (or *hidden variable models*), which are machine learning techniques devised for just this purpose: to



"File Chooser"	"Tool Creation"	"Undoable Edit"	"Bezier Path"
file	figure	edit	path
uri	tool	action	bezier
chooser	create	undo	node
save	view	change	mask
select	draw	undoable	point
directory	area	event	geom
open	prototype	override	pointd

```
...
final File file;
if (useFileDialog) {
    getFileDialog().setVisible(true);
    if (getFileDialog().getFile() != null) {
        file = new File(getFileDialog().getDirectory(), getFileDialog().getFile());
    } else {
        file = null;
    }
} else {
    if (getFileChooser().showOpenDialog(getView().getComponent()) == JFileChooser.APPROVE_OPTION){
        file = getFileChooser().getSelectedFile();
    } else {
        file = null;
    }
}
if (file != null) {
    Worker worker;
    if (file.getName().toLowerCase().endsWith(".svg") ||
            file.getName().toLowerCase().endsWith(".svgz")) {
        prototype = ((Figure) groupPrototype.clone());
        worker = new Worker<Drawing>() {
...
protected void done(Drawing drawing) {
    CompositeFigure parent;
    if (createdFigure == null) {
        parent = (CompositeFigure) prototype;
        for (Figure f : drawing.getChildren()) {
            parent.basicAdd(f);
         }
...
```

Fig. 5.3: Four example topics (their labels and top words) from JHotDraw 7.5.1. We also show a snippet of the file `SVGCreateFromFileTool.java`, with terms colored corresponding to the topic from which they came. In this example, no terms come from the "Undoable Edit" or "Bezier Path" topics.

associate observed variables (here, terms) with latent variables (here, topics). A rich literature exists on latent variable models [e.g., 74, 113]; for the purposes of this chapter, we omit the details necessary for computing the posterior distributions associated with such models.

Figure 5.3 shows example topics discovered by LDA from version 7.5.1 of the source code of JHotDraw[8], a framework for creating simple drawing applications. For each example topic, the figure shows an automatically-generated two-word topic label and the top (i.e., highest probable) words for the topic. The topics span a range of concepts, from opening files to drawing Bezier paths.

Tools. Implementations of LDA include MALLET [575], gensim, R's LDA and topicmodels packages, lucene-lda, lda-c, and Stanford's topic modeling toolbox[9], amongst others.

[8] `www.jhotdraw.org`

[9] `nlp.stanford.edu/software/tmt`

5.4 The State of The Art

Many software engineering tasks can be addressed or enhanced by incorporating unstructured data. These tasks include concept and feature location, traceability linking, calculating source code metrics, statistical debugging, studying software evolution and trends, searching software repositories, managing bug databases and requirements documents. In this section, we describe how these tasks can be performed using IR models, while also pointing the interested reader to further reading material.

5.4.1 Concept/Feature Location and AOP

The task of *concept location* (or *feature location*) is to identify the parts (e.g., documents or methods) of the source code that implement a given feature of the software system [707]. This is useful for developers wishing to debug or enhance a given feature. For example, if the so-called *file printing* feature contained a bug, then a concept location technique would attempt to find those parts of the source code that implement file printing, i.e., parts of the source code that are executed when the system prints a file.

Concept location is a straightforward application of IR models on source code. The general method is to preprocess the source code as outlined in Section 5.3.1, build an IR model on the preprocessed source code, accept a developer query such as "*file printing*", and use the IR model to return a list of related source code documents. Table 5.1 summarizes various approaches and studies that have been performed in this area. Many IR models have been considered, including VSM, LSI, LDA, and combinations thereof. While most researchers report some success with their concept location methods, there is not yet a consensus as to which IR model performs best under all circumstances.

Related to concept location is *aspect-oriented programming* (AOP), which aims to provide developers with the machinery to easily implement aspects of functionality whose implementation spans over many source code documents. Recently, a theory has been proposed that says software concerns are equivalent to the latent topics found by statistical topic models, in particular LDA [67]. In particular, aspects are exactly those topics that have a relatively high scatter metric value. After testing this theory on a large set of open-source systems, researchers find that this theory holds true most of the time [67].

5.4.2 Traceability Recovery and Bug Localization

An often-asked question during software development is: "*Which source code document(s) implement requirement X?*" *Traceability recovery* aims to automatically uncover links between pairs of software artifacts, such as source code documents and requirements documents. This allows a project stakeholder to trace a requirement to its implementation, for example to ensure that it has been implemented correctly, or at all. Traceability recovery between pairs of source code documents is also important for developers wishing to learn which source code documents are somehow related to the current source code file being worked on. *Bug localization* is a special case of traceability recovery in which developers seek traceability links between bug reports and source code, to help locate which source code files might be related to the bug report.

Typically, an IR model is first applied to the preprocessed source code, as well as the documentation or other textual repository. Next, a similarity score is calculated between each pair of documents (e.g., source code document and documentation documents). A developer then specifies a desired value for the similarity threshold, and any pair of documents with similarity greater than the threshold would be presented.

Table 5.1 outlines related research in this area. LSI has been the IR model of choice for many years, likely due to its success in other domains. Recently, however, multiple IR models have been empirically compared, as outlined in Table 5.1. From this research, we find that LDA is usually reported to achieve better results than LSI, but not in every case. Additional research is required to further determine exactly when, and why, one IR model outperforms another.

5.4.3 Source Code Metrics

Bug prediction (or *defect prediction* or *fault prediction*) tries to automatically predict which entities (e.g., documents or methods) of the source code are likely to contain bugs. This task is often accomplished by first collecting metrics on the source code, then training a statistical model to the metrics of documents that have known bugs, and finally using the trained model to predict whether new documents will contain bugs.

Often, the state-of-the-art in bug prediction is advanced by the introduction of new metrics. An impressive suite of metrics has thus far been introduced, counting somewhere in the hundreds. For example, the *coupling* metric measures how interdependent two entities are to each other, while the *cohesion* metric measure how related the elements of an entity are to each other. Highly coupled entities make maintenance difficult and bug-prone, and thus should be avoided. Highly cohesive entities, on the other hand, are thought to follow better design principles.

The majority of metrics are measured directly on the code (e.g., code complexity, number of methods per class) or on the code change process (methods that are frequently changed together, number of methods per change). Recently, researchers have used IR models to introduce *semantic* or *conceptual* metrics, which are mostly based on the linguistic data in the source code. Table 5.1 lists research that uses IR models to measure metrics on the linguistic data. Overall, we find that LSI-metrics have been used with success, although more recent work reports that LDA-based metrics can achieve better results.

5.4.4 Software Evolution and Trend Analysis

Analyzing and characterizing how a software system changes over time, or the *software evolution* [506] of a system, has been of interest to researchers for many years. Both *how* and *why* a software system changes can help yield insights into the processes used by a specific software system as well as software development as a whole.

To this end, LDA has been applied to several versions of the source code of a system to identify the trends in the topics over time [525, 834, 835]. Trends in source code histories can be measured by changes in the probability of seeing a topic at specific version. When documents pertaining to a particular topic are first added to the system, for example, the topics will experience a spike in overall probability. Researchers have evaluated the effectiveness of such an approach, and found that spikes or drops in a topic's popularity indeed coincided with developer activity mentioned in the release notes and other system documentation, providing evidence that LDA provides a good summary of the software history [832].

LDA has also been applied to the commit log messages in order to see which topics are being worked on by developers at any given time [401, 402]. LDA is applied to all the commit logs in a 30 day period, and then successive periods are linked together using a topic similarity score (i.e., two topics are linked if they share 8 out of their top 10 terms).

LDA has also been used to analyze the Common Vulnerabilities and Exposures (CVE) database, which archives vulnerability reports from many different sources [641]. Here, the goal is to find the trends of each vulnerability, in order to see which are increasing and which are decreasing. Research has found that using LDA achieves just as good results as manual analysis on the same dataset.

Finally, LDA has recently been used to analyze the topics and trends present in Stack Overflow[10], a popular question and answer forum [75]. Doing so allows researchers to quantify how the popularity of certain topics and technologies (e.g.: Git vs. SVN; C++ vs. Java; iPhone vs. Android) is changing over time, bringing new insights for vendors, tool developers, open source projects, practitioners, and other researchers.

[10] www.stackoverflow.com

5.4.5 Bug Database Management

As bug databases grow in size, both in terms of the number of bug reports and the number of users and developers, better tools and techniques are needed to help manage their work flow and content.

For example, a semi-automatic bug triaging system would be quite useful for determining which developer should address a given bug report. Researchers have proposed such a technique, based on building an LSI index on the the titles and summaries of the bug reports [7, 41]. After the index is built, various classifiers are used to map each bug report to a developer, trained on previous bug reports and related developers. Research reports that in the best case, this technique can achieve 45% classification accuracy.

Other research has tried to determine how easy to read and how focused a bug report is, in an effort to measure the overall quality of a bug database. Here, researchers measured the cohesion of the content of a bug report, by applying LSI to the entire set of bug reports and then calculating a similarity measure on each comment within a single bug report [253, 524]. The researchers compared their metrics to human-generated analysis of the comments and find a high correlation, indicating that their automated method can be used instead of costly human judgements.

Many techniques exist to help find duplicate bug reports, and hence reduce the efforts of developers wading through new bug reports. For example, Runeson et al. [737] use VSM to detect duplicates, calling any highly-similar bug reports into question. Developer can then browse the list to determine if any reports are actually duplicates. The authors preprocess the bug reports with many NLP techniques, including synonym expansion and spell correction. Subsequent research also incorporates execution information when calculating the similarity between two bug reports [907]. Other research takes a different approach and trains a discriminative model, using Support Vector Machines, to determine the probability of two bug reports being duplicates of each other [801]. Results are mixed.

Finally, recent work has proposed ways to automatically summarize bug reports, based on extracting key technical terms and phrases [540, 709]. Bug summaries are argued to save developers time, although no user studies have been performed.

5.4.6 Organizing and Searching Software Repositories

To deal with the size and complexity of large-scale software repositories, several IR-based tools have been proposed, in particular tools for organizing and searching such repositories.

MUDABlue is an LSI-based tool for organizing large collections of open-source software systems into related groups, called software categories [454]. MUDABlue applies LSI to the identifier names found in each software system and computes the pairwise similarity between whole systems. Studies show that MUDABlue can achieve recall and precision scores above 80%, relative to manually created tags of the systems, which are too costly to scale to the size of typical software repositories. LACT, an LDA-based tool similar to MUDABlue, has recently been shown to be comparable to MUDABlue in precision and recall [844].

Sourcerer is an LDA-based, internet-scale repository crawler for analyzing and searching a large set of software systems. Sourcerer applies LDA and the Author-Topic model to extract the concepts in source code and the developer contributions in source code, respectively. Sourcerer is shown to be useful for analyzing systems and searching for desired functionality in other systems [528, 529].

S^3 is an LSI-based technique for searching, selecting, and synthesizing existing systems [694]. The technique is intended for developers wishing to find code snippets from an online repository matching their current development needs. The technique builds a dictionary of available API calls and related keywords, based on online documentation. Then, developers can search this dictionary to find related code snippets. LSI is used in conjunction with Apache Lucene to provide the search capability.

5.4.7 Other Software Engineering Tasks

LSI has been used to detect high-level clones of source code methods by computing the semantic similarity between pairs of methods [563]. Related work has used ICA for the same purpose, arguing that since ICA can identify more distinct signals (i.e., topics) than LSI, then the conceptual space used to analyze the closeness of two methods will be of higher effectiveness [345].

Aside from establishing traceability links from requirements to source code (described previously in Section 5.4.2), researchers have proposed many techniques to help manage and use the natural language text in requirements documents. These techniques include generating UML models from requirements [231], detecting conflicts in aspect-oriented requirements [744], identifying aspects in requirements [742], and assessing the quality of requirements[672].

IR methods require many parameter values to be configured before using. Various methods have been proposed to (semi-)automatically tune the parameters for software engineering datasets [80, 346].

Researchers are beginning to consider how discussion forums and question and answer websites might help developers. For example, new tools include finding relevant answers in formats [343], finding experts for a particular question [519], analyzing the topics and trends in Stack Overflow [75], and semi-automatically extracting FAQs about the source code [389].

Methods that incorporate email are receiving more attention from researchers. For example, lightweight IR methods have been used to link emails to their discussed source code entities [55]. In similar work, more heavy-weight classification techniques are used to extract source code from emails, which can be a useful first step for the aforementioned linking methods [54]. This technique was later revised to use island grammars [53]. Finally, spell checkers [104] and clone detection techniques [102] have been used to locate and extract source code in emails.

Statistical debugging is the task of identifying a problematic piece of code, given a log of the execution of the code. Researchers have proposed Delta LDA, a variant of LDA, to perform statistical debugging [33]. Delta LDA is able to model two types of topics: usage topics and bug topics. Bug topics are those topics that are only found in the logs of failed executions. Hence, Delta LDA is able to identify the pieces of code that likely caused the bugs.

LSI has been used as a tool for root cause analysis (RCA), i.e., identifying the root cause of a software failure [132]. The tool builds and executes a set of test scenarios that exercise the system's methods in various sequences. Then, the tool uses LSI to build a method-to-test co-occurrence matrix, which has the effect of clustering tests that execute similar functionalities, helping to characterize the different manifestations of a fault.

Other tasks considered include automatically generating comments in source code [794], web service discovery [931], test case prioritization [833], execution log reduction [946], and analyzing code search engine logs [60, 61].

5.5 A Practical Guide: IR-based Bug Localization

In this section, we present a simple, yet complete tutorial on using IR models on unstructured data to perform bug localization, the task of identifying those source code files that are related to a given bug report. To make the tutorial concrete, we will use the source code and bug reports of the Eclipse IDE[11], specifically the JDT submodule.

To make the tutorial easily digestible, we will make the following simplifying assumptions:

- Only a single version of the source code will be analyzed, in our case, the snapshot of the code from July 1, 2009. Methods exist to analyze all previous versions of the code [832], but require some extra steps.
- Only basic source code representations will be considered. More advanced representations exist [836], such as associating a source code document with all the bug reports with which it has been previously associated.
- For simplicity, we will assume that we have access to all the source code and bug reports on the local hard drive. In reality, large projects may have their documents controlled by content management servers on the cloud, or via some other mechanism. In these cases, the steps below still apply, but extra care must be taken when accessing the data.

The basic steps to perform bug localization include (a) collecting the data, (b) preprocessing the data, (c) building the IR model on the source code, (d) and query-

[11] www.eclipse.org

Listing 5.1: An example Eclipse bug report (#282770) in XML.

```
<bug>
␣<bug_id>282770</bug_id>
␣<creation_ts>2009-07-07␣23:48:32</creation_ts>
␣<short_desc>
␣␣Dead␣code␣detection␣should␣have␣specific␣@SuppressWarnings
␣</short_desc>
␣<long_desc>
␣␣As␣far␣as␣I␣can␣tell␣there␣is␣no␣option␣to␣selectively␣turn␣off
␣␣dead␣code␣detection␣warnings␣using␣the␣@SuppressWarnings
␣␣annotation.␣␣The␣feature␣either␣has␣to␣be␣disabled␣...
␣</long_desc>
</bug>
```

ing the IR model with a particular bug report and viewing the results. We now expand on each step in more detail.

5.5.1 Collect data

Let's assume that that source code of Eclipse JDT is available in a single directory, `src`, with no subdirectories. The source code documents are stored in their native programming language, Java. There are 2,559 source code documents, spanning dozens of packages, with a total of almost 500K source lines of code.

We also assume that the bug reports are available in a single directory, `bugs`, with no subdirectories. Each bug report is represented in XML. (See Listing 5.1 for an example.) The Eclipse JDT project has thousands of bug reports, but in this tutorial we will focus on only one, shown in Listing 5.1.

5.5.2 Preprocess the source code

Several decisions must be made during the phase of preprocessing the source code. Which parts of the source code do we want to include when building the IR model? Should we tokenize identifier names? Should we apply morphological stemming? Should we remove stopwords? Should we remove very common and very uncommon words?

The correct answers are still an active research area, and may depend on the particular project. In this tutorial, we'll assume that identifiers, comments, and string literals are desired; we will tokenize identifier names; we will stem; and finally, we will remove stopwords.

Listing 5.2: Using lcsp to preprocess the source code in the `src` directory.

```
#!/usr/bin/perl
use lscp;
my $preprocessor = lscp->new;
$preprocessor->setOption("isCode", 1);
$preprocessor->setOption("doIdentifiers", 1);
$preprocessor->setOption("doStringLiterals", 1);
$preprocessor->setOption("doComments", 0);
$preprocessor->setOption("doTokenize", 1);
$preprocessor->setOption("doStemming", 1);
$preprocessor->setOption("doStopwordsKeywords", 1);
$preprocessor->setOption("doStopwordsEnglish", 1);
$preprocessor->setOption("inPath", "src");
$preprocessor->setOption("outPath", "src-pre");

$preprocessor->preprocess();
```

To do so, we can use any of the tools mentioned in 5.3.1, such as lcsp or TXL [198], or write our own code parser and preprocessor. In this chapter, we'll use lcsp, and preprocess the source code with the Perl script shown in Listing 5.2. The script specifies the preprocessing options desired, gives the path to the source code (in our case, `src`), and specifies where the resulting files should be placed (here, `src-pre`). After running the Perl script shown Listing 5.2, we'll have one document in `src-pre` for each of the documents in `src`.

5.5.3 *Preprocess the bug reports*

As with source code, several decisions need to be made when preprocessing a bug report. Should we include its short description only, long description only, or both? If stack traces are present, should we remove them? Should we tokenize, stem, and stop the words?

As with preprocessing source code, the best answers to these design decisions are yet unknown. In this tutorial, we'll include both the short and long description; leave stack traces if present; and tokenize, stem, and stop the words. Note that it is usually a good idea to perform the same preprocessing steps on the queries as we did on the documents, as we have done here, so that the IR model is dealing with a similar vocabulary.

To do the preprocessing, we again have a number of tools at our disposal. Here, we'll again use lscp and write a simple Perl script similar to that shown in Listing 5.2. The only differences will be setting the options `isCode` to 0, `inPath` to `bugs`, `outPath` to `bugs-pre`, and removing the `doStringLiterals` and `doComments` options, as the no longer apply. We leave the `doIdentifiers` option, in case the bug report contains snippets of code that contain identifiers.

Before inputting the bug reports into lscp, we must first use a simple XML parser to parse the bug report and strip out the text content from the two elements we desire, `<short_desc>` and `<long_desc>`.

5.5.4 Build the IR model on the source code

In this tutorial, we'll build an LDA model, one of the more advanced IR models. To build the LDA model, we'll use the lucene-lda tool[12], which is a tool to build and query LDA indices from text documents using the Apache Lucene framework.

We use the command prompt to issue the following command, assuming we're in the lucene-tool's base directory:

```
$ ./bin/indexDirectory --inDir src-pre --outIndexDir \
out-lucene --outLDADir out-lda --K 64
```

We pass the `src-pre` directory as the input directory, and specify two output directories, `out-lucene` and `out-lda`, where the Lucene index and LDA output will be stored, respectively. We also specify the number of topics for the LDA model to be 64 in this case, although choosing the optimal number of topics is still an active area of research. We'll use the tool's default behavior for the α and β smoothing parameters of LDA, which will use heuristics to optmize these values based on the data.

The tool will read all the files in the input directory and run LDA using the MALLET toolsuite. MALLET will create files to represent the topic-term matrix (i.e., which terms are in which topics) and the document-topic matrix (i.e., which topics are in which source code documents). The lucene-lda tool will use these files, along with the Lucene API, to build an index that can be efficiently stored and queried, which will be placed in the `out-lucene` directory.

5.5.5 Query the LDA model with a bug report

Now that the index is built, it is ready to be queried. Although we have the ability to query the index with any terms or phrases we desire, in the task of bug localization, the terms come from a particular bug report. Since we've already preprocessed all of the bug reports, we can choose any one and use it as a query to our pre-built index:

```
$ ./bin/queryWithLDA --inIndexDir out-lucene --intLDADir \
out-lda --query bugs-pre/#282770 --resultFile results.dat
```

[12] `github.com/doofuslarge/lucene-lda`

Listing 5.3: Results of querying the LDA model with Eclipse bug report #282770. Each row shows the similarity score between the bug report and a source code document in the index.

```
6.75, compiler.impl.CompilerOptions.java
5.00, internal.compiler.batch.Main.java
4.57, internal.formatter.DefaultCodeFormatterOptions.java
3.61, jdt.ui.PreferenceConstants.java
3.04, internal.compiler.ast.BinaryExpression.java
2.93, core.NamingConventions.java
2.85, internal.ui.preferences.ProblemSeveritiesConfigBlock.java
2.74, internal.formatter.DefaultCodeFormatter.java
2.59, internal.ui.text.java.SmartSemicolonAutoEditStrategy.java
...
```

Here, we specify the names of the directories holding the Lucene index and supporting LDA information, the name of the query (in this case, the preprocessed version of the bug report shown in Listing 5.1), and the file that should contain the results of the query.

The tool reads in the query and the Lucene index, and uses the Lucene API for execute the query. Lucene efficiently computes a similarity score between the query and each document, in this case based on their shared topics. Thus, the tool infers which topics are in the query, computes the *conditional probability* between the query and each document, and sorts the results.

After running the query, the tool creates the `results.dat` file, which contains the similarity score between the query and each document in the index. Listing 5.3 shows the top 10 results for the this particular query, ordered by the similarity score. These top files have the most similarity with the query, and thus should hopefully be relevant for fixing the given bug report. Indeed, as we know from another study [836], the file `internal.compiler.batch.Main.java` was changed in order to fix bug report #282770. We see this file as appearing second in the list in Listing 5.3.

The above result highlights the promising ability of IR models to help with the bug localization problem. Out of the 2,559 source code documents that *may* be relevant to this bug report, the IR model was able to pinpoint the *most relevant file* on its second try.

IR models are not always this accurate. Similar to issuing a web search and not being able to find what you're looking for, IR-based bug localization sometimes can't pinpoint the most relevant file. However, research research has found that IR models can pinpoint the most relevant file to a bug report within the top 20 results up to 89% of the time [836], a result that is sure to aid developers quickly wade through their thousands of source code files.

5.6 Conclusions

Over the past decade, researchers and practitioners have started to realize the benefits of mining their software repositories: using readily-available data about their software projects to improve and enhance performance on many software evolution tasks. In this chapter, we surveyed tools and techniques for mining the *unstructured* data contained in software repositories.

Unstructured data brings many challenges, such as noise and ambiguity. However, as we demonstrated throughout this chapter, many software evolution tasks can be enhanced by taking advantage of the rich information contained in unstructured data, including concept and feature location, bug localization, traceability linking, computing source code metrics to assess source code quality, and many more.

We focused our survey on natural language processing (NLP) techniques and information retrieval (IR) models, which were originally developed in other domains to explicitly handle unstructured data, and have been adopted by the software engineering community. NLP techniques, such as tokenization, identifier splitting, stop word removal, and morphological stemming, can significantly reduce the noise in the data. IR models, such as the Vector Space Model, Latent Semantic Indexing, and latent Dirichlet allocation, are fast and simple to compute and bring useful and practical results to software development teams. Together, NLP and IR models are an effective approach for researchers and practitioners to mine unstructured software repositories.

Research in the area of mining unstructured software repositories has become increasingly active over the past years, and for good reason. We expect to see continued advances in all major areas in the field: better NLP techniques for preprocessing source code, bug reports, and emails; better tailoring of existing IR models to unstructured software repositories; and novel applications of IR models in to solve software engineering tasks.

Table 5.1: Applications of IR models. A (**C**) indicates that the work compares the results of multiple IR models. *Continued in Table 5.2.*

Application	Summary
Concept loc. Markus et al. [564, 565]	First to use LSI for concept location. LSI provides better context than existing concept location methods, such as regular expressions and dependency graphs. Using LSI for concept location in object-oriented (OO) code is useful, contrary to previous beliefs.
Concept loc. (**C**) Cleary et al. [192]	Tests two IR techniques, VSM and LSI, against NLP techniques. NLP techniques do not offer much improvement over the two IR techniques.
Concept loc. van der Spek et al. [791]	Considers the effects of various preprocessing steps, using LSI. Both splitting and stopping improve results, and term weighting plays an important role, but no weighting scheme was consistently best.
Concept loc. Grant et al. [347]	Uses ICA, a model similar to LSI, to locate concepts in source code. The viability of ICA is demonstrated through a case study on a small system.
Concept loc. Compare Models Linstead et al. [526, 527]	Uses LDA to locate concepts in source code. Demonstrates how to group related source code documents based on the documents' topics. Also uses a variant of LDA, the Author-Topic model [731], to extract the relationship between developers (authors) and source code topics. The technique allows the automated answer of "who has worked on what".
Concept loc. Maskeri et al. [571]	Applies LDA to source code, using a weighting scheme for each keyword in the system, based on where the keyword is found (e.g., class name vs. method name). The technique is able to extract business topics, implementation topics, and cross-cutting topics from source code.
Concept loc. Poshyvanyk, Revelle et al. [696, 697, 712, 713]	Combines LSI with various other models, such as Formal Concept Analysis, dynamic analysis, and web mining algorithms (HITS and PageRank). All results indicate that combinations of models outperform individual models.
Traceability Marcus et al. [566]	Uses LSI to recover traceability links between source code and requirements documents. Compared to VSM, LSI performs at least as well in all case studies.
Traceability De Lucia et al [222–224]	Integrates LSI traceability into ADAMS, a software artifact management system. Also proposes a technique for semi-automatically finding an optimal similarity threshold between documents. Finally, performs a human case study to evaluate the effectiveness of LSI traceability recovery. LSI is certainly a helpful step for developers, but that its main drawback is the inevitable trade off between precision and recall.
Traceability (**C**) Hayes et al. [376]	Evaluates various IR techniques for generating traceability links between various high- and low-level requirements. While not perfect, IR techniques provide value to the analyst.
Traceability Lormans et al. [537–539]	Evaluates different thresholding techniques for LSI traceability recovering. Different linking strategies result in different results; no linking strategy is optimal under all scenarios. Uses LSI for constructing *requirement views*. For example, one requirement view might display only requirements that have been implemented.
Traceability Jian et al. [440]	Proposes a new technique, incremental LSI, to maintain traceability links as a software system evolves over time. Compared to standard LSI, incremental LSI reduces runtime while still producing high quality links.

Table 5.2: *Continued from Table 5.1.* Applications of IR models.

Application	Summary
Traceability de Boer et al. [220]	Develops an LSI-based tool to support auditors in locating documentation of interest.
Traceability Antoniol et al. [39]	Introduces a tool, ReORe, to help decide whether code should be updated or rewritten. ReORe uses static (LSI), manual, and dynamic analysis to create links between requirements and source code.
Traceability McMillan et al. [580]	Combines LSI and Evolving Inter-operation Graphs to recover traceability links. The combination modestly improves traceability results in most cases.
Traceability (**C**) Lukins et al. [544, 545]	Compares LSI and LDA for bug localization. After two case studies on Eclipse and Mozilla, the authors find that LDA often outperforms LSI.
Traceability (**C**) Nguyen et al. [644]	Introduces BugScout, an IR model for bug localization, which explicitly considers past bug reports as well as identifiers and comments. BugScout can improve performance by up to 20% over traditional LDA.
Traceability (**C**) Rao et al. [708]	Compares several IR models for bug localization, including VSM, LSI, and LDA, and various combinations. Simpler IR models (such as VSM) often outperform more sophisticated models.
Traceability (**C**) Copabianco et al. [165]	Compares VSM, LSI, Jenson-Shannon, and B-Spline for recovering traceability links between source code, test cases, and UML diagrams. B-Spline outperforms VSM and LSI and is comparable to Jenson-Shannon.
Traceability (**C**) Oliveto et al. [661]	Compares Jenson-Shannon, VSM, LSI, and LDA for traceability. LDA provides unique insights compared to the other three techniques.
Traceability (**C**) Asuncion et al. [49]	Introduces TRASE, an LDA-based tool for prospectively, rather than retrospectively, recovering traceability links. LDA outperforms LSI in terms of precision and recall.
Fault detection Marcus et al. [567]	Introduces C3, an LSI-based class cohesion metric. Highly cohesive classes correlate negatively with program faults.
Fault detection Gall et al. [307]	Presents natural-language metrics based on design and requirements documents. The authors argue that tracking such metrics can help detect problematic or suspect design decisions.
Fault detection Ujhazi et al. [864]	Defines two new conceptual metrics that measure the coupling and cohesion of methods, both based on LSI. The new metrics provide statistically significant improvements compared to previous metrics.
Fault detection Liu et al. [533]	Introduces MWE, an LDA-based class cohesion metric, based on the average weight and entropy of a topic across the methods of a class. MWE captures novel variation in models that predict software faults.
Fault detection Gethers et al. [322]	Introduces a new coupling metric, RTC, based on a variant of LDA called Relational Topic Models. RTC provides value because it is statistically different from existing metrics.
Fault detection Chen et al. [175]	Proposes new LDA-based metrics including: NT, the number of topics in a file and NBT, the number of buggy topics in a file. These metrics can well explain defects, while also being simple to understand.

Chapter 6
Leveraging Web 2.0 for software evolution

Yuan Tian and David Lo

Summary. In this era of Web 2.0, much information is available on the Internet. Software forums, mailing lists, and question-and-answer sites contain lots of technical information. Blogs contain developers' opinions, ideas, and descriptions of their day-to-day activities. Microblogs contain recent and popular software news. Software forges contain records of socio-technical interactions of developers. All these resources could potentially be leveraged to help developers in performing software evolution activities. In this chapter, we first present information that is available from these Web 2.0 resources. We then introduce empirical studies that investigate how developers contribute information to and use these resources. Next, we elaborate on recent technologies and tools that could mine pieces of information from these resources to aid developers in performing their software evolution activities. We especially present tools that support information search, information discovery, and project management activities by analyzing software forums, mailing lists, question-and-answer sites, microblogs, and software forges. We also briefly highlight open problems and potential future work in this new and promising research area of leveraging Web 2.0 to improve software evolution activities.

Much of the content of this chapter is a summary of, and is based on, the following articles having either one or both of the chapter authors as co-authors: [3, 343, 703, 805, 806, 841, 845]. We would like to thank the other co-authors of these articles: Swapna Gottipati, Jing Jiang, Ferdian Thung, Lingxiao Jiang, Didi Surian, Nian Liu, Hanghang Tong, Ee-Peng Lim, Christos Faloutsos, Hong Cheng, Achananuparp Palakorn, and Philips Kokoh Prasetyo.

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_6, © Springer-Verlag Berlin Heidelberg 2014

6.1 Introduction

Web 2.0 has revolutionized the use of web sites [667]. Prior to Web 2.0, most web sites were simply static pages that did not support much user interactions. With Web 2.0, users can post new content dynamically, update a long list of friends in real time, collaborate with one another, and more. This has changed the paradigm of how users use the Web. Web 2.0 sites include but are not limited to blogs, microblogging sites, social networking sites, and sharing and collaboration sites. Currently, most web users spend a substantial amount of time in Web 2.0 sites and the adoption of Web 2.0 sites is growing [235]. Much knowledge is shared in these Web 2.0 sites.

Web 2.0 also affects software developers. Currently, developers share a lot of information in Web 2.0 sites. These resources could be leveraged to help developers perform their tasks better. It is common for developers to consult information sharing sites like software forums, e.g., CNET[1], Oracle OTN forum[2], SoftwareTipsandTricks[3], and DZone[4]. Developers often encounter the same problems, and solutions found by one developer are disseminated to others via these sites. Developers of open source projects such as GNOME[5] use mailing lists as discussion forums to support communication and cooperation in project communities. Besides software forums and mailing lists, developers often turn to question-and-answer sites, such as StackOverflow[6], to seek help from other expert developers about software-related problems that they face. Blogging services are also popular among software developers. Developers use blogs to record knowledge including their ideas and experience gained during software evolution tasks. This knowledge could be discovered by other developers through web search. In the recent few years, with the advent of social media, developers have yet another means to share and receive information. Much information is shared via microblogging sites such as Twitter [863] and Sina Weibo [7]. Information shared in microblogging sites is often recent and informal. The unique nature of information in these sites makes them interesting resources that augment other Web 2.0 resources. Developers also use many collaboration sites to jointly work together to complete various software projects. These collaboration sites, also referred to as software forges, can host tens of thousands of projects or even more. Examples of these collaboration sites (or software forges) are GitHub[8] and SourceForge[9].

Web 2.0 can be leveraged to assist developers find information that help them in software evolution tasks. Developers often want to find answers to various develop-

[1] forums.cnet.com

[2] forums.sun.com/index.jspa

[3] www.softwaretipsandtricks.com/forum

[4] java.dzone.com

[5] www.gnome.org

[6] stackoverflow.com

[7] www.weibo.com

[8] github.com

[9] sourceforge.net

ment questions. Software forums often contain such answers. However, it can take much time for developers to sift the mass of contents shared there to find desired answers. Developers also often want to reuse programs satisfying some properties. Information stored in Web 2.0 resources can be leveraged for this task. For the above mentioned tasks, automation is needed to bring relevant pieces of information to developers.

Developers can also leverage Web 2.0 to discover new and interesting knowledge. For example, developers often want to be notified of new patches to security loop holes, new useful libraries, new releases of some libraries, new programming tips, and many other pieces of information. Today, developers often need to manually search for such new information by leveraging search engines or reading slightly outdated information from magazines or books. The more recent these pieces of information are, the more useful they are to developers, e.g., developers could quickly patch new security loop holes before they get exploited. For these purposes, microblogging sites are promising sources of information as information shared there is often recent and informal, providing timely and honest feedback on many recent issues that various developers find interesting.

Developers often work together in various open source projects hosted in many software forges.[10] Based on projects that developers have worked on, we can build various support tools that improve the management of projects in these software forges [805, 806]. These project management support tools can help to better allocate resources (i.e., developers) to tasks (i.e., projects) [805] or predict bad situations (e.g., project failures) such that appropriate mitigation actions can be taken [806]. In practice, often such project management tasks are done manually and thus support tools could reduce the amount of manual effort needed.

In this chapter, we first present information available in various Web 2.0 resources in Section 6.2. We then describe how developers use these Web 2.0 resources in Section 6.3. Next, we review recently proposed automated techniques that leverage Web 2.0 resources for information search, information discovery, and project management. We start by summarizing past studies that leverage Web 2.0 for information search including studies on answering software development questions and searching for similar projects [343, 841] in Section 6.4. We then summarize past studies that leverage Web 2.0 sites for information discovery, e.g., [3, 703], in Section 6.5. We also present past studies that leverage Web 2.0 to support project management, e.g., [805, 806], in Section 6.6. We describe open problems and future work in Section 6.7. Finally, we conclude in Section 6.8.

[10] Please refer to Chapter 10 of this book

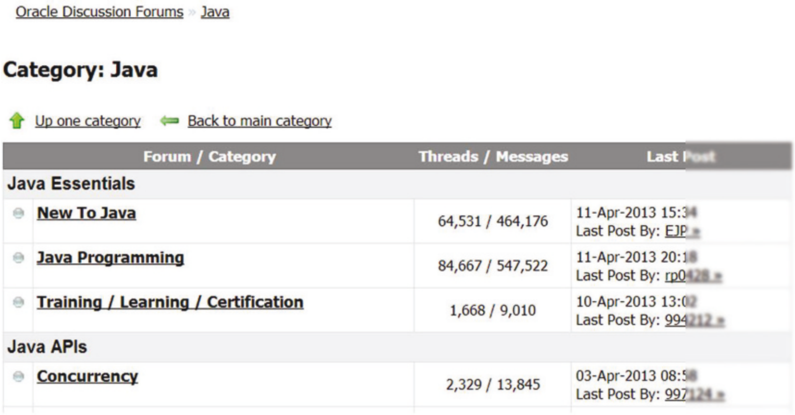
6.2 Web 2.0 Resources

There are various Web 2.0 resources that software developers often use to learn and exchange information and knowledge. In this section, we highlight several of these resources.

6.2.1 Software Forums, Mailing Lists and Q&A Sites

Developers often ask and discuss questions in software forums. Other more experienced developers that face the same problems can reply with some answers. These exchanges of information are documented and thus other developers facing similar problems in the future can also benefit. There are many software forums available online. Some forums are specialized such as Oracle OTN forum and DZone. Some others are general purpose such as CNET and SoftwareTipsandTricks. Figure 6.1 shows the Oracle OTN Forum where many people discuss Java programming.

A software forum is often organized into categories. For example, Figure 6.1 shows that inside Oracle OTN forum, questions and answers are categorized into: Java Essentials, Java API, etc. Inside Java Essentials, there are more sub-categories: New to Java, Java Programming, and Training / Learning / Certification. Within each sub-category, there are many threads. Within each thread, there are many posts. At times a thread can contain even hundreds of posts. The posts contain questions, answers, or other pieces of information (e.g., positive feedbacks, negative feedbacks, junk, etc).



Oracle Discussion Forums » Java

Category: Java

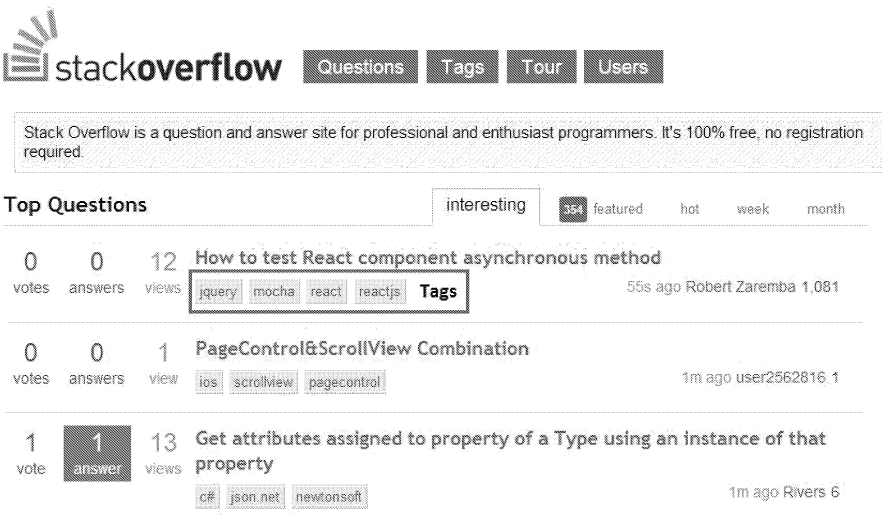
Up one category ⇦ Back to main category

Forum / Category	Threads / Messages	Last [redacted]
Java Essentials		
New To Java	64,531 / 464,176	11-Apr-2013 15:3[redacted] Last Post By: EJP[redacted]
Java Programming	84,667 / 547,522	11-Apr-2013 20:1[redacted] Last Post By: rp0[redacted]
Training / Learning / Certification	1,668 / 9,010	10-Apr-2013 13:0[redacted] Last Post By: 994[redacted]
Java APIs		
Concurrency	2,329 / 13,845	03-Apr-2013 08:5[redacted] Last Post By: 997[redacted]

Fig. 6.1: **Oracle OTN Forum**

Developers of open source projects use mailing lists to communicate and collaborate with one another. A *mailing list* works as a public forum for developers or users who have subscribed to the list. Anyone in the list can post messages to other people in the list by sending emails to a public account. Contents of these messages are usually related with changes made by developers or problems faced by developers or users during software development or product usage. For example, from *GNOME* website developers and users can get information about each mailing list and decide whether to subscribe to one or more lists.[11] These lists are created for various purpose: some are created for messages related to particular modules (e.g., “anjuta-devel-list” is for messages related to Anjuta IDE), some are created for messages related to special events (e.g., “asia-summit-list” is for messages related to GNOME.Asia Summit organization), some are created not for developers but for end users (e.g., “anjuta-list” is for messages from users of Anjuta IDE), etc.

Developers also can seek answers for questions from question-and-answer sites. In general question-and-answer sites like Yahoo! Answers, people can ask questions about various domains including news, education, computer & internet, etc. StackExchange[12] is a fast-growing network which contains 104 domain specific question-and-answer sites focusing on diverse topics from software programming to mathematics and IT security. Among the 104 question-and-answer sites, *StackOverflow* is the biggest and most famous one. It has more than 5 millions questions (most of them are related to software development) and more than 2 millions users since it was launched in 2008.



Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free, no registration required.

0 votes 0 answers 12 views How to test React component asynchronous method
jquery mocha react reactjs Tags
55s ago Robert Zaremba 1,081

0 votes 0 answers 1 view PageControl&ScrollView Combination
ios scrollview pagecontrol
1m ago user2562816 1

1 vote 1 answer 13 views Get attributes assigned to property of a Type using an instance of that property
c# json.net newtonsoft
1m ago Rivers 6

Fig. 6.2: **Question-and-Answer Threads in Stack Overflow**

[11] mail.gnome.org/mailman/listinfo

[12] stackexchange.com

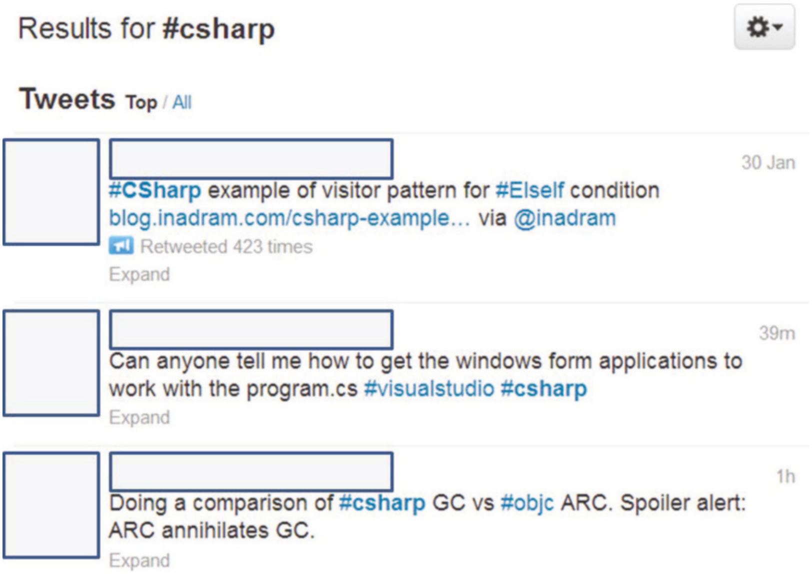
Figure 6.2 shows question-and-answer threads extracted from StackOverflow. Each question in StackOverflow has a short title that briefly summarizes the question. A user who asks the question can assign several *tags* like "java" according to the topics of the question. These tags are used to help users to search for similar questions and their answers. A user who views a question can vote up the question if he/she thinks the question is useful or vote down it if he/she thinks the question is unclear. The sum of votes, the number of people who has viewed the question, and the total number of provided answers are recorded for each question.

6.2.2 Software Blogs & Microblogs

Blogging is one of the typical features of the Web 2.0 era. Similar with home pages, *blogs* are created for individuals. They record personal thinking, experience and stories in a diary-like format. Different from home pages, blogs' contents change more often and blogs support discussions by allowing others to post comments. In addition, the *RSS (Really Simple Syndication)* technology allows people to not only link to a page containing a blog but also to subscribe to it. People who have subscribed to a blog will be informed if the blog's content has been modified.

Developers are using blogs to share their ideas, knowledge, and experiences on software development. Usually people find others' blogs through web search. For instance, a developer has encountered a problem but lacks experience to solve the problem; he or she might go to Google Search and seek for solutions using some keywords. Some of the top returned search results may link to other developers' blogs where the ways to solve the same or similar problem are presented. By this means, blogs provide knowledge for the whole community of software developers.

In the recent years, microblogging services are getting popular. Millions of people communicate with one another by broadcasting short messages which are made public to all. Different from traditional blogging services and other social media, microblogs are usually short and often contain information of very recent news and events; microblogs are also informal in nature and microbloggers are often unafraid to express their honest opinions about various topics. Software developers also make use of this new social media trend. Thus, one could potentially discover various information from microblogs, e.g., new features of a library, new methodologies to develop software systems, new conferences, new security loop holes, etc. The informal and timely nature of microblogs suit software development well. In software development, many new "events", e.g., releases of new libraries, etc., occur from time to time. Developers could learn from the wisdom of the masses that are available in the millions of microblogs about various software relevant topics. Furthermore, a number of studies have shown the important role of informal communication [108, 343, 675]. Microblogging is yet another kind of informal communication. Microbloggers can express honest opinions about various libraries, programming languages, etc. through their microblogs that are available for all to see.



Results for #csharp

Tweets Top / All

[redacted] [redacted] 30 Jan
#CSharp example of visitor pattern for #ElseIf condition blog.inadram.com/csharp-example... via @inadram
Retweeted 423 times
Expand

[redacted] [redacted] 39m
Can anyone tell me how to get the windows form applications to work with the program.cs #visualstudio #csharp
Expand

[redacted] [redacted] 1h
Doing a comparison of #csharp GC vs #objc ARC. Spoiler alert: ARC annihilates GC.
Expand

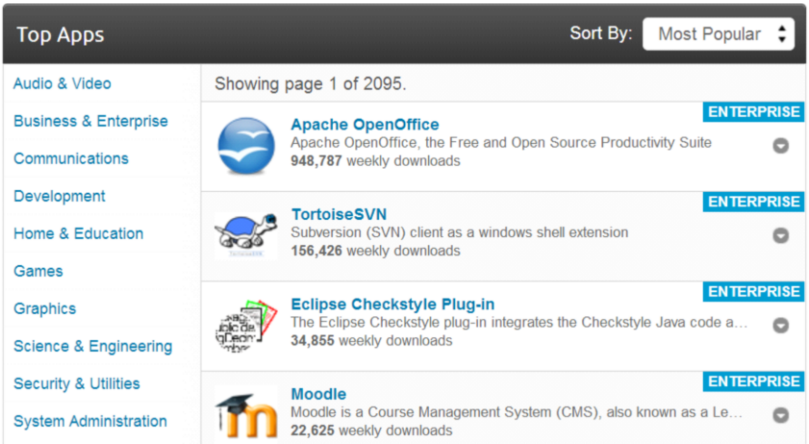
Fig. 6.3: **Microblogs in Twitter**

Figure 6.3 shows some sample microblogs (a.k.a. tweets) from Twitter, which is arguably the largest microblogging site. Microbloggers can post short contents (at most 140 characters in Twitter) that would then be broadcasted to those that subscribe to it. These microblogs are also publicly available for all to see. A microblogger can subscribe to (i.e., follow in Twitter) other microbloggers and get notified whenever new microblogs are generated. A microblogger can also forward microblogs to (i.e., retweet in Twitter) others, as well as reply to others' microblogs. Microblogs can be tagged with particular keywords (i.e., *hashtags* in Twitter). For instance, the microblogs in Figure 6.3 are all tagged with hashtag #csharp.

Developers can include various contents in their microblogs. For example, from Figure 6.3, the first microblogger shares a tip on visitor pattern. The second microblogger asks a question, while the third microblogger broadcasts a personal message on what he is currently doing.

6.2.3 Software Forges

With the advent of Web 2.0, developers can choose to work on various projects with many collaborators across the globe. Software forges provide additional support for this. A software forge, e.g., Google Code, SourceForge, GitHub, etc., hosts hundreds or even hundreds of thousands of projects. Developers can view these projects, be aware of development activities happening in them, download and use the projects, and also contribute code, test cases, bug reports, etc. to the projects. Figure 6.4 shows some projects that are hosted in SourceForge.



Showing page 1 of 2095.

Apache OpenOffice
Apache OpenOffice, the Free and Open Source Productivity Suite
948,787 weekly downloads

TortoiseSVN
Subversion (SVN) client as a windows shell extension
156,426 weekly downloads

Eclipse Checkstyle Plug-in
The Eclipse Checkstyle plug-in integrates the Checkstyle Java code a...
34,855 weekly downloads

Moodle
Moodle is a Course Management System (CMS), also known as a Le...
22,625 weekly downloads

Fig. 6.4: **Projects in SourceForge**

Software forges often provide *APIs* for others to query activities that happen within them. With these APIs, much information can be gathered. We can track how developers work with others across time. We can track the number of downloads. We can also track new projects and new collaborations that are created over time. Chapter 10 of this book describes the evolution of projects in software forges.

6.2.4 Other Resources

There are also other Web 2.0 resources like LinkedIn[13], Facebook[14], Wikipedia[15], Academic.edu[16], Foursquare[17], Google Talk[18], and many more. Many of these resources often contain information about subgroups devoted to specific topics including software evolution. For example LinkedIn and Facebook contain profiles of many software developers. Wikipedia defines many software evolution related terms. Academic.edu shares research studies including those related to software evolution. Foursquare provides geospatial locations of people including those of

[13] www.linkedin.com

[14] www.facebook.com

[15] www.wikipedia.org

[16] academia.edu

[17] foursquare.com

[18] support.google.com/talk/?hl=en

software developers. These resources also provide a wealth of information that can potentially be leveraged to improve software evolution activities.

6.3 Empirical Studies

In this section, we review several empirical studies that investigate how Web 2.0 resources have been used by software developers. We categorize these studies according to resources that they target, namely: software forums, mailing lists & question-and-answer sites, software blogs & microblogs, and software forges.

6.3.1 Software Forums, Mailing Lists and Q&A Sites

Rupakheti and Hou analyzed 150 discussion threads from Java Swing forum [738]. They found that *API problems* recur in software forums. This phenomenon was leveraged to design an *API critic* which advises how an API should be used. They manually categorized the threads into unclear threads, other threads (not related to layout and composition), application specific requirement threads, and threads that would benefit from the automated API critic.

Bird et al. constructed a *social network* from a mailing list [108]. Each node in the network is an individual and there is an edge between *a* to *b* if *b* replied to a message that is generated by *a*. They analyzed more than 100,000 messages from *Apache HTTP Server's* developer mailing list. They found that the out-degree (i.e., the number of people that replies to a person) and in-degree (i.e., the number of people to whom a person has replied to) follow *power-law distributions*. They also found that the level of activity in a mailing list is strongly related to the level of activity in the source code.

Sowe et al. investigated knowledge sharing interactions among knowledge providers and knowledge seekers in the mailing lists of the Debian project [789]. A knowledge provider is an expert that helps other participants in the project. A knowledge seeker refers to any participant who asks questions related to software development or software usage. They collected messages and replies generated by 3735 developers and 5970 users. They found that knowledge providers and knowledge seekers interact and share knowledge a lot. Developers generate more replies than messages while users generated more messages than replies.

Treude et al. analyzed how developers use question-and-answer sites like *StackOverflow* [854]. The authors collected 15 days of question-and-answer threads and manually analyzed a smaller set of 385 questions that are randomly sampled from the collected threads. They divided these questions into different groups and found that questions that ask for instructions are the most popular questions. They also found that questions that ask for code review, the questions made by novices, and abstract questions are answered more frequently than other types of questions.

Nasehi et al. compared high quality and low quality answers on *StackOverflow* to learn important attributes of good code examples [634]. An answer is of high quality if it has been accepted by the asker or it has a relatively high voting score. The authors sampled 163 question-and-answer threads. Each of them has at least one answer that contains a code example and receives 4 or more points. They summarized that high quality answers usually contain concise code example, use the context of the question, highlight important elements, give step-by-step solutions, and provide links to extra resources.

6.3.2 Software Blogs & Microblogs

Pagano and Maalej investigated how software developers use blogging services [675]. They collected blog posts generated by 11,00 developers from four open source project communities, namely Eclipse, *GNOME*, PostgreSQL, and Python. The authors matched the bloggers' identities to source code committers. They found that blogggers who are also committers post more frequently than single bloggers who never commit changes to the source code. They reported that commits are frequently, short, and precise while blog posts are less frequent, longer (14 times longer than commit), and contain less source code. They also found that developers are more likely to post blogs after corrective engineering or management tasks than after forward engineering or re-engineering tasks.

Parnin and Treude studied blog posts that are related to *API documentations* [680]. In their work, they collected developers' blog posts by performing Google searches for all methods' names in the jQuery API. They then manually categorized the top-10 returned search results and found that blog posts cover 87.9% of the API methods. They also found that tutorials and experience reports are the most common types of blog posts. A tutorial often describes problems to be solved and shows solutions in detailed steps. An experience report describes the experience gained from handling a problem. Different from the findings reported by Pagano and Maalej that only 1.8% of blog posts contain source code, this work found that 90% of the blog posts mentioning API methods contain code snippets. The authors concluded that these API related blog posts are used to: describe a philosophy of a design approach or a problem, support a niche community, and store information for bloggers' future personal use.

Parnin et al. investigated the motivation and challenges of blogging developer knowledge [681]. They extracted 55 blogs by performing Google searches using keywords related to three technology areas: *IDE plugin* development, mobile development, and web development.[19] They sent a survey to each of the authors of the 55 blogs and collected 30 responses in the end. They found that developers blog because it can help them educate employees, gain personal reputation, document their experiments, and get feedback that can be used to improve their code or product.

[19] They choose Eclipse and Visual Studio plugins, Android and iPhone development, and Django and jQuery development as representatives.

They also summarized that the biggest challenges for developers to use blogging services are time and the lack of a reward system. For instance, it takes much time for authors to write a high quality blog; it is also time consuming to manage all blog posts, such as porting blogs between systems and filter spam comments.

Bougie et al. conducted an empirical work to understand how developers use Twitter to support communication in their community and what they talk about on Twitter [136]. They sampled 68 developers from three project communities: Linux, Eclipse, and MXUnit. By analyzing 600 microblogs generated by these 68 microbloggers and comparing them with microblogs generated by normal Twitter users, they found that microblogs generated by sampled developers contain more conversations and information sharing. They categorized these 600 microblogs into four categories: software engineering-related, gadgets and technological topics, current events outside technical topics, and daily chatter.

We and a few others extended the work by Bougie et al.'s [845]. We analyzed 300 microblogs (a.k.a. tweets) that are tagged with software related *hashtags* (i.e., #csharp, #java, #javascript, #dotnet, #jquery, #azure, #scrum, #testing, and #opensource). Compared with Bougie et al's sampling method that extracts all microblogs generated by a special group of developers, these 300 microbloggs are more relevant to software development. We manually analyzed the contents of the 300 microblogs and categorized them into ten categories: commercials, news, tools&code, q&a, events, personal, opinions, tips, jobs, and miscellaneous. We found that jobs, news, and q&a are the top 3 most popular categories. We also calculated the percentages of microblogs that are retweeted for each category and found that the most diffused microblogs are from events and commercials categories. Some examples are: "... vote for Superdesk in Ashoka Changemakers Global Innovation Contest. ..." (events), "... GlobalStorage for #dotnetnuke 6 #azure, ... is 15% OFF ..." (commercials). Personal microblogs also get retweeted. The least diffused categories, aside from miscellaneous, are: tools&code, jobs, and q&a. Although these tweets are many in number, they are not widely diffused in the Twitter network.

6.3.3 Software Forges

Madey et al. analyzed open source projects that are hosted in SourceForge [552]. They analyzed 39,000 projects which are developed by more than 33,000 developers. They created a collaboration *social network* where developers are nodes and collaborations among developers (i.e., two or more developers work on the same project) are edges. A modified spanning tree algorithm was used to extract clusters (i.e., groups) of connected developers. Based on this collaboration social network they found that *power-law relationships* exist for project sizes (i.e., number of developers in a project), project memberships (i.e., number of projects that a developer joins), and cluster sizes (i.e., number of developers in a cluster).

Xu et al. investigated *social network properties* of projects and developers in SourceForge [932]. They found that the networks exhibit small world phenomena and are scale free. Small world phenomenon refers to a situation where each node in a network is connected to other nodes in the network by a small number of intermediary nodes. Scale free network refers to a situation where degree distribution of nodes follows a *power-law distribution*. For scale free networks, preferential attachment (i.e., probability of a new node to link to an existing node is proportional to the degree of the existing node) exists.

Ricca and Marchetto investigated 37 randomly selected projects in Google Code and SourceForge [716]. They investigated "heroes" in these projects; heroes refer to important developers that have critical knowledge on particular parts of a software system. They found that heroes are a common phenomenon in open source projects. They also reported that heroes are faster than non-heroes in completing change requests.

Dabbish et al. investigated a different software forge namely GitHub [210]. Different from SourceForge, GitHub is more transparent, i.e., other developers can track and follow the activities of other developers or changes made to a project. They interviewed a set of GitHub users to investigate the value of transparency. They found that transparency is beneficial for various reasons including: developer recruitment, identification of user needs, management of incoming code contributions, and identification of new technical knowledge.

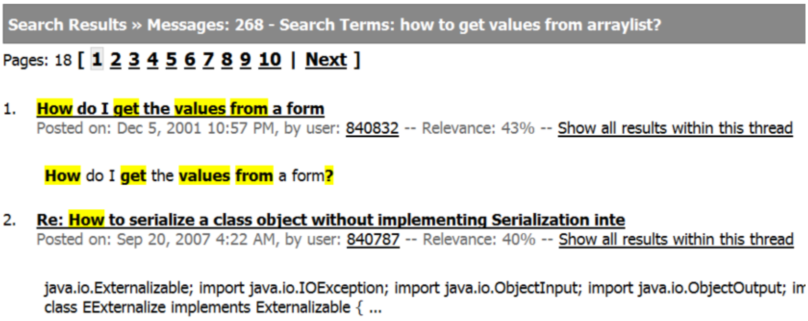
6.4 Supporting Information Search

In this section, we describe several studies that leverage Web 2.0 to support information search. We first describe two of our previous studies that consider two information search scenarios, namely searching for answers in software forums [343], and searching for similar applications in software forges [841]. We then highlight other studies.

6.4.1 Searching for Answers in Software Forums

Motivation. A thread in a software forum can contain a large number of posts. Our empirical study on 10 software forums found that a thread can contain up to 10,000 posts [343]. Scanning for relevant posts in these threads can be a painstaking process. Likely many posts are irrelevant to a user query. Some posts answer irrelevant questions. Some other posts are relevant but do not provide an answer to the problem that a developer has in mind. Furthermore, even after an exhaustive investigation, there might be no post that answers relevant questions or a correct answer might not have been provided in the forum.

To aid in searching for relevant answers, developers typically make use of general purpose *search engines* (e.g., Google, Bing, etc.) or customized search engines available in software forums. General purpose search engines return many webpages. Often many of them are not relevant to answer the questions that developers have in mind, e.g., searching for Java might return the island Java in Indonesia or the Java programming language. Customized search engines are likely to return more relevant results however the number of returned results can still be too many. For example, consider searching the Oracle forum with the following question: “How to get values from an arraylist?”. Figure 6.5 shows the returned results. There are 286 threads returned and some threads contain as many as 30 posts. Developers would then need to manually investigate and filter returned results to finally recover posts that answer the question. This could be very time consuming. Thus, we need a more advanced solution to help find relevant posts from software forums.



Search Results » Messages: 268 - Search Terms: how to get values from arraylist?

Pages: 18 [1 2 3 4 5 6 7 8 9 10 | Next]

1. How do I get the values from a form
   Posted on: Dec 5, 2001 10:57 PM, by user: 840832 -- Relevance: 43% -- Show all results within this thread

   How do I get the values from a form?

2. Re: How to serialize a class object without implementing Serialization inte
   Posted on: Sep 20, 2007 4:22 AM, by user: 840787 -- Relevance: 40% -- Show all results within this thread

   java.io.Externalizable; import java.io.IOException; import java.io.ObjectInput; import java.io.ObjectOutput; ir
   class EExternalize implements Externalizable { ...

Fig. 6.5: Search results (268 of them) from Oracle forum for query: “ How to get values from arraylist?”

Approach. Our proposed approach first labels posts in software forums with predefined *tags*; it then uses these tags to return relevant answers from threads in software forums. It utilizes two main components: tag inference engine and *semantic search engine*. Our tag inference engine automatically classifies posts in software forums with one of the following categories: answers, clarifying questions, clarifying answers[20], positive feedback, negative feedback, and junk (e.g., “today is Friday”). With the inferred tags, developers could focus on the answers that can be hidden deep inside long threads rather than investigating all the posts. With the inferred tags, questions with correct answers (identified based on the corresponding positive feedback) can also be identified. Our semantic search engine enhances standard search engine by making use of the inferred semantic tags to return more relevant answers.

To build a tag inference engine that classifies posts into the seven categories, we follow these steps:

1. We represent each post as a feature vector. To do this, we extract the text in the post and record the author of the post. The textual content of the post is then subjected to the following pre-processing steps: *stopword removal (i.e., removal of*

[20] Answers to clarifying questions.

non-descriptive words) and *stemming (i.e., reduction of a word to its root form)*. For example, the words "reads" and "reading" are reduced to "read". The resultant words are then weighted based on their *term frequency (i.e., the number of times the words appear in the post)*. These together with the author information are used as features (a.k.a. a feature vector) that represent a post.

2. Given a set of posts and their labels, we train a *machine learning* model that discriminates posts belonging to each of the 7 categories using Hidden Markov Support Vector Machine SVM^{HM}) [443]. We take the representative feature vectors that characterize the training set of posts to train this machine learning model. SVM^{HM} classifies a post not only based on its content and author but also the previous few posts. This is particularly effective as the category of a post is often dependent on the category of the previous few posts, e.g., if the previous post is a question, the next post is likely to be an answer or a clarifying question rather than a feedback.

The learned categories could be used to help conventional search engines. A conventional search engine takes in a set of documents (i.e., forum posts in our settings), pre-processes each document into a bag of words, and indexes each document. When a user enters a query, the index is used for fast retrieval of relevant documents in the document corpus. We enrich conventional search engines by leveraging the semantic information available from the inferred tags. To create this semantic search engine, we embed our tag inference engine to infer tags of the posts in the document corpus. These tags are then used to filter irrelevant posts, e.g., junk. Only documents that are potentially relevant would be returned.

Experiments. Three different forums are analyzed in our experiments: SoftwareTipsandTricks[21], DZone[22], and Oracle[23]. We infer the labels of 6068 posts from the forums manually - 4020, 680, and 1368 posts are from SoftwareTipsandTricks, DZone, and Oracle, respectively. Approximately half of the posts are used for training (i.e., 2000, 300, 620 posts from SoftwareTipsandTricks, DZone, and Oracle, respectively) and the remaining posts are used for testing. We build a search engine corpus using the same sets of posts and consider a set of 17 software queries.[24] We compare our semantic search engine with a standard information retrieval toolkit [25]. We consolidate results returned by the standard information retrieval toolkit and our semantic search engine. The consolidated results are then given to five human evaluators who would give a rating of 2, for correct answers, 1, for partially correct answers, and 0, for irrelevant answers.

We first evaluate the accuracy of our tag inference engine in terms of *precision*, *recall*, and *F-measure* (i.e., the harmonic mean of precision and recall) [557]. We use the manually inferred tags as the ground truth. The results are tabulated in Table 6.1. We can achieve an F-measure of 64-72%. Next, we evaluate the usefulness of our

[21] www.softwaretipsandtricks.com

[22] forums.dzone.com

[23] forums.sun.com/index.jspa

[24] E.g., "How to read files in Java?", please refer to [343] for detail.

[25] www.lemurproject.org

semantic search engine that leverages inferred tags. We compare our approach with a conventional search engine in terms of *mean average precision (MAP)* over a set of 17 queries [557]. The mean average precision a the set of queries is the mean of the average precision per query; the average precision of a query is computed by averaging the precision at the top-k positions, for different values of k. With our semantic search engine we can accomplish an MAP score of 71%, while the conventional search engine can only achieve an MAP score of 18%.

Table 6.1: Precision, Recall, and F-measure Results of Our Proposed Tag Inference Engine

Dataset	Precision	Recall	F-measure
SoftwareTipsandTricks	73%	71%	72%
DZone	68%	61%	64%
Oracle	71%	67%	69%

6.4.2 Searching for Similar Applications in Software Forges

Motivation. Web search engines allow users to search for similar webpages (or documents in the Web). Similarly, developers might want to find similar applications (i.e., applications that serve similar purposes). Finding similar applications could help various software engineering tasks including rapid prototyping, program understanding, plagiarism identification, etc. There have been a number of approaches that could retrieve applications that are similar to a target application [455, 581]. McMillan et al. proposed JavaClan [581] which has been shown to outperform MUDABlue [455]. JavaClan leverages similarities of API calls to identify similar applications. API calls within applications are treated as *semantic anchors* which are used to identify similar applications to a target application. However, the accuracy of these approaches can still be improved. In their user study, JavaClan only achieved a mean confidence score of around 2.5 out of 4.

Recently, many developers tag various resources with labels. This phenomenon is referred to as *collaborative tagging*. Many software forges allow users to tag various applications based on their perceived functionalities. In this study, we leverage collaborative tagging to find similar applications. Our goal is to improve the accuracy of the state-of-the-art approach.

Approach. Our approach, shown in Figure 6.6, consists of several steps including: data gathering, importance weighting, and similar application retrieval. We describe these steps as follows:

1. Data Gathering. We download a large number of applications as the base corpus to detect similar applications. In this study we invoke the API that comes with

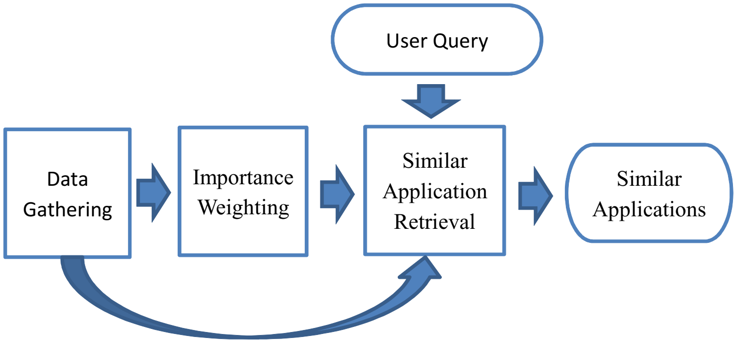
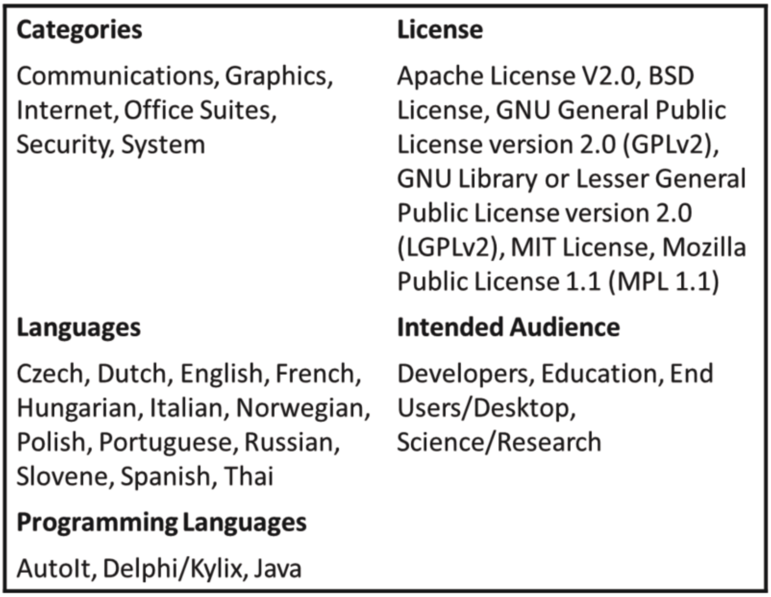


Fig. 6.6: Similar Application Retrieval: Block Diagram



Categories

Communications, Graphics, Internet, Office Suites, Security, System

License

Apache License V2.0, BSD License, GNU General Public License version 2.0 (GPLv2), GNU Library or Lesser General Public License version 2.0 (LGPLv2), MIT License, Mozilla Public License 1.1 (MPL 1.1)

Languages

Czech, Dutch, English, French, Hungarian, Italian, Norwegian, Polish, Portuguese, Russian, Slovene, Spanish, Thai

Intended Audience

Developers, Education, End Users/Desktop, Science/Research

Programming Languages

AutoIt, Delphi/Kylix, Java

Fig. 6.7: Example Tags from SourceForge

SourceForge[26] to collect tags from a large number of applications hosted there. An example of tags given to an application in SourceForge is shown in Figure 6.7. In this study, we treat each tag as a distinct entity and we ignore the semantic relationships between tags.

2. Importance Weighting. Not all tags are equally important. Some tags are very general and are used to label a large number of applications. These tags are not very useful for the retrieval of similar applications as otherwise all applications

[26] sourceforge.net/apps/trac/sourceforge/wiki/API

would be considered similar. On the other hand, tags that are only used by a few applications are more important for retrieval as they can help to differentiate one application from the others.

Based on the above rationale, we assign importance weights to tags based on applications tagged by them. If a tag is used by many different applications, it is assigned a low weight. On the other hand, if a tag is used by only a few applications, it is assigned a high weight. We use the concept of *inverse document frequency* first proposed in the information retrieval community [557] to assign weights to tags. Formally, the weight of a tag T is given by Equation 6.1 where $Applications(T)$ refers to the size of the application set tagged by T.

$$weight(T) = \frac{1}{Applications(T)} \tag{6.1}$$

3. Similar Application Retrieval. Each application is represented as a vector of its tags' weights. The similarity between two applications can then be measured based on the similarities of their representative vectors. Various *similarity measures* can be used. We use *cosine similarity* which is a standard similarity metrics in information retrieval [557]. The cosine similarity of two applications A and B is given by Equation 6.2 where $A.Tags$ and $B.Tags$ refer to the tags for application A and B respectively. From the numerator of the above formula, the cosine similarity of A and B is higher if they share many common tags that have high importance weights. The denominator of the formula normalizes cosine similarity to the range of zero to one. If an application is tagged with many tags, the chance for it to coincidentally share tags with other applications is higher. To address this, the denominator considers the number and weights of the tags that are given to each application.

$$CosSim(A,B) = \frac{\Sigma_{T\in(A.Tags\bigcap B.Tags)}.weight(T)^2}{\sqrt{\Sigma_{T\in A.Tags}.weight(T)^2} \times \sqrt{\Sigma_{T\in B.Tags}.weight(T)^2}} \tag{6.2}$$

Given a target application A, our system returns the top-n applications in our corpus that are most similar to A based on their cosine similarities.

Experiments. To investigate the effectiveness of our approach, in our data gathering step we collect 164,535 applications (i.e., projects) from SourceForge. These applications form our corpus to recommend similar applications. We use the following 20 queries: bcel, bigzip, certforge, chiselgroup, classgen, color-studio, confab, drawswf, genesys-mw, javum, jazilla, jsresources, opensymphony, psychopath, qform, redpos, sqlshell, tyrex, xflows, and yapoolman. Each of the above queries is an application. The 20 queries were also used in evaluating JavaClan which is the state-of-the-art approach to recommend similar applications [581].

We compare our approach with JavaClan. We use our approach and JavaClan to recommend 10 applications. We then perform a user study to evaluate the quality of the recommendations. We ask users to rate each recommendation using a 5-point

Likert scale [19]: 1. strongly disagree (i.e., the query and recommended applications are very dissimilar), 2. disagree, 3. neither agree or disagree, 4. agree, and 5. strongly agree (i.e., the query and recommended applications are very similar). Based on user ratings, we use the following three metrics to measure the effectiveness of our approach and JavaClan (the last two metrics have been used to evaluate JavaClan):

1. Success Rate. We deem a top-10 recommendation to be successful if at least one of the recommendations is given a rating 3 or above. The success rate is given by the proportion of top-10 recommendations that are successful for the queries.
2. Confidence. The confidence of a participant to a recommendation is reflected by his/her rating. We measure the *average confidence* which is the average of the ratings given by the participants for the top-10 recommendations.
3. Precision. The precision of a top-10 recommendation is the proportion of recommendations in the top-10 recommendation that are given ratings 4 or 5. We measure the average precision across the top-10 recommendations for the queries.

Table 6.2 shows the success rate, average confidence, and average precision of our proposed approach and JavaClan. In terms of success rate, our approach outperforms JavaClan: the success rate is increased by 23.08%. Our approach also achieves a higher average confidence than JavaClan. A *Mann-Whitney U test*, which is a non-parametric test to check the significance of a difference in means, shows that the difference in average confidence is significant (with a p-value of 0.001). Furthermore, out of the 20 queries, in terms of average confidence per query, our approach outperforms JavaClan in 13 queries and is equally as effective as JavaClan in 5 queries. Furthermore, our approach achieves a higher average precision score than JavaClan. We have also performed a A Mann-Whitney U test. The result shows that the difference in mean is not significant (with a p-value of 0.488). Furthermore, out of the 20 queries, in terms of precision per query, our approach outperforms JavaClan in 7 queries and is equally effective as JavaClan in 8 queries.

Table 6.2: Effectiveness of Our Proposed Approach and JavaClan: Success Rate, Confidence, and Precision

Approach	**Success Rate**	**Avg. Confidence**	**Avg. Precision**
Proposed Approach	80%	2.02	0.115
JavaClan	65%	1.715	0.095

6.4.3 Other studies

Aside from our studies, there are a number of other studies that also leverage Web 2.0 resources to help various software evolution activities. We highlight some of these studies in brief in the following paragraphs.

Thummalapenta and Xie proposed Parseweb which helps developers to reuse open source code [839]. Parseweb accepts as input a source object type and a destination object type. It then generates a sequence of method invocations that can convert the source object type to the destination object type. To realize its function, Parseweb interacts with a software forge namely Google Code and leverages the search utility available in it.

Thummalapenta and Xie proposed a technique named SpotWeb that detects hotspots and coldspots in a framework or *API* [838]. Hotspots refer to parts of the framework or API that are frequently used. Coldspots refer to parts of the framework or API that are rarely used. Their proposed technique works on top of Google Code search. It works by analyzing framework code and the framework usages among the projects in Google Code software forge. Experiments were conducted on a number of frameworks with promising results.

McMillan et al. proposed Portfolio which is a *search engine* that finds functions in a large code base containing numerous projects [582]. Their proposed search engine takes in user inputs in the form of free form natural language descriptions and returns relevant functions in the code base. Two information retrieval solutions are used to realize the proposed approach namely page rank and spreading activation network. Their search engine analyzes a software forge containing hundreds of projects from FreeBSD.

McMillan et al. proposed a tool named Exemplar (EXEcutable exaMPLes ARchive) which takes high-level concepts or descriptions and returns applications that realize these concepts [579]. The proposed approach ranked applications in a large application pool by considering several sources of information. These include textual description of the application, list of *API methods* that the application calls, and dataflow relations among the API method calls. Examplar had been evaluated on a set of 8,000 projects containing more than 400,000 files that are hosted in SourceForge with promising results.

Ponzanelli et al. proposed a technique that leverages crowd knowledge to recommend code snippets to developers [690]. Their tool named Seahawk is integrated to the *Eclipse IDE* and recommends code snippets based on the context that a developer is working on. To achieve this, Seahawk generates a query from the current context in the IDE, mines information from *StackOverflow* question-and-answer site, and recommends a list of code snippets to developers.

6.5 Supporting Information Discovery

In this section, we describe studies that develop tools that facilitate developers in discovering new information from Web 2.0 resources. We first highlight our visual analytics tool that supports developers in navigating through the mass of software-related microblogs in Twitter [3]. We also present our analytics tool that can automatically categorize microblogs to support information discovery [703]. We then highlight other studies.

6.5.1 Visual Analytics Tool for Software Microblogs

Motivation. Although microblogging is becoming a popular means to disseminate information, it is challenging to manually discover interesting software related information from microblogs. The first challenge comes from the sheer size of microblogs that are produced daily. Storing *all* microblogs is not an option. Second, many microbloggers do not microblog about software related topics. Indeed only a minority of microbloggers are software developers. Thus there is a need to filter many unrelated microblogs to recover those that are relevant to software development. Third, the large number of microblogs might make it hard for developers to "see" trends in the data. Motivated by these challenges, there is a need to develop an approach that can harvest and aggregate thousands or even millions of microblogs. It should also allow developers to perform *visual analytics* such that various kinds of trends and nuggets of knowledge can be *discovered* from the mass of microblogs. In this study, we propose such an approach.

Approach. We propose a visual analytics platform that filters and aggregates software related microblogs from Twitter. Our proposed platform identifies *topical* and *longitudinal* trends. Topical trends capture relative popularity of similar topics, e.g., relative popularity of various libraries. Longitudinal trends capture the popularity of a topic at various time points, e.g., the number of times people microblog about PHP at various time points. These trends can provide insight to developers, e.g., developers can discover popular programming languages to learn, or discover interesting events (e.g., notification of important security holes, etc.) in the past 24 hours.

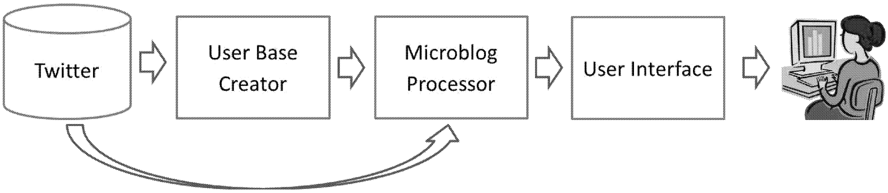


Fig. 6.8: **Proposed Approach: Visual Analytics Platform**

Our platform, illustrated in Figure 6.8, has 3 blocks: *User Base Creator*, *Microblog Processor*, and *User Interface*. *User Base Creator* recovers microbloggers that are likely to microblog about software related topics. *Microblog Processor* downloads and pre-processes microblogs from Twitter. It also identifies topical and longitudinal trends from the microblogs. *User Interface* presents the trends to end users as a web interface which allows users to analyze the trends and the underlying microblogs.

1. *User Base Creator* first processes a set of seed users which are well-known microbloggers that often microblog about software topics. We take the list of seed users available in [43]. In Twitter, a user can *follow* other users and receive up-

dates on microblogs made by the other users. Using these follow links, we expand the seed users to include microbloggers that follow at least *n* seed users (by default we set the value *n* to 5). We consider this user base as those that are likely to microblog about software related topics.

2. *Microblog Processor* uses Twitter REST API to continually download microblogs. We then perform standard text pre-processing including tokenization, stopword removal, and stemming. Some technical jargons, e.g., C#, C++, etc. are manually identified. These jargons are not stemmed. Since the identification of jargons is done manually, we focus on jargons corresponding to topics whose trends we would like to visualize. There are jargons that we do not identify and they are treated as regular words and are stemmed. We then index the resultant set of microblogs using *Apache Solr*.[27]
   Next, we perform trend analysis and compute both topical and longitudinal trends. To compute topical trend, we manually select a set of 100 software-related topics, e.g., JavaScript, Scrum, etc., from relevant Wikipedia pages and popular StackOverflow's tags. We then compute for each topic the number of microblogs mentioning the topic at a specific time period. Topics that are more frequently mentioned are more popular than others. To compute the longitudinal trend of a particular topic or keyword, we compute the number of tweets containing it at various points in time. We thus could compute the popularity of various topics and the popularity of a topic at various time points.
3. *User Interface* presents the resultant topical and longitudinal trends. To present topical trends, we display various topics using fonts of various sizes. The size of the font depends on the popularity (i.e., frequency) of the corresponding topic in the microblogs. To present longitudinal trends, for each topic, we plot a line graph that shows the popularity of the topic at various time points.

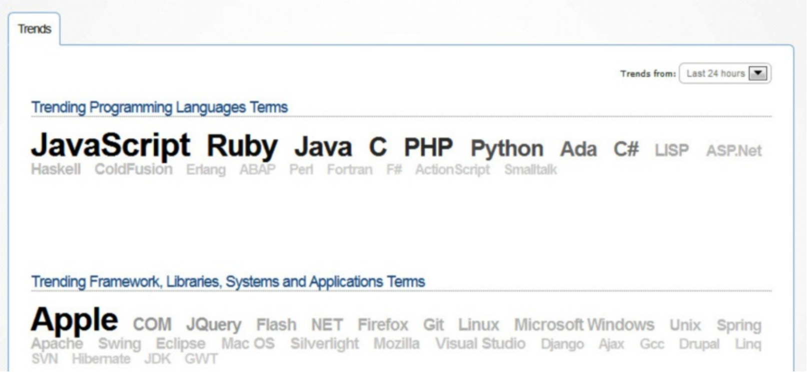
Experiments. Our dataset consists of approximately 58,000 microbloggers, 76 million microblogs, and 18 million follow links.

With topical trend analysis, popular topics of interest can be highlighted to users. Figure 6.9 shows our topical trend user interface. It shows the relative popularity of various topics. From the interface, users can find out that *JavaScript*, *Ruby*, and *Java* are the most popular programming language topics in the microblogs that we collected in a 24-hour period ending on the 25th of November 2011. For framework, libraries, and systems, *Apple*, *COM*, and *JQuery* are the most popular topics.

With longitudinal trend analysis, the popularity of a topic across different time points can be captured and shown to users. Figure 6.10 shows our longitudinal trend user interface for "JavaScript". We can notice that the number of microblogs related to JavaScript varies over time. We also notice a number of peaks. The highest peak is for the 10th of October 2011. At this date, Google released a new programming language called *Dart* [829]. Programs written in *Dart* can be compiled into JavaScript. We also notice that the number of microblogs related to JavaScript changes periodically - people tend to microblog more about JavaScript on some days than other

[27] `lucene.apache.org/solr`

days. Figure 6.11 shows another longitudinal trend for “Scrum”. We note that, similar to the popularity of JavaScript, the popularity of Scrum is also periodic. We do not notice much anomaly in the Scrum longitudinal trend though. In the future, it is interesting to develop approaches that can automatically highlight anomalies and recover important events.



Trends

Trends from: Last 24 hours

Trending Programming Languages Terms

JavaScript Ruby Java C PHP Python Ada C# LISP ASP.Net Haskell ColdFusion Erlang ABAP Perl Fortran F# ActionScript Smalltalk

Trending Framework, Libraries, Systems and Applications Terms

Apple COM JQuery Flash NET Firefox Git Linux Microsoft Windows Unix Spring Apache Swing Eclipse Mac OS Silverlight Mozilla Visual Studio Django Ajax Gcc Drupal Linq SVN Hibernate JDK GWT

Fig. 6.9: **Topical Trend User Interface**

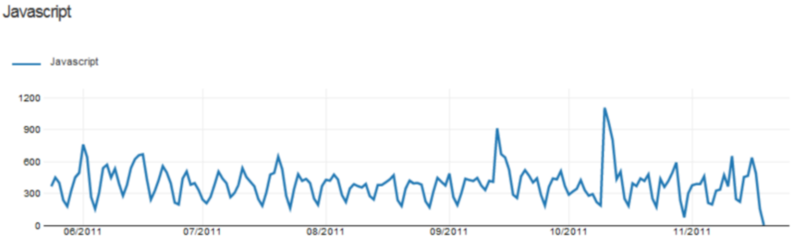


Fig. 6.10: **Longitudinal Trend User Interface for “JavaScript”**

6.5.2 *Categorizing Software Microblogs*

To leverage microblogging in software evolution tasks, we need to first understand how microblogging is currently used in software related contexts. One way to do this is to categorize software related microblogs. We present our *machine learning approach* that automatically assigns category labels to microblogs.

Motivation. By subscribing to and reading microblogs written by other developers, a developer can discover much information, e.g., a new programming trick, a new

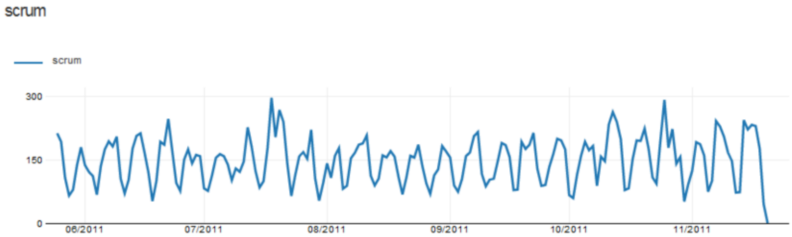


Fig. 6.11: **Longitudinal Trend User Interface for "Scrum"**

API, etc. Unfortunately, most of the microblogs are not informative [629]. Even if they are informative, they might not be relevant to *engineering* software systems. Our manual investigation on a few hundreds microblogs tagged with software related hashtags (see Section 6.3.2) shows that most of the microblogs belong to the category: jobs. They are job advertisements and are not relevant to engineering software systems. Thus, many interesting microblogs *relevant* to engineering software systems are buried in the mass of other *irrelevant* microblogs. In this work, we build a machine learning solution that can automatically differentiate *relevant* and *irrelevant* microblogs.

Approach. The framework of our proposed approach is shown in Figure 6.12. It works in two phases: training and deployment. In the training phase, the goal is to build a *machine learning model* (i.e., a discriminative model) that can discriminate relevant and irrelevant microblogs. In the deployment phase, this model is used to classify an unknown microblog as relevant or irrelevant. Our framework consists of 3 main blocks: webpage crawler, text processor, and classifier. A microblog can contain a URL; for such microblogs the webpage crawler block downloads the content of the webpage pointed by the URL. Our text processor block converts textual content in the microblogs and downloaded webpages' titles into word tokens after standard information retrieval pre-processing steps. These word tokens become features for our classifier which constructs a discriminative model. The model is then used to predict if a microblog is relevant or not.

We elaborate the webpage crawler, text processor, and classifier blocks as follows:

1. Webpage Crawler. A microblog in Twitter contains a maximum of 140 characters. To express longer contents, microbloggers often include a URL to a webpage, containing expanded content, in the microblog. Services like *bit.ly* are often used to shorten the URL. Information contained in the webpages pointed by these URLs can help to classify the relevance of the microblog. Thus, we want to download these external webpages. Our webpage crawler block performs this step by first checking if a microblog contains a URL. It uses a regular expression to detect this. It then expands any shortened URL into the original URL by checking the HTTP header. Finally, it downloads the webpages. It then extracts the titles of these webpages as they provide succinct yet very informative con-

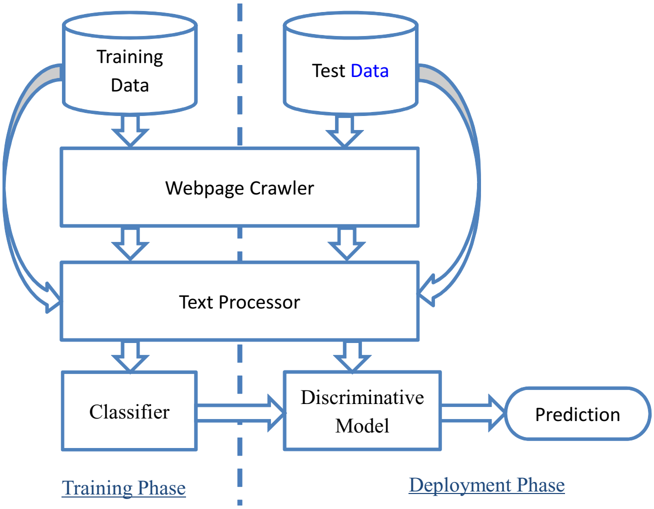


Fig. 6.12: Proposed Approach: Microblog Relevancy Categorization

tents. The body of a webpage is often long and contain extraneous information (e.g., advertisements, navigation links, etc.).

2. Text Processor. This block processes the text contents of the microblogs and webpage titles. It first removes stop words based on *Natural Language Toolkit (NLTK)'s* stopword list.[28] Next, it reduces each word to its root form (i.e., stemming) by using Porter stemmer [692]. Finally, each pre-processed word is treated as a feature and we combine these words to form a feature set that characterize a given microblog.
3. Classifier. This block takes in the feature sets, produced by the text processor block, of a set of microblogs whose relevancy label is known (i.e., relevant or irrelevant). It then constructs a discriminative model that differentiates relevant from irrelevant microblogs. We make use of *support vector machine (SVM)* [612] to construct the discriminative model. SVM has been widely used in past studies on software mining [489, 802]. SVM views a microblog as a point in a multi-dimensional space where each feature is a dimension. It then creates a hyperplane that best separates feature sets of the relevant microblogs with those of the irrelevant microblogs. This hyperplane is the discriminative model which is used in the deployment phase to assign relevancy labels to other microblogs.

Experiments. We use a dataset consisting of 300 microblogs which are tagged with either one of the following 9 *hashtags*: #csharp, #java, #javascript, #.net, #jquery, #azure, #scrum, #testing, and #opensource. Although the dataset does not cover all kinds of microblogs and hashtags, it is a good starting point to test the effectiveness of our proposed approach. These microblogs have been grouped into 10 categories

[28] `nltk.org`

listed in Table 6.3 (see [845]). Here, to create the ground truth data to evaluate the effectiveness of our approach, we manually re-categorize these 300 microblogs into 2 classes: relevant and irrelevant. The distribution of relevant and irrelevant microblogs across the 10 categories is shown in Table 6.4.

Table 6.3: Microblog Categories

	Category	**Details**
1.	Commercials	Advertisements about a commercial product or a company
2.	News	Objective reports
3.	Tools & Code	Sharing of code and/or links to open source tools
4.	Q&A	Questions or links to questions in Q&A sites
5.	Events	Notification of particular events or gatherings
6.	Personal	Personal messages, e.g., ramblings about programming, etc.
7.	Opinions	Subjective expressions of likes or dislikes
8.	Tips	Advice about a particular problem, e.g., how to do a particular programming task, etc.
9.	Jobs	Job advertisements
10.	Misc.	Other kinds of microblogs. This includes microblogs whose contents are unclear.

Table 6.4: Relevance Per Microblog Category

Category	**Proportion of Relevant Microblogs**
Tools & Code	100%
Tips	100%
Q&A	86.4%
Events	45.5%
Opinions	42.9%
Commercials	40%
News	29.5%
Personal	0%
Jobs	0%
Misc.	0%

Using the above data, we perform a 10-fold cross validation, and measure the precision, recall and F-measure of our proposed approach. In 10-fold cross validation, 90% of the data is used for training and only 10% is used for testing. We would like to investigate the sensitivity of our approach on the amount of training data.The experiment shows that we can predict the relevancy of a microblog with 74.67% accuracy, 76% precision, 67.38% recall, and 71.43% F-Measure.

Next, we investigate the effectiveness of our approach for each of the ten microblog categories. The result is shown in Table 6.5. It shows that we can more accurately predict relevancy labels of jobs, personal, Q & A, tools & code, opinions, and misc categories. Our approach needs to be further improved for tips category (low precision), and events category (low precision and recall). For the events category, the microblogs are more ambiguous and it is harder to predict if a microblog is relevant or not. In the future, we plan to use other approaches including sentiment analysis [677] to improve the accuracy of our proposed approach.

Table 6.5: Effectiveness Per Microblog Category

Category	Accuracy	Precision	Recall	F-Measure
Jobs	100%	0%	0%	0%
Personal	93.8%	0%	0%	0%
Q&A	79.6%	84.2%	91.4%	87.7%
Tools & Code	79.5%	79.5%	100%	88.6%
Opinions	76.2%	55.6%	83.3%	66.7%
Misc.	72%	0%	0%	0%
Tips	48.5%	48.5%	100%	65.3%
Commercials	60%	50%	50%	50%
News	54.5%	61.5%	34.8%	44.4%
Events	45.5%	20%	33.3%	25%

For the above results, we make use of 10-fold cross validation. Then we would like to investigate the sensitivity of our approach on the amount of training data. For this, we perform *k-fold cross validation*, where k is less than 10. We vary k from 2 to 9 and show the resulting accuracy, precision, recall, and F-measure for these values of k in Table 6.6. We notice that the F-measure scores do not vary much, this shows that our framework is effective enough on different amount of training data.

Table 6.6: Results using Different Amount of Training Data

k	Accuracy	Precision	Recall	F-Measure
9	75.43%	75.19%	70.92%	72.99%
8	74.29%	74.62%	68.79%	71.59%
7	73.98%	74.05%	68.79%	71.32%
6	74.33%	73.88%	70.21%	72%
5	73.67%	74.22%	67.38%	70.63%
4	75%	75.78%	68.79%	72.12%
3	74.67%	74.44%	70.21%	72.26%
2	75%	75.78%	68.79%	72.11%

6.5.3 Other studies

There are a number of other studies that leverage Web 2.0 resources for information discovery. We highlight a few of them in brief in the following paragraphs.

Hens et al. extracted frequently asked questions (FAQs) from mailing lists and software forums [389]. They employed a text mining approach that utilizes text pre-processing techniques and *Latent Dirichlet Allocation (LDA)* which is a topic modeling technique. After a topic model was learned from the mailing lists and software forums, several processing phases were employed to identify question and answer pairs that are associated with a topic, discard topics that are unfocused, and process the remaining question and answer pairs to improve their readability. They had investigated their proposed approach on mailing lists of 50 popular projects listed in ohloh.net.

Lungu et al. proposed a visualization tool named Small Project Observatory that analyzes projects in a software forge [548]. With their visualization tool, developers can investigate the evolution of project size (in terms of the number of classes), the level of activity (in terms of the number of commits) occurring within a repository over time, the dependencies among projects, the collaborations among developers, and many more. They have deployed their visualization tool on a software forge owned by Soops b.v, which is a Dutch software company, with promising results.

Sarma et al. proposed Tesseract which is a visualization tool that enables one to explore socio-technical relationships in a software project [746]. Tesseract simultaneously shows various pieces of information to users including: developers, their communications, code, and bugs. Tesseract also supports interactive explorations - it allows users to change various settings, filter information, highlight information, and link information in various ways. Tesseract had been evaluated on the *GNOME project* via a user study and the result is promising.

6.6 Supporting Project Management

In this section, we highlight how Web 2.0 resources could be leveraged to aid project management activities. Project management activities (e.g., planning, organizing, and managing resources) need to be performed repeatedly as software evolves over time. We first highlight studies that leverage software forges for the recommendation of developers to a project [805] and prediction of project success [806]. We also describe other related studies.

6.6.1 Recommendation of Developers

Motivation. It is a challenge to find compatible developers as not everyone works equally well with everyone else. Often there are hundreds or even thousands of developers. It is hard for a manager to know everyone well enough to make good recommendations. Past studies only recommend developers from a single project to fix a particular bug report [819]. Thus there is a need for a tool that can help recommend developers based on their past socio-technical behaviors and skills. In this work we focus on recommending developers from a software forges (i.e., SourceForge).

Approach. Our approach consists of 2 main steps: *Developer-Project-Property (DPP) graph construction*, and *compatibility scores computation*. In the first step, we represent the past history of developer interactions as a special Developer-Project-Property (*DPP*) graph. In the second step, given a developer we compute compatibility scores of the developer with other developers in the *DPP* graph. We propose a new compatibility metric based on *random walk with restart (RWR)*. We elaborate the above two steps in the following paragraphs.

Given a set of developers, their past projects, and the project properties, we construct a *DPP* graph. There are three node types in a *DPP* graph: developers, projects, and project properties. We consider two project properties: project categories and project programming languages. There are two types of edges in a *DPP* graph: one type links developers and projects that the developers have participated in before, another links projects and their properties. A developer can work on multiple projects. A project can have multiple properties: it can be associated with multiple categories and/or multiple programming languages. For forges where only one programming language is supported, other properties aside from programming language can be considered, e.g., tags [855], libraries used, etc.. Figure 6.13 gives an example of a *DPP* graph which is a tripartite graph.

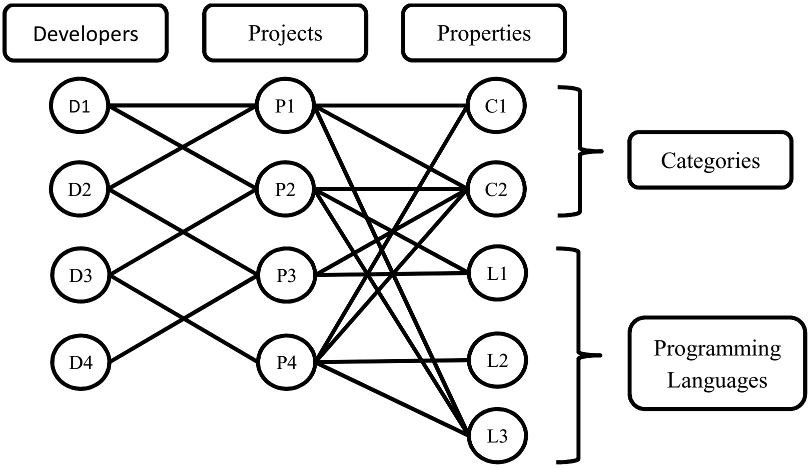


Fig. 6.13: **Example Developer-Project-Property (*DPP*) Graph**

After a *DPP* graph is constructed, we can compute compatibility scores between each pair of developers. A more compatible pair of developers should be assigned a higher score than a less compatible pair. Intuitively, a good compatibility metric should satisfy the following:

1. A pair of developers that have worked together in many joint projects are more likely to be compatible than another pair that have not worked together before.
2. A project is characterized by its properties: categories and programming languages. Intuitively, developers that have worked on similar projects (i.e., different projects of the same/similar properties) are more likely to be more compatible than those that have worked on completely unrelated projects.
3. Developers might not have worked together before. However, they might have a common collaborator. Developers with many common collaborators developing similar projects are more likely to be more compatible than "complete strangers". The same is true for collaborators of collaborators, albeit with lower impact on compatibility.

The above describes three qualitative criteria for a good compatibility metric. We find that computing node similarity using *random walk with restart (RWR)*, which was first proposed for web search engines in 1998 [676], fits the three criteria. Given a developer node d in the *DPP*, by performing many random walks with restart starting from developer node d, many nodes are visited. Some nodes are visited more often than other nodes. RWR assigns scores to these other nodes based on the probability that these nodes are visited during RWR starting from node d. After RWR, developers with higher scores are more likely to have worked with developer d on many common projects, or they have worked on projects with similar properties, or they share many common collaborators or collaborators of collaborators. Given the target developer d, we sort the other developers based on their RWR scores, and return the top-k most compatible developers.

Experiments. To evaluate the effectiveness of our proposed developer recommendation approach, we analyze projects in SourceForge. We make use of the curated data collected by Van Antwerp et al. [40].[29] We analyze the curated data collected from May 2008 until May 2010. Each month, Antwerp et al. release a snapshot of the curated data in the form of SQL tables. From these snapshots, we extract information about developers, projects that these developers work on, and project categories as well as programming languages. To recommend developers, we need sufficient information of developers' past activities. Thus, we only include developers that have worked on at least p projects. SourceForge contains many trivial projects; to filter these projects, we only include projects that have at least n developers. In this study, we set the value of p and n to be 7 and 3 respectively.

A good recommendation eventually leads to a collaboration. To evaluate our approach, we take multiple consecutive monthly snapshots of SourceForge. We consider new collaborations created between these consecutive snapshots. We then apply our approach and investigate if we can accurately predict these new collaborations. We consider a recommendation is successful if at least one of the recommended developer collaborates in the next snapshot. The accuracy of our approach is defined as the proportion of recommendations that are successful. If many new collaborations do not follow our recommendations then the accuracy would be low. This measure is also often referred to as recall-rate@k and has been used in many

[29] www3.nd.edu/~oss/Data/data.html

past studies [645, 737, 800]. This is a lower bound of the accuracy of our proposed approach. In practice, our approach would *actively* recommend developers and more collaborations could have been created.

Given a target developer d, our approach would recommend k developers with the highest RWR scores. Using k equals to 20, for the new collaborations created from May 2008 to May 2010, we find that our recommendation success rate is 83.33%. We also vary the value k and investigate the behavior of our approach. We find that the success rate (or accuracy) varies from 78.79% to 83.33% when k is varied from 5 to 20. Thus there is only a minimal change in accuracy (i.e., 4.54% reduction) when we drop k from 20 to 5. This shows that our top few recommendations are accurate. The runtime of our approach is 0.03 seconds for training (i.e., creating *DPP* and pre-computing internal data structures) and less than a second for query (i.e., running RWR with the pre-computed internal data structures). This shows that our approach is efficient and could be used to support interactive query.

6.6.2 Prediction of Project Success

Motivation. Project success is the eventual goal of software development efforts, be it open source or industrial. There are many projects that are successful - they get released, distributed, and used by many users. These projects bring much benefit in one form or another to the developers. However, many other projects are unsuccessful, they do not get completed, not used by many (if any at all), and bring little benefit to the developers despite their hard work. Investigating failed and successful projects could shed light on factors that affect project outcome. These factors can in turn be used to build an automated machine learning solution to predict the likelihood of a project to fail or be successful. Predicting project outcome is important for various reasons including planning, mitigation of risks, and management of resources.

With the adoption of Web 2.0, much data is available to be analyzed. We can collect information on various successful and failed projects. We can trace various projects that developers have worked on before. In this work, we leverage socio-technical information to differentiate successful and failed projects. Our goal is to find relevant socio-technical patterns and use them to predict project outcome.

Approach. Figure 6.14 illustrates the framework of our proposed approach. It has two phases: training and deployment. In the training phase, our framework processes a set of projects along with developers that work in them. The goal of the training phase is to build a discriminative model that can differentiate successful and failed projects. This model is then passed to the deployment phase to predict the outcomes of other projects. It has several processing blocks: socio-technical information extraction, discriminative graph mining, discriminative model construction, and outcome prediction. The following elaborates each of these processing blocks:

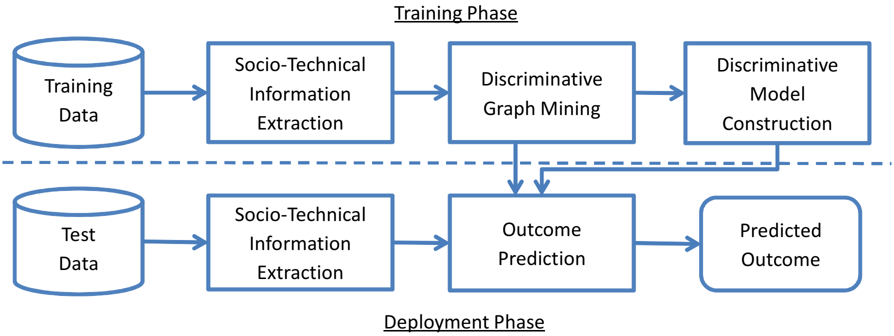

Training Phase
Training Data
Socio-Technical Information Extraction
Discriminative Graph Mining
Discriminative Model Construction
Test Data
Socio-Technical Information Extraction
Outcome Prediction
Predicted Outcome
Deployment Phase

Fig. 6.14: Overall Framework

1. The socio-technical information extraction block processes each of the projects and for each of them extracts socio-technical information in the form of a rich (i.e., multi-labeled) graphs. The nodes in the graph are developers that work on the project. The edges in the graph correspond to the relationships among the various developers. Multiple labels are attached to the nodes and edges to capture the socio-technical information about the developers and their relationships. For each node, we attach the following pieces of information:

   a. Past Successful Projects. This is the number of successful projects that a developer has participated before he joins the current project.
   b. Past Failed Projects. This is the number of failed projects that a developer has participated before he joins the current project.
   c. Length of Membership. This is the period of time that has passed since a developer has joined the software forges before he joins the current project.

   For each edge that links two developers, we attach the following pieces of information:

   a. Past Successful Collaborations. This is the number of successful projects that the two developers have worked together before.
   b. Past Failed Collaborations. This is the number of failed projects that the two developers have worked together before.
   c. Length of Collaboration History. This is the period of time that has passed since the two developers collaborated for the first time to the time they collaborate in the current project.

2. The discriminative graph mining block takes as input the graphs capturing the socio-technical information of the successful and failed projects. The goal of the block is to extract subgraphs that appear often in the socio-technical graphs of the successful projects but rarely in the graphs of the failed projects (or vice versa). These discriminative graphs highlight peculiar socio-technical patterns

that differentiate successful from failed projects. We propose a new discriminative subgraph mining algorithm that analyzes rich graph where each node and edge have multiple labels. We can assign a score $S(g)$ to evaluate the discriminativeness of a subgraph g. The goal of a discriminative graph mining algorithm is then to return top-k subgraphs g that have the highest $S(g)$ scores. Variosus measures of discriminativeness have been proposed in the literature. In this work, we make use of *information gain* [612]. Our algorithm extends the work by Cheng et al. [179] that works on a set of simple graphs (i.e., graphs where nodes and edges have one label each) by a translation-and-reverse-translation based approach:

a. We translate the set of rich graphs to their equivalent simple graph representations.
b. We mine discriminative subgraphs from the simple graphs using the algorithm proposed by Cheng et al. [179].
c. We reverse translate the resultant discriminative simple subgraphs back to their corresponding rich graphs.

3. The discriminative model construction block takes as input the set of discriminative subgraphs. Each of the subgraphs form a binary feature. A socio-technical graph is then represented by a vector of binary features. Each binary feature is assigned a score of 1 if a discriminative subgraph appears in it. The score would be 0 otherwise. Using this representation the successful and failed projects become points in a multi-dimensional space. We use *support vector machine (SVM)* to create a discriminative model which is a hyperplane that best separates the two sets of points.
4. The outcome prediction block takes as input the discriminative model learned in the training phase and vector representations of projects whose outcomes are to be predicted. These vector representations are generated by first extracting socio-technical graphs. Each of these graphs are then compared with the discriminative subgraph patterns extracted during the training phase. Each of the patterns form a binary feature that collectively characterize each of the socio-technical graphs. These features are then input to the discriminative model and a prediction would be outputted.

Experiments. We analyze successful and failed projects in SourceForge. We use the monthly database dumps created by Antwerp et al. [40] from February 2005 to May 2010. Projects that have been downloaded more than 100,000 times are deemed to be successful. On the other hand, those that have been downloaded less than 100 times are considered to have failed. Other definitions of success and failure can also be considered; we only investigate one definition in this work. We extract 224 successful projects and 3,826 failed projects. Using this dataset, we want to investigate if socio-technical information of participating developers (which could be gathered even when a project is at its inception) could be used to predict project success using our proposed approach. In the experiments, we first analyze the efficiency, followed by the effectiveness of our approach.

We find that our translation and reverse translation processes can complete in a short amount of time. The translation process only takes less than 15 seconds to translate the successful and failed projects. After translation the sizes of the graphs grow, however their growth is linear to the number of node labels and edge labels. The average sizes of the translated graphs are 31.54 nodes and 287.25 edges (for successful graphs) and 23.93 nodes and 204.68 edges (for failed graphs). The most expensive operation in our framework is to run the algorithm of Cheng et al. [179] which completes within 4 hours. We mine the top-20 most discriminative rich subgraph patterns.

To measure the effectiveness of our approach we use two measures: accuracy and *area under the ROC curve (AUC)* [362]. The *ROC curve (Receiver Operating Characteristic)* plots the false positive rate (x-axis) against the true positive rate (y-axis) at various settings. AUC is more suitable to be used than accuracy for skewed datasets. For our problem, we have a skewed dataset as there are more failed projects than successful projects. The maximum AUC score is 1.0. Using ten-fold cross validation, our proposed approach can achieve an accuracy of 94.99% and an AUC of 0.86.

6.6.3 Other studies

There are a number of other studies that leverage Web 2.0 resources for software project management to reduce the amount of wasted effort, to better manage resources (i.e., developer time and effort), and to coordinate activities. We highlight a few of them in brief in the following paragraphs.

Guzzi et al. combined microblogging with *IDE interactions* to support developers in their activities [356]. Guzzi et al. noted that developers often need to go through program understanding phase many times. This is a time consuming activity. To address this problem Guzzi et al. proposed a tool named James that integrates microblogging with interaction information that is automatically collected from an IDE. Developers can then share their program understanding experience to their colleagues using James. Thus with James, wasted effort can be reduced and developer resources can be better spent on more useful activities.

Ibrahim et al. investigated factors that encourage developers to contribute to a mailing list discussion [418]. There are numerous threads in a mailing list and developers can easily miss relevant threads to which he/she can contribute ideas and expertise. To address this problem, Ibrahim et al. proposed a personalized tool that recommends threads that a developer is likely to contribute to based on the developer past behaviors. The proposed tool combines two *machine learning algorithms* namely: Naive Bayes and Decision Tree. The proposed tool has been evaluated on mailing lists of three open source projects, Apache, PostgreSQL and Python, with promising results.

Carter and Dewan proposed a tool that is integrated with Google Talk [167]. This tool could highlight remote team members in a distributed development team who are having difficulty in their tasks, and thus foster more collaborations among developers. Expert developers or project managers could be aware of team members that require help. The proposed tool logs developer interactions with a development environment. A classification algorithm was then employed to infer a developer status based on his/her interaction log. A user study was conducted to evaluate the effectiveness of the proposed approach with promising results.

6.7 Open Problems and Future Work

In the previous sections, we have highlighted a number of studies that analyze how developers use Web 2.0 resources and how automated tools can leverage these resources for information search, information discovery and project management. Albeit the many existing work in this area, we believe much more work can be done to better leverage Web 2.0 resources for software evolution. We highlight some of the open problems and potential future work in this section.

There are many Web 2.0 resources that have not been tapped to improve software evolution. In Section 6.2 we highlighted resources such as directories of developers in LinkedIn, public profiles of developers in Facebook, definitions of software engineering terms in Wikipedia and geolocation coordinates of developers in Foursquare. To the best of our knowledge, there have not been any study that utilize these resources to help software evolution. Furthermore, various web systems evolve (see Chapter 7); thus, many additional functionalities and services are introduced to existing Web 2.0 resources regularly. Many innovative applications can potentially be built leveraging these resources and additional functionalities. For example, one can imagine a tool that enables one to search for a potential employee by leveraging information in LinkedIn and Facebook and correlating the information with the kinds of software evolution tasks that the future employee is supposed to perform. One could also better enhance many information retrieval based tools, e.g., [800, 846, 906, 949], by leveraging domain specific knowledge stored in Wikipedia. One can also imagine an application that tries to recommend more interactions among developers that live in a nearby area by leveraging geolocation coordinates in Foursquare. Thus there is plenty of room for future work.

Combining *many* different sources of information and leveraging them to improve software evolution activities is another interesting direction for future work. Most studies so far only focus on one or two Web 2.0 resources. Each Web 2.0 resources provides an incomplete picture of an entity (e.g., a developer). By combining these Web 2.0 resources, one can get a bigger picture of an entity and use this bigger picture to support various software evolution activities, e.g., recommend a fix/a developer to a bug in a corrective software evolution activities by leveraging information in multiple software forums, question-and-answer sites, software forges, etc.

Another interesting avenue for future work is to improve the effectiveness and efficiency of machine learning solutions that analyze and leverage Web 2.0 resources.

As highlighted in previous sections, the accuracy of existing techniques is not perfect yet. Often the accuracy (measured either in terms of precision, recall, accuracy, or ROC) is lower than 80%. Thus there is much room for improvement. Many new advances in machine learning research can be leveraged to improve the accuracy of these existing techniques. One can also design a new machine learning solution that is aware of the domain specific constraints and characteristics of software engineering data and thus could perform better than off-the-shelf or standard solutions. It is also interesting to investigate how search-based algorithms (described in Chapter 4) and information retrieval techniques (mentioned in Chapter 5) can be utilized to improve the accuracy of existing techniques that leverage Web 2.0 resources.

6.8 Conclusions

Web 2.0 provides rich sources of information that can be leveraged to improve software evolution activities. There are many Web 2.0 resources including software forums, mailing lists, question-and-answer sites, blogs, microblogs, and software forges. A number of empirical studies have investigated how developers contribute information to and use these resources. Automated tools can also be built to leverage these resources for information search, information discovery, and project management which are crucial activities during software evolution. For information search, we have highlighted some examples how Web 2.0 resources can be leveraged: tags in software forums can be used to build a semantic search engine, tags can also be used to recover similar applications, code fragments of interest can be extracted from Web 2.0 sites, etc.. For information discovery, we have also highlighted some examples how Web 2.0 resources can be leveraged: users can find interesting events by navigating through the mass of software microblogs using a visual analytics solution, users can be notified of relevant microblogs using a classification-based solution, frequently asked questions can be extracted from Web 2.0 sites, etc.. For supporting project management activities, Web 2.0 resources can also be leveraged in several ways: appropriate developers can be recommended to a project based on their socio-technical information stored in software forges, potentially unsuccessful projects can be highlighted early using developer socio-technical information stored in software forges, better collaboration among developers can be achieved by integrating microblogging with IDEs, etc.. Much more future work can be done to better leverage Web 2.0 and even Web 3.0 resources in various ways to improve many software evolution activities.

Part III
Evolution of specific types of software systems

Chapter 7
Evolution of Web Systems

Holger M. Kienle and Damiano Distante

Summary. The World Wide Web has led to a new kind of software, web systems, which are based on web technologies. Just like software in other domains, web systems have evolution challenges. This chapter discusses evolution of web systems on three dimensions: architecture, (conceptual) design, and technology. For each of these dimensions we introduce the state-of-the-art in the techniques and tools that are currently available. In order to place current evolution techniques into context, we also provide a survey of the different kinds of web systems as they have emerged, tracing the most important achievements of web systems evolution research from static web sites over dynamic web applications and web services to Ajax-based Rich Internet Applications.

Parts of this Chapter have been taken and adapted from other publications of the first author [469] [460] [467] and the second author [98] [97] [315] [316] [96].

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_7, © Springer-Verlag Berlin Heidelberg 2014

7.1 Introduction

The emergence of the World Wide Web (WWW), or the *web* for short, has led to a new kind of software that is based on web technologies: web sites, web applications, web services, and possibly others. In the following, if we do not make a distinction among these, we speak of them as *web systems*. We have chosen this term to convey that software that is based on the web platform can be of significant complexity, employing a wide range of technologies—starting from complex client-side code based on HTML5, Flash and JavaScript, to server-side code involving high-performance, cloud-based web servers and database back-ends. Web systems also have to deal with demanding non-functional requirements, which are cross-cutting the client and server side, such as scalability and security.

Research on the evolution of web systems is a relatively young research branch within the research on the evolution of software systems. Early research into web systems evolution started towards the end of the last millennium. The first International Workshop on Web Site Evolution (WSE), held in October 1999, evolved into the annual Web Systems Evolution workshop and symposia series sponsored by the IEEE. Since that time, this research branch has become more prominent, reflecting the web's increasing significance, and has broadened its scope, reflecting the web's increasing diversity.

Even though web systems have their own characteristics and idiosyncracies (cf. Section 7.1.2), almost all of the evolution techniques that have been proposed for software systems in general (e.g., refactoring [596] and clone detection [592, Chapter 2]) can be suitably adopted and applied to web systems (e.g., refactoring of PHP code [856] and detection of cloned web pages [243]). In addition, dedicated techniques have been developed to account for the specific characteristics of web systems (e.g., testing for browser-safeness, discussed in Section 7.4, and refactoring of web design models, discussed in Section 7.3.2). In this chapter we concentrate on techniques and tools that target web systems evolution that are meant to change one or more of the following aspects of a web system: its architecture, its (conceptual) design, and its technology.

7.1.1 Reengineering

One of the most involved methods to realize software evolution is *reengineering*, which conceptually can be defined as the composition of *reverse engineering* followed by *forward engineering*. The system to evolve is first reverse engineered to obtain higher-levels of meaning/abstractions from it, such as a model of its current design. Based on the information obtained in the reverse engineering step and the evolution objectives, a forward engineering step produces a new version of the system with a new architecture, design, and/or technology. The approaches discussed in Section 7.3 are mostly reengineering techniques.

Reengineering activities are performed with certain goals in mind, of which prominent ones are: (1) to "come to a new software system that is more evolvable [. . .] than the original one" [592, Chapter 1], or (2) to "improve the quality of the software" [781]. Regarding the first goal, making a system more evolvable often means adapting it so that it remains usable in reaction to changes in the real world (i.e., changing requirements or assumptions, cf. Chapter 1) in which the software operates (e.g., a new execution context, or interfacing it to another system). Regarding the second goal, quality improvements of a system can refer to internal quality, external quality, or quality in use (cf. ISO/IEC 9126 [12]).

7.1.2 Evolution Challenges and Drivers

Just like software in other domains, web systems have evolution challenges—perhaps even more so compared to other domains because standards, technologies, and platforms in the web domain are changing rapidly.[1] Web systems development is often equated with rapid development cycles coupled with ad-hoc development techniques, potentially resulting in systems of lower quality and reliability. An empirical study that tracked six web applications over five years for anomalies and failures found that "only 40% of the randomly selected web applications exhibit no anomalies/failures" and comes to the conclusion that "the common idea of researchers and practitioners is correct, i.e., process, tools and in general methodologies for web development and testing should be improved" [719].

Web systems have distinct characteristics from other domains, say desktop applications, that present unique evolution challenges. One of these challenges is their architecture: a web system is split between a client side and a server side, with possibly complex interactions between the two.[2] Another one is the use of the web browser as client platform. A web system needs to support several different web browsers consistently, whereas each of these browsers is evolving at a rapid pace. Also, the scope and complexity of standards that affect browsers is increasing with each iteration.

A related issue for modern web systems is the challenge to accommodate a wide range of devices. Previously, web development could assume a "classical" desktop browser as its target and design for this target in terms of technologies, user interface, user interactions, etc. With the emergence of smartphones and other mobile devices, web systems should be equally appealing across all of them, regardless of

[1] A graphical illustration of the evolution of browser versions and (web) technologies can be accessed at `http://evolutionofweb.appspot.com/?hl=en`.

[2] Conceptually, web systems adhere to the client-server model. This model is useful to understand the high-level architecture and the split in functionality between the web browser (client) and the back-end (server). However, it should be noted that the concrete architecture of a web system can differ in the sense that it may utilize multiple servers, such as for load balancing or for realizing a three-tier architecture that separates functionality into web browser, application server and database server. Also, a web system may be composed (or "mashed-up") of several services (accessible via web APIs) and each service may be hosted on a different server.

the devices' form factors. Approaches such as *responsive web design* address this challenge [562].

Lastly, modern web systems provide diverse content, often multimedia, along with sophisticated functionality for navigating and manipulating that content by the user. Access and manipulation of content are governed by complex business rules and processes. Due to the nature of web applications, both content manipulation and business rules are tightly interwoven.

To summarize, evolution challenges that are often pronounced in the web domain are:

- rapid churn of standards, technologies and platforms;
- rapid change of requirements and (domain) assumptions;
- ad-hoc development practices, which lack well-defined processes;
- complex interactions between client and server side that is difficult to comprehend, analyze and trace;
- use of multiple (web) technologies with complex interactions among them;
- support of multiple browsers and assurance of browser-safeness; and
- support for multiple devices with a wide spectrum of form and performance factors, including processing speed and connection bandwidth.

The evolution of web systems is caused by different drivers. While this chapter has a strong focus on tools and techniques in the context of changing web technologies (i.e., technological evolution), one should keep in mind that there are other drivers as well that are interacting with technological aspects and that also have a strong impact on the web's evolution. Examples of such drivers are consumers' satisfactions and demands, market competition among web-based *business models*[3] and e-commerce platforms, and laws and regulations (e.g., in the public administration domains). Depending on the web system's domain, the key drivers can differ, but regardless of the domains, technology serves as an enabling factor for evolution. In the following, we briefly reflect on important drivers and how they interact with technology.

Originally, the web's purpose was centered on the dissemination of (scientific) information and consequently the early web mostly had brochure-ware sites (cf. Section 7.2.1). Over the years, the web has seen an increasing commercialization driven by online shops with novel business models as well as traditional "bricks and mortar" businesses that started utilizing the web as a new sales channel ("bricks and clicks") [702]. As a result, the web presence of a company can represent an important (intangible) asset that may significantly affect its revenue and goodwill.

The concept of web applications along with improved technological capabilities (e.g., HTTPS, CSS, JavaScript, and plug-ins such as Flash) enabled organizations to establish and innovate on virtual stores and to offer increasingly sophisticated e-commerce capabilities. In this evolution, technology and business models are cross-fertilizing each other. User-generated content (UGC) is an example of a concept that was enabled by an interplay of both technology and business drivers. Blogs are

[3] While business models are typically associated with commercial gain, they can be defined as describing how an organization captures value, which can be economic, social and cultural.

an early example of UGC on the web, which was also commercially exploited by companies such as Open Diary (launched in 1998) and Blogger (launched in 1999); later examples of UGC are Wikipedia, YouTube and Facebook.

UGC enjoys high popularity with users, which has prompted many web systems to develop business models that entice users to provide diverse content, including personal information. UGC is also often highly interactive and real-time. By necessity, UGC is stored on the server, not the client. In effect, such web systems are now described as *hosted services* accessible through a cloud-based web application (cf. Section 7.2.5). These kinds of applications are often tightly coupled with service models on different levels: software (SaaS), platform (PaaS) and infrastructure (IaaS). Hosted services can utilize convenient payment functionality, ranging from more traditional credit-card services over online payments systems (e.g., PayPal and Google Wallet), to micro-payments (e.g., Flattr). As a result, desktop applications are increasingly replaced by, or alternatively offered as, hosted applications on a subscription bases (e.g., Microsoft's Office Web Apps and Adobe Creative Cloud).

The above developments, among others, have driven technological innovations in the areas of server architectures, browser features, caching, virtualization, (agile) software engineering methodologies, programming/scripting languages and their efficient compilation/interpretation, web-development platforms and frameworks (e.g., Ruby on Rails), and API design.

Both the web's reach and commercialization have contributed to the fact that legal issues are now an important concern [463] [466]. Legal issues are a driver in the sense that it restricts features and innovation in business models and technology. For example, copyright law has been at the center of many disputes around innovations [513]; examples on the web are deep and inline linking to other sites, framing of other sites, reverse engineering of client-side code, time-shifting (MP3.com) and space-shifting (Cablevision) of content [743], and UGC. Web systems that process personal data or UGC have to accommodate privacy, data protection and security concerns, which are partially governed by consumer protection and commercial laws.

7.1.3 Chapter's Organization

The remainder of the chapter is organized as follows. Section 7.2 presents techniques, tools, and challenges of web systems evolution research. The presentation is structured along a historical account of how the web has evolved in terms of the emergence of novel kinds of web systems: static web sites, dynamic web applications, web services, Ajax-based Rich Internet Applications, web systems leveraging cloud computing, and HTML5-based web systems (cf. Sections 7.2.1–7.2.6, respectively). For each kind of web system, where applicable, we highlight the most important research achievements in terms of state-of-the-art techniques and tools as they were proposed at the time.

In Section 7.3 we then focus on architecture, design and technology evolution of web systems. These three dimensions represent major challenges of web systems evolution research. Prominent challenges of architecture evolution are the migration of a web system towards SOA (cf. Section 7.3.1.1) or MDD (cf. Section 7.3.1.2); challenges of design evolution are the refactoring of a web system's design to meet new requirements (cf. Section 7.3.2.1) and to improve upon a certain quality, such as usability (cf. Section 7.3.2.2); a challenge of technology evolution is the migration towards a new platform and/or technology such as Ajax (cf. Section 7.3.3).

Section 7.4 provides a concise overview of the research topics of web systems evolution, including the topics covered in Sections 7.2 and 7.3. This section also describes evolution research topics that are unique for the web domain. Section 7.5 identifies research venues and journals as well as outstanding dissertations for further reading, and Section 7.6 concludes the chapter with parting thoughts.

7.2 Kinds of Web Systems and their Evolution

This section describes the different kinds of web systems that have been targeted by web systems evolution research: static web sites, web applications, web services, Ajax-based Rich Internet Applications, and cloud computing. These web systems—and the accompanying major research topics and challenges—are introduced in the following subsections as they have emerged over the history of the web. This structuring should allow readers that are not familiar with the overall research to better place and assess individual publications and research achievements. This section also highlights that each evolutionary step of the web itself had a corresponding impact on evolution research.

To better understand and classify approaches for web systems evolution, one can distinguish between different *views*—client, server/deployment or developer—that an approach supports [469]. These views address the user perspective of an approach or tool in the sense of what kinds of information are presented to web developers and maintainers.

Client view: The view of the web system that a user sees (typically using a web browser). For this view, information can be obtained by automatically crawling[4] the web system, which is accomplished without requiring direct access to a web system's sources: The web system has to be treated as a black box, only its output in terms of served web pages can be observed and analyzed.

Server/deployment view: The view of the web system that a web server sees (accessing the local file system). This view provides access to the web system's sources (such as, HTML pages, Common Gateway Interface (CGI) scripts, JavaServer Pages (JSP), PHP: Hypertext Preprocessor (PHP) scripts, and configuration files).

[4] Extracting facts from a web system based on the client view is called crawling or spidering.

Developer view: The view of the web system that a developer sees (using a web development tool such as Dreamweaver, or an IDE such as Eclipse, and a web server or an application server such as Apache or Apache Tomcat, respectively). This view is, by necessity, dependent on the tool's abstractions and features.

The three views introduced above are all of potential interest for web systems evolution. For example, the developer view shows the high-level web design such as information about templates; the server view is the one the web server uses and thus important for server maintenance and security; finally, the client view is the one that the end user sees and thus is important to assess external quality factors of the web system, such as navigability, learnability, accessibility, and usability. For effective web systems evolution an approach should ideally support all three views and track dependencies among them.

7.2.1 Static Web Sites

The first technological wave of the web consisted of static web sites that were primarily coded in HTML (so-called *brochure-ware* web sites [850]). A seminal paper raised awareness and popularized the notion that the web was predisposed to become "the next maintenance mountain" [141]. As a starting point for further evolution research, it was recognized that features of web sites could be conceptually mapped to software and, hence, that there was a need for web site evolution research in areas such as development process, version management, testing, and (structural) decay.

One key focus of research at the time was on metrics and (link) structure of web sites. Metrics for web sites typically analyze the properties of the HTML code. Actual metrics are often inspired by software and/or hypertext metrics. The evolution of a web site can then be tracked by analyzing historical snapshots and their associated metrics [141] [909]. The link structure of a web site is similar to the call structure of a program written in a procedural programming language. The nodes of the graph represent web pages and directed arcs between nodes represent a hypertext link between them. Different node types can be used to distinguish between HTML pages, image files, 'mailto:" URIs, etc.

The graph can be constructed by crawling the web site, starting from its home page.[5] One such tool adapted a customizable reverse engineering tool, Rigi [465], with functionalities for the web-domain. It allowed interactive exploration of the link structure of a crawled web site and to apply automated graph layout algorithms (cf. Figure 7.1) [569]. Based on such graph structures static properties can be verified, such as unreachable pages (i.e., pages that are available at the server side, but not accessible via navigation from the site's home page) and the shortest path to

[5] Typically there is the assumption that all pages are reachable from the home page. However, there are also analyses to detect unreachable pages (see below).

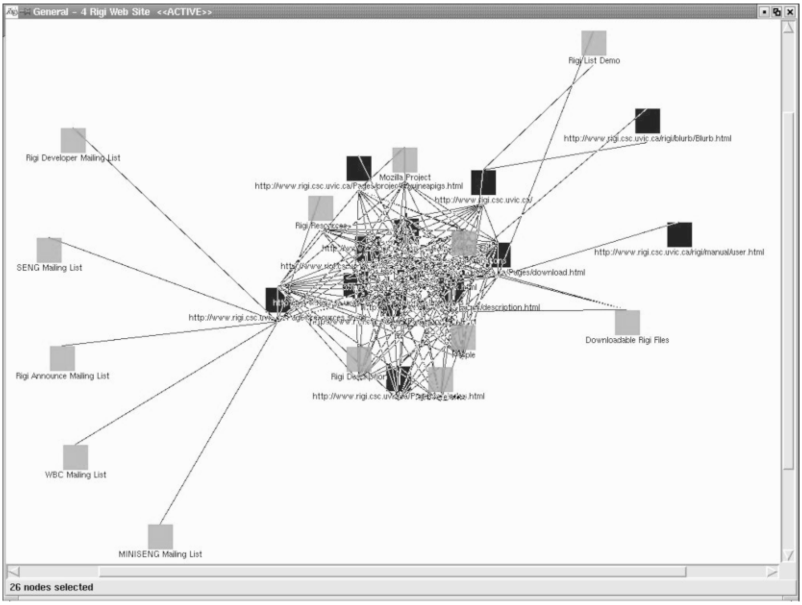

General - 4 Rigi Web Site <<ACTIVE>>
Rigi List Demo
http://www.rigi.csc.uvic.ca/rigi/blurb/Blurb.html
Rigi Developer Mailing List
Mozilla Project
http://www.rigi.csc.uvic.ca/Pages/projects/guineapigs.html
http://www.rigi.csc.uvic.ca/
Rigi Resources
http://www.rigi.csc.uvic.ca/rigi/manual/user.html
SENG Mailing List
Pages/download.html
description.html
http://www.rigi.csc.uvic.ca/
Downloadable Rigi Files
Rigi Announce Mailing List
Rigi Description
http://www.rigi.csc.uvic.ca/Pages/index.html
WBC Mailing List
MINISENG Mailing List
26 nodes selected

Fig. 7.1: Link structure of a web site consisting of 651 nodes rendered with the Rigi tool [569].

each page from the homepage [717]. The latter can be useful, for instance, for a rudimentary usability assessment.

7.2.2 *Web Applications*

Over the years, new web sites emerged (or existing web sites evolved) to support dynamic behavior both on the client-side (e.g., via JavaScript) and the server-side (e.g., via CGI and PHP).[6] This new breed of web sites were termed *web applications*. In order to accommodate the increasing sophistication of web applications over the years—which is also a reflection of the Web 2.0 (Chapter 6)—, the research

[6] Scripting languages have always played a prominent role in realizing web systems. On the server side, before dedicated scripting languages such as PHP and JSP became available, Perl was a popular approach for ad-hoc composition of web sites. (In 1999, Perl has been called "the duct tape of the Internet" [359].) Since around 2010 the ability of server-side JavaScript has gained momentum (e.g., the Node.js library). Its proponents want to close the conceptual gap between client and server technologies.

literature has also taken up the term Rich Internet Applications (RIAs) to distinguish these technically complex web applications from the more primitive ones.[7]

RIAs are web applications that are characterized by a user experience that is highly interactive and responsive so that they can rival the experience that desktop applications can offer. In this respect, the "rich" in RIA refers to the complexity of the underlying data that the user can manipulate as well as the user interface itself. The client side of RIAs is typically realized with a combination of JavaScript, CSS and HTML. While web sites use little JavaScript that is often self-contained and hand-coded, web applications often use a substantial amount of JavaScript that builds on top of existing libraries and frameworks (e.g., jQuery and Dojo). Compared to early web applications that can be characterized as thin client, RIAs are realizing more of the web system's functionality on the client side (i.e., fat client). Furthermore, links are often encoded with client-side scripting and their targets have no obvious semantic meaning [602]. As a consequence, such web applications cannot be simply crawled and understood based on a static link structure anymore.

Static web sites, which have HTML-encoded links, are straightforward to crawl (and because of this many tools could afford to implement a custom solution for this functionality). However, with the introduction of more and more dynamic web applications with scripted links these crawlers became very limited because the navigation model that they are producing reflects an increasingly smaller subset of a web system's whole navigation space. A web application is often based on events that trigger JavaScript code that manipulates part of the current page's Document Object Model (DOM), in effect causing a state change in the web application. At the extreme, a single-page, Ajax-based web application may not even offer a single static link, resulting in an empty navigation model for a traditional crawler. Another problem that makes it difficult or impossible to construct a complete model is the "hidden" web caused by interactive query forms that access a server-side database.

Since many analyses for web systems evolution are based on the client view an accurate crawler is highly desirable. Unfortunately, crawling techniques did consistently lag behind the latest web systems and handling the dynamic features was addressed only inadequately for many years. The Crawljax tool, introduced in 2008, offered a solution to this problem [602]. It automatically constructs a state-flow graph of the target web system where different states are based on comparing the states' DOM trees. State transitions are performed by a robot that simulates actions on "clickable" elements (i.e., DOM elements with attached listeners). However, the tool's authors caution that "there is no feasible way to automatically obtain a list of all clickable elements" [602]. State changes are determined by an edit distance between the source and target DOMs. The edit distance uses a similarity threshold that can be varied by the tool user, where one possible setting corresponds to matching for identical trees. Besides the edit distance's threshold other settings can be used to control the crawling behavior such as maximum number of states and ignoring of certain links based on regular expressions. ReAJAX is another example of a sophisticated crawler based on a similar approach than Crawljax (cf. Section 7.2.4).

[7] However, it should be noted that RIA is not clearly defined and different authors attach different meanings to it.

In response to the advent of web applications, new approaches were developed to capture the increasingly dynamic behavior of web sites and their increasing heterogeneity in terms of the employed standards and technologies. This research met a need because development tools lacked in functionality for web site evolution: In 2001, a study of two popular web development tools at the time (FrontPage and Dreamweaver) showed that they had rather limited support for understanding and reverse engineering of web sites and that support was mostly restricted to static features [850]. Maintenance activities that were supported by these tools at the time are, for example, validation of HTML and XML documents, reports of usage-violations of ALT and META tags, link checking, metrics that summarize characteristics of web pages, and page download-time estimates.

In order to provide suitable information to reverse engineers who have to understand ASP-based sites, Hassan and Holt extract information from HTML, VBScript, COM source code, and COM binaries [374]. During the extraction process, each file in the local directory tree that contains the web site is traversed, and the corresponding file's extractor (depending on the file type) is invoked. All extractors' output is first consolidated into a single model and then visualized as a graph structure. This graph structure provides an architectural view of the web system that can be used as a starting point for evolution (cf. Figure 7.2). There are also approaches that combine static and dynamic analysis. For example, one proposed method leverages an extension of UML to show the architecture of the web application as class diagrams and its dynamic behavior with sequence and collaboration diagrams [242].

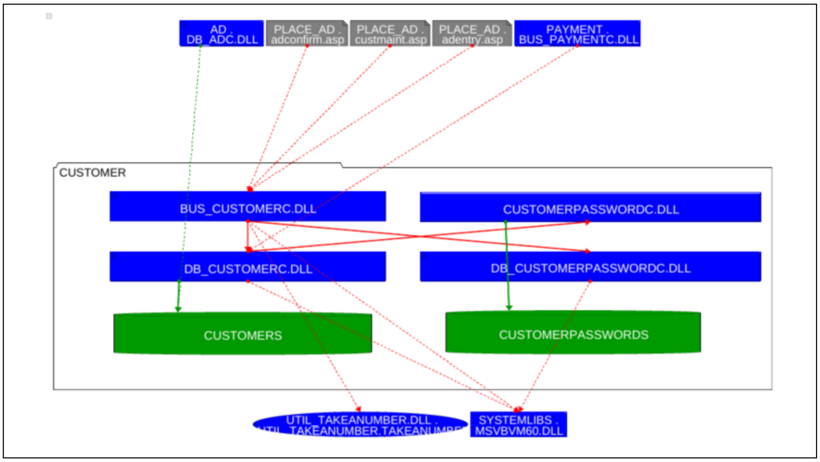


Fig. 7.2: Architectural view of a web system's components: Blue boxes are DLL files, gray boxes are ASP files, blue ovals are COM objects, green tubes are databases [372].

To improve evolvability, restructuring of server-side code has been proposed. For instance, Xu and Dean automatically transform legacy JSP to take advantage of an added JSP feature—the so-called custom tag libraries—to improve future maintainability by more clearly separating presentation from business logic [933]. Research has also tackled the migration away from static, HTML-only sites towards dynamic ones. For example, Estiévenart et al. have a tool-supported method to populate a database with content extracted from HTML pages [281]. This database can then be used to build and serve pages dynamically. Ricca and Tonella have realized a conceptually similar approach [718].

Web applications, and especially RIAs, are often developed with sophisticated frameworks and tools that provide higher-level concepts, which then need to be realized with a generator that produces code that can be executed by the web server. An unusual example is the Google Web Toolkit: it allows coding in Java with dedicated APIs and widgets and this code is then compiled to optimized JavaScript. Dedicated functionality for web site evolution can be added to tools if their architecture is plug-in based. Such an approach has the advantage that the user can work within the developer view. The REGoLive tool adds reverse engineering capabilities to the Adobe GoLive web authoring tool [354]. For example, REGoLive provides a graph-based structure of a web site, showing artifacts—including web pages, CSS files and JSPs as well as tool-specific entities such as templates and so-called smart objects—and their dependencies.

RIAs typically make extensive use of JavaScript on the client side and the resulting code base can be significant.[8] For instance, Google's GMail has more than 400,000 lines of hand-written JavaScript [430]. Thus, JavaScript is an important consideration for web systems evolution. It is a dynamic, weakly-typed language that offers an interesting combination of language features, mixing imperative, object-based and functional programming with concepts such as mutable objects, prototype-based delegation, closures, and (anonymous) functions objects. In JavaScript pretty much everything can be manipulated at run-time (introspection and intercession), there is no information hiding and there is "eval" functionality that allows to execute an arbitrary string as JavaScript code. As a result, JavaScript features make it difficult for static analyses to produce meaningful results, and dynamic analyses are a more promising approach for analyzing the behavior of JavaScript.

JavaScript has no explicit concept of classes and as a result various idioms are used to mimic this concept. If multiple idioms are used in a single code base maintainability becomes more difficult. Gama et al. studied 70 systems and found five different idioms in practice [310]. Based on these idioms they developed an automated code transformation that normalizes a code base to a common idiom. The authors observe that "there seems to be remarkably little other work on JavaScript style improvement" but one would expect that research interest in this area will pick up in the future. Another transformation example is an approach and tool for extracting a subset of client-side JavaScript code that encapsulates a certain behavior [560]. With this dynamic analysis a certain (usage) scenario such as using a UI widget is

[8] RIAs can be also realized without JavaScript if they are based on proprietary technology (e.g., Adobe Flex or Microsoft Silverlight).

first interactively executed and tracked. Based on this run a dependency graph is constructed that contains HTML, CSS, and JavaScript nodes along with their structural, data and control flow dependencies. This enables to extract a self-contained subset of the code that is able to reproduce the usage scenario. This approach can be also used for dead code removal (e.g., to speed up page load time) if the scenario is able to capture all expected behaviors of the web application.

7.2.3 Web Services

Around the time that web applications established themselves, the concept of *web services* started to become more prominent. The move towards web services was mostly driven from a business perspective that envisioned cost savings and increased flexibility [849] [20]. Web services are closely related to Service Oriented Architecture (SOA) in the sense that web services are an enabling technology for realizing a system that adheres to the service-oriented architectural style [830] [592, Chapter 7]. From this perspective, migration towards web services can be seen as architecture evolution and is discussed in more detail in Section 7.3.1.

Evolution of a web service entails significant challenges: distributed components with multiple owners, distributed execution where multiple workflows are executed concurrently, and machine-generated description files (e.g., WSDL, XSD and BPEL) [918] and messages (e.g., SOAP messages). Understanding a Web Service Description Language (WSDL) specification can be complex because it contains a number of concepts (i.e., types, messages, port types, bindings and services) that can be highly interrelated via referencing. Since WSDL provides a high-level description of the important aspects of a web service, it plays an important role when a service—or a system that uses the service—is evolved.

Examples of analyses based on WSDL files are clone detection of services and enabling of automated service discovery [568] [348]. To obtain meaningful results, the WSDL files are first suitably restructured (so-called contextualization) by inlining referenced information. This allows to apply established algorithms such as topic models [348] and near-miss clone detection [568] to find similar operations. These similarities can be used as input for maintenance activities, but also for web service discovery (i.e., finding an alternative service).

Fokaefs et al. [298] present an empirical study on the evolution of web services by applying a differencing technique to WSDLs. The WSDL files are first suitably stripped to form another valid XML representation that reflects the client's perspective of the service. Pairwise differencing of the XMLs are then performed by the VTracker tool, which is based on a tree-edit distance algorithm that calculates the minimum edit distance between two XML trees given a context-sensitive cost function for different edit operations. The evolution of the successive WSDLs can be studied with the percentage distribution of change, delete and insert operations. Among other cases, the authors have tracked the evolution of 18 WSDL versions of Amazon's Elastic Cloud. The analysis showed that the service underwent rapid expansion (additions are dominating) while the existing interface was kept relatively stable (deletions and radical changes are avoided). The authors could also find correlations between WSDL changes and business announcements.

SOAMiner is a static analysis tool for searching and indexing a collection of service description files, including WSDL [918]. It is based on Apache Solr, which provides a *faceted* search infrastructure. Examples of search facets are file types and XML tag names. A user study found that "faceted search capability proved to be very efficient in some cases" and that this approach can be more effective compared to text-based searching because the latter requires more domain knowledge.

The authors of SOAMiner also present a dynamic analysis tool. The feature sequence viewer (FSV) visualizes sequence diagrams of messages passed between web services [918]. The sequences are extracted from trace data and a simple heuristic is used to determine messages that are relevant for a certain scenario. De Pauw et al. have also realized an approach based on trace data that relies on message content and timestamps [225]. Their analysis is able to find identifying keys such as order numbers, which are used by the web services to correlate messages and to realize possibly asynchronous workflows. This enables to find "semantic correlation" between different kinds of messages (e.g., an order number that is first used in a message to order an item and then, in a subsequent message, to confirm its shipment). In effect, sequence diagrams, which only show the control flow (such as the FSV tool, see above), are augmented with dependencies that show correlations of message content.

As described above, a standard approach for a web service is to utilize WSDL and SOAP, which specify the service's contract and API in terms of permissible invocations and required data types. This flavor of web service is also referred to as *WS-* web service* [683]. Alternatively, the API of a web service can be also realized with Representational State Transfer (REST) [293]. The basic approach is to design the data model as a network of data items, or so-called resources, and to define URIs to access, navigate and manipulate these resources with HTTP request methods (i.e., GET, PUT, POST and DELETE). Typical data encodings for such RESTful web services are JSON or XML, but plain text is also conceivable. Alarcón and Wilde propose the Resource Linking Language (ReLL) to model RESTful web services [15]. They have developed a crawler, called RESTler, that can extract a resource network from a RESTful web service based on the service's ReLL model. The obtained resources in combination with the ReLL models can then be used for composing new, mashed-up services.

According to Pautasso and Wilde, "it is not possible to simply say that one variety is better than the other, but since RESTful web services gained momentum, it has become clear that they do provide certain advantages in terms of simplicity, loose coupling, interoperability, scalability and serendipitous reuse that are not provided to the same degree by WS-*" [683]. Thus, support for migrations from WS-* to RESTful web services, and vice versa, is desirable. Strauch and Schreier present RESTify, a semi-automated approach to transform a WSDL-based web service to a RESTful design [798]. The approach supports, for instance, the identification of resources from a WSDL description, and the definition of resource access by defining a mapping from a WSDL operation to URI request sequences. Research has strongly focused on REST as migration and reengineering target (e.g., [534] [816] [798]), neglecting the opposite direction.

7.2.4 Ajax-based Web Systems

Another major evolution of the web is marked by the possibility of a web application to initiate an asynchronous connection to obtain data and presentation information (i.e., *Ajax programming* [314]). RIAs that employ Ajax frameworks and technologies result in highly sophisticated web applications whose functionality rivals and surpasses native desktop applications. Compared to traditional web sites and web applications in which contents and functionalities are distributed among several pages that the user navigates, an Ajax-based RIA might consist of one single web page whose content and user functionalities change dynamically in consequence of the user actions, without any page reloads. More frequently, however, Ajax-based RIAs are a combination of traditional web applications and enriched user-interface features, which are made possible by Ajax technology. JavaScript is used as the main coding language to implement features, both on the client and, increasingly, on the server side.

Of course, for this new stage, yet again novel approaches are needed for effective web systems evolution. ReAJAX is a reverse engineering tool for single-page Ajax RIAs [561]. Similar to Crawljax (cf. Section 7.2.2) the web system is dynamically executed to construct a state model based on the pages' DOM representations. However, with ReAJAX the model extraction can be customized by abstracting the DOM representation with a function that maps DOM elements to higher-level values that strives to capture the state of the application based on its GUI elements. For instance, for a certain application it may make sense to abstract an HTML table with three distinct values that represent an empty table, a table with one row, or a table with more than one row. State transitions represent a change in the abstracted DOM rather than the actual DOM and hence can represent meaningful states in the application's logic. The state model is constructed by manual or automatic execution of a set of execution scenarios. The quality of the resulting state model varies depending on the coverage of the scenarios and the suitability of the DOM abstractions. CReRIA is another example of a reverse engineering tool that is based on dynamic analysis and creates a state model [26] [27].

Research has also tackled the migration from legacy web systems towards RIAs and/or Ajax-enabling them ("ajaxification"). In 2007 Mesbah and van Deursen observed that "today we have a new challenge of migrating classic web applications to single-page web applications" [601]. This kind of migration can be seen as technological evolution as discussed in Section 7.3.3.

7.2.5 *Web Systems Leveraging Cloud Computing*

Arguably, cloud computing marks another major step in the web's evolution. However, it should be stressed that cloud computing is an independent principle that applies to software systems in general. The "running [of] applications within a network server or downloading the software from the network each time the software is used" is one of its prominent characteristics [817]. Cloud computing can be defined as "a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [585]. Importantly, the access to these resources needs neither be realized with a browser-based user interface nor with web-based technologies.

Tupamäki and Mikkonen point out that there can be mismatches between web principles and cloud computing (e.g., single-server design) [861]. Still, existing web systems are among the most promising candidates to evolve towards the cloud—or often have already properties that are now associated with mainstream cloud computing. Ajax-based RIAs typically provide an application/service along with storing the corresponding user-data on the server-side (e.g., webmail such as Yahoo! Mail and GMail, and storage such as Dropbox and Microsoft's SkyDrive)—properties that are labeled as Software-as-a-Service (SaaS) by cloud computing.[9] The underlying service level, Platform-as-a-Service (PaaS), can be seen as web-based middleware to realize SaaS. For example, Windows Azure and Google App Engine provide a platform of different services, which can be realized to build SaaS-based web systems based on an integrated set of diverse APIs. Thus, similar to SOA and web services, different (heterogeneous) PaaS and Infrastructure-as-a-Service (IaaS) based services can be orchestrated to realize a higher-level service.

Every advent of a new technology poses the challenge of how to best utilize it in the context of an existing (legacy) system. Established techniques for transitioning to a new technology include wrapping, migration, and reengineering (cf. Section 7.3). In this context the potential mismatch between the principles and architectures of web application frameworks and cloud computing platforms present a significant challenge [861]. Kienle et al. observed in 2010 that "surprisingly, there is still comparably little discussion of [web] system evolution towards the cloud such as when and how systems should be migrated to the cloud and techniques to accomplish such a migration in an effective manner" [462]. This observation still holds true as of this writing.

[9] HTML5 enables clients to utilize local storage; thus, the location of (personal) data may partially move from the server to the client. However, leveraging this capability may not be in the interest of the service provider.

7.2.6 HTML5-based Web Systems

The HTML5 standard is still at a draft stage and far from being finalized, but it is already well on its way to become a ubiquitous platform for building all kinds of (web) systems. Thus, it can be expected that HTML5 will play a significant role in web systems evolution.

It appears at this point that there are two major competing approaches: HTML5-based web systems, which are vendor-agnostic and represent the *open web*, and native web systems, which are vendor-specific [608]. Examples of the latter are *apps* running on Apple's iPhone and iPad or Google's Android devices. While these apps can utilize web-based protocols and principles (e.g., HTML and REST) and while their look-and-feel can be similar to browser-based systems, they are built with vendor-specific platforms and native graphics libraries, and are typically distributed in binary form. It remains to be seen if both approaches will coexist or if one will extinguish the other. In the following we focus on open web systems only, because in many respects native apps are close to traditional desktop software.

HTML5 significantly expands the scope of previous HTML standards with the aim to enable web systems to come even closer to native desktop applications (without having to rely on browser plug-ins) [608]. Specifically, web systems can utilize local storage and built-in audio/video support, temporarily function without network connection, draw on a 2D canvas for procedural, interactive graphics, and render hardware-accelerated (3D) graphics. HTML5's capabilities may cause development to the move away from native, vendor-specific apps, meaning in effect that "browser technology will dominate over operating systems" [50]. This would further strengthen the trend towards truly web-based software.

In the past, web systems evolution research did address the migration from web applications to early mobile devices because this was rightly perceived as "the next big thing" [410]. If the trend towards HTML5-enabled mobile applications holds true, then research should tackle the migration from mobile applications written for native platforms to HTML5 technologies.

7.3 Dimensions of Evolution

As the previous section illustrates, research in web systems evolution has come up with a rich set of approaches and techniques, typically with accompanying tool support. In the following, we structure the discussion along three dimensions of evolution: architecture (cf. Section 7.3.1), (conceptual) design (cf. Section 7.3.2) and technology (cf. Section 7.3.3). Each of these have in common that an existing web system is migrated or reengineered, and, in any case, updated towards a new system. Depending on the dimensions, the new system differs in its architecture, design, or technology compared to the old one. However, it should be stressed that these dimensions are not orthogonal to each other, meaning that evolution in one dimension can imply or require evolution in the other ones as well.

7.3.1 Architecture Evolution

Evolving the architecture of a system means a fundamental change on how functionality is distributed into components and how these components communicate among each other. For instance, a monolithic system may be "split" into (loosely coupled) components for user interface, application/business logic, and data persistence, by using a client-server three-tier architecture with the client side running the user interface component [157]. Also, such a three-tiered client-server system may be further evolved to move part of the business logic onto the client. Such an architecture evolutions often also impacts the system's technology. In the previous example, the client-server architecture may be realized with web-based middleware technology (such as Java Business Integration or IBM Integration Bus), the business logic to be moved onto the client side may be implemented by using a client-side scripting language (such as JavaScript), and data persistence may be realized using some data persistence framework (such as Hibernate).

7.3.1.1 Towards Service Oriented Architecture

As mentioned before (cf. Section 7.2.3) web services are a popular approach—but not the only one—to realize a system that adheres to SOA. Since our focus is on web systems evolution, we restrict our discussion mostly to web services. However, there are generic approaches for migrating systems to SOA (e.g., SMART [518] and Heckel et al. [592, Chapter 7]) that can be instantiated to suit the context of web systems migration: the existing system is then any kind of web system and the SOA-enabled target system is based on web services. Generally, the purpose of migrating towards web services is the identification of existing (higher-level) functionality, isolating and extracting them from the existing system and making them accessible via web service interfaces.

One design principle for SOA—and, by extension, for web services as well—is composability where each (basic) service provides a well-defined and self-contained (business) need. An application is then realized by service composition. Web services are realized with WSDL (to specify their interfaces) and Simple Object Access Protocol (SOAP) (for messaging). A more lightweight approach for web services based on so-called web APIs eschews WSDL/SOAP in favor of HTTP-encoded request messages and JSON for data exchange. The latter approach is often used in combination with REST (cf. Section 7.2.3) and Ajax (cf. Section 7.2.4).

In 2008, Hainaut et al. observed that migration to SOA "appears as one of the next challenges" [592, Chapter 6, page 109]. Indeed, migrating existing code towards web services is a common evolution scenario, and Almonaies et al. present a survey of approaches [21]. Arguably, a web site may already offer "services" through HTTP response/request pairs that are initiated through user interactions. Jiang and Stroulia analyze these interactions with pattern mining to semi-automatically identify service candidates [441]. Almonaies et al. present a semi-automatic approach to migrate a monolithic PHP-based web application towards web services [20]. Potential business services are first identified and separated, then PHP code is restructured to expose functionality as services (based upon the Service Component Architecture (SCA) specification, which is available as a PHP implementation) and to allow communication between services (based upon SCA's Service Data Objects).

A migration does not necessarily start with "web-enabled" code, it may as well be an EJB application that is rearchitected towards a system composed of a set of web services [531]. Another scenario is the wrapping of COBOL mainframe programs so that their functionality is exposed as web services [782]. Compared to a full-fledged reengineering approach, "the complexity of the wrapping techniques is low, since there is no deep analysis of the legacy system" [21].

7.3.1.2 Towards Model-Driven Engineering

The emergence of the web as a prominent platform to develop and deploy systems and services—i.e., in other words cloud computing's SaaS, PaaS and IaaS (cf. Section 7.2.5)—and the increasing complexity of these systems has led to the need of tools and methodologies that streamline their design, development, and maintenance. Model-Driven Engineering (MDE) advocates the systematic use of models and model transformations throughout the life cycle of a software system, from requirement specification, to design, implementation (usually envisioned as automatic code generation), and maintenance [756]. By promoting automation of tasks and reuse of artifacts, MDE aims to increase productivity and to reduce development time and costs. One of the better known MDE initiatives is the Model-Driven Architecture (MDA) software development framework defined by the OMG [650]. MDA proposes a particular approach for realizing MDE by leveraging OMG standards (e.g., UML and MOF).

The engineering of web systems is a specific application domain in which the principles of MDE have been successfully applied, originating the rich and lively research area of Model-Driven Web Engineering (MDWE). By decoupling the functional description of applications (models at various levels of abstraction) from their implementation (code for a given platform) and by enabling the modification of the first and re-generation of the last out of the first, MDWE methods ease the evolution and adaptation of web systems to continuous changing requirements and emerging technologies. In this process, model evolution plays an important role (cf. Chapter 2)

The list of web engineering methods which natively adopt a model-driven approach to the development of web applications or which have been extended towards MDE includes: UWA [252], WebML [750] and its extension towards RIAs [304], UWE [480] and OOHDM [752], and OOWS [299]. Valderas and Pelechano present a comparative study of these and other MDWE methods which analyzes the techniques they propose for specifying functional, data and navigational requirements as well as the mechanisms provided for automatically translating these requirements into conceptual models [866]. According to their survey, only the WebML model explicitly addresses evolution concerns in a dedicated "Maintenance and Evolution" phase.

The reverse engineering of an existing web application with one of the above listed methods opens the door to reaping the benefits of MDE. An approach that enables recovering user–centered conceptual models from an existing web application according to the UWA methodology [865] is RE-UWA [98]. The approach is able to recover the content, navigation, and presentation models of the application, as observed from a user's perspective. These models can provide effective support for maintainer when deciding on some change/improvement to be applied to the application, with respect to its external, user-perceived, quality. The recovered models, eventually evolved according to new requirements, can be used then as input for a UWA model-driven forward engineering process, which is supported by dedicated tools [97]. Similarly, an approach that abstracts WebML hypertext models [137] from existing web applications is proposed by Rodriguez-Echeverria [724]. The approach can be used as the initial step towards modernizing the existing application into a RIA-based one using the WebML model-driven web engineering method [304].

7.3.2 Design Evolution

Often the intent of evolution is improving some external and user-perceivable quality characteristics of a system, rather than properties related to its internal realization. When this intent applies, the evolution process is usually first accomplished (conceptually) at the design level with the help of a (web) design model (cf. Section 7.3.1.2), to move then to the implementation level where it may be realized by selecting suitable technologies or algorithms.

In this section we present two approaches for web application design evolution: a redesign approach and a refactoring approach. The intent of the former is to modify the behavior of the web system while the latter preserves it.

7.3.2.1 Meeting New Requirements

When evolution is driven by the need to modify the behavior of the application (e.g., to implement new business rules and meet new or evolved requirements [592, Chapter 1]) or by the opportunity to improve aspects influencing the external quality characteristics of the application (e.g., its usability), evolution should be carried out at the design level. To this aim, it is desirable to rely on approaches that enable (1) to recover the current design model of a web application, and (2) to modify the recovered design model in order to effect the evolution's goals. RE-UWA is such an approach for the domain of web systems design evolution (already discussed in Section 7.3.1 in the context of MDE) [96].

The RE-UWA design evolution approach is based on a three-step process:

Reverse Engineering: A semi-automatic reverse engineering phase is used to analyze the HTML pages of the system's front-end with the goal to abstract its "as-

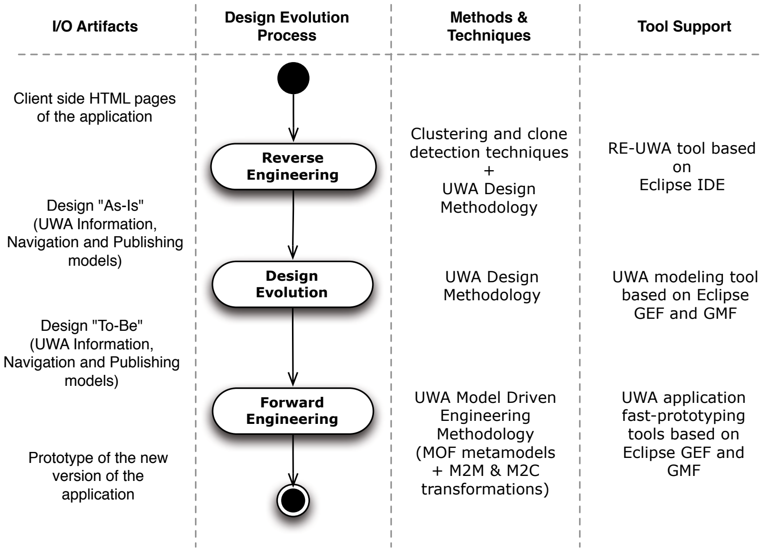

I/O Artifacts
Design Evolution Process
Methods & Techniques
Tool Support
Client side HTML pages of the application
Reverse Engineering
Clustering and clone detection techniques + UWA Design Methodology
RE-UWA tool based on Eclipse IDE
Design "As-Is" (UWA Information, Navigation and Publishing models)
Design Evolution
UWA Design Methodology
UWA modeling tool based on Eclipse GEF and GMF
Design "To-Be" (UWA Information, Navigation and Publishing models)
Forward Engineering
UWA Model Driven Engineering Methodology (MOF metamodels + M2M & M2C transformations)
UWA application fast-prototyping tools based on Eclipse GEF and GMF
Prototype of the new version of the application

Fig. 7.3: The RE-UWA web application design evolution approach.

is” design. This phase applies clustering and clone detection techniques on the HTML pages of the application and is supported by the Eclipse IDE environment.

Design Evolution: This phase leverages the recovered models from the previous phase. The (new) requirements—which identify shortcomings and opportunities for improvements in the current design—are then used to construct the desired “to-be” design. This phase is supported by a set of modeling tools, which are build on top of the Eclipse Graphical Editing Framework (GEF) and the Eclipse Graphical Modeling Framework (GMF).

Forward Engineering: In this phase, the “to-be” design model is used to produce the “to-be” version of the web system. The UWA fast prototyping tools [97] can be also used to quickly implement a prototype of the new application and to verify/validate the new design.

Thus, the whole approach leverages the UWA design methodology to guide both the reverse and forward engineering design processes, and the UWA design models as the formalism to represent the “as-is” and “to-be” designs of the web system. Figure 7.3 summarizes the whole approach with the involved activities and supporting tools and techniques.

7.3.2.2 Improving Usability

The design of a web application can also be evolved to improve *quality in use* characteristics such as *usability* while preserving its behavior and business rules. To this aim, refactoring techniques can be applied to the design models of the application [316].

Refactoring is a technique that applies step-by-step transformations to a software system to improve its quality while preserving its behavior [596]. Originally introduced by Opdyke and Johnson in the early 90's [664] and mainly focused on restructuring a class hierarchy of some object-oriented design, refactoring became popular a few years later with Fowler's book [301], which broadened the perspective to a wider spectrum of code-centric refactorings and motivated its application for improving internal quality characteristics of software such as maintainability and evolvability. Since then, refactoring has further broadened both in scope and intent, as refactoring techniques have been applied to UML models [804], databases [29], and HTML documents [368]. In all cases the basic philosophy of refactoring has been kept (i.e., each refactoring is a small behavior-preserving transformation), but, as the scope, also the intent of refactoring has expanded to target external and quality in use characteristics of software, such as learnability and effectiveness.

Garrido et al. present catalogs of refactorings for the navigation and presentation models of a web application aimed at improving its usability [315] [316]. Each of the refactorings included in the catalog is characterized by a scope (i.e., the software artifact to which it applies), an intent (i.e., one or more usability factors it aims to improve), and a bad smell (i.e., the symptoms that may suggest applying the refactoring). The usability improvement approach that results from the application of the refactorings in the catalog is agnostic with respect to the method and the technologies adopted to develop the application, as all refactorings are described by showing how they affect the corresponding web page. Table 7.1 summarizes a subset of the refactorings included in the aforementioned catalogs.

7.3.3 Technology Evolution

Technology evolution of a system can be triggered by the retirement of a technology because the vendor supports it no longer, or by the realization that the employed technology no longer matches the system's requirements. In the former case, the vendor may provide a migration tool. For instance, a helper tool supported the migration from IBM's Net.Data legacy technology to JSP [499]. The latter case can mean, for instance, that internal qualities (e.g., maintainability), external qualities (e.g., performance), or both are increasingly difficult or impossible to meet because new requirements and old technology are at a mismatch.

For web systems, each new wave of the web (as described in Section 7.2) has triggered a technological evolution. However, web systems, especially early ones

Table 7.1: A subset of refactorings for usability improvement in web applications proposed by Garrido et al. [315] [316].

Refactoring	**Intent**	**Scope**
Convert images to text In web pages, replace any images that contain text with the text they contain, along with the markup and CSS rules that mimic the styling.	Accessibility	Code
Add link Shorten the navigation path between two nodes.	Navigability	Navigation model
Turn on autocomplete Save users from wasting time in retyping repetitive content. This is especially helpful to physically impaired users.	Effectiveness, accessibility	Code
Replace unsafe GET with POST Avoid unsafe operations, such as confirming a subscription or placing an order without explicit user request and consent, by performing them only via POST.	Credibility	Code
Allow category changes Add widgets that let users navigate to an item's related subcategories in a separate hierarchy of a hierarchical content organization.	Customization	Presentation model
Provide breadcrumbs Help users keep track of their navigation path up to the current page.	Learnability	Presentation model

with limited functionality, were not necessarily migrated in the strict sense but rather rebuilt from the ground up with new technology.

Another technological evolution is driven by mobile devices. They do not only bring new form factors and input devices, but also new technical challenges to keep web systems responsive. Web systems are typically developed with the expectation of a PC and a wired base connection, but smartphones and tablets can be less powerful than PCs and latency is more pronounced for wireless connections. Also, the performance when executing JavaScript on mobile devices is much reduced compared to PCs [944]. As a result, to make web systems responsive and fast for mobile devices, different techniques and algorithms are needed. For example, basic rules for achieving this are reducing of HTTP requests by concatenating files, avoiding redirects, limiting the number of connections, and replacing images with CSS3-based renderings. To accommodate mobile devices, the web system's APIs for the client can be changed and extended to handle device profiles and to allow the client to control limits on data [572].

Towards Ajax

From a technical point of view the introduction of asynchronous connections in the browser is a minor functional addition whose full potential was not recognized until

Ajax programming was proposed (cf. Section 7.2.4). However, as Ajax-based RIAs have demonstrated, Ajax can have a huge impact on the user experience. Consequently, if a web system is migrated towards Ajax it is often done with the goal to improve the system's usability (i.e., a design evolution, cf. Section 7.3.2.2) by taking advantage of Ajax's unique capabilities. Similarly, the prior technical shift towards more dynamic web systems and RIAs was intertwined with the evolution for usability improvements.

Chu and Dean have developed a transformation for "ajaxification" of JSP-based web applications [185]. They are handling the use of a user interface (UI) component to navigate a list of items where only a subset of list items is displayed for each page. A typical example would be the list of search results of a search engine with controls for "previous"/"next" as well as direct links to all result pages. The basic idea goes as follows. The user has to identify and annotate the sections of the JSP code that is responsible for rendering the list (e.g., a loop with a database query and markup code for a table row). The marked code is then sliced to produce an executable subset of the marked JSP code. The slicing criteria can be controlled with suitable annotations. As a result, the sliced code provides, when called, the rendering data that is sent in response to an Ajax request. Also based on manual markup, the JSP source is transformed to make an Ajax call that pulls in the list items. To make the list item's rendering data match with the needed HTML/DOM structure of the hosting page, the transformation has to make suitable adaptations. The approach has been tried out on four applications and for all of them the visual rendering is preserved.

Mesbah and van Deursen propose Retjax, an approach and tool for migrating multi-page RIAs to Ajax-based, single-page web systems [601]. The goal is to find UI components that can be transformed to leverage Ajax. Multi-page RIAs have to build a whole new page whenever information in a UI component changes. A single-page approach, in contrast, would refresh only the UI component, which is embedded within the page. To identify the UI components, first a model of the existing web application is constructed. Retjax uses HTML Parser for this purpose, but this step could employ other advanced crawling and reverse engineering techniques. Based on the model, the navigation paths that users can follow when exercising the web application are followed; the depth of the followed links can be set. When a new page is encountered then all of the target pages are retrieved and only these are clustered based on schema similarity. The schema of each page is obtained by converting the HTML into a well-formed XHTML (with the JTidy tool) and automatically extracting a DTD from it (with the DTDGenerator tool). The clustering is performed with the edit distance between the pages' DTDs using the Levenshtein method [517]. The similarity threshold can be set. On the clustered navigation paths a differencing algorithms determines the HTML structural changes between page source and targets. The result is a set of candidate components where each one is examined for promising UI elements (e.g., button or tab) that can be converted to Ajax. The authors propose to describe the candidates with the help of a generic Ajax-based representation model (which could take inspiration from static UI models such as Mozilla's XUL) that can then be used to drive the transformation from the generic model to a platform-specific representation.

7.4 Research Topics

Sections 7.2 and 7.3 have covered a large part of the research landscape of web systems evolution. In Table 7.2, the references of research contributions that are given in bold face have already been discussed in previous sections.

Table 7.2: Examples of "classical" research topics and selected research contributions.

Research topic	Examples of web-related research	F	M	A	T	V
architecture recovery	of web sites **[569]**			●		●
	of web applications **[373]** **[374]**			●		●
clone detection	in web sites [243]			●		
	in web applications [815] [494]			●		
	in web services (WSDL) **[568]**			●		●
clustering	of web pages via keyword extraction [853]			●		
	of web applications **[601]** [244]			●		
dead/unreachable code	removal of JavaScript code **[560]**			●		
fact extraction	HTML **[569]** **[909]**	●				
	crawling of RIAs (with Ajax) (Crawljax) **[602]**	●				
	crawling of single-page Ajax **[561]**	●				
	crawling of RESTful web services **[15]**	●				
	J2EE web projects (in WebSphere) [464]	●				
metrics	based on HTML code **[909]** **[141]**	●		●		
	based on WSDL differencing **[298]**			●		
migration	from static sites to dynamic web applications **[718]**				●	
	from ASP to NSP [375]				●	
	from EJB to web services**[531]**				●	
	from web application to single-page Ajax **[601]**				●	
	to Ajax **[185]**				●	
	to web services **[20]**				●	
	involving MDE **[98]** **[97]** **[724]**	●	●	●	●	
refactoring	of web applications design models **[315]** **[316]** [154]			●	●	
restructuring	of multilingual web sites [852]				●	
	of JSP code for renovation **[933]**				●	
	of web transactions [848]	●	●	●	●	
sequence diagrams	for web applications [38]			●		●
	for web services **[918]** [225]			●		●
slicing	of web applications **[851]**		●			
testing	of web applications **[717]**		●			
	of web services [783] [780]		●			
wrapping	of web applications with WSDL **[441]**				●	
	of COBOL with web services **[782]**				●	

Legend: F: fact extraction, M: modeling, A: analysis, T: transformation, V: visualization.

Many of these topics have their roots in the more general and traditional areas of software evolution and maintenance research. To illustrate this point, Table 7.2 provides some examples of "classical" research topics (first column) along with research that has addressed—and suitably adapted—these topics for the web's domain (second column). The table also identifies if a research's main contribution lies in the areas of fact extraction (F), modeling (M), analysis (A), transformation (T), or visualization (V). For understanding a certain research contribution, it is often beneficial to know the functionalities that its techniques cover and how these functionalities are relating to each other. Reverse engineering approaches are primarily concerned with fact extraction (F) and analysis (A), whereas forward engineering primarily deals with modeling (M) and transformations (T). Both can also employ visualizations (V) to present (intermediary) results. Examples of analyses in reverse engineering are clustering, clone detection, and architecture recovery; examples of transformations in forward engineering are restructuring, refactoring, dead code removal, wrapping and migratwion.

Besides the "classical" research topics covered in Table 7.2, there are also research topics that are unique—or much more pronounced—for web systems evolution. Important topics that exemplify this point—content analysis, accessibility, and browser-safeness—are discussed in the following.

Originally, metrics-based evolution research has exclusively focused on the code and structure of a web system, but it was then realized that evolution can be also tracked by analyzing the (textual) content of a web system with appropriate metrics. An early example of such research is textual analysis of multilingual sites to find matching pairs of pages for different languages [852]. More recently, the evolution of Wikipedia in terms of number of edits and unique contributors has been analyzed [263], and another evolution study looked at legal statements of web sites, analyzing their length and readability [468]. The latter was measured with established readability metrics, namely SMOG and Flesch Reading Ease Score (FRES) [468]. Content analysis is an example of how web system evolution research has continuously broadened its focus.

An interesting example of a research area that has much broader and more prominent focus in web systems evolution than classical evolution research is *accessibility*, which is concerned with "how people with [varying degrees of] disabilities can perceive, understand, navigate, and interact with the web, and that they can contribute to the web" [410]. In fact, accessibility has been a constant throughout the history of web evolution research [467]. In 1999, Eichmann cautioned that for the development of many web systems "little attention is paid to issues of comprehension, navigation or accessibility" [271]. Cesarano et al. tackle the problem of usability of web pages for blind users [168]. They point out that web pages are designed for viewing on a two-dimensional screen while screen-reader tools for the blinds are reading the content in a linear, one-dimensional fashion. Consequently, the reading order should be redefined for blinds. Di Lucca et al. present refactoring heuristics for the automatic reordering of the items on a web page based on structural analysis and on summarization, with the purpose to reduce the "reaching time" (i.e. the time needed to reach the most relevant content of the web page) [245].

In contrast, Berry provides a detailed classification of characteristics of hearing impaired individuals and their respective accessibility issues [99]. These issues became more and more acute in the last years with the popularity of services that exploit the Internet as a medium to transmit voice and multimedia such as Skype and YouTube. Berry also points out that accessibility requirements of sight-impaired individuals can contradict the ones of hearing-impaired individuals. While accessibility research is often focused on seeing and hearing impaired, Boldyreff points out that "web accessibility encompasses a variety of concerns ranging from societal, political, and economic to individual, physical, and intellectual through to the purely technical. Thus, there are many perspectives from which web accessibility can be understood" [125].

An example of a research area that is unique to web site evolution is *browser-safeness*, meaning the requirement that a web system should look and behave the same across different web browsers. This problem is almost as old as the web and it is increasingly challenging to satisfy because of the large number of browsers and browser versions. While browsers try to adhere to (ambiguous) web standards, they are also trying to accommodate legacy web systems that are (partially) violating these standards. Also, JavaScript engines of different browsers differ in more or less subtle ways (e.g., as exposed by the test262 suite [182]).

The state-of-the-practice to address this problem are tools that take screenshots of the web system running in different browsers so that they can be inspected manually for differences. The WebDiff tool identifies cross-browser issues by taking pairs of screenshots of a web page in combination with analyzing the respective DOMs [183]. Variable elements in the DOM that are changing when pages are reloaded (e.g., advertisements) are filtered out, and then a structural comparison of the DOMs is performed. The DOM is also used to guide a visual analysis that does a graphical matching of DOM elements. The tool reports each mismatch so that it can be further inspected by the tool user. The CrossT tool offers a complementary approach that can be applied to a whole web application [600]. CrossT crawls the web system in the same manner using different browsers (i.e., Internet Explorer, Firefox and Chrome are supported), constructing for each a navigation model based on a finite state machine. A state corresponds to a "screen" as observed by the user, transitions are actions executed on the screen such as clicking a button. The constructed models are compared pairwise for discrepancies. Differences are detected in the navigation model as a set of traces that exist in one model but not in the other, and for two state pairs by comparing the underlying DOMs of the states' screens. When comparing DOMs the algorithm has to account for different DOM representations in the browsers.

7.5 Sources for Further Reading

Web systems evolution has been an active research area for over 15 years and consequently there are many resources available for studying. Thus, this chapter, by necessity, only represents a (biased) selection of this growing field, but constitutes a good starting point for the reader to explore further. In this vain, this section identifies the field's key research venues and journals as well as outstanding dissertations.

The annual WSE symposium has targeted this research area since 1999 and has featured the most influential research trends as they have emerged and changed through the years. In fact, many of the references in this chapter are WSE publications. Special issues for WSE 2002, 2006 and 2010 have been published with Wiley's JSEP [461].

Research on web systems evolution can also be found in venues for software maintenance (ICSM, CSMR), reverse engineering (WCRE), program comprehension (IWPC/ICPC), and software engineering (ICSE). Publications in the aforementioned venues are typically dealing with techniques and tools for web systems comprehension, analysis and migration. The communities on web, hypertext, multimedia and documentation/communication are also conducting research on web systems evolution, albeit at a context that is typically broader than what is covered in this chapter. Examples of interesting venues are the ACM Special Interest Group on the Design of Communication (SIGDOC) conference, the ACM Web Science Conference (WebSci), the International Conference on Web Information Systems Engineering (WISE), the International Conference on Web Engineering (ICWE), and the International World Wide Web Conference (WWW). In terms of dedicated journals, there are the ACM's Transaction on the Web (TWEB), Rinton Press' Journal of Web Engineering (JWE), Inderscience's International Journal of Web Engineering and Technology (IJWET), and Emerald's International Journal of Web Information Systems (IJWIS).

Last but not least, there is a growing number of Ph.D. theses that target web systems evolution, testifying that the research community has recognized the relevance of this subfield of software evolution. To name some outstanding dissertations: "Analysis, Testing, and Re-structuring of Web Applications" by Ricca [715], "Reengineering Legacy Applications and Web Transactions: An extended version of the UWA Transaction Design Model" by Distante [251], "Analysis and Testing of Ajax-based Single-page Web Applications" by Mesbah [599], and "Reverse Engineering and Testing of Rich Internet Applications" by Amalfitano [27].

7.6 Conclusions

This chapter described the key research topics and achievements in the domain of web systems evolution. It can be expected that web systems evolution research will continue to be of great relevance due to the fact that more and more software functionality is made available via web-based infrastructure. The impact of HTML5 is already felt in this respect and should further intensify the move towards open web systems. Web technologies are highly dynamic by nature, making them predisposed as a platform for building *highly-dynamic systems* (cf. Chapter 7.6), incorporating features such as customization/personalization, resource discovery, late binding, and run-time composition of services. Thus, future evolution research should provide techniques and tools for reasoning about dynamic properties.

Other evolution drivers that may have a strong influence on future research are the Internet of Things (IoT) and the Web 3.0. IoT would expand the web's reach significantly, incorporating devices of any scale. Web 3.0 would make large-scale semantic reasoning feasible. In a sense, the web will increasingly blend the ecosystems of cyberspace and biological space (cf. Chapter 10). Both IoT and Web 3.0 would open up many new research avenues, but also demand a much more inter/multi-disciplinary approach to research, which does not only address technical concerns, but also others, such as society, law/regulation, economics and environmental sustainability.

Chapter 8
Runtime Evolution of Highly Dynamic Software

Hausi Müller and Norha Villegas

Summary. Highly dynamic software systems are applications whose operations are particularly affected by changing requirements and uncertainty in their execution environments. Ideally such systems must evolve while they execute. To achieve this, highly dynamic software systems must be instrumented with self-adaptation mechanisms to monitor selected requirements and environment conditions to assess the need for evolution, plan desired changes, as well as validate and verify the resulting system. This chapter introduces fundamental concepts, methods, and techniques gleaned from self-adaptive systems engineering, as well as discusses their application to runtime evolution and their relationship with off-line software evolution theories. To illustrate the presented concepts, the chapter revisits a case study conducted as part of our research work, where self-adaptation techniques allow the engineering of a dynamic context monitoring infrastructure that is able to evolve at runtime. In other words, the monitoring infrastructure supports changes in monitoring requirements without requiring maintenance tasks performed manually by developers. The goal of this chapter is to introduce practitioners, researchers and students to the foundational elements of self-adaptive software, and their application to the continuos evolution of software systems at runtime.

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_8, © Springer-Verlag Berlin Heidelberg 2014

8.1 Introduction

Software evolution has been defined as the application of software maintenance actions with the goal of generating a new operational version of the system that guarantees its functionalities and qualities, as demanded by changes in requirements and environments [170, 598]. In the case of continuously running systems that are not only exposed frequently to varying situations that may require their evolution, but also cannot afford frequent interruptions in their operation (i.e., 24/7 systems), software maintenance tasks must be performed ideally while the system executes, thus leading to *runtime software evolution* [598]. Furthermore, when changes in requirements and environments cannot be fully anticipated at design time, maintenance tasks vary depending on conditions that may be determined only while the system is running.

This chapter presents runtime evolution from the perspective of *highly dynamic software systems*, which have been defined as systems whose operation and evolution are especially affected by uncertainty [646]. That is, their requirements and execution environments may change rapidly and unpredictably. Highly dynamic software systems are context-dependent, feedback-based, software intensive, decentralized, and quality-driven. Therefore, they must be continually evolving to guarantee their reliable operation, even when changes in their requirements and context situations are frequent and in many cases unforeseeable. Müller et al. have analyzed the complexity of evolving highly dynamic software systems and argued for the application of evolution techniques at runtime [621]. They base their arguments on a set of problem attributes that characterize feedback-based systems [622]. These attributes include (i) the uncertain and non-deterministic nature of the environment that affects the system, and (ii) the changing nature of requirements and the need for regulating their satisfaction through continuous evolution, rather than traditional software engineering techniques.

Control theory, and in particular feedback control, provides powerful mechanisms for uncertainty management in engineering systems [625]. Furthermore, a way of exploiting control theory to deal with uncertainty in software systems is through self-adaptation techniques. Systems enabled with self-adaptive capabilities continuously sense their environment, analyze the need for changing the way they operate, as well as plan, execute and verify adaptation strategies fully or semi-automatically [177, 221]. On the one hand, the goal of software evolution activities is to extend the life span of software systems by modifying them as demanded by changing real-world situations [595]. On the other hand, control-based mechanisms, enabled through self-adaptation, provide the means to implement these modifications dynamically and reliably while the system executes.

Rather than presenting a comprehensive survey on runtime software evolution and self-adaptive systems, this chapter introduces the notion of runtime software evolution, and discusses how foundational elements gleaned from self-adaptation are applicable to the engineering of runtime evolution capabilities. For this we organized the contents of this chapter as follows. Section 8.2 describes a case study, based on dynamic context monitoring, that is used throughout the chapter to explain the presented concepts. Section 8.3 revisits traditional software evolution to introduce the need for applying runtime software evolution, and discusses selected aspects that may be taken into account when deciding how to evolve software systems dynamically. Section 8.4 characterizes dimensions of runtime software evolution, and discusses Lehman's laws in the context of runtime evolution. Section 8.5 introduces the application of feedback, feedforward, and adaptive control to runtime software evolution. Sections 8.6 and 8.7 focus on foundations and enablers of self-adaptive software that apply to the engineering of runtime software evolution, and Section 8.8 illustrates the application of these foundations and enablers in the case study introduced in Section 8.2. Section 8.9 discusses selected self-adaptation challenges that deserve special attention. Finally, Section 8.10 summarizes and concludes the chapter.

8.2 A Case Study: Dynamic Context Monitoring

As part of their collaborative research on self-adaptive and context-aware software applications, researchers at University of Victoria (Canada) and Icesi University (Colombia) developed SMARTERCONTEXT [822, 895–898, 900]. SMARTERCONTEXT is a service-oriented context monitoring infrastructure that exploits self-adaptation techniques to evolve at runtime with the goal of guaranteeing the relevance of monitoring strategies with respect to changes in monitoring requirements [894].

In the case study described in this section, the SMARTERCONTEXT solution evolves at runtime with the goal of monitoring the satisfaction of changing quality of service (QoS) contracts in an e-commerce scenario [822]. Changes in contracted conditions correspond to either the addition/deletion of quality attributes to the contracted service level agreement (SLA), or the modification of desired conditions and corresponding thresholds. Suppose that an online retailer and a cloud infrastructure provider negotiate a performance SLA that specifies *throughput*, defined as the time spent to process a purchase order request ($ms/request$), as its quality factor. Suppose also that a first version of SMARTERCONTEXT was developed to monitor the variable relevant to the throughput quality factor (i.e., processing time per request). Imagine now that later the parties renegotiate the performance SLA by adding *capacity*, defined in terms of bandwidth, as a new quality factor. Since SMARTERCONTEXT has been instrumented initially to monitor processing time only, it will have to evolve at runtime to monitor the system's bandwidth. Without these runtime evolution capabilities, the operation of the system will be compromised until the new monitoring components are manually developed and deployed.

SMARTERCONTEXT relies on behavioral and structural self-adaptation techniques to realize runtime evolution. Behavioural adaptation comprises mechanisms that tailor the functionality of the system by modifying its parameters or business logic, whereas structural adaptation uses techniques that modify the system's software architecture [899]. SMARTERCONTEXT implements behavioral adaptation by modifying existing monitoring conditions or adding new context types and reasoning rules at runtime, and structural adaptation by (un)deploying context gatherers and context processing components. All these operations are realized without requiring the manual development or deployment of software artifacts, and minimizing human intervention.

8.3 Assessing the Need for Runtime Evolution

The need for evolving software systems originates from the changing nature of system requirements and the changing environment that can influence their accomplishment by the system [123]. Indeed, as widely discussed in Chapter 1, changing requirements are inherent in software engineering. For example, in the case of SMARTERCONTEXT the need for generating new versions of the system arises from the renegotiation of contracted SLAs, which implies changes in monitoring requirements. Several models have been proposed to characterize the evolution process of software systems [590]. In particular, the *change mini-cycle* model proposed by Yau et al. defines a feedback-loop-based software evolution process comprising the following general activities: change request, analysis, planning, implementation, validation and verification, and re-documentation [936]. Figure 8.1 depicts the flow among the general activities of the change mini-cycle process model. These activities were proposed for off-line software evolution, which implies the interruption of the system's operation.

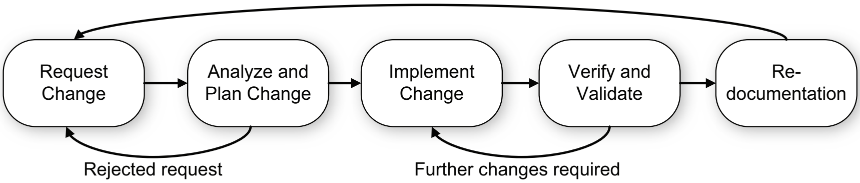


Fig. 8.1: Activities of the *change mini-cycle* process model for software evolution [936] (adapted from [590]). This model implements a feedback loop mechanism with activities that are performed off-line.

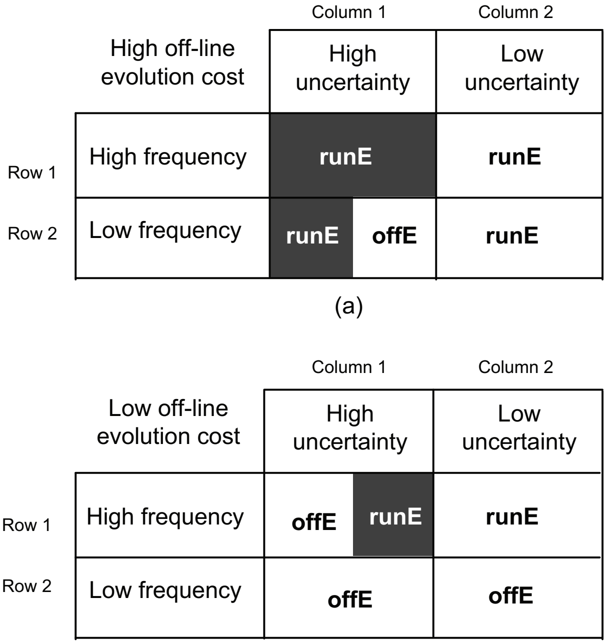
We define *off-line software evolution* as the process of modifying a software system through actions that require intensive human intervention and imply the interruption of the system operation. We define the term *runtime software evolution* as the process of modifying a software system through tasks that require minimum human intervention and are performed while the system executes. This section characterizes software evolution from a runtime perspective.

The need for evolving software systems emerges from changes in environments and requirements that, unless addressed, compromise the operation of the system. The need for evolving software systems *at runtime* arises from the frequency and uncertainty of these changes, as well as the cost of implementing off-line evolution. In the context of software evolution, we define *frequency* as the number of occurrences of a change event per unit of time that will require the evolution of the system. For example, the number of times an SLA monitored by SMARTERCONTEXT is modified within a year. We define *uncertainty* as the reliability with which it is possible to characterize the occurrence of changes in requirements and execution environments. The level of uncertainty in a software evolution process depends on the deterministic nature of these changes. That is, the feasibility of anticipating their frequency, date, time, and effects on the system. In the case of SMARTERCONTEXT, changes in SLAs will be less uncertain to the extent that it is possible to anticipate the date and time contract renegotiation will occur, as well as the aspects of the SLA that will be modified including quality factors, metrics and desired thresholds. The cost of implementing off-line evolution can be quantified in terms of metrics such as system's size, time to perform changes, personnel, engineering effort, and risk [123, 509, 510]. Further information about the cost of evolving SMARTERCONTEXT at runtime are available in the evaluation section presented in [822].

As introduced in Section 8.2 engineering techniques applicable to self-adaptive software systems [177] can be used to evolve software systems dynamically. Nevertheless, runtime evolution is not always the best solution given the complexity added by the automation of evolution tasks. As an alternative to decide whether or not to implement runtime software evolution we envision the analysis of its benefit-cost ratio (BCR). We define the BCR of runtime software evolution as a function of the three variables mentioned above: frequency, uncertainty, and off-line evolution cost.

Figure 8.2 characterizes the application of runtime versus off-line software evolution according to the variables that affect the BCR function. Figure 8.2(a) concerns scenarios where the cost of off-line software evolution is high, whereas Figure 8.2(b) scenarios where this cost is low. In both tables rows represent the frequency of changes in requirements and environments, columns the level of uncertainty of these changes, and cells whether the recommendation is to apply runtime or off-line software evolution. Dark backgrounds indicate high levels of complexity added by the application of runtime evolution. We refer to each cell as $cell_{i,j}$ where i and j are the row and column indices that identify the cell.

According to Figure 8.2, runtime evolution is the preferred alternative when the cost of evolving the system off-line is high, as for example in the application of SMARTERCONTEXT to the monitoring of SLAs that may be renegotiated while the e-commerce platform is in production. When off-line evolution is affordable and the frequency of changes is high, both alternatives apply. Nevertheless, it is important to analyze the value added by runtime evolution versus its complexity (cf. $cell_{1,1}$ in Figure 8.2(b)). The complexity added by runtime evolution lies in the automation of activities such as the ones defined in the change mini-cycle process (cf. Figure 8.1). That is, the system must be instrumented with monitors to identify the need for evolution, analyzers and planners to correlate situations and define evolution actions dynamically, executors to modify the system, runtime validation and verification tasks to assure the operation of the system after evolution, and runtime modelling mechanisms to preserve the coherence between the running system and its design



		Column 1	Column 2
	High off-line evolution cost	High uncertainty	Low uncertainty
Row 1	High frequency	runE	runE
Row 2	Low frequency	runE offE	runE

(a)

		Column 1	Column 2
	Low off-line evolution cost	High uncertainty	Low uncertainty
Row 1	High frequency	offE runE	runE
Row 2	Low frequency	offE	offE

Fig. 8.2: A characterization of runtime versus off-line software evolution in light of frequency of changes in requirements and environments, uncertainty of these changes, and cost of off-line evolution. Dark backgrounds indicate high levels of complexity added by the application of runtime evolution.

artifacts. The complexity of the system is augmented since the functionalities that realize these tasks at runtime not only require qualified engineers for their implementation, but also become system artifacts that must be managed and evolved.

Under high levels of uncertainty runtime evolution is an alternative to be considered despite its complexity. This is because the variables that characterize the execution environment of the running system are unbound at design or development time, but bound at runtime. For example, in many cases it is infeasible to anticipate at design time or development time changes to be implemented in SMARTERCONTEXT. On the contrary, the runtime evolution scenarios that expose the lowest complexity are those with low uncertainty (cf. column 2 in both matrices). This is because the system operates in a less open environment where the evolution process can be characterized better at design and development time. As a result, the functionalities used to evolve the system at runtime are less affected by unforeseeable context situations. For example, it is possible to manually program runtime evolution functionalities to guarantee more demanding throughput levels during well known shopping seasons such as *Black Friday*.[1] When the off-line evolution cost is high and the uncertainty

[1] On Black Friday, the day after Thanksgiving Day in USA, sellers offer unbeatable deals to kick off the shopping season thus making it one of the most profitable days. Black Friday is

is low, runtime evolution seems to be the best alternative (cf. $cell_{1,2}$ and $cell_{2,2}$ in Figure 8.2(a)). Complexity is directly proportional to uncertainty. Therefore, when uncertainty and the cost of off-line evolution are low, runtime evolution is still a good option if the frequency of changes that require the evolution of the system are extremely high (cf. $cell_{1,2}$ in Figure 8.2(b)). In contrast, under low change frequencies, high levels of uncertainty and low off-line evolution costs, off-line evolution constitutes the best alternative (cf. $cell_{2,1}$ in Figure 8.2(b)). Moreover, off-line evolution is the best option when evolution cost, change frequencies and uncertainty are low (cf. $cell_{2,2}$ in Figure 8.2(b)).

Recalling the evolution scenario described in Section 8.2, it is possible to illustrate the application of the characterization of the BCR variables presented in Figure 8.2 to decide whether to implement runtime or off-line software evolution for adding new functionalities to the SMARTERCONTEXT monitoring infrastructure. Regarding the cost of off-line software evolution, the implementation of manual maintenance tasks on the monitoring infrastructure of the e-commerce platform is clearly an expensive and complex alternative. First, the manual deployment and undeployment of components is expensive in terms of personnel. Second, the evolution time can be higher than the accepted levels, therefore the risk of violating contracts and losing customers increases. In particular, this is because renegotiations in SLAs cannot always be anticipated and must be addressed quickly. Uncertainty can also be high due to the variability of requirements among the retailers that use the e-commerce platform, thus increasing the frequency of changes in requirements. For example, new retailers may subscribe to the e-commerce platform with quality of service requirements for which the monitoring instrumentation is unsupported. Therefore, the system must evolve rapidly to satisfy the requisites of new customers. These high levels of uncertainty and frequency of changes in requirements require runtime support for monitoring, system adaptation, and assurance.

8.4 Dimensions of Runtime Software Evolution

Software evolution has been analyzed from several dimensions comprising, among others, the *what*, *why* and *how* dimensions of the evolution process [590]. The *what* and *why* perspectives focus on the software artifacts to be evolved (e.g., the software architecture of SMARTERCONTEXT) and the reason for evolution (e.g., a new monitoring requirement), respectively. The *how* view focuses on the means to implement and control software evolution (e.g., using structural self-adaptation to augment the functionality of the system by deploying new software components at runtime). Figure 8.3 summarizes these perspectives as dimensions of *runtime* software evolution. With respect to the *why* perspective, the emphasis is on changing requirements (cf. Chapter 1), malfunctions, and changing execution environments as causes of run-

the day in which retailers earn enough profit to position them “in the black” – an accounting expression that refers to the practice of registering profits in black and losses in red. Source: http://www.investopedia.com.

time evolution. Regarding the *what*, the focus is on system goals, system structure and behavior, and design specifications as artifacts susceptible to runtime evolution. For example models and metamodels as studied in Chapter 2. Concerning the *how*, the attention is on methods, frameworks, technologies, and techniques gleaned from control theory and the engineering of self-adaptive software (SAS) systems. The elements highlighted in gray correspond the answers to the why, what and how questions for the case study presented in Section 8.2.

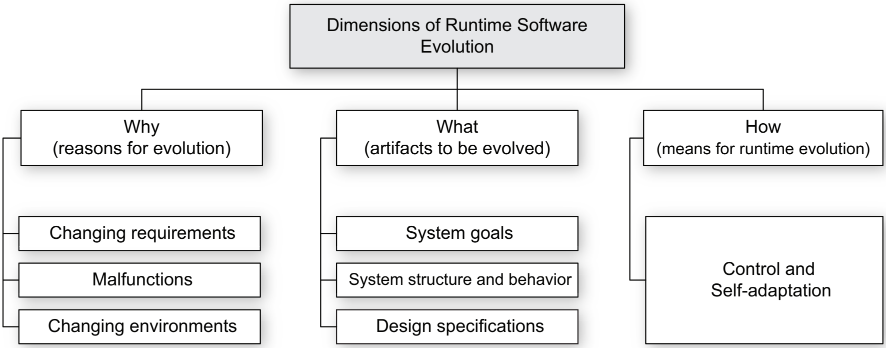

Dimensions of Runtime Software Evolution
Why
(reasons for evolution)
What
(artifacts to be evolved)
How
(means for runtime evolution)
Changing requirements
Malfunctions
Changing environments
System goals
System structure and behavior
Design specifications
Control and Self-adaptation

Fig. 8.3: Characterizing dimensions of runtime software evolution. The highlighted elements relate to the SMARTERCONTEXT case study.

Software evolution has been characterized through the *laws of software evolution* proposed by Lehman [509, 553]. Lehman's laws apply to *E-type* systems, the term he used to refer to software systems that operate in real world domains that are potentially unbounded and susceptible to continuing changes [507]. Therefore, it is clear that when Lehman proposed his laws of software evolution back in the seventies, he focused on systems that operate in real world situations and therefore are affected by continuing changes in their operation environments [509, 553]. Hence, Lehman's laws corroborate the need for preserving the qualities of software systems in changing environments, which may require the implementation of runtime evolution capabilities depending on the cost of off-line evolution, the uncertainty of the environment and the frequency of changes. Lehman's laws of software evolution can be summarized as follows:

1. *Continuing change.* E-type systems must adapt continuously to satisfy their requirements.
2. *Increasing complexity.* The complexity of E-type systems increases as a result of their evolution.
3. *Self-regulation.* E-type system evolution processes are self-regulating.
4. *Conservation of organizational stability.* Unless feedback mechanisms are appropriately adjusted in the evolution process, the average effective rate in an evolving E-type system tends to remain constant over the product lifetime.

5. *Conservation of familiarity.* The incremental growth and long term growth rate of E-type systems tend to decline.
6. *Continuing growth.* The functional capability of E-type systems must be continually increased to maintain user satisfaction over the system lifetime.
7. *Declining quality.* Unless adapted according to changes in the operational environment, the quality of E-type systems decline.
8. *Feedback system.* E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems [553].

Considering the *continuing change* law, highly dynamic software systems such as SMARTERCONTEXT need not only adapt continuously to satisfy their requirements, but in many cases do it at runtime. One of the key benefits of evolving a system at runtime is to be able to verify assumptions made at design time. Although valid when the system was designed, some of these assumptions become invalid over time. For example, the QoS requirements of the e-commerce company in the application scenario described in Section 8.2 may vary over time. Moreover, new monitoring requirements must be addressed to satisfy changes in SLAs. To counteract the effects of this first law, traditional software evolution focuses on the off-line activities defined in the change mini-cycle process (cf. Figure 8.1). On the contrary, runtime evolution argues for the instrumentation of software systems with feedback-control capabilities that allow them to manage uncertainty and adapt at runtime, by performing these tasks while the system executes. Of course, due to the *increasing complexity* law, the trade-off between runtime and off-line evolution affects the level of automation and the instrumentation required to evolve the system without risking its operation. The complexity of highly dynamic software systems includes aspects such as the monitoring of the execution environment, the specification and management of changing system requirements, the implementation of dynamic mechanisms to adjust the software architecture and business logic, the preservation of the coherence and causal connection among requirements, specifications and implementations, and the validation of changes at runtime.

The *self-regulation* law stated by Lehman characterizes software evolution as a self-regulating process. In highly dynamic software systems self-regulation capabilities are important across the software life cycle, in particular at runtime. One implication, among others, is the capability of the system to decide when to perform maintenance and evolution tasks and to assure the evolution process. Self-regulating software evolution can be realized by enabling software systems with feedback control mechanisms [147, 212, 553, 622].

Laws *continuing growth* and *declining quality* are undeniably connected with the capabilities of a software system to evolve at runtime. Continuing growth argues for the need of continually increasing the functionalities of the system to maintain user satisfaction over time. Similarly, declining quality refers to the need for continually adapting the system to maintain the desired quality properties.

Regarding declining quality, runtime software evolution can effectively deal with quality attributes whose preservation depends on context situations [820, 899].

8.5 Control in Runtime Software Evolution

Control is an enabling technology for software evolution. At runtime, control mechanisms can be realized using self-adaptation techniques [147]. *Feedback control* concerns the management of the behavior of dynamic systems. In particular, it can be applied to automate the control of computing and software systems [147, 388]. From the perspective of software evolution, feedback control can be defined as the use of algorithms and feedback for implementing maintenance and evolution tasks [553, 625].

8.5.1 Feedback Control

Feedback control is a powerful tool for uncertainty management. As a result, self-evolving software systems based on feedback loops are better prepared to deal with uncertain evolution scenarios. This is the reason why the BCR of runtime evolution is higher under uncertain environments (cf. Section 8.3). Uncertainty management using feedback control is realized by monitoring the operation and environment of the system, comparing the observed variables against reference values, and adjusting the system behavior accordingly. The goal is to adapt the system to counteract disturbances that can affect the accomplishment of its goals.

8.5.2 Feedforward Control

Feedforward control, which operates in an open loop, can also benefit the evolution of software systems. The fundamental difference between feedback and feedforward control is that in the first one control actions are based on the deviation of measured outputs with respect to reference inputs, while in the latter control actions are based on plans that are fed into the system. These plans correspond to actions that are not associated with uncertain changes in the execution environment. Another application of feedforward control is the empowerment of business and system administrators to modify system goals (e.g., non-functional requirements) and policies that drive the functionalities of feedback controllers. Feedforward control could be exploited using policies and pre-defined plans as the mechanism to control runtime software evolution tasks. Therefore, feedforward control provides the means to manage short-term evolution according to the long-term goals defined in the general evolution process.

The application of feedback and feedforward control to runtime software evolution can be analyzed in light of the dimensions of software evolution depicted in Figure 8.3. With respect to the *why* dimension, feedback control applies to runtime evolution motivated by malfunctions or changing environments, and feedforward to runtime evolution originated in changing requirements. The reasons for this distinction are that in the first case the symptoms that indicate the need for evolution result from the monitoring of the system and its environment and the analysis of the observed conditions. In the second case, the evolution stimulus comes from the direct action of an external entity (e.g., a system administrator). For example, in the case of SMARTERCONTEXT feedback control allows the monitoring infrastructure to replace failing sensors automatically, whereas feedforward enables SMARTERCONTEXT to deploy new monitoring artifacts or modify the monitoring logic to address changes in SLAs. With respect to the *what* dimension, pure control-based evolution techniques better apply to the modification of the system behavior or its computing infrastructure (this does not involve the software architecture). The reason is that pure control-based adaptation actions are usually continuous or discrete signals calculated through a mathematical model defined in the controller [899].

The feedback loop, a foundational mechanism in control theory, is a reference model in the engineering of SAS systems [622, 772]. Therefore, the implementation of runtime evolution mechanisms based on feedback control and self-adaptation requires the understanding of the feedback loop, its components, and the relationships among them. Figure 8.4 depicts the feedback loop model from control theory. To explain its components and their relationships, we will extend our runtime software evolution case study described in Section 8.2. These extensions are based on examples written by Hellerstein et al. [388].

Imagine that the service-oriented e-commerce system that is monitored by the SMARTERCONTEXT monitoring infrastructure provides the online retailing platform for several companies worldwide. Important concerns in the operation of this system are its continuous availability and its operational costs. The company that provides this e-commerce platform is interested in maximizing system efficiency and guaranteeing application availability, according to the needs of its customers. Concerning efficiency, the company defines a performance quality requirement for the service that processes purchase orders. The goal is to maximize the number of purchase orders processed per time unit. Regarding availability, the business implements a strategy based on redundant servers. The objective is to guarantee that the operation of the e-commerce platform upon eventual failures in its infrastructure. After the system has been in production for a certain time period, the company realizes that the system capacity to process purchase orders is insufficient to satisfy the demand generated by *Black Friday* and by different special offers placed at strategic points during the year. To solve this problem, the company may decide to extend its physical infrastructure capacity to guarantee availability under peak load levels. Nevertheless, this decision will affect efficiency and costs since an important part of the resources will remain unused for long periods of time, when the system load is at its normal levels. To improve the use of resources, the company may decide to perform periodic maintenance tasks manually to increase or decrease system capacity according to the demand. However, this strategy not only increases the costs and complexity of maintaining and evolving the system, but is also ineffective since changes in the demand cannot always be anticipated and may arise quickly (e.g., in intervals of minutes when a special discount becomes popular in a social network) and frequently. Similarly, the strategy based on redundant servers to guarantee availability may challenge the maintenance and evolution of the system when performed manually. First, the use of redundant servers to address failures in the infrastructure must not compromise the contracted capacity of the system. Thus, servers must be replaced only with machines of similar specifications. Second, this strategy must take into account the capacity levels contracted with each retailer, which may vary not only over time, but also from one retailer to another.

The target system represents the system to be evolved dynamically using self-adaptation (e.g., our e-commerce platform). System requirements correspond to *reference inputs* (label (A) in Figure 8.4). Suppose that for the e-commerce company to guarantee the availability of its platform, it implements the redundant server strategy by controlling the CPU utilization of each machine. For this, it must maintain servers working below their maximum capacity in such a way that when a server fails, its load can be satisfied by the others. The reference input corresponds to the desired CPU utilization of the servers being controlled. The system is monitored continuously by comparing the actual CPU usage, the *measured output* (label (B)), against the reference input. The difference between the measured output and the reference input is the *control error* (label (C)). The controller implements a mathematical function (i.e., transfer function) that calculates the *control input* (label (D)) based on the error. Control inputs are the stimuli used to affect the behavior of the target system. To control the desired CPU usage, the control input is defined as the maximum number of connections that each controlled server must satisfy. The measured output can also be affected by external *disturbances* (label (E)), or even by the *noise* (label (F)) caused by the system evolution. *Transducers* (label (G)) translate the signals coming from sensors, as required by the comparison element (label (H), for example to unify units of measurement). In this scenario the *why* dimension of runtime software evolution corresponds to malfunctions, the *what* to the processing infrastructure, and the *how* to self-adaptation based on feedback control.

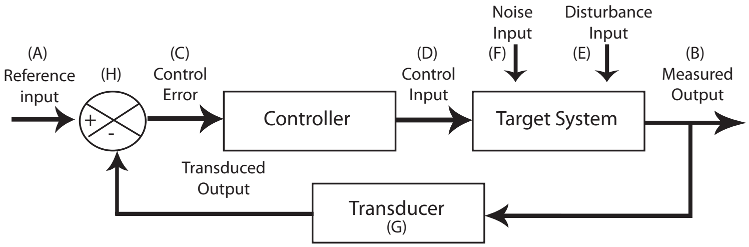


Fig. 8.4: Classical block diagram of a feedback control system [388]. Short-term software evolution can be realized through feedback loops that control the behavior of the system at runtime.

To Probe Further

Readers interested in studying the application of feedback loops to the engineering of self-adaptation mechanisms for software systems may refer to [147, 388, 622, 772, 900].

8.5.3 Adaptive Control

From the perspective of control theory, *adaptive control* concerns the automatic adjustment of control mechanisms. Adaptive control researchers investigate parameter adjustment algorithms that allow the adaptation of the control mechanisms while guaranteeing global stability and convergence [268]. Control theory offers several reference models for realizing adaptive control. We focus our attention on two of them, *model reference adaptive control* (MRAC) and *model identification adaptive control* (MIAC).

8.5.3.1 Model Reference Adaptive Control (MRAC)

MRAC, also known as *model reference adaptive system* (MRAS), is used to implement controllers that support the modification of parameters online to adjust the way the target system is affected (cf. Figure 8.5). A reference model, specified in advance, defines the way the controller's parameters affect the target system to obtain the desired output. Parameters are adjusted by the adaptation algorithm based on the control error, which is the difference of the measured output and the expected result according to the model.

In runtime software evolution, MRAC could applied to the modification of the evolution mechanism at runtime. Since the controller uses the parameters received from the adaptation algorithm to evolve the target system, the control actions implemented by the controller could be adjusted dynamically by modifying the reference model used by the adaptation algorithm. The application of MRAC clearly improves the dynamic nature of the evolution mechanism, which is important for scenarios where the *why* dimension focuses on changes in requirements. Nevertheless, MRAC has a limited application in scenarios with high levels of uncertainty because it is impractical to predict changes in the reference model.

8.5.3.2 Model Identification Adaptive Control (MIAC)

In MIAC, the reference model that allows parameter estimation is identified or inferred at runtime using system identification methods. As depicted in Figure 8.6, the

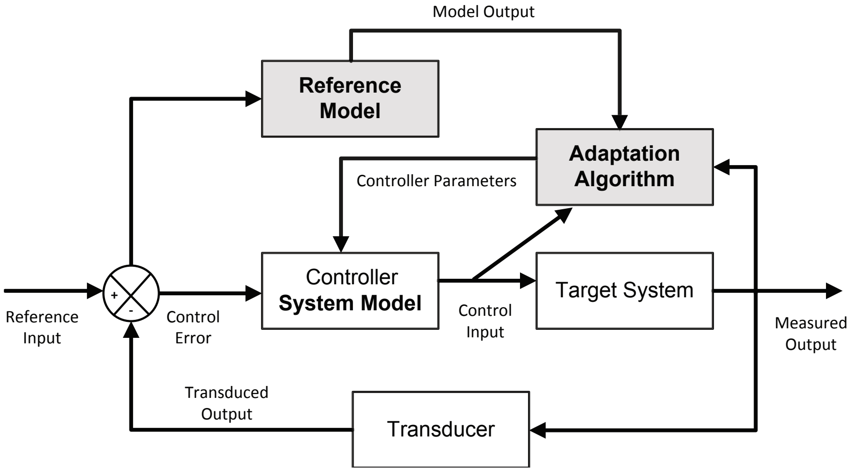


Fig. 8.5: Model Reference Adaptive Control (MRAC)

control input and measured output are used to identify the reference model (system identification). Then, the new model parameters are calculated and sent to the adjustment mechanism which calculate the parameters that will modify the controller.

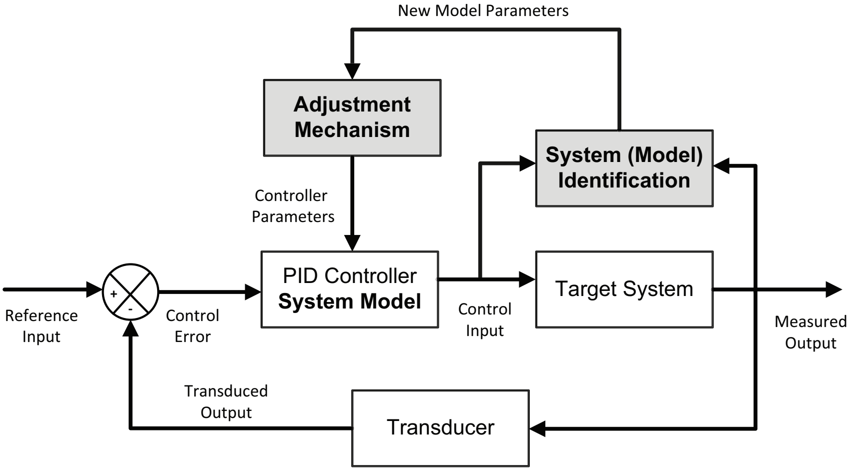


Fig. 8.6: Model Identification Adaptive Control (MIAC)

In the context of runtime software evolution, MIAC could support the detection of situations in which the current evolution strategy is no longer effective. Moreover, it could be possible to exploit MIAC to adjust the evolution mechanism fully or semi automatically. Since the reference model used to realize the controller's actions is synthesized from the system, MIAC seems more suitable for highly uncertain scenarios where the *why* dimension of software evolution corresponds to changes in the environment.

8.6 Self-Adaptive Software Systems

Another dimension that has been used to characterize software evolution refers to the *types of changes* to be performed in the evolution process [590], for which several classifications have been proposed [170, 813]. In particular, the ISO/IEC standard for software maintenance proposes the following familiar classification: *adaptive maintenance*, defined as the modification of a software product after its delivery to keep it usable under changing environmental conditions; *corrective maintenance*, as the reactive modification of the system to correct faults; and *perfective maintenance*, as the modification of the software to improve its quality attributes [424]. These maintenance types can be implemented using different self-adaptation approaches [739]. For example, adaptive maintenance can be realized through context-aware self-reconfiguration, corrective maintenance through self-healing, and perfective maintenance through self-optimization. Self-adaptive software (SAS) systems are software applications designed to adjust themselves, at runtime, with the goal of satisfying requirements that either change while the system executes or depend on changing environmental conditions. For this, SAS systems are usually instrumented with a feedback mechanism that monitors changes in their environment—including their own health and their requirements—to assess the need for adaptation [177, 221]. In addition to the monitoring component, the feedback mechanism includes components to analyze the problem, decide on a plan to remedy the problem, effect the change, as well as validate and verify the new system state. This feedback mechanism, also called adaptation process, is similar to the continuous feedback process of software evolution characterized by the change mini-cycle model (cf. Figure 8.1).

As analyzed in Section 8.3, under highly changing requirements and/or execution environments, it is desirable to perform evolution activities while the system executes. In particular when the off-line evolution is expensive. By instrumenting software systems with control capabilities supported by self-adaptation, it is possible to manage short-term evolution effectively. Indeed, the activities performed in the adaptation process can be mapped to the general activities of software evolution depicted by the change mini-cycle model. Therefore, self-adaptation can be seen as a short-term evolution process that is realized at runtime. SAS system techniques can greatly benefit the evolution of highly dynamic software systems such in the case of the SMARTERCONTEXT monitoring infrastructure of our case study. The monitoring requirements addressed by SMARTERCONTEXT are highly changing since they depend on contracted conditions that may be modified after the e-commerce system is in production. Therefore, to preserve the relevance of monitoring functionalities with respect to the conditions specified in SLAs, SMARTERCONTEXT relies on self-adaptation to address new functional requirements by evolving the monitoring infrastructure at runtime. This section introduces the engineering foundations of SAS systems and illustrates their application to runtime software evolution.

To Probe Further

The survey article by Salehie and Tahvildari presents an excellent introduction to the state of the art of SAS systems [739]. Their survey presents a taxonomy, based on *how*, *what*, *when* and *where* to adapt software systems, as well as an overview of application areas and selected research projects. For research roadmaps on SAS systems please refer to [177, 221].

8.6.1 Self-Managing Systems

Self-managing systems are systems instrumented with self-adaptive capabilities to manage (e.g., maintain or evolve) themselves given high-level policies from administrators. and with minimal human intervention [457]. *Autonomic computing*, an IBM initiative, aims at implementing self-managing systems able to anticipate changes in their requirements and environment, and accommodate themselves accordingly, to address system goals defined by policies [457]. The ultimate goal of autonomic computing is to improve the efficiency of system operation, maintenance, and evolution by instrumenting systems with autonomic elements that enable them with self-management capabilities. Systems with self-management capabilities expose at least one of the four self-management properties targeted by autonomic computing: *self-configuration, self-optimization, self-healing,* and *self-protection.* This subsection presents concepts from autonomic computing and self-managing systems that are relevant to the evolution of highly dynamic software systems.

8.6.2 The Autonomic Manager

The *autonomic manager*, illustrated in Figure 8.7, is the fundamental building block of self-managing systems in autonomic computing. The autonomic manager can be used to realize runtime evolution. For this, it implements an intelligent control loop (cf. Figure 8.4) that is known as the *MAPE-K* loop because of the name of its elements: the *monitor*, *analyzer*, *planner*, *executor*, and *knowledge base*. *Monitors* collect, aggregate and filter information from the environment and the target system (i.e., the system to be evolved), and send this information in the form of symptoms to the next element in the loop. *Analyzers* correlate the symptoms received from monitors to decide about the need for adapting the system. Based on business policies, *planners* define the maintenance activities to be executed to adapt or evolve the system. *Executors* implement the set of activities defined by planners. The knowledge base enables the information flow along the loop, and provides persistence for historical information and policies required to correlate complex situations.

The autonomic manager implements *sensors* and *effectors* as manageability endpoints that expose the state and control operations of managed elements in the system (e.g., the service that processes purchase orders and the managed servers in our e-commerce scenario). Sensors allow autonomic managers to gather information from both the environment and other autonomic managers. Effectors have a twofold function. First they provide the means to feed autonomic managers with business policies that drive the adaptation and evolution of the system. Second they provide the interfaces to implement the control actions that evolve the managed element. Managed elements can be either system components or other autonomic managers.

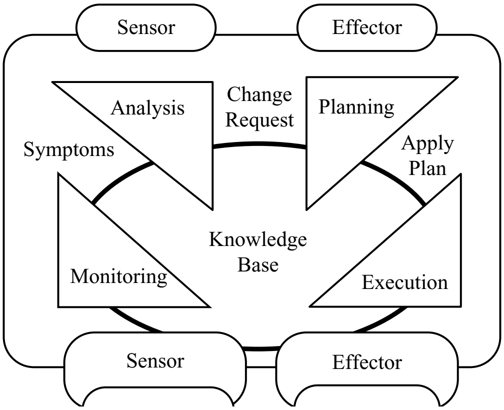


Fig. 8.7: The autonomic manager [457]. Each autonomic manager implements a feedback loop to *monitor* environmental situations that may trigger the adaptation of the system, *analyze* the need for adapting, *plan* the adaptation activities, and *execute* the adaptation plan.

To Probe Further

One of the most important articles on the notion of an autonomic manager and its applications is the 2006 IBM Technical Report entitled “An Architectural Blueprint for Autonomic Computing (Fourth Edition)” [417]. In another article, Dobson et al. argue for the integration of autonomic computing and communications and thus surveyed the state-of-the-art in autonomic communications from different perspectives [258].

Figure 8.8 depicts the mapping (cf. dashed arrows) between phases of the software evolution process defined by the change mini-cycle model (cf. white rounded boxes in the upper part of the figure) and the phases of the MAPE-K loop implemented by the autonomic manager (cf. gray rounded boxes at the bottom of the figure). On the one hand, the change mini-cycle model characterizes the activities of the "traditional" long-term software evolution process which is performed off-line [936]. On the other hand, the phases of the MAPE-K loop are realized at runtime. Therefore, autonomic managers can be used to realize short-term software evolution at runtime. The last two activities of the change mini-cycle model have no mapping to the MAPE-K loop process. However, this does not mean that these activities are not addressed in the implementation of self-adaptation mechanisms for software systems. Validation and verification (V&V) refer to the implementation of assurance mechanisms that allow the certification of the system after its evolution. Re-documentation concerns the preservation of the relevance of design artifacts with respect to the new state of the evolved system. To realize runtime evolution through self-management capabilities of software systems these activities must also be performed at runtime. Indeed, these are challenging topics subject of several research initiatives in the area of software engineering for self-adaptive and self-managing software systems. In particular, assurance concerns can be addressed through the implementation of V&V tasks along the adaptation feedback loop [823]. The preservation of the relevance between design artifacts and the evolving system can be addressed through the implementation of runtime models [92]. Runtime V&V and runtime models are foundational elements of SAS systems also addressed in this chapter.

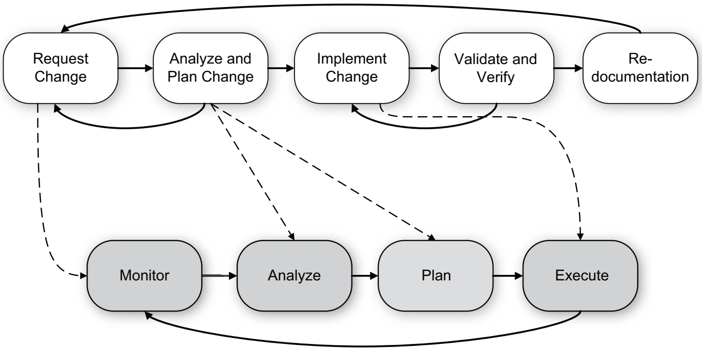


Fig. 8.8: Mapping the phases of the software evolution change mini-cycle (cf. Figure 8.1) to the phases of the MAPE-K loop implemented by the autonomic manager (cf. Figure 8.7)

8.6.3 The Autonomic Computing Reference Architecture

The autonomic computing reference architecture (ACRA), depicted in Figure 8.9, provides a reference architecture as a guide to organize and orchestrate self-evolving (i.e., autonomic) systems using autonomic managers. Autonomic systems based on ACRA are defined as a set of hierarchically structured building blocks composed of orchestrating managers, resource managers, and managed resources. Using ACRA, software evolution policies can be implemented as resource management policies into layers where system administrator (manual manager) policies control lower level policies. System operators have access to all ACRA levels.

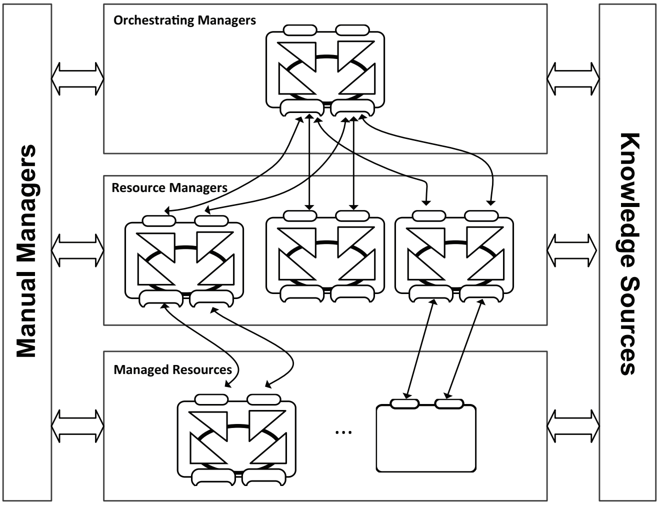


Fig. 8.9: The Autonomic Computing Reference Architecture (ACRA) [417]

To Probe Further

Over the past thirty years, researchers from different fields have proposed many three-layer models for dynamic systems. In particular, researchers from control engineering, AI, robotics, software engineering, and service-oriented systems have devised—to a certain extent independently—reference architectures for designing and implementing control systems, mobile robots, autonomic systems, and self-adaptive systems. Seminal three-layer reference architectures for self-management are the hierarchical intelligent control system (HICS) [745], the concept of adaptive control architectures [48], Brooks' layers of competence [145], Gats Atlantis architecture [317], IBM's ACRA—autonomic computing reference architecture [417, 457], and Kramer and Magee's self-management architecture [478, 479]. Oreizy et al. introduced the concept of *dynamic architecture evolution* with their the *Figure 8* model separating the concerns of adaptation management and evolution management [668–670]. Dynamico is the most recent three-layer reference model for context-driven self-adaptive systems [900]. The key idea in these layered architectures is to build levels of control or competence. The lowest level controls individual resources (e.g., manage a disk). The middle layer aims to achieve particular goals working on individual goals concurrently (e.g., self-optimizing or self-healing). The highest level orchestrates competing or conflicting goals and aims to achieve overall system goals (e.g., minimizing costs).

ACRA is useful as a reference model to design and implement runtime evolution solutions based on autonomic managers. For example, as depicted in Figure 8.10, the ACRA model can be used to derive architectures that orchestrate the interactions among managers, deployed at different levels, to control the evolution of goals, models, and systems. Recall from Figure 8.3 that goals, models, and the structure and behavior of systems correspond to the artifacts that characterize the *what* dimension of runtime software evolution (i.e., the elements of the system susceptible to dynamic evolution). Runtime evolution mechanisms based on ACRA can take advantage of both feedforward and feedback control. Feedforward control can be implemented through the interactions between manual managers (i.e., business and system administrators) and the autonomic managers deployed at each level of the runtime evolution architecture (cf. dashed arrows in Figure 8.10). Feedback control can be exploited to orchestrate the evolution of related artifacts located at two different levels (i.e., inter-level feedback represented by continuous arrows), and to implement autonomic managers at each level of the architecture (i.e., intra-level feedback control represented by autonomic managers). Thick arrows depict the information flow between knowledge bases and autonomic managers.

The evolution architecture depicted in Figure 8.10 applies to our e-commerce scenario (cf. Section 8.2). Goals correspond to QoS requirements that must be satisfied for the retailers that use the e-commerce platform (e.g., performance as a measure of efficiency). Models correspond to specifications of the software architecture configurations defined for each QoS requirement. System artifacts correspond to the actual servers, components, and services. As explained in Section 8.5, feedback control can be exploited to evolve the configuration of servers with the goal of guaranteeing system availability. For this, autonomic managers working at the top level of the architecture monitor and manage changes in system goals. Whenever a goal is modified, these managers use inter-level control to trigger the evolution of models at the next level. Subsequently, autonomic managers in charge of controlling the evolution of models provide managers at the bottom level with control actions that will trigger the reconfiguration of the system. In this application scenario feedforward enables the runtime evolution of the system to satisfy changes in requirements of retailers. For example, when a new retailer becomes a customer, business administrators can feed the evolution mechanism with new goals to be satisfied. The definition of a new goal triggers the inter- and intra-level feedback mechanisms to evolve models and systems according to the new requirement. When the new requirement cannot be satisfied with existing models and adaptation strategies, feedforward allows system administrators to feed the system with new ones.

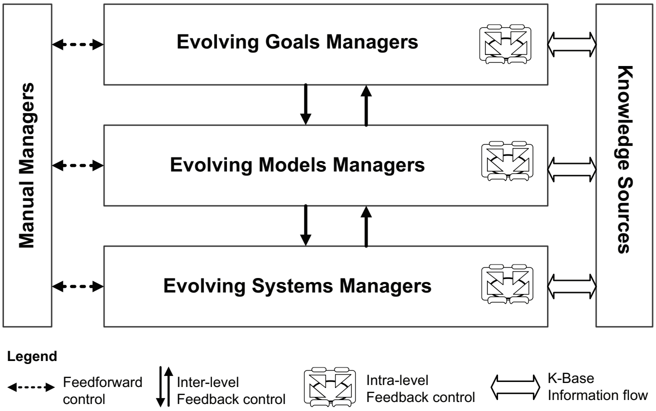


Fig. 8.10: A runtime evolution architecture based on ACRA supporting the evolution of software artifacts at the three levels of the *what* dimension: goals, models, and system structure and behavior. Dashed arrows represent feedforward control mechanisms, continuous arrows inter-level feedback control mechanisms, autonomic managers intra-level feedback control, and thick arrows the information flow between autonomic managers and knowledge bases.

8.6.4 Self-Management Properties

In autonomic computing self-managing systems expose one or more of the following properties: self-configuration, self-optimization, and self-protection, self-healing [457]. These properties are also referred to as the *self-* adaptation properties* of SAS systems and concern the *how* perspective of the runtime software evolution dimensions depicted in Figure 8.3. Of course, they also relate to the *why* and *what* perspectives since runtime evolution methods target evolution goals and affect artifacts of the system. Self-* properties allow the realization of maintenance tasks at runtime. For example, self-configuration can be used to realize adaptive maintenance, which goal is to modify the system to guarantee its operation under changing environments; self-healing and self-protection to realize corrective maintenance, which goal is to recover the system from failures or protect it from hazardous situations; and self-optimization to implement perfective maintenance, which goal is usually to improve or preserve quality attributes. Each self-* property can be further sub-classified. For example self-recovery is often classified as a self-healing property.

Kephart and Chess characterized self-* properties in the context of autonomic computing [457]. In this subsection we characterize these properties in the context of runtime software evolution. Table 8.1 presents selected examples of self-adaptive solutions characterized with respect to the three dimensions of runtime software evolution depicted in Figure 8.3, where the *how* dimension corresponds to self-* properties.

Table 8.1: Examples of self-adaptation approaches characterized with respect to the *why*, *what* and *how* dimensions of runtime software evolution.

Approach	Why	What	How
Cardellini et al. [166]	Changing environments	System structure	Self-configuration
Dowling and Cahill [261]		System structure	
Parekh et al. [678]		System behavior	
Tamura et al. [821]		System structure	
Villegas et al. [898]	Changing requirements	Structure/behavior	
Kumar et al. [484]	Changing environments	System structure	Self-optimization
Solomon et al. [786]			
Appleby et al. [44]			
Baresi and Guinea [73]	Malfunctions	System behavior	Self-healing
Candea et al. [156]		System structure	
Ehrig et al. [270]		System behavior	
White et al. [917]	Malfunctions	System behavior	Self-protection

Self-Configuration

Self-configuration is a generic property that can be implemented to realize any other self-* property. Systems with self-configuration capabilities reconfigure themselves, automatically, based on high level policies that specify evolution goals, as well as reconfiguration symptoms and strategies. Self-configuring strategies can affect either the system's behavior or structure. Behavioral reconfiguration can be achieved by modifying parameters of the managed system, whereas structural reconfiguration can be achieved by modifying the architecture. Self-configuration strategies can be applied to our exemplar e-commerce system to guarantee new quality attributes that may be contracted with retailers. This could be realized by reconfiguring the system architecture at runtime based on design patterns that benefit the contracted qualities [820]. In self-configuration approaches the *why* dimension of runtime software evolution may correspond to changing requirements or environments, and malfunctions, whereas the *what* dimension concerns the system structure or behavior, depending on the way the target system is adapted.

The first group of approaches in Table 8.1 illustrates the application of self-configuration as the means to runtime software evolution. The prevalent reason for evolution (i.e., the *why* dimension) is changing environments, except the approach by Villegas et al. whose reason for evolution is the need for supporting changes in requirements. Cardellini and her colleagues implemented Moses (MOdel-based SElf-Adaptation of SOA systems), a self-configuration solution for service selection and workflow restructuring, with the goal of preserving selected quality attributes under environmental disruptions [166]. To reconfigure the system, MOSES's self-adaptive solution, based on a runtime model of the system, calculates the service composition that better satisfies the QoS levels contracted with all the users of the system. This mechanism is based on a linear programming optimization problem that allows them to efficiently cope with changes in the observed variables of the execution environment. Downling and Cahill [261], as well as Tamura et al. [820, 821] proposed self-configuration approaches applied to component-based software systems with the goal of dealing with the violation of contracted qualities due to changes in the environment. Both approaches use architectural reflection to affect the system structure. Self-adaptation approaches can be classified along a spectrum of techniques that ranges from pure control-based to software-architecture-based solutions [899]. Control-based approaches are often applied to the control of the target system's behavior and hardware infrastructure rather than its software architecture. Parekh et al. proposed a more control theory oriented approach for self-configuration [678], where the *what* dimension of software evolution concerns the system behavior. The evolution objective in this case is also the preservation of quality properties, and the evolution is realized through a controller that manipulates the target system's tuning parameters. Villegas et al. implemented SmarterContext, a context management infrastructure that is able to evolve itself to address changes in monitoring requirements [894, 898]. SmarterContext can support dynamic monitoring in several self-adaptation and runtime evolution scenarios. In particular, they have applied their solution to support the runtime evolution of software systems with the goal of addressing changes in SLA dynamically.

Self-Optimization

Perfective maintenance often refers to the improvement of non-functional properties (i.e., quality requirements and *ility* properties) of the software system such as performance, efficiency, and maintainability [170]. Perfective maintenance can be realized at runtime using self-optimization. Self-optimizing systems adapt themselves to improve non-functional properties according to business goals and changing environmental situations. For example, in our e-commerce scenario the capacity of the system can be improved through a self-configuration mechanism implemented to increase the number of services available for processing purchase orders. This operation affects the software architecture of the system, and is performed to address changing context situations (e.g., critical changes in the system load due to shopping seasons or special offers that become extremely popular). Therefore, the *what* dimension concerns the structure of the system, and the *why* dimension changing environments that must be monitored continuously.

Examples of self-optimization mechanisms implemented through self-adaptation are the Océno approach proposed by Appleby et al. [44], the middleware for data stream management contributed by Kumar et al. [484], and the self-adaptation approach for business process optimization proposed by Solomon et al. [786]. Appleby and her IBM colleagues proposed a self-optimization approach that evolves the infrastructure of an e-business computing utility to satisfy SLAs under peak-load situations. In their approach, self-adaptive capabilities support the dynamic allocation of computing resources according to metrics based on the following variables: active connections/server, overall response time, output bandwidth, database response time, throttle rate, admission rate, and active servers. These variables constitute the relevant context that is monitored to decide whether or not to evolve the system. Kumar and colleagues proposed a middleware that exposes self-optimizing capabilities, based on overlay networks, to aggregate data streams in large-scale enterprise applications. Their approach deploys a data-flow graph as a network overlay over the enterprise nodes. In this way, processing workload is distributed thus reducing the communication overhead involved in transmitting data updates. The configuration and optimization of the deployed overlay is performed by an autonomic module that focuses on maximizing a utility function where monitoring conditions involve several internal context variables. Solomon et al. proposed an autonomic approach that evolves software systems to optimize business processes. At runtime, their approach predicts the business process that better addresses SLAs while optimizing system resources. The prediction is based on a simulation model whose parameters are tuned at runtime by tracking the system with a particle filter.

Balasubramanian et al. introduce a mathematical framework that adds structure to problems in the realm of self-optimization for different types of policies [66]. Structure is added either to the objective function or the constraints of the optimization problem to progressively increase the quality of the solutions obtained using the greedy optimization technique. They characterized and analyzed several optimization problems encountered in the area of self-adaptive and self-managing systems to provide quality guarantees for their solutions.

Self-Healing

Corrective maintenance is a main issue in software evolution. In complex systems it is particularly challenging due to the difficulty of finding the root cause of failures. This is in part because the situation that triggers the failure may be no longer available by the time the analysis is performed. Self-healing systems are equipped with feedback loops and runtime monitoring instrumentation to detect, diagnose, and fix malfunctions that are originated from the software or hardware infrastructure. In our e-commerce runtime evolution example, self-healing applies to the assurance of the system availability. In particular, the runtime evolution strategy described in Section 8.5 applies behavioral reconfiguration to control the maximum number of connections that each server can satisfy, with the goal of having always processing capacity available to replace faulty servers. In this situation, the *what* dimension of runtime evolution corresponds to the behavior of the system, whereas the *why* corresponds to malfunctions. The third group in Table 8.1 corresponds to runtime evolution approaches focused on self-healing. Self-healing evolution is usually triggered by malfunctions, and can affect both the structure (e.g., Candea et al. [156]) and behavior of the system (e.g., Baresi and Guinea [73], as well as Ehrigh et al. [270]). Candea et al. built a self-recovery approach that uses recursive micro-reboots to optimize the maintenance of Mercury, a commercial satellite ground station that is based on an Internet service platform. Baresi and Guinea implemented a self-supervising approach that monitors and recovers service-oriented systems based on user-defined rules. Recovery strategies in their approach comprise a set of atomic recovery actions such as the rebinding and halt of services. Ehrigh et al. implemented self-healing capabilities using typed graphs and graph-based transformations to model the system to be adapted and the adaptation actions, respectively.

Self-Protection

Self-protection can be classified as perfective maintenance. Self-protected systems are implemented with feedback loops to evolve themselves at runtime to counteract malicious attacks and prevent failures. White et al. proposed an approach and methodology to design self-protecting systems by exploiting model-driven engineering [917].

8.7 Self-Adaptation Enablers for Runtime Evolution

The continuing evolution of software systems, and the uncertain nature of execution environments and requirements have contributed to blur the line between design time (or development time) and runtime [72, 280]. As a result, activities that have been traditionally performed off-line must now also be performed at runtime. These activities include the maintenance of requirements and design artifacts, as well as the validation and verification (V&V) of software system. This section summarizes self-adaptation enablers from the perspective of runtime software evolution. We concentrate on requirements and models at runtime as well as runtime monitoring and V&V.

8.7.1 Requirements at Runtime

The concept *requirements at runtime* refers to the specification of functional and non-functional requirements using machine-processable mechanisms [748]. As discussed in Chapter 1 requirements form part of the evolving artifacts in a software system. Therefore, having requirement specifications available at runtime is particularly important for evolution scenarios where the *why* dimension corresponds to changing requirements, and the *what* to system goals. Aspects of runtime software evolution that rely on runtime specifications of requirements include:

- The specification of evolution goals and policies by business and system administrators.
- The control of evolving goals. For example, by the autonomic managers defined at the first level of runtime software evolution in the ACRA-based architecture depicted in Figure 8.10.
- The specification of adaptation properties that must be satisfied by evolution controllers. Adaptation properties refer to qualities that are important for controllers to operate reliably and without compromising the quality of the evolving system. Examples of adaptation properties include properties gleaned from control engineering such as short settling time, accuracy, small overshoot, and stability. A characterization of these properties from the perspective of software systems is available in [899].
- The monitoring of the execution environment and the system to identify the need for evolution.
- The preservation of the context-awareness along the evolution process. Runtime requirement specifications must maintain an explicit mapping with monitoring strategies to the adaptation of monitoring infrastructures as required by the evolution of the system.
- The management of uncertainty due to changes in requirements. Requirements are susceptible to changes in the environment, user preferences, and business goals. Thus, runtime specifications of requirements allow the system to maintain an explicit mapping between requirements and aspects that may affect them [921]. Moreover, they are crucial for the *re-documentation* phase of the software evolution process (cf. Figure 8.8), since they ease the maintenance of the coherence between the actual system implementation and its documentation.
- The validation and verification of the system along the runtime evolution process (cf. Figure 8.8). Requirements at runtime allow the specification of aspects to validate and verify [823].

8.7.2 Models at Runtime

The concept *models at runtime* refers to representations of aspects of the system that are specified in a machine-readable way, and are accessible by the system at runtime. In the context of runtime software evolution, runtime models provide the system with up-to-date information about itself and its environment. Moreover, runtime models are themselves artifacts that evolve with the system (i.e., design specifications as defined in the *what* dimension of software evolution, cf. Figure 8.3).

To Probe Further

The *models@run.time* research community defines a runtime model as a "causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective" [52, 92, 115]. In the context of self-adaptation and runtime evolution, the environment that affects the system in the accomplishment of its goals is also an aspect that requires runtime models for its specification [177, 221, 823, 900].

Runtime models provide effective means to evolve software systems at runtime. For example, in our e-commerce scenario administrators can modify the runtime model that specifies the software architecture to be implemented to improve the capacity of the system for processing purchase orders. The modification of the software architecture model will trigger the adaptation of the software system. This model-based evolution mechanism can be implemented using ACRA as depicted in Figure 8.10. The middle layer contains the autonomic managers in charge of evolving runtime models.

Runtime models also support the implementation of runtime evolution mechanisms based on adaptive control. In particular using MRAC and MIAC. In MRAC, the reference model used by the adaptation algorithm corresponds to a runtime model that can be adjusted dynamically to change the controller's parameters (cf. Figure 8.5). In this case the runtime model is adjusted using feedforward control, that is by a mechanism external to the system (e.g., a human manager). In MIAC, the reference model is also a runtime model but adjusted using system identification methods. That is, the model is automatically adapted based on stimuli generated within the boundaries of the system (cf. Figure 8.6).

In the context of runtime software evolution, runtime models are important among others to represent evolution conditions, requirements and properties that must be assured, monitoring requirements and strategies, and to evolve the system or the evolution mechanism via model manipulation.

8.7.3 Runtime Monitoring

The *why* dimension of runtime software evolution characterizes reasons for evolving the system (cf. Figure 8.3). Runtime monitoring concerns the sensing and analysis of context information from the execution environment, which includes the system itself, to identify the need for evolution.

Context can be defined as any information useful to characterize the state of individual entities and the relationships among them. An entity is any subject that can affect the behavior of the system and/or its interaction with the user. Context information must be modeled in such a way that it can be pre-processed after its acquisition from the environment, classified according to the corresponding domain, handled to be provisioned based on the systems requirements, and maintained to support its dynamic evolution [896]. Based on this definition of context information, and from the perspective of runtime software evolution, runtime monitoring must support context representation and management to characterize the system's state with respect to its environment and evolution goals. Regarding context representation, operational specifications of context information must represent semantic dependencies among properties and requirements to be satisfied, evolution and V&V strategies, and the context situations that affect the evolution of the system. In highly uncertain situations, an important requisite is the representation of context such that its specifications can adapt dynamically, according to changes in requirements and the environment. Regarding context management, monitoring solutions must support every phase of the context information life cycle, that is context acquisition, handling, provisioning, and disposal. Context acquisition concerns the sensing of environmental information, handling refers to the analysis of the sensed information to decide whether or not to evolve the system, context provisioning allows evolution planers and executors to obtain environmental data that affect the way of evolving the system, and context disposal concerns the discard of information that is no longer useful for the evolution process. Moreover, to preserve context-awareness along the evolution process, monitoring infrastructures must be instrumented with self-adaptive capabilities that support the deployment of new sensors and handlers. For example, in our e-commerce scenario changes in the requirements due to the need for serving a new customer may imply new context types to be monitored.

8.7.4 Runtime Validation and Verification

Software validation and verification (V&V) ensures that software products satisfy user requirements and meet their expected quality attributes throughout their life cycle. V&V is a fundamental phase of the software evolution process [936]. Therefore, when the evolution is performed at runtime, V&V tasks must also be performed at runtime [823].

Aspects of runtime V&V that require special attention in the context of runtime software evolution include: (i) the dynamic nature of context situations; (ii) what to validate and verify, and its dependency on context information; (iii) where to validate—whether in the evolving system or the evolution mechanism; and (iv) when to perform V&V with respect to the adaptation loop implemented by evolution controllers. Researchers from communities related to SAS systems have argued for the importance of instrumenting the adaptation process with explicit runtime V&V tasks [221, 823]. In particular by integrating *runtime validators and verifiers*—associated with the planning phase of evolution controllers, and *V&V monitors*—associated with the monitoring process. The responsibility of runtime validators & verifiers is to verify each of the outputs (i.e., evolution plans) produced by the planner with respect to the properties of interest. The execution of an adaptation plan on a given system execution state implies a change of the system state. Therefore, the verification requirements and properties should be performed before and/or after instrumenting the plan. The responsibility of V&V monitors is to monitor and enforce the V&V tasks performed by runtime validators & verifiers.

8.8 Realizing Runtime Evolution in SmarterContext

Figure 8.11 depicts a partial view of the software architecture of SmarterContext using Service Component Architecture (SCA) notation. Further details about this architecture are available in [894]. SCA defines a programming model for building software systems based on Service Oriented Architecture (SOA) design principles [665]. It provides a specification for both the composition and creation of service components. *Components* are the basic artifacts that implement the program code in SmarterContext. *Services* are the interfaces that expose functions to be consumed by other components. *References* enable components to consume services. *Composites* provide a logical grouping for components. *Wires* interconnect components within a same composite. In a composite, interfaces provided or required by internal components can be *promoted* to be visible at the composite level. *Properties* are attributes modifiable externally, and are defined for components and composites. *Composites* are deployed within an *SCA domain* that generally corresponds to a processing node.

Component *GoalsManager* has a twofold function. First, it allows system administrators to modify system goals (e.g. SLAs) at runtime. These changes in SLAs imply modifications in the monitoring requirements thus triggering the runtime evolution of SmarterContext. Second, it enables system administrators to define and modify context reasoning rules as part of the evolution of the monitoring infrastructure at runtime. Components *ContextManager* and *DynamicMonitoringInfrastructure* implement the context monitoring functionalities of SmarterContext. *ContextManager* includes software artifacts that integrate context information into context repositories, maintain and dispose existing contextual data, provide introspection support, and update the inventory of components managed dynamically as part of the runtime evolution process. *DynamicMonitoringInfrastructure* corresponds to the adaptive part of the monitoring infrastructure, and implements the context gathering, processing and provisioning tasks. In the case study described in Section 8.2, these tasks are performed by the components depicted within the dark gray box in Figure 8.11, *ContextGatheringAndPreprocessing* and *ContextMonitoring*. These components are deployed as part of the runtime evolution process to monitor the new bandwidth quality factor that is added after renegotiating the performance SLA. Component *MFLController* includes the artifacts that implement the feedback loop in charge of controlling the runtime evolution of our context management infrastructure.

The two *AdaptationMiddleware* components are an abstraction of FraSCAti [763] and QoS-CARE [820, 821], which constitute the middleware that provides the structural adaptation capabilities that allow the evolution of SMARTERCONTEXT at runtime.

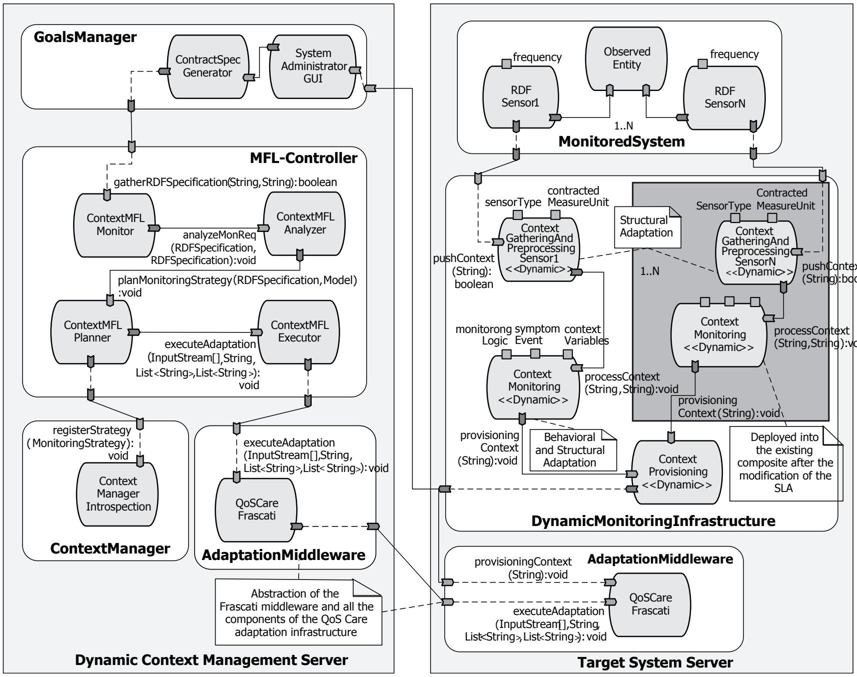

GoalsManager
ContractSpec Generator
System Administrator GUI
MFL-Controller
gatherRDFSpecification(String,String):boolean
ContextMFL Monitor
analyzeMonReq (RDFSpecification, RDFSpecification):void
ContextMFL Analyzer
planMonitoringStrategy(RDFSpecification,Model) :void
ContextMFL Planner
executeAdaptation (InputStream[],String, List<String>,List<String>): void
ContextMFL Executor
registerStrategy (MonitoringStrategy): void
Context Manager Introspection
ContextManager
executeAdaptation (InputStream[],String, List<String>,List< String>): void
QoSCare Frascati
AdaptationMiddleware
Abstraction of the Frascati middleware and all the components of the QoS Care adaptation infrastructure
Dynamic Context Management Server
frequency
Observed Entity
frequency
RDF Sensor1
RDF SensorN
1..N
MonitoredSystem
contracted
sensorType MeasureUnit
Context GatheringAnd Preprocessing Sensor1 <<Dynamic>>
pushContext (String): boolean
Structural Adaptation
Contracted
SensorType MeasureUnit
Context GatheringAnd Preprocessing SensorN <<Dynamic>>
1..N
pushContext (String):boolean
Context Monitoring <<Dynamic>>
processContext (String,String):void
monitorong Logic
symptom Event
context Variables
Context Monitoring <<Dynamic>>
processContext (String,String):void
provisioning Context(String):void
provisioning Context (String):void
Behavioral and Structural Adaptation
Context Provisioning <<Dynamic>>
Deployed into the existing composite after the modification of the SLA
DynamicMonitoringInfrastructure
provisioningContext (String):void
AdaptationMiddleware
QoSCare Frascati
executeAdaptation (InputStream[],String, List<String>,List<String>): void
Target System Server

Fig. 8.11: SMARTERCONTEXT's software architecture. The components depicted within the dark gray box represent software artifacts deployed dynamically as a result of the runtime evolution process.

8.8.1 Applying the MAPE-K Loop Reference Model

Composite *MFL-Controller* is an implementation of the MAPE-K loop reference model that enables our monitoring infrastructure with dynamic capabilities to evolve at runtime through the adaptation of (i) the set of context reasoning rules supported by the reasoning engine, (ii) the context monitoring logic that evaluates gathered context against monitoring conditions, and (iii) the context gathering and provisioning infrastructure.

The initial component of composite *MFL-Controller* is *ContextMFLMonitor*. This component receives the SLA specification in XML/RDF format from the user, creates an *RDFSpecification* object from the received specification, looks for a previous version of this SLA, stores the new SLA specification in its knowledge base, and finally provides component *ContextMFLAnalyzer* with two *RDFSpecification* objects that represent the new and former (if applicable) SLA specifications. SLA specifications in SMARTERCONTEXT are named *control objectives (COb) specifications* since they refer to the goals that drive the adaptive behavior of the system. [822]. SMARTERCONTEXT realizes COb specifications using Resource Description Framework (RDF) models [894]. RDF is a semantic web framework for realizing semantic interoperability in distributed applications [558].

The second component of the *MFL-Controller* composite is *ContextMFLAnalyzer*, which is in charge of analyzing changes in COb specifications, and specifying these changes in an RDF model. After analyzing changes in COb specifications, this component invokes *ContextMFLPlanner* and provides it with the new COb specification and the model that specifies the changes. If there is not previous COb specification, *ContextMFLAnalyzer* simply provides *ContextMFLPlanner* with the new COb specification and a null Model.

The third component of this MAPE-K loop is *ContextMFLPlanner*. This component is in charge of synthesizing new monitoring strategies as well as changes in existing ones. We define monitoring strategies as an object that contains (i) a set of implementation files for the SCA components to be deployed, the specification of the corresponding SCA composite, and two lists that specify SCA services and corresponding references. These services and references allow the connection of sensors exposed by the monitored third parties to context gatherers exposed by the SMARTERCONTEXT infrastructure, and context providers' gatherers exposed by third parties to context providers exposed by the SMARTERCONTEXT infrastructure; or (ii) a set of context reasoning rules to be added or deleted from the SMARTERCONTEXT's reasoning engine.

The last component of composite *MFL-Controller* is *ContextMFLExecutor*. This component invokes the services that will trigger the adaptation of the context monitoring infrastructure. The SMARTERCONTEXT monitoring infrastructure can be adapted at runtime by either (i) changing the set of context reasoning rules, (ii) modifying the monitoring logic, (iii) deploying new context sensors, and context gathering, monitoring and provisioning components. The case study presented in this chapter concerns mechanisms (ii) and (iii).

Table 8.2 summarizes the self-adaptive capabilities of SMARTERCONTEXT that allow its runtime evolution. The first column refers to changes in COb specifications that may trigger the evolution of SMARTERCONTEXT at runtime; the second column presents the type of control action used to adapt the monitoring infrastructure; the third column describes the evolution effect obtained on SMARTERCONTEXT after performing the adaptation process.

Table 8.2: Self-adaptive capabilities of SmarterContext that support its evolution at runtime

3pt4pt

Changes in COb Specifications	Control Actions	Evolution Effects
Addition/deletion of reasoning rules	Parameters (i.e., RDF rules) affecting the behavior of SmarterContext	Modified reasoning capabilities of the reasoning engine
Addition/deletion of context providers and/or consumers	Discrete operations affecting the SmarterContext software architecture	Changes in the set of deployed context sensing, gathering, and provisioning components
Addition or renegotiation of system objectives	Parameters (i.e., arithmetic and logic expressions) affecting the behavior of SmarterContext	Changes in existing monitoring logic
	Discrete operations affecting the SmarterContext software architecture	Changes in the set of deployed context sensing, gathering, monitoring and provisioning components

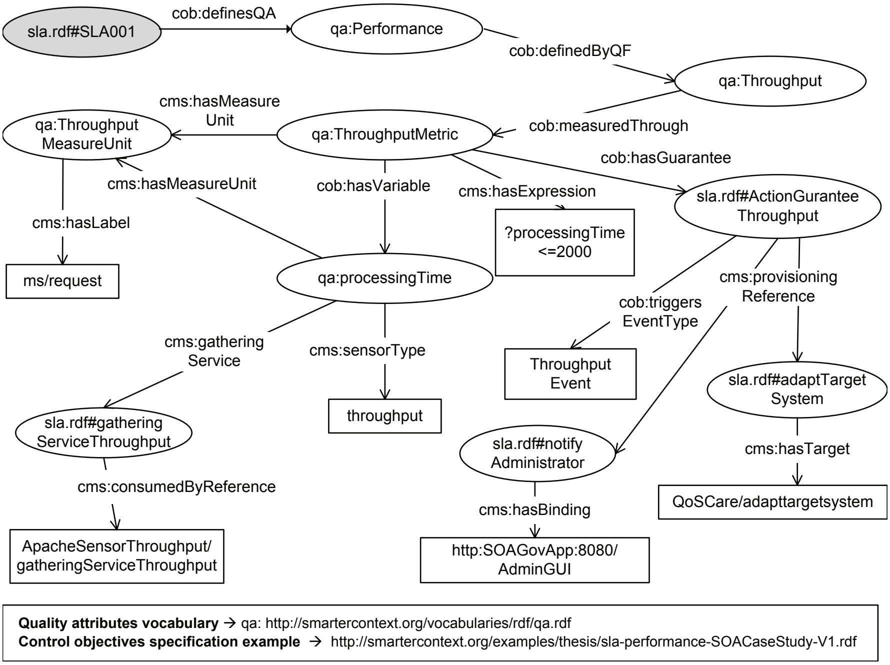
8.8.2 Applying Requirements and Models at Runtime

To control the relevance of the monitoring mechanisms implemented by SmarterContext with respect to control objectives (e.g., the contracted QoS specified in SLAs), it is necessary to model and map these objectives explicitly to monitoring requirements. These models must be manipulable at runtime. This section illustrates the use of models at runtime to maintain operative specifications of requirements and evolving monitoring strategies during execution.

Control objectives (COb) specifications allow SmarterContext to synthesize new and change existing monitoring strategies according to changes in contracted conditions. In the case study described in Section 8.2, COb specifications correspond to SLAs that not only define the contracted QoS and corresponding metrics, but also specify monitoring conditions associated with metrics, sensing interfaces and guarantee actions.

8.8.2.1 Control Objectives Specifications

Figure 8.12 represents, partially, a COb specification for the performance SLA that resulted from the first negotiation in our case study. This specification is a concrete instantiation of the *control objectives* ontology for QoS contracts in SmarterContext. This ontology allows the specification of control objectives (e.g., SLAs) mapped to elements of both monitoring strategies and adaptation mechanisms represented by entities derived from the *context monitoring strategy (cms)* ontology [894].



Quality attributes vocabulary → qa: http://smartercontext.org/vocabularies/rdf/qa.rdf
Control objectives specification example → http://smartercontext.org/examples/thesis/sla-performance-SOACaseStudy-V1.rdf

Fig. 8.12: A control objectives specification example for the throughput quality attribute defined in the first negotiation of the performance SLA.

Namespace *qa:* corresponds to the vocabulary that characterizes quality attributes mapped to quality factors in the study. This version of the performance SLA defines a throughput quality factor, measured through a throughput metric (*qa:ThroughputMetric*) that is composed of a single variable (*qa:processingTime*). This variable is involved in the metric expression $?processingTime \leq 2000$, measured in terms of *ms/request* (as defined in the element *qa:ThroughputMeasureUnit*) and associated with a service identified as *sla.rdf#gatheringServiceThroughput*. The action guarantee *sla.rdf#ActionGuaranteeThroughput* associated with the throughput quality factor is associated with two provisioning references. The first one, *sla.rdf#adaptTargetSystem* is to invoke the service in charge of activating the adaptation process. The second one, *sla.rdf#notifyAdministrator*, is to inform business administrators about the violation of the contracted throughput conditions.

8.8.2.2 Synthesizing Monitoring Strategies at Runtime

In SMARTERCONTEXT monitoring strategies can be generated dynamically from COb specifications such as the one depicted in Figure 8.12. A monitoring strategy is defined as *DynamicMonitoringInfrastructure* composite (cf. the architecture depicted in Figure 8.11) that specifies components for context gathering, preprocessing, monitoring, and provisioning. These strategies are dynamic because SMARTERCONTEXT supports at runtime the modification of the monitoring logic, and, enabled by an architectural adaptation middleware, the deployment of new context management components.

Figure 8.13 illustrates, for the case study presented in this chapter, the generation of the *DynamicMonitoringInfrastructure* composite (cf. the highlighted composite in the same figure) from the COb specification presented in Figure 8.12. The RDF subgraphs associated with elements *sla.rdf#ActionGuaranteeThorughput* and *qa:ThroughputMetric* constitute the foundational elements for generating the *DynamicMonitoringInfrastructure* composite. The dotted connectors associate elements of the COb specification with the corresponding architectural artifact. For example, the connector labeled with number 1 indicates that component *ContextMonitoring* is dynamically generated from metric *qa:ThroughputMetric*.

When an existing SLA is renegotiated, the *GoalsManager* composite generates a new COb specification. Then, component *ContextMFLMonitor* defined in composite *MFL-Controller* gathers this specification, analyzes whether it corresponds to renegotiated SLA, and if so, generates a new plan with consists of the set of new context gathering and monitoring components to be deployed. Finally, the *ContextMFLExecutor* component executes the plan to deploy the new components.

8.9 Open Challenges

Many challenges remain open in the engineering of software systems with self-adaptive capabilities. Cheng et al. [177], as well as de Lemos et al. [221] characterize a comprehensive set of open questions and opportunities that are important to advance the field. The research roadmap by Cheng et al. focuses on development methods, techniques and tools required for the engineering of self-adaptive systems. This first roadmap groups challenges into four main topics: modeling dimensions, requirements, engineering, and assurances. The research roadmap by de Lemos et al. complements the first one while focusing on a different set of topics: design space, processes, decentralization of control loops, and practical runtime verification and validation (V&V).

Most difficult challenges in self-adaptation relate to the lack of effective methods for assuring the dynamic behavior of adaptive systems under high levels of uncertainty. In this realm control science and runtime models are research areas that deserve special attention. Control science can be defined as a systematic way to study certifiable V&V methods and tools to allow humans to trust decisions made

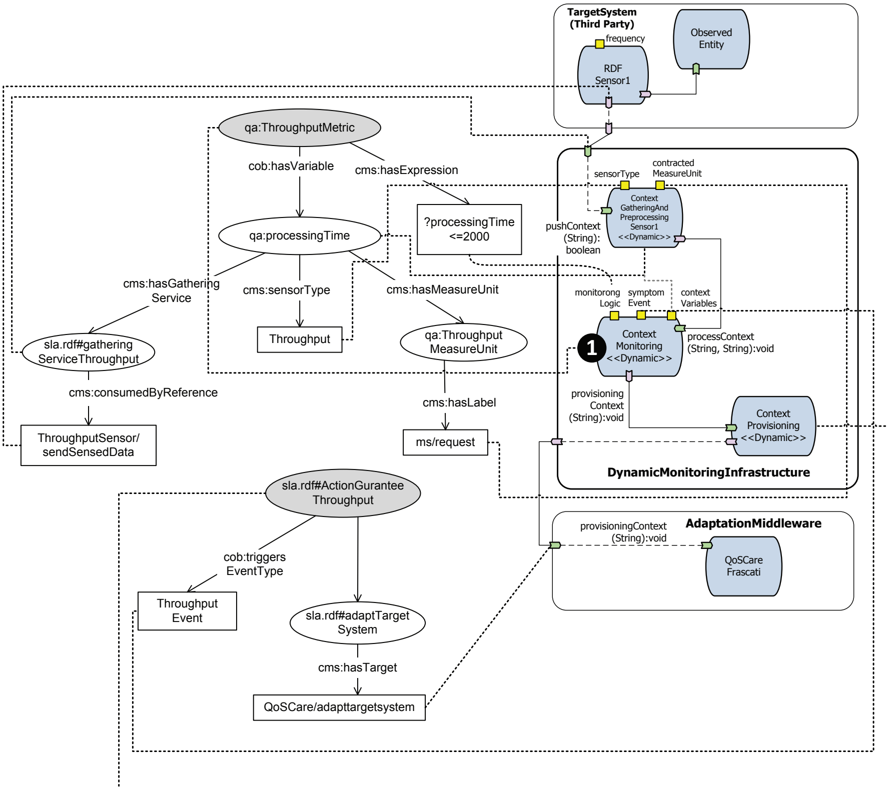


Fig. 8.13: Synthesizing monitoring strategies dynamically. The dashed connectors associate the element from the COb specification to the corresponding architectural artifact in the architecture.

by self-adaptive systems. In a 2010 report, Dahm identified control science as a top priority for the US Air Force (USAF) science and technology research agenda for the next 20 years [212]. Certifiable V&V methods and tools are critical for the success of self-adaptive systems. One systematic approach to control science for adaptive systems is to study V&V methods for the mechanisms that sense the dynamic environmental conditions and the target system behavior, and act in response to these conditions by answering the questions *what*, *when* and *how* to adapt [823].

Research on models at runtime study the exploitation of models available while the system executes. The goal is to provide effective mechanisms for complexity management in software systems whose behavior evolves at runtime [52]. Runtime models have been recognized as important enablers for the assurance of self-adaptive systems. We identified three subsystems that are key in the design of effective context-driven self-adaptation: the control objectives manager, the adaptation controller, and the context monitoring system [900]. These subsystems represent three levels of dynamics in self-adaptation that can be controlled through three feedback loops, i.e., the control objectives, the adaptation, and the monitoring feedback loops, respectively. We argue that runtime models provide abstractions that are crucial to support the feedback loops that control these three levels of dynamics. From this perspective, models at runtime could be developed specifically for each level of dynamics to support the control objectives manager, adaptation controller, and the monitoring system. At the control objectives level, models at runtime represent requirements specifications subject to assurance in the form of functional and non-functional requirements. At the adaptation level, models at runtime represent states of the managed system, adaptation plans and their relationships with the assurance specifications. At the monitoring level, models at runtime represent context entities, monitoring requirements, as well as monitoring strategies and their relationships with assurance criteria and adaptation models.

8.10 Conclusions

This chapter presented fundamental concepts of control and self-adaptive systems engineering, and their application to runtime software evolution. Departing from seminal aspects of "traditional" software evolution such as the change mini-cycle, Lehman's laws, and dimensions of software evolution, we discussed the complex dynamics of software systems and the way runtime software evolution can help deal with this complexity. Self-adaptation can be considered as short-term software evolution. Therefore, foundational concepts of self-adaptive software such as feedback, feedforward and adaptive control, the MAPE-K loop, ACRA reference architecture and self-* properties; and enabling mechanisms, such as requirements and models at runtime, context monitoring, and runtime V&V must be well understood by researchers, engineers and students interested in the evolution of highly dynamic and continuously running software systems.

It is important to point out that despite its benefits, runtime evolution is not always the best solution. We analyzed the need for runtime software evolution using the notion of its benefit cost ratio based on three selected variables: frequency of changes in requirements and environments, uncertainty, and off-line evolution cost. As part of this analysis we discussed trade-offs between runtime software evolution and the complexity added by the software artifacts required to automate software evolution tasks.

Rather than providing an exhaustive explanation of the application of self-adaptation and control theory to the engineering of runtime evolution mechanisms, or present new approaches to solve existing software evolution challenges, the goal of this chapter is to provide practitioners, researchers, and students with an overview of how the research work that is being conducted in the field of self-adaptive software relates to software evolution.

Chapter 9
Evolution of Software Product Lines

Goetz Botterweck and Andreas Pleuss

Summary. A *Software Product Line* (*SPL*) aims to support the development of a family of similar software products from a common set of shared assets. SPLs represent a long-term investment and have a considerable life-span. In order to realize a return-on-investment, companies dealing with SPLs often plan their product portfolios and software engineering activities strategically over many months or years ahead. Compared to single system engineering, SPL evolution exhibits higher complexity due to the variability and the interdependencies between products. This chapter provides an overview on concepts and challenges in SPL evolution and summarizes the state of the art. For this we first describe the general *process* for SPL evolution and general *modeling concepts* to specify SPL evolution. On this base, we provide an overview on the state-of-the-art in each of the main process tasks which are *migration* towards SPLs, *analysis* of (existing) SPL evolution, *planning* of future SPL evolution, and *implementation* of SPL evolution.

We thank Howell Jordan for comments on an early version of the text. 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).

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_9, © Springer-Verlag Berlin Heidelberg 2014

9.1 Introduction

A *Software Product Line* (*SPL*) aims to support the development of a family of similar software products from a common set of shared assets [193, 688, 910]. By applying SPL practices, organizations are able to achieve significant improvement in time-to-market, engineering and maintenance costs, portfolio size, and quality [193]. SPLs have been commercially applied in many industry domains [784] including embedded systems, web and mobile applications.

SPLs represent a long-term investment and have a considerable life-span. Moreover, SPLs scale to a considerable size and are often embedded in larger structures, i. e. they consist of subsystems and are part of a larger supersystem. Hence, changes on an SPL can have a complex impact not only on the whole product family but also on related systems. When a change is introduced, inconsistencies are unavoidable until the change has been propagated through the system and related systems. Since usually multiple parties are involved and there are multiple changes, this can easily lead to further inconsistencies. All this needs to be taken into account when considering potential changes.

Because of the long term perspective, size, and complexity, organizations dealing with SPLs need to address evolution in a systematic fashion. In this chapter, we give an overview of such systematic approaches to SPL evolution. Our goal is to provide the reader with an introduction and given an overview of the field, which allows to identify more specialized literature that provides more details of a particular aspect.

As a background, Section 9.2 summarizes the main ideas of SPLs and Section 9.3 covers basic concepts for SPL evolution. Then, Section 9.4 gives an overview of approaches to SPL evolution. More concretely, we cover modeling of SPL evolution, processes for SPL evolution, migration towards SPLs, as well as the analysis, planning and implementation of SPL evolution. Section 9.5 concludes the chapter with an overview of remaining research challenges and final thoughts.

9.2 Software Product Lines

An SPL aims to support the development of a whole family of software products through systematic reuse of shared assets [193, 688, 910]. By an *asset* we refer to any artifact that is part of the software production process, such as an architecture, a software component, a domain model, a requirements document, a formal specification, documentation, a plan, a test case, or a process description [193].

As an example for an SPL consider online shop software: while different online shops usually differ from each other – e. g. in the supported payment methods, shipping options or article types – their underlying concepts are very common and can be implemented from reusable assets. Hence, a company offering a spectrum of online shop applications can use SPL techniques to achieve systematic reuse by (1) first identifying and creating the required reusable assets and (2) deriving the individual products (i. e. different online shop implementations) from the assets created in the first step in a systematic way.

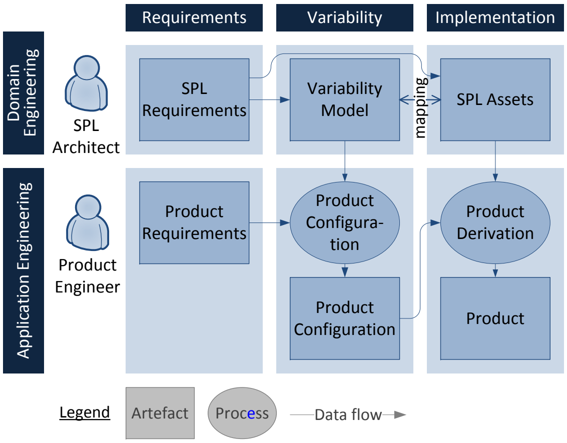


Fig. 9.1: Software Product Line Engineering (SPLE) framework

SPL engineering (*SPLE*) provides concepts on how to develop SPLs. A basic SPLE framework is shown in Figure 9.1: SPLE approaches often distinguish between *domain engineering* and *application engineering*[1]. Domain engineering deals with creating (and maintaining) the whole SPL. First, requirements for the SPL are elicited and the scope of the SPL is defined, i. e. a definition which potential products are to be supported. The variability between potential products is captured in a *variability model*. It defines the available variants, e. g. different payment methods and shipping options in an online shop, and the allowed combinations. To allow the creation of products the variants identified in the variability model need to be implemented by reusable SPL assets. A *mapping* is then specified to define which variant from the variability model is implemented by which assets.

Figure 9.2 shows example domain engineering models from an fictitious SPL for online shop software (*e-shop*). The left-hand side shows the variability model. There are several variability modeling approaches that could all be applied here, e. g. *feature models* [757], *decision models* [755] or *the orthogonal variability model* [688]. In this chapter we use feature models as a basis for the discussion. Other approaches are conceptually similar [208] and could be applied in a similar fashion.

A *feature* is a “distinguishable characteristic of a concept (e. g. component, system, etc.) that is relevant to some stakeholder of the concept” [207]. The example model shows features of an e-shop such as support for a Catalog, a Search function, and different ArticleTypes.

[1] Some approaches use different terms, like *core asset development* and *product development*, but provide essentially a similar distinction.

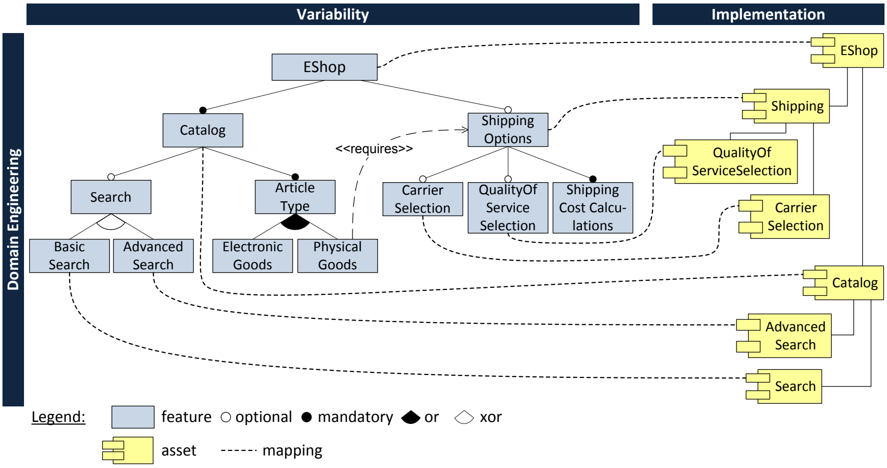


Fig. 9.2: Example domain engineering models: A feature model (left) and associated reusable implementation assets (right).

A feature model specifies all features supported by the SPL and the dependencies between them. Features are structured in a tree hierarchy. Additional constraints express further restrictions on which features can be selected or eliminated when specifying a concrete product. *Mandatory* features must always be selected if their parent is selected while *optional* features are facultative depending on the choices of the user. Features can also be grouped into *or-groups* (if the parent is selected at least one child must be selected) or *xor-groups* (if the parent is selected exactly one child must be selected). Selecting a child feature mandates that its parent is selected as well.

The feature model shown in Figure 9.2 specifies that each e-shop must support a Catalog (mandatory feature) which may include a Search function (optional) and must include an ArticleType feature (mandatory) from which at least one child has to be selected (or-group).

In addition, cross-tree constraints can be defined between arbitrary features in the model like *requires* (selecting a features requires to select another one) or *excludes* (two features mutually exclude each other). In the example, a requires constraint defines that PhysicalGoods always requires ShippingOptions to be selected.

The features in a feature model have to be implemented by reusable assets, represented by software components on the right-hand side of Figure 9.2. Additional mappings specify which features are implemented by which assets. In practice, these mappings are not always one-to-one but more complex. Moreover, features are usually mapped to different types of assets which in combination specify the complete implementation, including the *product line architecture* (*PLA* [130]), code fragments, test cases, and documentation.

During *application engineering* (see Figure 9.1), concrete products are developed based on the assets provided by the SPL. A product is defined by a *product config-*

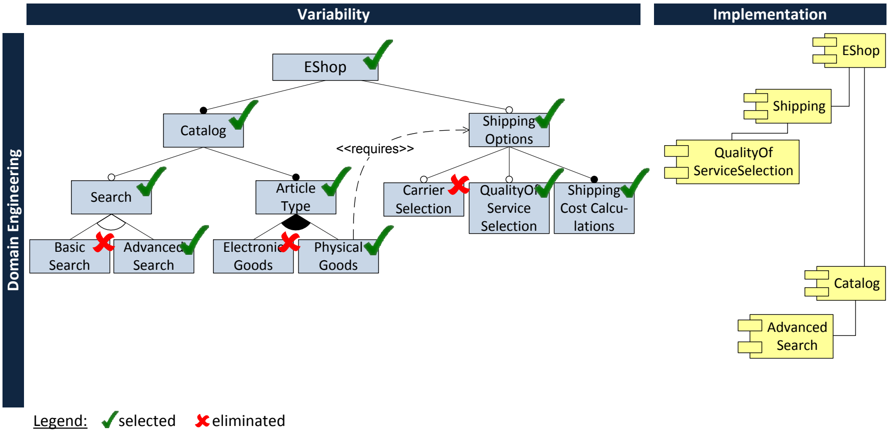


Fig. 9.3: Example product definition during application engineering: A product configuration (left) and the resulting product implementation (right).

uration, which resolves the variability by selecting from the given variants while considering the defined constraints. In the case of a feature model, this is done by selecting or eliminating features. Based on the product configuration and the feature mappings it is then possible to derive the resulting product (*product derivation*).

Figure 9.3 shows an example for one particular product configuration created during application engineering. Here, a concrete product of the example SPL is defined by selected and eliminated features (left-hand side of Figure 9.3). The sample product configuration defines an e-shop that supports AdvancedSearch, PhysicalGoods, and ShippingOptions with QualityOfServiceSelection while BasicSearch, ElectronicGoods and CarrierSelection have been eliminated from the product. The corresponding implementation for this product (right-side of Figure 9.3) is then derived based on the mappings defined during domain engineering. For instance, the component implementing AdvancedSearch is included into the product implementation while the component Search, which implements BasicSearch, is eliminated.

Often, SPL concepts are combined with techniques from model-driven software development [795]. In a model-driven SPL, the product derivation is realized by model transformations that, ideally, generate the complete product together with all documentation, test cases, etc., in a fully automated way [349, 901]. However, a more extensive use of modeling frameworks, as required for automation, can also lead to a higher maintenance effort [238]. For instance, changes in a metamodel might require all existing model instances to be migrated (*co-evolution*, see Chapter 2).

9.3 Characteristics of SPL Evolution

SPL evolution faces several challenges caused by the characteristics of SPLs:

- *Long life-span*. On the one hand, an SPL is a long-term investment that pays off the more, the more products are derived from the SPL. On the other hand, an SPL must evolve to reflect new and changed requirements for its products. Hence, an SPL will often evolve to a greater extent and over a longer period of time than the single products.
- *Large size and complexity*. As an SPL represents a whole family of products, it is of larger size and complexity than its individual products. Usually, multiple teams are involved in its creation and maintenance. Hence, knowledge can be more distributed and evolution of different the parts of an SPL can happen at different speeds.
- *More interdependencies*. Due to the systematic reuse in an SPL, there are more interdependencies between software assets. For instance, changes on the SPL level (e. g. a bug fix in a reusable asset) can affect many individual products created based on the SPL, and new requirements on individual product level can require changes with the whole SPL (e. g. substituting a reusable asset).

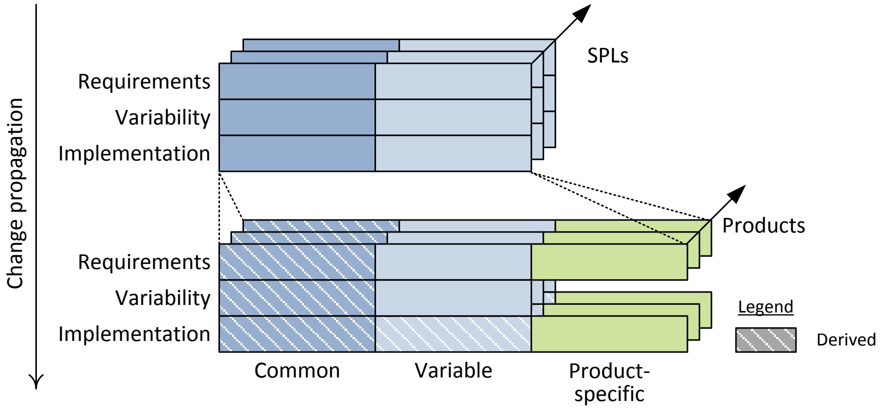


Fig. 9.4: SPL levels and assets subject to evolution.

The additional complexity in an SPL is partly caused by the different abstraction layers that have to be considered together (see Figure 9.4). In an SPL, one has to distinguish between the SPL (upper part in Figure 9.4) and its products (lower part). In addition, there can also be multiple SPLs to manage complex product portfolios [753], for instance, (i) to modularize development of very large systems into multiple SPLs with a shared architecture (*program of product lines* [230]), (ii) to handle very large variability by specifying main variability decisions on a top-level SPL while lower level SPLs specialize this further (*hierarchical product* *lines* [130]), or (iii) to support reuse across multiple domains by multiple SPLs that share some assets (*product populations* [873]). Such very large systems are sometimes developed not only by the organization's internal developers but by a whole developer community, including external developers and third-party contributors, leading to so-called *software ecosystems* (see Chapter 10).

SPLs and products, as any other software, can be defined by assets on different levels of abstraction, from requirements to the final implementation. As indicated in Figure 9.4, different abstraction levels need to be considered both in the SPL and its products: on SPL level there are the requirements and implementation for the whole SPL. In addition, variability between the products is defined. On product level there are the product requirements that influence the product-specific variability resolution (i. e. the product configuration) and the corresponding implementation.

To analyze the impact of evolutionary changes, the assets in an SPL can be further classified into three categories:

- *Common assets* are defined on the SPL level. They are part of all products and are hence directly derived from the SPL (derived assets are represented by shaded areas in Figure 9.4). For instance, in the example from Figure 9.2, the feature Catalog is defined as mandatory child feature of each e-shop. Hence, the feature Catalog, the associated requirements, and the corresponding asset Catalog are by definition part of every product.
- *Variable assets* are defined on the SPL level as well. They are part of some products depending on each product's configuration. Hence, on product level, there must be a variability decision about each variable asset (e. g. selecting or eliminating a feature) which is driven by the requirements for the particular product. The assets within the product's implementation are then derived according to this decision (i. e. including or excluding variable assets into the product). For instance, the Search feature is optional in the e-shop example, so it has to be decided on product level (based on the product's requirements) whether to include it or not.
- *Product-specific assets* are used to add functionality to individual products, e. g. some customer-specific functionality not supported by the reusable assets in the SPL. Hence, product-specific assets and their corresponding requirements reside on product level only and there is no variability configuration for them. An example in the e-shop might be a customized search function optimized for a specific article type. Usage of product-specific assets should ideally be minimized within an SPL approach as it diminishes reuse and increases maintenance effort. However, depending on the market and the business model it is not always possible to reject product-specific requirements.

Considering Figure 9.4, evolution can occur on three different levels (see [753, 809]). On the level of the *set of SPLs*, new SPLs can be added or deprecated ones can be deleted. In addition, SPLs can be merged, for instance, due to an acquisition or if SPLs become similar over time [753]. SPLs can also be split, e. g. when parts of the SPL are likely to evolve in a different direction in the future [809].

On the level of the *set of products*, new products can be added (by product derivation) and old deprecated ones can be deleted. Basically, adding new products should not require any changes to the SPL or other products. However in practice, as pointed out in [407], new feature combinations in a product configuration can sometimes lead to unforeseen effects in the implementation (e. g. *feature interactions* [133]) which then require implementation changes.

On the level of *single assets*, assets can be added, deleted, or modified. Changes have to be propagated towards lower levels of abstraction (e. g. from requirements to implementation). Changes on common assets are performed on the SPL level and affect all derived products. Changes on variable assets that are performed on the SPL level affect all products where the respective variants are selected. On the product level, variable assets are added or removed by changing the product configuration. Changes on product-specific assets affect only individual products.

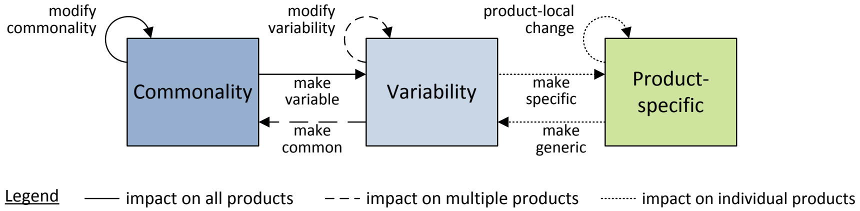


Fig. 9.5: Types of changes on assets (based on [753]).

A specific type of change in SPLs is “moving” assets between the categories common, variable, and product-specific. Figure 9.5 shows the different types of changes and their impact (based on [753]). For instance, common functionality can be made variable if it should be excluded from some products. Usually this requires changing the implementation (to make it variable) which then affects all products. Making a variable asset common influences at least those products that did not contain the asset before. Making a variable asset specific or a product-specific asset generic influences only the specific product.

Of course, changes on SPL level take only effect on an existing product if the product is re-derived, e. g. to release a new version of the product that includes the changes made on SPL level. It depends on the company strategy and the importance of a change (e. g. important bug-fixes) if and when changes on SPL level are propagated to existing products.

To summarize this section, we can say that evolving an SPL can be particularly complex as one has to consider (1) both the SPL and its products and (2) the variability of the assets.

9.4 Approaches to SPL Evolution

In this section we will give an overview of the state-of-the-art in SPL evolution. To give the reader some orientation, Figure 9.6 shows a graphical summary of the areas that we will cover.

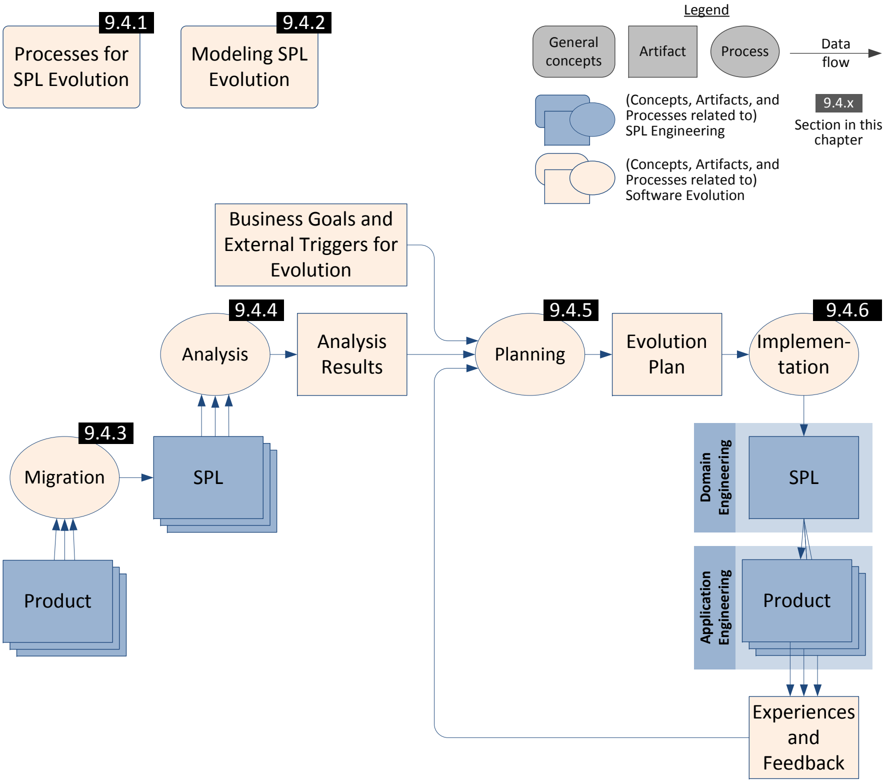


Fig. 9.6: Overview of activities and aspects in the evolution of SPLs.

First, we will address general concepts, i. e. *process models for SPL evolution* (Section 9.4.1) and *modeling techniques for SPL evolution* (Section 9.4.2). We will then roughly follow a process of evolution, as follows: An SPL is often initiated through the *migration* of existing products into an SPL (Section 9.4.3). A second step in initiating SPL evolution is the *analysis* of past evolution (Section 9.4.4), which leads to an overview of which changes happened in previous evolution steps. Subsequently, SPL evolution is performed by iterations of *planning* future evolution (Section 9.4.5), and *implementing* it (Section 9.4.6). In each iteration, the *evolution plan* is updated and the change relative to the earlier version implemented to reach the next evolution step of the SPL. Experiences and feedback are then used as input to the planning of the next iteration.

9.4.1 Process Models for SPL Evolution

In the context of SPL evolution, the literature provides various suggestions for processes and methods. These range from evolution-oriented extensions of general SPLE frameworks [85] to methods that address a particular aspect of SPL evolution, e. g. the mining of legacy assets [655].

9.4.1.1 Process framework for SPL evolution

We will now introduce a process framework for SPL evolution, which is based on the generic framework for SPLE (Figure 9.1) introduced earlier in Section 9.2. We extend and refine that to cover the specific aspects of SPL evolution (see Figure 9.7).

Just like in the basic SPLE framework, we vertically distinguish activities for the creation of the product (*domain engineering*) and the derivation of products (*application engineering*). Horizontally, we distinguish various types of artifacts, for instance, *requirements*, *features* (as a common type of variability specification), and *implementation*. On a higher level, i. e. *method engineering*, we deal with the set-up and configuration of a process and organizational structures (the method) in the other two layers.

To handle evolution, this framework includes activities taking care of adaptation and change. This begins with the initial setup through *method configuration* ❶ according to the particular context. Here we have to take into account specific *method requirements* given by the context, which influence the process structures on domain engineering and application engineering level. For instance, in domains that deal with co-design by various disciplines (e. g. mechanics, electronics, software) we might have to execute and synchronize multiple parallel design activities. Similarly, in domains with regulated software we might have to include special review activities into the process. These variations are not shown in Figure 9.7, but give examples of why an adaptation of the process might be necessary.

Once this setup has been completed, the activities of domain engineering and application engineering are performed. As their main objectives these activities aim to create the SPL (in domain engineering) and derive products from it (in application engineering). However, as side results they also initiate change and evolution: The activities of *product configuration* and *product derivation* yield information on *mismatches and suggested changes*, e. g. when the current SPL is not able to cover all *product-specific requirements*. In some cases the engineers might decide to implement *product-specific assets* to overcome these gaps between the current capabilities of the SPL and product-specific requirements.

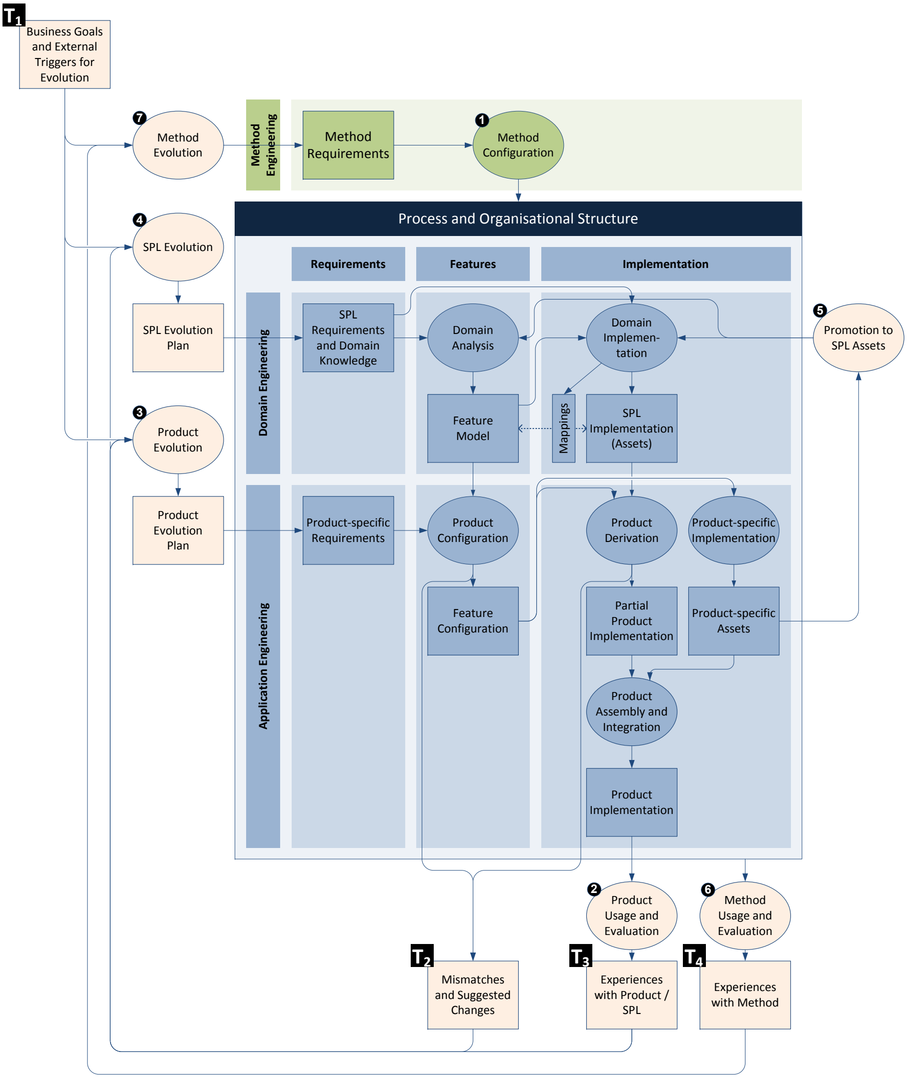


Fig. 9.7: Process framework for SPL evolution.

Eventually, application engineering yields *product implementations*. Then, *product usage and evaluation* ❷ results in *experiences with the product and the SPL* respectively. These experiences combined with mismatches and suggested changes provide input to *product evolution* ❸ as well as *SPL evolution* ❹. The latter often includes the *promotion of product-specific assets to SPL assets* ❺. The execution of domain engineering and application engineering, i. e. *method usage and evaluation* ❻ yields *experiences with the method* and can trigger *method evolution* ❼, e. g. adaptation of the process and organizational structure. For instance, it might be decided that to improve product quality additional testing activities will be introduced on application engineering level.

9.4.1.2 Evolution strategies

The process framework in Figure 9.7 shows multiple ways how SPLs evolve: evolution can take place on different levels and be caused by different triggers.

Concerning the level of evolution, there is SPL Evolution (❹ in Figure 9.7) and product evolution ❸. A specific case is when product-specific assets are promoted to SPL level ❺. (Note that this also complies with the discussion in Section 9.3 where making a product-specific asset generic corresponds to promotion to SPL level.)

Concerning triggers for evolution, there are *business goals and external triggers* T_1 for evolution, *mismatches and suggested changes* T_2 resulting from product derivation, and *experiences with the product/SPL* T_3 (leaving aside method evolution here).

Deelstra et al. [230] and Schmid et al. [754] describe several SPL evolution strategies that commonly occur in practice. They can easily be related to our process framework by classifying them according to the level of evolution and the triggers. Figure 9.8 shows a taxonomy of evolution strategies where these strategies (or situations) are classified according to its trigger and the level of evolution on which it takes place. We describe each situation in the following.

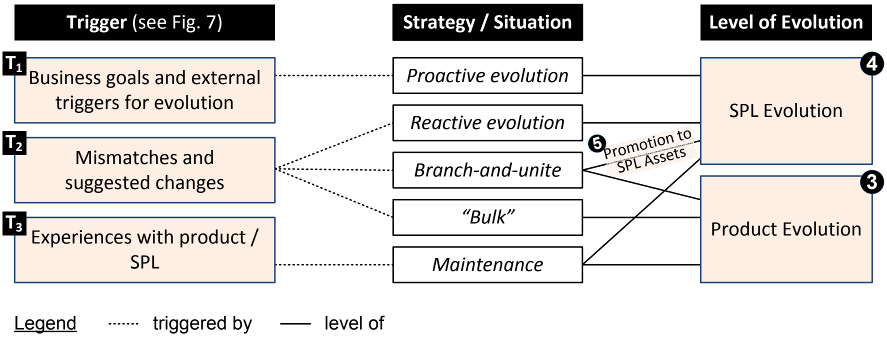


Fig. 9.8: Categorization of SPL evolution strategies with respect to trigger and level of evolution (from Figure 9.7).

Proactive evolution refers to proactively planning future requirements and adding them on the SPL level. This is a pure domain engineering activity where evolution planning is based on business goals and external triggers for evolution (such as market changes).

There are three common ways how to deal with mismatches and suggested changes that arise during product derivation:

Reactive evolution refers to integrating new requirements that arise during product derivation directly into the SPL, e. g. as variable assets. This means that reactive evolution is performed on SPL level. The advantages are the immediate possibility for reuse in future products and the avoidance of product-specific implementations or multiple branches. Highly automated approaches, like model-driven SPLs, often aim for this strategy to avoid product-specific implementations so that complete products can be derived automatically from the SPL. The disadvantages are required frequent changes on SPL level and the potential need to co-evolve already existing products. Also, creating product-specific functionality as a reusable asset can result in extra effort.

In the *branch-and-unite* approach, product-specific requirements are initially handled on product level, e. g. by creating a new product-specific branch. Later on, the product-specific branches can then be reunified with the SPL after releasing the product (promotion to SPL assets). In this way, the frequency of changes to the overall SPL can be reduced and emphasis can be put on the concrete product first. On the other hand the merge can become complex. A related concept is the grow-and-prune model which states that in large systems quick reaction to changes often requires copying and specialization (grow) and later on needs to be cleaned up by merging and refactoring (prune) [284].

The *bulk* situation occurs when an organization ends up with too many branches by evolving on product level only. This can lead to quality and maintenance problems and major effort is required to reintegrate the branches into the SPL.

Beside the strategies above that mainly deal with changing requirements, there are also other *maintenance* activities caused by experiences with the product and the SPL, like refactorings and correction of bugs that occur over time. The level of evolution then depends on whether the assets to be changed reside on SPL level or are product-specific.

9.4.1.3 Other process models

All SPLE frameworks described in the literature (e. g. [193, 688]) cover the main SPLE activities (*process and organizational structure* in the center of Figure 9.7). Some approaches extend this to address evolution on various levels (activities around *process and organizational structure*).

For instance, Bayer et al. [85] present PuLSE, a generic framework for SPLE, including the PuLSE-EM module, which covers evolution and maintenance. Based on information provided as a result of other PuLSE modules, PuLSE-EM accumulates knowledge and history information (e. g. a product configuration history and

PLA history) and restarts other modules (for scoping, domain engineering, and application engineering) with adaptations.

Similarly, the ConIPF method suggested by Hotz et al. [407] considers "mismatches" arising during product configuration and realization on the application engineering level and feeds them back into domain engineering where they are assessed and required changes are identified. These required changes are processed by an "evolution and development" activity, which leads to evolved and new assets as well as updates in configuration models.

There are several other SPLE methods that describe process structures for SPL evolution, e. g. [9, 176, 339, 809]. Further approaches which are more focused on the migration of existing groups of products towards SPLE are discussed later in Section 9.4.3.

9.4.2 Modeling Evolution and Change

A prerequisite to handle evolution in a systematic way is the ability to explicitly specify evolutionary changes. This is required during analysis of the evolution history of an SPL (to capture and specify observed changes), during planning of future evolution (to specify potential future changes and reason about them), and during implementation of evolution (to specify the changes to be realized).

On a lower abstraction level, like source code files, changes can be handled with the same tools as for single product development, like source code versioning systems. However, the higher the abstraction level (e. g. to view the evolution of an SPL as a whole), the more SPL specifics, like variability, need to be taken into account.

In earlier work in [134, 135, 686] we suggested feature models as a suitable means of abstraction to describe the overall evolution of an SPL, as features represent an SPL in a way that is meaningful to different stakeholders. Hence, evolution of an SPL is represented as a sequence of feature models over time. We will now first introduce an example and then discuss concepts for modeling evolution and change on this base.

Figure 9.9 shows a small example from the e-shop domain. It shows the four versions of the SPL's feature model at four different points in time, including historic evolution (2012) and planned future evolution steps (2014 and 2015). In this example, the version in 2012 supports only a Catalog and ShippingOptions with optional support for CarrierSelection. The version in 2013 (today) has been extended by support for Search which is available either as BasicSearch or, with extra costs, as an AdvancedSearch that supports a more intelligent search algorithm. The planned version for 2014 will distinguish between ElectronicGoods (which can be either shipped or downloaded directly) and PhysicalGoods that need to be shipped. Hence, Shipping-Options has become an optional feature and a cross-tree constraint has been added. In this version, AdvancedSearch will not be supported as it requires additional time to integrate it with the changes on Catalog. For 2015 it is planned to support an

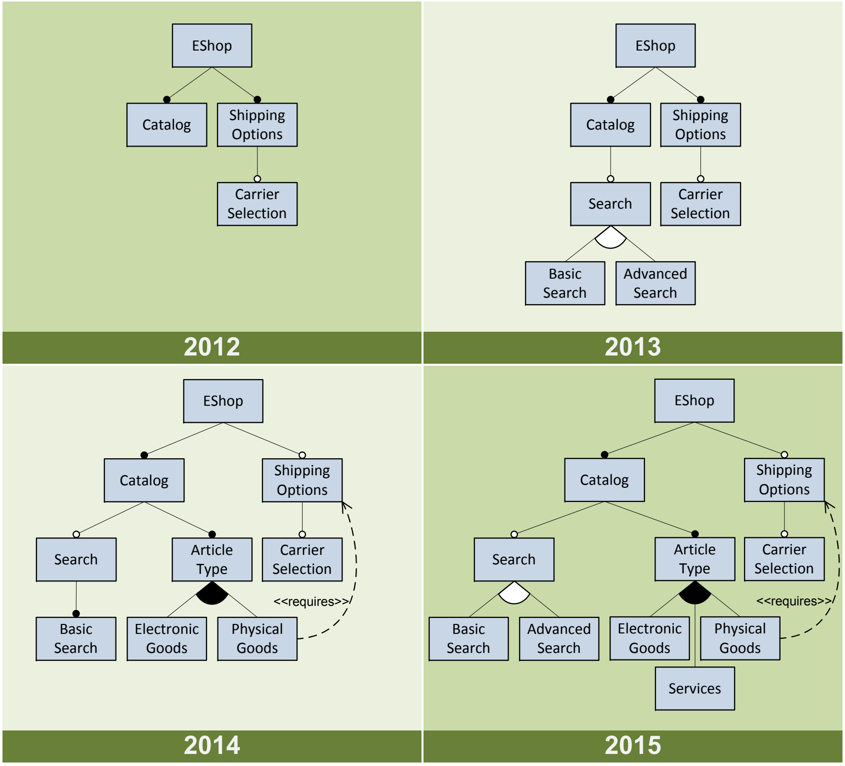


Fig. 9.9: Evolution of an SPL as a sequence of feature models

additional article type Services and to support AdvancedSearch again for all article types.

The remainder of this section describes how to model the changes between different versions of an artifact using the example of feature models above. Analogous to other areas like metamodel evolution (Chapter 2), there are two basic ways to specify such changes: 1) by modeling the *differences* between them (Section 9.4.2.1) or 2) by describing the performed modifications in terms of *change operators* (Section 9.4.2.2). Finally, Section 9.4.2.3 provides a more detailed example using a *combined approach.*

9.4.2.1 Difference Models

Approaches which are specifying the *differences* between versions work similar to approaches for program differencing [470] or common source code versioning systems that determine differences between versions of text-based files based on heuristics on the level of lines or characters. On the level of models, a *difference model* can be used that contains the changes between two versions in terms of added, removed, and modified elements.

Figure 9.10 shows an example for the evolution step from 2013 to 2014 in our e-shop example: the xor-group and its child feature AdvancedSearch has been removed. The features ArticleType, ElectronicGoods, and PhysicalGoods and their relationships and constraints have been added. In addition, ShippingOptions has been modified to become an optional feature. Context elements (represented by light color in Figure 9.10) are used to specify the locations in the model, e. g. where to add new elements.

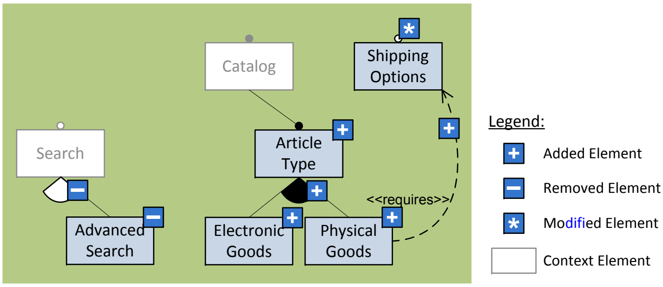


Fig. 9.10: Difference model for the evolution step from 2013 to 2014.

Several approaches have applied such concepts in context of SPLs: Acher et al. [5] provide a formal approach to identify the syntactic and the semantic difference between two feature models. Schäfer et al. [749] define a concept of difference models (called *delta models*) and apply it e. g. to specify multiple products in terms of differences to a core product. Hendrickson et al. [390] use difference models (called *change sets*) and relationships between them to specify the architecture of different products by combinations of change sets.

At this point, an important observation can be made: Specifying changes is not only relevant in context of evolution but also in context of variability, e. g. to specify the differences between multiple product variants in an SPL. In context of SPLE the latter is called *variability in space* while evolution can be considered as *variability in time*. Hence, it is not only possible to apply change modeling concepts to describe variability in an SPL (like Schäfer et al. and Hendrickson et al. mentioned above) but also to apply variability modeling concepts to specify evolution. An approach that makes use of this idea is EvoPL described later in Section 9.4.2.3.

9.4.2.2 Change Operators

The second basic concept to specify changes are *change operators*. A change operator describes an operation performed on a model to achieve a change. There are three atomic change operators *add*, *delete*, and *modify* that have the same semantics as the elements in difference models. However, the main difference is the possibility of more complex operators that allow to express richer semantics about a change like "*split feature f into f_1 and f_2*".

Semantically rich operators are usually defined for a specific modeling concept (e. g. feature models or metamodels, see Chapter 2) and can also be optimized for a specific purpose. For instance, in context of SPL refactoring Alves et al. [25] define a set of change operators on feature models that do not change the feature model's semantics (e. g. "*convert or to optional*" or "*push up node*"). Seidl et al. [761] define change operators in context of implementing SPL evolution which are discussed later in Section 9.4.6.

9.4.2.3 Combined Approach

An approach that combines concepts of difference models and change operators to model long-term evolution of an SPL is *EvoPL* [686]. It also leverages the idea of considering evolution as *variability in time* introduced above.

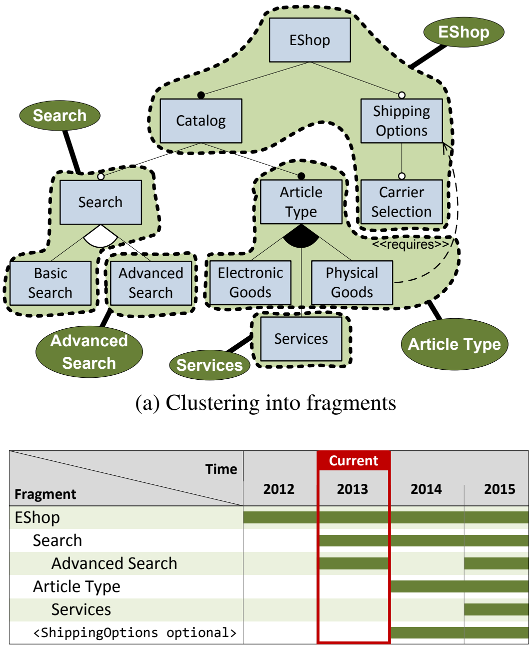
EvoPL focuses on feature models as main artifact to manage SPL evolution. The approach is intended to be used for both, analyzing past evolution (see Section 9.4.4) and planning future evolution (see Section 9.4.5).

In EvoPL, each feature model version is composed of model *fragments*. Figure 9.11a shows the fragments for the evolution in Figure 9.9. A fragment clusters related feature model elements that are added or removed only together during the same evolution step. The purpose of fragments is to raise the level of abstraction by representing multiple related elements. Each fragment has a unique name and is stored together with a context element (the parent feature) specifying its location within the overall feature model. In this way, each feature model at a certain point in time can be described as a composition of fragments.

Changes within fragments, e. g. changing a feature from mandatory to optional or adding a cross-tree constraint, are specified by change operators (called *evoOperators*, see [686] for details) associated with the fragments. For instance, ShippingOptions are changed from mandatory to optional in 2014 which is defined by a change operator <ShippingOptions optional> that is applied to the versions for 2014 and 2015.

The overall evolution is then specified using the concept of "variability in time": The fragments and evoOperators themselves are stored in a specific kind of feature model (called *EvoFM*) that specifies their hierarchy and other dependencies between them. Each evolution step can then be represented by a "configuration" of the EvoFM, i. e. a selection of fragments and evoOperators that together make up a feature model. The evolution of a feature model can, hence, be represented by a sequence of EvoFM configurations.

We visualize this by a representation that we call *evolution plan* (Figure 9.11b). The horizontal dimension represents the time line; each column represents an evolution step. The vertical dimension represents the EvoFM; each row represents an EvoFM element, i. e. a fragment or an evoOperator (the latter denoted in angle



(a) Clustering into fragments

Fragment \ Time	2012	Current 2013	2014	2015
EShop	🟩	🟩	🟩	🟩
Search		🟩	🟩	🟩
Advanced Search		🟩		🟩
Article Type			🟩	🟩
Services				🟩
<ShippingOptions optional>			🟩	🟩

(b) Evolution plan

Fig. 9.11: Fragments and resulting evolution Plan

brackets). Each cell in the plan represents a configuration decision, i. e. whether the fragment or evoOperator is selected (i. e. applied) in that version or not.

For instance, the evolution plan in Figure 9.11b represents the evolution steps from Figure 9.9: In 2012, only the fragment EShop is selected (applied). In 2013, the fragments EShop, Search and AdvancedSearch are applied. In 2014, AdvancedSearch is no longer applied while ArticleType and the change operator <ShippingOptions optional> are applied. In 2015 all fragments and change operators are applied.

Figure 9.12 shows the overall workflow with EvoPL: A model transformation enables to automatically extract an evolution plan from a given sequence of feature models. The evolution plan is then used to plan future evolution by adding new evoConfigurations (and, if necessary, new fragments and evoOperators). Please note that fragments are never modified (as evoOperators are used instead) except for splitting fragments which can become necessary if in a future evolution step a subset of a fragment should be removed. Once planning of future versions has been finished, another model transformation supports automated composition of the resulting feature models. Due to the incremental nature of the model transformations it is possible at any time to update the evolution plan to include changes on feature model level and, in turn, to re-generate feature models after the evolution plan has been modified.

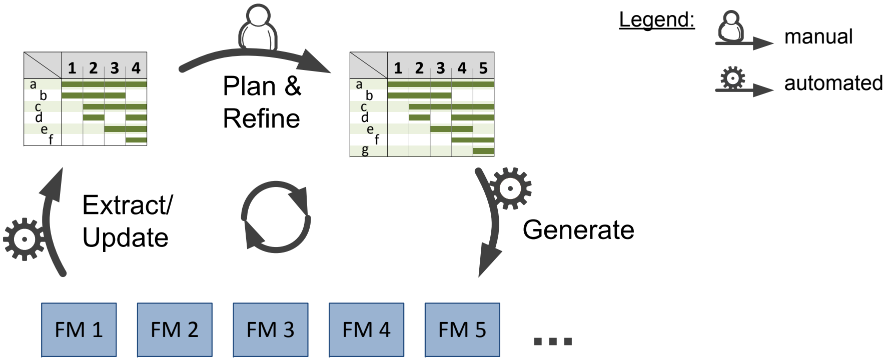


Fig. 9.12: Workflow and transformations on feature model level with EvoPL

An advantage of the evolution plan representation is its degree of abstraction. As demonstrated in [686], abstraction into fragments can significantly reduce complexity when dealing with large evolving feature models while the evolution plan provides a comprehensive overview of the different versions. Another advantage of the approach is its support for order-independent planning. Changes are not specified relative to the previous version (or a common baseline) but by selecting or eliminating fragments and change operators. This enables incremental planning of multiple versions in parallel or specifying a later version before its predecessors have been fully defined.

9.4.3 Migration to SPLE

In practice, the introduction of SPLE often arises when after some success in a market segment a company finds itself with a family of products. Hence, when discussing the *adoption* of SPLE, besides starting SPLE from scratch we have to consider approaches which evolve SPLs from legacy products and focus on *migration* and *mining of existing assets*. We can distinguish four types of SPLE adoption [228, 754]:

- *Independent* - A new SPL is created independently of any existing products.
- *Leveraged* - A new SPL is set up based on an existing one.
- *Project-integrating* - With an existing product base as background, a set of projects (developing new products) is selected to contribute to an SPL.
- *Reengineering* - From existing legacy products, assets are extracted and reengineered to contribute to a new SPL.

In addition, we can distinguish between *revolutionary* ("big bang") and *evolutionary* (incremental) models [130, 754]. This is somewhat orthogonal to the four adoption types, however, adoption types that take existing systems into account (*project-integrating*, *reengineering*) are most amenable for an evolutionary approach.

In the literature there are numerous approaches to SPL migration, e. g. [86, 140, 228, 284, 472, 778]. In the remainder of this section we will describe various aspects and activities of such migration approaches, i. e. *initiation of a migration project* (Section 9.4.3.1), *scoping* (Section 9.4.3.1), *variability analysis* (Section 9.4.3.3), *refactoring* (Section 9.4.3.4), *extraction of assets* (Section 9.4.3.5), and *assessment* (Section 9.4.3.6). While conceptually such aspects can be interpreted as a logical sequence of activities, in practice they are often performed in an iterative fashion (see, e. g. [86]). For instance, an initial analysis of variability among existing products might lead to a preliminary selection, which is followed by a more detailed variability analysis.

9.4.3.1 Initiation of the migration project

At the beginning of a migration project the relevant base information needs to be collected. This might include, e. g. information on product capabilities, evolution so far, existing software architectures, documents about scope and existing assets, and preliminary estimates of required changes (interface vs. deeper changes) [93, 785]. Relevant information can be extracted from artifacts or gathered by interviewing product experts, maintainers, and users [785].

Then, based on a first assessment an approach for the mining of assets can be drafted. It needs to be decided on which abstraction levels (e. g. features, components) the mining will happen and whether further processing (e. g. refactoring) is necessary.

A migration project must also consider business and organizational aspects. First, the advantages and potential drawbacks of the various options (current situation vs. introducing SPLE) need to be considered, e. g. with an estimation of costs and productivity benefits. Second, various organizational structures are possible. For instance, SPLE can be performed in product teams or in a separate dedicated SPL team [112].

9.4.3.2 Scoping and assessment of migration options

Similarly to general SPLE approaches [444, 785], in SPL migration the scope of the overall effort needs to be defined, i. e. deciding about which features to include in the SPL and which are out of scope. Since this might require additional input (e. g. a prioritization of features) the scoping might have to be performed after or in combination with other activities (e. g. after an initial variability analysis).

In scoping we can take a problem-oriented perspective (“What does the customer value most?”), but we also need to consider solution-oriented aspects (“What can we implement most easily?”) [778].

9.4.3.3 Variability analysis

The migration of existing products into an SPL often starts with an analysis of their commonalities and variabilities. This can be performed on various abstraction levels, e. g. (1) on the level of requirements, customer visible functionality and features or (2) on the level of implementation artifacts.

On higher abstraction levels we can apply techniques for *commonality and variability analysis* [688], e. g. an application-requirements matrix (table of products vs. requirements), priority-based techniques (requirements are rated by different stakeholders) or checklist-based analysis (collecting and analyzing requirements with the help of various checklists).

On more concrete levels, we can analyze implementation artifacts to *extract variability models*. When reverse engineering higher level variability models, we might have to apply heuristics and involve experts to (re-) construct their structure [773] (cf. Section 9.4.4.2).

Just like in general SPLE approaches, in a migration project we have to anticipate future changes. This includes product and feature planning, the anticipation of future features [778] and the analysis of consequences for the PLA and implementation assets.

9.4.3.4 Refactoring

Before the actual extraction of SPL assets is performed, it is often necessary to refactor existing artifacts [472], e. g. to remove accidental differences and increase commonalities. This can occur on various abstraction levels, e. g. when refactoring code [532, 857] or when restructuring the software architectures of existing products to prepare them for merging them into a shared PLA [228].

We have to distinguish such *preparative* and *transformative* refactoring (preparing assets for a migration, transforming them into SPL assets) from refactoring of the SPL after it has been established, e. g. on the conceptual and feature-model level [25, 128] or of PLAs [204].

Related techniques are feature-oriented restructurings, which are not predominantly aimed *towards* an SPL, but rather use feature-orientation as guiding concepts, e. g. with the help of regression tests [583] or when aiming to untangle and separate concerns on the implementation level [624].

9.4.3.5 Extraction of assets

One of the key artifacts when establishing SPL practices is a PLA. While in other scenarios it might be appropriate to design a PLA from scratch, we have to take a different approach when migrating existing products into an SPL. Here, techniques for architecture and component recovery can be applied, e. g. *Option Analysis for Reengineering* (*OAR*) [94] and *Mining Architectures for Product Lines*

(*MAP*) [655, 796]. In many cases, the approach will be to extract product-specific architectures [228] and then analyze and merge them, e. g. using techniques for model merging [174, 735].

In that course, mechanisms for variability realization [810] have to be selected and, based on earlier variability analysis, variation points have to be chosen.

Alongside the extraction of a PLA, the corresponding core asset implementations need to be extracted and refined [93, 94]. Here, we have to address the mapping of variability models onto implementation artifacts (*feature location*) [255], the identification of similar implementation artifacts (*clone detection*) [475, 587] and the analysis of feature implementations with respect to dependencies as well as interactions (*dependency analysis*, *feature interaction*) [42]. Dependencies that are detected on the implementation level need to be propagated up to higher abstraction levels (PLA, variability/feature model).

During a migration to SPLE, often SPLE activities and reverse engineering activities are performed side by side. For instance, Bayer et al. [86] suggest to integrate the reverse engineering of existing assets and the creation of SPL models via a "blackboard", i. e. a shared workspace allowing reengineering and SPL activities to exchange and incrementally enrich information.

9.4.3.6 Assessment

After the SPL infrastructure has been established, the created artifacts, in particular the PLA, should be evaluated [130]. Here, architecture evaluation methods [257] and analysis of selected product instances [130] can be applied. Of particular interest in the context of this chapter is the assessment of the PLA with respect to is evolvability/maintainability. Since the context (requirements, business goals) and the PLA will change over time, reevaluations should be performed [785].

The result of such a migration project provides input for subsequent activities. For instance, migration can provide a first draft of an evolution plan for the near future, based on features that do not exist yet but are anticipated.

9.4.4 Analyzing Evolution

This section discusses the analysis of the current status and the evolution history of an SPL as a basis for planning (Section 9.4.5) or to predict future evolution. We first briefly discuss repository mining techniques, then analysis approaches on the feature and architectural levels, and finally prediction of maintenance effort and evolution via simulation.

9.4.4.1 Mining software repositories

Approaches for *Mining of Software Repositories* (*MSR*) [216, 451] (cf. Chapter 5) process various data sources, e. g. source code repositories, bug databases, and mailing lists. Often different input sources are combined to gain better results. Approaches aim to uncover relationships and trends, e. g. using data mining techniques. Examples of extracted information are the growth of a system, change relationships between assets, or the reuse degree of components. Kagdi et al. [451] categorize MSR approaches into two types. Some approaches answer *market-based questions*, i. e. "If A occurs then what else occurs on a regular basis?" (resulting, e. g. in association rules). Other approaches answer *prevalence questions*, e. g. the number of functions reused or if a particular function was changed. Orthogonal to that, Kagdi et al. distinguish between approaches measuring *changes to properties*, i. e. calculating metrics for each version and then comparing over different versions, and approaches focusing on *changes to artifacts*. There is a large spectrum across levels of abstraction (e. g. features, architecture, source code) and granularity addressed by such approaches. SPL-specific issues, like variability or distinction between SPL and products, have received little attention to date.

9.4.4.2 Analyzing features

When analyzing the evolution of an SPL, feature-oriented analyses are of obvious interest. Various work on *feature location* [255] aims to establish traceability between features and assets that implement them (see Section 9.4.3.5). In an existing (model-driven) SPL, such traceability might already exist explicitly. However, in less structured SPLs, e. g. with many product-specific implementation parts, feature location can be essential for refactoring and migration (see Section 9.4.3).

Other approaches aim to reverse engineer the feature model, starting, e.g., from a set of unstructured features [773], the architecture [4], or even informal product descriptions [219]. One can expect that an automatically extracted feature model differs from one that is manually crafted by a human software architect; however there is not sufficient empirical data on this yet. To close this gap, Hsi and Pots [409] suggest to extract features from an application's user interface and to link them to code assets, e. g. via the operations called by user interface actions.

9.4.4.3 Architecture assessment

Most existing SPL-specific approaches for analysis of evolution address the assessment of the PLA. This can be useful both during creation of an SPL (see Section 9.4.3.6) as well as on existing SPLs. Maccari [549] applies the *Architecture Trade-off Analysis Method* (*ATAM*) [257] to assess the suitability of the PLA for future requirements.

Johnsson and Bosch [445] aim to quantify SPL aging. They measure average costs per maintenance task as well as the relative distribution of effort among adding components, adding functionality to components, and changes to existing functionality. They argue this can be used to detect architecture erosion and the related increase in maintenance effort. This can in turn be used to decide on the reorganization or retirement of SPLs.

9.4.4.4 Prediction based on simulation

Heider et al. [381] propose to simulate SPL evolution to predict the long-term development of maintenance effort and model complexity. The analysis is performed on a problem space model (i. e. decision or feature model), a solution space model, and dependencies within and between them. The simulation modifies the models with random operations based on probabilities defined in profiles. For instance, the "evolution profile" describes the type of evolution to be performed, like "continuous evolution", "refactoring", or "product placement" (i. e. changing mostly the problem space while keeping the solution space mostly unchanged). The evolution profile can be created based on existing evolution history.

9.4.5 Planning Evolution

This section deals with planning of evolution, i. e. how to decide about changes to the SPL to be implemented in the upcoming versions.

Usually important planning decisions on the evolution of complex software systems require careful consideration. Ad-hoc planning would bear the risk of deficiencies like insufficient anticipation of future requirements, lack of resources to realize new requirements, or loss of knowledge about previous decisions [736]. In single system engineering there are several research strands – like rationale management and release planning – that aim to reduce this risk by supporting systematic planning and decision-making. These concepts can be applied to SPLE.

An important prerequisite for deciding on future changes is to gain knowledge on the impact of a potential change. However, for an SPL this can be much more complex than in single system engineering due to the complexity of interdependencies between artifacts (see Section 9.3).

In this section we first discuss change impact analysis in SPLs and then present approach for decision-making in SPL evolution.

9.4.5.1 Change Impact Analysis

When deciding about a change we have to predict the required effort and potential pitfalls for its realization. This is supported by approaches for impact analysis [124, 173].

An important aspect of impact analysis is traceability, i. e. storing links between all logically related assets in the software development process to understand what other assets might be affected if an asset changes. An example are traces between a requirement and its implementation assets. In context of SPLs, the mapping between features and implementation assets (if fully specified) can be considered as a kind of trace link. However, traceability approaches consider additional types of links and often add some extra information to each link (like a rationale description).

For instance, Anquetil et al. [34] propose a traceability framework for SPLs. They propose four general categories of trace links: *Refinement* traceability relates artifacts from different level of abstraction like an element in the design model and its implementation. *Similarity* traceability relates artifacts at the same level of abstraction such as similar requirements that have some logical relationships or similar elements from different architectural views. *Variability* traceability relates artifacts as relevant for variability management like the mapping between a feature and its implementation. *Versioning* traceability relates successive versions of an artifact. As pointed out by Heider et al. [386], traceability needs to cover not only all assets on SPL level but also on product level. Other traceability approaches for SPLs can be found e. g. in [8, 442, 616].

Defining (and maintaining) trace links requires much effort, so there is a need for tool support. There are two ways how to acquire trace links in a tool-supported way: 1) ex-post by statically analyzing existing artifacts or 2) during development when artifacts are created. Heider et al. present *EvoKing* [383], an IDE for SPLs which supports both strategies. EvoKing supports monitoring evolution by keeping track of all assets and their relationships within an SPL. To provide some degree of abstraction, users can define the types of assets and relationships and how the tool interprets events like creating or modifying an asset of a certain type. Trace links are established according to user-defined rules. For instance, whenever a product configuration is created, a trace link is established to the underlying variability model. These rules can also be applied to existing artifacts. However, heuristics or statistical analysis to acquire trace links automatically have not been applied yet. Other approaches that aim to provide automated extraction of traceability links for SPLs are e. g. [442, 747].

Beside traceability approaches, there are only few other approaches for impact analysis in context of SPLs. Heider et al. [385] present an approach using regression testing to analyze the impact of changes on SPL level on products. Whenever the SPL is changed, the tool first analyses whether the existing product configurations need to be changed as well, e. g. whether configuration decisions need to be modified due to changes on the variability model. In a second step, the tool re-derives all products and compares them with their previous version and reports the differences.

In this way it provides instant feedback to developers about the consequences of a change on the SPL.

9.4.5.2 Decision making in SPL evolution

Planning evolution means to make decisions that may have essential impact on the future success. Concepts like the *QOC* approach (*Questions, Options and Criteria*) [550] provide general support for systematic decision-making. The first step in QOC is to define the issue on which to decide (*question*). Second, available solution *options* are identified and specified. In addition, *criteria* are defined by which the available options can be rated. Examples for criteria are the expected development effort (e. g. estimated by an impact analysis as above), market value, strategic benefit or risk. Each solution option is rated according to these criteria. Finally, an option is selected on this base. This allows systematic decision making and captures the reasons behind a decision.

Approaches in the area of *release planning* [736] apply such concepts to decide about new requirements (or features). For instance, when a set of new candidate requirements is given (e. g. due to customer requests and market analysis) they support to prioritize them and to select those to be implemented in the next release(s). Similar to QOC, criteria have to be defined by which the requirements can be rated. Usually ratings are performed by multiple stakeholders including e. g. prime customers. Criteria and stakeholders can be prioritized by assigning weights. Moreover, constraints can be defined to specify preconditions such as available resources (e. g. person months until next release) and dependencies between the candidate requirements (e. g. two requirements exclude each other). After each requirement has been rated according to the criteria, approaches like EVOLVE [643] automatically propose a candidate release plan that conforms to the defined constraints.

Similar concepts have been applied to SPLs for scoping (see Section 9.4.3.2), i. e. selecting which features from an existing set of related to include in an SPL. The PuLSE-Eco approach by DeBaud and Schmid [229] proposes to refine business goals into “benefit functions” (e. g. effort saved by making a feature reusable) which are decomposed into basic “characterization functions” (e. g. implementation effort in person months) by which each potential feature is judged. In this way the benefit of each feature is calculated as a base for the decision which features to include into an SPL.

Besides deciding about new features or requirements, which is similar to single system engineering, SPL evolution also needs SPL-specific decisions like 1) deciding about whether changes should become product-specific or be performed on SPL level and 2) about the variability of features on SPL level. In the following we describe an approach for each of these issues.

Heider et al. [382] address decision making about whether new requirements that arise on product level should be promoted to SPL level. Their tool EvoKing (see Section 9.4.5.1) provides SPL engineers an overview on new requirements that have arisen on product level. SPL engineers can then decide about each requirement to either lift it to SPL level or to assign it to developers on product level otherwise. In [380] the authors describe how this decision is supported by a Win-Win model negotiation approach. *Win-Win* [120] is a general approach similar like QOC but with a focus on brainstorming and negotiation: Different stakeholders define their objectives as *win conditions*. Win conditions where all stakeholders agree on are stored as *agreements*. Otherwise conflicts, risks or uncertainties are defined as *issues*. Stakeholders then brainstorm for *options* to resolve these issues and to explore trade-offs with the goal to find an option that can be turned into an agreement. In context of new SPL requirements, the win conditions are the new requirements proposed by SPL engineers and product engineers. Issues raise, for instance, when there are inconsistencies between the requirements on these two levels or between requirements of different products.

Thurimella and Brügge [843] address decision-making about the variability in SPLs. They apply similar concepts like in QOC. To decide about variability, the possible solution options are identified (e. g. whether a feature is mandatory or optional) and rated by criteria. The same principle is also applied to product configuration decisions where available variants can be considered as options.

Basically, concepts like QOC or Win-Win can be applied to any particular evolution decision [842]. Besides the concrete approach used, any additional tacit knowledge underlying a decision (e. g. why an option was finally selected) should be explicitly documented by a *rationale description* to preserve the knowledge for future decision-making [824]. For instance, the EvoPL approach from Section 9.4.2.3 has been extended by support for decision-making by modeling high level goals, criteria, rationale, and the relationships between them [758].

As discussed in [759], it should be considered that planning information – such as goals, criteria, and rationale – evolves itself (e. g. changing business goals). It is useful to handle this evolution in a structured way as well (e. g. traceability of previous versions of a goal description) to preserve the information and understand previous decisions.

9.4.6 Implementing Evolution

Existing work on the implementation of SPL evolution aims to support realizing changes in a systematic way. It can be classified according to the abstraction layers in Figure 9.4. On the SPL level, changes to requirements lead to changes in the variability model. Several works aim to support changes to variability models and the associated mappings while preventing inconsistencies (Section 9.4.6.1). Other work focuses on realizing changes on the implementation level, e. g. by structuring the implementation according to features (Section 9.4.6.2). When an SPL has changed, this has to be propagated to existing products (Section 9.4.6.3).

9.4.6.1 Evolution of the variability model and its mappings to assets

Changes to requirements often lead to changes in the variability model, e. g. adding or removing features or splitting a feature to make some part of it variable. Also, the variability model has to be maintained itself, e. g. by restructuring to improve readability. Changes to the variability model can be (tool-) supported by change operators (cf. Section 9.4.2) to systematize changes and to update the mappings to implementation assets.

Thüm et al. [837] provide a tool that analyses changes performed on a feature model and classifies them into one of the four categories: (1) *refactoring*, not changing the set of valid products, (2) *generalization*, only adding products, (3) *specialization*, reducing the set of products, or (4) *arbitrary changes* otherwise:

Refactoring, as used by Thüm et al. [837], refers to changes of the feature model that do not change its semantics in terms of valid product configurations. The source and target feature model are then referred to as "equivalent" [803]. Such changes include, for instance, restructuring of the feature model, and changes that do not influence product configurations (e. g. renaming a feature).

Generalization refers to changes that only add products (i. e. valid configurations) to the SPL. All existing products remain valid and can still be derived from the SPL. Simple examples for such changes are adding a new optional variant or changing a feature from mandatory to optional. A catalog of feature model operations that preserve the set of products is presented in [25]. In contrast to [837], the authors call these types of changes "refactorings" or "refinements". Based on a formal notation for refinements introduced by Borba et al. [129], Neves et al. [642] specify several complex change operations for common behavior-preserving changes of SPLs which include not only the feature model itself but also the mappings and the associated assets. For instance, a new mandatory feature can be safely added to a feature model only if it represents functionality that is already part of all products (e. g. to convert it into an optional feature later on).

Specialization is mainly used during staged configuration [209], i. e. configuration performed in multiple steps, e. g. by various stakeholders. Variability is reduced in each step until it is completely resolved and exactly one product remains.

A tool that aims to support *arbitrary changes* is Feature Mapper [762]. It focuses on the co-evolution of feature models, implementation assets, and the mappings between them. On the level of feature models it supports several change operators, like "add feature", "split feature", "remove feature", or "remove feature and owned asset". On the level of assets (also represented as a model) changes depend on the concrete type of model. The authors provide some examples for UML models (e. g. "replace method with method object") and for models representing Java code (e. g. "extract method"). The authors classify three types of changes (focusing on consistency of feature mappings): (1) changes that only have effects within one model (like changing an optional feature to mandatory or renaming an asset); (2) changes that affect a model and the mapping (like "split feature" or "extract method"); and (3) changes that affect the model, the mapping, and the mapped model (like "remove feature and owned assets"). In the second and third case, the tool automatically updates the mappings to keep them consistent.

Beside the work on feature models, there are also approaches addressing other types of variability models, e. g. decision models and their associated assets [385].

The change operators and tools described above cover only a subset of possible changes to an SPL. Other changes that require manual implementation (like adding new functionality to an SPL) cannot be specified just in terms of predefined operators. However, tool support should indicate potential inconsistencies after a change. In general, this can be achieved by analyzing mappings between assets (similarly to Feature Mapper as described above) or additional traceability links between dependent assets (see Section 9.4.5.1).

Finally, after performing changes, the consistency should be checked between the different abstraction levels in the SPL as shown by Vierhauser et al. [892]. Guo et al. [355] show how to check the consistency of large feature models so that only those parts which are affected by an evolutionary change need to be checked again. In the context of formal validation of SPLs, Cordy et al. [199] provide a model-checking approach that supports evolution. They define a method to identify specific types of features and show that for such features, when added to an evolving SPL, only a subset of the products need to be model-checked again.

9.4.6.2 Evolution of assets

Work supporting changes on lower levels of abstraction mainly addresses mechanisms to increase modularity and maintainability of assets. Garg et al. [313] provide specific tool support to specify changes or multiple variants for SPL architectures. The presented tool *Ménage* represents visual architecture models in terms of components and connectors based on xADL 2.0. Variability and different versions are visually highlighted.

Other work addresses evolution on the code level. Here, one challenge is to modularize code so that changes on higher abstraction levels, e. g. new features, can be implemented with as few side effects as possible. For this, techniques similar to those for implementing variability in code can be used. For instance, aspect-oriented development can be used to implement cross-cutting features [902]. Loghran et al. [541] propose supporting evolution by a combination of aspect-oriented techniques and frames, which are hierarchically ordered code templates. They provide code examples showing how these "framed aspects" can be used to support reuse and the easier integration of new features.

Some work addresses evolution at runtime [339] (cf. Chapter 7.6). For this, software reconfiguration patterns are used that allow the configuration in component-based systems to be updated during runtime. The authors describe multiple reconfiguration patterns based on existing architectural patterns, e. g. "master/slave reconfiguration" or "centralized control reconfiguration", and discuss how to perform evolutionary changes based on them.

After changing the implementation, the SPL has to be tested. Here, the testing strategy should take variability into account to avoid that all possible feature combinations and all existing products have to be tested again [256].

9.4.6.3 Propagating changes from the SPL to products

Heider et al. [384] provide tool support to propagate changes from the SPL to individual products. In theory, model-driven SPLs allow products to be regenerated after the SPL has changed. However, as Heider et al. point out, in practice product configuration can be a complex and time-consuming process which requires decisions by multiple stakeholders. Hence, configuration of different products and evolution of the SPL is often performed in parallel. After a variability model has changed, product configurations must be updated by considering the dependencies in the variability model, between the assets, and between variants and assets. Hence, updating the product configuration can be challenging. The authors address this with a tool that supports automated updates of products, resolves conflicts, and assists users in manually resolving conflicts based on trace data when automated update fails.

9.5 Conclusions

In this chapter, we provided an overview of basic concepts and state-of-the-art in SPL evolution, as well as a short introduction to our own work in feature-oriented software evolution. Many challenges remain.

We need to improve support for handling changes, this includes *understanding consequences of potential changes* (taking all dependencies into account, across abstraction layers) and better support for the *propagation of changes*, for instance techniques that tolerate inconsistencies and resolve them, while propagating the changes.

With increasing scale and complexity it becomes infeasible to adapt the whole system (i. e. the whole SPL) to change in a short time frame. Hence, when an organization aims to react to market events or urgent customer requests, we require *strategies and techniques to support fast adaptation*, e. g. with product-specific extensions, which are later propagated into the SPL. Here, we have to consider an oscillation between adaptation/extensions and creating a consolidated shared infrastructure (“grow-and-prune model”, [284]).

Tracing concepts in the problem space to the solution space, from high abstraction levels to details of the software design and lines of code, is a fundamental problem in software engineering. Such mappings are often not one-to-one and ambiguous. For instance, in SPLE and SPL evolution we have to map features to their implementations, a problem addressed by *feature location*. Here, (1) a feature is potentially implemented by multiple classes, a class potentially contributes to multiple features and (2) it is not a clear-cut decision whether a class is part of a feature's implementation.

Finally, we have to deal with *evolution for PLE "in-the-large*", for instance in hierarchical SPLs or a systems-of-systems context, which requires propagation of changes up or down the systems hierarchy. When dealing with evolution of large SPLs in all its aspects (migration, analysis, planning, implementation, etc.) we need to consider the potential hierarchical structure of such systems. For instance, Gall et al. [308] report that the characteristics of the evolution of a particular subsystem deviated substantially from that of the main system, an effect that can be masked when we would only consider the system as a whole.

As a final thought, we concur with Dhungana et al. [241] who argue that SPLE should treat evolution as a normal case and not as the exception. Hence, improved concepts and techniques are required that are able to handle evolution of large software-intensive systems while taking the particular characteristics of SPLs into account. We believe that there lies great potential in a smart combination of automated and interactive techniques, which combine the best of both worlds–efficiency through automated mechanisms and guidance towards creative solutions through the capabilities of the human engineer.

Chapter 10
Studying Evolving Software Ecosystems based on Ecological Models

Tom Mens, Maëlick Claes, Philippe Grosjean and Alexander Serebrenik

Summary. Research on software evolution is very active, but evolutionary principles, models and theories that properly explain why and how software systems evolve over time are still lacking. Similarly, more empirical research is needed to understand how different software projects co-exist and co-evolve, and how contributors collaborate within their encompassing software ecosystem.

In this chapter, we explore the differences and analogies between natural ecosystems and biological evolution on the one hand, and software ecosystems and software evolution on the other hand. The aim is to learn from research in ecology to advance the understanding of evolving software ecosystems. Ultimately, we wish to use such knowledge to derive diagnostic tools aiming to predict survival of software projects within their ecosystem, to analyse and optimise the fitness of software projects in their environment, and to help software project communities in managing their projects better.

This work has been partially supported by F.R.S-F.N.R.S. research grant BSS-2012/V 6/5/015 and ARC research project AUWB-12/17-UMONS-3, "Ecological Studies of Open Source Software Ecosystems" financed by the Ministère de la Communauté française - Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique, Belgium.

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_10, © Springer-Verlag Berlin Heidelberg 2014

10.1 Introduction

Mathematics and computer science have been very helpful to advance research in biology, even so much that it has spawned a research field of its own: bioinformatics [512]. In the other direction, inspiration from biology has lead to numerous new achievements and improvements in computer science, such as neural networks [377], genetic algorithms [338, 611], optimization and artificial intelligence algorithms inspired by ant colonies and swarms of bees [126]. As explained in Chapter 4, some of these techniques have found their use in the context of search-based software engineering.

More specifically, *ecology* has been a fruitful source of inspiration for software engineering research. Huberman and Hogg considered a distributed computing system of concurrent agents as a *computational ecosystem*, analogous to biological ecosystems [411]. They studied the dynamics and chaotic behavior of such computational systems and showed how reward mechanisms may stabilize the system, thereby optimizing its performance. Calzolari et al. adapted the ecological *predator-prey* model to empirically study and predict the relation between software defects (prey) and programmers (predators) [155]. Lawrance et al. leveraged predator-prey relationships to apply information foraging theory to software maintenance, by considering developers as predators and the information they seek as prey [500]. Posnett et al. studied the risk of using aggregation techniques in empirical software engineering through its relation to the notions of ecological inference and ecological fallacy from sociology and epidemiology [699]. More recently, they also compared the developer-artifact contribution network to a predator-prey relationship, leading to a conceptually unified view of measuring focus and ownership [698].

In this chapter, we explore similar analogies with *software ecosystems* and *software evolution*. Several researchers have advocated biological evolution and ecological principles as a source of inspiration for software evolution [79, 638, 811, 940] but, until now, this has remained mostly at the level of the intention. Although research on software ecosystems is emerging, the application of ideas transposed from ecosystems in nature seems to be underexploited. The transfer of knowledge has essentially limited itself to a reuse of terms.

Despite the fact that natural ecosystems have been studied for many decades, and that many evolutionary theories and ecological models have been proposed and experimentally validated, little research exists that tries to adopt or adapt such theories to the domain of evolving software ecosystems. Although it is true that biological species, systems and ecosystems are quite different from what we can find in software, we share the belief of [240] that ecological models and biological evolutionary theories can be adapted to study how software ecosystems and their constituent projects evolve. Even if biological and software ecosystems do not evolve at the same pace, life on earth got a much longer history and thus, had more opportunities to explore and find optimized pathways through natural selection.

The evolutionary processes that can be observed in nature may therefore be very inspiring for software engineers and researchers. It allows them to gain an increased understanding in how software projects compete or collaborate in their surrounding environment, and how this differs from biological environments. This insight will hopefully lead to guidelines and tool support to help the software project communities in predicting and improving survival of their projects. This will allow them to stay ahead of the competition, produce higher quality products and increase their fitness, resilience and stability over time in a rapidly changing environment.

The remainder of this chapter is structured as follows. Section 10.2 starts by exploring and comparing the notions of ecosystem and ecological principles that exist in biology and software engineering. Section 10.3 compares the notions of biological evolution and software evolution. Section 10.4 presents our emerging research to study the evolution of open source software ecosystem based on insights from the dynamics of natural ecosystems. Finally, Section 10.5 concludes.

10.2 Ecosystem terminology

The term *ecosystem* exists both in ecology and software. We present the characteristics and examples for both types of ecosystems in Section 10.2.1 and Section 10.2.2, respectively. In Section 10.2.3 we go beyond a simple reuse of terminology by drawing analogies between both types of ecosystem, despite the fact that the domain and discipline in which they are used and studied is completely different. In particular, we explain how ecological principles can be adapted and applied in the context of software ecosystems.

10.2.1 Natural ecosystems and ecology

According to [481], *ecology* is *the scientific study of the interactions that determine the distribution and abundance of organisms.* Typically, the dynamics of these interactions are studied in the context of an ecosystem. The term *ecosystem* was originally coined in 1930 by Roy Clapham, to denote the physical and biological components of an environment considered in relation to each other as a unit [927]. In other words, an ecosystem combines all living organisms (plants, animals, microorganisms) and physical components (light, water, soil, rocks, minerals) that interact with one another.

More generally, the ecosystem *dynamics* are traditionally represented in a *trophic web* (more commonly known as the so-called *food web* or *food chain*). This trophic web forms an interaction network that relates predator to prey or organism to resource [47, 682, 926]. Such a network usefully captures the relationship between *consumers* and the ecosystem's *resources* (such as food, nutrients and space), and the effect of this relationship on the population of different species in an ecosystem. A trophic web is organized in trophic levels corresponding to families of functionally consistent species. Consumer-resource relationships typically take place between different levels of the trophic web.

An ecosystem is the result of a delicate and dynamic balance between its interacting components. Trophic webs can be constrained from the bottom up, limited by the resources available to primary producers, or from the top down, driven by predation by top consumers. Ecosystems with "wasp-waist" control combine both mechanisms with partial effects in both directions acting simultaneously. Several marine ecosystems exhibit such a wasp-waist structure, where a single species, or at most several species, entirely dominate the population [64, 206, 412]. A typical example of the top-down control dynamics in an ecosystem is the so-called *predator-prey model*, representing a biological interaction in which some organisms (the *predators*) hunt for, and feed on, other organisms (their *prey*). The dynamics of such interaction can be described using linear or nonlinear models consisting of parametric differential equations [720].

Since an ecosystem's resources are finite, they need to be recycled or reused whenever possible. To achieve this, *energy* needs to be put into ecosystems constantly, typically in the form of light to drive the necessary biochemical processes that enable recycling of resources. An ecosystem has a *static equilibrium* if there are no exchanges between the components constituting it. Natural ecosystems typically have a *dynamic equilibrium* since there are always major exchanges between its components. For example, there may be important exchanges between the various levels in the trophic web, and an equilibrium is reached by fluxes in opposite directions whose total sum is zero.

The capacity of a biological ecosystem to maintain an equilibrium over longer periods of time is called its *stability*. Systems that can attain the most stable equilibrium survive the longest [826]. Often, this stability is put into peril by human interference, e. g. through the use of some of the resources required by the ecosystem. Examples of such disturbances for the ecosystem of coral reefs are, for example, climate change, water pollution and overfishing. *Sustainability* refers to the ability to maintain the ecosystem despite of humans deriving their needs from its natural resources.

The *resistance* of an ecosystem characterizes its ability to withstand environmental changes without (too much) disturbances of its biological communities. If the disturbances become too important, ecosystems may get out of balance (e. g. a meteorite impact that made all dinosaurs extinct). The ability of an ecosystem to reorganize itself and return to an equilibrium close to the initial one is called its *resilience* [405]. Because of the disturbance, the new equilibrium that is reached may be different from the original one (some types of organisms may have disappeared, and others may have taken up their place), so the ecosystem will have evolved.

Ecologists emphasize the importance of *biodiversity* [570, 576, 671], and generally acknowledge that the stability and resilience of an ecosystem is favored by a higher diversity. If the ecosystem has a large species diversity of producers and consumers that respond in different ways to disturbances, it is more likely that the ecosystem will be able to heal itself after a disturbance, since some species can compensate for others that disappear. Relating diversity to the aforementioned predator-prey relationship, Williams and Martinez considered two symmetric perspectives, from a prey's perspective and from a predator's perspective [570]. Other types of diversity have been studied by ecologists such as genetic diversity, functional diversity, spatio-temporal diversity, etc.

An *ecological niche* of a species determines the environmental conditions necessary for the species to maintain its population in response to the distribution of physical conditions, resources and predators in the ecosystem. Among others, it characterizes the subregions of the ecosystem's habitat that are usable or accessible to the species (e.g., land animals will not live under water).

Example 10.1 (coral reefs). Coral reefs are among the most biologically diverse ecosystems on earth [925]. Competition for resources such as food, space and sunlight are the primary factors determining the biodiversity and population of organisms on a reef. The single most important species of the ecosystem are the scleractinian coral polyps. They secrete hard skeletons that form the coral reef structure required for the other species to thrive: sea anemones (soft coral polyps), sponges, crustaceans, mollusks, sea urchins, fish, sea turtles, algae, sea grasses, and many more. These species have established a dynamic equilibrium with a delicate balance between predators and prey. Fluctuations in the population of one species can drastically alter the population of other species. External forces that may disturb the equilibrium of the coral reef ecosystem are for example hurricanes, but other human-inflicted changes may play an even more important role. Overfishing, for example, may lead to an increased growth of algae and sea grasses, resulting in an increase of the population of sea urchins that may destroy the corals.

10.2.2 Software ecosystems

Software systems are among the most complex artefacts ever created by humans. Collaborative software development has become increasingly popular over the last two decades. It represents a successful model of software development where communities of developers collaborate on a voluntary basis, while users and developers of the software can submit bug reports and requests for changes.

To reflect this increase in complexity and scale, the term *software ecosystem* has been coined by Messerschmitt and Szyperski [603] to refer to such systems. It has now become a very active area of research, as can be seen in a recent systematic literature review [556]. Unfortunately, in contrast to natural ecosystems, there is no common definition of software ecosystem. It can be defined and interpreted in different ways, depending on the point of view.

10.2.2.1 Business-centric viewpoint

One of the first occurrences of the term software ecosystem can be found in [131] where it is used to refer to the way in which software suppliers, vendors, competitors, users, and third-party developers interact in software product lines. This view emphasizes the *business* perspective of a software system. A similar view, including the socio-economic environment and regulatory framework is adopted by Jansen et al. [433, 434], who define a software ecosystem as a *"a set of actors functioning as a unit and interacting with a shared market for software and services, together with the relationships among them."* This view is schematically presented in Figure 10.1. An entire book is devoted to this perspective of software ecosystems [435]. A typical, but not exclusive, characteristic of these types of software ecosystems is the *competitive* aspect. The different projects in the ecosystem are in competition, either because they target the same end-users or offer the same type of service.

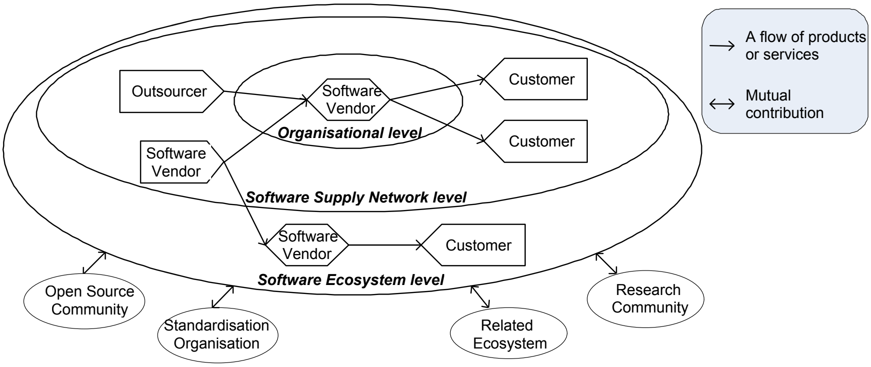


Fig. 10.1: Actors in a software ecosystem. Figure reproduced from [893] with permission from Edward Elgar publishers

Since, as illustrated above, business-centric software ecosystems often constitute a core strategic asset for its contributors and supporting companies, it is crucial to gain more insight in how ecosystems evolve and can be maintained successfully over time.

10.2.2.2 Development-centric viewpoint

An alternative, more fine-grained definition of software ecosystem is provided in the seminal work of Messerschmitt [603] to refer to *"a collection of software products that have some given degree of symbiotic relationships."* A similar definition is given by Lungu [546, 547], who defines a software ecosystem as *"a collection of software projects which are developed and evolve together in the same environment."* This environment refers to the development environment, *i.e.* the software and hardware tools used during the development process.

We extend these definitions to take into account the *collaborative* and *social* aspects as well, by explicitly considering the communities involved (e. g. user and developer communities) as being part of the software ecosystem. Like software projects, the communities involved evolve over time (users and developers come and go). In addition, there is a high degree of interaction, even some kind of symbiosis, between the software projects and the communities of the ecosystems. This viewpoint is adopted by [321, 333, 335, 594, 684, 723, 886] that focus both on the technical aspects of the software produced and the social aspects of the communities producing and using this software.

It is especially in ecosystems where the community works towards a common goal that the collaborative nature wins over the competitive nature. Typically, software ecosystems consist of a relatively closed core software system that provides the basic functionality and that is developed by a more or less stable core team of developers, surrounded by a large collection of contributions provided by peripheral developers or even end-users [631, 689, 723].

We can provide numerous examples of software ecosystems, and many of them can be interpreted from both the business-centric and the development-centric viewpoint.

Mobile app stores, commercial or free application repositories for mobile operating systems (such as *iOS*, *Android* and *Windows 8*), form a business-centric ecosystem. While these operating systems are provided by Apple, Google and Microsoft, respectively, the SDKs and APIs allow third-party developers to build mobile applications on top of these operating systems. The mobile app ecosystems consist of the users, developers, managers of the mobile OS and the third-party mobile applications built on top of them. The official mobile app stores allow for applications to be sold to end-users, with a shared profit. For Android, there is also a free and open source software repository of applications, called *F-Droid*.

The empirical study of the evolution of mobile applications is an emerging area of research. For example, Battacharya et al. [78] carried out an empirical study on the evolution of bug-related issues in 24 widely-used open source Android apps, while Basole et al. [77] studied the emergence and growth of mobile app stores in the mobile service ecosystem. McDonnell et al. [578] studied the rapid evolution of APIs and their adoption by client apps in the Android ecosystem.

IDEs for programming languages such as *Java* (e. g. *Eclipse* and *NetBeans*) or *Smalltalk* (e. g. *Squeak* and *Pharo* [721]) can be seen from a business-centric viewpoint. For example, the non-profit Eclipse Foundation is involved in the strategic direction, marketing and promotion of Eclipse and contains representatives of different companies such as IBM (the founder of Eclipse), Google, OBEO, Oracle, SAP, Talend. Eclipse is supported by numerous software vendors, and each of these vendors may provide different plugins with similar functionality, that are in direct competition with one another.

From a development-centric viewpoint, the Eclipse ecosystem is the universe of Eclipse *plugins* [191] together with the developers of these plugins. Studying the evolution of plugins is an active area of research [151–153, 915, 916]. All different Eclipse plugins rely on a common underlying architecture, platform and set of libraries without which they are unable to function correctly. The community of plugin developers therefore shares the common goal of improving a complete integrated software development environment. NetBeans, the main open source competitor for Eclipse, has a similar modular architecture with a common core.

Linux distributions form an ecosystem comprising several hundreds of actively competing Linux distributions, that are all based on a common core (the kernel of the Linux operating system [428] and a set of GNU libraries and utilities). The distributions vary in the system they target (e. g. desktop computers, laptops, tablets, smartphones, embedded systems) and the applications that are bundled with the distribution. Some distributions are commercially driven (e. g. Fedora Red Hat, SUSE, Ubuntu, and Mandriva), while others are entirely community-driven (e. g. Debian and Gentoo). An excerpt of the evolution of Linux distributions is shown in Figure 10.2. While the family of all Linux distributions is an ecosystem, each of the distributions that belong to this family can also be considered as an ecosystem of their own, composed of the packages (together with the necessary building and configuration files) contained in the distribution. Gonzalez et al. [341] have taken a closer look at the evolution of the Red Hat and Debian distributions.

Forges. Open Source Software (OSS) repositories, commonly known as *forges*, can be considered as business-centric, since there is no control on the governance of the projects hosted in the forge. Examples of such forges are SourceForge, GitHub, Bitbucket, Launchpad and Savannah. There are also many forges that are dedicated to particular programming languages, such as the CCAN archive network for the C programming language, the CPAN archive network for PERL, RubyGems for the Ruby language, the Python Packaging Index for Python programs, and so on. Because of the lack of control, within and across these forges there are often different projects with similar functionality between which the users can freely choose.
Capiluppi and Beecher [161] performed an interesting empirical study in which they studied the type of software forge (they refer to them as FLOSS repositories) and their mode of governance on the projects they host. They compared SourceForge (which they consider to be an *open* repository) with Debian (which they consider to be a *controlled* repository). They concluded that Debian hosted larger, more active and more complex structures. As a side-effect, more effort is needed to maintain these projects. Chapter 6 of this book explains how socio-technical information recorded in OSS forges (but also in microblogs and software forums) can be leveraged for different types of development and evolution activities, using a variety of information discovery and retrieval techniques.

Social networks, such as *Facebook*, *LinkedIn*, *MySpace* and *Google+* can also be regarded as business-centric software ecosystems. They allow application developers to develop and integrate third-party applications, through a well-defined API. This provides significant added value to both the social network and the application providers.

GNU (which is a recursive acronym for GNU's not UNIX) aims to provide a full free operating system based on the GNU General Public License (GPL) and the principles of UNIX. It is composed of GNU projects which are often ecosystems themselves. Examples of such sub-ecosystems are *R* and GNOME. Unlike most other software ecosystems, R is targeted towards end-user programming [321] since, the

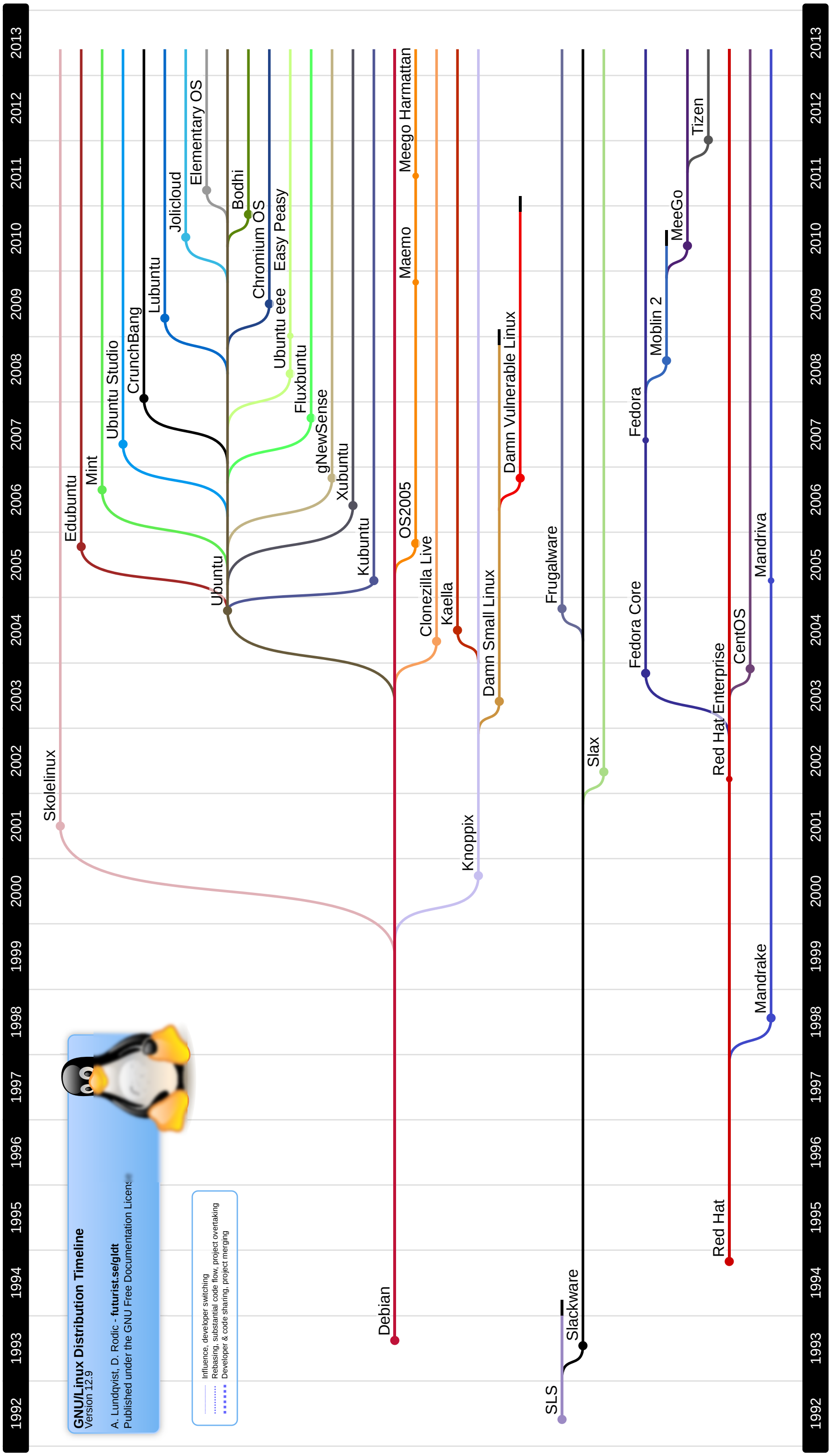


Fig. 10.2: Linux distribution timeline (simplified version based on http://futurist.se/gldt/)

majority of its contributors are statisticians and scientists rather than professional software engineers.

Archive networks. The GNU *R* community shares the goal of creating a statistical computing environment. It achieves this through the Comprenhensive R Archive Network (CRAN), a developer-centric ecosystem in which each community member can contribute packages implementing specific statistical analysis functions and visualization tools. Similarly, the TEX community has its CTAN archive network containing all kinds of material around TEX. There exists similar archives for other languages such as CEAN for Erlang, RubyGems for Ruby and PyPI for Python.

Graphical dekstop environments. GNOME and KDE are two full desktop environments for Linux and BSD operating systems. Both are based on a specific graphic toolkit (respectively GTK+ and Qt4). The developer communities share the common goal of delivering a complete user-friendly desktop environment. GNOME has been the topic of study for many researchers [320, 524, 640, 886].

Apache is an ecosystem with a community of developers based around the Apache Software Foundation and the Apache License. One of its most famous projects is the Apache HTTP server. Apache is a decentralized community that uses a consensus-like development process. The aim is to provide stable, open and quality software developed by technical experts. Mockus et al. compared the Apache development process with the one of Mozilla [613]. Bavota et al. [81] studied the evolution of the dependencies between projects constituting the Apache ecosystem. Weiss *et al.* [911] studied the emails exchanged by the contributors of the Apache projects for discovering simple migration patterns between projects and from the outside to a project. Gala-Pérez *et al.* observed that the ratio of email messages in public mailing lists to versioning system commits has remained relatively constant along the history of Apache, and therefore advocate this ratio as a way to measure healthiness of an ecosystem's evolution [306].

10.2.2.3 Collaborative and socio-technical aspects of software ecosystems

From the two aforementioned definitions of software ecosystems we have seen that projects belonging to a software ecosystem can vary in a continuum ranging from highly *competitive* (if the business-centric viewpoint prevails) to highly *collaborative* (if the sense of community is very strong and there is strong incentive to work together towards a common goal). Many ecosystems fall somewhere in between, with some degree of collaboration and some degree of competition. It is clear that the competitiveness will have an important influence on the way the ecosystem will evolve over time.

Example 10.2 (The R ecosystem). Let us have a look at the collaboration and competition in the previously mentioned R ecosystem. It only minimally complies to the business-centric view because of its open nature: all packages in the CRAN archive network are required to comply to an open source license. Because of this there is much less competition in the sense of having many different packages with similar functions. When packages do contain similar functions (this tends to be more common for "basic" functionalities), it is mainly because some contributor needed more advanced features for that function in its own package than what was available in existing packages. In many cases, that contributor will write his own function inside his own package instead of proposing to contribute changes to the existing one. Thus, there is little collaboration, but a more fragmented implementation of features across packages that are developed rather separately from each other. Formally verifying the above claims is outside the scope of the current chapter, as it requires an extensive empirical study of R packages.

Technical aspects are essential for software ecosystems. They need to rely on a sophisticated software and hardware infrastructure and tools needed for their proper functioning, distribution, development, maintenance and evolution. Typical support that is provided are SDKs, APIs, download repositories, package management, dependency management and installation tools, version control systems, tools for change tracking, bug tracking and defect management, mailing lists, websites and other communication fora.

Social aspects and communication between the members of the software development team are at least as important as the technical aspects for the success of any software project [90, 236, 265, 868]. This is especially true for OSS projects where it is, in most cases, easier to become involved in the development team. This implies that the team structure needs to be more flexible in order to accommodate the easy integration of newcomers and to deal with the frequent departure of developers. Chapter 6 of this book proposes a number of techniques to recommend "compatible" developers to a project.

Fitzgerald [295] coined the term OSS 2.0 to reflect the new generation of OSS ecosystems that significantly "evolved" over the last decade or so from its single-project antecedents. Empirical results and insights obtained for individual OSS projects do not necessarily apply to projects that are part of a bigger, highly collaborative ecosystem of interacting parts. Nakakoji et al. [631] distinguished between different types of OSS community members: developers, bug fixers, bug reporters, readers and passive users. They further subdivided developers into peripheral developers, active developers, core members and project leaders. They proposed a so-called *onion model* for the OSS community structure, suggesting that there are very few project leaders, a bit more core members, even more active developers, and so on, and that promotion and migration of contributions tends to follow the layers of this model. Jergensen contested this onion model in an OSS 2.0 setting [439], by showing that contributor migrations do not tend to follow this model in many cases. Many other empirical studies have studied the activity patterns of, and differences between, core developers and peripheral developers [162, 250, 689, 723, 828, 941]. A detailed discussion of these is, however, beyond the scope of this chapter. We refer the interested reader to [336].

Still related to developer communication, Abreu and Premraj [2] studied the correlation with software quality. They observed a statistically significant correlation between communication frequency and number of injected bugs in the software.

Through mining the source code repository and mailing lists of the well-known Apache and Mozilla OSS projects, Mockus et al. [613] investigated the roles and responsibilities of developers, and observed a set of implicit conventions among developers that implies an intensive communication. Madey et al. [305, 868] analysed the social networks involved in OSS development and observed power laws at many scales. Bird et al. [108] analysed social networks emerging from mailing lists discussions and observed a Pareto distribution. Mailers tend to form a small-world network at several points of view; for instance, few mailers received messages from an important number of persons while most of mailers received messages from few senders. A strong correlation between mailing and coding activities was found and evidence was provided that the role of developers in mailing lists is more important than the other mailers.

10.2.3 Comparing natural and software ecosystems

The premise of this chapter is to learn from ecology and natural ecosystems, that have evolved over millions of years, and use this knowledge to improve our understanding of software ecosystems. Existing research on natural ecosystems has already provided many useful insights on the underlying mechanisms and how we could better manage and preserve these ecosystems. Our hope is to learn from this research, and to apply some of its insights to obtain better strategies for managing, developing and maintaining software ecosystems, and to come up with processes that increase the fitness of projects and contributors belonging to the software ecosystem.

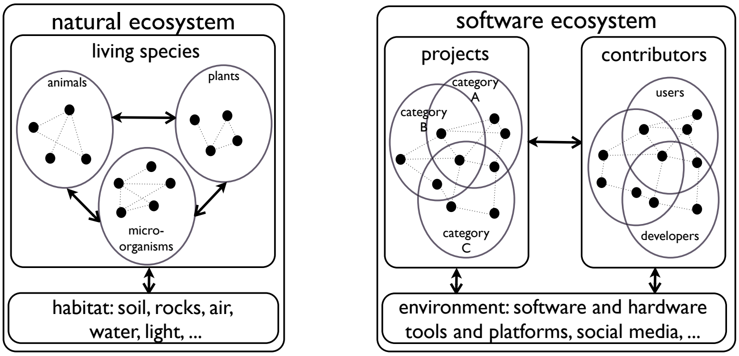


Fig. 10.3: Natural versus software ecosystems

When comparing biological evolution with software evolution, despite their obvious differences, we can also draw many analogies. This analogy is illustrated in Figure 10.3. If we take the development-centric viewpoint of a software ecosystem, we can consider the software projects as being the equivalent of the "living species" of a natural ecosystem, and the physical habitat is replaced by the socio-technical environment in which these projects co-exist and evolve. The projects require software and hardware resources for developing, installing and executing the software products belonging to the ecosystem. All software projects interact with each other and with the user and developer communities and available resources. The software ecosystem can also be interpreted in an alternative way, by considering the contributors to the software projects as the equivalent of the "living species" and the software products then become part of the software and hardware environment of these species. This view may be particularly suited if we wish to study the social aspects of a software ecosystem. In practice, both of the above views are complementary and need to be combined in order to fully understand how software ecosystems evolve.

Example 10.3 (Coral reefs). In a coral reef ecosystem, the scleractinian coral polyps are responsible for creating the coral reef structure required for the other species to thrive. We find a similar idea in most business-centric software ecosystems, where there is typically a core set of projects (or core architecture), developed by a core group of developers, based on which the other projects are created.

Like natural ecosystems, a desirable property of software ecosystems is to be *sustainable*, in that their user and developer communities can use, maintain and improve the ecosystem's projects over longer periods of time. Just like the habitat of a natural ecosystem, the environment of a software ecosystem may undergo important changes, whether they be planned or unexpected. The *resilience* of a software ecosystem then refers to its ability to return to a stable *equilibrium* after minor or major disturbances. Examples of such disturbances are the appearance of a new competitor products, a loss of interest by the user or developer community, a change of technology (e. g. switch from the use of a centralized version repository to a distributed version repository), the introduction of new communication channels (mailing lists, StackOverflow—cf. Chapter 5 and Chapter 6, respectively) and other ways of collaboration.

Biological species evolve through mutation and crossover of genes between individuals of the same or different species. An analogy of such gene transfer in software projects could be the reuse of code from one project to another, or the migration of software developers from one software project to another.

Natural ecosystems require *energy* (e. g. air, water and sunlight) to thrive. The same is true for software ecosystems, but the type of energy required is quite different. If we consider the software projects as the species of a software ecosystem, the energy required to maintain and evolve them is the time and effort invested by the users and developers contributing to the software ecosystem, through commits in the version repository, bug and change requests, mails in the mailing lists, communication in forums and websites, and so on.

The notion of *biodiversity* also exists in software ecosystems, at different levels (as illustrated, in part, in Figure 10.3). First of all, there is a diversity of contributors involved in software development. The role of contributors may range from more passive (e. g. users) to more active (e. g. developers, translators, UI specialists, etc.). Zooming in on the developers, we can distinguish between core developers, active developers and peripheral developers at a more fine-grained level [631, 723, 828]. For the software projects that are part of the ecosystem we observe a similar diversity. Some projects will be more user-oriented (i. e. they can be installed and used by end-users) while others will offer the core functionality that is needed by others in order to function properly. Sometimes there may be different projects with a similar functionality. This may be beneficial for the biodiversity since the disappearance of such projects will not be detrimental to the ecosystem since the other project could take its place. Another example of diversity is *conditional compilation*, which allows for a software product to create different variants adapted to specific platforms or user needs. *Software product lines* encourage controlled diversity across different software products with some shared common features (see Chapter 9).

It is likely that the mechanisms controlling the ecosystem *dynamics* (top-down, bottom-up or wasp-waist) can be adapted to software ecosystems as well. If a software ecosystem is mainly driven by its core developers or by limited hardware resources it might follow a bottom-up control process. If it is mainly driven by change and bug requests from the end users, it might rather have a top-down control. In many cases, the type of control is probably a mix between both, in the sense that some projects of the ecosystem (typically the core projects) will be driven or initiated by the developers, while others will be driven by the end-users' change requests and desire for new or modified functionality. A better understanding of the type of dynamics that control a software ecosystem may ultimately lead to better management strategies for maintaining the ecosystem over time.

The notion of *ecological niche* of a species also has a counterpart in software ecosystems: if we consider contributors (e. g. developers) to be the equivalent of a species, their ecological niche is determined by environmental factors such as the operating system they are using, their preferred programming language, the APIs they are using, their domain of interest, and so on. These characteristics will constrain the ecological niche of a developer to a subset of the total set of projects she could potentially contribute to.

10.3 Evolution

10.3.1 Biological evolution

A biological species corresponds to a group of organisms capable of interbreeding and producing fertile offspring. Biological evolution is characterized by the fact that a species is composed of many individuals whose genetic code differs. Those individuals can reproduce, leading to mutations and crossing in the genetic code. The evolutionary driving force is *variation* and *natural selection*. A central idea in the evolutionary theory of natural selection is the notion of *fitness*. It describes the ability of a species to both survive and reproduce, and is equal to the average contribution to the gene pool of the next generation that is made by an average individual of the specified genotype or phenotype [673].

Different theories have been proposed by biologists to explain the evolution of biological species, and the field still evolves today. The *Darwinian evolution* model is generally considered as the major mechanism driving *biological speciation* (i.e. one species differentiating into two) in life on earth [218]. The field of *phylogenetics* studies, among others, the biological evolution history of a set of species [764]. In the Darwinian model, the evolution history can be represented by *phylogenetic trees* [294]. Such a tree describes the evolutionary relationships among species assuming that they share a common ancestor and that evolution takes place in a tree like manner.

There are other, less well-known evolutionary models, such as *reticulate evolution* [523, 779]. These models cannot be represented using a tree structure, but require some graph-like or network-like structure instead [414]. Reticulate evolution refers to the dependence between two evolutionary lineages. This is radically different from pure Darwinism where there cannot exist such transfer of information between two different species. When reticulation occurs, two or more evolutionary lineages are combined at some level of biological organization. Because life is organized hierarchically, reticulation can occur at different levels: chromosomes, genomes and species. At the species level, events such as *hybrid speciation* (by which two lineages recombine to create a new one) and *horizontal gene transfer* (by which genes are transferred across species) are the main causes of reticulate evolution. A group of animals where reticulate evolution is suspected to be of major importance is the scleractinian corals [891].

Apart from Darwinism and reticulation, other evolutionary theories have been proposed, such as *Lamarckism* [488]. Lamarck considered that the evolution is based on uses and needs rather than on natural selection. While this theory has been superseded by Darwinism in biology, this does not necessarily mean that we should exclude it as a possibly useful theory for modeling the evolution of software ecosystems. Indeed, software is developed with the aim to fill a need and its survival fitness is partially constrained by its likelihood to be used.

A fairly recent evolutionary theory is the so-called *hologenome theory of evolution*, originating from studies on coral reefs [732]. In this theory, the object of natural selection is not the individual organism, but the organism together with its associated microbial communities. This theory may perhaps be more closely related to what one observes in software ecosystems, where one should not consider the object of evolution (the software project) in isolation, but rather together with its associated community of contributors (e.g., users and developers).

The biological phenomenon of *co-evolution* [862] arises if the genetic composition of one species changes in response to a genetic change in another one. This can occur, for example, when two or more species interact and influence each other, or live in *symbiosis* (e.g., host-parasite, plant-pollinator).

The notion of ecological *refuge* is also very relevant in the context of ecosystem evolution [101]. The conditions in a refuge are such that the species are protected from certain threats such as predation. A key characteristic of refuges is that they are a reservoir of diversity since they provide a means to sustain species that are not the fittest at some point in time. Refuges are important in an evolution context, since species in refuges may become dominant species in the future in response to environmental changes.

10.3.2 Comparing biological evolution with software evolution

To be able to apply the aforementioned and other biological evolutionary models to study the evolution of software ecosystems, these models will need to be adapted because there are notable differences between software projects and living species.

While biological species evolve due to changes and variations in the genetic code of its individuals, it is difficult to consider a software project as a collection of individuals. Of course, we could view the different instances of a software system that are deployed on particular machines as individuals of the biological species. The major difference is that there is strictly no variation in the code of the various software project instances installed (to draw the parallel with differences in the genetic code of living organisms), while even small genetic differences between biological individuals is a major driving force of biological evolution. It is worth noting, however, that the equivalent of phenotypic changes in living organisms is represented at a varying degree in software: configuration files, installable plugins or packages can modulate how a particular instance operates in a given context.

Another type of software where one can observe a sufficient level of variation necessary for being able to apply biological evolutionary theories are so-called *software product families*. These are addressed in Chapter 9 of this book. Each member of a product family is a variant that has similarities and differences with the other product family members, and the family as a whole evolves over time.

The main driver for evolution of biological species is the creation of offspring through biological reproduction. This is not true for the elements that constitute a software ecosystem: software projects cannot “reproduce” themselves to produce new generations (read: versions or releases) of offspring.[1] Note that one could also consider project forking or branching as some kind of reproduction. A similar argument as above holds for the members of the ecosystem’s communities: new generations of developers and users are not produced through interbreeding of existing members, but rather through the intake of new members from outside the ecosystem.

The rate at which software projects evolve is several orders of magnitude higher than the evolution of biological species. Hence, one has to determine the relative temporal scale at which comparison is possible between biological mutations and changes in software projects.

[1] This argument does not necessarily hold for self-adaptive systems, which are capable of dynamically changing their runtime behavior. For more information on this specific type of software system we refer to Chapter 7.6 of this book.

We can only collect very partial records of the evolution of natural ecosystems, restricted to limited sampling in time and space. Models in ecology are thus always applied with a large degree of uncertainty. On the contrary, more exhaustive historical records exist for many open source software ecosystems, from their onset, thanks to version control systems[2] where every change is recorded and documented.

Scientific research on biology is primarily observational and passive. One can observe how natural ecosystems have evolved in a self-organised way over long period of times, and develop theories that explain this evolution. Given the long time scales involved it is hard to carry out "in vivo" experimental research to study how actual ecosystems and the species populating it evolve by modifying certain parameters in the ecosystem. For software ecosystems, it is really possible to carry out applied, in vivo research, since the software environment involves human beings (developers and users). This makes it possible, in principle, to interact with them in order to find out how and why a software project has evolved over time, and making it easier to alter the way in which the ecosystem will evolve in the future.

10.3.3 Transposing biological models to the software realm

Given these many differences, the question arises whether ideas from biological evolution can be easily adapted to gain a better understanding of software evolution. Nehaniv [638] discussed the differences between software systems and biological species from an evolutionary point of view. Svetinovic [811] suggested that a comparison between software evolution and biological evolution is a fertile field of study. Yu and Ramaswamy [940] suggested that software systems share similar evolvability properties with biological systems, implying that studying the evolution of these biological systems can help us understand and improve development of software systems. None of the aforementioned papers, however, have empirically studied this potential.

Some researchers have gone a step further in adapting biological models or mechanisms in the context of software evolution. For example, Hutchins [416] used genetic algorithms to understand evolutionary software development processes. Each branch of a software project is compared to an individual of a biological species and merging of branches is similar to the crossover operation (reproduction of two individuals). Software evolution is then described as a form of human-guided search for a program meeting requirements. Jaafar *et al.* [429] used phylogenetic trees to show the evolutionary history of object-oriented programs. They suggest to use such trees to facilitate the detection of code decay and fault-proneness.

Baudry [79] studied the relevance of the notion of *ecological refuge* in the context of evolving OSS projects. More in particular, they analyzed the potential of largely inactive projects as alternatives for biodiversity and evolution: some of these "unsuccessful" projects may survive and increase diversity by seeding future, successful

[2] While some data can still be incomplete in software repositories, it remains far more complete than for biological species where historical data like fossils are very sparse and incomplete.

projects. They empirically analyzed this by studying project forks for 48 projects in the GitHub forge, and found 3 occurrences of the refuge effect. Similar to national parks, that serve to protect endangered species, software forges may therefore serve to protect unsuccessful projects and reuse or revive them in the future.

Calzolari et al. [155] explored the use of the biological predator-prey model in the context of software evolution. This model has been used in biology to describe the dynamics of an ecological systems using linear or nonlinear models consisting of two parametric differential equations [720]. The basic idea is that software defects (requiring corrective actions) can be seen as the equivalent of biological prey, whereas the programmers act as predators (removing the defects by correcting them). Empirical evidence of the usefulness of this model was given by analyzing the evolution of two industrial software systems and accurately predicting their dynamics using the proposed model. Some adaptations of the original biological model were needed since, unlike species, software defects cannot reproduce themselves, implying the elimination of the reproduction term in the dynamic model.

Posnett et al. [698] explored a similar idea, by considering software modules as predators that feed upon the limited cognitive resources of developers (their prey). They combined this with the notion of biodiversity [570] to measure how focused the activities on a module are, as well as how focused the activities of a developer are. They found empirical evidence that more focused developers introduce fewer defects. Conversely, increased module activity focus leads to a larger number of defects.

To transpose other theories of evolution and speciation of living species to software ecosystems we might require a mix of different evolutionary mechanisms, with probably a domination of reticulate-like mechanisms over pure Darwinian differentiation. For example, we could transpose the notion of *fitness* to reflect the ability of projects to survive and maintain themselves within the ecosystem of which they are part. We could also transpose the notion of *biological speciation* to software ecosystems to represent the mechanism of *software project forking*.

Example 10.4 (Evolution of Linux distributions). One illustration of this phenomenon is the different GNU/Linux distributions that have forked from a few main distributions (Fig. 10.2). The distribution timeline of Linux distributions does not represent a tree structure but forms a directed acyclic graph with some connections between different branches of the tree, indicating the exchange or sharing of ideas, code, and developers (corresponding to horizontal gene transfer across species). This may ultimately lead to *project merging*, if the level of sharing becomes sufficiently high. Such project merging fits the phenomenon of *reticulation* that occurs when two or more evolutionary lineages are combined at some level of biological organization. The following examples illustrate these phenomena: (i) Maemo (Nokia's mobile OS based on Debian) and Moblin (an Intel Atom optimized GNU/Linux distribution) merged to form Meego; (ii) Crunchbang was first based on Ubuntu, but since 2010 it has been based on Debian rather than Ubuntu.

The biological phenomenon of *co-evolution* can also be useful to explain and model certain aspects of software ecosystem evolution. The term co-evolution has been borrowed by software engineering researchers on numerous occasions and for various purposes, but only at a very shallow level. A typical usage is to reflect the need for different types of software artefacts (e.g., design models and code) to be kept synchronised while they are changing from one version to the next [188, 239, 285]. Chapter 2 of this book discusses the need to co-evolve software models and their metamodels. In the context of open source, Ye et al. [937] explored the co-evolution between software systems and their developer communities. Yu [939] has studied the co-evolution between 12 kernel modules of Linux in 597 different releases and found that co-evolution arises when one module changes in response to a change in another component. Jaafar et al. [429] studied the fault-proneness of co-evolved classes in object-oriented programs. Fluri et al. [297] analyzed the co-evolution between source code and comments. Zaidman et al. [943] explored the co-evolution between production code and test code.

In the context of software ecosystems, we propose to study the co-evolution between different projects belonging to the same ecosystem. Two software projects fulfilling a similar purpose inside the same ecosystem (e. g. two games in a mobile app store, or two drawing tools or text editors in an OSS forge) can be seen as being in a state of competition. This can lead to co-evolution in the sense that a new feature in one of the projects may disavantage the other one and may force its developers to adapt the project if they want it to maintain its fitness for purpose. Similarly, if two software projects are complementary and useless if used separately, developers of both projects will need to collaborate when evolving their software. The latter scenario can be viewed as a kind of symbiosis.

10.4 Exploratory case study

In this section, we report on techniques used to study natural ecosystems and their adaptation and application to software ecosystems. We do this through a case study on the well-known GNOME ecosystem that will be presented in subsection 10.4.1. In subsection 10.4.2, we explore to which extent the characteristics of GNOME, an example of a software ecosystem, differ from the characteristics of a biological vegetation ecosystem. In subsection 10.4.3, we study the immigration of new developers in GNOME and the local migration of developers across GNOME projects, motivated by the fact that the success and sustainability of a software ecosystem depends on its ability to attract and retain developers.

10.4.1 The GNOME *OSS ecosystem*

In order to assess to which extent biological models, techniques and tools for ecosystems and evolution are applicable to software ecosystems, we need to carry out empirical studies. These studies will allow us to determine what are the main commonalities and differences in the characteristics and dynamics of biological and software ecosystems.

To carry out such empirical studies, we need access to ecosystems that are sufficiently large (in terms of number of projects), active (in terms of number of contributors) and long-lived (in terms of number of years of activity). To avoid confidentiality issues and to facilitate reproducibility and replication of results by other researchers, we also require the analyzed data to be freely accessible. These requirements naturally lead us to OSS ecosystems. OSS is generally established as an important software development practice, and all major software vendors rely, to some extent, on OSS. In some cases, the OSS products they rely on are even critical to the company's success.

Lehman's laws of software evolution [505, 511] have inspired many researchers and have significantly influenced research on OSS project evolution [291, 330]. Many of these studies focus on understanding and predicting the evolution of individual software projects and their developer communities. Much less empirical research exists on the evolutionary study of long-lived OSS *ecosystems* containing hundreds or even thousands of projects and contributors.

As an exploratory case study, we analyse the GNOME OSS ecosystem, since it has been the subject of a lot of research in the past [320, 331, 336, 536, 640, 886]. The historical data of all GNOME projects is accessible through their Git version control repositories. We have shared our extracted data set with the research community [332]. According to `git.gnome.org`, GNOME has been under development since January 1997, and currently contains more than 1400 projects (more than half of which are archived) to which over 5000 contributors have contributed over the entire lifetime of GNOME. Table 10.1 provides some basic historical metrics for the GNOME ecosystem, obtained over a period of 15 years. Figure 10.4 gives an idea of the size distribution of GNOME's projects.

Table 10.1: Basic historical metrics for GNOME from January 1997 to December 2012. A *file touch* corresponds to the addition, removal or modification of a particular file in a particular commit.

Metric	Value
number of projects	1,418
number of projects with coding activity	1,353
number of commits	1,303,649
number of commits containing code files	685,007
number of file touches	12,394,786
number of code file touches	6,183,282
number of contributors having made at least 1 commit	5,885
number of coders (authors having made code file touches)	4,321
considered lifetime	5844 days (16 years) Jan 1997 → Dec 2012
number of considered 6-month activity periods	32

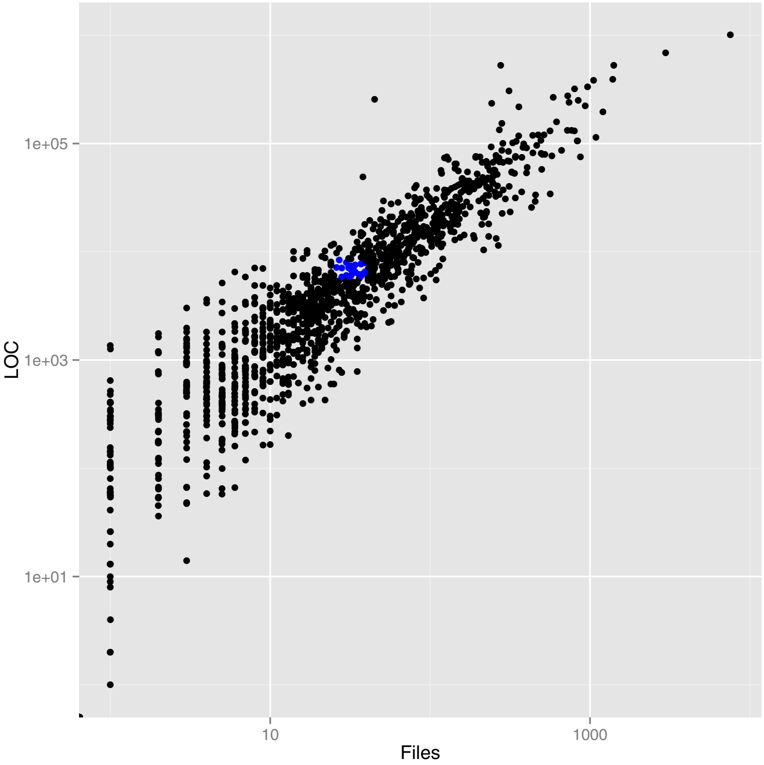


Fig. 10.4: Size (on log-log scale) in number of lines of code (LOC) and number of files of GNOME projects. Extracted using CLOC from the latest version of each git repository of January 8, 2013. Total size: 2,2251,913 LOC and 104,594 files.

In previous work [886], we have observed that the contributors to the GNOME ecosystem can be classified in different, partially overlapping, subcommunities according to their types of activity. The principal activity type of a contributor (approximated by the number of file touches of a particular type in her commits) determines to a large extent her work pattern and part of her ecological niche.

The current case study focuses solely on the coding activity. Our results will therefore be restricted to coders, code files, commits containing code file touches, and projects containing such commits. *Coders* are GNOME authors having an account and code commit activity in at least one of GNOME's Git repositories. *Code files* are files in a commit that are considered to contain source code, based on their file extension (e.g. `.java` for Java files, `.c` and `.h` for C files, `.cpp` for C++ files, `.py` for Python files, `.pl` for Perl files, and so on). Of all thirteen activity types we defined for GNOME in [886], we observed that coding was the most important activity of the frequent GNOME contributors. Figure 10.5 gives information on the usage of programming languages across all GNOME repositories. It was extracted using the CLOC code lines counting tool (cloc.sourceforge.net). We see that C and C++ are by far the most frequently used programming languages in GNOME, followed at a distance by Python and C#, and then followed by Perl.

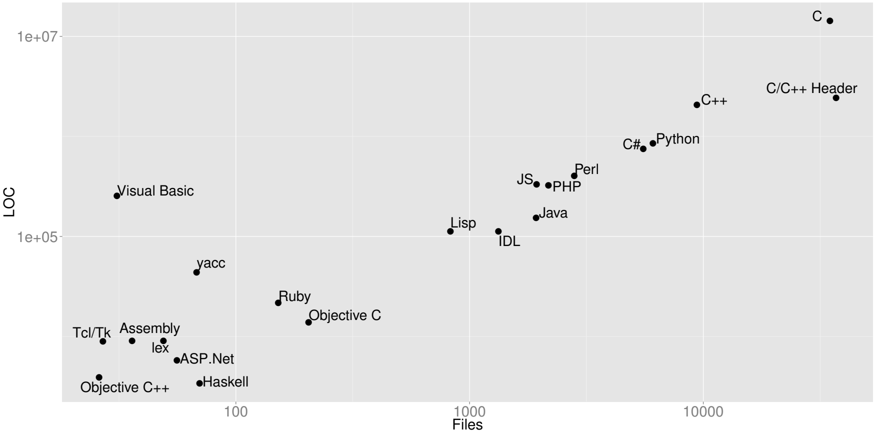


Fig. 10.5: Language usage in GNOME. Extracted using CLOC from the latest version of each repository of January 8 2013.

A challenge during data extraction is that coders may use different accounts. To avoid counting such coders as separate identities, we used identity matching. Multiple techniques have been proposed for this [108, 334, 476, 722]. We merged the different identities belonging to the same person using a semi-automatic approach. First we applied an automatic algorithm detailed in [886] and then we manually post-checked the results to remove false positives.

We chose 6-month activity periods, since GNOME has a 6-month release policy (two releases per year in March and in September). The first considered period starts on 1 January 1997 and the last one starts on 1 July 2012. For each period, we only consider commits containing at least one code file touch. Similarly, we only consider a coder to be active in a GNOME project during a period if she made at least one code commit using one of her accounts during that period. Her number of code commits for that period is the sum of the number of code commits of all her accounts for all GNOME projects during the period. The number of code file touches of a coder during a period is the sum of the number of code file touches in each of her project commits during the period. As we can observe from the boxplots in Figure 10.6, the majority of coders contribute to a single or very few projects (median value of 1, mean value of 4.866) and have a limited number of code commits (median value of 4, mean value of 156.4). The distributions are strongly skewed with a long tail.

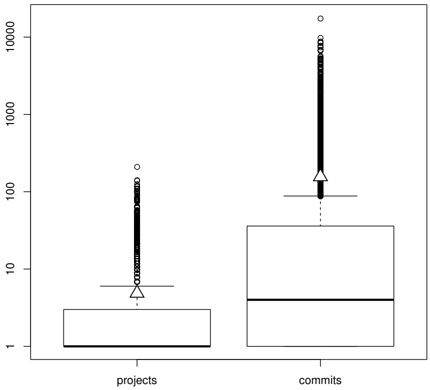


Fig. 10.6: Boxplots showing the distribution of projects and commits per coder. (The white triangle shows the mean value.)

10.4.2 Comparing GNOME *with a natural ecosystem*

In Section 10.2.3 we presented different ways to compare natural ecosystems to software ecosystems. There is, however, another useful analogy that we can draw. When studying natural ecosystems, such as a vegetation community of different species of plants in a forest [880], one can take samples of individual plants at different arbitrarily chosen locations (so-called sampling stations), and use this to get an idea of the coverage of the location by each species and the variation of this coverage across the ecosystem, for example in order to assess the biodiversity. For software systems, one can adopt a similar approach: randomly select a number of software projects belonging to the ecosystem, and count the coverage (in number of commits, or any other measure of activity) of each contributor to the ecosystem. In this analogy, contributors correspond to the equivalent of a plant species, and their number of commits to the project correspond to the coverage. One can then use the same portfolio of techniques as those used for studying natural ecosystems.

One such technique is hierarchical clustering. For the considered GNOME lifetime, we computed a matrix with projects (i. e. locations) as columns, coders (i. e. species) as rows and the number of code commits per coder as cell values. We have found in the boxplots of Figure 10.6 that more than half of the coders (54.5% to be more precise) were not involved in more than one project. Thus in the remainder of this section, we will ignore these “singleton” coders, since we would like to group together projects based on the similarities of their community and “singleton” coders do not provide useful information on such similarities or dissimilarities. We removed the columns containing only zeroes (i. e. projects without coding activity) and the rows with less than two non-zero cells (i. e. coders that were active in zero or only one project). This gives a matrix containing a total of 1352 projects and 1966 coders.

After applying a hierarchical clustering on this matrix, in contrast to the results for a natural ecosystem, we observe a large number of small clusters, implying that coders are much more restricted to a few projects than plants are on sampling stations, resulting into most items connected much higher in the clustering dendrogram.

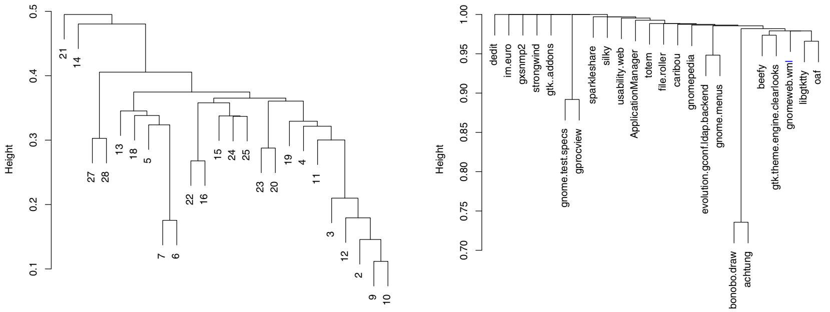


Fig. 10.7: Comparison of hierarchical clustering applied on: [left figure] a vegetation dataset at 24 randomly chosen locations on 44 plant species; and [right figure] a GNOME dataset of 24 projects chosen randomly from the fourth quartile and 44 randomly chosen coders chosen randomly from the fourth quartile.

On the left of Figure 10.7, the aforementioned vegetation community [880] measured at 24 randomly chosen locations is hierarchically clustered.[3] The Bray-Curtis distance was used as a basis for the clustering process [139]. On the right of Figure 10.7 the same hierarchical clustering technique is applied to a sampling of GNOME software ecosystem and its code contributors for 24 projects chosen randomly from the last quartile (i. e. projects with at least 283 commits) and 44 coders chosen from the last quartile. The values of 24 projects and 44 coders were chosen so that the clustering contains the same amount of species and locations as the vegetation ecosystem data.

From this comparison, we observe that a vegetation ecosystem seems to behave quite differently from a software ecosystem. The survival strategy of plants is to be as ubiquitous as possible at all locations of the ecosystem (through direct competition for sunlight and other nutrients with the other plants in its direct surroundings). In contrast, the survival strategy of code contributors appears to be by specialising themselves in very few projects of the software ecosystem. As such, there is much less competition with the other coders, and the dynamics of the ecosystem are based primarily on collaboration, as opposed to competition with other coders.

[3] We applied a hierarchical clustering with single linkage using the R function `hclust`.

After ignoring all coders that are involved in a single project, and carrying out a hierarchical clustering on *all* GNOME projects, we observed an interesting pattern: the majority of GNOME projects related to the programming languages Perl and Python, respectively, were clustered together. The fragments of the cluster dendogram illustrating this phenomenon are shown in Figure 10.8. Hence, the programming language used in projects appears to be both a barrier limiting expansion of developers across projects, and a subdomain inside which developers tend to interact more closely. This allows us to confirm and further refine the notion of ecological niche for GNOME code contributors.

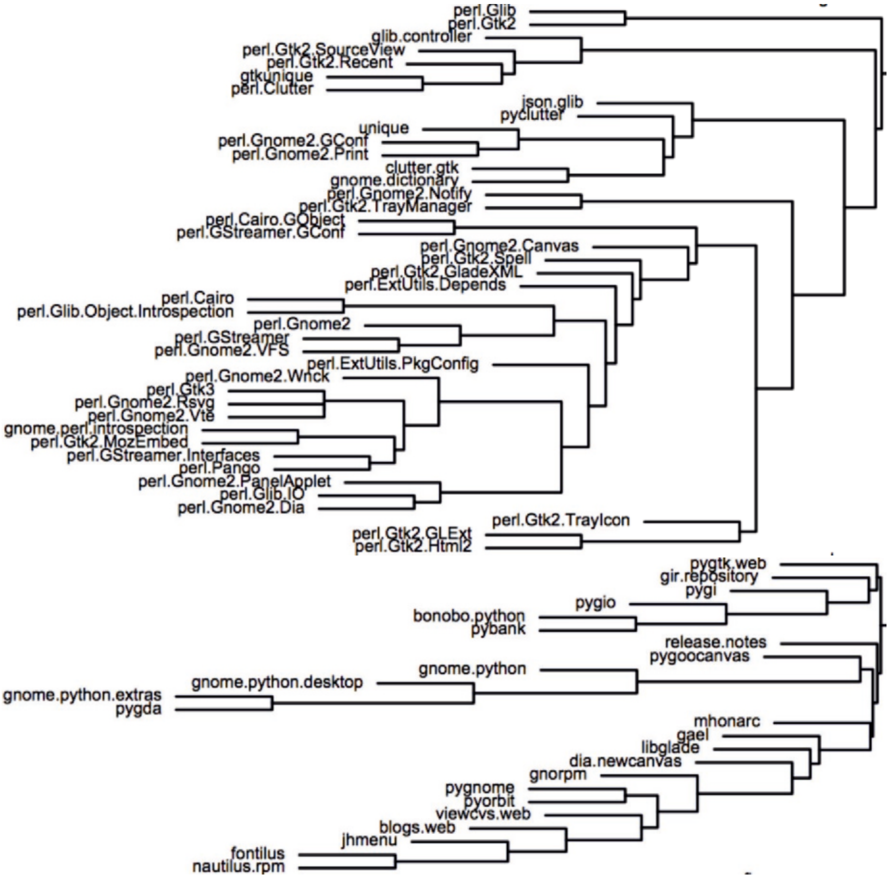


Fig. 10.8: Zoom on two interesting clusters (representing the communities of Perl coders and Python coders, respectively) in the dendograms obtained through hierarchical clustering of GNOME project and coder data. Those clusters contains a majority of Perl and Python projects. This shows that those projects' communities are very similar and tied.

Another technique frequently used for studying natural ecosystems is *principal component analysis* (PCA). Figure 10.9 again compares the vegetation community measured at 24 randomly selected locations to the coder's commits measured at 24 randomly selected Gnome projects. The PCA is carried out on correlation matrices in both cases. Figure 10.9 shows how the total variance decreases among the first 10 principal axes. On the left, we observe that the vegetation data can easily be reduced down to the first three axes while loosing less that 20% of the total variance. This means the dataset is highly structured with essentially three degrees of freedom in the distribution of the vegetation. On the right, we do not observe an important decrease of variance of the 10 principal axes for the Gnome dataset. The variance is therefore more homogeneously distributed, meaning there are rather different groups of coders working on each of the 24 projects. This confirms our previous findings that, in contrast to the vegetation ecosystem, GNOME has a relatively well-balanced community.

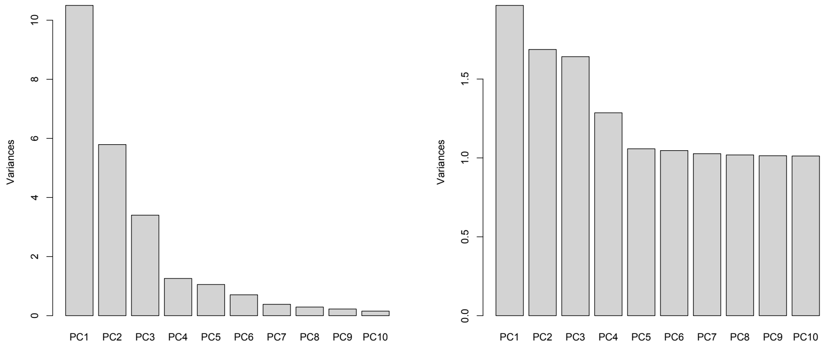


Fig. 10.9: Comparison of the variance of the first 10 principal components of PCA applied on the biological vegetation dataset (left) and the GNOME dataset (right).

To summarize, the results we obtained for GNOME are quite different from what one typically observes in natural ecosystems, where there is a high degree of competition between the species. This usually leads to well-differentiated subcommunities with identifiable key species that largely structure the whole dataset, leading to well-separated clusters in the dendrogram and to most of the variance caught by the few first principal components in the principal component analysis. It remains to be seen if this major difference with natural ecosystems is found in other software ecosystems as well. If this turns out to be the case, the traditional biological evolutionary theories (such as Darwinian evolution) are probably not applicable to OSS evolution, because of the much lower level of competition observed, while competition is an essential driver of biological evolution. Future studies on other software ecosystems will allow us to shed more light on this issue.

10.4.3 Migration of GNOME developers

The process of intake (also known as immigration) and retention of developers to OSS projects has been the subject of study by many researchers. Von Krogh et al. [903] have studied how one can join a project, get write access to a source code repository and then how the newcomer specialisation is related to contribution barriers. Canfora et al [160] have designed an approach to identify which contributor could be assigned as a mentor to a newcomer. Zhou and Mockus [950, 951] have shown that the social environment impacts both rate at which people joins a project and the chance that a new developer becomes a long-term one.

The reason for this interest is that the success and sustainability of a project depends on its ability to attract and retain developers. There is a crucial difference with natural ecosystems, where populations of individuals can create new generations through reproduction. In OSS projects, the only way to increase or renew the population is to attract new contributors from the outside. If a software ecosystem is not interesting enough, it will not attract new developers, or worse it may even loose its developers to other systems.

New developers are interested in joining OSS projects for variety of reasons, such as personal interest in, need for the software, increasing their personal reputation, out of altruism or because they are being paid for it [114, 292, 369, 400, 637].

Little empirical studies exist, however, on the migration of software developers across projects. Weiss et al. [911] studied the emails exchanged by the contributors of the Apache projects for discovering simple migration patterns between projects and from the outside to a project. They observed that many developers joining a project come from another project. These developers tend to migrate together with their workmates. Based on three case studies (Apache web server, Postgres and Python), Bird et al. [109] found three factors that influence immigration, i. e., intake of new developers: their technical commitment, skill level and social status. Among others, they found evidence that demonstration of skill level by submitting patches to known bugs will increase the likelihood of becoming an official developer of the project.

Jergensen et al. [439] studied how GNOME developers start using social mediums and move progressively to socio-technical and technical mediums. They tried to see if migrating from one project to another could result in bigger centrality of the developer in the newly joined project.

Due to the little studies of developer migration at the level of software ecosystems, we started to study the effect of the intake, retention and loss of developers at the level of individual projects of the GNOME ecosystem. For each 6-month activity period we counted the number of *joiners* and *leavers*. We distinguished between *local joiners* to a project (resp. *local leavers*) and *global joiners* (resp. *global leavers*). Local joiners are incoming coders in the considered project that were not active in this project during the preceding 6-month period, but that were involved in some activity in other GNOME projects instead. Global joiners are incoming coders in the considered project that were not active in any of the GNOME projects during the preceding period. A similar definition holds for the local and global leavers.

The formal definition of these metrics is given in Equation 10.1. Let p be a GNOME project, t a 6-month activity period, $t-1$ the previous period, c a coder, *Gnome* the set of GNOME's code projects, and $isDev(c,t,p)$ a predicate which is true if and only if c made a code commit in p during t:

$$\begin{aligned}
&localLeavers(p,t) = \\
&\quad \{c|isDev(c,t-1,p) \wedge \neg isDev(c,t,p) \wedge \exists p_2\ (p_2 \in Gnome \wedge isDev(c,t,p_2))\} \\
&globalLeavers(p,t) = \\
&\quad \{c|isDev(c,t-1,p) \wedge \forall p_2\ (p_2 \in Gnome \Rightarrow \neg isDev(c,t,p_2))\} \\
&localJoiners(p,t) = \\
&\quad \{c|isDev(c,t,p) \wedge \neg isDev(c,t-1,p) \wedge \exists p_2\ (p_2 \in Gnome \wedge isDev(c,t-1,p_2))\} \\
&globalJoiners(p,t) = \\
&\quad \{c|isDev(c,t,p) \wedge \forall p_2\ (p_2 \in Gnome \Rightarrow \neg isDev(c,t-1,p_2))\}
\end{aligned} \tag{10.1}$$

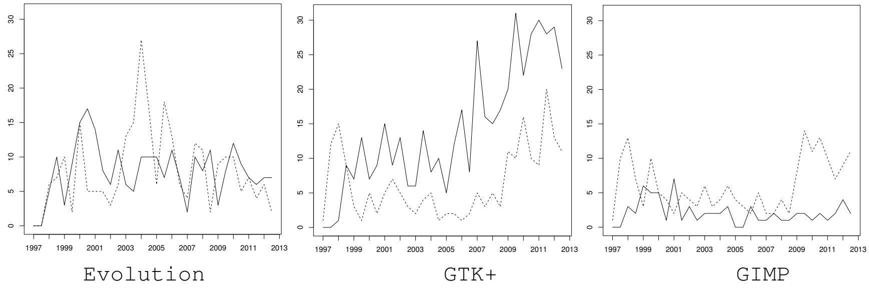


Fig. 10.10: Historical evolution (timeline on x-axis) of the number of local (solid) and global (dashed) joiners (y-axis) for three GNOME projects.

We did not find any general trend, the patterns of intake and loss of coders are highly project-specific. Figure 10.10 illustrates the evolution of the number of local and global joiners for some of the more important GNOME projects (the figures for leavers are very similar). For some projects (e. g. `evolution`) we do not observe a big difference between the number of local and global joiners, respectively. These projects seem to attract new developers both from within and outside of GNOME. Other projects, like GIMP (a popular image manipulation program that can be used and installed separately from other Gnome applications), attract most of its incoming developers from outside GNOME. A third category of projects attracts most of its incoming developers from other GNOME projects. This is the case for GTK+ which can be considered as belonging to the core of GNOME. This observation seems to suggests that libraries, toolkits and auxiliary projects attract more inside developers, while projects that are well-known to the outside world (such as GIMP) attract outside developers.

However, it is also important to measure if the projects that attract developers from the outside of the ecosystem tend to keep those developers inside the project or also "diffuse" them to other projects of GNOME. In order to give an idea of this on the three previously mentioned projects we defined a metrics we called the *collaboration factor* of a project. It represents the percentage of coders contributing to the project and who are also contributing to another project of GNOME. The collaboration factors for Evolution, GIMP and GTK+ are respectively 65.1%, 85% and 94.7%. This leads us to think that while GIMP attracts a lot of people from the outside of GNOME it seems that its community is not integrated into the GNOME community as well as other projects like GNOME. At the opposite, the GTK+ community appears to be more integrated in the GNOME community, which is probably not surprising since GTK+ is the core user interface library which is used by all GNOME end-user programs. It is worthwhile to study this phenomenon in more detail to find empirical evidence of this. One might consider, e. g., concentration of project participants' contributions to projects within the ecosystem which can be measured using inequality indices (cf. [768, 881, 888]). Presence of many developers with highly concentrated contributions would suggest low integration within the community.

10.5 Conclusions

This chapter presented an in-depth analysis of the analogy between natural and OSS ecosystems, from the evolutionary point of view. While there are many similarities between both types of ecosystems a lot of differences can be observed.

From a technical viewpoint, many techniques and models that have been proposed and used in ecology may provide new insights for the study of evolving software ecosystems. Some examples of techniques are the use of phylogenetic trees and cluster dendograms. Some ecological models, such as the dynamic predator-prey model have already been adapted with success in a software evolution setting [155, 500].

Some other models, even after adaptation, appear to give different results when applied to OSS ecosystems. For example, for the GNOME ecosystem there appears to be a much higher degree of collaboration than what is found in many natural ecosystems, and a lower degree of competition. For such collaborative ecosystems, the more recent hologenome theory of evolution that has been proposed to explain the evolution of coral reef ecosystems [732] may perhaps be closer to how software ecosystems evolve, since it considers the evolving organism together with its associated communities, just like a software project co-evolves by the grace of its associated user and developer communities.

Because the traditional biological evolutionary theories are essentially driven by competition between species in a shared resource pool, they are not always readily applicable to explain the dynamics of highly collaborative OSS ecosystems. Other, more business-driven proprietary software ecosystems, such as the app stores for mobile devices, are likely to have a higher degree of competition since all apps struggle for a larger market share in order to increase their profits. The developers of commercial software ecosystems are also remunerated, while contributors to OSS ecosystems often work on a voluntary basis and usually have no direct financial benefits from their involvement.

The main challenge is that historical data of commercial software ecosystems is much harder to obtain, making it difficult to study evolutionary theories on such ecosystems. OSS ecosystems like GitHub and SourceForge do not have this limitation and probably fall somewhere between both extremes, with some amount of competition but also a certain degree of collaboration.

Seen from a complex systems viewpoint, OSS ecosystems seem to be closer to their biological counterpart than business software ecosystems [435]. Commercial ecosystems are typically governed by a decision maker that decides how the ecosystem should evolve, while OSS ecosystems often have a much more flexible decisional structure. Like in biological ecosystems, decisions are taken at the level of individual species (read: projects), with an emergent overall effect on the software ecosystem as a whole.

In the current state of software ecosystems research, it is still too early to make any general conclusions, and much more empirical results are required to understand how one can benefit the most from existing research on natural ecosystems.

Appendices

Appendix A
Emerging trends in software evolution

Alexander Serebrenik, Tom Mens

Software evolution research is a thriving area of software engineering research. Recent years have seen a growing interest in variety of evolution topics, many of which have been covered in this book. Still, a number of research topics has recently emerged and is not covered in the current volume. Without attempting to be complete, this appendix provides an overview of such topics.

Beyond software

The first group of newly investigated research directions expands the idea of "software".

Traditionally, software evolution research has focused on the program code, and to lesser extent on databases [358]. While the topic of co-evolution or coupled evolution between software artefacts and other artefacts produced during software development is an active area of research [297, 943], its application to data-intensive software systems is not trivial [194, 328, 704]. The study of how code-related and data-related artefacts co-evolve is therefore an emerging area of research that is becoming increasingly relevant and challenging, as more and more software applications tend to rely on data, and the datasets that need to be dealt with are rapidly growing [606].

More and more attention goes to collections of projects (called software ecosystems, cf. Chapter 10), as well as to the need to take into account the social dimension, for example through the use of social network analysis (cf. Chapter 6).

Evolving software developed with modeling languages or *domain-specific languages*, necessitates additional insights in model evolution (cf. Chapter 2). A particularly example of domain-specific languages are those used by mechanical and electrical engineers, using languages such as MATLAB and Simulink® [211]. First steps towards addressing the evolution of such models have been taken [14, 213, 214, 710].

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4, © Springer-Verlag Berlin Heidelberg 2014

Spreadsheet applications are another example of artifacts that are not usually considered as software. Nevertheless, they are extremely common in industry, and can be considered as a specific kind of end-user programming [150]. Studying the evolution of such spreadsheet applications therefore constitutes an important emerging trend in software evolution research [56, 391].

While individual software applications studied in the past usually were relatively large, there is a growing attention to evolution of software installed on *embedded devices* [486, 790] such as cars [213, 840] or consumer appliances. Similarly, a better understanding of the evolution of mobile apps for smartphones [78, 814], as well as the app stores in which they reside [77, 367] is becoming increasingly important.

Robots are another example of embedded devices for which software is indispensable. The field of *evolutionary robotics* [127] applies the idea of self-adaptive, autonomous systems (cf. Chapter 7.6) to robotic systems. It involves evolutionary algorithms and metaheuristics (cf. Chapter 4), as well as machine learning techniques (Chapter 5 and Chapter 6). Moreover, the approach is inspired by biological evolution (cf. Chapter 10).

Finally, verification of software correctness typically involves additional artifacts such as tests or formal models (e. g. finite automata and transition systems). While evolution of software tests has been considered in the past [542, 617], this is much less the case for formal verification techniques. Most of the existing approaches to formal software verification assume that the specification of a software system is fixed and does not change over time. Techniques, mechanisms and theories for incremental verification [200, 288, 924] embrace change as a fact of life: if a program that has been formally verified evolves, how to go about reverifying the evolved program without having to do the full verification again from scratch?

How is software developed?

Newly proposed software development approaches are frequently accompanied by new evolutionary challenges that call for adaptation of existing analysis techniques or design of new ones. For instance, while migration of (legacy) applications has been extensively studied in the literature [358, 378], more research is needed to support migration to different computing paradigms such as cloud computing, parallel computing, multi-core computing, mobile computing and the like. For example, it has been shown that migration from a pseudo-cloud environment to a large-scale cloud environment, a common step in deploying applications on massively parallel processing frameworks (e.g., Hadoop [919]), leads to new challenges [771]. Similarly, migration of software from a general-purpose computer to a device with limited resources such as a smartphone or a gaming console, requires novel techniques [17]. The same is true for migration to many-core and multi-core processors [878], clouds [172, 462, 861] and parallelization in general [22, 879].

Another example of new software development approach calling for novel analysis techniques comes from the area of reverse engineering [590, 617]. For example, while reverse engineering UML sequence diagrams is not new [143, 353, 474, 734], Enterprise JavaBeans interceptors [272] altered the semantics of traditional Java programs and necessitated development of a new technique [733, 767]. Similarly, while GUI reverse engineering has been introduced already in [586] growing popularity of mobile applications called for new reverse engineering techniques [28, 447] aiming at discovery of a comprehensible model of the user interface states and transitions between them.

From a managerial point of view, more insight is needed in the relation between how software evolves and its impact of the technical debt of a software project or organization [146, 522]. By gaining a better understanding in this, better debt management strategies can be adopted.

Socio-technical networks

Another active research trend pertains to the analysis of *socio-technical networks*. This goes beyond studying artifacts used and produced by software stakeholders (e. g. developers and users), and includes studying the stakeholders themselves and their activities and interactions. This trend is illustrated by studies of personality traits of StackOverflow users [87], their gender [882, 883], age [620], locations [751], expertise [700], working rhytms [934] and knowledge sharing strategies [884, 885].

Understanding the differences among individuals involved in software system evolution leads to the conclusion that different stakeholders have different needs. In addition to the traditional focus on technical issues important for software developers, maintainers, architects [70, 378] and quality assessors (Chapter 3), recent work also pertains to release engineers [6, 639, 808], legal advisors [246, 300, 309], user interface designers [687] and translators [886]. Moreover, individuals developing software are not necessarily trained as software engineers: e. g. statistical applications in R [321] are often being developed by statisticians and data analysts rather than software engineers [885]. The needs of these individuals are likely to be different from those of traditional software engineers.

Interdisciplinary research

Similar to how software evolution research has been influenced by research in social networks, inspiration can be drawn from techniques originating from a wide variety of different disciplines, including biology (Chapter 10 of this book, but also [867]), bibliometrics [164], economics [768, 881], linguistics [889], psychology [87, 267], seismology [370] and complex systems [626].

In their turn, software evolution studies have inspired recent research in green computing and power consumption modeling [403], bibliometrics [887] and recruitment [163].

Reproducible research

To conclude this chapter, we would like to stress that empirical research in software evolution (and in software engineering in general), is in need of processes, tools, techniques that facilitate reproduction and replication of studies. While reproducibility of research is one of the main principles of the scientific method, in practice there is still a long way to go. There are many reasons why replication of research studies in software evolution is notoriously hard: because the datasets used in the original study are difficult to obtain, because important implementation details or environmental settings of the study are not documented, because the proper tools for replicating the study or no longer available, because the statistical findings are not significant or are not correctly interpreted, and many other factors can be cited.

Gonzalez-Barahona et al. have analyzed important aspects that render reproduction of empirical software engineering studies difficult or impossible [340]. Ghezzi and Gall [324] have proposed the SOFAS framework to facilitate reproduction of software mining studies, and used it to try to reproduce 88 empirical studies. Of these, only 25 studies could be fully replicated, and 27 studies partially. Dit et al. [254] proposed the TraceLab component library, a research framework that supports and facilitates reproducible research in software evolution. While these are important steps in the right direction, more research along the same lines is needed in order to make the reproduction and replication of empirical research in software evolution commonplace, thereby contributing to the trustworthiness and reliability of empirical software engineering as a scientific discipline.

Appendix B
List of acronyms

ACM Association for Computing Machinery (`www.acm.org`)

ACRA Autonomic Computing Reference Architecture

ADG Attribute Dependency Graph

ALM Application Lifecycle Management

AOP Aspect-Oriented Programming

API Application Programmer Interface

ASP Active Server Pages

AST Abstract Syntax Tree

AUC Area Under the ROC Curve

BCR Benefit-Cost Ratio

BPMN Business Process Modeling Notation

CFG Control Flow Graph

CGI Common Gateway Interface

CMMI Capability Maturity Model Integration (`www.sei.cmu.edu/cmmi`)

CSS Cascading Style Sheet

DAS Dynamically Adaptive System

DOM Document Object Model

DPP Developer-Project-Property

DSL Domain-Specific Language

DTD Document Type Definition

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4, © Springer-Verlag Berlin Heidelberg 2014

EJB Enterprise JavaBeans

EMF Eclipse Modeling Framework (www.eclipse.org/modeling/emf)

FLOSS Free/Libre Open Source Software

FODA Feature-Oriented Domain Analysis

GA Genetic Algorithm

GEF Graphical Editing Framework (www.eclipse.org/gef)

GMF Graphical Modeling Framework (www.eclipse.org/modeling/gmf)

GP Genetic Programming

GNU GNU's not Unix (recursive acronym, www.gnu.org)

GPL General Public Licence

GUI Graphical User Interface

HC Hill Climbing

HMM Hidden Markov Model

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IaaS Infrastructure-as-a-Service

IDE Integrated Development Environment

IEC International Electrotechnical Commission (www.iec.ch)

IEEE Institute of Electrical and Electronics Engineers (www.ieee.org)

IR Information Retrieval

ISO International Organisation for Standardisation (www.iso.org)

JSP Java Server Pages

LDA Latent Dirichlet Allocation

LOC Lines of Code. SLOC = Source Lines Of Code. KLOC = thousand (kilo) lines of code

LSI Latent Semantic Indexing

MAP Mean Average Precision

MDA Model-Driven Architecture

MDD Model-Driven Development

MDE Model-Driven Engineering

MDWE Model-Driven Web Engineering

MI Maintainability Index

MIAC Model Identification Adaptive Control

MOF Meta-Object Facility (www.omg.org/mof)
EMOF = Essential MOF
CMOF = Complete MOF

MRAC Model Reference Adaptive Control

MTBE Model Transformation By Examples

NFR Non-Functional Requirements, also known as quality attributes or 'ilities'

NLP Natural Language Processing

NLTK Natural Language Toolkit

OMG Object Management Group (www.omg.org)

OS Operating System

OSLC Open Services for Lifecycle Collaboration (open-services.net)

OSS Open Source Software

PaaS Platform-as-a-Service

PCA Principal Component Analysis

PHP Recursive acronym for PHP Hypertext Processor

PLA Product Line Architecture

PCI-DSS Payment Card Industry Data Security Standard, a set of principles for ensuring cardholder information is protected by those accepting, e.g., VISA and Mastercard as payment.
www.pcisecuritystandards.org/security_standards

PSO Particle Swarm Optimization

QMOOD Quality Model for Object-Oriented Design

QoS Quality of Service

QI Quality Index

QVT Query -View-Transformation

RCA Root Cause Analysis

RE Requirements Engineering

REKB Requirements Engineering Knowledge Base

REP Requirements Evolution Problem

RIA Rich Internet Application

RML Requirements Modeling Language

ROC Receiver Operating Characteristic

RWR Random Walk with Restart

SA Simulated Annealing

SaaS Software-as-a-Service

SAS Self-Adaptive Software

SAP Self Adaptation Problem

SBSE Search-Based Software Engineering

SCA Service Component Architecture

SDK Software Development Kit

SEI Carnegie Mellon Software Engineering Institute (www.sei.cmu.edu)

SLA Service Level Agreement

SOA Service-Oriented Architecture

SOAP Simple Object Access Protocol

SPL Software Product Line

SPLE Software Product Line Engineering

SQALE Software Quality Assessment based on Lifecycle Expectations

SQUALE Software QUALity Enhancement [619]

SQuaRE Systems and Software Quality Requirements and Evaluation

SRS Software Requirements Specification, typically as defined in the IEEE-830 standard [421]

SVG Scalable Vector Graphics

SVM Support Vector Machine

UML Unified Modeling Language (www.uml.org)

URI Uniform Resource Identifier

VSM Vector Space Model

WWW World Wide Web

WSDL Web Services Description Language

XML eXtensible Markup Language

Appendix C
Glossary of Terms

This appendix contains a glossary of terms and definitions that have been introduced and used in the various chapters contributing to this book.

as-is utility According to Boehm [122], the extent to which the as-is software can be used (i.e. ease of use, reliability and efficiency).

conformance A relationship between models and metamodels. A model conforms to its metamodel if it obeys the syntactic rules defined by the metamodel.

content model A model describing the business and data objects of a web system, including their properties and relationships.

coupled evolution A coupled evolution is a triple (μ, μ', m) of the original metamodel μ, the evolved metamodel μ', and migration m, a partial function from the extension of μ to the extension of μ'.

dynamic software systems Software systems whose operation is especially affected by uncertainty, that is their requirements and execution environments may change rapidly and unpredictably [646].

ecology The scientific study of the interactions that determine the distribution and abundance of living organisms.

ecosystem The physical and biological components of an environment considered in relation to each other as a unit.

efficiency According to Boehm [122], the optimum use of system resources during correct execution.

evolution A process of *progressive*, for example beneficial, change in the attributes of the evolving entity or that of one or more of its constituent elements. What is accepted as progressive must be determined in each context. It is also appropriate to apply the term evolution when long-term change trends are beneficial even though isolated or short sequences of changes may appear degenerative. For example, an entity or collection of entities may be said to be evolving if their value or fitness

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4, © Springer-Verlag Berlin Heidelberg 2014

is increasing over time. Individually or collectively they are becoming more meaningful, more complete or more adapted to a changing environment. (Chapter 1 of [553].)
Alternative definition: The application of software maintenance actions with the goal of generating a new operational version of the system that guarantees its functionalities and qualities, as demanded by changes in requirements and environments [170, 598].

extension The extension of a software language is the set of utterances that are syntactically correct with respect to the software language.

feature Distinguishable characteristics of a concept (e. g. component, system, etc.) that are relevant to some stakeholder of the concept[207].

free software A popular mode of software distribution as a common good in which users can access, modify and re-distribute the code, under the terms of the license and some parts (e.g., notices) that should not been modified.

flexibility Acording to McCall [574], the ability to make changes required as dictated by the business. According to Boehm [122], the ease of changing the software to meet revised requirements.

frequency The number of occurrences of a change event per unit of time that will require the evolution of the system.

intensional definition An intensional definition of a software language defines the rules to check whether an utterance is syntactically correct with respect to the software language.

interoperability Acording to McCall [574], the extent or ease to which software components work together.

maintainability According to McCall [574], the ability to find and fix a defect. According to Boehm [122], the ease of identifying what needs to be changed as well as the ease of modification and retesting.

maintenance According to the ISO Standard 12207 [423], the software product undergoes modification to code and associated documentation due to a problem or the need for improvement. The objective of software maintenance is to modify the existing software while preserving its integrity.
According to the IEEE Standard 1219 [420], *software maintenance* is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment. In the ISO/IEC Standard 14764 [424], maintenance is further subdivided into four categories:

Perfective maintenance is any modification of a software product after delivery to improve performance or maintainability.
Corrective maintenance is the reactive modification of a software product performed after delivery to correct discovered faults.

Adaptive maintenance is the modification of a software product performed after delivery to keep a computer program usable in a changed or changing environment.

Preventive maintenance refers to software modifications performed for the purpose of preventing problems before they occur. This type of maintenance, that does not alter the system functionality, is also referred to as *anti-regressive work*.

metamodel An intensional definition of a modeling language. It specifies the abstract syntax of the language.

model An abstract specification of a part of a software system. A model is an utterance of a modeling language.

model migration A transformation that transforms a model that conforms to the old version of the metamodel to the new version of the metamodel.

modeling language A software language to specify models. Its abstract syntax is defined by a metamodel, and its semantics usually defines how to map models to programs.

navigation model A model describing the user interactions of a web system (e.g., navigation through links and form submission).

off-line software evolution The process of modifying a software system through actions that require intensive user intervention and imply the interruption of the system operation.

open source software Software of which the source code is available for users and third parties to be inspected and used. It is made available to the general public with either relaxed or non-existent intellectual property restrictions. It is generally used as a synonym of free software even though the two terms have different connotations. *Open* emphasises the accessibility to the source code, while *free* emphasises the freedom to modify and redistribute under the terms of the original license.

portability According to McCall [574], the ability to transfer the software from one environment to another. According to Boehm [122], the ease of changing software to accommodate a new environment, or the extent to which the software will work under different computer configurations (i.e. operating systems, databases etc.).

presentation model A model describing the layout and the look and feel of a web systems interface, as well as the widgets that enable user interactions.

programming language A software language to specify executable programs.

reliability According to McCall [574], the extent to which the system fails. According to Boehm [122], the extent to which the software performs as required, i. e. the absence of defects.

re-engineering According to [181], *re-engineering* is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Re-engineering generally includes some form of *reverse engineering* (to achieve a more abstract description) followed by some form of *forward engineering* or *restructuring*. This may include modifications with respect to new requirements not met by the original system.

refactoring According to [301], *refactoring* is [the process of making] a change to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour. If applied to programs, we talk of program refactoring. If applied to models, we talk of model refactoring.

reusability Acording to McCall [574], the ease of using existing software components in a different context.

reverse engineering According to [181], *reverse engineering* is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. Reverse engineering generally involves extracting design artefacts and building or synthesizing abstractions that are less implementation-dependent.

rich internet applications (RIA) Web applications that are characterized by a user experience that is highly interactive and responsive so that they can rival the experience that desktop software applications can offer.

runtime software evolution The process of modifying a software system through tasks that require minimum human intervention and are performed while the system executes.

search-based software engineering The application of meta-heuristic search techniques like genetic algorithms to software engineering problems.

self-adaptive software systems are software applications designed to adjust themselves, at runtime, with the goal of satisfying requirements that either change while the system executes or depend on changing environmental conditions.

software ecosystem We provide two alternative definitions:

1. A collection of software projects which are developed and evolve together in the same environment. [547]
2. A set of actors functioning as a typically is interconnected with institutions, such as standardisation organisations, unit and interacting with a shared market for software and services, together with the relationships among them. [433, 434]

software engineering We provide two alternative definitions:

1. The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines. [635]
2. The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. [419]

software evolution See **evolution**.

software language A general term for artificial languages that are used to develop software. A software language consists of an abstract and concrete syntax as well as a semantics.

software product line A software product line aims to support the development of a family of similar software products from a common set of shared assets.

software repository A kind of database, file system or other kind of repository in which historical information of a software system is stored. The repository may be used to store source code, executable code, bug reports, change requests, documentation or any other type of software-related artefact for which it is useful to store historical information.

static website A website whose content is primarily based on HTML and that offers no dynamic features such as content generation.

testability According to McCall [574], the ability to validate the software requirements. According to Boehm [122], the ease of validation that the software meets the requirements.

uncertainty The reliability with which it is possible to characterize the occurrence of changes in requirements and execution environments.

understandability According to Boehm [122], the extent to which the software is easily comprehended with regard to purpose and structure.

unstructured data Data which does not have clear, semantically overt, easy-for-a-computer structure. It is the opposite of structured data, the canonical example of which is a relational database, of the sort companies usually use to maintain product inventories and personnel records [557].

usability According to McCall [574] and Boehm [122], the ease of use.

version A snapshot of a certain software system at a certain point in time. Whenever a change is made to the software system, a new version is created.

version history The historical collection of all versions of a software system and their relationships.

version repository A software repository containing different versions of the software over time.

web application A system that is based on web technologies and access via a web browser.

Appendix D
Resources

This appendix lists a number of additional resources for those readers that wish to gain more detailed information on particular topics addressed in this book.

Books

Over the years, many books have been published on the topics of software maintenance, software evolution and related areas. It is not our intent to provide a complete list of such books here, especially since many of the older books are either outdated or out of print. Therefore, we have preferred to present in reverse chronological order our personal, subjective, list of recent books (less than 10 years old) that we believe to be of relevance for the interested reader.

- *Reverse Engineering – Recent Advances and Applications* [827]. This book presents applications of reverse engineering in the software engineering, shape engineering, medical and life sciences application domains. The 12 contributed chapters provide the state-of-the-art in reverse engineering techniques, tools, and use-cases, as well as an overview of open challenges for reverse engineering researchers.
- *Software Engineering – The Current Practice* [706]. This book presents recent developments in object-oriented software engineering, including techniques to cope with software changes in iterative, agile, and traditional software development.
- *Software Maintenance Success Recipes* [711] identifies success recipes in effective software maintenance projects based on in-depth analysis of more than 200 real-world projects. This includes creating a robust management infrastructure, ensuring that proper resources are available, establishing a user support structure, conducting a meaningful measurement program, and determining the best way and time to retire software systems.

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4, © Springer-Verlag Berlin Heidelberg 2014

- *Making Software: What Really Works, and Why We Believe It* [666] discusses such software evolution topics as software measurement and quality, and bug prediction.
- The *Encyclopedia of Software Engineering* [496] contains several chapters specifically dedicated to software evolution and software maintenance.
- *Software Evolution* [592]. This book can be considered as the predecessor of the book you are currently reading, but it is complementary to it. The chapters contained in both books focus on different research topics related to software evolution. Together, they cover a very large part of software evolution research.
- *Effective Software Maintenance and Evolution – A Reuse-Based Approach* [437]. Stan Jarzabek explores tools for program analysis, reverse engineering, and reengineering in-depth and explains the best ways to deploy them. It also discusses the role of XML, software components, object technology, and metaprogramming in improving systems maintenance, as well as how to align software with business goals through strategic maintenance.
- *Software Maintenance Management: Evaluation and Continuous Improvement* [45]. This book explores the domain of software maintenance management and provides road maps and maturity models for improving software maintenance organizations, aligned with the maturity models of CMMI and ISO 15504.
- *Software Evolution and Feedback: Theory and Practice* [553]. This book scientifically explores what software evolution is and why it is inevitable. It addresses the phenomenological and technological underpinnings of software evolution, and it explains the role of feedback in software development and maintenance.
- *Refactoring Databases: Evolutionary Database Design* [29]. This book applies the ideas of refactoring to database schemas.
- *Working Effectively with Legacy Code* [287]. Michael Feathers shows how to deal with the "testing versus reengineering" dilemma. Before you reengineer you need a good suite of regression tests to ensure that the system does not break. However, the design of a system that needs reengineering typically makes testing very difficult and would benefit from reengineering.
- *Software Evolution with UML and XML* [935]. This collection of contributed chapters addresses some of the potential applications of UML and XML in the field of software evolution.
- *Refactoring to patterns* [458]. Joshua Kerievsky's book explains how to introduce design patterns in your code, by listing typical code smells and ways to refactor them away. An appealing way to teach reluctant designers how to clean up their code base.

It is also worthwhile to mention the 1985 book *Program Evolution: Processes of Software Change* written by Lehman and Belady, one of the very first books that has been published on the topic of software evolution [508]. Although it is no longer available in print, an electronic version of the book may be downloaded for free on the internet from `http://informatique.umons.ac.be/genlog/BeladyLehman1985-ProgramEvolution.pdf`.

Journals

The only dedicated international journal on the topic of software evolution and software maintenance is Wiley's *Journal on Software: Evolution and Process* (JSEP). Some other international journals in which scientific articles on software maintenance and evolution are published occasionally are (ordered by publisher):

ACM

- TOPLAS: *Transactions on Programming Languages and Systems*
- TOSEM: *Transactions on Software Engineering and Methodology*
- TWEB: *Transactions on the Web*

Elsevier

- JSS: *Journal on Systems and Software*

Inderscience

- IJWET: *International Journal of Web Engineering and Technology*

IEEE

- TSE: *Transactions on Software Engineering*

Kluwer

- ASE: *Automated Software Engineering*

Rinton Press

- JWE: *Journal of Web Engineering*

Springer

- SoSyM: *Software and Systems Modeling*
- EMSE: *Empirical Software Engineering*

Wiley

- SPE: *Software: Practice and Experience*

Standards

The following ISO/IEC approved standards are very relevant in the field of software evolution, though some of them may be a bit outdated with respect to the state-of-the-art in research:

- Standard 25000 on "Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE)" [426]
- Standard 9001 on "Quality Management Systems - Requirements" [427]

- Standard 14764 on "Software Maintenance" [424]
- Standard 12207 (and its amendments) on "Information Technology - Software Life Cycle Processes" [423]
- Standard 9126 on "Information technology - Software product evaluation - Quality characteristics and guidelines for their use" [422]

Events

Many events are organised each year around the themes of software evolution, software maintenance and reengineering, or related areas. We list only the most well-known international events here.

Conferences

A number of international conferences are organised each year, devoted to the topics of software evolution, software maintenance, reverse engineering and re-engineering:

CSMR *European Conference on Software Maintenance and Reengineering*
http://www.csmr.eu

ICPC *International Conference on Program Comprehension*
http://www.program-comprehension.org

ICSM *International Conference on Software Maintenance*[1]
http://conferences.computer.org/icsm

SCAM International Working Conference on *Source Code Analysis and Manipulation*
http://www.ieee-scam.org/

SEAMS International *Symposium on Software Engineering for Adaptive and Self-managing Systems*
http://www.self-adaptive.org

SLE International Conference on *Software Language Engineering*
http://planet-sl.org/sle2013

MSR Working Conference on *Mining Software Repositories*
http://www.msrconf.org

WCRE *Working Conference on Reverse Engineering*
http://www.reengineer.org/wcre

[1] Starting from 2014 the conference will be called *International Conference on Software Maintenance and Evolution (ICSME).*

Beyond these, many other international conferences in computer science are being organised that include contributions on software evolution. We do not list those conferences here as there are too many of them. We invited the reader to look at the references at the end of this book to find out which conferences may be relevant.

Workshops

A wide range of international workshops are organised each year on the topic of software evolution or a subdomain thereof:

EVOL International workshop on software evolution organised by the ERCIM Working Group on Software Evolution

IWPSE *International Workshop on Principles of Software Evolution*

ME International Workshop on *Models and Evolution*

MESOCA *International Symposium on Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*

RE@Runtime International Workshop on *Requirements at RunTime*. Usually co-located with the International Conference on Requirements Engineering.

SATTOSE *Seminar on Advanced Tools and Techniques on Software Evolution*

WSE International workshop on *Web Site Evolution*

Appendix E
Datasets

Empirical software evolution research very regularly relies on the use of datasets, containing detailed information about the history of many (typically open source) software projects. Below, we list the most prominent datasets we have found in the research literature.

Bug Prediction Dataset A collection of models and metrics of software systems and their histories. The goal of this dataset is to allow people to compare different bug prediction approaches and to evaluate whether a new technique is an improvement over existing ones.
`bug.inf.usi.ch`

COMETS - Code metrics time series dataset A dataset of source code metrics collected from several Java-based systems to support empirical studies on source code evolution.
`java.llp.dcc.ufmg.br/comets`

DaCapo - Benchmark Suite A benchmark tool suite intended for Java benchmarking by the programming language, memory management and computer architecture communities. It consists of a set of open source, real world applications with non-trivial memory loads.
`www.dacapobench.org`

Edapt Metamodel Histories A set of metamodel evolution histories that have been reverse engineered from snapshots using the operator-based tool Edapt.

```
git.eclipse.org/c/edapt/org.eclipse.emf.edapt.git
     /tree/tests/org.eclipse.emf.edapt.tests/data
```

FLOSSMetrics A database, provided by the FLOSSMetrics project, containing data and metrics about open source software development coming from several thousands of software projects.
`melquiades.flossmetrics.org`

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4, © Springer-Verlag Berlin Heidelberg 2014

FLOSSMole A dataset, provided by the FLOSSMole project, containing over 1TB of data and metrics about open source software projects coming from several thousands of software projects [408].
`flossmole.org`

GHTorrent GHTorrent monitors the Github public event time line. It retrieves the contents and dependencies of each event and stores this in a database. The data collected for every couple of months is released as downloadable archives through the Bittorent protocol.
`ghtorrent.org`

Helix A compilation of release histories of a number of non-trivial Java open source software systems. It has been developed to assist researchers in the field of empirical software engineering with a focus on software evolution.
`www.ict.swin.edu.au/research/projects/helix/`

Ohloh A public directory that indexes (as opposed to hosting) free and open source software projects and provides analytics and search services. By connecting to project source code repositories, analyzing the project's history and attributing commits to contributors, Ohloh can report on the composition and activity of project code bases.
`www.ohloh.net`

PROMISE - PRedictOr Models In Software Engineering This website makes data available for a compilation of release histories of a number of non-trivial Java open source software systems. It has been developed to assist researchers in the field of empirical software engineering with a focus on software evolution and predictive models.
`code.google.com/p/promisedata/`

Qualitas Corpus A curated collection of open source Java software systems intended to be used for empirical studies of code artefacts. The primary goal is to provide a resource that supports reproducible studies of software.
`qualitascorpus.com`

Sourcerer An ongoing research project at the University of California, Irvine aimed at exploring the open source phenomenon through the use of code analysis. Sourcerer's Managed Repository stores local copies of projects aggregated from numerous open source repositories. The repository contains 18,000 Java projects downloaded from Apache, Java.net, Google Code and SourceForge.
`sourcerer.ics.uci.edu/repository.html`

TraceLab A workbench for designing, running and sharing traceability experiments using a visual modeling environment [254]. Along with the workbench, the authors provide datasets containing issues, associated functionality (methods), queries and execution traces.
`www.coest.org/index.php/tracelab`

References

[1] A. Abran, R. Al Qutaish, J. Desharnais, and N. Habra, *ISO-based Models to Measure Software Product Quality*. Institute of Chartered Financial Analysts of India (ICFAI) - ICFAI Books, 2007. *cited on page 67*

[2] R. Abreu and R. Premraj, "How developer communication frequency relates to bug introducing changes," in *Joint Int'l Workshop on Principles of software evolution (IWPSE) and ERCIM software evolution workshop*. ACM, 2009, pp. 153–158. *cited on page 307*

[3] P. Achananuparp, I. N. Lubis, Y. Tian, D. Lo, and E.-P. Lim, "Observatory of trends in software related microblogs," in *Int'l Conf. Automated Software Engineering*, 2012, pp. 334–337. *3 citations on pages 163, 165, and 181*

[4] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire, "Reverse engineering architectural feature models," in *European Conf. Software Architecture*, ser. ECSA'11. Berlin, Heidelberg: Springer, 2011, pp. 220–235. *cited on page 287*

[5] M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle, "Feature model differences," in *Int'l Conf. Advanced Informations Systems Engineering*, ser. CAiSE'12. Berlin, Heidelberg: Springer, 2012, pp. 629–645. *cited on page 280*

[6] B. Adams, C. Bird, F. Khomh, and K. Moir, "Int'l workshop on release engineering (RELENG 2013)," in *Int'l Conf. Software Engineering*, 2013, pp. 1545–1546. *cited on page 331*

[7] S. N. Ahsan, J. Ferzund, and F. Wotawa, "Automatic software bug triage system (BTS) based on Latent Semantic Indexing and Support Vector Machine," in *Int'l Conf. Software Engineering Advances*, 2009, pp. 216–221. *cited on page 153*

[8] S. A. Ajila and A. B. Kaba, "Using traceability mechanisms to support software product line evolution," in *Proc. of 2004*, 2004, pp. 157–162. *cited on page 289*

[9] ——, "Evolution support mechanisms for software product line process," *J. Systems and Software*, vol. 81, no. 10, pp. 1784–1801, 2008. *cited on page 278*

[10] F. Akiyama, "An Example of Software System Debugging," in *IFIP Congress (2)*, 1971, pp. 353–359. *cited on page 69*

[11] J. Al Dallal and L. C. Briand, "A precise method-method interaction-based cohesion metric for object-oriented classes," *ACM Trans. Software Engineering and Methodology*, vol. 21, no. 2, pp. 8:1–8:34, Mar. 2012. *cited on page 132*

[12] H. Al-Kilidar, K. Cox, and B. Kitchenham, "The use and usefulness of the ISO/IEC 9126 quality standard," in *IEEE Int'l Symp. Empirical Software Engineering (ISESE)*, Nov. 2005, pp. 126–132. *cited on page 203*

[13] R. E. Al-qutaish, "Quality Models in Software Engineering Literature: An Analytical and Comparative Study," *Journal of American Science*, vol. 6, no. 3, pp. 166–175, Feb. 2010. *cited on page 74*

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4, © Springer-Verlag Berlin Heidelberg 2014

[14] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson, "Models are code too: Near-miss clone detection for Simulink models," in *Int'l Conf. Software Maintenance*, 2012, pp. 295–304. *cited on page 329*

[15] R. Alarcón and E. Wilde, "RESTler: Crawling RESTful services," in *Int'l Conf. World Wide Web*, Apr. 2010, pp. 1051–1052. *2 citations on pages 213 and 224*

[16] C. E. Alchourrón, P. Gärdenfors, and D. Makinson, "On the Logic of Theory Change: Partial Meet Contraction and Revision Functions," *Journal of Symbolic Logic*, vol. 50, no. 2, pp. 510–530, 1985. *cited on page 29*

[17] N. Ali, W. Wu, G. Antoniol, M. Di Penta, Y.-G. Guéhéneuc, and J. H. Hayes, "MoMS: Multi-objective miniaturization of software," in *Int'l Conf. Software Maintenance*, 2011, pp. 153–162. *3 citations on pages 117, 136, and 330*

[18] R. Ali, F. Dalpiaz, P. Giorgini, and V. E. S. Souza, "Requirements Evolution: From Assumptions to Reality," in *Int'l Conf. Exploring Modeling Methods in Systems Analysis and Design*, 2011, pp. 1–10. *2 citations on pages 16 and 22*

[19] E. Allen and C. Seaman, "Likert scales and data analyses," *Quality Progress*, vol. July, 2007. *cited on page 180*

[20] A. A. Almonaies, M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Towards a framework for migrating web applications to web services," in *Conf. Center for Advanced Studies on Collaborative Research*, Nov. 2011, pp. 229–241. *3 citations on pages 212, 217, and 224*

[21] A. A. Almonaies, J. R. Cordy, and T. R. Dean, "Legacy system evolution towards service-oriented architecture," in *Int'l Workshop on SOA Migration and Evolution (SOAME)*, Mar. 2010, pp. 53–62. *2 citations on pages 217 and 218*

[22] S. M. Alnaeli, A. Alali, and J. I. Maletic, "Empirically examining the parallelizability of open source software system," in *Working Conf. Reverse Engineering*, 2012, pp. 377–386. *cited on page 330*

[23] T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. E. Shore, "Software Requirements for the A-7E Aircraft," Naval Research Laboratory, Tech. Rep. NRL/FR/5530-92-9194, Aug. 1992. *cited on page 11*

[24] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Int'l Conf. Software Maintenance*. IEEE Computer Society, 2010, pp. 1–10. *cited on page 80*

[25] V. Alves, R. Gheyi, and T. Massoni, "Refactoring product lines," in *Int'l Conf. Generative Programming*, 2006, pp. 201–210. *3 citations on pages 281, 285, and 292*

[26] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "Reverse engineering finite state machines from rich internet applications," in *Working Conf. Reverse Engineering*, Oct. 2008, pp. 59–68. *cited on page 214*

[27] D. Amalfitano, "Reverse engineering and testing of rich internet applications," Ph.D. dissertation, Universitá degli Studi di Napoli Federico II, Nov. 2011. *2 citations on pages 214 and 227*

[28] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A GUI crawling-based technique for android mobile application testing," in *ICST Workshops*, 2011, pp. 252–261. *cited on page 331*

[29] S. W. Ambler and P. J. Sadalage, *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley, 2006. *2 citations on pages 221 and 344*

[30] D. J. Anderson, *Kanban*. Blue Hole Press, 2010. *cited on page 14*

[31] S. Anderson and M. Felici, "Controlling Requirements Evolution: An Avionics Case Study," in *Int'l Conf. Computer Safety, Reliability and Security*, F. Koornneef and M. van Der Meulen, Eds., 2000. *cited on page 12*

[32] ——, "Requirements Evolution: From Process to Product Oriented Management," in *Int'l Conf. Product Focused Software Process Improvement*, 2001, pp. 27–41. *cited on page 12*

[33] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu, "Statistical debugging using latent topic models," in *European Conf. Machine Learning*, 2007, pp. 6–17. *cited on page 155*

[34] N. Anquetil, U. Kulesza, R. Mitschke, A. Moreira, J.-C. Royer, A. Rummler, and A. Sousa, "A model-driven traceability framework for software product lines," *Software and Systems Modeling*, vol. 9, no. 4, pp. 427–451, Sep. 2010. *cited on page 289*

[35] A. I. Antón, "Goal-Based Requirements Analysis," in *Int'l Conf. Req. Engineering*, 1996, pp. 136–144. *cited on page 10*

[36] A. I. Antón and C. Potts, "Functional paleontology: system evolution as the user sees it," in *Int'l Conf. Software Engineering*, 2001, pp. 421–430. *cited on page 11*

[37] G. Antoniol, M. Di Penta, and M. Neteler, "Moving to smaller libraries via clustering and Genetic Algorithms," in *European Conf. Software Maintenance and Reengineering*. IEEE Computer Society, 2003, pp. 307–316. *2 citations on pages 116 and 117*

[38] G. Antoniol, M. Di Penta, and M. Zazzara, "Understanding web applications through dynamic analysis," in *Int'l Workshop Program Comprehension*, Jun. 2004, pp. 120–129. *cited on page 224*

[39] G. Antoniol, J. Huffman Hayes, Y.-G. Guéhéneuc, and M. Di Penta, "Reuse or rewrite: Combining textual, static, and dynamic analyses to assess the cost of keeping a system up-to-date," in *Int'l Conf. Software Maintenance*, 2008, pp. 147–156. *cited on page 162*

[40] M. Antwerp and G. Madey, "Advances in the SourceForge research data archive (SRDA)," in *OSS*, 2008. *2 citations on pages 191 and 194*

[41] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Int'l Conf. Software Engineering*, 2006, pp. 361–370. *2 citations on pages 124 and 153*

[42] S. Apel, W. Scholz, C. Lengauer, and C. Kästner, "Detecting dependences and interactions in feature-oriented design," in *Int'l Symp. Software Reliability Engineering*, 2010, pp. 161–170. *cited on page 286*

[43] J. Appelo, "Twitter top 100 for software developers," http://www.noop.nl/2009/02/twitter-top-100-for-software-developers.html, last accessed: 20 September 2003. *cited on page 182*

[44] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger, "Océano - SLA Based Management of a Computing Utility," in *IFIP/IEEE Int'l Symp. Integrated Network Management*. IEEE, 2001, pp. 855–868. *2 citations on pages 250 and 252*

[45] A. April and A. Abran, *Software Maintenance Management: Evaluation and Continuous Improvement*. Wiley, 2008. *cited on page 344*

[46] J. Aranda, S. M. Easterbrook, and G. V. Wilson, "Requirements in the wild: How small companies do it," in *Int'l Conf. Req. Engineering*, 2007. *cited on page 15*

[47] R. Arditi and L. R. Ginzburg, "Coupling in predator-prey dynamics: Ratio-dependence," *J. Theoretical Biology*, vol. 139, pp. 311–326, 1989. *cited on page 299*

[48] K. Astrom and B. Wittenmark, *Adaptive Control*. Addison-Wesley, 1995. *cited on page 247*

[49] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Int'l Conf. Software Engineering*, 2010, pp. 95–104. *cited on page 162*

[50] V. Atluri, U. Cakmak, R. Lee, and S. Varanasi, "Making smartphones brilliant: Ten trends," McKinsey & Company, Tech. Rep. 20, Jun. 2012. *cited on page 216*

[51] M. Azuma, "Software Products Evaluation System: Quality Models, Metrics and Processes – International Standards and Japanese Practice," *Information and Software Technology*, vol. 38, no. 3, pp. 145–154, 1996. *cited on page 67*

[52] U. Amann, N. Bencomo, B. H. C. Cheng, and R. B. France, Eds., *Models@run.time*, ser. Dagstuhl Seminar, no. 11481, 2011. *2 citations on pages 255 and 263*

[53] A. Bacchelli, A. Cleve, M. Lanza, and A. Mocci, "Extracting structured data from natural language documents with island parsing," in *Int'l Conf. Automated Software Engineering*, 2011, pp. 476–479. *2 citations on pages 146 and 155*

[54] A. Bacchelli, M. D'Ambros, and M. Lanza, "Extracting source code from e-mails," in *Int'l Conf. Program Comprehension*, 2010, pp. 24–33. *cited on page 154*

[55] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Int'l Conf. Software Engineering*, 2010, pp. 375–384. *cited on page 154*

[56] S. Badame and D. Dig, “Refactoring meets spreadsheet formulas,” in *Int’l Conf. Software Maintenance*, 2012, pp. 399–409. *cited on page 330*

[57] R. Baeza-Yates and B. Ribeiro-Neto, *Modern information retrieval*. ACM Press, 1999, vol. 463. *cited on page 147*

[58] R. Baggen, K. Schill, and J. Visser, “Standardized code quality benchmarking for improving software maintainability,” in *Int’l Workshop Software Quality and Maintainability*, 2010. *cited on page 80*

[59] E. Bagheri and D. Gasevic, “Assessing the Maintainability of Software Product Line Feature Models using Structural Metrics,” *Software Quality Journal*, vol. 19, no. 3, pp. 579–612, 2011. *cited on page 77*

[60] S. K. Bajracharya and C. V. Lopes, “Mining search topics from a code search engine usage log,” in *Int’l Conf. Mining Software Repositories*, 2009, pp. 111–120. *cited on page 155*

[61] ——, “Analyzing and mining a code search engine usage log,” *J. Empirical Software Engineering*, pp. 1–43, Sep. 2010. *cited on page 155*

[62] T. Bakota, P. Hegedűs, G. Ladányi, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy, “A Cost Model Based on Software Maintainability,” in *Int’l Conf. Software Maintenance*, 2012, pp. 316–325. *2 citations on pages 67 and 87*

[63] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy, “A Probabilistic Software Quality Model,” in *Int’l Conf. Software Maintenance*. IEEE Computer Society, 2011, pp. 368–377. *4 citations on pages 66, 67, 69, and 81*

[64] A. Bakun, “Wasp-waist populations and marine ecosystem dynamics: Navigating the “predator pit” topographies,” *Progress In Oceanography*, vol. 68, no. 2-4, pp. 271–288, 2006. *cited on page 300*

[65] D. Balasubramanian, T. Levendovszky, A. Narayanan, and G. Karsai, “Continuous migration support for domain-specific languages,” in *The 9th OOPSLA Workshop on Domain-Specific Modeling*, 2009. *cited on page 52*

[66] S. Balasubramanian, R. Desmarais, H. A. Müller, U. Stege, and S. Venkatesh, “Characterizing Problems for Realizing Policies in Self-Adaptive and Self-Managing Systems,” in *Int’l Symp. Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011, pp. 70–79. *cited on page 252*

[67] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya, “A theory of aspects as latent topics,” *ACM SIGPLAN Notices*, vol. 43, no. 10, pp. 543–562, 2008. *cited on page 150*

[68] J. Bansiya and C. Davis, “A Hierarchical Model for Object-Oriented Design Quality Assessment,” *IEEE Trans. Soft. Eng.*, vol. 28, pp. 4–17, 2002. *2 citations on pages 67 and 86*

[69] J. Bansiya and C. G. Davis, “A hierarchical model for object-oriented design quality assessment,” *IEEE Trans. Soft. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002. *2 citations on pages 132 and 134*

[70] O. Barais, A.-F. L. Meur, L. Duchien, and J. L. Lawall, *Software Evolution*. Springer, 2008, ch. Software Architecture Evolution, pp. 233–262. *2 citations on pages x and 331*

[71] M. Barbacci, M. Klein, T. Longstaff, and C. Weinstock, “Quality Attributes,” Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Tech. Rep. CMU/SEI-95-TR-021, 1995. *cited on page 77*

[72] L. Baresi and C. Ghezzi, “The Disappearing Boundary Between Development-Time and Run-Time,” in *Workshop on Future of Software Engineering Research (FoSER)*. ACM, 2010, pp. 17–22. *cited on page 253*

[73] L. Baresi and S. Guinea, “Self-Supervising BPEL Processes,” *IEEE Trans. Soft. Eng.*, vol. 37, no. 2, pp. 247–263, 2011. *2 citations on pages 250 and 253*

[74] D. J. Bartholomew, *Latent variable models and factors analysis*. Oxford University Press, 1987. *cited on page 149*

[75] A. Barua, S. W. Thomas, and A. E. Hassan, “What are developers talking about? An analysis of topics and trends in Stack Overflow,” *J. Empirical Software Engineering*, pp. 1–36, 2012. *2 citations on pages 152 and 154*

[76] V. Basili and B. T. Perricone, “Software Errors and Complexity: An Empirical Investigation,” *Comm. ACM*, vol. 27, pp. 42–52, 1984. *2 citations on pages 7 and 9*

[77] R. C. Basole and J. Karla, "Value transformation in the mobile service ecosystem: A study of app store emergence and growth," *Service Science*, vol. 4, no. 1, pp. 24–41, 2012. *2 citations on pages 303 and 330*

[78] P. Battacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru, "An empirical analysis of bug reports and bug fixing in open source Android apps," in *European Conf. Software Maintenance and Reengineering*. IEEE Computer Society, 2013, pp. 133–143. *2 citations on pages 303 and 330*

[79] B. Baudry and M. Monperrus, "Towards ecology inspired software engineering," INRIA, Tech. Rep. 7952, May 2012. *2 citations on pages 298 and 313*

[80] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *Int'l Conf. Software Engineering*, 2013, pp. 1–10. *cited on page 154*

[81] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: the case of Apache," in *Int'l Conf. Software Maintenance*, 2013. *cited on page 306*

[82] G. Bavota, F. Carnevale, A. De Lucia, M. Di Penta, and R. Oliveto, "Putting the developer in-the-loop: An interactive GA for software re-modularization," in *Int'l Symp. Search Based Software Engineering*, ser. Lect. Notes in Computer Science, vol. 7515. Springer, 2012, pp. 75–89. *3 citations on pages 118, 120, and 136*

[83] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, and F. Palomba, "Supporting extract class refactoring in Eclipse: The ARIES project," in *ICSE*, 2012, pp. 1419–1422. *cited on page 128*

[84] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *Trans. Software Engineering and Methodologies*, 2013. *cited on page 128*

[85] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud, "Pulse: a methodology to develop software product lines," in *Symp. Software Reusability*, 1999, pp. 122–131. *2 citations on pages 274 and 277*

[86] J. Bayer, J. F. Girard, M. Wuerthner, J.-M. DeBaud, and M. Apel, "Transitioning legacy assets to a product line architecture," in *ESEC*, 1999, pp. 446–463. *2 citations on pages 284 and 286*

[87] B. Bazelli, A. Hindle, and E. Stroulia, "On the personality traits of StackOverflow users," in *Int'l Conf. Software Maintenance*, 2013. *cited on page 331*

[88] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2004. *cited on page 128*

[89] S. Becker, T. Goldschmidt, B. Gruschko, and H. Koziolek, "A process model and classification scheme for semi-automatic meta-model evolution," in *Workshop MDD, SOA und IT-Management (MSI)*. GiTO-Verlag, 2007, pp. 35–46. *2 citations on pages 40 and 53*

[90] A. Begel, R. DeLine, and T. Zimmermann, "Social media for software engineering," in *Workshop on Future of Software Engineering (FoSE)*, 2010. *cited on page 307*

[91] L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Systems Journal*, vol. 3, pp. 225–252, 1976. *cited on page 7*

[92] N. Bencomo, G. Blair, R. France, F. Munoz, and C. Jeanneret, "4th int'l workshop on models@run.time," in *MoDELS Workshops*, ser. Lect. Notes in Computer Science, vol. 6002. Springer, 2010, pp. 119–123. *2 citations on pages 246 and 255*

[93] J. Bergey, L. O'Brien, and D. Smith, "Mining existing assets for software product lines," SEI, CMU, Technical Note CMU/SEI-2000-TN-008, May 2000. *2 citations on pages 284 and 286*

[94] ——, "(OAR): A method for mining legacy assets," SEI, CMU, Technical Note CMU/SEI-2001-TN-013, June 2001. *2 citations on pages 285 and 286*

[95] B. Berliner, "CVS II: Parallelizing software development," in *USENIX Winter 1990 Technical Conf.*, vol. 341, 1990, p. 352. *cited on page 143*

[96] M. L. Bernardi, M. Cimitile, and D. Distante, "Web applications design recovery and evolution with RE-UWA," *J. Software: Evolution and Process*, vol. 25, no. 8, pp. 789–814, 2013. *2 citations on pages 201 and 219*

[97] M. L. Bernardi, G. A. Di Lucca, and D. Distante, "A model-driven approach for the fast prototyping of web applications," in *Int'l Symp. Web Systems Evolution*, Sep. 2011, pp. 65–74. *4 citations on pages 201, 219, 220, and 224*

[98] M. L. Bernardi, G. A. Di Lucca, and D. Distante, "The RE-UWA approach to recover user centered conceptual models from web applications," *J. Software Tools for Technology Transfer*, vol. 11, no. 6, pp. 485–501, 2009. *3 citations on pages 201, 219, and 224*

[99] D. M. Berry, "Requirements for maintaining web access for hearing-impaired individuals," in *Int'l Workshop on Web Site Evolution*, Nov. 2001, pp. 33–41. *cited on page 226*

[100] D. M. Berry, B. H. C. Cheng, and J. Zhang, "The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems," in *Int'l Conf. Req. Engineering*, 2005, pp. 113–120. *2 citations on pages 23 and 28*

[101] A. A. Berryman and B. A. Hawkins, "The refuge as an integrating concept in ecology and evolution," *Oikos*, vol. 115, no. 1, pp. 192–196, 2006. *cited on page 311*

[102] N. Bettenburg, B. Adams, A. Hassan, and M. Smidt, "A lightweight approach to uncover technical artifacts in unstructured data," in *Int'l Conf. Program Comprehension*, 2011, pp. 185–188. *2 citations on pages 146 and 155*

[103] N. Bettenburg, E. Shihab, and A. E. Hassan, "An empirical study on the risks of using off-the-shelf techniques for processing mailing list data," in *Int'l Conf. Software Maintenance*, 2009. *2 citations on pages 143 and 146*

[104] N. Bettenburg, S. W. Thomas, and A. E. Hassan, "Using fuzzy code search to link code fragments in discussions to source code," in *European Conf. Software Maintenance and Reengineering*, 2012, pp. 319–328. *cited on page 155*

[105] N. Bettenburg and B. Adams, "Workshop on Mining Unstructured Data (MUD) because "Mining Unstructured Data is like fishing in muddy waters"!" in *Working Conf. Reverse Engineering*, 2010, pp. 277–278. *cited on page 141*

[106] J. Bézivin, "Model driven engineering: An emerging technical space," in *Summer School on Generative and Transformational Techniques in Software Engineering*, ser. Lect. Notes in Computer Science, vol. 4143. Springer, 2006, pp. 36–64. *cited on page 34*

[107] W. Binder and J. Hulaas, "Using bytecode instruction counting as portable cpu consumption metric," *Electronic Notes in Theoretical Computer Science*, vol. 153, no. 2, pp. 57–77, 2006. *cited on page 117*

[108] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Int'l Conf. Mining Software Repositories*. ACM Press, 2006, pp. 137–143. *4 citations on pages 168, 171, 308, and 318*

[109] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu, "Open borders? immigration in open source projects," in *Int'l Conf. Mining Software Repositories*, 2007. *cited on page 323*

[110] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, and P. T. Devanbu, "The promises and perils of mining Git," in *Int'l Conf. Mining Software Repositories*, 2009, pp. 1–10. *cited on page 143*

[111] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python*. O'Reilly Media, 2009. *cited on page 146*

[112] A. Birk, G. Heller, I. John, K. Schmid, T. von der Maßen, and K. Müller, "Product line engineering: The state of the practice," *IEEE Software*, vol. 20, no. 6, pp. 52–60, November/December 2003. *cited on page 284*

[113] C. M. Bishop, "Latent variable models," *Learning in graphical models*, 1998. *cited on page 149*

[114] J. Bitzer, W. Schrettl, and P. J. Schröder, "Intrinsic motivation in open source software development," *Journal of Comparative Economics*, vol. 35, no. 1, pp. 160–169, 2007. *cited on page 323*

[115] G. Blair, N. Bencomo, and R. B. France, “Models@run.time,” *IEEE Computer*, vol. 42, pp. 22–27, 2009. *cited on page 255*

[116] D. M. Blei and J. D. Lafferty, “Topic models,” in *Text Mining: Classification, Clustering, and Applications*. Chapman & Hall, 2009, pp. 71–94. *cited on page 148*

[117] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent Dirichlet allocation,” *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003. *cited on page 148*

[118] R. Blumberg and S. Atre, “The problem with unstructured data,” *DM Review*, vol. 13, pp. 42–49, 2003. *cited on page 140*

[119] J. Boegh, S. Depanfilis, B. Kitchenham, and A. Pasquini, “A Method for Software Quality Planning, Control, and Evaluation,” *IEEE Software*, vol. 16, pp. 69–77, 1999. *cited on page 77*

[120] B. Boehm, P. Bose, E. Horowitz, and M.-J. Lee, “Software requirements as negotiated win conditions,” in *Int’l Conf. Req. Engineering*, 1994, pp. 74–83. *cited on page 291*

[121] B. W. Boehm, “A Spiral Model of Software Development and Enhancement,” *IEEE Computer*, vol. 21, no. 5, pp. 61–72, 1988. *cited on page 8*

[122] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. Macleod, and M. J. Merrit, *Characteristics of Software Quality. Vol. 1*, ser. TRW series of software technology. Elsevier, 1978. *7 citations on pages 66, 68, 71, 337, 338, 339, and 341*

[123] B. W. Bohem, “Perspectives: The Changing Nature of Software Evolution,” *IEEE Software*, vol. 27, no. 4, pp. 26–29, 2010. *2 citations on pages 232 and 233*

[124] S. A. Bohner, “Impact analysis in the software change process: a year 2000 perspective,” in *Int’l Conf. Software Maintenance*, 1996, pp. 42–51. *cited on page 289*

[125] C. Boldyreff, “Keynote: Web accessibility,” in *Int’l Workshop on Web Site Evolution*, Nov. 2001, p. 3. *cited on page 226*

[126] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999. *cited on page 298*

[127] J. C. Bongard, “Evolutionary robotics,” *Comm. ACM*, vol. 56, no. 8, pp. 74–83, Aug. 2013. *cited on page 330*

[128] P. Borba, L. Teixeira, and R. Gheyi, “A theory of software product line refinement,” *Int’l Colloquium on Theoretical Aspects of Computing*, pp. 15–43, 2010. *cited on page 285*

[129] ——, “A theory of software product line refinement,” *Theor. Comput. Sci.*, vol. 455, pp. 2–30, 2012. *cited on page 292*

[130] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000. *4 citations on pages 268, 271, 283, and 286*

[131] J. Bosch and P. Bosch-Sijtsema, “From integration to composition: on the impact of software product lines, global development and ecosystems,” in *Int’l Conf. Software Product Lines*. Springer, 2009. *cited on page 301*

[132] J. C. Bose and U. Suresh, “Root cause analysis using sequence alignment and Latent Semantic Indexing,” in *Australian Conf. Software Engineering*, 2008, pp. 367–376. *cited on page 155*

[133] G. Botterweck and K. Lee, “Feature dependencies have to be managed throughout the whole product life-cycle,” in *Software Engineering (Workshops)*, ser. Lecture Notes in Informatics, vol. 150. Gesellschaft für Informatik, 2009, pp. 101–106. *cited on page 272*

[134] G. Botterweck, A. Pleuss, D. Dhungana, A. Polzer, and S. Kowalewski, “EvoFM: feature-driven planning of product-line evolution,” in *Workshop on Product Line Approaches in Software Engineering*. ACM, 2010, pp. 24–31. *cited on page 278*

[135] G. Botterweck, A. Pleuss, A. Polzer, and S. Kowalewski, “Towards feature-driven planning of product-line evolution,” in *1st Int’l Workshop on Feature-oriented Software Development (FOSD)*, October 2009. *cited on page 278*

[136] G. Bougie, J. Starke, M.-A. Storey, and D. Germán, “Towards understanding Twitter use in software engineering: Preliminary findings, ongoing challenges, and future questions,” in *Int’l Workshop on Web 2.0 for Software Engineering*, 2011. *cited on page 173*

[137] M. Brambilla, S. Comai, P. Fraternali, and M. Matera, “Designing web applications with WebML and Webratio,” in *Web Engineering: Modelling and Implementing Web Applications*, ser. Human-Computer Interaction Series, G. Rossi, O. Pastor, D. Schwabe, and L. Olsina, Eds. Springer, 2008, pp. 221–261. *cited on page 219*

[138] S. Brand, *How Buildings Learn: What Happens After They’re Built*. Viking Press, 1995. *cited on page 9*

[139] J. R. Bray and J. T. Curtis, “An ordination of upland forest communities of southern wisconsin,” *Ecological Monographs*, vol. 27, no. 325-349, 1957. *cited on page 320*

[140] H. P. Breivold, S. Larsson, and R. Land, “Migrating industrial systems towards software product lines: Experiences and observations through case studies,” in *Conf. Software Engineering and Advanced Applications (SEAA)*, 2008, pp. 232–239. *cited on page 284*

[141] P. Brereton, D. Budgen, and G. Hamilton, “Hypertext: The next maintenance mountain,” *IEEE Computer*, vol. 31, no. 12, pp. 49–55, Dec. 1998. *2 citations on pages 207 and 224*

[142] L. Briand, K. E. Emam, and S. Morasca, “Theoretical and Empirical Validation of Software Product Measures,” International Software Engineering Research Network, Tech. Rep. ISERN-95-03, 1995. *cited on page 75*

[143] L. C. Briand, Y. Labiche, and J. Leduc, “Toward the reverse engineering of UML sequence diagrams for distributed Java software,” *IEEE Trans. Soft. Eng.*, vol. 32, pp. 642–663, 2006. *cited on page 331*

[144] F. P. Brooks, *The mythical man-month*, 1st ed. Addison Wesley, 1975. *cited on page 8*

[145] R. A. Brooks, “A Robust Layered Control System for a Mobile Robot,” *IEEE Journal on Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986. *cited on page 247*

[146] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya *et al.*, “Managing technical debt in software-reliant systems,” in *FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 47–52. *2 citations on pages 69 and 331*

[147] Y. Brun, G. D. M. Serugendo, C. Gacek, H. M. Giese, H. Kienle, M. Litoiu, H. A. Müller, M. Pezzè, and M. Shaw, *Engineering Self-Adaptive Systems through Feedback Loops*, ser. Lect. Notes in Computer Science. Springer, 2009, vol. 5525, pp. 48–70. *3 citations on pages 237, 238, and 241*

[148] W. Bruyn, R. Jense, D. Keskar, and P. Ward, “An extended systems modeling language (esml),” *ACM SIGSOFT Software Engineering Notes*, vol. 13, no. 1, pp. 58–67, 1988. *cited on page 52*

[149] J. A. Bubenko, “Information modeling in the context of system development,” in *IFIP Congress*, 1980, pp. 395–411. *cited on page 10*

[150] M. Burnett, C. Cook, and G. Rothermel, “End-user software engineering,” *Comm. ACM*, vol. 47, no. 9, pp. 53–58, Sep. 2004. *cited on page 330*

[151] J. Businge, A. Serebrenik, and M. G. J. van den Brand, “An empirical study of the evolution of Eclipse third-party plug-ins,” in *Joint Int’l Workshop on Principles of software evolution (IWPSE) and ERCIM software evolution workshop*, 2010, pp. 63–72. *cited on page 303*

[152] ——, “Survival of Eclipse third-party plug-ins,” in *Int’l Conf. Software Maintenance*, 2012, pp. 368–377. *No citation*

[153] ——, “Analyzing the Eclipse API usage: Putting the developer in the loop,” in *European Conf. Software Maintenance and Reengineering*. IEEE Computer Society, 2013, pp. 37–46. *cited on page 303*

[154] J. Cabot and C. Gomez, “A catalogue of refactorings for navigation models,” in *Int’l Conf. Web Engineering*, 2008, pp. 75–85. *cited on page 224*

[155] F. Calzolari, P. Tonella, and G. Antoniol, “Maintenance and testing effort modeled by linear and nonlinear dynamic systems,” *Information and Software Technology*, vol. 43, no. 8, pp. 477 – 486, 2001. *3 citations on pages 298, 314, and 325*

[156] G. Candea, J. Cutler, and A. Fox, “Improving Availability with Recursive Microreboots: a Soft-State System Case Study,” *Performance Evaluation*, vol. 56, no. 1-4, pp. 213–248, 2004. *2 citations on pages 250 and 253*

[157] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca, “Decomposing legacy programs: a first step towards migrating to client-server platforms,” *J. Systems and Software*, vol. 54, no. 2, pp. 99–110, Oct. 2000. *cited on page 217*

[158] G. Canfora and M. Di Penta, “New frontiers of reverse engineering,” in *Workshop on the Future of Software Engineering Research*, 2007, pp. 326–341. *cited on page 137*

[159] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, “A framework for QoS-aware binding and re-binding of composite web services,” *J. Systems and Software*, vol. 81, no. 10, pp. 1754–1769, 2008. *cited on page 137*

[160] G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, “Who is going to mentor newcomers in open source projects?” in *Int’l Symp. Foundations of Software Engineering*. ACM, 2012, pp. 44:1–44:11. *cited on page 323*

[161] A. Capiluppi and K. Beecher, “Structural complexity and decay in FLOSS systems: An inter-repository study,” in *European Conf. Software Maintenance and Reengineering*, March, pp. 169–178. *cited on page 304*

[162] A. Capiluppi, P. Lago, and M. Morisio, “Characteristics of open source projects,” in *European Conf. Software Maintenance and Reengineering*. IEEE Computer Society, 2003, pp. 317–327. *cited on page 307*

[163] A. Capiluppi, A. Serebrenik, and L. Singer, “Assessing technical candidates on the social web,” *IEEE Software*, vol. 30, no. 1, pp. 45–51, 2013. *cited on page 332*

[164] A. Capiluppi, A. Serebrenik, and A. Youssef, “Developing an h-index for OSS developers,” in *Int’l Conf. Mining Software Repositories*, 2012, pp. 251–254. *cited on page 331*

[165] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, and S. Panichella, “Traceability recovery using numerical analysis,” in *Working Conf. Reverse Engineering*, 2009, pp. 195–204. *cited on page 162*

[166] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. Mirandola, “QoS-driven runtime adaptation of service oriented architectures,” in *Int’l Symp. Foundations of Software Engineering*. ACM, 2009, pp. 131–140. *2 citations on pages 250 and 251*

[167] J. Carter and P. Dewan, “Are you having difficulty?” in *Int’l Conf. Computer Supported Cooperative Work*, 2010, pp. 211–214. *cited on page 195*

[168] C. Cesarano, A. R. Fasolino, and P. Tramontana, “Improving usability of web pages for blinds,” in *Int’l Symp. Web Systems Evolution*, Sep. 2007, pp. 97–104. *cited on page 225*

[169] N. Chapin, J. E. Hale, J. C. Fernandez-Ramil, and W.-G. Tan, “Types of software evolution and software maintenance,” *J. Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001. *cited on page 9*

[170] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan, “Types of Software Evolution and Software Maintenance,” *J. Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, Jan 2001. *4 citations on pages 230, 243, 251, and 338*

[171] E. B. Charrada, A. Koziolek, and M. Glinz, “Identifying outdated requirements based on source code changes,” in *Int’l Conf. Req. Engineering*, Jun. 2012. *2 citations on pages 20 and 22*

[172] M. A. Chauhan and M. A. Babar, “Migrating service-oriented system to cloud computing: An experience report,” in *IEEE CLOUD*, 2011, pp. 404–411. *cited on page 330*

[173] C.-Y. Chen and P.-C. Chen, “A holistic approach to managing software change impact,” *J. Syst. Softw.*, vol. 82, no. 12, pp. 2051–2067, Dec. 2009. *cited on page 289*

[174] P. Chen, M. Critchlow, A. Garg, C. Van der Westhuizen, and A. van der Hoek, “Differencing and merging within an evolving product line architecture,” in *Intl. Workshop on Product Family Engineering*, 2003. *cited on page 286*

[175] T. Chen, S. W. Thomas, M. Nagappan, and A. E. Hassan, “Explaining software defects using topic models,” in *Int’l Conf. Mining Software Repositories*, 2012, pp. 189–198. *cited on page 162*

[176] Y. Chen, G. C. Gannod, J. S. Collofello, and H. S. Sarjoughian, “Using simulation to facilitate the study of software product line evolution,” in *Int’l Workshop on Principles of Software Evolution*, 2004, pp. 103–112. *cited on page 278*

[177] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "Software Engineering for Self-Adaptive Systems: A Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. Lect. Notes in Computer Science. Springer, 2009, pp. 1–26. *6 citations on pages 230, 233, 243, 244, 255, and 262*

[178] B. H. C. Cheng and J. M. Atlee, "Research Directions in Requirements Engineering," *Workshop on Future of Software Engineering*, pp. 285–303, Feb. 2007. *cited on page 23*

[179] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying bug signatures using discriminative graph mining." in *Int'l Symp. Software Testing and Analysis*, 2009. *2 citations on pages 194 and 195*

[180] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Soft. Eng.*, vol. 20, no. 6, pp. 476–493, June 1994. *cited on page 69*

[181] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990. *2 citations on pages 339 and 340*

[182] S. R. Choudhary, M. Prasad, and A. Orso, "Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications," in *Int'l Conf. Software Testing, Verification and Validation*, Apr. 2012, pp. 171–180. *cited on page 226*

[183] S. R. Choudhary, H. Versee, and A. Orso, "WebDiff: Automated identification of cross-browser issues in web applications," in *Int'l Conf. Software Maintenance*, Oct. 2010. *cited on page 226*

[184] M. B. Chrissis, M. Konrad, and S. Shrum, *CMMI: Guidelines for Process Integration and Product Improvement (Sei Series in Software Engineering)*, 2nd ed. Addison-Wesley Longman, Nov. 2006. *cited on page 66*

[185] J. Chu and T. R. Dean, "Automated migration of list based JSP Web pages to Ajax," in *Working Conf. Source Code Analysis and Manipulation*, Oct. 2008, pp. 217–226. *2 citations on pages 223 and 224*

[186] L. Chung, J. Mylopoulos, and B. A. Nixon, "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach," *IEEE Trans. Soft. Eng.*, vol. 18, pp. 483–497, 1992. *cited on page 16*

[187] L. Chung, B. A. Nixon, and E. S. Yu, "Dealing with change: An approach using nonfunctional requirements," *Int'l Conf. Req. Engineering*, vol. 1, no. 4, pp. 238–260, 1996. *cited on page 11*

[188] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *Enterprise Distributed Object Computing Conf. (EDOC)*. IEEE Computer Society, 2008, pp. 222–231. *2 citations on pages 53 and 315*

[189] A. Cicchetti, D. Di Ruscio, and A. Pierantonio, "Managing dependent changes in coupled evolution," in *Int'l Conf. Model Transformation (ICMT)*, ser. Lect. Notes in Computer Science. Springer, 2009, pp. 35–51. *cited on page 53*

[190] J. A. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. F. Jones, M. Lumkin, B. S. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. J. Shepperd, "Formulating software engineering as a search problem," *IEE Proceedings - Software*, vol. 150, no. 3, pp. 161–175, 2003. *cited on page 104*

[191] E. Clayberg and D. Rubel, *Eclipse: Building Commercial-Quality Plug-ins*, 2nd ed. Addison-Wesley Professional, April 2006. *cited on page 303*

[192] B. Cleary, C. Exton, J. Buckley, and M. English, "An empirical analysis of information retrieval based concept location techniques in software comprehension," *J. Empirical Software Engineering*, vol. 14, no. 1, pp. 93–130, 2008. *cited on page 161*

[193] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. *2 citations on pages 266 and 277*

[194] A. Cleve, T. Mens, and J.-L. Hainaut, "Data-intensive system evolution," *Computer*, vol. 43, no. 8, pp. 110–112, 2010. *cited on page 329*

[195] C. A. Coello Coello, “Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: A survey of the state of the art,” *Computer Methods in Applied Mechanics and Engineering*, vol. 191, no. 11-12, January 2002. *cited on page 119*

[196] D. Coleman, D. Ash, B. Lowther, and P. Oman, “Using metrics to evaluate software system maintainability,” *IEEE Computer*, vol. 27, no. 8, pp. 44–49, Aug. 1994. *cited on page 76*

[197] D. Coleman, B. Lowther, and P. Oman, “The Application of Software Maintainability Models in Industrial Software Systems,” *J. Systems and Software*, vol. 29, no. 1, pp. 3–16, Apr. 1995. *cited on page 76*

[198] J. R. Cordy, “The TXL source transformation language,” *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006. *2 citations on pages 146 and 157*

[199] M. Cordy, A. Classen, P. Schobbens, P. Heymans, and A. Legay, “Managing evolution in software product lines: A model-checking perspective,” in *Int’l Workshop on Variability Modeling of Software-Intensive Systems*, 2012, pp. 183–191. *cited on page 293*

[200] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay, “Towards an incremental automata-based approach for software product-line model checking,” in *Int. Software Product Line Conf.* ACM, 2012, pp. 74–81. *cited on page 330*

[201] J. P. Correia, Y. Kanellopoulos, and J. Visser, “A survey-based study of the mapping of system properties to ISO/IEC 9126 maintainability characteristics,” in *Int’l Conf. Software Maintenance*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 61–70. *cited on page 80*

[202] J. P. Correia and J. Visser, “Benchmarking Technical Quality of Software Products,” in *Working Conf. Reverse Engineering*. IEEE Computer Society, 2008, pp. 297–300. *cited on page 80*

[203] ——, “Certification of Technical Quality of Software Products,” in *Int’l Workshop on Foundations and Techniques for Open Source Software Certification*, 2008, pp. 35–51. *cited on page 80*

[204] M. Critchlow, K. Dodd, J. Chou, and A. Van Der Hoek, “Refactoring product line architectures,” *IWR: Achievements, Challenges, and Effects*, pp. 23–26, 2003. *cited on page 285*

[205] J. S. Cuadrado and J. G. Molina, “Building domain-specific languages for model-driven development,” *IEEE Software*, vol. 24, no. 5, pp. 48–55, 2007. *cited on page 34*

[206] P. Cury, L. Shannon, and Y.-J. Shin, “The functioning of marine ecosystems,” in *Reykjavik Conf. Responsible Fisheries in the Marine Ecosystem*, October 2001. *cited on page 300*

[207] K. Czarnecki and U. W. Eisenecker, *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., 2000. *2 citations on pages 267 and 338*

[208] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski, “Cool features and tough decisions: a comparison of variability modeling approaches,” in *Int’l Workshop on Variability Modeling of Software-Intensive Systems*. ACM , 2012, pp. 173–182. *cited on page 267*

[209] K. Czarnecki, S. Helsen, and U. W. Eisenecker, “Staged configuration using feature models,” in *Int. Software Product Line Conf.*, 2004, pp. 266–283. *cited on page 292*

[210] L. A. Dabbish, H. C. Stuart, J. Tsay, and J. D. Herbsleb, “Social coding in GitHub: transparency and collaboration in an open software repository,” in *Int’l Conf. Computer Supported Cooperative Work*, 2012, pp. 1277–1286. *cited on page 174*

[211] J. B. Dabney and T. L. Harman, *Mastering Simulink 4*, 1st ed. Prentice Hall, 2001. *cited on page 329*

[212] W. J. A. Dahm, “Technology Horizons a Vision for Air Force Science & Technology During 2010-2030,” U.S. Air Force, Tech. Rep., 2010. *2 citations on pages 237 and 263*

[213] Y. Dajsuren, M. G. J. van den Brand, and A. Serebrenik, “Modularity analysis of automotive control software,” *ERCIM News*, vol. 2013, no. 94, pp. 20–21, 2013. *2 citations on pages 329 and 330*

[214] Y. Dajsuren, M. G. J. van den Brand, A. Serebrenik, and S. Roubtsov, “Simulink models are also software: modularity assessment,” in *ACM Sigsoft conference on Quality of Software Architectures*. ACM, 2013, pp. 99–106. *cited on page 329*

[215] F. Dalpiaz, P. Giorgini, and J. Mylopoulos, “An Architecture for Requirements-Driven Self-reconfiguration,” in *Int’l Conf. Advanced Informations Systems Engineering*, 2009, pp. 246–260. *3 citations on pages 17, 22, and 23*

[216] M. D’Ambros, H. Gall, M. Lanza, and M. Pinzger, *Software Evolution*. Springer, 2008, ch. Analysing Software Repositories to Understand Software Evolution, pp. 37–67. *2 citations on pages x and 287*

[217] A. Dardenne, A. van Lamsweerde, and S. Fickas, “Goal-directed requirements acquisition,” *Science of Computer Programming*, vol. 20, no. 1-2, pp. 3–50, 1993. *cited on page 17*

[218] C. Darwin, *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. John Murray, Nov. 1859. *cited on page 311*

[219] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans, “Feature model extraction from large collections of informal product descriptions,” in *Joint European Software Engineering Conf. and ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM , 2013, pp. 290–300. *cited on page 287*

[220] R. C. de Boer and H. van Vliet, “Architectural knowledge discovery with Latent Semantic Analysis: constructing a reading guide for software product audits,” *J. Systems and Software*, vol. 81, no. 9, pp. 1456–1469, 2008. *cited on page 162*

[221] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cikic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, D. B. Smith, J. P. Sousa, L. Tahvildari, K. Wong, and J. Wuttke, “Software Engineering for Self-Adaptive Systems: A second Research Roadmap,” in *Software Engineering for Self-Adaptive Systems II*, ser. Lect. Notes in Computer Science, R. Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds. Springer, 2013, vol. 7475, pp. 1–32. *6 citations on pages 230, 243, 244, 255, 257, and 262*

[222] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, “Enhancing an artefact management system with traceability recovery features,” in *Int’l Conf. Software Maintenance*, 2004, pp. 306–315. *cited on page 161*

[223] ——, “Can information retrieval techniques effectively support traceability link recovery?” in *Int’l Conf. Program Comprehension*, 2006, pp. 307–316. *No citation*

[224] ——, “Recovering traceability links in software artifact management systems using information retrieval methods,” *ACM Trans. Software Engineering and Methodology*, vol. 16, no. 4, 2007. *cited on page 161*

[225] W. De Pauw, R. Hoch, and Y. Huang, “Discovering conversations in web services using semantic correlation analysis,” in *Int’l Conf. Web Services (ICWS)*. IEEE, Jul. 2007, pp. 639–646. *2 citations on pages 213 and 224*

[226] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, 2001. *2 citations on pages 116 and 117*

[227] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multi-objective genetic algorithm: NSGA-II,” *Trans. Evolutionary Computation*, vol. 6, no. 2, pp. 182 – 197, 2002. *cited on page 112*

[228] J.-M. DeBaud and J.-F. Girard, “The relation between the product line development entry points and reengineering,” in *ESPRIT ARES Workshop*, 1998, pp. 132–139. *4 citations on pages 283, 284, 285, and 286*

[229] J.-M. DeBaud and K. Schmid, “A systematic approach to derive the scope of software product lines,” in *Int’l Conf. Software Engineering*, 1999, pp. 34–43. *cited on page 290*

[230] S. Deelstra, M. Sinnema, and J. Bosch, “Product derivation in software product families: a case study,” *J. Systems and Software*, vol. 74, no. 2, pp. 173–194, 2005. *2 citations on pages 270 and 276*

[231] D. Deeptimahanti and M. Ali Babar, "An automated tool for generating UML models from natural language requirements," in *Int'l Conf. Automated Software Engineering*, 2009, pp. 680–682. *cited on page 154*

[232] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990. *cited on page 147*

[233] F. Deissenboeck, L. Heinemann, M. Herrmannsdoerfer, K. Lochmann, and S. Wagner, "The Quamoco tool chain for quality modeling and assessment," in *Int'l Conf. Software Engineering*. ACM, 2011, pp. 1007–1009. *cited on page 92*

[234] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. Mas y Parareda, and M. Pizka, "Tool Support for Continuous Quality Control," *IEEE Software*, vol. 25, no. 5, pp. 60–67, Sep. 2008. *2 citations on pages 80 and 93*

[235] J. Delaney, N. Salminen, and E. Lee, "Infographic: The growing impact of social media," http://www.sociallyawareblog.com/2012/11/21/time-americans-spend-per-month-on-social-media-sites/, 2012. *cited on page 164*

[236] T. DeMarco and T. Lister, *Peopleware: Productive Projects and Teams*, 2nd ed. Dorset House, February 1999. *cited on page 307*

[237] S. Demeyer, *Software Evolution*. Springer, 2008, ch. Object-Oriented Reengineering, pp. 105–138. *cited on page x*

[238] G. Deng, G. Lenz, and D. C. Schmidt, "Addressing domain evolution challenges in software product lines," in *Workshop Model-Driven Development for Product-Lines*, 2005. *cited on page 269*

[239] T. D'Hondt, K. De Volder, K. Mens, and R. Wuyts, "Co-evolution of object-oriented design and implementation," in *Int'l Symp. Software Architectures and Component Technology*. Kluwer, January 2000. *cited on page 315*

[240] D. Dhungana, I. Groher, E. Schludermann, and S. Biffl, *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Edward Elgar, 2013, ch. Guiding principles of natural ecosystems and their applicability to software ecosystems. *cited on page 298*

[241] D. Dhungana, T. Neumayer, P. Grünbacher, and R. Rabiser, "Supporting the evolution of product line architectures with variability model fragments," in *Working Conf. Software Architecture*, 2008, pp. 327–330. *cited on page 295*

[242] G. A. Di Lucca, M. Di Penta, G. Antoniol, and G. Casazza, "An approach for reverse engineering of web-based applications," in *Working Conf. Reverse Engineering*, Oct. 2001, pp. 231–240. *cited on page 210*

[243] G. A. Di Lucca, M. Di Penta, and A. R. Fasolino, "An approach to identify duplicated web pages," in *Int'l Computer Software and Applications Conf.*, 2002. *2 citations on pages 202 and 224*

[244] G. A. Di Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini, "Comprehending web applications by a clustering based approach," in *Int'l Workshop Program Comprehension*, Jun. 2002, pp. 261–270. *cited on page 224*

[245] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana, "Web site accessibility: Identifying and fixing accessibility problems in client page code," in *Int'l Symp. Web Systems Evolution*, Sep. 2005, pp. 71–78. *cited on page 225*

[246] M. Di Penta, D. M. Germán, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the evolution of software licensing," in *Int'l Conf. Software Engineering*, 2010, pp. 145–154. *cited on page 331*

[247] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo, "A language-independent software renovation framework," *J. Systems and Software*, vol. 77, no. 3, pp. 225–240, 2005. *3 citations on pages 116, 117, and 136*

[248] D. Di Ruscio, R. Lämmel, and A. Pierantonio, "Automated co-evolution of gmf editor models," in *Software Language Engineering*, ser. Lect. Notes in Computer Science, vol. 6563. Springer, 2011, pp. 143–162. *cited on page 63*

[249] D. Dig and R. E. Johnson, "How do APIs evolve? a story of refactoring," *J. Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006. *cited on page 34*

[250] T. T. Dinh-Trong and J. M. Bieman, "The FreeBSD project: A replication case study of open source development," *IEEE Trans. Soft. Eng.*, vol. 31, no. 6, pp. 481–494, 2005. *cited on page 307*

[251] D. Distante, "Reengineering legacy applications and web transactions: An extended version of the UWA transaction design model," Ph.D. dissertation, University of Salento, Italy, 2004. *cited on page 227*

[252] D. Distante, P. Pedone, G. Rossi, and G. Canfora, "Model-driven development of Web applications with UWA, MVC and JavaServer faces," in *Int'l Conf. Web Engineering*, ser. Lect. Notes in Computer Science, vol. 4607, 2007, pp. 457–472. *cited on page 218*

[253] B. Dit, D. Poshyvanyk, and A. Marcus, "Measuring the semantic similarity of comments in bug reports," in *Int'l Workshop on Semantic Technologies in System Maintenance*, 2008. *cited on page 153*

[254] B. Dit, E. Moritz, M. Linares-Vsquez, and D. Poshyvanyk, "Supporting and accelerating reproducible research in software maintenance using tracelab component," in *Int'l Conf. Software Maintenance*, 2013. *2 citations on pages 332 and 350*

[255] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013. *2 citations on pages 286 and 287*

[256] I. do Carmo Machado, J. D. McGregor, and E. Santana de Almeida, "Strategies for testing products in software product lines," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–8, 2012. *cited on page 294*

[257] L. Dobrica and E. Niemela, "A survey on software architecture analysis methods," *IEEE Trans. Soft. Eng.*, vol. 28, no. 7, pp. 638–653, 2002. *2 citations on pages 286 and 287*

[258] S. Dobson, S. Denazis, A. Fernandez, D. Gaiti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli, "A Survey of Autonomic Communications," *ACM Trans. Autonomous and Adaptive Systems (TAAS)*, vol. 1, no. 2, pp. 223–259, 2006. *cited on page 245*

[259] A. Dorling, "SPICE: Software Process Improvement and Capability Determination," *Software Quality Journal*, vol. 2, no. 4, pp. 209–224, Dec. 1993. *cited on page 66*

[260] D. Doval, S. Mancoridis, and B. S. Mitchell, "Automatic clustering of software systems using a genetic algorithm," in *Proc. Software Technology and Engineering Practice*. IEEE Computer Society, 1999, pp. 73–82. *4 citations on pages 115, 116, 117, and 118*

[261] J. Dowling and V. Cahill, "Self-Managed Decentralised Systems using K-Components and Collaborative Reinforcement Learning," in *SIGSOFT Workshop on Self-Managed Systems*. ACM , 2004, pp. 39–43. *2 citations on pages 250 and 251*

[262] R. G. Dromey, "A Model for Software Product Quality," *IEEE Trans. Soft. Eng.*, vol. 21, no. 2, pp. 146–162, 1995. *cited on page 71*

[263] A. C. Duarte Pimentel, A. Capiluppi, and C. Boldyreff, "Patterns of creation and usage in the evolution of Wikipedia," in *Int'l Symp. Web Systems Evolution*, Sep. 2012. *cited on page 225*

[264] E. Dubois, J. Hagelstein, E. Lahou, F. Ponsaert, and A. Rifaut, "A knowledge representation language for requirements engineering," *Proc. IEEE*, vol. 74, pp. 1431–1444, 1986. *cited on page 10*

[265] N. Ducheneaut, "Socialization in an open source software community: A socio-technical analysis," *Computer Supported Cooperative Work (CSCW)*, vol. 14, no. 4, pp. 323–368, Aug. 2005. *cited on page 307*

[266] R. M. Dudley, *Uniform Central Limit Theorems*. Cambridge University Press, 1999, vol. 23. *cited on page 83*

[267] S. T. Dumais and T. K. Landauer, "A solution to platos problem: The latent semantic analysis theory of acquisition, induction and representation of knowledge," *Psychological review*, vol. 104, pp. 211–240, 1997. *cited on page 331*

[268] G. Dumont and M. Huzmezan, “Concepts, Methods and Techniques in Adaptive Control,” in *IEEE American Control Conf. (ACC)*, vol. 2, 2002, pp. 1137–1150. *cited on page 241*

[269] S. M. Easterbrook and B. A. Nuseibeh, “Managing inconsistencies in an evolving specification,” in *Int’l Conf. Req. Engineering*, 1995, pp. 48–55. *cited on page 10*

[270] H. Ehrig, C. Ermel, O. Runge, A. Bucchiarone, and P. Pelliccione, *Formal Analysis and Verification of Self-Healing Systems*, ser. Lect. Notes in Computer Science. Springer, 2010, vol. 6013, pp. 139–153. *2 citations on pages 250 and 253*

[271] D. Eichmann, “Evolving an engineered web,” in *Int’l Workshop on Web Site Evolution*, Oct. 1999. *cited on page 225*

[272] EJB 3.1 Expert Group, *Interceptors 1.1*, 2009. *cited on page 331*

[273] D. R. Engler, D. Y. Chen, and A. Chou, “Bugs as inconsistent behavior: A general approach to inferring errors in systems code,” in *Symp. Operating Systems Principles*, 2001, pp. 57–72. *cited on page 124*

[274] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, “Mining source code to automatically split identifiers for software analysis,” in *Int’l Conf. Mining Software Repositories*, 2009, pp. 71–80. *cited on page 145*

[275] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, “Model evolution by run-time parameter adaptation,” in *Int’l Conf. Software Engineering*, 2009, pp. 111–121. *3 citations on pages 17, 22, and 23*

[276] N. Ernst, “Software Evolution: A Requirements Engineering Approach,” Ph.D. dissertation, University of Toronto, 2012. *cited on page 3*

[277] N. Ernst, A. Borgida, and I. Jureta, “Finding Incremental Solutions for Evolving Requirements,” in *Int’l Conf. Req. Engineering*, Feb. 2011, pp. 15–24. *6 citations on pages 5, 6, 24, 26, 30, and 31*

[278] N. Ernst, A. Borgida, I. J. Jureta, and J. Mylopoulos, “Agile requirements engineering via paraconsistent reasoning,” *Information Systems*, Jun. 2013. *2 citations on pages 19 and 22*

[279] N. Ernst and G. C. Murphy, “Case Studies in Just-In-Time Requirements Analysis,” in *Empirical Requirements Engineering Workshop at RE*, Sep. 2012, pp. 1–8. *cited on page 12*

[280] N. Esfahani, E. Kouroshfar, and S. Malek, “Taming Uncertainty in Self-Adaptive Software,” in *Int’l Symp. Foundations of Software Engineering*. ACM, 2011, pp. 234–244. *cited on page 253*

[281] F. Estievenart, A. Francois, J. Henrard, and J.-L. Hainaut, “A tool-supported method to extract data and schema from web sites,” in *Int’l Workshop on Web Site Evolution*, Sep. 2003, pp. 3–11. *cited on page 211*

[282] S. Fahmy, N. Haslinda, W. Roslina, and Z. Fariha, “Evaluating the quality of software in e-book using the ISO 9126 model,” *Int’l J. Control and Automation*, vol. 5, no. 2, pp. 115–122, 2012. *2 citations on pages 72 and 74*

[283] D. Fatiregun, M. Harman, and R. M. Hierons, “Evolving transformation sequences using genetic algorithms,” in *Working Conf. Source Code Analysis and Manipulation*. IEEE CS Press, 2004, pp. 66–75. *3 citations on pages 122, 123, and 136*

[284] D. Faust and C. Verhoef, “Software product line migration and deployment,” *J. Software: Practice and Experience*, vol. 33, no. 10, pp. 933–955, 2003. *3 citations on pages 277, 284, and 294*

[285] J.-M. Favre, “Meta-model and model co-evolution within the 3D software space,” in *ELISA workshop Evolution of Large-scale Industrial Software Evolution*, 2003. *3 citations on pages 34, 36, and 315*

[286] ——, “Languages evolve too! changing the software time scale,” in *Int’l Workshop on Principles of Software Evolution*. IEEE Computer Society, 2005, pp. 33–44. *cited on page 34*

[287] M. C. Feathers, *Working Effectively with Legacy Code*. Prentice Hall, 2005. *cited on page 344*

[288] G. Fedyukovich, O. Sery, and N. Sharygina, “eVolCheck: Incremental upgrade checker for C,” in *Int’l Conf. Tools and Algorithms for Construction and Analysis of Systems*, ser. Lect. Notes in Computer Science. Springer, 2013, vol. 7795, pp. 292–307. *cited on page 330*

[289] N. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. International Thomson Computer Press, 1997. *cited on page 75*

[290] N. E. Fenton and M. Neil, "Software Metrics: Roadmap," in *Int'l Conf. Software Engineering*, 2000, pp. 357–370. *cited on page 69*

[291] J. Fernández-Ramil, A. Lozano, M. Wermelinger, and A. Capiluppi, *Software Evolution*. Springer, 2008, ch. Empirical Studies of Open Source Evolution, pp. 263–288. *2 citations on pages x and 316*

[292] C. Fershtman and N. Gandal, "Open source software: Motivation and restrictive licensing," *International Economics and Economic Policy*, vol. 4, no. 2, pp. 209–225, 2007. *cited on page 323*

[293] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *Trans. Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, May 2002. *cited on page 213*

[294] W. Fitch and M. E., "Construction of phylogenetic trees," *Science*, vol. 155, no. 3760, pp. 279–284, January 1967. *cited on page 311*

[295] B. Fitzgerald, "The transformation of open source software," *MIS Quarterly*, vol. 30, no. 3, 2006. *cited on page 307*

[296] G. Flouris, D. Manakanatas, H. Kondylakis, D. Plexousakis, and G. Antoniou, "Ontology change: classification and survey," *Knowledge Eng. Review*, vol. 23, no. 2, pp. 117–152, 2008. *cited on page 34*

[297] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Control*, vol. 17, no. 4, pp. 367–394, Dec. 2009. *2 citations on pages 315 and 329*

[298] M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, and A. Lau, "An empirical study on web services evolution," *Int'l Conf. Web Services (ICWS)*, pp. 49–56, 2011. *2 citations on pages 212 and 224*

[299] J. Fons, V. Pelechano, O. Pastor, P. Valderas, and V. Torres, "Applying the OOWS Model-Driven Approach for Developing Web Applications. The Internet Movie Database Case Study," in *Web Engineering: Modelling and Implementing Web Applications*, ser. Human-Computer Interaction Series, G. Rossi, O. Pastor, D. Schwabe, and L. Olsina, Eds. Springer, 2008, pp. 65–108. *cited on page 218*

[300] C. Forbes, I. Keivanloo, and J. Rilling, "When open source turns cold on innovation - the challenges of navigating licensing complexities in new research domains," in *Int'l Conf. Software Engineering*, 2012, pp. 1447–1448. *cited on page 331*

[301] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. *7 citations on pages 46, 54, 105, 128, 135, 221, and 340*

[302] ——, *Domain-specific languages*. Addison-Wesley, 2010. *cited on page 34*

[303] M. Fowler and K. Scott, *UML distilled: A brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing Co., 2000. *cited on page 143*

[304] P. Fraternali, S. Comai, A. Bozzon, and G. T. Carughi, "Engineering rich internet applications with a model-driven approach," *ACM Transaction on the Web*, vol. 4, no. 2, pp. 7:1–7:47, apr 2010. *2 citations on pages 218 and 219*

[305] R. T. G. Madey, V. Freeh, "The open source software development phenomenon: An analysis based on social network theory," in *Americas Conf. on Information Systems*, 2002, pp. 1806–1813. *cited on page 308*

[306] S. Gala-Pérez, G. Robles, J. M. González-Barahona, and I. Herraiz, "Intensive metrics for the study of the evolution of open source projects: case studies from apache software foundation projects," in *Int'l Conf. Mining Software Repositories*, 2013, pp. 159–168. *cited on page 306*

[307] C. S. Gall, S. Lukins, L. Etzkorn, S. Gholston, P. Farrington, D. Utley, J. Fortune, and S. Virani, "Semantic software metrics computed from natural language design specifications," *IET Software*, vol. 2, no. 1, pp. 17–26, 2008. *cited on page 162*

[308] H. Gall, M. Jazayeri, R. Klosch, and G. Trausmuth, "Software evolution observations based on product release history," in *Int'l Conf. Software Maintenance*, oct 1997, pp. 160–166. *cited on page 295*

[309] K. Gallagher, C. Caner, and J. Deignan, "The law and reverse engineering," in *Working Conf. Reverse Engineering*, 2012, pp. 3–4. *cited on page 331*

[310] W. Gama, M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Normalizing object-oriented class styles in JavaScript," in *Int'l Symp. Web Systems Evolution*, Sep. 2012. *cited on page 211*

[311] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin, "Managing model adaptation by precise detection of metamodel changes," in *Model Driven Architecture - Foundations and Applications*, ser. Lect. Notes in Computer Science, vol. 5562. Springer, 2009, pp. 34–49. *cited on page 53*

[312] M. Garey and D. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979. *3 citations on pages 104, 113, and 135*

[313] A. Garg, M. Critchlow, P. Chen, C. Van der Westhuizen, and A. van der Hoek, "An environment for managing evolving product line architectures," in *Int'l Conf. Software Maintenance*, 2003, pp. 358–. *cited on page 293*

[314] J. J. Garrett, "Ajax: A new approach to Web applications," *Adaptive Path*, Feb. 2005, http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications. *cited on page 214*

[315] A. Garrido, G. Rossi, and D. Distante, "Systematic improvement of web applications design," *Journal of Web Engineering*, vol. 8, no. 4, pp. 371–404, 2009. *4 citations on pages 201, 221, 222, and 224*

[316] ——, "Refactoring for usability in web applications," *IEEE Software*, vol. 28, no. 3, pp. 60–67, 2011. *4 citations on pages 201, 221, 222, and 224*

[317] E. Gat, *On Three-layer Architectures*. MIT/AAAI, 1998, pp. 1–26. *cited on page 247*

[318] G. de Geest, S. D. Vermolen, A. van Deursen, and E. Visser, "Generating version convertors for domain-specific languages," in *Working Conf. Reverse Engineering*. IEEE Computer Society, 2008, pp. 197–201. *cited on page 53*

[319] M. Genero, J. Olivas, M. Piattini, and F. Romero, "Using Metrics to Predict OO Information Systems Maintainability," in *Int'l Conf. Advanced Informations Systems Engineering*. Springer, 2001, pp. 388–401. *cited on page 76*

[320] D. M. Germán, "The GNOME project: a case study of open source, global software development," *Software Process: Improvement and Practice*, vol. 8, no. 4, pp. 201–215, 2003. *2 citations on pages 306 and 316*

[321] D. M. Germán, B. Adams, and A. E. Hassan, "The evolution of the R software ecosystem," in *European Conf. Software Maintenance and Reengineering*, 2013, pp. 243–252. *3 citations on pages 303, 304, and 331*

[322] M. Gethers and D. Poshyvanyk, "Using relational topic models to capture coupling among classes in object-oriented software systems," in *Int'l Conf. Software Maintenance*, 2010, pp. 1–10. *cited on page 162*

[323] S. Ghaith and M. Ó Cinnéide, "Improving software security using search-based refactoring," in *Int'l Symp. Search Based Software Engineering*, vol. 7515. Springer, 28-30 September 2012, pp. 121–135. *5 citations on pages 129, 130, 132, 133, and 136*

[324] G. Ghezzi and H. C. Gall, "Replicating mining studies with sofas," in *Int'l Conf. Mining Software Repositories*, 2013, pp. 363–372. *cited on page 332*

[325] A. K. Ghose, "Formal tools for managing inconsistency and change in RE," in *Int'l Workshop on Software Specification and Design*, 2000, pp. 171–181. *3 citations on pages 18, 21, and 22*

[326] T. Gilb, *Software Metrics*. Chartwell-Bratt, 1976. *cited on page 69*

[327] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani, "Formal Reasoning Techniques for Goal Models," *Journal on Data Semantics*, vol. 2800, pp. 1–20, 2003. *2 citations on pages 16 and 24*

[328] M. Gobert, J. Maes, A. Cleve, and J. Weber, "Understanding schema evolution as a basis for database reengineering," in *Int'l Conf. Software Maintenance*. IEEE Computer Society, 2013. *cited on page 329*

[329] M. W. Godfrey, A. E. Hassan, J. Herbsleb, G. C. Murphy, M. Robillard, P. Devanbu, A. Mockus, D. E. Perry, and D. Notkin, "Future of mining software archives: A roundtable," *IEEE Software*, vol. 26, no. 1, pp. 67–70, 2008. *cited on page 140*

[330] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *Int'l Conf. Software Maintenance*. IEEE Computer Society, 2000, pp. 131–142. *cited on page 316*

[331] M. Goeminne, "Understanding the evolution of social aspects in open source ecosystems: An empirical analysis of Gnome," Ph.D. dissertation, University of Mons, 2013. *cited on page 316*

[332] M. Goeminne, M. Claes, and T. Mens, "A historical dataset for the GNOME ecosystem," in *Int'l Conf. Mining Software Repositories*. IEEE Computer Society, 2013, pp. 167–170, https://bitbucket.org/mgoeminne/sgl-flossmetric-dbmerge. *cited on page 316*

[333] M. Goeminne and T. Mens, "A framework for analysing and visualising open source software ecosystems," in *Int'l Workshop on Principles of Software Evolution*, 2010, pp. 42–47. *cited on page 303*

[334] ——, "A comparison of identity merge algorithms for software repositories," *Science of Computer Programming*, 2011. *cited on page 318*

[335] ——, "Evidence for the Pareto principle in open source software activity," in *Workshop on Software Quality and Maintainability (SQM)*, ser. CEUR Workshop Proceedings, vol. 701. CEUR-WS.org, 2011, pp. 74–82. *cited on page 303*

[336] ——, *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Edward Elgar, 2013, ch. Analyzing ecosystems for open source software developer communities. *2 citations on pages 307 and 316*

[337] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Longman Publishing Co., 1989. *2 citations on pages 105 and 109*

[338] ——, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1989. *cited on page 298*

[339] H. Gomaa and M. Hussein, "Software reconfiguration patterns for dynamic evolution of software architectures," in *Working Conf. Software Architecture*, 2004, pp. 79–88. *2 citations on pages 278 and 293*

[340] J. M. González-Barahona and G. Robles, "On the reproducibility of empirical software engineering studies based on data retrieved from development repositories," *J. Empirical Software Engineering*, vol. 17, no. 1-2, pp. 75–89, 2012. *cited on page 332*

[341] J. M. Gonzalez-Barahona, G. Robles, M. Ortuño-Pérez, L. Rodero-Merino, J. Centeno-González, V. Matellán-Olivera, E. M. Castro, and P. de las Heras Quirós, *Analyzing the anatomy of GNU/Linux distributions: methodology and case studies (Red Hat and Debian)*. Idea Group Publishing, 2005, ch. 2, pp. 27–58. *cited on page 304*

[342] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 7 edition*, Oracle, 2013. *cited on page 34*

[343] S. Gottipati, D. Lo, and J. Jiang, "Finding relevant answers in software forums," in *Int'l Conf. Automated Software Engineering*, 2011, pp. 323–332. *6 citations on pages 154, 163, 165, 168, 174, and 176*

[344] R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, 1992. *cited on page 72*

[345] S. Grant and J. R. Cordy, "Vector space analysis of software clones," in *Int'l Conf. Program Comprehension*, 2009, pp. 233–237. *cited on page 154*

[346] ——, "Estimating the optimal number of latent concepts in source code analysis," in *Working Conf. Source Code Analysis and Manipulation*, 2010, pp. 65–74. *cited on page 154*

[347] S. Grant, J. R. Cordy, and D. Skillicorn, "Automated concept location using independent component analysis," in *Working Conf. Reverse Engineering*, 2008, pp. 138–142. *cited on page 161*

[348] S. Grant, D. Martin, J. R. Cordy, and D. B. Skillicorn, "Contextualized semantic analysis of web services," in *Int'l Symp. Web Systems Evolution*, Sep. 2011, pp. 33–42. *cited on page 212*

[349] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software factories: assembling applications with patterns, models, frameworks and tools*. Wiley, 2004. *cited on page 269*

[350] S. Greenspan, J. Mylopoulos, and A. Borgida, “Capturing more world knowledge in the requirements specification,” in *Int’l Conf. Software Engineering*, 1982, pp. 225–234. *cited on page 10*

[351] S. Grimes, “Unstructured data and the 80 percent rule,” *Clarabridge Bridgepoints*, 2008. *cited on page 140*

[352] B. Gruschko, D. S. Kolovos, and R. F. Paige, “Towards synchronizing models with evolving metamodels,” in *Workshop on Model-Driven Software Evolution*, 2007. *2 citations on pages 40 and 53*

[353] Y.-G. Guéhéneuc and T. Ziadi, “Automated reverse-engineering of UML v2.0 dynamic models,” in *Proc. ECOOP workshop on Object-Oriented Reengineering*. Springer, July 2005. *cited on page 331*

[354] G. Gui, H. M. Kienle, and H. A. Müller, “REGoLive: Web site comprehension with viewpoints,” in *Int’l Workshop Program Comprehension*, May 2005, pp. 161–164. *cited on page 211*

[355] J. Guo, Y. Wang, P. Trinidad, and D. Benavides, “Consistency maintenance for evolving feature models,” *Expert Syst. Appl.*, vol. 39, no. 5, pp. 4987–4998, 2012. *cited on page 293*

[356] A. Guzzi, M. Pinzger, and A. van Deursen, “Combining micro-blogging and IDE interactions to support developers in their quests,” in *Int’l Conf. Software Maintenance*, 2010. *cited on page 195*

[357] T. Gyimóthy, R. Ferenc, and I. Siket, “Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction,” *IEEE Trans. Soft. Eng.*, pp. 897–910, 2005. *cited on page 77*

[358] J.-L. Hainaut, A. Cleve, J. Henrard, and J.-M. Hick, *Software Evolution*. Springer, 2008, ch. Migration of Legacy Information Systems, pp. 105–138. *3 citations on pages x, 329, and 330*

[359] J. N. Hall, “Perl: Internet duct tape,” *IEEE Internet Computing*, vol. 3, no. 4, pp. 95–96, Jul.-Aug. 1999. *cited on page 208*

[360] M. Hall, N. Walkinshaw, and P. McMinn, “Supervised software modularisation,” in *Int’l Conf. Software Maintenance*. IEEE Computer Society, 2012, pp. 472–481. *2 citations on pages 118 and 136*

[361] M. H. Halstead, *Elements of Software Science*. Elsevier Science, 1977. *cited on page 69*

[362] J. Han and M. Kamber, *Data Mining Concepts and Techniques*, 2nd ed. Morgan Kaufmann, 2006. *cited on page 195*

[363] S. D. P. Harker, K. D. Eason, and J. E. Dobson, “The change and evolution of requirements as a challenge to the practice of software engineering,” in *Int’l Conf. Req. Engineering*, 1993, pp. 266–272. *cited on page 9*

[364] M. Harman, “Search-based software engineering for maintenance and reengineering,” in *European Conf. Software Maintenance and Reengineering*. IEEE Computer Society, 2006, p. 311. *cited on page 104*

[365] ——, “The current state and future of search based software engineering,” in *Workshop on the Future of Software Engineering Research*, 2007, pp. 342–357. *cited on page 104*

[366] ——, “Search based software engineering for program comprehension,” in *Int’l Conf. Program Comprehension*. IEEE Computer Society, 2007, pp. 3–13. *cited on page 104*

[367] M. Harman, Y. Jia, and Y. Zhang, “App store mining and analysis: MSR for app stores,” in *Int’l Conf. Mining Software Repositories*, 2012, pp. 108–111. *cited on page 330*

[368] E. R. Harold, *Refactoring HTML: Improving the Design of Existing Web Applications*. Addison-Wesley, 2012. *cited on page 221*

[369] A. Hars and S. Ou, “Working for free? motivations of participating in open source projects,” in *Annual Hawaii Int’l Conf. System Sciences*. IEEE, 2001, pp. 9–pp. *cited on page 323*

[370] S. Hassaine, F. Boughanmi, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, “A seismology-inspired approach to study change propagation,” in *Int’l Conf. Software Maintenance*, 2011, pp. 53–62. *cited on page 331*

[371] A. E. Hassan, “The road ahead for mining software repositories,” in *Frontiers of Software Maintenance*, 2008, pp. 48–57. *cited on page 140*

[372] ——, “Architecture recovery of web applications,” Master’s thesis, University of Waterloo, Ontario, Canada, 2001. *cited on page 210*

[373] A. E. Hassan and R. C. Holt, “Towards a better understanding of Web applications,” in *Int’l Symp. Web Systems Evolution*, Nov. 2001, pp. 112–116. *cited on page 224*

[374] ——, “Architecture recovery of Web applications,” in *Int’l Conf. Software Engineering*, May 2002, pp. 349–359. *2 citations on pages 210 and 224*

[375] ——, “Migrating web frameworks using water transformations,” in *Int’l Computer Software and Applications Conf.*, Nov. 2003, pp. 296–303. *cited on page 224*

[376] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, “Advancing candidate link generation for requirements tracing: The study of methods,” *IEEE Trans. Soft. Eng.*, pp. 4–19, 2006. *cited on page 161*

[377] S. O. Haykin, *Neural Networks and Learning Machines*. Prentice Hall, 2008. *cited on page 298*

[378] R. Heckel, R. Correia, C. M. P. Matos, M. El-Ramly, G. Koutsoukos, and L. F. Andrade, *Software Evolution*. Springer, 2008, ch. Architectural Transformations: From Legacy to Three-Tier and Services, pp. 139–170. *3 citations on pages x, 330, and 331*

[379] P. Hegedűs, T. Bakota, G. Ladányi, C. Faragó, and R. Ferenc, “A Drill-Down Approach for Measuring Maintainability at Source Code Element Level,” *Electronic Communications of the EASST*, vol. 60, 2013. [Online]. Available: http://journal.ub.tu-berlin.de/eceasst/article/download/852/846 *cited on page 85*

[380] W. Heider, P. Grünbacher, and R. Rabiser, “Negotiation constellations in reactive product line evolution,” in *Int’l Workshop Software Project Mamangement*, 2010, pp. 63–66. *cited on page 291*

[381] W. Heider, R. Froschauer, P. Grünbacher, R. Rabiser, and D. Dhungana, “Simulating evolution in model-based product line engineering,” *Information and Software Technology*, vol. 52, no. 7, pp. 758–769, July 2010. *cited on page 288*

[382] W. Heider and R. Rabiser, “Tool support for evolution of product lines through rapid feedback from application engineering,” in *Int’l Workshop on Variability Modeling of Software-Intensive Systems*, 2010, pp. 167–170. *cited on page 290*

[383] W. Heider, R. Rabiser, D. Dhungana, and P. Grünbacher, “Tracking evolution in model-based product lines,” in *Model-based Product Line Engineering (MAPLE)*, 2009, pp. 59–63. *cited on page 289*

[384] W. Heider, R. Rabiser, and P. Grünbacher, “Facilitating the evolution of products in product line engineering by capturing and replaying configuration decisions,” *J. Software Tools for Technology Transfer*, vol. 14, pp. 613–630, 2012. *cited on page 294*

[385] W. Heider, R. Rabiser, P. Grünbacher, and D. Lettner, “Using regression testing to analyze the impact of changes to variability models on products,” in *Int. Software Product Line Conf.*, 2012, pp. 196–205. *2 citations on pages 289 and 293*

[386] W. Heider, M. Vierhauser, D. Lettner, and P. Grunbacher, “A case study on the evolution of a component-based product line,” in *Joint Working IEEE/IFIP Conf. Software Architecture and European Conference on Software Architecture (WICSA-ECSA)*. IEEE Computer Society, 2012, pp. 1–10. *cited on page 289*

[387] I. Heitlager, T. Kuipers, and J. Visser, “A Practical Model for Measuring Maintainability,” *Int’l Conf. Quality of Information and Communications Technology*, pp. 30–39, 2007. *5 citations on pages 66, 67, 69, 75, and 80*

[388] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. John Wiley & Sons, 2004. *4 citations on pages 238, 239, 240, and 241*

[389] S. Hen*β*, M. Monperrus, and M. Mezini, “Semi-automatically extracting FAQs to improve accessibility of software development knowledge,” in *Int’l Conf. Software Engineering*, 2012, pp. 793–803. *2 citations on pages 154 and 189*

[390] S. A. Hendrickson and A. van der Hoek, “Modeling product line architectures through change sets and relationships,” in *Int’l Conf. Software Engineering*, 2007, pp. 189–198. *cited on page 280*

[391] F. Hermans, B. Sedee, M. Pinzger, and A. van Deursen, "Data clone detection and visualization in spreadsheets," in *Int'l Conf. Software Engineering*, 2013, pp. 292–301. *cited on page 330*

[392] A. Herrmann, A. Wallnöfer, and B. Paech, "Specifying Changes Only – A Case Study on Delta Requirements ," in *Int'l Conf. Req. Engineering*, Apr. 2009, pp. 45–58. *3 citations on pages 12, 20, and 22*

[393] M. Herrmannsdoerfer, "COPE - a workbench for the coupled evolution of metamodels and models," in *Software Language Engineering*, 2010. *cited on page 56*

[394] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "Automatability of coupled evolution of metamodels and models in practice," in *Int'l Conf. Model Driven Engineering Languages and Systems*, ser. Lect. Notes in Computer Science, vol. 5301. Springer, 2008, pp. 645–659. *2 citations on pages 38 and 44*

[395] ——, "COPE - automating coupled evolution of metamodels and models," in *ECOOP 2009 - Object-Oriented Programming*. Springer, 2009. *cited on page 54*

[396] M. Herrmannsdoerfer and M. Koegel, "Towards semantics-preserving model migration," in *Int'l Workshop on Models and Evolution*, 2010. *cited on page 62*

[397] M. Herrmannsdoerfer and D. Ratiu, "Limitations of automating model migration in response to metamodel adaptation," in *Models in Software Engineering*, ser. Lect. Notes in Computer Science, vol. 6002. Springer, 2010, pp. 205–219. *cited on page 62*

[398] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth, "Language evolution in practice: The history of GMF," in *Int'l Conf. Software Language Engineering*, ser. Lect. Notes in Computer Science, vol. 5969. Springer, 2010, pp. 3–22. *cited on page 45*

[399] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth, "An extensive catalog of operators for the coupled evolution of metamodels and models," in *Int'l Conf. Software Language Engineering*, ser. Lect. Notes in Computer Science. Springer, 2010. *4 citations on pages 38, 54, 55, and 56*

[400] G. Hertel, S. Niedner, and S. Herrmann, "Motivation of software developers in open source projects: an internet-based survey of contributors to the Linux kernel," *Research policy*, vol. 32, no. 7, pp. 1159–1177, 2003. *cited on page 323*

[401] A. Hindle, M. W. Godfrey, and R. C. Holt, "What's hot and what's not: Windowed developer topic analysis," in *Int'l Conf. Software Maintenance*, 2009, pp. 339–348. *cited on page 152*

[402] ——, "Software process recovery using recovered unified process views," in *Int'l Conf. Software Maintenance*, 2010, pp. 1–10. *cited on page 152*

[403] A. Hindle, "Green mining: Investigating power consumption across versions," in *Int'l Conf. Software Engineering*, 2012, pp. 1301–1304. *cited on page 332*

[404] A. Hindle, C. Bird, T. Zimmermann, and N. Nagappan, "Relating Requirements to Implementation via Topic Analysis: Do Topics Extracted from Requirements Make Sense to Managers and Developers?" in *Int'l Conf. Software Maintenance*, 2012, pp. 1–12. *3 citations on pages 12, 20, and 22*

[405] C. S. Holling, "Resilience and stability of ecological systems," *Annual Review of Ecology and Systematics*, vol. 4, pp. 1–23, 1973. *cited on page 300*

[406] J. Hößler, M. Soden, and H. Eichler, *Models and Human Reasoning*. Wissenschaft und Technik Verlag, 2005, ch. Coevolution of Models, Metamodels and Transformations, pp. 129–154. *cited on page 54*

[407] L. Hotz, K. Wolter, T. Krebs, J. Nijhuis, S. Deelstra, M. Sinnema, and J. MacGregor, *Configuration in Industrial Product Families - The ConIPF Methodology*. IOS Press, 2006. *2 citations on pages 272 and 278*

[408] J. Howison, M. Conklin, and K. Crowston, "Flossmole: A collaborative repository for floss research data and analyses," *Int'l J. Information Technology and Web Engineering*, vol. 1, no. 3, pp. 17–26, 2006. *cited on page 350*

[409] I. Hsi and C. Potts, "Studying the evolution and enhancement of software features," in *Int'l Conf. Software Maintenance*, 2000, pp. 143–151. *cited on page 287*

[410] S. Huang, "Frontiers of website evolution," in *Int'l Conf. Software Maintenance*, Sep. 2008, pp. 78–86. *2 citations on pages 216 and 225*

[411] B. A. Huberman and T. Hogg, "The emergence of computational ecologies," in *Lectures in Complex Systems*. Addison-Wesley, 1993, pp. 185–205. *cited on page 298*

[412] G. Hunt and S. McKinnell, "Interplay between top-down, bottom-up, and wasp-waist control in marine ecosystems," in *Progress In Oceanography*, vol. 68, no. 2-4, 2006, pp. 115–124. *cited on page 300*

[413] A. Hunter and B. A. Nuseibeh, "Managing inconsistent specifications: reasoning, analysis, and action," *ACM Trans. Software Engineering and Methodology*, vol. 7, no. 4, 1998. *2 citations on pages 19 and 22*

[414] D. H. Huson, R. Rupp, and C. Scornavacca, *Phylogenetic Networks: Concepts, Algorithms and Applications*. Cambridge University Press, 2010. *cited on page 311*

[415] K. Hussey and M. Paternostro, "Advanced features of EMF," Tutorial at EclipseCon 2006, California, USA. [Accessed 07 September 2009] Available at: http://www.eclipsecon.org/2006/Sub.do?id=171, 2006. *cited on page 60*

[416] D. Hutchins, "A biologist's view of software evolution," in *Reflection, AOP and Meta-Data for Software Evolution*, 2005, pp. 95–105. *cited on page 313*

[417] IBM Corporation, "An Architectural Blueprint for Autonomic Computing," IBM Corporation, Tech. Rep., 2006. *2 citations on pages 245 and 247*

[418] W. Ibrahim, N. Bettenburg, E. Shihab, B. Adams, and A. Hassan, "Should I contribute to this discussion?" in *Int'l Conf. Mining Software Repositories*, 2010. *cited on page 195*

[419] IEEE, *Standard 610.12-1990: Glossary of Software Engineering Terminology*. IEEE Press, 1999, vol. 1. *cited on page 340*

[420] ——, *Standard IEEE Std 1219-1999 on Software Maintenance*. IEEE Press, 1999, vol. 2. *cited on page 338*

[421] IEEE Software Engineering Standards Committee, "IEEE Recommended Practice for Software Requirements Specifications," IEEE, Tech. Rep., 1998. *2 citations on pages 13 and 336*

[422] International Standards Organization, "Standard 9126 on information technology – software product evaluation – quality characteristics and guidelines for their use," 1991. *6 citations on pages 66, 68, 72, 77, 92, and 346*

[423] ——, "Software life cycle processes," 1995. *2 citations on pages 338 and 346*

[424] ——, "Standard 14764 on software engineering – software life cycle processes – software maintenance," 1999. *5 citations on pages 45, 46, 243, 338, and 346*

[425] ——, *ISO/IEC 25000:2005. Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*. ISO/IEC, 2005. *3 citations on pages 66, 68, and 72*

[426] ——, "Standard 25000 on software engineering – software product quality requirements and evaluation (SQuaRE)," 2005. *cited on page 345*

[427] ——, "Standard 9001 on quality management systems – requirements," 2008. *2 citations on pages 66 and 345*

[428] A. Israeli and D. G. Feitelson, "The Linux kernel as a case study in software evolution," *J. Systems and Software*, vol. 83, no. 3, pp. 485–501, 2010. *cited on page 304*

[429] F. Jaafar, S. Hassaine, Y. Guéhéneuc, S. Hamel, and B. Adams, "On the relationship between program evoluton and fault-proneness: an empirical study," in *European Conf. Software Maintenance and Reengineering*. IEEE Press, 2013, pp. 15–24. *2 citations on pages 313 and 315*

[430] J. Jackson, "Google to use HTML5 in Gmail," *Computerworld*, Jun. 2010. *cited on page 211*

[431] M. Jackson, *Problem Frames: Analysing & Structuring Software Development Problems*. Addison-Wesley, 2000. *cited on page 16*

[432] I. Jacobson, M.Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley, 1992. *cited on page 87*

[433] S. Jansen, A. Finkelstein, and S. Brinkkemper, "A sense of community: A research agenda for software ecosystems," in *Int'l Conf. Software Engineering*, May 2009, pp. 187–190. *2 citations on pages 302 and 340*

[434] S. Jansen, S. Brinkkemper, and A. Finkelstein, "Business Network Management as a Survival Strategy: A Tale of Two Software Ecosystems," in *Int'l Workshop on Software Ecosystems*, ser. CEUR Workshop Proceedings. CEUR-WS.org, 2009, pp. 34–48. *2 citations on pages 302 and 340*

[435] S. Jansen, M. Cusumano, and S. Brinkkemper, Eds., *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Edward Elgar, 2013. *2 citations on pages 302 and 326*

[436] M. Jarke, P. Loucopoulos, K. Lyytinen, J. Mylopoulos, and W. N. Robinson, "The Brave New World of Design Requirements: Four Key Principles," in *Int'l Conf. Advanced Informations Systems Engineering*, 2010, pp. 470–482. *cited on page 4*

[437] S. Jarzabek, *Effective Software Maintenance and Evolution: A Reuse-Based Approach*. Auerbach Publications, 2008. *cited on page 344*

[438] A. C. Jensen and B. H. Cheng, "On the use of genetic programming for automated refactoring and the introduction of design patterns," in *Genetic and Evolutionary Computation Conf.* ACM, 7-11 July 2010, pp. 1341–1348. *3 citations on pages 129, 134, and 136*

[439] C. Jergensen, A. Sarma, and P. Wagstrom, "The onion patch: migration in open source ecosystems," in *Joint European Software Engineering Conf. and ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*. ACM, 2011, pp. 70–80. *2 citations on pages 307 and 323*

[440] H. Jiang, T. N. Nguyen, I. Chen, H. Jaygarl, and C. Chang, "Incremental Latent Semantic Indexing for automatic traceability link evolution management," in *Int'l Conf. Automated Software Engineering*, 2008, pp. 59–68. *cited on page 161*

[441] Y. Jiang and E. Stroulia, "Towards reengineering Web sites to Web-services providers," in *European Conf. Software Maintenance and Reengineering*, Mar. 2004, pp. 296–305. *2 citations on pages 217 and 224*

[442] W. Jirapanthong and A. Zisman, "Xtraque: traceability for product line systems," *Software and Systems Modeling*, vol. 8, pp. 117–144, 2009. *cited on page 289*

[443] T. Joachims, "SVM-HMM: Sequence tagging with SVMs," http://www.cs.cornell.edu/people/tj/svm_light/svm_hmm.html. *cited on page 176*

[444] I. John, J. Knodel, T. Lehner, and D. Muthig, "A practical guide to product line scoping," in *Int. Software Product Line Conf.*, 2006, pp. 3–12. *cited on page 284*

[445] S. Johnsson and J. Bosch, "Quantifying software product line ageing," in *Workshop on Software Product Lines: Economics, Architectures, and Implications*, 2000, pp. 27–32. *cited on page 288*

[446] Joint ACM and IEEE CS Taks Force on Computing Curricula, "Computer science curricula 2013," ACM and IEEE Computer Society, Tech. Rep. Ironman Draft (Version 1.0), February 2013. *cited on page ix*

[447] M. E. Joorabchi and A. Mesbah, "Reverse engineering iOS mobile applications," in *Working Conf. Reverse Engineering*, 2012, pp. 177–186. *cited on page 331*

[448] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 12, pp. 31 – 39, 2008, special Issue on Experimental Software and toolkits (EST). *2 citations on pages 51 and 53*

[449] I. J. Jureta, A. Borgida, N. Ernst, and J. Mylopoulos, "Techne: Towards a New Generation of Requirements Modeling Languages with Goals, Preferences, and Inconsistency Handling," in *Int'l Conf. Req. Engineering*, Sydney, Australia, 2010, pp. 115–124. *cited on page 5*

[450] I. J. Jureta, J. Mylopoulos, and S. Faulkner, "Revisiting the Core Ontology and Problem in Requirements Engineering," in *Int'l Conf. Req. Engineering*, 2008, pp. 71–80. *cited on page 5*

[451] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *J. Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 2, pp. 77–131, 2007. *cited on page 287*

[452] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep., November 1990. *cited on page 49*

[453] H. Katsuno and A. O. Mendelzon, “On the difference between updating a knowledge base and revising it,” in *Int’l Conf. Knowledge Representation and Reasoning*. Cambridge University Press, 1991, pp. 1–21. *cited on page 29*

[454] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, “Mudablue: An automatic categorization system for open source repositories,” *J. Systems and Software*, vol. 79, no. 7, pp. 939–953, 2006. *cited on page 153*

[455] ——, “Mudablue: An automatic categorization system for open source repositories,” *Journal of Systems and Software*, vol. 79, no. 7, pp. 939–953, 2006. *cited on page 177*

[456] J. Kennedy and R. C. Eberhart, “Particle swarm optimization,” in *Int’l Conf. Neural Networks*, vol. 4, 1995, pp. 1942 – 1948. *cited on page 108*

[457] J. O. Kephart and D. M. Chess, “The Vision of Autonomic Computing,” *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003. *5 citations on pages 244, 245, 247, 249, and 250*

[458] J. Kerievsky, *Refactoring to patterns*. Addison-Wesley, 2004. *2 citations on pages 128 and 344*

[459] M. Kessentini, H. A. Sahraoui, M. Boukadoum, and O. B. Omar, “Search-based model transformation by example,” *Software and Systems Modeling*, vol. 11, no. 2, pp. 209–226, 2012. *3 citations on pages 126, 127, and 136*

[460] H. M. Kienle, “Building reverse engineering tools with software components,” Ph.D. dissertation, Department of Computer Science, University of Victoria, Nov. 2006. *cited on page 201*

[461] H. M. Kienle, G. A. Di Lucca, and K. Kontogiannis, “Special issue: Selected papers from the 12th international symposium on web systems evolution (wse 2010),” *J. Software: Evolution and Process*, vol. 25, no. 8, 2013. *cited on page 227*

[462] H. M. Kienle, G. A. Di Lucca, and S. R. Tilley, “Research directions in web systems evolution IV: Migrating to the cloud,” in *Int’l Symp. Web Systems Evolution*, Sep. 2010, pp. 121–122. *2 citations on pages 215 and 330*

[463] H. M. Kienle, D. German, S. Tilley, and H. A. Müller, “Managing legal risks associated with web content,” *Int’l J. Business Information Systems (IJBIS)*, vol. 3, no. 1, pp. 86–106, Dec. 2008. *cited on page 205*

[464] H. M. Kienle and H. A. Müller, “A WSAD-based fact extractor for J2EE web projects,” in *Int’l Symp. Web Systems Evolution*, Oct. 2007, pp. 57–64. *cited on page 224*

[465] ——, “Rigi–an environment for software reverse engineering, exploration, visualization, and redocumentation,” *Science of Computer Programming*, vol. 75, no. 4, pp. 247–263, Apr. 2010. *cited on page 207*

[466] ——, “Legal aspects of web systems,” in *Int’l Symp. Web Systems Evolution*, Sep. 2013. *cited on page 205*

[467] H. M. Kienle, P. Tramontana, S. R. Tilley, and D. Bolchini, “Ten years of access for all from wse 2001 to wse 2011,” in *Int’l Symp. Web Systems Evolution*, Sep. 2011, pp. 99–104. *2 citations on pages 201 and 225*

[468] H. M. Kienle and C. A. Vasiliu, “Evolution of legal statements on the web,” in *Int’l Symp. Web Systems Evolution*, Oct. 2008, pp. 73–82. *cited on page 225*

[469] H. M. Kienle, A. Weber, J. Martin, and H. A. Müller, “Development and maintenance of a web site for a bachelor program,” in *Int’l Symp. Web Systems Evolution*, Sep. 2003, pp. 20–29. *2 citations on pages 201 and 206*

[470] M. Kim and D. Notkin, “Discovering and representing systematic code changes,” in *Int’l Conf. Software Engineering*, 2009, pp. 309–319. *cited on page 279*

[471] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008. *cited on page 34*

[472] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, “Refactoring a legacy component for reuse in a software product line: a case study,” *J. Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 109–132, 2006. *2 citations on pages 284 and 285*

[473] D. Kolovos, “An extensible platform for specification of integrated languages for model management,” Ph.D. dissertation, University of York, United Kingdom, 2009. *cited on page 57*

[474] E. Korshunova, M. Petković, M. G. J. van den Brand, and M. R. Mousavi, "CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code," in *Working Conf. Reverse Engineering*, 2006, pp. 297–298. *cited on page 331*

[475] R. Koschke, *Software Evolution*. Springer, 2008, ch. Identifying and Removing Software Clones, pp. 15–36. *2 citations on pages x and 286*

[476] E. Kouters, B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand, "Who's who in Gnome: using LSA to merge software repository identities," in *Int'l Conf. Software Maintenance*. IEEE, 2012, pp. 592–595. *cited on page 318*

[477] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992. *cited on page 110*

[478] J. Kramer and J. Magee, "Dynamic Structure in Software Architectures," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 6, pp. 3–14, 1996. *cited on page 248*

[479] ——, "Self-Managed Systems: an Architectural Challenge," in *Workshop on the Future of Software Engineering (FoSE)*. IEEE, 2007, pp. 259–268. *cited on page 248*

[480] A. Kraus, A. Knapp, and N. Koch, "Model-driven generation of web applications in uwe," in *Int'l Workshop on Model-Driven Web Engineering (MDWE)*, ser. CEUR Workshop Proceedings, vol. 261. CEUR-WS.org, 2007. *cited on page 218*

[481] C. Krebs, *Ecology: The experimental analysis of distribution and abundance*. Harper and Row, 1972. *cited on page 299*

[482] A. Kuhn, S. Ducasse, and T. Girba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007. *cited on page 142*

[483] T. Kuipers and J. Visser, "Maintainability Index Revisited - position paper," in *Int'l Workshop on Software Quality and Maintainability (SQM)*. IEEE Computer Society Press, 2007. *2 citations on pages 75 and 80*

[484] V. Kumar, B. F. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan, "Middleware for Enterprise Scale Data Stream Management using Utility-driven Self-Adaptive Information Flows," *Cluster Computing*, vol. 10, pp. 443–455, 2007. *2 citations on pages 250 and 252*

[485] I. Kurtev, J. Bézivin, and M. Aksit, "Technological spaces: An initial appraisal," in *Int'l Symp. Distributed Objects and Applications (DOA)*, 2002. *cited on page 34*

[486] P. van de Laar and T. Punter, *Views on Evolvability of Embedded Systems*. Springer, 2010. *cited on page 330*

[487] W. Lam and M. Loomes, "Requirements Evolution in the Midst of Environmental Change: A Managed Approach," in *European Conf. Software Maintenance and Reengineering*, 1998, pp. 121–127. *cited on page 9*

[488] J. Lamarck, *Philosophie zoologique*. Dentu, Paris, 1809. *cited on page 311*

[489] A. Lamkanfi, S. Demeyer, Q. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *European Conf. Software Maintenance and Reengineering*, 2011. *cited on page 186*

[490] R. Lämmel, "Grammar adaptation," in *Int'l Conf. Formal Methods Europe (FME)*, ser. Lect. Notes in Computer Science, vol. 2021. Springer, 2001, pp. 550–570. *cited on page 54*

[491] R. Lämmel and W. Lohmann, "Format evolution," in *Int'l Conf. Reverse Engineering for Information Systems (RETIS)*, vol. 155. OCG, 2001. *cited on page 34*

[492] R. Lämmel and V. Zaytsev, "Recovering grammar relationships for the Java language specification," *Software Quality Journal*, vol. 19, no. 2, pp. 333–378, March 2011. *cited on page 34*

[493] P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdoerfer, M. Seidl, K. Wieland, and G. Kappel, "A posteriori operation detection in evolving software models," *J. Software Maintenance and Evolution: Research and Practice*, vol. 86, no. 2, pp. 551 – 566, 2013. *cited on page 63*

[494] F. Lanubile and T. Mallardo, "Finding function clones in Web applications," in *European Conf. Software Maintenance and Reengineering*, Mar. 2003, pp. 379–386. *cited on page 224*

[495] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Springer, 2005. *2 citations on pages 70 and 86*

[496] P. A. Laplante, Ed., *Encyclopedia of Software Engineering*. Auerbach Publications, 2010. *cited on page 344*

[497] A. Lapouchnian and J. Mylopoulos, "Modeling Domain Variability in Requirements Engineering with Contexts," in *Int'l Conf. Conceptual Modelling*, 2009, pp. 115–130. *2 citations on pages 16 and 22*

[498] C. Larman and V. R. Basili, "Iterative and incremental development: A brief history," *IEEE Computer*, vol. 36, no. 6, pp. 47–56, June 2003. *cited on page 8*

[499] T. C. Lau, J. Lu, E. Hedges, and E. Xing, "Migrating e-commerce database applications to an enterprise Java environment," in *Conf. Center for Advanced Studies on Collaborative Research*, Nov. 2001. *cited on page 221*

[500] J. Lawrance, R. Bellamy, and M. Burnett, "Scents in programs: Does information foraging theory apply to program maintenance?" in *IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC)*, 2007, pp. 15–22. *2 citations on pages 298 and 325*

[501] D. Lawrie and D. Binkley, "Expanding identifiers to normalize source code vocabulary," in *Int'l Conf. Software Maintenance*, 2011, pp. 113–122. *cited on page 145*

[502] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Soft. Eng.*, vol. 38, no. 1, pp. 54–72, 2012. *3 citations on pages 124, 125, and 136*

[503] D. Leffingwell, *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*. Addison-Wesley, 2011. *cited on page 14*

[504] D. Leffingwell and D. Widrig, *Managing Software Requirements: A Use Case Approach*, 2nd ed. Addison-Wesley Professional, 2003. *cited on page 13*

[505] M. M. Lehman, "On understanding laws, evolution and conservation in the large program life cycle," *J. Systems and Software*, vol. 1, no. 3, pp. 213–221, 1980. *cited on page 316*

[506] ——, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980. *cited on page 152*

[507] ——, "Feedback in the Software Evolution Process," *Information and Software Technology*, vol. 38, no. 11, pp. 681–686, 1996. *cited on page 236*

[508] M. M. Lehman and L. A. Belady, *Program Evolution: Processes of Software Change*, ser. Apic Studies In Data Processing. Academic Press, 1985. *cited on page 344*

[509] M. M. Lehman and J. Fernandez Ramil, "Rules and tools for software evolution planning and management," *Annals of Software Engineering*, vol. 11, pp. 16–44, 2001, special Issue on Software Management. *2 citations on pages 233 and 236*

[510] ——, *Software Evolution and Feedback: Theory and Practice*. Wiley, 2006, ch. Software evolution, pp. 7–40. *2 citations on pages 6 and 233*

[511] M. M. Lehman, J. Fernandez Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution – the nineties view," in *Int'l Symp. Software Metrics*. IEEE Computer Society, 1997, pp. 20–32. *2 citations on pages 88 and 316*

[512] A. Lesk, *Introduction to Bioinformatics*. Oxford University Press, 2008. *cited on page 298*

[513] L. Lessig, *Free Culture: How Big Media Uses Technology and the Law to Lock Down Culture and Control Creativity*. The Penguin Press, 2004, http://www.free-culture.cc/freeculture.pdf. *cited on page 205*

[514] T. C. Lethbridge, R. Laganiere, and C. King, *Object-oriented software engineering: practical software development using UML and Java*. McGraw-Hill, 2005. *2 citations on pages 141 and 143*

[515] E. Letier and A. van Lamsweerde, "Reasoning about partial goal satisfaction for requirements and design engineering," in *Int'l Symp. Foundations of Software Engineering*. ACM Press, 2004, pp. 53—62. *2 citations on pages 18 and 22*

[516] J. L. Letouzey and T. Coq, "The SQALE Analysis Model: An Analysis Model Compliant with the Representation Condition for Assessing the Quality of Software Source Code," in

Int'l Conf. Advances in System Testing and Validation Lifecycle (VALID). IEEE, Aug. 2010, pp. 43–48. *4 citations on pages 66, 67, 69, and 78*

[517] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet Physics Doklady*, vol. 10, 1966, pp. 707–710. *cited on page 223*

[518] G. Lewis, E. Morris, and D. Smith, "Analyzing the reuse potential of migrating legacy components to a service-oriented architecture," in *European Conf. Software Maintenance and Reengineering*, Mar. 2006, pp. 15–23. *cited on page 217*

[519] W. Li, C. Zhang, and S. Hu, "G-finder: Routing programming questions closer to the experts," in *ACM Sigplan Notices*, vol. 45, no. 10. ACM, 2010, pp. 62–73. *cited on page 154*

[520] S. Liaskos, A. Lapouchnian, Y. Yu, E. S. Yu, and J. Mylopoulos, "On Goal-based Variability Acquisition and Analysis," in *Int'l Conf. Req. Engineering*, 2006. *cited on page 16*

[521] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," *Programming language design and implementation*, pp. 141–154, 2003. *cited on page 124*

[522] E. Lim, N. Taksande, and C. Seaman, "A balancing act: What software practitioners have to say about technical debt," *IEEE Software*, pp. 22–27, November/December 2012. *cited on page 331*

[523] C. R. Linder, B. M. E. Moret, L. Nakhleh, and T. Warnow, "Network (reticulate) evolution: biology, models, and algorithms," in *Pacific Symp. Biocomputing*, 2004. *cited on page 311*

[524] E. Linstead and P. Baldi, "Mining the coherence of GNOME bug reports with statistical topic models," in *Int'l Conf. Mining Software Repositories*, 2009, pp. 99–102. *2 citations on pages 153 and 306*

[525] E. Linstead, C. Lopes, and P. Baldi, "An application of latent Dirichlet allocation to analyzing software evolution," in *Int'l Conf. Machine Learning and Applications*, 2008, pp. 813–818. *cited on page 152*

[526] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining concepts from code with probabilistic topic models," in *Int'l Conf. Automated Software Engineering*, 2007, pp. 461–464. *cited on page 161*

[527] ——, "Mining Eclipse developer contributions via author-topic models," in *Int'l Conf. Mining Software Repositories*, 2007, pp. 30–33. *cited on page 161*

[528] ——, "Mining internet-scale software repositories," in *Advances in Neural Information Processing Systems*, vol. 2007, 2008, pp. 929–936. *cited on page 154*

[529] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Mining and Knowledge Discovery*, vol. 18, no. 2, pp. 300–336, 2008. *cited on page 154*

[530] A. Liso, "Software Maintainability Metrics Model: An Improvement in the Coleman-Oman Model," *Crosstalk*, p. 1517, 2001. *cited on page 75*

[531] M. Litoiu, "Migrating to web services - latency and scalability," in *Int'l Symp. Web Systems Evolution*, Oct. 2002, pp. 13–20. *2 citations on pages 218 and 224*

[532] J. Liu, D. S. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *Int'l Conf. Software Engineering*, 2006, pp. 112–121. *cited on page 285*

[533] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimothy, and N. Chrisochoides, "Modeling class cohesion as mixtures of latent topics," in *Int'l Conf. Software Maintenance*, 2009, pp. 233–242. *cited on page 162*

[534] Y. Liu, Q. Wang, M. Zhuang, and Y. Zhu, "Reengineering legacy systems with RESTful web service," in *Int'l Computer Software and Applications Conf.*, Jul. 2008, pp. 785–790. *cited on page 214*

[535] X. Llorà, K. Sastry, D. E. Goldberg, A. Gupta, and L. Lakshmi, "Combating user fatigue in iGAs: partial ordering, support vector machines, and synthetic fitness," in *Genetic and Evolutionary Computation Conf.* ACM, 2005, pp. 1363–1370. *cited on page 136*

[536] L. Lopez-Fernandez, G. Robles, J. Gonzalez-Barahona, and I. Herraiz, "Applying social network analysis techniques to community-driven libre software projects," *Int'l J. Information Technology and Web Engineering*, vol. 1, no. 3, pp. 27–48, 2006. *cited on page 316*

[537] M. Lormans, "Monitoring requirements evolution using views," in *European Conf. Software Maintenance and Reengineering*, 2007, pp. 349–352. *cited on page 161*

[538] M. Lormans, H. G. Gross, A. van Deursen, and R. van Solingen, "Monitoring requirements coverage using reconstructed views: An industrial case study," in *Working Conf. Reverse Engineering*, 2006, pp. 275–284. *No citation*

[539] M. Lormans and A. van Deursen, "Can LSI help reconstructing requirements traceability in design and test?" in *European Conf. Software Maintenance and Reengineering*, 2006, pp. 47–56. *cited on page 161*

[540] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the 'hurried' bug report reading process to summarize bug reports," in *Int'l Conf. Software Maintenance*, 2012, pp. 1–10. *cited on page 153*

[541] N. Loughran, A. Rashid, W. Zhang, and S. Jarzabek, "Supporting product line evolution with framed aspects," in *Workshop APC4IS*, 2004, pp. 22–26. *cited on page 293*

[542] Z. Lubsen, A. Zaidman, and M. Pinzger, "Using association rules to study the co-evolution of production and test code," in *Int'l Conf. Mining Software Repositories*, 2009, pp. 151–154. *cited on page 330*

[543] B. Luijten and J. Visser, "Faster Defect Resolution with Higher Technical Quality Software," in *Int'l Workshop on Software Quality and Maintainability (SQM)*. IEEE Computer Society Press, 2010, pp. 11–20. *cited on page 81*

[544] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Source code retrieval for bug localization using latent Dirichlet allocation," in *Working Conf. Reverse Engineering*, 2008, pp. 155–164. *cited on page 162*

[545] ——, "Bug localization using latent Dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010. *cited on page 162*

[546] M. Lungu, "Towards reverse engineering software ecosystems," in *Int'l Conf. Software Maintenance*, 2008, pp. 428–431. *cited on page 302*

[547] ——, "Reverse engineering software ecosystems," Ph.D. dissertation, University of Lugano, 2009. *2 citations on pages 302 and 340*

[548] M. Lungu, M. Lanza, T. Girba, and R. Heeck, "Reverse engineering super-repositories," in *Working Conf. Reverse Engineering*, 2007. *cited on page 189*

[549] A. Maccari, "Experiences in assessing product family software architectures for evolution," in *Int'l Conf. Software Engineering*, 2002, pp. 585–592. *cited on page 287*

[550] A. MacLean, R. M. Young, V. M. E. Bellotti, and T. P. Moran, "Questions, options, and criteria: elements of design space analysis," *Hum.-Comput. Interact.*, vol. 6, pp. 201–250, September 1991. *cited on page 290*

[551] N. Madani, L. Guerrouj, M. Di Penta, Y. Guéhéneuc, and G. Antoniol, "Recognizing words from source code identifiers using speech recognition techniques," in *European Conf. Software Maintenance and Reengineering*, 2010, pp. 68–77. *cited on page 145*

[552] G. Madey, V. Freeh, and R. Tynan, "The open source software development phenomenon: An analysis based on social network theory," in *Americas Conf. Information Systems*, 2002. *cited on page 173*

[553] N. H. Madhavji, J. F. Ramil, and D. E. Perry, *Software Evolution and Feedback: Theory and Practice*. John Wiley & Sons, 2006. *5 citations on pages 236, 237, 238, 338, and 344*

[554] R. Madsen, S. Sigurdsson, L. Hansen, and J. Larsen, "Pruning the vocabulary for better context recognition," in *Int'l Conf. Pattern Recognition*, 2004, pp. 483–488. *cited on page 145*

[555] S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *Int'l Conf. Program Comprehension*, 1998, pp. 45–. *2 citations on pages 113 and 136*

[556] K. Manikas and K. M. Hansen, "Software ecosystems: A systematic literature review," *J. Systems and Software*, 2012. *cited on page 301*

[557] C. D. Manning, P. Raghavan, and H. Schutze, *Introduction to information retrieval*. Cambridge University Press, 2008, vol. 1. *6 citations on pages 141, 146, 176, 177, 179, and 341*

[558] F. Manola and E. Miller, "RDF Primer," W3C, Tech. Rep., 2004. [Online]. Available: http://www.w3.org/TR/2004/REC-rdf-primer-20040210/ *cited on page 259*

[559] O. Maqbool and H. A. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Trans. Soft. Eng.*, vol. 33, no. 11, pp. 759–780, 2007. *cited on page 113*

[560] J. Maras, J. Carlson, and I. Crnkovic, "Extracting client-side web application code," in *Int'l Conf. World Wide Web*, Apr. 2012, pp. 819–828. *2 citations on pages 211 and 224*

[561] A. Marchetto, P. Tonella, and F. Ricca, "ReAjax: a reverse engineering tool for Ajax Web applications," *IET Software*, vol. 6, no. 1, pp. 33–49, 2012. *2 citations on pages 214 and 224*

[562] E. Marcotte, *Responsive Web Design*. A Book Apart, Jan. 2012. *cited on page 204*

[563] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Int'l Conf. Automated Software Engineering*, 2001, pp. 107–114. *cited on page 154*

[564] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev, "Static techniques for concept location in object-oriented code," in *Int'l Workshop Program Comprehension*, 2005, pp. 33–42. *cited on page 161*

[565] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Working Conf. Reverse Engineering*, 2004, pp. 214–223. *cited on page 161*

[566] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using Latent Semantic Indexing," in *Int'l Conf. Software Engineering*, 2003, pp. 125–135. *cited on page 161*

[567] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Trans. Soft. Eng.*, vol. 34, no. 2, pp. 287–300, 2008. *cited on page 162*

[568] D. Martin and J. R. Cordy, "Analyzing web service similarity using contextual clones," in *Int'l Workshop on Software Clones (IWSC)*, May 2011. *2 citations on pages 212 and 224*

[569] J. Martin and L. Martin, "Web site maintenance with software-engineering tools," in *Int'l Symp. Web Systems Evolution*, Nov. 2001, pp. 126–131. *3 citations on pages 207, 208, and 224*

[570] N. D. Martinez, R. Williams, and J. A. Dunne, *Diversity, complexity, and persistence in large model ecosystems*. Oxford University Press, 2006, pp. 163–185. *2 citations on pages 300 and 314*

[571] G. Maskeri, S. Sarkar, and K. Heafield, "Mining business topics in source code using latent Dirichlet allocation," in *India software engineering conference*, 2008, pp. 113–120. *cited on page 161*

[572] K. Matsudaira, "Making the mobile web faster," *ACM Queue*, vol. 11, no. 1, Jan. 2013. *cited on page 222*

[573] T. J. McCabe, "A complexity measure," *IEEE Trans. Soft. Eng.*, vol. 2, no. 4, pp. 308–320, July 1976. *cited on page 69*

[574] J. McCall, P. Richards, and G. Walters, "Factors in software quality: Vol. 1: Concepts and definitions of software quality," General Electric, Tech. Rep., 1977. *8 citations on pages 66, 68, 70, 92, 338, 339, 340, and 341*

[575] A. K. McCallum, "Mallet: A machine learning for language toolkit," http://mallet.cs.umass.edu, 2002. *cited on page 149*

[576] K. S. McCann, "The diversity-stability debate," *Nature*, vol. 405, pp. 228–233, 2000. *cited on page 300*

[577] C. McDonald, "From Art Form to Engineering Discipline?: A History of US Military Software Development Standards, 1974-1998," *IEEE Annals of the History of Computing*, vol. 32, no. 4, pp. 32–45, 2010. *cited on page 8*

[578] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the Android ecosystem," in *Int'l Conf. Software Maintenance*, 2013. *cited on page 303*

[579] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Trans. Soft. Eng.*, vol. 38, no. 5, pp. 1069–1087, 2012. *cited on page 181*

[580] C. McMillan, D. Poshyvanyk, and M. Revelle, "Combining textual and structural analysis of software artifacts for traceability link recovery," in *Workshop on Traceability in Emerging Forms of Software Engineering*, 2009, pp. 41–48. *cited on page 162*

[581] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *Int'l Conf. Software Engineering*, 2012, pp. 364–374. *2 citations on pages 177 and 179*

[582] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Int'l Conf. Software Engineering*, 2011. *cited on page 181*

[583] A. Mehta and G. T. Heineman, "Evolving legacy system features into fine-grained components," in *Int'l Conf. Software Engineering*, 2002, pp. 417–427. *cited on page 285*

[584] P. Meirelles, C. Santos Jr., J. Miranda, F. Kon, A. Terceiro, and C. Chavez, "A Study of the Relationships between Source Code Metrics and Attractiveness in Free Software Projects," in *Brazilian Symp. Software Engineering*. IEEE Computer Society, 2010, pp. 11–20. *cited on page 77*

[585] P. Mell and T. Grance, "The NIST definition of cloud computing," National Institute of Standards and Technology, Tech. Rep., 2009, http://csrc.nist.gov/groups/SNS/cloud-computing/cloud-def-v15.doc. *cited on page 215*

[586] A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *Working Conf. Reverse Engineering*, 2003, pp. 260–269. *cited on page 331*

[587] T. Mende, F. Beckwermert, R. Koschke, and G. Meier, "Supporting the grow-and-prune model in software product lines evolution using clone detection," in *European Conf. Software Maintenance and Reengineering*, april 2008, pp. 163 –172. *cited on page 286*

[588] D. Méndez, A. Etien, A. Muller, and R. Casallas, "Transformation migration after metamodel evolution," in *Int'l Workshop on Models and Evolution*, 2010. *cited on page 63*

[589] K. Mens and T. Tourwé, *Software Evolution*. Springer, 2008, ch. Evolution Issues in Aspect-Oriented Programming, pp. 203–232. *cited on page x*

[590] T. Mens, *Software Evolution*. Springer, 2008, ch. Introduction and Roadmap: History and Challenges of Software Evolution, pp. 1–11. *5 citations on pages x, 232, 235, 243, and 331*

[591] ——, "Future Research Challenges in Software Evolution," in *Presentation to ERCIM Working Group on Software Evolution*, Sep. 2009, pp. 1–17. *cited on page 4*

[592] T. Mens and S. Demeyer, *Software Evolution*. Springer, 2008. *9 citations on pages ix, x, xi, 202, 203, 212, 217, 219, and 344*

[593] T. Mens, L. Doctors, N. Habra, B. Vanderose, and F. Kamseu, "QUALGEN: Modeling and Analysing the Quality of Evolving Software Systems," in *European Conf. Software Maintenance and Reengineering*. IEEE, 2011, pp. 351–354. *cited on page 86*

[594] T. Mens and M. Goeminne, "Analysing the evolution of social aspects of open source software ecosystems," in *Int'l Workshop on Software Ecosystems*, ser. CEUR Workshop Proceedings. CEUR-WS.org, June 2011, pp. 1–14. *cited on page 303*

[595] T. Mens, Y.-G. Guehéneuc, J. Fernández-Ramil, and M. D'Hondt, "Guest Editors' Introduction: Software Evolution," *IEEE Software*, vol. 27, no. 4, pp. 22–25, Jul. 2010. *cited on page 230*

[596] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Soft. Eng.*, vol. 30, no. 2, pp. 126–139, 2004. *3 citations on pages 128, 202, and 221*

[597] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, 2006. *2 citations on pages 51 and 122*

[598] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," in *Int'l Workshop on Principles of Software Evolution*, 2005. *3 citations on pages 4, 230, and 338*

[599] A. Mesbah, "Analysis and testing of Ajax-based single-page Web applications," Ph.D. dissertation, Delft University of Technology, The Netherlands, 2009. *cited on page 227*

[600] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Int'l Conf. Software Engineering*, May 2011, pp. 561–570. *cited on page 226*

[601] A. Mesbah and A. van Deursen, "Migrating multi-page Web applications to single-page Ajax interfaces," in *European Conf. Software Maintenance and Reengineering*, Mar. 2007, pp. 181–190. *3 citations on pages 214, 223, and 224*

[602] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based Web applications through dynamic analysis of user interface state changes," *ACM Trans. Web*, vol. 6, no. 1, Mar. 2012. *2 citations on pages 209 and 224*

[603] D. Messerschmitt and C. Szyperski, *Software ecosystem: Understanding and indispensable technology and industry*. MIT Press, 2003. *2 citations on pages 301 and 302*

[604] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *Journal of Chemical Physics*, vol. 21, pp. 1087–1092, 1953. *cited on page 107*

[605] B. Meyer, "Schema evolution: Concepts, terminology, and solutions," *IEEE Computer*, vol. 29, no. 10, pp. 119–121, 1996. *cited on page 34*

[606] K. Michael and K. W. Miller, "Big data: New opportunities and new challenges [guest editors' introduction]," *Computer*, vol. 46, no. 6, pp. 22–24, 2013. *cited on page 329*

[607] Z. Michalewicz and D. B. Fogel, *How to Solve It: Modern Heuristics*, 2nd ed. Springer, 2004. *4 citations on pages 105, 106, 107, and 109*

[608] T. Mikkonen and A. Taivalsaari, "Apps vs. open web: The battle of the decade," in *Workshop on Mobile Software Engineering*, Oct. 2011, http://mobileseworkshop.org/papers/6-Mikkonen_Taivalsaari.pdf. *cited on page 216*

[609] S. C. Misra, "Modeling Design/Coding Factors That Drive Maintainability of Software Systems," *Software Quality Control*, vol. 13, no. 3, pp. 297–320, Sep. 2005. *cited on page 76*

[610] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Trans. Soft. Eng.*, vol. 32, no. 3, pp. 193–208, 2006. *6 citations on pages 113, 114, 115, 116, 118, and 136*

[611] M. Mitchell, *An Introduction to Genetic Algorithms*, 3rd ed. A Bradford Book, 1998. *cited on page 298*

[612] T. Mitchell, *Machine Learning*. McGraw-Hill, 1997. *2 citations on pages 186 and 194*

[613] A. Mockus, R. Fielding, and J. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Trans. Software Engineering and Methodology*, vol. 11, no. 3, pp. 309–346, 2002. *2 citations on pages 306 and 308*

[614] I. H. Moghadam and M. Ó Cinnéide, "Code-imp: A tool for automated search-based refactoring," in *Workshop on Refactoring Tools (WRT)*, 2011. *2 citations on pages 129 and 131*

[615] ——, "Automated refactoring using design differencing," in *European Conf. Software Maintenance and Reengineering*, 2012. *cited on page 129*

[616] M. Moon, H. S. Chae, T. Nam, and K. Yeom, "A metamodeling approach to tracing variability between requirements and architecture in software product lines," in *Int'l Conf. Computer and Information Technology*. IEEE, 2007, pp. 927–933. *cited on page 289*

[617] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, *Software Evolution*. Springer, 2008, ch. On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension, pp. 173–202. *3 citations on pages x, 330, and 331*

[618] I. Moore, "Automatic inheritance hierarchy restructuring and method refactoring," in *Int'l Conf. Object-Oriented Programming Systems, Languages and Applications*. ACM Press, 1996, pp. 235–250. *cited on page 128*

[619] K. Mordal-Manet, F. Balmas, S. Denier, S. Ducasse, H. Wertz, J. Laval, F. Bellingard, and P. Vaillergues, "The Squale Model – A Practice-based Industrial Quality Model," in *Int'l Conf. Software Maintenance*. IEEE Computer Society, 2009, pp. 531–534. *5 citations on pages 66, 67, 69, 77, and 336*

[620] P. Morrison and E. R. Murphy-Hill, "Is programming knowledge related to age? an exploration of Stack Overflow," in *Int'l Conf. Mining Software Repositories*, 2013, pp. 69–72. *cited on page 331*

[621] H. A. Müller, H. M. Kienle, and U. Stege, *Autonomic Computing: Now You See it, Now You Don't–Design and Evolution of Autonomic Software Systems*, ser. Lect. Notes in Computer Science. Springer, 2009, vol. 5413, pp. 32–54. *cited on page 230*

[622] H. A. Müller, M. Pezzè, and M. Shaw, "Visibility of Control in Adaptive Systems," in *Int'l Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS)*, 2008, pp. 23–26. *4 citations on pages 230, 237, 239, and 241*

[623] G. C. Murphy, "Houston: We are in overload," in *Int'l Conf. Software Maintenance*. IEEE, 2007, p. 1. *cited on page 135*

[624] G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard, "Separating features in source code: an exploratory study," in *Int'l Conf. Software Engineering*, 2001, pp. 275–284. *cited on page 285*

[625] R. M. Murray, Ed., *Control in an Information Rich World: Report of the Panel on Future Directions in Control, Dynamics, and Systems*. Society for Industrial and Applied Mathematics, 2003. *2 citations on pages 230 and 238*

[626] C. R. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," *Physical Reviews E*, vol. 68, p. 046116, Oct 2003. *cited on page 331*

[627] J. Mylopoulos, "The Requirements Problem Revisited," in *Presentation to IFIP Working Group 2.9*, Cancun, 2011. *cited on page 5*

[628] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis, "Telos: Representing Knowledge About Information Systems," *ACM Trans. Information Systems*, vol. 8, pp. 325–362, 1990. *cited on page 10*

[629] M. Naaman, J. Boase, and C.-H. Lai, "Is it really about me? Message content in social awareness streams," in *Int'l Conf. Computer Supported Cooperative Work*, 2010. *cited on page 185*

[630] N. Nagappan, T. Ball, and B. Murphy, "Using Historical In-Process and Product Metrics for Early Estimation of Software Failures," in *Int'l Symp. Software Reliability Engineering*. IEEE Computer Society, 2006, pp. 62–74. *cited on page 66*

[631] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, "Evolution patterns of open-source software systems and communities," in *Int'l Workshop on Principles of Software Evolution*. ACM, 2002, pp. 76–85. *3 citations on pages 303, 307, and 310*

[632] V. Nanda and N. H. Madhavji, "The Impact of Environmental Evolution on Requirements Changes," in *Int'l Conf. Software Maintenance*, 2002, pp. 452–461. *cited on page 11*

[633] A. Narayanan, T. Levendovszky, D. Balasubramanian, and G. Karsai, "Automatic domain model migration to manage metamodel evolution," in *Model Driven Engineering Languages and Systems*, ser. Lect. Notes in Computer Science, vol. 5795. Springer, 2009, pp. 706–711. *cited on page 52*

[634] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming Q&A in StackOverflow," in *Int'l Conf. Software Maintenance*, 2012, pp. 25–34. *cited on page 172*

[635] P. Naur and B. Randell, *Software Engineering*. NATO, Scientific Affairs Division, Brussels, 1969, report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968. *cited on page 340*

[636] I. Navarro, N. Leveson, and K. Lunqvist, "Semantic decoupling: reducing the impact of requirement changes," *Int'l Conf. Req. Engineering*, vol. 15, no. 4, pp. 419–437, 2010. *cited on page 12*

[637] D. Neary, V. David, and N. Consulting, "The GNOME census: Who writes GNOME?" in *GNOME users and developers European conference*, 2010. *cited on page 323*

[638] C. L. Nehaniv, J. Hewitt, B. Christianson, and P. Wernick, "What software evolution and biological evolution don't have in common," in *Int'l IEEE Workshop on Software Evolvablility*, 2006, pp. 58–65. *2 citations on pages 298 and 313*

[639] A. Neitsch, K. Wong, and M. W. Godfrey, "Build system issues in multilanguage software," in *Int'l Conf. Software Maintenance*, 2012, pp. 140–149. *cited on page 331*

[640] S. Neu, M. Lanza, L. Hattori, and M. D'Ambros, "Telling stories about GNOME with Complicity," in *Working Conf. Software Visualisation*. IEEE, 2011, pp. 1–8. *2 citations on pages 306 and 316*

[641] S. Neuhaus and T. Zimmermann, “Security trend analysis with CVE topic models,” in *Int’l Symp. Software Reliability Engineering*, 2010, pp. 111–120. *cited on page 152*

[642] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulezsa, and P. Borba, “Investigating the safe evolution of software product lines,” in *Int’l Conf. Generative Programming*, 2011, pp. 33–42. *cited on page 292*

[643] A. Ngo-The and G. Ruhe, “A systematic approach for solving the wicked problem of software release planning,” *Soft Computing*, vol. 12, pp. 95–108, 2008. *cited on page 290*

[644] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, “A topic-based approach for narrowing the search space of buggy files from a bug report,” in *Int’l Conf. Automated Software Engineering*, 2011, pp. 263–272. *cited on page 162*

[645] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, “Duplicate bug report detection with a combination of information retrieval and topic modeling,” in *Int’l Conf. Automated Software Engineering*, 2012, pp. 70–79. *cited on page 192*

[646] L. Northrop, P. Feiler, R. Gabriel, J. Goodenough, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau, “Ultra-Large-Scale Systems–The Software Challenge of the Future,” Carnegie Mellon University Software Engineering Institute (SEI), Tech. Rep., 2006. *2 citations on pages 230 and 337*

[647] B. A. Nuseibeh and S. M. Easterbrook, “Requirements Engineering: A Roadmap,” in *Workshop on The Future of Software Engineering*, 2000, pp. 35–46. *cited on page 23*

[648] B. A. Nuseibeh, S. M. Easterbrook, and A. Russo, “Making inconsistency respectable in software development,” *J. Systems and Software*, vol. 58, no. 2, pp. 171–180, 2001. *cited on page 19*

[649] M. Ó Cinnéide, D. Boyle, and I. H. Moghadam, “Automated refactoring for testability,” in *Workshop on Refactoring and Testing*, 2011. *5 citations on pages 128, 129, 130, 132, and 136*

[650] Object Management Group , *Model Driven Architecture (MDA)*. *cited on page 218*

[651] ——, *Unified Modeling Language (UML), Infrastructure. Version 2.0*, Jul. 2005. *cited on page 43*

[652] ——, *Unified Modeling Language (UML), Superstructure. Version 2.0*, formal/2005-07-04, Jul. 2005. *2 citations on pages 34 and 43*

[653] ——, *Business Process Model and Notation (BPMN). Version 2.0*, Jan. 2011. *cited on page 34*

[654] ——, *Meta Object Facility (MOF) Core Specification. Version 2.4.1*, Aug. 2011. *cited on page 35*

[655] L. O’Brien and D. Smith, “MAP and OAR methods: Techniques for developing core assets for software product lines from existing assets,” Software Engineering Institute – Carnegie Mellon University, Technical Note SEI/CMU-2002-TN-007, April 2002. *2 citations on pages 274 and 286*

[656] M. O’Keeffe and M. Ó Cinnéide, “A stochastic approach to automated design improvement,” in *Int’l Conf. Principles and Practice of Programming in Java*. Computer Science Press, 2003, pp. 59–62. *cited on page 129*

[657] ——, “Getting the most from search-based refactoring,” in *Genetic and Evolutionary Computation Conf.* ACM, 7-11 July 2007, pp. 1114–1120. *4 citations on pages 130, 131, 132, and 136*

[658] ——, “Search-based refactoring: An empirical study,” *J. Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 5, pp. 345–364, September 2008, special Issue Search Based Software Engineering. *2 citations on pages 129 and 133*

[659] ——, “Search-based refactoring for software maintenance,” *J. Systems and Software*, vol. 81, no. 4, pp. 502–516, April 2008. *6 citations on pages 128, 129, 130, 132, 133, and 136*

[660] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, “Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes,” *IEEE Trans. Soft. Eng.*, vol. 33, no. 6, pp. 402–419, 2007. *cited on page 77*

[661] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the equivalence of information retrieval methods for automated traceability link recovery," in *Int'l Conf. Program Comprehension*, 2010, pp. 68–71. *cited on page 162*

[662] P. Oman and J. Hagemeister, "Metrics for Assessing a Software System's Maintainability," in *Int'l Conf. Software Maintenance*. IEEE Computer Society Press, 1992, pp. 337–344. *2 citations on pages 75 and 80*

[663] ——, "Construction and Testing of Polynomials Predicting Software Maintainability," *J. Systems and Software*, vol. 24, no. 3, pp. 251–266, Mar. 1994. *2 citations on pages 68 and 76*

[664] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at UrbanaChampaign, 1992. *2 citations on pages 128 and 221*

[665] Open Service Oriented Architecture, "SCA Assembly Model version 1.0," http://www.osoa.org, 2007. *cited on page 257*

[666] A. Oram and G. Wilson, *Making Software: What Really Works, and Why We Believe It*. O'Reilly Media, 1998. *cited on page 344*

[667] T. O'Reilly, "What is web 2.0. design patterns and business models for the next generation of software." http://oreilly.com/web2/archive/what-is-web-20.html, 2005. *cited on page 164*

[668] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54–62, 1999. *cited on page 248*

[669] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-Based Runtime Software Evolution," in *Int'l Conf. Software Engineering*. IEEE, 1998, pp. 177–186. *No citation*

[670] ——, "Runtime Software Adaptation: Framework, Approaches, and Styles," in *Int'l Conf. Software Engineering*. ACM, 2008, pp. 899–910. *cited on page 248*

[671] G. Orians, "Diversity, stability, and maturity in natural ecosystems," in *Unifying concepts in ecology*, W. van Dobben and R. Lowe-McConnel, Eds., 1975, pp. 139–150. *cited on page 300*

[672] O. Ormandjieva, I. Hussain, and L. Kosseim, "Toward a text classification system for the quality assessment of software requirements written in natural language," in *Int'l Workshop on Software Quality Assurance*, 2007, pp. 39–45. *cited on page 154*

[673] H. Orr, "Fitness and its role in evolutionary genetics," *Nature Reviews Genetics*, vol. 10, no. 8, pp. 531–539, August 2009. *cited on page 311*

[674] I. Ozkaya, L. Bass, R. S. Sangwan, and R. L. Nord, "Making Practical Use of Quality Attribute Information," *IEEE Software*, vol. 25, no. 2, pp. 25–33, 2008. *cited on page 86*

[675] D. Pagano and W. Maalej, "How Do Developers Blog? An Exploratory Study," in *Int'l Conf. Mining Software Repositories*, 2011. *2 citations on pages 168 and 172*

[676] L. Page, S. Brin, R. Motwani, and T. Winograd, "Pagerank citation ranking: Bringing order to the web," in *Technical Report, Stanford University*, 1998. *cited on page 191*

[677] B. Pang and L. Lee, *Opinion Mining and Sentiment Analysis*. NOW Publisher, 2008. *cited on page 188*

[678] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus, "Using Control Theory to Achieve Service Level Objectives In Performance Management," *Real-Time Systems*, vol. 23, pp. 127–141, 2002. *2 citations on pages 250 and 251*

[679] D. L. Parnas, "Software Aspects of Strategic Defense Systems," *Comm. ACM*, vol. 28, pp. 1326–1335, 1985. *cited on page 8*

[680] C. Parnin and C.Treude, "Measuring API documentation on the web," in *Int'l Workshop on Web 2.0 for Software Engineering*, 2011, pp. 25–30. *cited on page 172*

[681] C. Parnin, C.Treude, and M. Storey, "Blogging developer knowledge: Motivations, challenges and future directions," in *Int'l Conf. Program Comprehension*, 2013. *cited on page 172*

[682] M. Pascual and J. A. Dunne, Eds., *Ecological networks: linking structure to dynamics in food webs*. Oxford Univ. Press, 2006. *cited on page 299*

[683] C. Pautasso and E. Wilde, *REST: From Research to Practice*. Springer, 2011, ch. Introduction, pp. 1–18. *cited on page 213*

[684] J. Perez, R. Deshayes, M. Goeminne, and T. Mens, “Seconda: Software ecosystem analysis dashboard,” in *European Conf. Software Maintenance and Reengineering*, T. Mens, A. Cleve, and R. Ferenc, Eds., 2012, pp. 527–530. *cited on page 303*

[685] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick, *Version Control with Subversion*. O’Reilly Media, 2008. *cited on page 143*

[686] A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, and S. Kowalewski, “Model-driven support for product line evolution on feature level,” *J. Systems and Software*, vol. 85, no. 10, pp. 2261–2274, October 2012, special Issue on Automated Software Evolution. *3 citations on pages 278, 281, and 283*

[687] A. Pleuss, S. Wollny, and G. Botterweck, “Model-driven development and evolution of customized user interfaces,” in *Symp. Engineering Interactive Computing Systems (EICS)*, 2013, pp. 13–22. *cited on page 331*

[688] K. Pohl, G. Boeckle, and F. van der Linden, *Software Product Line Engineering : Foundations, Principles, and Techniques*. Springer, 2005. *4 citations on pages 266, 267, 277, and 285*

[689] W. Poncin, A. Serebrenik, and M. G. J. van den Brand, “Process mining software repositories,” in *European Conf. Software Maintenance and Reengineering*, 2011, pp. 5–14. *2 citations on pages 303 and 307*

[690] L. Ponzanelli, A. Bacchelli, and M. Lanza, “Leveraging crowd knowledge for software comprehension and development,” in *European Conf. Software Maintenance and Reengineering*, 2013, pp. 57–66. *cited on page 181*

[691] D. Poole, “A logical framework for default reasoning,” *Artificial Intelligence*, vol. 36, no. 1, pp. 27–47, 1988. *cited on page 18*

[692] M. F. Porter, “An algorithm for suffix stripping,” *Program*, vol. 14, no. 3, pp. 130–137, 1980. *2 citations on pages 145 and 186*

[693] D. Poshyvanyk, M. Di Penta, and H. Kagdi, Eds., *Int’l Workshop on Traceability in Emerging Forms of Software Engineering*, 2011. *cited on page 24*

[694] D. Poshyvanyk and M. Grechanik, “Creating and evolving software by searching, selecting and synthesizing relevant source code,” in *Int’l Conf. Software Engineering*, 2009, pp. 283–286. *cited on page 154*

[695] D. Poshyvanyk, Y. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, “Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval,” *IEEE Trans. Soft. Eng.*, vol. 33, no. 6, pp. 420–432, 2007. *cited on page 146*

[696] D. Poshyvanyk and A. Marcus, “Combining formal concept analysis with information retrieval for concept location in source code,” in *Int’l Conf. Program Comprehension*, 2007, pp. 37–48. *cited on page 161*

[697] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Guéhéneuc, and G. Antoniol, “Combining probabilistic ranking and Latent Semantic Indexing for feature identification,” in *Int’l Conf. Program Comprehension*, 2006, pp. 137–148. *cited on page 161*

[698] D. Posnett, R. D’Souza, P. Devanbu, and V. Filkov, “Dual ecological measures of focus in software development,” in *Int’l Conf. Software Engineering*. IEEE, 2013, pp. 452–461. *2 citations on pages 298 and 314*

[699] D. Posnett, V. Filkov, and P. Devanbu, “Ecological inference in empirical software engineering,” in *Int’l Conf. Automated Software Engineering*. IEEE Computer Society, 2011, pp. 362–371. *2 citations on pages 85 and 298*

[700] D. Posnett, E. Warburg, P. T. Devanbu, and V. Filkov, “Mining stack exchange: Expertise is evident from initial contributions,” in *SocialInformatics*, 2012, pp. 199–204. *cited on page 331*

[701] K. Praditwong, M. Harman, and X. Yao, “Software module clustering as a multi-objective search problem,” *IEEE Trans. Soft. Eng.*, vol. 37, no. 2, pp. 264–282, 2011. *4 citations on pages 113, 116, 118, and 136*

[702] P. Prasarnphanich and M. L. Gillenson, "The hybrid clicks and bricks business model," *Comm. ACM*, vol. 46, no. 12ve, pp. 178–185, Dec. 2003. *cited on page 204*

[703] P. K. Prasetyo, D. Lo, P. Achananuparp, Y. Tian, and E.-P. Lim, "Automatic classification of software related microblogs," in *Int'l Conf. Software Maintenance*, 2012, pp. 596–599. *3 citations on pages 163, 165, and 181*

[704] D. Qiu, B. Li, and Z. Su, "An empirical analysis of the co-evolution of schema and code in database applications," in *Joint European Software Engineering Conf. and ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*. ACM , 2013. *cited on page 329*

[705] N. A. Qureshi, I. J. Jureta, and A. Perini, "Requirements Engineering for Self-Adaptive Systems : Core Ontology and Problem Statement," in *Int'l Conf. Advanced Informations Systems Engineering*, 2011, pp. 1–15. *2 citations on pages 17 and 22*

[706] V. Rajlich, *Software Engineering - The Current Practice*. Chapman & Hall/CRC, 2011. *cited on page 343*

[707] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Int'l Workshop Program Comprehension*, 2002, pp. 271–278. *cited on page 150*

[708] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Int'l Conf. Mining Software Repositories*, 2011, pp. 43–52. *cited on page 162*

[709] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *Int'l Conf. Software Engineering*, 2010, pp. 505–514. *cited on page 153*

[710] R. Reicherdt and S. Glesner, "Slicing MATLAB Simulink models," in *Int'l Conf. Software Engineering*, 2012, pp. 551–561. *cited on page 329*

[711] D. J. Reifer, Ed., *Software Maintenance Success Recipes*. Auerbach Publications, 2011. *cited on page 343*

[712] M. Revelle, B. Dit, and D. Poshyvanyk, "Using data fusion and web mining to support feature location in software," in *Int'l Conf. Program Comprehension*, 2010, pp. 14–23. *cited on page 161*

[713] M. Revelle and D. Poshyvanyk, "An exploratory study on assessing feature location techniques," in *Int'l Conf. Program Comprehension*, 2009, pp. 218–222. *cited on page 161*

[714] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *Int'l Symp. Empirical Software Engineering and Measurement*. IEEE Computer Society, 2009, pp. 367–377. *2 citations on pages 75 and 76*

[715] F. Ricca, "Analysis, testing, and re-structuring of Web applications," Ph.D. dissertation, Universitá degli Studi di Genova, Italy, Sep. 2003. *cited on page 227*

[716] F. Ricca and A. Marchetto, "Heroes in FLOSS projects: An explorative study," in *Working Conf. Reverse Engineering*, 2010, pp. 155–159. *cited on page 174*

[717] F. Ricca and P. Tonella, "Analysis and testing of Web applications," in *Int'l Conf. Software Engineering*, May 2001, pp. 25–34. *2 citations on pages 208 and 224*

[718] ——, "Using clustering to support the migration from static to dynamic web pages," in *Int'l Workshop Program Comprehension*, May 2003, pp. 207–216. *2 citations on pages 211 and 224*

[719] ——, "Anomaly detection in Web applications: a review of already conducted case studies," in *European Conf. Software Maintenance and Reengineering*, Mar. 2005, pp. 385–394. *cited on page 203*

[720] S. Rinaldi, Y. Muratori, and Y. Kuznetsov, "Multiple attractors, catastrophes and chaos in seasonally perturbed predator-prey communities," *Bulletin of Mathematical Biology*, vol. 55, no. 1, pp. 15–35, 1993. *2 citations on pages 300 and 314*

[721] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation?: the case of a Smalltalk ecosystem," in *Int'l Symp. Foundations of Software Engineering*. ACM , 2012. *cited on page 303*

[722] G. Robles and J. M. González-Barahona, "Developer identification methods for integrated data from various sources," in *Int'l Conf. Mining Software Repositories*. ACM, 2005. *cited on page 318*

[723] G. Robles, J. M. Gonzalez-Barahona, and I. Herraiz, “Evolution of the core team of developers in libre software projects,” in *Int’l Conf. Mining Software Repositories*. IEEE Computer Society, 2009, pp. 167–170. *3 citations on pages 303, 307, and 310*

[724] R. Rodriguez-Echeverra, J. M. Conejero, P. J. Clemente, M. D. Villalobos, and F. Sanchez-Figueroa, “Generation of WebML hypertext models from legacy web applications,” in *Int’l Symp. Web Systems Evolution*. IEEE Computer Society, 2012, pp. 91–95. *2 citations on pages 219 and 224*

[725] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel, “Web service modeling ontology,” *Appl. Ontol.*, vol. 1, no. 1, pp. 77–106, Jan. 2005. *cited on page 34*

[726] L. M. Rose, A. Etien, D. Mendez, D. S. Kolovos, F. A. C. Polack, and R. F. Paige, “Comparing Model-Metamodel and Transformation-Metamodel Co-evolution,” in *Model and Evolution Wokshop*, Olso, Norway, Oct. 2010. *cited on page 63*

[727] L. M. Rose, M. Herrmannsdoerfer, S. Mazanek, P. Van Gorp, S. Buchwald, T. Horn, E. Kalnina, A. Koch, K. Lano, B. Schaetz, and M. Wimmer, “Graph and model transformation tools for model migration,” *Software and Systems Modeling*, pp. 1–37, 2012. *cited on page 58*

[728] L. M. Rose, M. Herrmannsdoerfer, J. Williams, D. Kolovos, K. Garcés, R. Paige, and F. Polack, “A comparison of model migration tools,” in *Model Driven Engineering Languages and Systems*, ser. Lect. Notes in Computer Science, vol. 6394. Springer, 2010, pp. 61–75. *cited on page 60*

[729] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. Polack, “Model migration with Epsilon Flock,” in *Int’l Conf. Model Transformation (ICMT)*. Springer, 2010, pp. 184–198. *3 citations on pages 44, 52, and 57*

[730] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack, “An analysis of approaches to model migration,” in *Models and Evolution (MoDSE-MCCM) Workshop*, 2009, pp. 6–15. *cited on page 48*

[731] M. Rosen-Zvi, T. Griffiths, M. Steyvers, and P. Smyth, “The author-topic model for authors and documents,” in *Conf. Uncertainty in Artificial Intelligence*, 2004, pp. 487–494. *cited on page 161*

[732] E. Rosenberg, O. Koren, L. Reshef, R. Efrony, and I. Zilber-Rosenberg, “The role of microorganisms in coral health, disease and evolution,” *Nature Reviews Microbiology*, vol. 5, no. 5, pp. 355–362, 2007. *2 citations on pages 311 and 325*

[733] S. Roubtsov, A. Serebrenik, A. Mazoyer, M. G. J. van den Brand, and E. Roubtsova, “I2SD: reverse engineering sequence diagrams from Enterprise JavaBeans with interceptors,” *IET Software*, vol. 7, pp. 150–166, June 2013. *cited on page 331*

[734] A. Rountev and B. H. Connell, “Object naming analysis for reverse-engineered sequence diagrams,” in *Int’l Conf. Software Engineering*, 2005, pp. 254–263. *cited on page 331*

[735] J. Rubin and M. Chechik, “Combining related products into product lines,” *Fundamental Approaches to Software Engineering*, pp. 285–300, 2012. *cited on page 286*

[736] G. Ruhe, *Product Release Planning Methods, Tools and Applications*. Auerbach Publications, 2010. *2 citations on pages 288 and 290*

[737] P. Runeson, M. Alexandersson, and O. Nyholm, “Detection of duplicate defect reports using natural language processing,” in *Int’l Conf. Software Engineering*, 2007, pp. 499–510. *2 citations on pages 153 and 192*

[738] C. R. Rupakheti and D. Hou, “Evaluating forum discussions to inform the design of an API critic,” in *Int’l Conf. Program Comprehension*, 2012, pp. 53–62. *cited on page 171*

[739] M. Salehie and L. Tahvildari, “Self-Adaptive Software: Landscape and Research Challenges,” *ACM Trans. Autonomous and Adaptive Systems*, vol. 4, pp. 14:1–14:42, 2009. *2 citations on pages 243 and 244*

[740] G. Salton, *Introduction to modern information retrieval*. Mcgraw-Hill, 1983. *cited on page 147*

[741] G. Salton, A. Wong, and C. S. Yang, “A vector space model for automatic indexing,” *Comm. ACM*, vol. 18, no. 11, p. 620, 1975. *cited on page 147*

[742] A. Sampaio, A. Rashid, and P. Rayson, “Early-aim: An approach for identifying aspects in requirements,” in *Int’l Conf. Req. Engineering*, 2005, pp. 487–488. *cited on page 154*

[743] P. Samuelson, “Statutory damages as a threat to innovation,” *Comm. ACM*, vol. 56, no. 7, pp. 24–26, Jul. 2013. *cited on page 205*

[744] A. Sardinha, R. Chitchyan, N. Weston, P. Greenwood, and A. Rashid, “EA-Analyzer: Automating conflict detection in aspect-oriented requirements,” in *Int’l Conf. Automated Software Engineering*, 2009, pp. 530–534. *cited on page 154*

[745] G. Saridis and H. E. Stefanou, “A Hierarchically Intelligent Control for a Bionic Arm,” in *IEEE Conf. Decision and Control including the 14th Symp. on Adaptive Processes*. IEEE, 1975, pp. 99–104. *cited on page 247*

[746] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, “Tesseract: Interactive visual exploration of socio-technical relationships in software development,” in *Int’l Conf. Software Engineering*, 2009. *cited on page 189*

[747] T. K. Satyananda, D. Lee, S. Kang, and S. I. Hashmi, “Identifying traceability between feature model and software architecture in software product line using formal concept analysis,” in *ICCSA Workshops*, 2007, pp. 380–388. *cited on page 289*

[748] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein, “Requirements-Aware Systems. A Research Agenda for RE for Self-Adaptive Systems,” in *Int’l Conf. Req. Engineering*. IEEE, 2010, pp. 95–103. *cited on page 254*

[749] I. Schäfer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, “Delta-oriented programming of software product lines,” in *Int. Software Product Line Conf.*, 2010, pp. 77–91. *cited on page 280*

[750] A. Schauerhuber, M. Wimmer, E. Kapsammer, W. Schwinger, and W. Retschitzegger, “Bridging WebML to model-driven engineering: from document type definitions to meta object facility,” *IET Software*, vol. 1, no. 3, pp. 81–97, 2007. *cited on page 218*

[751] D. Schenk and M. Lungu, “Geo-locating the knowledge transfer in Stack Overflow,” in *Social Software Engineering*, 2013. *cited on page 331*

[752] H. Schmid and O. Donnerhak, “OOHDMDA – An MDA Approach for OOHDM,” in *Int’l Conf. Web Engineering*, ser. Lect. Notes in Computer Science, vol. 3579. Springer, 2005, pp. 569–574. *cited on page 218*

[753] K. Schmid and H. Eichelberger, “A requirements-based taxonomy of software product line evolution,” *Electronic Communications of the EASST*, vol. 8, 2008. *3 citations on pages 270, 271, and 272*

[754] K. Schmid and M. Verlage, “The economic impact of product line adoption and evolution,” *IEEE Software*, vol. 19, no. 4, pp. 50 – 57, jul/aug 2002. *2 citations on pages 276 and 283*

[755] K. Schmid, R. Rabiser, and P. Grünbacher, “A comparison of decision modeling approaches in product lines,” in *Int’l Workshop on Variability Modeling of Software-Intensive Systems*, 2011, pp. 119–126. *cited on page 267*

[756] D. C. Schmidt, “Model-driven engineering,” *IEEE Computer*, vol. 39, no. 2, pp. 25–31, Feb. 2006. *cited on page 218*

[757] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux, “Feature diagrams: A survey and a formal semantics,” in *Int’l Conf. Req. Engineering*, 2006, pp. 136–145. *cited on page 267*

[758] M. Schubanz, A. Pleuss, G. Botterweck, and C. Lewerentz, “Modeling rationale over time to support product line evolution planning,” in *Int’l Workshop on Variability Modeling of Software-Intensive Systems*, 2012, pp. 193–199. *cited on page 291*

[759] M. Schubanz, A. Pleuss, L. Pradhan, G. Botterweck, and A. K. Thurimella, “Model-driven planning and monitoring of long-term software product line evolution,” in *Int’l Workshop on Variability Modeling of Software-Intensive Systems*, 2013, pp. 18:1–18:5. *cited on page 291*

[760] R. Sebastiani, P. Giorgini, and J. Mylopoulos, “Simple and Minimum-Cost Satisfiability for Goal Models,” in *Int’l Conf. Advanced Informations Systems Engineering*, 2004, pp. 20–35. *3 citations on pages 16, 22, and 24*

[761] C. Seidl, “Evolution in feature-oriented model-based software product line engineering,” Diploma Thesis, TU Dresden, 2011. *cited on page 281*

[762] C. Seidl, F. Heidenreich, and U. Aßmann, "Co-evolution of models and feature mapping in software product lines," in *Int. Software Product Line Conf.*, 2012, pp. 76–85. *cited on page 292*

[763] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani, "Reconfigurable SCA Applications with the FraSCAti Platform," in *IEEE Int'l Conf. Services Computing (SCC)*. IEEE, 2009, pp. 268–275. *cited on page 258*

[764] C. Semple and M. Steel, *Phylogenetics*, ser. Oxford Lecture Series in Mathematics and Its Applications. Oxford University Press, 2003. *cited on page 311*

[765] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Genetic and Evolutionary Computation Conf.* ACM, 8-12 July 2006, pp. 1909–1916. *5 citations on pages 128, 129, 133, 134, and 136*

[766] A. Serebrenik, A. Mishra, T. Delissen, and M. Klabbers, "Requirements Certification for Offshoring Using LSPCM," in *Int'l Conf. Quality of Information and Communications Technology*. IEEE Computer Society, 2010, pp. 177–182. *cited on page 86*

[767] A. Serebrenik, S. A. Roubtsov, E. E. Roubtsova, and M. G. J. van den Brand, "Reverse engineering sequence diagrams for Enterprise JavaBeans with business method interceptors," in *Working Conf. Reverse Engineering*, 2009, pp. 269–273. *cited on page 331*

[768] A. Serebrenik and M. G. J. van den Brand, "Theil index for aggregation of software metrics values," in *Int'l Conf. Software Maintenance*, 2010, pp. 1–9. *2 citations on pages 325 and 331*

[769] A. Serebrenik, M. G. J. van den Brand, and B. Vasilescu, "Seeing the Forest for the Trees with New Econometric Aggregation Techniques," *ERCIM News*, vol. 88, p. 21, 2012. *cited on page 85*

[770] N. Serrano and I. Ciordia, "Bugzilla, ITracker, and other bug trackers," *IEEE Software*, vol. 22, no. 2, pp. 11–13, 2005. *cited on page 142*

[771] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin, "Assisting developers of big data analytics applications when deploying on Hadoop clouds," in *Int'l Conf. Software Engineering*, 2013, pp. 402–411. *cited on page 330*

[772] M. Shaw, "Beyond Objects: A Software Design Paradigm Based on Process Control," *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 1, pp. 27–38, 1995. *2 citations on pages 239 and 241*

[773] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *Int'l Conf. Software Engineering*, 2011, pp. 461–470. *2 citations on pages 285 and 287*

[774] K. Shibata, K. Rinsaka, T. Dohi, and H. Okamura, "Quantifying Software Maintainability Based on a Fault-Detection/Correction Model," in *Pacific Rim Int'l Symp. Dependable Computing*. IEEE Computer Society, 2007, pp. 35–42. *cited on page 76*

[775] E. Shihab, N. Bettenburg, B. Adams, and A. Hassan, "On the central role of mailing lists in open source projects: An exploratory study," in *Int'l Workshop on Knowledge Collaboration in Software Development*, 2009, pp. 91–103. *2 citations on pages 142 and 146*

[776] E. Shihab, Z. M. Jiang, and A. E. Hassan, "On the use of IRC channels by developers of the GNOME GTK+ open source project," in *Int'l Conf. Mining Software Repositories*, 2009. *cited on page 143*

[777] ——, "Studying the use of developer irc meetings in open source projects," in *Int'l Conf. Software Maintenance*, 2009. *cited on page 143*

[778] D. Simon and T. Eisenbarth, "Evolutionary introduction of software product lines," in *Int. Software Product Line Conf.*, August 2002, pp. 272–283. *2 citations on pages 284 and 285*

[779] P. H. A. Sneath, "Cladistic representation of reticulate evolution," *Systematic Zoology*, vol. 24, no. 3, pp. 360–368, 1975. *cited on page 311*

[780] H. Sneed, *Software Testing in the Cloud: Perspectives on an Emerging Discipline*. IGI Global, Nov. 2012, ch. Testing Web Services in the Cloud, pp. 136–173. *cited on page 224*

[781] H. M. Sneed, "20 years of software-reengineering: A resume," in *10th Workshop Software Reengineering (WSR'08)*, may 2008, pp. 115–124. *cited on page 203*

[782] ——, "A pilot project for migrating COBOL code to web services," *J. Software Tools for Technology Transfer*, vol. 11, no. 6, pp. 441–451, dec 2009. *2 citations on pages 218 and 224*

[783] H. M. Sneed and S. Huang, "WSDLTest – a tool for testing web services," in *Int'l Symp. Web Systems Evolution*, Sep. 2006, pp. 14–21. *cited on page 224*

[784] Software Engineering Institute, "SPL Hall of Fame," Web site, 2008, http://splc.net/fame.html. *cited on page 266*

[785] ——, "A framework for software product line practice, version 5.0," 2011. [Online]. Available: http://www.sei.cmu.edu/productlines/frame_report/index.html *2 citations on pages 284 and 286*

[786] A. Solomon, M. Litoiu, J. Benayon, and A. Lau, "Business Process Adaptation on a Tracked Simulation Model," in *Conf. Center for Advanced Studies on Collaborative Research*. ACM , 2010. *2 citations on pages 250 and 252*

[787] I. Sommerville and P. Sawyer, *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, 1997. *cited on page 9*

[788] V. E. S. Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos, "Awareness Requirements for Adaptive Systems," in *Int'l Symp. Software Engineering for Adaptive and Self-Managing Systems*, 2011, pp. 1–10. *cited on page 28*

[789] S. K. Sowe, I. Stamelos, and L. Angelis, "Understanding knowledge sharing activities in free/open source software projects: An empirical study," *Journal of Systems and Software*, vol. 81, no. 3, pp. 431–446, 2008. *cited on page 171*

[790] P. van der Spek, "Managing software evolution in embedded systems," Ph.D. dissertation, Vrije Universiteit Amsterdam, The Netherlands, 2010. *cited on page 330*

[791] P. van der Spek, S. Klusener, and P. van de Laar, "Towards recovering architectural concepts using Latent Semantic Indexing," in *European Conf. Software Maintenance and Reengineering*, 2008, pp. 253–257. *cited on page 161*

[792] J. M. Sprinkle, "Metamodel driven model migration," Ph.D. dissertation, Vanderbilt University, 2003. *2 citations on pages 52 and 62*

[793] J. M. Sprinkle and G. Karsai, "A domain-specific visual language for domain model evolution," *Journal of Visual Languages and Computing*, vol. 15, no. 3-4, pp. 291–307, 2004. *2 citations on pages 52 and 59*

[794] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Int'l Conf. Automated Software Engineering*, 2010, pp. 43–52. *cited on page 155*

[795] T. Stahl and M. Voelter, *Model-driven software development : technology, engineering, management*. John Wiley, 2006. *cited on page 269*

[796] C. Stoermer and L. O'Brien, "Map - mining architectures for product line evaluations," in *Working Conf. Software Architecture*, 2001, pp. 35–44. *cited on page 286*

[797] G. Stoneburner, A. Goguen, and A. Feringa, "Risk Management Guide for Information Technology Systems," National Institute of Standards and Technology, Tech. Rep. 800-30, Jul. 2002. *cited on page 7*

[798] J. Strauch and S. Schreier, "RESTify: From RPCs to RESTful HTTP design," in *Int'l Workshop on RESTful Design (WS-REST)*. ACM , Apr. 2012, pp. 11–18. *2 citations on pages 213 and 214*

[799] J. A. Street and R. G. Pettit, "The impact of UML 2.0 on existing UML 1.4 models," in *Model Driven Engineering Languages and Systems*, ser. Lect. Notes in Computer Science, vol. 3713. Springer, 2005, pp. 431–444. *cited on page 43*

[800] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Int'l Conf. Automated Software Engineering*, 2011, pp. 253–262. *2 citations on pages 192 and 196*

[801] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Int'l Conf. Software Engineering*, 2010, pp. 45–54. *cited on page 153*

[802] ——, "A discriminative model approach for accurate duplicate bug report retrieval," in *Int'l Conf. Software Engineering*, 2010, pp. 45–54. *cited on page 186*

[803] J. Sun, H. Zhang, and H. Wang, "Formal semantics and verification for feature modeling," in *Int'l Conf. Engineering of Complex Computer Systems*, 2005, pp. 303–312. *cited on page 292*

[804] G. Sunyé, D. Pollet, Y. L. Traon, and J.-M. Jézéquel, "Refactoring UML models," in *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, ser. Lect. Notes in Computer Science, vol. 2185. Springer, 2001, pp. 134–148. *cited on page 221*

[805] D. Surian, N. Liu, D. Lo, H. Tong, E.-P. Lim, and C. Faloutsos, "Recommending people in developers' collaboration network," in *Working Conf. Reverse Engineering*, 2011, pp. 379–388. *3 citations on pages 163, 165, and 189*

[806] D. Surian, Y. Tian, D. Lo, H. Cheng, and E.-P. Lim, "Predicting project outcome leveraging socio-technical network patterns," in *European Conf. Software Maintenance and Reengineering*, 2013. *3 citations on pages 163, 165, and 189*

[807] W. Suryn, P. Bourque, A. Abran, and C. Laporte, "Software product quality practices quality measurement and evaluation using TL9000 and ISO/IEC 9126," *Int'l Workshop on Software Technology and Engineering Practice*, pp. 156–162, 2002. *cited on page 77*

[808] R. Suvorov, M. Nagappan, A. E. Hassan, Y. Zou, and B. Adams, "An empirical study of build system migrations in practice: Case studies on kde and the linux kernel," in *Int'l Conf. Software Maintenance*, 2012, pp. 160–169. *cited on page 331*

[809] M. Svahnberg and J. Bosch, "Evolution in software product lines: two cases," *J. Software Maintenance and Evolution: Research and Practice*, vol. 11, no. 6, pp. 391–422, 1999. *2 citations on pages 271 and 278*

[810] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxonomy of variability realization techniques," *J. Software: Practice and Experience*, vol. 35, pp. 705–754, 2005. *cited on page 286*

[811] D. Svetinovic and M. Godfrey, "Software and biological evolution: Some common principles, mechanisms, and a definition," in *Int'l Workshop on Principles of Software Evolution*, 2005, http://plg.uwaterloo.ca/~migod/papers/2005/iwpse05.pdf. *2 citations on pages 298 and 313*

[812] E. B. Swanson, "The dimensions of maintenance," in *Int'l Conf. Software Engineering*, 1976, pp. 492–497. *cited on page 7*

[813] ——, "The Dimensions of Maintenance," in *Int'l Conf. Software Engineering*. IEEE, 1976, pp. 492–497. *cited on page 243*

[814] M. D. Syer, B. Adams, Y. Zou, and A. E. Hassan, "Exploring the development of micro-apps: A case study on the blackberry and android platforms," in *Working Conf. Source Code Analysis and Manipulation*, 2011, pp. 55–64. *cited on page 330*

[815] N. Synytskyy, J. R. Cordy, and T. Dean, "Resolution of static clones in dynamic Web pages," in *Int'l Workshop on Web Site Evolution*, Sep. 2003, pp. 49–56. *cited on page 224*

[816] C. Szymański and S. Schreier, "Case study: Extracting a resource model from an object-oriented legacy application," in *Int'l Workshop on RESTful Design (WS-REST)*. ACM , Apr. 2012, pp. 19–24. *cited on page 214*

[817] A. Taivalsaari and T. Mikkonen, "Objects in the cloud may be closer than they appear: Towards a taxonomy of web-based software," in *Int'l Symp. Web Systems Evolution*, sep 2011, pp. 59–64. *cited on page 215*

[818] H. Takagi, "Interactive evolutionary computation: Fusion of the capacities of EC optimization and human evaluation," *Proc. IEEE*, vol. 89, no. 9, pp. 1275–1296, 2001. *3 citations on pages 104, 118, and 135*

[819] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set-based automatic bug triaging," in *Int'l Conf. Software Engineering*, 2011, pp. 884–887. *cited on page 189*

[820] G. Tamura, "QoS-CARE: A reliable system for preserving QoS contracts through dynamic reconfiguration," Ph.D. dissertation, University of Lille 1 - Science and Technology, and Universidad de Los Andes, 2012. *4 citations on pages 237, 250, 251, and 258*

[821] G. Tamura, R. Casallas, A. Cleve, and L. Duchien, "QoS contract-aware reconfiguration of component architectures using E-Graphs," in *Int'l Workshop on Formal Aspects of Component Software (FACS)*, ser. Lect. Notes in Computer Science, vol. 6921. Springer, 2012, pp. 34–52. *3 citations on pages 250, 251, and 258*

[822] G. Tamura, N. M. Villegas, H. A. Müller, L. Duchien, and L. Seinturier, "Improving context-awareness in self-adaptation using the dynamico reference model," in *Int'l Symp. Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 2013, pp. 153–162. *3 citations on pages 231, 233, and 259*

[823] G. Tamura, N. M. Villegas, H. A. Müller, J. P. Sousa, B. Becker, M. Pezzè, G. Karsai, S. Mankovskii, W. Schäfer, L. Tahvildari, and K. Wong, *Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems*, ser. Lect. Notes in Computer Science. Springer, 2013, vol. 7475, pp. 108–132. *6 citations on pages 246, 254, 255, 256, 257, and 263*

[824] A. Tang, M. A. Babar, I. Gorton, and J. Han, "A survey of architecture design rationale," *J. Systems and Software*, vol. 79, no. 12, pp. 1792–1804, 2006. *cited on page 291*

[825] J. Tang, H. Li, Y. Cao, and Z. Tang, "Email data cleaning," in *Int'l Conf. Knowledge Discovery in Data Mining*, 2005, pp. 489–498. *cited on page 146*

[826] A. G. Tansley, "The use and abuse of vegetational concepts and terms," *Ecology*, vol. 16, no. 3, pp. 284–307, Jul. 1935. *cited on page 300*

[827] A. C. Telea, Ed., *Reverse Engineering: Recent Advances and Applications*. InTech, 2012. *cited on page 343*

[828] A. Terceiro, L. Rios, and C. Chavez, "An empirical study on the structural complexity introduced by core and peripheral developers in free software projects," in *Brazilian Symp. Software Engineering*, Oct. 2010, pp. 21 –29. *2 citations on pages 307 and 310*

[829] The DART Team, *Dart Programming Language Specification*, http://www.dartlang.org/docs/spec/latest/dart-language-specification.pdf, Jun. 2012, 0.42. *cited on page 183*

[830] The Open Group, *SOA Source Book*. The Open Group, 2009. *cited on page 212*

[831] L. G. Thomas, "An Analysis of Software Quality and Maintainability Metrics with an Application to a Longitudinal Study of the Linux Kernel," Ph.D. dissertation, Vanderbilt University, Nashville, TN, USA, 2008. *cited on page 75*

[832] S. W. Thomas, B. Adams, D. Blostein, and A. E. Hassan, "Studying software evolution using topic models," *Science of Computer Programming*, pp. 1–23, 2013. *2 citations on pages 152 and 155*

[833] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *J. Empirical Software Engineering*, pp. 1–31, 2012. *cited on page 155*

[834] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Validating the use of topic models for software evolution," in *Working Conf. Source Code Analysis and Manipulation*, 2010, pp. 55–64. *cited on page 152*

[835] ——, "Modeling the evolution of topics in source code histories," in *Int'l Conf. Mining Software Repositories*, 2011, pp. 173–182. *cited on page 152*

[836] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan, "The impact of classifier configuration and classifier combination on bug localization," pp. 1–16, 2012, submitted to *IEEE Trans. Software Engineering*. *2 citations on pages 155 and 159*

[837] T. Thüm, D. Batory, and C. Kästner, "Reasoning about edits to feature models," in *Int'l Conf. Software Engineering*, 2009, pp. 254–264. *cited on page 292*

[838] S. Thummalapenta and T. Xie, "Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web," in *Int'l Conf. Automated Software Engineering*, 2008, pp. 327–336. *cited on page 181*

[839] ——, "Parseweb: a programmer assistant for reusing open source code on the web." in *Int'l Conf. Automated Software Engineering*, 2007, pp. 204–213. *cited on page 181*

[840] A. Thums and J. Quante, "Reengineering embedded automotive software," in *ICSM*, 2012, pp. 493–502. *cited on page 330*

[841] F. Thung, D. Lo, and L. Jiang, "Detecting similar applications with collaborative tagging," in *Int'l Conf. Software Maintenance*, 2012, pp. 600–603. *3 citations on pages 163, 165, and 174*

[842] A. K. Thurimella and B. Brügge, "Evolution in product line requirements engineering: A rationale management approach," in *Int'l Conf. Req. Engineering*, 2007, pp. 254–257. *cited on page 291*

[843] ——, "Issue-based variability management," *Information and Software Technology*, vol. 54, no. 9, pp. 933–950, 2012. *cited on page 291*

[844] K. Tian, M. Revelle, and D. Poshyvanyk, "Using latent Dirichlet allocation for automatic categorization of software," in *Int'l Conf. Mining Software Repositories*, 2009, pp. 163–166. *cited on page 154*

[845] Y. Tian, P. Achananuparp, I. N. Lubis, D. Lo, and E.-P. Lim, "What does software engineering community microblog about?" in *Int'l Conf. Mining Software Repositories*, 2012, pp. 247–250. *3 citations on pages 163, 173, and 187*

[846] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *European Conf. Software Maintenance and Reengineering*, 2012, pp. 385–390. *cited on page 196*

[847] W. Tichy, "An interview with Prof. Andreas Zeller: Mining your way to software reliability," *Ubiquity*, vol. 2010, Apr. 2010. *cited on page 140*

[848] S. R. Tilley, D. Distante, and S. Huang, "Web site evolution via transaction reengineering," in *Int'l Symp. Web Systems Evolution*, 2004, pp. 31–40. *cited on page 224*

[849] S. R. Tilley, J. Gerdes Jr., T. Hamilton, S. Huang, H. A. Müller, D. B. Smith, and K. Wong, "On the business value and technical challenges of adopting Web services," *J. Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 1-2, pp. 31–50, Jan.–Apr. 2004. *cited on page 212*

[850] S. R. Tilley and S. Huang, "Evaluating the reverse engineering capabilities of Web tools for understanding site content and structure: A case study," in *Int'l Conf. Software Engineering*, May 2001, pp. 514–523. *2 citations on pages 207 and 210*

[851] P. Tonella and F. Ricca, "Web application slicing in presence of dynamic code generation," *J. Automated Software Engineering*, vol. 12, no. 2, pp. 259–288, Apr. 2005. *cited on page 224*

[852] P. Tonella, F. Ricca, E. Pianta, and C. Girardi, "Restructuring multilingual web sites," in *Int'l Conf. Software Maintenance*, Oct. 2003, pp. 290–299. *2 citations on pages 224 and 225*

[853] ——, "Using keyword extraction for web site clustering," in *Int'l Workshop on Web Site Evolution*, sep 2003, pp. 41–48. *cited on page 224*

[854] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web?" in *Int'l Conf. Software Engineering*, 2011. *cited on page 171*

[855] C. Treude and M. Storey, "How tagging helps bridge the gap between social and technical aspects in software development?" in *Int'l Conf. Software Engineering*, 2009. *cited on page 190*

[856] F. Trucchia and J. Romei, *Pro PHP Refactoring*. Apress, 2010. *cited on page 202*

[857] S. Trujillo, D. Batory, and O. Diaz, "Feature refactoring a multi-representation program into a product line," in *Int'l Conf. Generative Programming*, 2006, pp. 191–200. *cited on page 285*

[858] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Trans. Soft. Eng.*, vol. 35, pp. 347–367, 2009. *cited on page 128*

[859] T. T. Tun, T. Trew, M. Jackson, R. Laney, and B. A. Nuseibeh, "Specifying features of an evolving software system," *J. Software: Practice and Experience*, vol. 39, no. 11, pp. 973–1002, 2009. *2 citations on pages 16 and 22*

[860] T. T. Tun, Y. Yu, R. Laney, and B. A. Nuseibeh, "Recovering problem structures to support the evolution of software systems," The Open University, Milton Keynes, UK, Tech. Rep. 2008/08, Apr. 2008. *3 citations on pages 12, 16, and 22*

[861] J. Tupamäki and T. Mikkonen, "On the transition from the web to the cloud," in *Int'l Symp. Web Systems Evolution*, sep 2013. *2 citations on pages 215 and 330*

[862] M. M. Turcotte, M. S. C. Corrin, and M. T. J. Johnson, "Adaptive evolution in ecological communities," *PLoS Biology*, vol. 10, no. 5, 2012. *cited on page 311*

[863] https://twitter.com/. *cited on page 164*

[864] B. Ujhazi, R. Ferenc, D. Poshyvanyk, and T. Gyimothy, "New conceptual coupling and cohesion metrics for object-oriented systems," in *Working Conf. Source Code Analysis and Manipulation*, 2010, pp. 33–42. *cited on page 162*

[865] UWA Project Consortium, "Ubiquitous Web Applications," in *eBusiness and eWork Conf.*, 2002. *cited on page 219*

[866] P. Valderas and V. Pelechano, "A survey of requirements specification in model-driven development of web applications," *ACM Transaction on the Web*, vol. 5, no. 2, may 2011. *cited on page 218*

[867] S. Valverde, G. Theraulaz, J. Gautrais, V. Fourcassie, and R. Sole, "Self-organization patterns in wasp and open source communities," *Intelligent Systems, IEEE*, vol. 21, no. 2, pp. 36 – 40, march-april 2006. *cited on page 331*

[868] M. Van Antwerp and G. R. Madey, "The importance of social network structure in the open source software developer community," in *Hawaii Int'l Conf. System Sciences*. IEEE Computer Society, 2010, pp. 1–10. *2 citations on pages 307 and 308*

[869] C. van Koten and A. R. Gray, "An Application of Bayesian Network for Predicting Object-Oriented Software Maintainability," *Information and Software Technology*, vol. 48, no. 1, pp. 59–67, Jan. 2006. *2 citations on pages 68 and 76*

[870] A. van Lamsweerde, "Goal-oriented requirements engineering: A guided tour," in *Int'l Conf. Req. Engineering*, 2001, pp. 249–262. *2 citations on pages 23 and 24*

[871] ——, "Reasoning About Alternative Requirements Options," in *Conceptual Modeling: Foundations and Applications*. Springer, 2009, pp. 380–397. *2 citations on pages 18 and 22*

[872] A. van Lamsweerde and E. Letier, "Handling obstacles in goal-oriented requirements engineering," *ACM Trans. Software Engineering and Methodology*, vol. 26, pp. 978–1005, 2000. *2 citations on pages 18 and 22*

[873] R. van Ommering, "Software reuse in product populations," *IEEE Trans. Soft. Eng.*, vol. 31, no. 7, pp. 537 – 550, july 2005. *cited on page 271*

[874] R. Van Solingen and E. Berghout, *The Goal/Question/Metric Method: a Practical Guide for Quality Improvement of Software Development*. McGraw-Hill, 1999. *cited on page 92*

[875] B. Vanderose, "Supporting a Model-driven and Iterative Quality Assessment Methodology: The MoCQA Framework," Ph.D. dissertation, University of Namur, Namur, Belgium, 2012. *cited on page 86*

[876] B. Vanderose, N. Habra, F. Kamseu, and T. Mens, "A Feasibility Study of Quality Assessment During Software Maintenance," in *Int'l Workshop Software Quality and Maintainability*, 2012. *cited on page 86*

[877] B. Vanderose, F. Kamseu, and N. Habra, "Towards a Model-centric Quality Assessment," in *Int'l Workshop on Software Measurement*, 2010, pp. 21–34. *cited on page 86*

[878] H. Vandierendonck and T. Mens, "Averting the next software crisis," *IEEE Computer*, vol. 44, no. 4, pp. 88–90, 2011. *cited on page 330*

[879] ——, "Techniques and tools for parallelizing software," *IEEE Software*, vol. 29, no. 2, pp. 22–25, 2012. *cited on page 330*

[880] H. Väre, R. Ohtonen, and J. Oksanen, "Effects of reindeer grazing on understorey vegetation in dry pinus sylvestris forests," *Journal of Vegetation Science*, vol. 6, p. 523530, 1995. *2 citations on pages 319 and 320*

[881] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz, "Comparative analysis of evolving software systems using the gini coefficient," in *Int'l Conf. Software Maintenance*, 2009, pp. 179–188. *2 citations on pages 325 and 331*

[882] B. Vasilescu, A. Capiluppi, and A. Serebrenik, "Gender, representation and online participation: A quantitative study of StackOverflow," in *SocialInformatics*, 2012, pp. 332–338. *cited on page 331*

[883] ——, "Gender, representation and online participation: A quantitative study," *Interacting with Computers*, pp. xx–xx, 2013. *cited on page 331*

[884] B. Vasilescu, V. Filkov, and A. Serebrenik, "StackOverflow and GitHub: Associations between software development and crowdsourced knowledge," in *SocialCom/PASSAT*, 2013, pp. 188–195. *cited on page 331*

[885] B. Vasilescu, A. Serebrenik, P. Devanbu, and V. Filkov, "How social Q&A sites are changing knowledge sharing in open source software communities," in *Int'l Conf. Computer Supported Cooperative Work*, 2014, pp. xx–xx. *cited on page 331*

[886] B. Vasilescu, A. Serebrenik, M. Goeminne, and T. Mens, "On the variation and specialisation of workload: A case study of the Gnome ecosystem community," *J. Empirical Software Engineering*, pp. 1–54, 2013. *6 citations on pages 303, 306, 316, 317, 318, and 331*

[887] B. Vasilescu, A. Serebrenik, and T. Mens, "A historical dataset of software engineering conferences," in *Int'l Conf. Mining Software Repositories*, 2013, pp. 373–376. *cited on page 332*

[888] B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand, "You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics," in *Int'l Conf. Software Maintenance*, 2011, pp. 313–322. *cited on page 325*

[889] B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand, "The Babel of software development: Linguistic diversity in open source," in *International Conference on Social Informatics*, ser. Lect. Notes in Computer Science, vol. 8238. Springer, 2013, pp. 391–404. *cited on page 331*

[890] S. Vermolen, G. Wachsmuth, and E. Visser, "Reconstructing complex metamodel evolution," in *Int'l Conf. Software Language Engineering*, ser. Lect. Notes in Computer Science, vol. 6940. Springer, 2012, pp. 201–221. *cited on page 63*

[891] J. E. N. Veron, "Coral taxonomy and evolution," in *Coral Reefs: An Ecosystem in Transition*. Springer, 2011, pp. 37–45. *cited on page 311*

[892] M. Vierhauser, P. Grünbacher, W. Heider, G. Holl, and D. Lettner, "Applying a consistency checking framework for heterogeneous models and artifacts in industrial product lines," in *MoDELS*, 2012, pp. 531–545. *cited on page 293*

[893] M. Viljainen and M. Kauppinen, *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Edward Elgar, 2013, ch. Framing Management Practices for Keystones in Platform Ecosystems. *cited on page 302*

[894] N. M. Villegas, "Context Management and Self-Adaptivity for Situation-Aware Smart Software Systems," Ph.D. dissertation, University of Victoria, Canada, February 2013. *5 citations on pages 231, 251, 257, 259, and 261*

[895] N. M. Villegas and H. A. Müller, "Context-driven Adaptive Monitoring for Supporting SOA Governance," in *Int'l Workshop on a Research Agenda for Maintenance and Evolution of Service-Oriented Systems (MESOA)*. Carnegie Mellon University Software Engineering Institute, 2010, pp. 111–133. *cited on page 231*

[896] ——, *Managing Dynamic Context to Optimize Smart Interactions and Services*. Springer, 2010, pp. 289–318. *cited on page 256*

[897] N. M. Villegas, H. A. Müller, J. C. Muñoz, A. Lau, J. Ng, and C. Brealey, "A Dynamic Context Management Infrastructure for Supporting User-driven Web Integration in the Personal Web," in *Conf. Center for Advanced Studies on Collaborative Research*. ACM, 2011, pp. 200–214. *No citation*

[898] N. M. Villegas, H. A. Müller, and G. Tamura, "Optimizing Run-Time SOA Governance through Context-Driven SLAs and Dynamic Monitoring," in *International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA)*. IEEE, 2011, pp. 1–10. *3 citations on pages 231, 250, and 251*

[899] N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, and R. Casallas, "A Framework for Evaluating Quality-driven Self-Adaptive Software Systems," in *Int'l Symp. Soft-*

ware Engineering for Adaptive and Self-Managing Systems. ACM, 2011, pp. 80–89. *5 citations on pages 231, 237, 239, 251, and 254*

[900] N. M. Villegas, G. Tamura, H. A. Müller, L. Duchien, and R. Casallas, *DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems*, ser. Lect. Notes in Computer Science. Springer, 2013, vol. 7475, pp. 265–293. *5 citations on pages 231, 241, 248, 255, and 263*

[901] M. Voelter and E. Visser, "Product line engineering using domain-specific languages," in *Int. Software Product Line Conf.*, 2011, pp. 70–79. *cited on page 269*

[902] M. Voelter and I. Groher, "Product line implementation using aspect-oriented and model-driven software development," in *Int. Software Product Line Conf.*, 2007, pp. 233–242. *cited on page 293*

[903] G. von Krogh, S. Snaeth, and K. Lakhani, "Community, joining, and specialization in open source software innovation: a case study," *Research Policy*, vol. 32, no. 7, pp. 1217–1241, 2003. *cited on page 323*

[904] G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *European Conf. Object-Oriented Programming*, ser. Lect. Notes in Computer Science, vol. 4609, 2007, pp. 600–624. *5 citations on pages 38, 39, 52, 53, and 54*

[905] S. Wagner, K. Lochmann, L. Heinemann, M. Kläs, A. Trendowicz, R. Plösch, A. Seidl, A. Goeb, and J. Streit, "The Quamoco product quality modelling and assessment approach," in *Int'l Conf. Software Engineering*. IEEE Press, 2012, pp. 1133–1142. *4 citations on pages 66, 67, 69, and 79*

[906] S. Wang, D. Lo, and L. Jiang, "Code search via topic-enriched dependence graph matching," in *Working Conf. Reverse Engineering*, 2011, pp. 119–123. *cited on page 196*

[907] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Int'l Conf. Software Engineering*, 2008, pp. 461–470. *cited on page 153*

[908] M. P. Ward, "The FermaT maintenance environment tool demonstration," in *Working Conf. Source Code Analysis and Manipulation*. IEEE Computer Society, 2009, pp. 125–126. *cited on page 123*

[909] P. Warren, C. Boldyreff, and M. Munro, "The evolution of websites," in *Int'l Workshop Program Comprehension*, May 1999, pp. 178–185. *2 citations on pages 207 and 224*

[910] D. M. Weiss and C. T. R. Lai, *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999. *cited on page 266*

[911] M. Weiss, G. Moroiu, and P. Zhao, "Evolution of open source communities," in *Open Source Systems*, ser. IFIP International Federation for Information Processing, vol. 203, 2006, pp. 21–32. *2 citations on pages 306 and 323*

[912] K. D. Welker, P. W. Oman, and G. G. Atkinson, "Development and Application of an Automated Source Code Maintainability Index," *J. Software Maintenance and Evolution: Research and Practice*, vol. 9, no. 3, pp. 127–159, May 1997. *cited on page 76*

[913] K. Welsh and P. Sawyer, "Requirements Tracing to Support Change in Dynamically Adaptive Systems," *Int'l Conf. Req. Engineering*, pp. 59–73, Apr. 2009. *3 citations on pages 20, 21, and 22*

[914] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Int'l Conf. Program Comprehension*. IEEE Computer Society, 2004, pp. 194–203. *cited on page 115*

[915] M. Wermelinger and Y. Yu, "Analyzing the evolution of Eclipse plugins," in *Int'l Conf. Mining Software Repositories*. ACM Press, 2008, pp. 133–136. *cited on page 303*

[916] M. Wermelinger, Y. Yu, and A. Lozano, "Design principles in architectural evolution: a case study," in *Int'l Conf. Software Maintenance*, 2008. *cited on page 303*

[917] J. White, "Simplifying Autonomic Enterprise Java Bean Applications via Model-driven Development: a Case Study," *Software and Systems Modeling*, pp. 601–615, 2005. *2 citations on pages 250 and 253*

[918] L. J. White, T. Reichherzer, J. Coffey, N. Wilde, and S. Simmons, "Maintenance of service oriented architecture composite applications: static and dynamic sup-

port," *J. Software: Evolution and Process*, vol. 25, no. 1, pp. 97–109, Jan. 2013. *3 citations on pages 212, 213, and 224*

[919] T. White, *Hadoop: The Definitive Guide*. O'Reilly, 2012. *cited on page 330*

[920] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel, "RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems," in *Int'l Conf. Req. Engineering*, 2009, pp. 79–88. *4 citations on pages 17, 22, 23, and 25*

[921] ——, "RELAX: A Language to Address Uncertainty in Self-Adaptive Systems Requirement," *Requirements Engineering*, vol. 15, no. 2, pp. 177–196, 2010. *cited on page 254*

[922] K. E. Wiegers, *Software Requirements*. Microsoft Press, 2003. *cited on page 13*

[923] T. A. Wiggerts, "Using clustering algorithms in legacy systems remodularization," in *Working Conf. Reverse Engineering*. IEEE Computer Society, 1997, p. 33. *cited on page 113*

[924] A. Wijs and L. Engelen, "Efficient property preservation checking of model refinements," in *Int'l Conf. Tools and Algorithms for Construction and Analysis of Systems*, ser. Lect. Notes in Computer Science. Springer, 2013, vol. 7795, pp. 565–579. *cited on page 330*

[925] C. Wilkinson, "Status of the coral reefs of the world: 2008," Global Coral Reef Monitoring Network and Reef and Rainforest Research Centre, Tech. Rep., 2008. *cited on page 301*

[926] R. J. Williams and N. D. Martinez, "Simple rules yield complex food webs," *Nature*, vol. 404, pp. 180–183, 2000. *cited on page 299*

[927] A. J. Willis, "The ecosystem: an evolving concept viewed historically," *Functional Ecology*, vol. 11, pp. 268–271, 1997. *cited on page 299*

[928] A. Wingkvist, M. Ericsson, and W. Löwe, "Making Sense of Technical Information Quality: A Software-based Approach," *J. Software Technology*, vol. 14, no. 3, pp. 12–18, 2011. *cited on page 86*

[929] World Wide Web Consortium, *HTML 4.01 Specification*, 1999. *cited on page 34*

[930] ——, *Scalable Vector Graphics (SVG) 1.1*, 2nd ed., 2011. *cited on page 34*

[931] C. Wu, E. Chang, and A. Aitken, "An empirical approach for semantic web services discovery," in *Australian Conf. Software Engineering*, 2008, pp. 412–421. *cited on page 155*

[932] J. Xu, Y. Gao, S. Christley, and G. Madey, "A topological analysis of the open source software development community," in *Hawaii Int'l Conf. System Sciences*, 2005. *cited on page 173*

[933] S. Xu and T. Dean, "Modernizing JavaServer pages by transformation," in *Int'l Symp. Web Systems Evolution*, sep 2005, pp. 111–118. *2 citations on pages 211 and 224*

[934] Q. Xuan, M. Gharehyazie, P. T. Devanbu, and V. Filkov, "Measuring the effect of social communications on individual working rhythms: A case study of open source software," in *SocialInformatics*, 2012, pp. 78–85. *cited on page 331*

[935] H. Yang, Ed., *Software Evolution with UML and XML*. Idea Group Publishing, 2005. *cited on page 344*

[936] S. Yau, J. Collofello, and T. MacGregor, "Ripple effect analysis of software maintenance," in *Int'l Computer Software and Applications Conf.* IEEE, 1978, pp. 60–65. *3 citations on pages 232, 246, and 256*

[937] Y. Ye, K. Nakakoji, Y. Yamamoto, and K. Kishida, "The co-evolution of systems and communities in free and open source software development," in *Free/Open Source Software Development*. IDEA Group Publishing, 2005, pp. 59–82. *cited on page 315*

[938] E. S. Yu, "Towards modelling and reasoning support for early-phase requirements engineering," in *Int'l Conf. Req. Engineering*, 1997, pp. 226–235. *cited on page 20*

[939] L. Yu, "Understanding component co-evolution with a study on Linux," *J. Empirical Software Engineering*, vol. 12, no. 2, pp. 123–141, Apr. 2006. *cited on page 315*

[940] L. Yu and S. Ramaswamy, "Software and biological evolvability: A comparison using key properties," in *Int'l IEEE Workshop on Software Evolvability*, 2006, pp. 82– 88. *2 citations on pages 298 and 313*

[941] ——, "Mining CVS repositories to understand open-source project developer roles," in *Mining Software Repositories*. IEEE Computer Society, 2007. *cited on page 307*

[942] Y. Yu, T. T. Tun, A. Tedeschi, V. N. L. Franqueira, and B. A. Nuseibeh, "OpenArgue: Supporting argumentation to evolve secure software systems," in *Int'l Conf. Req. Engineering*. IEEE Computer Society, 2011. *2 citations on pages 16 and 22*

[943] A. Zaidman, B. Rompaey, A. Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *J. Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011. *2 citations on pages 315 and 329*

[944] N. C. Zakas, "The evolution of web development for mobile devices," *ACM Queue*, vol. 11, no. 2, Feb. 2013. *cited on page 222*

[945] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering," *ACM Trans. Software Engineering and Methodology*, vol. 6, pp. 1–30, 1997. *cited on page 5*

[946] H. Zawawy, K. Kontogiannis, and J. Mylopoulos, "Log filtering and interpretation for root cause analysis," in *Int'l Conf. Software Maintenance*, 2010, pp. 1–5. *cited on page 155*

[947] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *Joint European Software Engineering Conf. and ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*. Springer, 1999, pp. 253–267. *cited on page 125*

[948] C. X. Zhai, "Statistical language models for information retrieval," *Synthesis Lectures on Human Language Technologies*, vol. 1, no. 1, pp. 1–141, 2008. *cited on page 146*

[949] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *Int'l Conf. Software Engineering*, 2012, pp. 14–24. *cited on page 196*

[950] M. Zhou and A. Mockus, "Does the initial environment impact the future of developers?" in *Int'l Conf. Software Engineering*. ACM, 2011, pp. 271–280. *cited on page 323*

[951] ——, "What make long term contributors: willingness and opportunity in OSS community," in *Int'l Conf. Software Engineering*. IEEE Press, 2012, pp. 518–528. *cited on page 323*

[952] Y. Zhou and H. Leung, "Predicting Object-oriented Software Maintainability Using Multivariate Adaptive Regression Splines," *J. Systems and Software*, vol. 80, no. 8, pp. 1349–1361, Aug. 2007. *2 citations on pages 68 and 76*

[953] Y. Zhou and B. Xu, "Predicting the Maintainability of Open Source Software Using Design Metrics," *Wuhan University Journal of Natural Sciences*, vol. 13, no. 1, pp. 14–20, Feb. 2009. *cited on page 76*

[954] T. Zimmermann, N. Nagappan, and A. Zeller, *Software Evolution*. Springer, 2008, ch. Predicting Bugs from History, pp. 69–88. *cited on page x*

[955] D. Zowghi and V. Gervasi, "On the interplay between consistency, completeness and correctness in requirements evolution," *Information and Software Technology*, vol. 45, no. 14, pp. 993–1009, 2003. *cited on page 10*

[956] D. Zowghi and R. Offen, "A Logical Framework for Modeling and Reasoning about the Evolution of Requirements," in *Int'l Conf. Req. Engineering*, 1997, pp. 247–257. *2 citations on pages 18 and 22*

Index

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4, © Springer-Verlag Berlin Heidelberg 2014