Walid Maalej
Anil Kumar Thurimella *Editors*

# Managing Requirements Knowledge

Springer

# Managing Requirements Knowledge

Walid Maalej • Anil Kumar Thurimella
Editors

# Managing Requirements Knowledge

Springer

*Editors*
Walid Maalej
University of Hamburg
Department of Informatics / MOBIS
Hamburg
Germany

Anil Kumar Thurimella
Harman Becker Automotive Systems GmbH
Munich
Germany

ACM Computing Classification (1998): D.2, K.6, I.2

ISBN 978-3-642-34418-3 ISBN 978-3-642-34419-0 (eBook)
DOI 10.1007/978-3-642-34419-0
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013940938

Printed on acid-free paper

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

# Preface

## The Story

This book synthesizes the work of the managing requirements knowledge (MARK) community during the last 5 years. The first idea to organize a workshop on this topic came to our minds in winter 2007. We were both working on our Ph.D. projects at the Technische Universität München (TUM) under the supervision of Bernd Brügge. Anil was focusing on software product lines, while Walid was looking at the application of ontologies and machine learning to collaborative software engineering, in particular during bug fixing and API reuse. Our fields of interest seemed divergent at first glance. However, after a couple of discussions – also with colleagues from industry – we found that some of the problems we were trying to address are very similar. Valuable experiences and knowledge gained in the course of software projects, in particular during the work with requirements, remain tacit in the mind of people. The same problems in understanding and implementing requirements occur again and again.

We were convinced about the need for a new perspective on requirements – considering them as a knowledge asset in software organizations – in addition to the engineering and lifecycle perspectives. We were convinced about the huge potentials of recent trends such as ontologies, wikis, Web 2.0, recommendation systems, and data mining, to the requirements engineering community.

In the last years, the MARK workshop successfully took place in Barcelona, Atlanta, Sydney, and Trento. It has been one of the most successful workshops at the IEEE International Conference on Requirements Engineering that is based on submission and registration statistics, as well as the feedback of the participants. The achievements are remarkable. Novel approaches such as "recommending features and stakeholders by analyzing requirements repositories" or "using semantic wikis to represent and reason about requirements" have found their way to main conferences and journal in the field. Some of the tools are already being used in practice.

With this book, we hope to present a baseline for the community discussion, enabling more people to join and contribute. We also hope to bring more research questions and initiate even more discussions. Managing requirements knowledge is a new evolving field. Its requirements and its knowledge are evolving as well. We invite you to contribute. Enjoy reading!

## The Structure

In addition to the introduction and conclusion chapters, which motivate the field, introduce the foundations and definitions, overview the approaches proposed so far, and discuss the road ahead, the rest of this book is structured into five parts.

*Part I. Identifying Requirements Knowledge* shows the importance of identifying and externalizing tacit knowledge about requirements such as rationale and presuppositions. It covers theoretical frameworks to model tacit knowledge, empirical studies to investigate mining requirements knowledge from project artifacts, as well as pragmatic and practical discussion on what is requirements knowledge in practice and how to manage without introducing additional overhead.

*Part II. Representing Requirements Knowledge for Reuse* introduces techniques such as patterns and ontologies to represent requirements knowledge for both humans and machine, enabling an efficient knowledge access for various stakeholders. We focus on techniques which support reuse of knowledge within and between software projects.

*Part III. Sharing Requirements Knowledge* is about people, i.e., requirements stakeholders, and the exchange of knowledge among them. This part discusses knowledge-sharing tools such as social media and Web 2.0 for requirements as well as methodologies such as agile requirements and question asking.

*Part IV. Reasoning About Requirements* discusses how to reason about the interdependencies of requirements and their knowledge. The goal is to check consistency and derive new knowledge. Also the integration of requirements knowledge into other software engineering knowledge is discussed.

Finally, *Part V Intelligent Tool Support* focuses on the tool perspective, and on how to apply novel techniques such as recommendation systems, experience-based tools, as well as integrated development environments to deal with the information overload, and the huge amount of knowledge related to requirements in large, complex, distributed projects.

## The Audience

There are no special prerequisites to read this book. We tried our best to address the needs of the following target groups:

- Researchers from the area of knowledge management with interests on requirements engineering
- Researchers from the area of requirements engineering with interests on knowledge management
- Industrial practitioners involved in requirements engineering and outsourcing projects
- Lecturers, students, and practitioners interested in the state of the art of requirements engineering

# Acknowledgments

This work would not have been possible without the great support of many individuals. First, we would like to greatly thank the MARK community, including all authors, attendants, reviewers, and the other co-organizers for all the contributions and the great constructive discussions. In particular, we would like to thank the authors of the book chapters, whose names are listed in the author list at the end of this book. Thanks to these authors for writing the chapters, reviewing other chapters, their commitments, and all the hard work and iterations to bring our book vision into reality.

We would also like to thank our external reviewers, namely, Barbara Paech, Rick Rabiser, Yang Li, Pete Sawyer, Smita S Ghaisas, Dennis Pagano, and Zardosht Hodaie, for their detailed and constructive feedback.

Heidi Oskarsson helped with formatting matters. Ralf Gerstner and Viktoria Meyer of Springer gave invaluable advice for management and publishing issues. Finally, we can hardly thank enough our families for their patience when we were spending our nights editing this book and for their constant love.

November 2012

Walid Maalej
Anil Kumar Thurimella

# Contents

# List of Contributors

**Nirav Ajmeri** Tata Consultancy Services, Mumbai, India

**K.K. Biswas** IIT Delhi, Delhi, India

**Juan M. Carrillo De Gea** Universidad de Murcia, Murcia, Spain

**Daniela Damian** University of Victoria, Victoria, BC, Canada

**Olawande Daramola** Covenant University Nigeria, Ota, Nigeria

**Dov Dori** Technion, Haifa, Israel

**Christof Ebert** Vector Consulting Services, Stuttgart, Germany

**Abdelrahman Elfaki** Management and Science University, Shah Alam, Malaysia

**Alexander Felfernig** Graz University of Technology, Graz, Austria

**Jose L. Fernández Alemán** Universidad de Murcia, Murcia, Spain

**Anthony Finkelstein** University College London, London, UK

**Xavier Franch** Universitat Politècnica de Catalunya, Barcelona, Spain

**Ricardo Gacitua** Lancaster University, Lancaster, UK

**Rosalva Gallardo-Valencia** University of California, Irvine, CA, USA

**Vincenzo Gervasi** University of Pisa, Pisa, Italy

**Smita S. Ghaisas** Tata Consultancy Services, Mumbai, India

**Harald Grabner** Graz University of Technology, Graz, Austria

**Cindy Guerlain** Centre de Recherche Public Henri Tudor, Kirchberg, Luxembourg

**Fuyuki Ishikawa** National Institute of Informatics, Chiyoda-ku, Japan

**Michael Jastram** Formal Mind/University of Düsseldorf, Düsseldorf, Germany

**Leonid Kof** Technische Universität München, Munich, Germany

**Eric Knauss** Leibniz Universität Hannover, Hannover, Germany

**Milton Lavin** Jet Propulsion Lab/Caltech, Pasadena, CA, USA

**Soo Ling Lim** University College London, London, UK

**Robyn Lutz** Iowa State University, Ames, IA, USA

**James Lux** Jet Propulsion Lab/Caltech, Pasadena, CA, USA

**Lin Ma** The Open University, Buckinghamshire, UK

**Walid Maalej** University of Hamburg, Hamburg, Germany

**Sebastian Meyer** Leibniz Universität Hannover, Hannover, Germany

**Joaquín Nicolás** Universidad de Murcia, Murcia, Spain

**Gerald Ninaus** Graz University of Technology, Graz, Austria

**Bashar Nuseibeh** The Open University, Buckinghamshire, UK; Lero, Limerick, Ireland

**Inah Omoronyia** University of Glasgow, Glasgow, Scotland

**Dennis Pagano** Technische Universität München, Munich, Germany

**Cristina Palomares** Universitat Politècnica de Catalunya, Barcelona, Spain

**Kenneth Peters** Jet Propulsion Lab/Caltech, Pasadena, CA, USA

**Paul Piwek** The Open University, Buckinghamshire, UK

**Carme Quer** Universitat Politècnica de Catalunya, Barcelona, Spain

**Florian Reinfrank** Graz University of Technology, Graz, Austria

**Samuel Renault** Centre de Recherche Public Henri Tudor, Kirchberg, Luxembourg

**Anne de Roeck** The Open University, Buckinghamshire, UK

**Nicolas Rouquette** Jet Propulsion Lab/Caltech, Pasadena, CA, USA

**Mark Rouncefield** Lancaster University, Lancaster, UK

**Pete Sawyer** Lancaster University, Lancaster, UK

**Richa Sharma** IIT Delhi, Delhi, India

**Susan Elliott Sim** University of California, Irvine, CA, USA

**Guttorm Sindre** NTNU, Trondheim, Norway

**Avi Soffer** ORT-Braude College of Engineering, Karmiel, Israel

**Tor Stålhane** NTNU, Trondheim, Norway

**Anil Kumar Thurimella** Harman Becker Automotive Systems GmbH, Munich, Germany

**Ambrosio Toval** Universidad de Murcia, Murcia, Spain

**Aurora Vizcaíno** Universidad de Castilla – La Mancha, Cuenca, Spain

**Leopold Weninger** wsop, Vienna, Austria

**Alistair Willis** The Open University, Buckinghamshire, UK

**Hui Yang** The Open University, Buckinghamshire, UK

# Chapter 1
# An Introduction to Requirements Knowledge

**W. Maalej and A.K. Thurimella**

**Abstract** Requirements represent a verbalisation of decision alternatives on the functionality and quality of a system. Engineering, planning, and implementing requirements are collaborative, problem-solving activities, where stakeholders consume *and* produce considerable amounts of knowledge. Managing requirements knowledge is about efficiently identifying, accessing, externalising, and sharing this knowledge by and to all stakeholders, including analysts, developers, and users. This chapter introduces five foundations of managing requirements knowledge, which are discussed in the book parts. First, *identifying* requirements knowledge aims at externalising tacit knowledge such as rationale or presuppositions. Second, *representing* requirements knowledge targets an efficient information access and artefact reuse within and between projects. Third, *sharing* requirements knowledge improves stakeholders' collaboration and ensures that their experiences do not get lost. Fourth, *reasoning* about requirements and their interdependencies aims at detect inconsistencies and deriving new knowledge. Finally, *intelligent tool support* reduces the overhead to manage requirements knowledge.

## 1.1 What Is Requirements Engineering?

We use the term requirements in the context of systems engineering, which is the discipline concerned by designing, developing, deploying, and maintaining systems. A system is an organised set of communicating parts designed for a specific purpose

W. Maalej (✉)
University of Hamburg, Department of Informatics/MOBIS, Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
e-mail: maalej@informatik.uni-hamburg.de

A.K. Thurimella
Harman Becker Automotive Systems GmbH, Moosacher Str. 48, 80809, Munich, Germany
e-mail: anil.thurimella@gmail.com

[1]. For example, a mobile phone composed of a display, a battery, an antenna, a microphone, a speaker, and a processor is designed to enable users to make calls, while they are on the go. In this book we target software engineering, which is a subdiscipline that focuses on engineering software-intensive systems. Bruegge and Dutoit characterise software engineering as a modelling and problem-solving activity, which is knowledge intensive and rationale driven [1].

Participants in a software engineering project create formal and informal models to (a) reason about software systems, (b) communicate about them, and (c) document their properties. These might be functional models, object models, dynamic models, or feature models. A functional model, for example, a use case diagram, describes the *functionality* of a system, for instance, dialling a number or terminating a call. An object model, for example, a class diagram, describes the *structure* of a system in terms of components, objects, attributes, and operations. A dynamic model, for example, a sequence diagram, represents the interactive behaviour of the system or of its parts, for instance, how the user interacts with the keyboard to dial a number or how the antenna interacts with the processor to send signals. The high-level description of a system is often communicated to clients and end users in terms of features, which are prominent or distinctive visible characteristics or qualities of a system [2]. For example, an mp3 player and a multitouch interface are two features of a modern mobile phone.

The term *requirement* is similar to the term feature but has a larger scope and a more technical focus. The IEEE standard glossary of software engineering terminology defines a requirement [3] as "a statement of what the system must do, how it must behave, the properties it must exhibit, the qualities it must possess, and the constraints that the system and its development must satisfy [4]".

*Requirements engineering* (RE) is the branch of systems engineering concerned with the *desired properties and constraints* of software-intensive systems, the goals to be achieved in the software's environment, and assumptions about the environment [5]. In another frequently cited definition, Sommerville and Sawyer state that requirements engineering is the activity that emphasises the utilisation of *systematic* and *repeatable* techniques that ensure the completeness, consistency, and relevance of requirements [6]. We call this the engineering view of requirements. Nuseibeh and Easterbrook [7] define requirements engineering as the process of *discovering* the purpose of the system being developed, by identifying stakeholders and their needs and *documenting* these in a form that is *amenable* to analysis, communication, and subsequent implementation. We call this the life cycle view of requirements. Finally, Aurum and Wohlin consider requirements as *verbalisation of decision alternatives* regarding the functionality and quality of a system [8]. Requirements engineering can then be considered as the complex task of dealing with, making, and documenting these decisions. We call this the decision or the knowledge view of requirements.

Requirements engineering is considered as one of the most critical phases in software projects [9]. Poorly implemented requirements engineering is a major risk for projects failure [10]. Today's software projects still have a high probability to be cancelled or to significantly exceed available resources [11]. For example, Leffingwell [12] found that 40 % of the total project costs are associated with rework triggered by low-quality requirements.

Requirements engineering rarely receives more than 2–4 % of the overall project effort [13], even if more effort in getting the requirements right results in significantly higher project success rates. A recent Gartner report [14] states that requirements defects are the third most significant source of product defects after coding and design but are the first source of delivered defects (i.e. defects delivered to the user). Fixing a defect in production is approximately 200 times more expensive for a software project than fixing it during requirements engineering, Gartner says. The damage and costs caused to the customers and their users when delivering a defect are excluded from this calculation and cannot be truly quantified since it depends on the domain and the business. Improving the quality of requirements and the efficiency requirements engineering can reduce the overall cost of software, improve its quality, and dramatically shorten the time to market.

### *1.1.1 Requirements Engineering Activities*

Requirements engineering covers several activities, including requirements elicitation, analysis, specification, verification, and management [15]:

- Requirements elicitation is the process of discovering, reviewing, documenting, and understanding the user's needs and constraints for a system.
- The process of refining the user's needs and constraints is called requirements analysis.
- Requirements specification is the process of documenting the user's needs and constraints clearly and precisely.
- Ensuring that the system requirements are complete, correct, consistent, and clear is done as a part of requirements verification.
- Scheduling, negotiating, coordinating, and documenting the requirements engineering activities are called requirements management.

Requirements engineering overlaps with planning. A project plan, including work packages, releases, iterations, or milestones, is created by analysing requirements. Later, the plan is detailed further. Tasks and action items are created and assigned to the project participants. The delivery of requirements is committed to the customer based on the project plan.

Requirements evolve over time. Change requests are often used to refer to changes on requirements. Change requests might originate from customers after the initial requirements elicitation phase, as well as from other sources such as regulators, development, testing, or marketing. Change requests are decided by analysing corresponding change impacts on the system. A well-implemented requirements elicitation often reduces the number of change requests in a project. Similarly, a large number of change requests are an indicator for poor requirements engineering.

Requirements engineering can also be performed for a product family (or a family of related systems) for systematically reusing artefacts and assets that are shared across multiple systems such as requirements, decisions, activities, and processes. Such a product family (e.g. a particular generation of mobile phones such as data phones or smart phones) is called a software product line.

Clements and Northrop define a *software product line* (SPL) as a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [4]. Product line engineering uses variability as an abstraction to deal with customisation and reuse [2, 16].

SPLs offer several benefits such as improved reuse, quicker time to market, and decreased defect rates, in particular, for manufacturing and mass customisation companies. These benefits have been reported in the form of experiences and best practices, for example, in the product line hall of fame [17]. On the other hand, SPLs need large upfront investments, in particular, when systems have been in place over decades or in IT service industry where the customers' needs, infrastructures, and constraints drive the projects.

### *1.1.2 Requirements Artefacts*

Requirements engineering involves various artefacts and document types. Business and marketing stakeholders, for instance, typically conduct negotiations with customers in the early project phases based on customer requirements and feature catalogues. Detailed requirements can be captured in natural language text, mathematical models, visual models such as UML, or using multimedia. A mathematical model represents the requirements formally, for example, using the Z specification language [18] and is used if, for example, the correctness of the system behaviour is critical. For instance, errors while making an emergency call can cost human lives. The behaviour of the system should be formally specified and validated in such cases – in particular if regulators mandate this. UML use case diagrams, sequence diagrams, and activity diagrams can be used to model requirements semiformally. For instance, these diagrams might show, respectively, use cases provided by a phone; interactions between the phone, its user, and its environment (e.g. an operator network or a Bluetooth device); and the overall process flow from dialling to getting the invoice.

More recently, the requirements engineering community started exploring and using multimedia requirements [19], including images (e.g. pictures of a typical hand positions during a call), drawings (e.g. mock-ups of the screen), or videos (e.g. showing a typical scenario of a mobile user in the metro). In industry, requirements are often documented in unstructured or semi-structured natural language documents [20], called requirements specification documents.

Requirements are classified into functional and non-functional requirements. A *functional requirement* expresses functionality (or a functional property) of a system and is specified based on inputs, outputs, and a process or a functional behaviour. An example for a mobile phone functional requirement is "the user shall be able to send and receive SMS to other users of mobile phones". A functional requirement can be decomposed into several sub-requirements.

A *non-functional requirement* (NFR) also called a quality requirement should express measurable properties of the system [79]. For example, “when switched on, the user should be able to dial a number within 3 s”, which constrains the performance of the system. Other categories of NFRs include usability, availability, safety, security, privacy, and maintainability.

Requirements are documented and handled based on their types. Functional requirements are often described in use case documents or mock-ups, while NFRs are captured using text documents, in change requests, or as comments in source code. Different NFRs are handled differently. For instance, safety is often handled by specifying additional requirements to address hazards and misuse cases identified by safety engineers.

### *1.1.3 Stakeholders, Collaboration, and Decisions*

Requirements engineering involves people with different backgrounds including business, marketing, law, project management, design, development, and testing. These people are called *stakeholders*. They perform relevant tasks in a requirements engineering process depending on their backgrounds and collaborate to capture requirements and make decisions about them and their priorities.

Making decisions is about choosing between alternative solutions for an issue [21]. An issue can be, for example, which authentication feature should the phone provide? The alternatives are a username and password, a pin code, a lock screen pattern, or face recognition. Effective decision-making occurs when stakeholders select the best choice based on the knowledge available at the time [22].

Rationale is the *reasoning* behind decisions, that is, the answer to the why question. Rationale can be described in natural language text or can be structured based on alternatives, reasons, and justifications. Kunz and Rittel introduced a rhetorical model to mange rationale called IBIS (issue-based information systems) [23]. IBIS uses abstractions like issue, option, argument, and resolution. An option is a potential solution for an issue. An assessment is a stakeholder argument that supports or hinders an option. A resolution contains a set of options that solve an issue. IBIS allows the expression of interdependencies between issues, which can lead to complex issue networks. QOC [24] extends IBIS with criteria, for example, the cost of implementing the authentication option, or its usability. Dutoit [25] introduced QOC to requirements engineering by modelling QOC criteria as goals or non-functional requirements, supporting decision-making in product management meetings.

Research has shown that rationale knowledge is useful in many ways. For example, it may be helpful to assess changes [26]. Alternative solutions and arguments documented in rationale may be used to forecast potential changes. Furthermore, rationale may be reused when similar issues are raised or when changes in previous decisions occur. However, rationale is barely managed systematically and often remains in the mind of people. When people leave the organisation, this knowledge gets lost [27]. In practice, rationale is found sporadically across documents, emails, or discussion threads [27].

As software projects are getting more distributed and the development cycles are getting shorter, stakeholders are often located in different places [28], sometimes even without having the resources to get to know each other in a face to face meeting. For example, the clients and the users might be in an Asian country with special regulations and infrastructures for communication systems and where people present certain habits and preferences in how they use their phones. The requirements engineers and the developers might be at the development site in a different country, speaking a different language, knowing different laws, and having different habits and preferences. Distributed development settings introduce additional collaboration challenges for stakeholders to understand each other, reach a common understanding of requirements, and make effective decisions.

## 1.2 What Is Managing Requirements Knowledge?

We introduce the terms “knowledge”, “knowledge management”, and “requirements knowledge” and motivate the need for managing requirements knowledge.

### *1.2.1 What Is Knowledge?*

*Knowledge* is a popular term used in our everyday language as well as in several disciplines such as philosophy, management science, and computer science. According to the Oxford Dictionary, the term knowledge refers to facts, information, and skills acquired by a person through *experience or education*. It also refers to *the awareness or familiarity* gained by experience of a fact or situation.

In philosophy the study of knowledge is called epistemology. Plato gave one of the oldest and most famous definitions for knowledge as *justified true belief* [29]. However, there exists a large and still active debate between philosophers about Plato’s definition and about the term knowledge and associated concepts [30].

In computer and management science, the term knowledge is often mixed with data and information. According to Theirauf [31]: “*data* represents the unstructured facts and figures, which has the least impact for the typical manager. [. . .] At the next level *information* is structured data that is useful to the manager in analysing and resolving critical problems. [. . .] At the next level there is *knowledge*, which is obtained from experts based upon actual experience. While information is data about data, knowledge is basically information about information”.

In the late 1990s and beginning of the 2000s, a new field called *knowledge management* emerged and has become popular amongst people from academia and industry. Wikipedia defines knowledge management as *a range of practices used by organisations to identify, create, represent, and distribute knowledge for reuse, awareness and learning across the organisation* [32]. In this book we adopt Hansen’s definition [33]:

**Def. 1.** Knowledge management is the dual process of accessing (searching for and identifying) and sharing (capturing and transferring) knowledge across organisational subunits.

Knowledge transfer as an aspect of knowledge management has always existed in one form or another, for example, through on-the-job peer discussions, formal apprenticeship, corporate libraries, professional training, and mentoring programmes. However, since the late twentieth century, additional theories and technologies have been applied to this task, such as knowledge bases, expert systems, and knowledge repositories. Knowledge management initiatives attempt to manage the process of creation or identification, accumulation, and application of knowledge or intellectual capital (i.e. the intangible assets of a company which contribute to its valuation) across an organisation.

### *1.2.2 What Is Requirements Knowledge?*

Requirements knowledge can be any kind of knowledge, which emerges during requirements engineering or more generally while working with requirements:

**Def. 2.** Requirements knowledge consists of the implicit or explicit information that is created or needed while engineering, managing, implementing, or using requirements, and that is useful for answering requirements-related questions in any phase of a software project.

Requirements knowledge is diverse, because requirements affect different engineering activities (including design and implementation) and because requirements engineering involves different stakeholders. We distinguish between five types of requirements knowledge:

- *Domain knowledge* refers to common knowledge in a particular area or a specialised discipline. This is usually the domain, for which a system should be developed. Domain knowledge includes a vocabulary, standards used in the domain (e.g. telecommunication or banking standards), and business rules (i.e. domain constraints, standards, and regulations to be satisfied when designing or using the system).
- *Engineering knowledge* includes requirements "content", such as the requirements specifications, dependencies between requirements, as well as other artefacts needed to understand and implement the requirements such as models, test cases, or system architecture. Also informal notes and personal comments typically annotating artefacts such as models, requirements, or plans might include useful engineering information.
- *Management knowledge* includes quality measures, templates, and properties of requirements such as status, priority, and stakeholder preferences. Moreover, emerging requirements-related issues, decisions, and action items are part of this

knowledge. For example, during a requirement review, open issues, decisions, and action items on requirements might be identified, discussed, and planned.

- *Collaboration knowledge* includes information about people, their interactions, discussions, argumentation chains, and presuppositions. Discussions include information exchanged or shared between different stakeholders on various problems related to requirements. Discussion might also include requirements rationale or the reasoning behind the requirements, a crucial piece of knowledge to understand and implement requirements especially when people leave the projects. Finally, presuppositions are assumptions for realising a requirement. The lack of common understanding of presuppositions often leads to misunderstanding of requirements [34].
- *How-to Knowledge* includes information on tools, methods, and processes to be used for a particular situation while engineering and managing requirements. Organisation or vendor guidelines include information on how to perform requirements engineering activities or how to use a tool.

Requirements knowledge does necessarily exist in the form of information or data. A considerable amount of requirements knowledge such as rationale behind decisions or domain assumptions is tacit and remains in the heads of people. For example, aspects of the system that seems trivial such as the performance, usability, or localisation of a mobile phone might not be captured and discussed explicitly during the requirements engineering work.

**Def. 3.** Managing requirements knowledge is about efficiently identifying, accessing, externalising, and sharing all types of requirements knowledge by and to all stakeholders, including analysts, developers, and users.

Managing requirements knowledge aims at externalising tacit knowledge and solving requirements-related issues by using the dual process of accessing and sharing knowledge (see Def. 1). For example, during requirement elicitation a requirements analyst might spend weeks to access (i.e. searching and identifying) privacy regulations and laws about mobile telephony. The analyst might then share this knowledge to other stakeholders such as architects, managers, or users by *capturing* and *communicating* summaries and links to the regulations that are relevant for the envisioned system.

### *1.2.3 Why Managing Requirements Knowledge?*

Requirements engineering, management, and implementation are complex, knowledge-intensive activities. Working with requirements involves many stakeholders from different backgrounds working in different phases and activities. To make, document, refine, or understand the requirements decisions, stakeholders need diverse information from diverse sources. For example, requirement analysts need information on the domain for defining correct and complete requirements. Change requesters need information on the processes followed for tracking the status of their requests. Architects need information on the technologies used in order to assess the requirements feasibility [35].

Moreover, software engineering is a field where constant changes take place, making the work of stakeholders extremely dynamic. New problems are discovered (e.g. users do not want to use an extra pen to enter information to the mobile phones), new solutions are designed (e.g. new multitouch technology which enable tracking multiple fingers at same time), and experiences are made (e.g. users prefer simple user interfaces or that the operator infrastructure API only allows for a certain error tolerance) on a daily basis. The requirements knowledge in software projects is diverse and its proportions immense and continuously expanding. Thus, a systematic way of managing and treating the knowledge and its owners as valuable assets could help organisations leverage the knowledge they possess.

The need for systematic knowledge management in requirements engineering has its root in the following:

- Acquiring knowledge about the application domain. This is one of the major challenges in software engineering, since this discipline supports diverse industries such as telecom, health care, insurance, or gaming. Many software vendors are discovering more and more the importance of "mastering" domain knowledge as a way of distinction from competitors. Knowing programming languages, application programming interfaces, engineering tools, and techniques is one half of the assets of a software vendor. The other half is to know the domain, its customers, and its users.
- Capturing and using process and product knowledge. While process knowledge describes how particular tasks should be achieved and how to handle certain issues, product knowledge rather focuses on the work product itself, what it does and how it does it. Capturing and using process and product knowledge allows for shortening the time and cost for developing software systems and for increasing the quality of the delivered software.
- Acquiring knowledge about new technologies, which often affect not only the design of the system but also its goal and behaviour. New technologies can set trends and change the behaviour of users and customers and then the goal of the system itself. For example, modern hardware changed the purpose of a mobile phone from communication to a personal computing device used for work, entertainment, and communication.
- Knowing who knows what. Since software organisations get more and more distributed, this aspect becomes more and more relevant. Software projects are rarely conducted in an isolated manner and from the scratch. Even small projects carried on by a few developers often reuse open-source frameworks with complex functionality, where numerous stakeholders were involved. People get to know each other due to informal talks in the coffee hall [36]. A colleague might then report to the others about issues encountered, how they were solved, and which experiences were gained. Informal knowledge sharing and knowing "who knows what" becomes difficult in distributed settings [37, 38].

Unfortunately, managing requirements knowledge is often pushed to the extreme, that is, either formalised for the purpose of validation and completeness or considered as a "second class" citizen, creating requirements documents just because someone ordered that this task must be done. Our vision of requirements engineering and managing its knowledge is different. *We think that requirements engineering is a knowledge access and sharing activity. In addition to the validation, completeness, and formality of requirements, there must be a second dimension of efficiently capturing this knowledge and sharing it with the right people in the right context.* Managing requirements knowledge is therefore crucial to both requirements engineers and analysts as well as other stakeholders. Examples of the questions that should be answered are who needs this information? What exactly should be implemented in this feature? Why is requirement important? Which restrictions must be considered? What does this concept mean?

Systematically managing requirements knowledge brings several advantages:

- Improved understandability of requirements [39, 40] and reduced mismatch between requirements and their implementations [41]
- Identification of new requirements from the knowledge that is captured in the previous projects [42]
- Solving repetitive problems that occur in requirements engineering by systematising experiences and guiding stakeholders [43]
- Speeding up decision-making by sharing relevant information [40]
- Increased requirements reuse [44] and hence components reuse in general
- Improved evolvability of requirements by providing rationales helpful to decide on future changes [26, 45]
- Improved traceability by capturing implicit links [46] and identifying hidden interdependencies

These advantages are discussed in detail in the following chapters of this book.

### 1.2.4 Knowledge Management in Software Engineering

Over the last two decades, software engineering researchers have given special attention to studying developers' knowledge needs and to suggesting approaches and tools, which improve the access and sharing of software engineering knowledge [47, 48]. Some of the popular and early approaches include rationale management [27], design patterns [49], the experience factory [50], the knowledge dust-to-pearls approach [51], the personal software process [52], the team software process [53], and process-based knowledge management support for software engineering [54]. Except rationale management these approaches have been paid little attention to requirements engineering, focussing on design, implementation, and maintenance. For example, unlike architecture patterns there has been little research on systematically collecting, managing, and using requirements patterns, despite their wide usage in practice.

Also recent empirical studies on the knowledge needs in software engineering barely focussed of requirements stakeholders. For instance, Ko et al. [36] studied the information needs of source code developers at Microsoft and identified 21 questions such as “What is the programme supposed to do” or “What code could have caused this behaviour”. Similarly, Sillito et al. [55] observed developers and identified 44 questions specific to software maintenance tasks. Robillard [56] studied obstacles faced by developers when reusing components. We are unaware of studies on the knowledge needs of stakeholders and questions encountered while capturing or implementing requirements. Such studies are essential for understanding the nature of stakeholders’ work and providing effective tool support.

Finally, knowledge management tools such as document management systems [57], information retrieval and search tools [58], ontology-based repositories [59], wikis [60], and recommendation systems [61, 62] are getting more popular in the software engineering and more recently, also in the requirements engineering community, as the following chapters of this book show.

## 1.3 Foundations of Managing Requirements Knowledge

Managing requirements knowledge involves tasks, methods, and tools, which are scattered across all phases of a software engineering project. We see it as an integrated, continuous process, which includes two main activities as introduced in Def. 1: accessing and sharing knowledge [63]. There are however at least five main foundations for this process, which we introduce below and which correspond to the parts of this book. These are identifying requirements knowledge, representing requirements knowledge for reuse, sharing requirements knowledge, reasoning about requirements, and intelligent tool support. These foundations correspond roughly to the main research topics of this emergent field.

### *1.3.1 Identifying Requirements Knowledge (Part I)*

The first goal of managing requirements knowledge is the identification of relevant knowledge, in particular, in its tacit form, answering the following questions:

- What is requirements knowledge and what are its forms and types?
- How can requirements knowledge be identified, extracted, and externalised systematically?

Identifying tacit requirements knowledge is a complex task. In projects which evolve over a long period of time, or which reuse existing frameworks and libraries, information related to requirements remains “unknown” or undocumented for “historical reasons”. A mobile phone is, for example, developed over decades, and new generations are based on features of older generations. Often several questions remain in the minds of people or "somewhere" in a non-updated requirements document: What exactly does this system or this component do and what it does not? Why is this functionality or quality provided? What are stakeholders' preferences? Or who knows more about this requirement? In contrast, design and engineering questions can at least be partly answered by studying the source code.

Moreover, requirements engineering tasks are often based on assumptions and presuppositions [34]. Customers assume that the developers know the domain, while the developers assume that the customers will tell them about everything that should be implemented – a vicious circle.

Also identifying requirements knowledge in documents and artefacts is not a trivial task. The various stakeholders might have different understanding on what is relevant knowledge, where it should be documented, with which level of detail, and how. As a result, requirements knowledge can easily get scattered across different sources including requirements documents, emails, websites, marketing brochures, contractual documents, and technical documentation. Thus, identifying requirements knowledge is also about identifying which types of information can be captured and found where.

In recent years, researchers paid more attention to the importance of (tacit) requirements knowledge and suggested novel approaches to understand, identify, externalise, and extract this knowledge. This book will discuss some of these approaches such as using machine-learning techniques to extract requirements knowledge [64] from bug repositories or formally capturing requirements using predicate logic to deduce tacit knowledge. Chapter 2 provides a theoretical foundation for tacit knowledge by considering multidisciplinary views of tacit knowledge. Chapter 3 reports on an empirical study on how recording knowledge about defects aids identification of requirements for SPLs. Chapter 4 introduces guidelines to identify and manage requirements knowledge in practice, for example, by drawing a knowledge landscape, interacting with external communities, and establishing a knowledge culture.

### *1.3.2 Representing Requirements Knowledge for Reuse (Part II)*

Knowledge representation is a key issue in successfully applying any knowledge management programme in an organisation. Representing requirements knowledge includes two main challenges: (a) the *efficient* access by all stakeholders and (b) the support of *reuse* in case similar issues arise.

Several requirements-related tasks are repetitive, time consuming, and require a lot of human involvement [39]. For example, requirements analysis for safety critical systems may include hazard and operability analysis or failure mode and effect analysis tasks in order to identify potential system hazards and risks and to mitigate them to acceptable levels before a system is certified. Another example of repetitive time-consuming tasks can be found in large contract-based projects. There, requirements elicitation typically includes the creation of a requirements specification document, which is used as a contractual document. Large IT service providers, which conduct similar projects in the same domain, typically spend valuable time on creating separate requirements specification documents for each project – with copy and paste as the only reuse instrument if at all. Requirements knowledge should be represented in a way that allows for reuse to cope with the repetitive tasks.

There are different approaches to represent requirements knowledge, each with advantages and disadvantages:

- Natural language: This is perhaps the most widely used approach to capture requirements knowledge because it is the most convenient for stakeholders. No tools need to be installed or new techniques learned. The disadvantage is that both querying and reuse are difficult. A promising approach from the social computing paradigm is to annotate text with tags (e.g. #screen, #version4, #issue), which can be either predefined or freely defined and refined by the stakeholders. Over time a folksonomy of tags emerge [65], which can be used to easily browse and find particular information related to a tag. One other important strength of tagging and folksonomies is that they "directly reflect the vocabulary of users" [65]. On the other hand, tags might be ambiguous and allow redundant synonyms.
- Models: Even if the software engineering community has not explicitly considered modelling as a knowledge representation approach, we think that it provides a toolkit for externalising, formalising, and communicating knowledge about complex and manifold software systems. One particular type of requirements knowledge is rationale to decisions. Researchers have suggested different decision models such as "Issue-Based Information System", "Question, Option, and Criteria", "Decision Representation Language", and "NFR Framework" and identified similarities between them. These models enable the representation of rationale knowledge but are rarely used in practice, due to the overhead needed to create them.
- Patterns: Generally speaking, patterns are solution templates to recurring problems. Patterns can be used in requirements to, for example, guide the capturing of specific types of requirements (e.g. patterns to capture use cases or NFRs). Their potential, however, are not yet exploited such as in design or management. Recurrent business rules, domain-specific issues, user tasks, or preference conflicts are just a few examples where patterns can be used to provide solution templates.
- Cases: Capturing requirements as issues with their context enables reuse by analogy, so-called case-based reasoning. Also machine-learning, heuristics, or pattern-matching techniques can be used to identify similar requirements knowledge for the purpose of reuse [64, 66].
- Ontologies: These are formal, explicit specification of shared conceptualisations, which can be understood by both machines and humans. Since the emergence of the Semantic Web standards, ontologies have become a popular alternative for representing reusable knowledge, also in requirements engineering. Several researchers have suggested ontology-based tools and methods to

capture and reason about requirements knowledge [67, 68]. This book includes an experience report on such approaches in Chap. 7.

- Formal approaches: Formal approaches have been studied for decades to capture and validate requirements. This knowledge representation approach focuses on the computer rather than on humans as its correctness is typically high but its usability and understandability is low. These approaches are especially used in to develop safety critical systems.

This book includes three chapters on representing requirements knowledge for reuse. Chapter 5 focuses on eliciting, documenting, and reusing requirements based on patterns. Chapter 6 presents an approach that combines case-based reasoning, natural language processing, and ontologies to systematise the representation of NFR knowledge, in particular security and safety. Chapter 7 presents a similar approach based on ontologies and Web 2.0, focussing on reusing domain knowledge between projects within the same domain.

## *1.3.3 Sharing Requirements Knowledge (Part III)*

Sharing requirements knowledge forms the bridge between capture and reuse. This activity is of particular importance in large distributed projects, where the means for informal exchange “during the coffee break” or “quickly asking questions to the neighbour colleague” are limited.

Methods such as agile include instruments, which systematically encourages knowledge sharing. For instance, the daily stand-up meetings in Scrum enforce people to share the problems they have encountered and the solutions they used. Other methods such as code reviews also enforce knowledge sharing but focussing on design knowledge.

Unfortunately, software engineering processes and tools do not give enough room for sharing requirements knowledge. Most knowledge sharing occurs in meetings and during discussions or at best by delivering requirements documents between stakeholders, which might include hundreds of pages. Distributed settings, lack of domain knowledge, different vocabularies and background, as well as the complexity of requirements knowledge frequently lead to misunderstanding of these documents. It is then more about sharing data and at best information, then sharing knowledge.

Collaborative approaches such as wikis or social media bring new potentials for tightening requirements knowledge sharing. Several authors have suggested the use of wikis to capture and share requirements and related knowledge. For example, Uenalan et al. [69] argue that traditional features of requirements engineering such as projects, folders, specification modules, traceability, and baselines may be provided by simple extensions of wikis. Lohmann et al. [68, 70] introduce a promising approach based on semantic wikis, which enables all stakeholders to collect and semantically annotate requirements. Underlying ontologies enable reasoning about various properties of requirements.

This book includes three chapters on sharing requirements knowledge amongst stakeholders. Chapter 8 focuses on global distributed project and introduces a new knowledge-sharing method and tool called PANEGA. Chapter 9 reports on an empirical study about requirements knowledge sharing in agile projects, distinguishing between performative knowledge, which occurs through actions such as question asking, gestures, or informal speeches, and lexical knowledge sharing, which occurs through inscribed texts. Chapter 10 introduces a Web 2.0 approach for identifying and prioritising stakeholders (i.e. who should know what) and reports on a large empirical evaluation of the approach.

### 1.3.4 Reasoning About Requirements (Part IV)

Reasoning about requirements means considering the requirements as a set rather than single entities, analysing their interdependencies to derive a new knowledge and discover inconsistencies.

Reasoning about requirements and their *interdependencies* is essential in particular for consistency and compatibility management as well as for requirements prioritisation and release planning [71]. Requirements planned for a certain release should be compatible. Incompatibilities can be triggered by not having enough time for consistency checking or by stakeholders' different perceptions and goals. Karlsson et al. [72] indicate that requirements prioritisation and planning approaches have to support handling the interdependencies. Requirements should not be treated independently: Choosing a low-cost–high-priority requirement may also entail the need to include a low-priority–high-cost requirement.

A pairwise comparison of requirements becomes infeasible for larger projects. Ramesh and Jarke [73] point out that traceability maintenance then becomes an issue and that stakeholders should focus on the traceability maintenance for the critical requirements. A common problem of traceability tools is that they do a good job in storing the relationships, but they do not provide clear semantics for the concepts used, which would enable to reason about the basic properties of a given set of requirements. Therefore, it is important to provide a means to identify the most critical dependencies [74].

Especially for informally defined requirements, the complete automation of consistency management is unrealistic [75], but semiautomated tools can help to keep the efforts acceptable. For example, Göknil et al. [76] introduce a requirement meta-model and formalise its language elements. Based on this formalisation, the authors show how to detect inconsistencies in a given instantiation of the meta-model (concrete set of requirements and their interdependencies).

A recent promising approach to reason about requirements uses semantic wiki technologies, enabling all stakeholders (especially in large, distributed settings) to collect and semantically enrich requirements [68]. In order to establish a conceptual foundation, Lohmann et al. [68] have developed the SoftWiki ontology for RE (called SWORE [70]). This ontology defines major RE modelling concepts, such as goal, scenario, or textual descriptions. Furthermore, different types of dependencies between requirements such as "requirement A1 details requirement A" or "requirement A is in conflict with requirement C" are taken into account. Requirements are associated with stakeholders who define and maintain them. Stakeholders discuss the requirements and positively or negatively evaluate them [70]. The dependency types enable the definition of the relationships between requirements and also to reason about different properties. For example, can requirements A, B, and C be part of the same release? Existing semantic wiki-based environments applied in RE require a huge set-up overhead and are limited in the way stakeholders are supported [77].

This book includes three chapters on reasoning about requirements. Chapter 11 suggests a courteous logic-based approach to resolve inconsistency and incompleteness issues. Chapter 12 presents a rule-based approach for detecting dead features and defects in variability. Chapter 13 discusses how reasoning about requirements and their interdependencies should also be propagated to the other activities such as design and implementation.

### *1.3.5 Intelligent Tool Support (Part V)*

Requirements knowledge can become huge and scattered across different sources. Much effort is needed to identify and retrieve relevant information in requirements repositories. This would entail an overhead of capturing, maintaining, and accessing requirements knowledge.

To address these problems, researchers started investigating techniques like data mining, social network analysis, and recommendation technologies and developing information retrieval tools to enable efficient capture, access, and sharing of requirements knowledge.

Recently, traditional requirements databases have been enhanced such that data is modelled and stored in a way that allows learning and querying. Furthermore, researchers have started investigating how recommendation technologies [64, 78] can be applied to existing requirements infrastructures and tools.

Intelligent tool support is crucial for the implementation of any requirements knowledge management programme. Thereby intelligent does not only mean the ability of the tool to reason about knowledge, derive new knowledge, or deal with incomplete and scattered knowledge. Intelligent also means integrated and pragmatic solutions, which neither require additional learning effort, nor impose heavyweight processes and workflows, nor introduce new interruptions to the stakeholders workflows, for example, by having to switch back and forth between tools.

This book includes three chapters on intelligent tool support for managing requirements knowledge. Chapter 14 introduces various recommendation technologies and relevant discusses visionary scenarios of applying them to support stakeholders' tasks. Chapter 15 proposes the use of experience-based tools to improve the quality of software requirements specification by learning from previous experiences. Finally, Chap. 16 introduces the requirements modelling framework, which is integrated into the Eclipse Development Environment allowing a traceability of different types of knowledge such as natural language requirements and formal models.

## 1.4 Summary

In this chapter, we reviewed the concepts of requirements engineering from the knowledge management perspective. We discussed the needs for establishing a new field for managing requirements knowledge and defined its key concepts. Finally, we introduced five foundations for this field: identifying requirements knowledge, capturing requirements knowledge for reuse, sharing requirements knowledge, reasoning about requirements, and intelligent tool support. These foundations are discussed in detail in the corresponding parts of the book.

**Acknowledgements** We are grateful to Pete Sawyer, Smita S Ghaisas, Yang Li, and Zardosht Hodaie for their constructive reviews.

## References

1. Bruegge B, Dutoit A (2010) Object-oriented software engineering using UML, Patterns, and Java, vol 3. Prentice Hall, Upper Saddle River
2. Kang K, Cohen S, Hess J, Nowak W, Peterson S (1990) Feature-oriented domain analysis (FODA) feasibility study. Technical report, CMU/SEI-90-TR-21. Software Engineering Institute, Carnegie Mellon University, Pittsburgh
3. Institute of Electrical and Electronic Engineers (1990) IEEE standard glossary of software engineering terminology (IEEE Std 610.12-1990). Institute of Electrical and Electronics Engineers, New York
4. Clements P, Northrop L (2006) A framework for software product line practice-version 4.2 (2006). Carnegie Mellon, Software Engineering Institute, Pittsburgh. http://www.sei.cmu.edu/prodvolnuctlines/framework.html. Last visited Nov 2012
5. Davis A (2003) The art of requirements triage. IEEE Comput 36(3):42–49
6. Sommerville I, Sawyer P (1997) Requirements engineering: a good practice guide. Wiley, New York
7. Nuseibeh B, Easterbrook S (2000) Requirements engineering: a roadmap. In: Proceedings of the conference on the future of software engineering (ICSE'00). ACM, New York, pp 35–46
8. Aurum A, Wohlin C (2003) The fundamental nature of requirements engineering activities as a decision-making process. Inf Softw Technol 45(14):945–954
9. Pohl K (1996) Process-centered requirements engineering. Wiley, New York
10. Hofmann H, Lehner F (2001) Requirements engineering as a success factor in software projects. IEEE Softw 18(4):58–66

11. Yang D, Wu D, Koolmanojwong S, Brown A, Boehm B (2008) Wikiwinwin: a wiki based system for collaborative requirements negotiation. In: Proceedings of the HICCS, p 24, Waikoloa
12. Leffingwell D (1997) Calculating the return on investment from more effective requirements management. Am Program 10(4):13–16
13. Firesmith D (2004) Prioritizing requirements. J Object Technol 3(8):35–47
14. Gartner Group (2011) Hype cycle for application development: requirements elicitation and simulation. Gartner Group
15. Dorfman M, Thayer RH (1997) Software requirements engineering. IEEE Computer Society Press, Los Alamitos
16. Pohl K, Böckle G, van der Linder F (2005) Software product line engineering foundations, principles, and techniques. Springer, New York
17. Software Engineering Institute (2012) Product line hall of fame. http://www.sei.cmu.edu/productlines/plp_hof.html
18. Smith G (2000) The object-Z specification language, Advances in formal methods series. Kluwer, Boston
19. Creighton O, Software Cinema (2006) Employing digital video in requirements engineering. Dissertation, Technische Universtät München
20. Neill CJ, Laplante PA (2003) Requirements engineering: the state of the practice. IEEE Softw 20(6):40–45, IEEE CS
21. Peterson M (2009) An introduction to decision theory. Cambridge University Press, Cambridge/New York
22. Cooke S, Slack N (1984) Making management decisions. Prentice Hall, Englewood cliffs
23. Kunz W, Rittel H (1970) Issues as elements of information systems. Working paper no. 131. University of California at Berkeley, Institute of Urban and Regional Development, Berkeley
24. MacLean A, Young RM, Bellotti VME, Moran TP (1991) Questions, options, and criteria: elements of design space analysis. Hum Comput Interact 6(3):201–250
25. Dutoit AH (1996) Rationale management in requirements engineering. Ph.D. dissertation, Carnegie Mellon University
26. Dutoit A, Paech B (2003) Eliciting and maintaining knowledge for requirements evolution. In: Aurum A, Jeffery R, Wohlin C, Handzic M (eds) Managing software engineering knowledge. Springer, Berlin
27. Dutoit A, McCall R, Mistrik I, Paech B (2006) Rationale management in software engineering. Springer, Berlin
28. Damian D, Zowghi D (2003) Requirements engineering challenges in multi-site software development organizations. Requir Eng J 8:149–160
29. Chisholm RM (1982) The foundations of knowing. The University of Minnesota Press, Minneapolis
30. Resher N (2003) Epistemology: an introduction to the theory of knowledge. State University of New York Press, Albany
31. Thierauf RJ (1999) Knowledge management systems for business. Praeger
32. Wikipedia, the free encyclopaedia (2012) http://en.wikipedia.org/wiki/Knowledge_management. Last visited in Nov 2012
33. Hansen MT (1999) The search-transfer problem: the role of weak ties in sharing knowledge across organization subunits. Adm Sci Q 44(1):82–111
34. Ma L, Nuseibeh B, Piwek P, De Roeck A, Willis A (2009) On presuppositions in requirements. In: 2nd international workshop on managing requirements knowledge, MaRK'09 IEEE, Atlanta, USA, pp. 27–31
35. Finkelstein A, Kramer J, Nuseibeh B, Finkelstein L, Goedicke M (1992) Viewpoints: a framework for integrating multiple perspectives in system development. Int J Softw Eng Knowl Eng 2–1:31–57
36. Ko AK, DeLine R, Venolia G (2007) Information needs in collocated software development teams. In: Proceedings of the 29th international conference on software engineering, Minneapolis, USA, pp 344–353

37. Herbsleb JD, Mockus A (2003) An empirical study of speed and communication in globally-distributed software development. IEEE Trans Softw Eng 29(6):481–494
38. Milewski AE, Tremaine A, Egan R, Zhang S, Kobler F, O'Sullivan P (2008) Guidelines for effective bridging in global software engineering. In: Proceedings of the 2008 I.E. international conference on global software engineering, pp 23–32. IEEE Computer Society, Washington, DC
39. Daramola O, Stålhane T, Omoronyia I, Sindre G (2013) Using ontologies and machine learning for hazard identification and safety analysis. In: Managing requirements knowledge. Springer
40. Ghaisas S, Ajmeri N (2013) Knowledge-assisted ontology-based requirements evolution. In: Managing requirements knowledge (Chapter 7 in this volume). Springer, Heidelberg
41. Soffer A, Dori D (2012) Model-based requirements engineering framework for systems lifecycle support. In: Managing requirements knowledge (Chapter 13 in this volume). Springer, Heidelberg
42. Lutz R, Lavin M, Lux J, Peters K, Rouquette NF (2013) Mining requirements from operational experience. In: Managing requirements knowledge (Chapter 3 in this volume). Springer, Heidelberg
43. Franch X, Quer C, Renault S, Guerlain C, Palomares C (2012) Constructing and using software requirements patterns. Springer
44. Carrillo de Gea JM, Nicolás J, Alemán JLF, Toval A, Vizcaíno A, Ebert C (2013) Reusing requirements in global software engineering. In: Managing requirements knowledge (Chapter 8 in this volume). Springer, Heidelberg
45. Thurimella AK, Bruegge B (2012) Issue-based variability management. Inf Softw Technol 54(9):933–950
46. Narayan N, Delater A, Paech B, Bruegge B (2011) Enhanced traceability in model-based CASE tools using ontologies and information retrieval. In: Proceedings of the 4th international workshop on managing requirements knowledge (MaRK'11), Trento
47. Bjørnson FO, Dingsøyr T (2008) Knowledge management in software engineering: a systematic review of studied concepts, findings and research methods used. Inf Softw Technol 50:1055–1068
48. Lago P, van Vliet H, Babar MA, Dingsoyr T (eds) (2009) Software architecture knowledge management: theory and practice, 1st edn. Springer
49. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading
50. Basili VR, Caldiera G, Rombach DH (1994) The experience factory, Encyclopedia of software engineering – 2 volume set. Wiley, New York, pp 469–476
51. Basili V, Costa P, Lindvall M, Mendonca M, Seaman C, Tesoriero R, Zelkowitz M (2001) An experience management system for a software engineering research organization. In: Proceedings of the 26th annual NASA Goddard Software engineering workshop. Greenbelt, Maryland, USA
52. Humphrey WS (2005) PSP: a self-improvement process for software engineers. Addison-Wesley, Reading. ISBN 03213054931
53. Humphrey WS (1999) Introduction to the team software process. Addison-Wesley, Reading. ISBN 0-201-47719-X
54. Holz H (2003) Process-based knowledge management support for software engineering. Doctoral dissertation. University of Kaiserslautern, dissertation.de Online- Press
55. Sillito J, Murphy GC, De Volder K (2008) Asking and answering questions during a programming change task. Trans Softw Eng 34:434–451
56. Robillard MP (2009) What makes APIs hard to learn? Answers from developers. IEEE Softw 26:27–34
57. Rus I, Lindvall M, Sinha SS (2001) Knowledge management in software engineering: a state-of-the-art-report. Fraunhofer Center for Experimental Software Engineering Maryland and the University of Maryland for Data and Analysis Center for Software, Department of Defence
58. Bajracharya S, Lopes C (2009) Mining search topics from a code search engine usage log. In: Proceedings of the 2009 6th IEEE international working conference on mining software repositories (MSR'09). IEEE Computer Society, Washington, DC, pp 111–120

59. Happel H-J, Maalej W, Seedorf S (2010) Applications of ontologies in collaborative software development. In: Mistrík I, Grundy J, van der Hoek A, Whitehead J (Hrsg.) Collaborative software engineering. Springer, Berlin/Heidelberg. ISBN 978-3642102936
60. Aguiar A, Dekel U, Merson P (2009) Wikis4SE'2009: Wikis for software engineering. ICSE companion 2009, pp 480–481
61. Happel H-J, Maalej W (2008) Potentials and challenges of recommendation systems for software development. In: RSSE'08: proceedings of the 2008 international workshop on recommendation systems for software engineering, ACM
62. Robillard MP, Walker RJ, Zimmermann T (2010) Recommendation systems for software engineering. IEEE Softw 27(4):80–86
63. Maalej W, Thurimella A (2013) DUFICE – guidelines for a lightweight management of requirements knowledge. In: Managing requirements knowledge. Springer
64. Dumitru H, Gibiec M, Hariri N, Cleland-Huang J, Mobasher B, Castro-Herrera C, Mirakhorli M (2011) On-demand feature recommendations derived from mining public product descriptions. ICSE 2011, pp 181–190
65. Mathes A (2004) Folksonomies: cooperative classification and communication through shared metadata. In: Computer mediated communication – LIS590CMC http://www.adammathes.com/academic/computer-mediated-communication/folksonomies.html
66. Jürgens E, Deissenboeck F, Feilkas M, Hummel B, Schätz B, Wagner S, Domann C, Streit J (2010) Can clone detection support quality assessments of requirements specifications? ICSE (2): 79–88
67. Ajmeri N, Vidhani K, Bhat M, Ghaisas G (2011) An ontology-based method and tool for cross-domain requirements visualization. In: Fourth workshop on managing requirements knowledge, MaRK11, pp 22–23, Trento
68. Lohmann S, Heim P, Auer S, Dietzold S, Riechert R (2008) Semantifying requirements engineering – the softWiki approach, I-SEMANTICS, Graz, pp 182–185
69. Uenalan O, Riegel N, Weber S, Doerr J (2008) Using enhanced wiki-based solutions for managing requirements. First international workshop on managing requirements knowledge (MARK), Barcelona, Spain, pp 63–67
70. Lohmann S, Riechert T, Auer S (2008) Collaborative development of knowledge bases in distributed requirements elicitation. Software engineering (workshops): agile knowledge sharing for distributed software teams, Munich, Germany, pp 22–28
71. Ruhe G, Saliu M (2005) The art and science of software release planning. IEEE Softw 22(6):47–53
72. Karlsson J, Olsson S, Ryan K (1998) Improved practical support for large-scale requirements prioritization. Require Eng J 2(1):51–60
73. Ramesh B, Jarke M (2001) Toward reference models for requirements traceability. IEEE Trans Softw Eng 27(1):58–93
74. Dahlstedt A, Persson A (2003) Requirements interdependencies – moulding the state of research into a research agenda, REFSQ'03, Klagenfurt, pp 71–80
75. Iyer J, Richards D (2004) Evaluation framework for tools that manage requirements inconsistency. In: 9th Australian workshop on requirements engineering (AWRE'04). Adelaide, Australia
76. Göknil A, Kurtev I, and van den Berg K (2008) A metamodeling approach for reasoning about requirements. In: 4th European conference on model driven architecture – foundations and applications, Berlin. LNCS, vol 5095, pp 310–325, Berlin
77. Hoenderboom B, Liang P (2009) A survey of semantic wikis for requirements engineering. Technical report RUG-SEARCH-09-L03, University of Groningen
78. Mobasher B, Cleland-Huang J (2011) Recommender systems in requirements engineering. AI Mag 32(3):81–89
79. Glinz M (2007) On non-functional requirements. In: 15th IEEE international requirements engineering conference, New Delhi, 15–19 Oct 2007, pp 21–26

# Part I
# Identifying Requirements Knowledge

"Knowing yourself is the beginning of all wisdom."
— Aristotle
© Walid Maalej. Printed with permission

# Chapter 2
# Unpacking Tacit Knowledge for Requirements Engineering

**V. Gervasi, R. Gacitua, M. Rouncefield, P. Sawyer, L. Kof, L. Ma, P. Piwek, A. de Roeck, A. Willis, H. Yang, and B. Nuseibeh**

**Abstract** The use of tacit knowledge is a common feature in everyday communication. It allows people to communicate effectively without forcing them to make everything tediously and painstakingly explicit, provided they all share a common understanding of whatever is not made explicit. If this latter criterion does not hold, confusion and misunderstanding will ensue. Tacit knowledge is also commonplace in requirements where it also affords economy of expression. However, the use of tacit knowledge also suffers from the same risk of misunderstanding, with the associated problems of anticipating where it has the potential for confusion and of unravelling where it has played an actual role in misunderstanding. Thus, the effective communication of requirements knowledge (whether verbally, through a document or some other medium) requires an understanding of what knowledge is and isn't (necessarily) held in common. This is very hard to get right as people from different professional and cultural backgrounds are typically involved. At its worst,

V. Gervasi (✉)
University of Pisa, Pisa, Italy
e-mail: gervasi@di.unipi.it

R. Gacitua • M. Rouncefield • P. Sawyer
Lancaster University, Lancaster, United Kingdom
e-mail: r.gacitua@lancaster.ac.uk; m.rouncefield@lancaster.ac.uk; p.sawyer@lancaster.ac.uk

L. Kof
Technische Universität München, Munich, Germany
e-mail: leonid.kof@googlemail.com

L. Ma • P. Piwek • A. de Roeck • A. Willis • H. Yang
The Open University, Milton Keynes, United Kingdom
e-mail: l.ma@open.ac.uk; p.piwek@open.ac.uk; a.deroeck@open.ac.uk; a.g.willis@open.ac.uk; h.yang@open.ac.uk

B. Nuseibeh
The Open University, Milton Keynes, United Kingdom

Lero, Limerick, Ireland
e-mail: Bashar.Nuseibeh@lero.ie

tacit requirements knowledge may lead to software that fails to satisfy the customer's requirements. In this chapter, we review the diverse views of tacit knowledge discussed in the literature from a wide range of disciplines, reflect on their commonalities and differences and propose a conceptual framework for requirements engineering that characterises the different facets of tacit knowledge that distinguish the different views. We then identify methodological and technical challenges for future research on the role of tacit knowledge in requirements engineering.

## 2.1 Introduction

In a celebrated US Department of Defense press briefing, Donald Rumsfeld identified three classes of knowledge. In order of difficulty of acquisition, these were known knowns, known unknowns and unknown unknowns. Although Rumsfeld was talking about military intelligence, the three knowledge classes map surprisingly well onto requirements knowledge. To illustrate this statement, consider two of the roles in requirements engineering (RE): those of requirements analyst and customer. Further, assume that the customer holds all the knowledge that the analyst needs to elicit in order to formulate the customer's requirements. The analyst and customer are just used here for illustration. Bear in mind that there are other roles in RE and that knowledge resides not only in people's heads but also in documents and other media.

Known knowns represent knowledge that the analyst has elicited successfully from the customer so now, they both know it. A known unknown is an item of knowledge that the analyst has not successfully elicited from the customer. However, the analyst knows the knowledge is held by the customer and can thus direct his or her attention to eliciting it from the customer. An unknown unknown is an item of knowledge that the analyst has not successfully elicited from the customer, but in this case, the analyst is unaware that it exists. Thus, the analyst will never elicit the knowledge unless something happens to make them aware that it exists.

Interesting though, these categories of knowledge are, the focus of this chapter is on what Rumsfeld missed, an additional and, for RE, particularly pertinent category: unknown knowns. An unknown known is knowledge that a customer holds but which they withhold from the analyst. Thus, the customer knows it and the analyst doesn't. The interesting point is that there is a range of possible reasons why the customer withholds the knowledge from the analyst. The customer may withhold the knowledge deliberately, perhaps for some perceived personal advantage. They may withhold it accidentally, perhaps not realising the value of their knowledge. They may strive to share the knowledge but end up withholding it because they are unable to articulate it. Or it may be knowledge they don't even realise they hold. Unknown knowns therefore fit at least one of several definitions that exist of tacit knowledge (TK).

The study of tacit knowledge has its roots in Polanyi's seminal work [1] and is inherently interdisciplinary, although much of the work is rooted in linguistics. In RE, TK has long been recognised to pose a problem since eliciting knowledge from a customer can be confounded by any of the factors listed above. However, TK is also essential to RE, as it is at every stage of the software process (and probably to every other enterprise). This is because of the simple economy of expression it affords actors who share an understanding of the TK that forms part of their domain of interest.

Our interest in this chapter is in deconstructing TK as it applies to RE and characterising its different facets to help understand it. The chapter makes three primary contributions. First, in Sect. 2.2, we look to philosophy, linguistics, management science and requirements engineering to provide a wide-ranging critical review of the diverse, multidisciplinary views of tacit knowledge that exist. Secondly, from the review and our analysis of TK exemplars, Sect. 2.3 describes our synthesis of a framework to help identify TK and other forms of hard-to-elicit requirements knowledge. Thirdly, in Sect. 2.4 we identify what we believe to be the outstanding remaining challenges for understanding and dealing with TK in RE, before summarising the chapter in Sect. 2.5.

## 2.2 Review

Tacit knowledge has been defined in a number of ways, as knowledge that we know we have but can't articulate or knowledge that we don't know that we have but nevertheless use. In our everyday life and work, we rely on tacit knowledge to communicate effectively: we do not, we need not and we probably cannot make every assumption we hold explicit; instead, we focus on the essence of what we know and wish to communicate. For example, when ordering a cup of tea from a cafeteria, we do need to ask for the water to be boiled before pouring it into the teapot, *unless* we believe that the person serving the tea does not know that using boiling water is a necessary precondition for a good cup of tea.

Despite considerable interest in tacit knowledge and a general recognition of its importance, there remains some considerable interdisciplinary dispute as to both what tacit knowledge is and how we might reasonably recognise it or measure its effects. In part this is because ideas about tacit knowledge are frequently conceptually confused and generally methodologically weak. There seems to be some agreement on its general importance but considerable disagreement concerning exactly how it might be defined and how we might go about finding it on any particular occasion and mitigate its effects. In this section, we go about 'unpacking' tacit knowledge, pointing to some of the conceptual and disciplinary issues involved in defining and differentiating tacit knowledge and suggesting some of the methodological problems involved in the process of identifying tacit knowledge as part of the requirements process.

The notion of tacit knowledge was first extensively explored by Polanyi in ‘The Tacit Dimension’ [1], and current notions of tacit knowing, or tacit knowledge generally, draw on Polanyi’s famous distinction between tacit and explicit knowing as epitomised in the phrase ‘we can know more than we can tell’ [1]. At the heart of the debate was Polanyi’s concern to understand the precise character of scientific work, much of which appears to be hidden, and hence his argument that much of human knowledge is tacit, difficult to articulate and dependent on different forms of skill and know-how acquired through experience. As Collins writes: ‘The significance of tacit knowledge (for science) is that it shows that much of what passed, in traditional philosophy (of science), as formal, logical and calculative is really deeply invested with the taken-for-granted, the unspoken and the unspeakable. These are things that can be known only through living the collective life (of science) in specific expert communities, not as universal, immutable, articulated truths, knowable by anyone, or anything, anywhere’ [2].

Since Polanyi’s work, tacit knowledge has been described and defined in a variety of (slightly different) ways as skills [3], know-how [4], implicit knowledge [5] and uncodifiable knowledge [6] and more. These apparently small differences crucially reflect important disciplinary differences and motivations, for example, between philosophy, linguistics, business and management science and requirements engineering. Each of these disciplines has rather different approaches to tacit knowledge, reflecting various uncertainties over how tacit knowledge should be defined and operationalised. So, for example, philosophical and disciplinary differences can emerge over whether we are talking about problems of *defining* knowledge or problems of *communicating* knowledge or whether we are adopting a performative or a container-like view of ‘knowing’, knowledge and knowledge management.

While explicit and tacit knowledge may rarely appear as easily separated or distinguished in practice, there are important differences between them in terms of codifiability, storage and transfer. These differences essentially concern whether tacit knowledge (commonly seen as skill or ‘know-how’) can be put in propositional form, abstracted and stored. For example, the readily available procedural organisational operating manuals observable in most companies testify to the capacity of explicit knowledge to be codified, stored and transferred. In contrast, tacit knowledge, however, needs close personal interaction and trust, since it is generally acquired through practical experience, through some form of ‘learning by doing’. These two forms of knowledge therefore differ in both their potential for aggregation and in modes of appropriation. Explicit knowledge can be aggregated, it can be recorded and learned and it can be ‘written down, encoded, explained, or understood’ [7]. But because tacit knowledge is personal and contextual, it cannot easily be written down, formalised or aggregated. Since tacit knowledge cannot be expressed propositionally, exactly how particular work tasks are accomplished and any decision rules that may underlie any skilled performance cannot easily be provided. In these circumstances eliciting requirements obviously becomes very difficult.

### *2.2.1 Tacit Knowledge and Business and Management*

Despite the acknowledged methodological difficulties in researching tacit knowledge, sometimes the existence and importance of tacit knowledge is clearly observable in everyday work. In fields as different as air-traffic control [8], banking [9] or steel-plate rolling [10], participants often refer to their decisions as being based on tacit knowledge interpreted as some form of 'gut instinct' or what Goodwin might call 'professional vision'. Our own ethnographic studies of everyday work in different domains contain frequent references to the existence and importance of tacit knowledge of various kinds. For example, these recorded comments from a steel worker in a rolling mill looking at a slab of white-hot steel that needs rolling into a steel plate [9] '... sometimes you can sit here and look at it and think, "that one's going to be a bastard"' or these comments from a bank manager 'You usually find that the decision you make from your gut is the one you go with' [11] all point to the importance of a range of tacit skills without, of course, giving much insight into what exactly those skills are or how they might be transferred or taught. Our interviews with software developers suggest this is perhaps especially obvious in software development for legacy systems; 'then as the project went ahead we'd send them back prototypes saying we've got something that pulls the data out... is this what it should look like?... and they'd say "no", because we'd expect to see X, Y and Z... so we'd scratch our heads and look at the database and discover more weird anomalies... there was this hidden task... to unravel 5 years of lost knowledge and mess... whenever we are doing legacy integration there is nearly always that scope for ambiguity and tacit knowledge...' (abbreviated MaTREx ethnographic fieldwork notes).

Because tacit knowledge is a notable feature of working life, understanding its business and economic relevance has been a particular focus of research. For some 'sceptical' economists, the very notion of tacit knowledge is anathema [12] 'The concept of the inextricable tacitness of human knowledge forms the basis of arguments... against... every construction of rational decision processes as the foundation of modeling and explaining the actions of individual human agents' [13]. However, while for some economists, for requirements engineers and for software developers tacit knowledge can be seen as a source of *problems*, in business terms, tacit knowledge can sometimes be considered as a critical, if intangible, resource and a source of *competitive advantage*, as Nonaka [14] and others [5, 15] have argued. Nonaka and Takeuchi further distinguish between two types of tacit knowledge that appear important in business and may prove similarly important in requirements: technical tacit knowledge and cognitive tacit knowledge. Technical tacit knowledge is that displayed by skilled craftsmen as a product of repetitive action over time and can be hard to articulate. As Polanyi writes, 'the aim of a skilled performance is achieved by the observance of a set of rules which are not known as such to the person following them' [1]. Cognitive tacit knowledge is internalised by exposure to knowledge, such as in written instructions, and is socially acquired and shared within a group. This can make it hard both to recognise and to explain, perhaps leading a stakeholder to fail to reveal information they mistakenly assume is 'common knowledge'.

For business and management science, tacit knowledge can be considered a resource and confer competitive advantage to a company – indeed it might be considered one of the key sources of competitive advantage. This is because while tangible resources such as equipment, land or stock can simply be purchased or, if necessary, copied, the particular, idiosyncratic character of tacit knowledge means that it cannot [7]. Tacit knowledge is unique, imperfectly mobile and difficult to copy, imitate or substitute. One cannot start up a new company by simply copying the procedure manual of another because the tacit knowledge that lies behind those procedures, the knowledge that makes the procedures work in particular circumstances, with particular customers or particular products, is missing. That tacit knowledge remains 'in the heads' of the workers in the original company. Consequently when a company turns to automating or modernising its systems and procedures, it seems an essential part of the requirements process to try and capture such tacit knowledge in order to maintain competitive advantage since obviously failure to do so means that competitive advantage is lost (though the extent to which such knowledge is codifiable is disputed [13, 16, 17]). Other researchers identify other aspects of tacit knowledge that confer business advantage as a key to managerial success [18] and strategic thinking [19]. While managers are taught the rational or analytical method of management and strategic planning, experience reveals the importance and effectiveness of tacit knowledge in decision-making. Tacit knowledge also becomes relevant as businesses seek to improve their management and organisational support processes through various forms of technology, in particular forms of 'organisational memory' [20, 21] involving the capturing and codification of tacit knowledge and skills into automated, programmed operations.

### *2.2.2 Tacit Knowledge and Requirements Engineering*

The importance of tacit knowledge in requirements engineering is widely acknowledged, since the inability to surface and codify tacit knowledge can contribute to both loss of business opportunities and possible systems failure. However, tacit knowledge appears a problematic phenomenon that is rarely systematically pursued and, at best, tends to be handled in an ad hoc manner when discovered or, rather, chanced upon, in the requirements elicitation process. Because tacit knowledge is so poorly understood, it remains a problem for software development. Developing a deeper understanding of it has the potential to lead to tangible improvements in requirements elicitation, analysis and management practice [11, 22]. Consequently, developing a systematic means to both discover and manage tacit knowledge poses a challenging research problem.

In requirements engineering, there is a strong need to get requirements right since errors in requirements can produce systems that fail, either because they do not function as desired or because they do not match the real needs of users and other stakeholders. In terms of requirements engineering, if people as Polanyi puts it 'know more than they can tell', then clearly sometimes they are not telling all that they know, and this may prove important when eliciting requirements. If stakeholders cannot or will not articulate their objectives, the emerging requirements will appear incomplete, and possible conflicts will be missed threatening failure. The failure to surface tacit knowledge also appears to be implicated in a common range of causes associated with software project failures, such as lack of user input, incomplete requirements, changing requirements, unclear objectives and time frames and so on although their relationship to tacit knowledge is not always clear.

Similarly, when critical knowledge, goals, expectations or assumptions of key stakeholders remain hidden or unshared, then poor requirements and poor systems are a likely, and costly, consequence. Even with good elicitation practice, there may be missing requirements or requirements may be unclear or ambiguous because the analysts lack the context needed to interpret them correctly. As analysts depend on knowledge synthesis from a problem domain in order to derive a requirements specification, it is plausible that the presence of tacit knowledge may adversely affect the resulting requirements specification. For instance, failure to explicate the reasoning behind a tacit knowledge-based process will result in an incomplete requirements specification. An incomplete description of a process requires that readers must use their intuition to arrive at a description possibly producing ambiguity.

Other benefits accruing from the successful identification of tacit knowledge include improvements to the requirements process itself. Finkelstein [23], for example, suggests that tacit knowledge particularly impacts on the ability of requirements engineers to accurately and clearly define a system's boundary. Finkelstein comments that ad hoc decisions are then made as to the required scope of the requirements elicitation process. If many of the system processes are held as tacit knowledge, then defining a reasonable system boundary can never take place, unless this tacit knowledge is somehow exposed, that is, is dependent on appropriate methods for elicitation.

### *2.2.3 Tacit Knowledge and Talking Through Requirements*

Linguistic notions of tacit knowledge suggest that communication takes place against a set of shared background beliefs and that efficient communication effectively depends on such 'common ground' since it avoids endless instances of 'what do you mean by. . .?' Clark [24] uses the term 'common ground' to refer to the shared assumptions and knowledge of people in conversation. Unfortunately in requirements elicitation, such 'common ground' cannot, and probably should not, be simply assumed since the requirements engineer and the stakeholders may possess very different ideas about how processes are performed and what particular phrases are supposed to mean. Unfortunately, experience and our own interviews with software developers as part of the MaTREx project suggest that such common ground is not easily established, even among experienced software teams:

(Software developer A) 'We were thinking about a requirement the other day...relating to a particular aspect of the implementation.... We both had different ideas as to what was implied.... We wanted multiple plug-in windows....'

(Software developer B) 'In fact the low-level ticket says "it would be nice to have multiple plug-in windows". "Multiple plug-in windows" was what we had written down... and (B) understood by that multiple windows by multiple plug-ins.... So each plug-in would have multiple windows... and I understood by it multiple plug-ins each with its own individual window... a slightly smaller step....'

(Software developer A) 'To be fair I don't know whether I had actually solidly internalised the requirement and was interpreting it incorrectly... or if there was an element of "this is what I thought we must have meant by that"'... because it was my own mental image.... 'I think most of our requirements I ever come across are under-specified.... If you specify everything that could be specified in an unambiguous way... you'd run out of ink... maybe I'm just being pessimistic.... They are always pitched at a level where there is a certain amount of tacit knowledge.... When you say FU should do BA you need to understand what FU is and why BAness is worth doing.... It's not feasible to explain all that in every context....' (abbreviated MaTREx ethnographic field notes)

Sutcliffe [25] uses Brennan and Clark's [26] common ground framework to evaluate the affordances of different representations in requirements such as scenarios, storyboards and models suggesting that this approach enables the participants to see the world from each other's viewpoint and provides useful criteria to compose the requirements process enabling stakeholders to choose representations to suit the particular circumstances: 'Common ground in RE involves integrating the abstract and concrete sub-spaces. Juxtaposing abstract (models) and concrete presentations (scenarios, sketches, storyboards) helps to bridge the gap' [25].

A number of other techniques have been developed for uncovering such instances of linguistic tacit knowledge. Maiden and Rugg [22], for example, distinguish between tacit and semi-tacit knowledge where semi-tacit knowledge includes knowledge that is hard to recall without cues, such as the set of functions offered by a large software library, and taken-for-granted knowledge. Such taken-for-granted knowledge can be problematic when the holder of the knowledge assumes wrongly that it is held by others as well. Stakeholders immersed in their problem domain often fail to understand the extent to which their knowledge and expertise is not shared with those working outside the domain, such as an analyst new to the domain, and the effective discovery of semi-tacit knowledge is acutely sensitive to the use of appropriate elicitation techniques – tacit knowledge presents a methodological as well as a conceptual challenge.

There are other developing approaches that can be used to prompt the elicitation of tacit knowledge. For instance, Rugg et al. [27] have used laddering as an aid to tacit knowledge elicitation, in particular for eliciting information about organisational practices and culture. Similarly, Stallinger and Grunbacher [28] have extended WinWin to create EasyWinWin [11], a requirements elicitation technique that seeks to identify instances of tacit knowledge in the problem domain. EasyWinWin is designed to identify, refine and reach consensus on the requirements for a system over a series of steps. EasyWinWin uses prompts and the staged revelation of stakeholders' requirements and priorities to help tease out concealed knowledge. Both the laddering and EasyWinWin techniques are designed to guide the questioning of analysts and in the case of EasyWinWin, to help the negotiation process that occurs whenever conflict arises. Laddering and EasyWinWin are designed to be used during the elicitation phase, as they assume that direct answers from stakeholders, or multiple stakeholders in the case of EasyWinWin, are available. However, Eraut [29] indicates some of the problems surrounding tacit knowledge elicitation using these approaches: highlighting how elicitation exercises are social interactions whose outcome may be influenced by a range of personal, social, organisational and political factors. Consequently, we are most likely to remember 'memorable' rather than common events, and our preconceptions and personal constructs shape behaviour.

The use of requirements documents as the basis for client-contractor agreements means that requirements capture and formulation still need to be carried out as a distinct, initial exercises, and a range of social science techniques have been employed in the requirements process, most notably ethnographic techniques [30]. Ethnographic studies have provided a number of empirical instances of tacit knowledge at work. Ethnography can provide an informed sense of what the work is like in a way that can be useful for designers in scoping their design and providing a better sense of the setting and its work, without necessarily suggesting a rigorous, methodical approach to the surfacing of tacit knowledge. There are, as yet, no formal and rigorous methods for relating the materials obtained through ethnography to orthodox requirements definition. With ethnography, the discovery of tacit knowledge is not guaranteed, far less its translation into explicit knowledge, and so the methodological challenges of surfacing tacit knowledge remain.

## 2.3 A Systematic Framework for TK

When we reflect on the diverging understanding of what tacit knowledge is, as it appears in the literature, we can isolate a small number of properties whose composition gives rise to the different definitions. Although this has been attempted before, as in Collins' [12] distinction between 'relational', 'somatic' and 'collective' tacit knowledge, these distinctions offer little practical help to the hard-pressed requirements engineer. In this section, we propose a novel framework which aims to systematise the subject matter and use it to show how several phenomena commonly occurring in RE practice can be understood as manifestations of TK at work.

### 2.3.1 Knowledge, Information and Documents

Central in our framework is the concept of mental state of a subject. We denote with $k$ a desire, intention, judgement, belief, fact, reasoning rule or algorithm, which is held by a person or conveyed by a document. In other words, any information held by a subject, and any procedure the subject uses in reasoning on that information, is a $k$ in our model.

It is often the case that a given $k$ is amenable to being recorded into a communicable form, usually by being written down in a *document*. We will consider a documented bit of information $d$ as any $k$ that has been expressed by a subject, encoded in some form of language and made permanent and transferable among subjects by rendering it in a concrete physical form, which is perceivable by another subject. A book, an email, the video recording of an interview, an equation on a blackboard and the plumbing drawings of a building are all forms of $d$ according to our definition.

While $k$ and $d$ are both pieces of information, their nature is substantially different. For example, in requirements engineering, *requirements* (as goals or desires of a customer) are a $k$, whereas software requirements specifications (SRS) are a $d$. The process of turning interesting pieces of $k$ into a $d$ is what is commonly called, in this particular context, *elicitation*.

It is worth observing that while $d$s are normally, in some form, tangible objects (and we include electronic documents in this category), $k$s are not. There is no way to inspect and examine directly what is contained in someone's head; so, the next best option is to hypothesise it based on their communication and behaviour. In the classical example, if we observe a person riding a bicycle, we can hypothesise that the information about how to ride a bicycle is available to him, without need of further communication.

### 2.3.2 Stakeholders and Their Goals

The definitions of $k$ and $d$ given above refer to the existence of certain subjects who are the ones that possess and use information. We use $s$ to indicate one such subject. At this stage, we are not interested in the various roles that subjects can serve in a communication, rather just in the fact that they are an involved party. For example, in a software development project, the customer, the analyst and the developer will be considered stakeholders.

It would be an error to consider that in some way, communication patterns among stakeholders are linked to their role. On the contrary, while the main flow of information is traditionally thought to go from customer to analyst and to developers, each exchange is actually made up of a large number of individual communication acts. For example, in an interview with customers, an analyst can use his or her own $k$ to formulate a question and communicate it to the customers by articulating it verbally. Then the customers can use their respective $k$ to formulate an answer and articulate it; the analyst would then interpret the answer in light again of his or her $k$. In each and every of these passages, tacit knowledge could (and usually does) play a role.

The common goal of a given group of stakeholders is to obtain some final results in the interest of all participants. We will call such a goal the project in which stakeholders are participating. The notion of project (which we will denote by $p$) provides both purpose and scope to our reasoning. In fact, individual stakeholders are likely – almost certain – to be involved in several projects at a time, probably multiple ones at work, as well as in their personal and social lives. Different projects will thus have different sets of stakeholders associated with them; we will indicate with $S_p$ the set of stakeholders for project $p$. Moreover, a project implies the notion of a goal to be reached – essentially, the successful performance of the task embodied by the project. In software development, this final goal often coincides with the final delivery of a product. A significant project will always require a concerted effort by multiple participants (we ignore here the solipsist programmer working for his or her own pleasure). The existence of a project does not imply in any way that each stakeholder can have different, and possibly conflicting, goals for the project. The common goal in such a case is to be part of the project and seeing it to completion – but what the final product should be, or even if a successful completion or rather a failure to deliver is more desirable, is still open to each of the stakeholders.

We will not investigate in detail the granularity at which a project should be considered. For example, in developing a software product line, we could consider a larger project concerning the whole product line and smaller projects for each member of the family, as well as micro-projects for the various phases and steps of the development of each member. Similarly, in bespoke development, signing the initial contract could be considered a separate project from the development proper (e.g. company lawyers would not want to be involved in the programming), but at the same time the entire customer-developer relationship, until final acceptance and sign-off, would constitute another project of its own.

As we will see in the following, our analysis about the nature of tacit knowledge happens mostly at a categorical level and is not affected, in principle, by the particularities of each different project. The only important features we ask of $p$ are that there exists a common project and that the project defines a common goal, scope and set of stakeholders. In practical use, of course, categorical reasoning will have to be tempered with the specific circumstances of the project.

## 2.3.3 Writing It Down

Having clarified the scope of our framework, we can define a number of properties (in the form of predicates) that characterise the context in which tacit knowledge can emerge.

In the following, we will use a few symbols from standard predicate logic, namely, $\Rightarrow$ (for implication, $a \Rightarrow b$ means that $b$ is true if $a$ is true), $\wedge$ (for conjunction, $a \wedge b$ is true if and only if both $a$ and $b$ are true) and $\neg$ (for negation, $\neg$a is true if and only if $a$ is false).

A unit of information $k$ is said to be *expressible* by a stakeholder $s$, denoted expressible$_s$($k$), if $s$ is able to commit it to a documented form $d$. This is a predicate of potential, not of act. For example, a given stakeholder can be unable to express some information he possesses for lack of an appropriate language (e.g. inability to write a musical score for a known tune) or because he does not have sufficient understanding of his own knowledge (e.g. inability to express precisely a given colour in words, lacking in-depth knowledge of the concepts of hue, saturation, luminance). It is important to remark that the same bit of information can be available to several stakeholders, but not equally expressible to them. For example, a businesswoman and an economist can have the same knowledge about some facet of the market economy, but only the latter will be able to express it via a set of differential equations, whereas the former will rely on her 'gut feeling' of customers' behaviour. When a $k$ is expressible by all $s \in S_p$, we will simply write *expressible*($k$) dropping the subscript.

A bit of information committed to transferable form is said to be *articulated*. We will write *articulated*$_{s,d}$($k$) to mean that stakeholder s has articulated a given unit $k$ into a document $d$. As usual, we will omit subscripts where they are obvious by the context: for example, *articulated*$_d$*(k)* means that $k$ is in $d$, and we don't care about the author.

The ability to express something and the ability to actually articulate it are different concepts (and as we will see in Sect. 2.3.4, their confusion is one of the causes of differing views of tacit knowledge). In other words, expressible$_s$($k$) is a *potentiality* (meaning that the stakeholder $s$ has the ability to express $k$), whereas articulated$_{s,d}$($k$) is an *actuality* (meaning that the stakeholder $s$ has performed an action to record $k$ in $d$). Generally, articulated$_s$($k$) $\Rightarrow$ expressible$_s$($k$), but it might well be that a $k$ is expressible but not articulated (i.e. because the stakeholder has forgotten to mention it, or actively wants to conceal the information), which is formally written expressible$_s$($k$) $\wedge$ $\neg$articulated$_s$($k$), or that a $k$ is not expressible for a given stakeholder s, but he or she still tries to articulate it, ending up with something different being communicated (formally, $\neg$expressible$_s$($k$) $\wedge$ articulated$_s$($k'$) $\wedge$ $k \neq k'$).

We will denote by *accessible*$_s$($k$) the fact that $k$ is accessible to s in reasoning or acting, or some form of decision-making. To avoid getting into the speculative, we will consider that a $k$ is accessible to $s$ only upon proof that $s$ has used $k$ in performing some task (reasoning or otherwise). In other words, we will not discuss information that could be accessible, but is never accessed. Also, there is no notion of introspection in accessible($k$): a stakeholder might know something and not realise that he knows it (although he might use $k$ continuously, but unconsciously). Again, in the normal case, we have that expressible$_s$($k$) $\Rightarrow$ accessible$_s$($k$), but also in this case, as above, it might happen that a stakeholder tries to express and even articulate something he cannot access – for example, in trying to 'fill in' holes about someone else's knowledge. The result of such an effort would be, most probably, nonsensical or plain wrong.

Finally, to express the scope of the project, we use the predicate *relevant*$_p(k)$ to express that $k$ is a unit of knowledge that is relevant for $p$ to reach its goal. Different stakeholders might have different views of what is relevant for their own subtasks, but this distinction is already incorporated in our notion of project. For example, the programmer's knowledge of a given programming language might be relevant for the success of the implementation subproject, whereas the same knowledge appears irrelevant to the customer (who does not care about which language will be selected, as long as the system delivers what he needs).

Among these four predicates (expressible, articulated, accessible, relevant), a complex web of relationships exists, some of which are illustrated in Table 2.1. We cannot present a full analysis here, but two points are worth highlighting.

First, we are not concerned with whether any specific $k$ is true or not, whether plans formulated according to them will lead to successful completion of the project or whether a certain document is the right place to store information. We focus instead on each single communication act and how it can be influenced by tacit knowledge; the overall management of a software project is better left to other studies.

Second, while we treat the various properties above as boolean predicates, in practice situations can be more nuanced. So, for example, being expressible is not a black and white property, but rather a matter of effort needed to express something: when trying to express which colour she would like to have, Ann could take one year off and devote it to studying colorimetry, then come up with a set of tristimulus curves subtly deviating from the CIE 1964 10° standard observer model... but the effort needed would be incompatible with the goals of the project; hence, we could assume that for all practical purposes, the colour is not expressible for Ann. This is consistent with the notion of 'modicum effort' in the economic view of tacit knowledge. To represent this notion, we can imagine using a fuzzy logic instead of predicate calculus, in handling our properties. We spare the reader the formal details in this chapter.

## 2.3.4 Tacitness

With the definition above, we can now describe our understanding of tacit knowledge succinctly:

$$\text{tacit}_s(k) = \text{accessible}_s(k) \land \neg\text{expressible}_s(k) \tag{2.1}$$

which closely mimics the definition by Polanyi [1]. But we can also express clearly the other views mentioned in Sect. 2.2. For example, the *competitive advantage* idea in (2.1) is

$$\text{relevant}_p(k) \land \text{accessible}_s(k) \land \neg\text{accessible}_c(k) \tag{2.2}$$

**Table 2.1** Different situations in RE practice and how they are characterised by our framework (*N.B empty cells represent 'don't care' values*)

| | *Customer* | | | *Analyst* | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| # | accessible$_c(k)$ | expressible$_c(k)$ | articulated$_{c,i}(k)$ | accessible$_a(k)$ | expressible$_a(k)$ | articulated$_{ant}(k)$ | | Description | Action |
| 1 | Yes | Yes | No | No | | | Yes | *Missing or concealed information:* the customer has the needed information, which is relevant for the development project, and could articulate it Š but it has not been articulated yet. This is the most common case when doing elicitation | Do more elicitation, ask questions |
| 2 | Yes | | No | Yes | Yes | Yes | Yes | *Domain knowledge*: the analyst's domain knowledge avoids the customer having to state everything | Validate any requirements derived from this domain knowledge with the customer to make sure the customer and the analyst share the same understanding |
| 3 | Yes | Yes | Yes | Yes | Yes | Yes | Yes | *Documented requirements*: the customer has communicated the information in such a way that the analyst could understand it; the analyst in turn has | Job completed, well done! |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | articulated (quite possibly, in a different form) it in a URD for later use | |
| 4 | Yes | Yes | Yes | | | | No | | *Noise*: the customer is telling stuff which is not relevant for the project. The analyst's task is to Ņfilter outÓ such*k*, so that it does not pollute the requirements | Filter out irrelevant stuff |
| 5 | Yes | No | | No | | | Yes | (2.1) | *Tacit knowledge*: the customer has some information, can use it, but does not know how to express it. The analyst doesn't have the same knowledge: risk incompleteness | Find or invent a communication medium so that K becomes expressible in that medium (e.g.: video recording of operations) |
| 6 | Yes | No | | Yes | Yes | Yes | Yes | (2.1) | *Tacit knowledge*: common ground or action taken in case 5 makes *k* accessible to the analyst, who articulates it in the form of a requirement and rationale | Add rationale formulated for the derived requirement to the sum of domain knowledge (forming an organizational memory) |
| 7 | Yes | Yes | Yes | No | | | Yes | | *Domain jargon*: the customer is telling important | Call for help, a domain expert might act as ŅbridgeÓ so that |

(continued)

**Table 2.1** (continued)

| | *Customer* | | | *Analyst* | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| # | accessible$_c(k)$ | expressible$_c(k)$ | articulated$_{c,i}(k)$ | accessible$_a(k)$ | expressible$_a(k)$ | articulated$_{ant}(k)$ | | Description | Action |
| | | | | | | | | information, in a way that the analyst cannot understand. Probably, the analyst does not know enough about the domain to access (meaning: use in his own reasoning) that information | the analyst can extract $k$ from $i$ |
| 8 | Yes | Yes | Yes | Yes | | No | Yes | *Lost information*: the analyst got valuable information by the customer, but somehow ŅforgotÓ to document it in a proper requirements document | Use efficient recording (e.g. on-the-field elicitation with mobile devices; interview transcripts; quick cards, etc.) |
| 9 | No | No | | | | | Yes | *Wrong stakeholder*: the customer does not have the sought $k$. Hence, he cannot serve as a source for it | Find the right stakeholder for that piece of knowledge |
| 10 | No | Yes | Yes | | | | Yes | *Babbling*: the customer is telling about things s/he doesn't know about | Probably the customer is trying to act as proxy for someone else; go to the source |

(where the stakeholder $c$ represents the competition and $p$ is the product or service they are competing on). Here, the focus is on having $k$ hard to access by $c$, but whether this is due to difficulties in expressing it or to explicitly not articulating it (e.g. secrets of the trade) is irrelevant for the competitive advantage. Also in (2.1), the notion of *important but difficult to purchase or copy* is

$$\text{relevant}_p(k) \wedge \neg\text{expressible}_s(k) \tag{2.3}$$

whereas the *organisational memory* which is of interest to capture for automation is

$$\text{relevant}_p(k) \wedge \neg\text{articulated}_s(k) \tag{2.4}$$

Then of course, we could have parts of organisational memory which are difficult to articulate because they are not expressible, which would be tacit according to definition (2.1), and others that simply nobody bothered to write down yet, which would not be tacit according to definition (2.1) but is tacit according to (2.4).

Finkelstein'*s* comments in Sect. 2.2.2 on using TK to define system boundaries and scoping could be expressed as suggesting a way to infer $\text{relevant}_p(k)$ by analysing explicit knowledge (i.e. information that has been articulated) and observing that tacit knowledge (according to definition (2.4)) makes inferring $\text{relevant}_p(k)$ difficult.

*Common ground* between two stakeholders $a$ and $b$ (as discussed in Sect. 2.2.3) can be defined by:

$$\text{accessible}_a(k) \wedge \text{accessible}_b(k) \tag{2.5}$$

When common ground is simply assumed (which is deemed risky in RE), additionally $\neg\text{articulated}_{\text{a,b}}(k)$ holds. On the other hand, if $\neg\text{expressible}_{\text{a,b}}(k)$ holds, then just assuming that there is common ground, and that it is shared between stakeholders, is the normal course (and justified when common professional background or previous experiences allow).

Also in Sect. 2.2.3, the whole problem of requirements elicitation can be formalised (in an idealised, perfect form) in our framework as follows: given a set of pre-existing documents $D$ about a project $p$, then create a further document $d^*$ (our SRS) such that

$$\forall k\, s.\text{t. relevant}_p(k),\ \ (\forall\, d\ \in D, \neg\text{articulated}_d(k))\ \Rightarrow\ \text{articulated}_{d*}(k) \tag{2.6}$$

Of course, creating $d^*$ might imply finding the right stakeholder to contribute the information; for example, finding $s\ \in\ S_p$ such that $\text{accessible}_s(k) \wedge \text{expressible}_s(k)$ for each of the $k$ that needs to be articulated in $d^*$ per our previous definition. If no such $s$ can be found in $S_p$, then we are dealing with tacit knowledge according to definition (2.1).

Finally, what ethnographic studies can contribute (Sect. 2.2.3) is identifying if accessible$_s(k)$ holds for a certain stakeholder $s$. In fact, in these studies, we can witness $s$ using $k$ on his or her own, acting and decision-making. Their usefulness lies in the fact that once we witness $k$ being accessed, we can start investigating whether it is expressible (e.g. by asking a direct question to $s$), and if so recording its articulated form, or otherwise at least gaining the knowledge that some tacit knowledge (per definition (2.1)) exists. Take, for example, these abbreviated field notes from an ethnographic study of air-traffic control:

> 11.17 SA [Sector Assistant]: "Chief, there's this he wants" [a neighbouring sector controller requesting a plane's deviation from its planned course]
>
> Chief [Chief Controller]: "all levels are blocked through there" (Spends a moment thinking).
>
> Chief: "no, he's a slow one there's no way he'll be clear then so we'll take him through Liffy"

Note that the aircraft'$s$ unique 'call sign' is never articulated. Tacit knowledge is being used. Formalising what the ethnographer ($a$) observed of the controller ($c$),

$$\text{accessible}_c(k) \wedge \neg\text{articulated}(k) \wedge \neg\text{accessible}_a(k) \wedge \text{relevant}(k)$$

Subsequent interviewing of the Chief revealed in outline how he knew what to do. He realised that the coordination had been done by the Assistant Sector Controllers. It involved an airplane coming from a neighbouring sector of airspace, and from the flight strips and/or radar, he was able to tell which plane must have made the request. Such teamwork is an aspect of the control room members' tacit knowledge that is not normally explicitly expressed except on the occasions when 'normal and routine' working breaks down.

### *2.3.5 Application of Our Framework to RE*

Tacit knowledge can be at work in each communication path in requirements engineering. Whenever two or more stakeholders have an exchange of information, the existence of tacit knowledge can impede, or improve, their communication ability.

Lacking space for a full analysis, in this chapter we discuss only a few cases for one of the most recognisable relationships in RE: an elicitation meeting for a project $p$ between a customer $c$ and an analyst $a$, with $c,\ a \in S_p$. We know from our framework that the predicates expressible(), articulated() and accessible() apply to each of the two stakeholders, whereas relevant() is unique throughout the project. Moreover, the customer will probably articulate his $k$ orally or via short text notes, during the interview (let'$s$ call this document $i$), whereas the analyst will articulate the requirements in written form in a SRS (let'$s$ call this document $t$). The main flow of information will thus start with the customer articulating in $i$ the information he has about the project goals. The analyst will recover information from $i$, integrate it with his or her own $k$ and finally articulate the resulting requirements in $t$.

It is useful to point out that we are not especially concerned with cases of misunderstandings, that is, when the customer articulates some information $k$ and the analyst erroneously understands a different bit of information $k'$. In our analysis, we will model this situation as the composition of two cases: articulated$_{c,i}(k)$ $\wedge$ $\neg$accessible$_a(k)$ (i.e. the customer has articulated $k$, but the analyst has not understood it) and $\neg$articulated$_{c,i}(k')$ $\wedge$ accessible$_a(k')$ (i.e. the analyst has understood a $k'$ which was not articulated by the customer).

With three predicates for each of two subjects, plus the relevant() predicate for the whole project, we have a total of 128 different communication patterns, most of them expressing pathological cases. In addition, we might consider different pairs of subjects (e.g. analyst-developer when the SRS is used to design and implement the system, analyst-tester when the tests are being planned, tester-developer when the results of running tests are returned, analyst-technical writer for preparing a user manual, technical writer-user when such a manual is read by the final user). Clearly, a complete analysis is out of scope, and would likely be of relatively little interest, if performed abstractly. We advocate instead that our framework be applied in concrete cases and mostly when there is evidence of difficulty caused by tacit knowledge.

To give an impression of the kind of situations that can be described, we refer to Table 2.1, where each row represents a situation and empty cells express ‘don’t care’. For each situation, the table provides a description and some (very coarse) advice about which action to perform in the given situation.

Notice how most of the situations described do *not* involve tacit knowledge in sense (2.1) – which is in our opinion the most interesting facet of the concept to be used in RE – but would be considered to so involve TK according to alternative definitions of TK used in different disciplines.

## 2.4 Challenges and Research Agenda

Given our definition of tacit knowledge, it poses three fundamental challenges. These are identifying its presence and, once its presence has been discovered, discerning whether it needs to be articulated and ensuring that the knowledge that needs to be articulated.

*Challenge 1.* Identifying the presence of tacit knowledge[1] is a difficult problem; how can we find something that isn’t there? If the knowledge needs to be made accessible to the analyst but isn’t, then damaging effects will eventually surface – in the worst case, as rejection of the system by a customer or users or as in-service

[1] Here, we mean the presence of tacit knowledge to mean that there is knowledge that is not accessible to anyone except the actor who holds the knowledge.

failure. At this stage, it does not matter whether the knowledge is tacit or whether it is missing information, missing domain knowledge or lost information. Effective mitigation requires that the absence of the knowledge is detected as early as possible, ideally during the requirements elicitation and analysis phase. We need to find clues to its presence. For example, there should be a clear rationale for every requirement. Rationale is itself requirements knowledge and may be crucial for understanding the requirement or for informing trade-offs, so missing rationale may mean that it is tacit. Similarly, a requirement may presuppose knowledge on the part of the reader, perhaps by reference to some domain entity that is not defined. Here, the presupposition is knowledge that isn't expressed, and it may be because there are elements of this knowledge that are hard to articulate.

Of course some knowledge may not be known, knowable or articulated until a system is up and running. It is therefore increasingly accepted that systems may need to be instrumented in ways that allow stakeholders to provide at runtime knowledge that is missing [31]. Indeed Ali et al. [32] argue that in this way, systems can elicit ('sense') tacit social knowledge. Such systems need to be designed to allow users to monitor and communicate social knowledge at runtime and the system can then adapt as necessary.

*Challenge 2*. Knowing that there is some knowledge that is accessible to the customer but not to the analyst is only part of the problem. The other element is discerning whether the knowledge needs to be made accessible to the analyst so that they can articulate it. Note that this is not the same as $\neg\text{articulated}(k) \wedge \text{relevant}(k)$ because not all relevant knowledge needs to be articulated if it can be confidently assumed that it is shared by all the actors who need to use it. Thus, for example, it should not be necessary to specify that a car has a steering wheel. That the car has a steering wheel is clearly relevant and we might need to specify properties of the steering wheel (such as that it holds the stereo controls), but it is reasonable to presuppose the existence of a steering wheel without first needing to articulate its existence. Thus, the problem is knowing what is relevant but should not be presupposed.

Clarke'*s* theory of common ground [24] offers a way of understanding how two actors may be able to tolerate tacit knowledge without its lack of articulation proving problematic for them. However, characterising the common ground that two actors hold is very hard. Shared domain knowledge may serve as a proxy and go some way towards building confidence in mutual comprehension. However, even if the customer and analyst share a deep understanding of the problem domain, the actors downstream in the development process may not, and so the analyst faces the problem of knowing what needs to be articulated. This may be a particular problem for outsourced development where lack of personal contact between developers and analysts may make the level of shared understanding hard to gauge, perhaps complicated further by different cultural norms.

The above two challenges assume that there is a body of elicited requirements knowledge for which we seek to improve our understanding by finding the tacit knowledge and articulating the bits that would otherwise remain inaccessible.

*Challenge 3*. There is a third challenge (or rather, set of challenges) to do with ensuring the knowledge is always articulated where it needs to be. In other words, designing requirements processes that by the use of appropriate techniques helps ensure that all knowledge that is relevant, and needs to be articulated, is documented. Note that this applies to missing information (Table 2.1), domain knowledge and lost information, as much as it does to tacit knowledge. However, our definition of tacit knowledge, and the *expressible* property in particular, is inherently linguistic; just because a customer cannot write what they know down or articulate it verbally, it does not necessarily mean that their knowledge is inaccessible to an analyst. The analyst may not need the knowledge to be expressed if they are able to discern the knowledge for themselves, perhaps by using nonlinguistic means of elicitation, such as observation of the customer in their work setting. Thus, further research is needed to understand the extent to which different elicitation techniques are able to reliably discern tacit knowledge. Ethnography is often mooted as a means to do this [30], but ethnographic fieldwork may miss key events, the handling of which may require tacit knowledge on the part of the actors involved.

In our own work, we are developing tools both to find clues to the presence of the tacit in requirements knowledge and to help ensure that what needs to be is articulated. These tools use a variety of natural language processing techniques to identify domain abstractions [33] to help set the universe of discourse $\{k\ \text{relevant}_p(k)\}$ for requirements discovery and to help inform presupposition analysis. Detection of ambiguity [34] is also being investigated, since ambiguity may be a side effect of obfuscation that is itself a product of trying to express $\neg\text{accessible}(k)$.

To illustrate why we believe tool support is important to helping with the tacit knowledge problem, consider the specification of a new admissions' support system for a UK university [35]. The requirements were expressed in several documents: a high-level document describing the system'*s* motivation and goals and five low-level documents that concretised the system goals as sets of envisioned scenarios. In the implementation, some requirements were misinterpreted or even overlooked. Our specific interest was in tracing, to link rationale from the high-level document to scenarios. However, there were several instances where there was too little shared vocabulary between the documents for the information retrieval techniques used for automatic link generation [36] to identify when the requirements implicit in a scenario contributed to satisfaction of a high-level goal. Our analysis was that these trivial linguistic inconsistencies contributed to a failure to mitigate the lack of common ground between developer and admissions officer. Without access to the rationale for the low-level requirements that existed but was not explicit in the high-level document, the developers lacked the means to correctly interpret or prioritisation of the requirements. This failure of documentation can be plausibly characterised as TK; there was requirements knowledge that was elicited from the admissions officers which was accessible to the analysts ($a$) but not (adequately) expressible by the analysts and so not accessible to the programmers ($p$):

$$\text{accessible}_a(k) \wedge \neg\text{expressible}_a(k) \wedge \neg\text{accessible}_p(k)$$

But there are knowledge elicitation techniques that render almost any accessible knowledge expressible, so the expressible property is at least partially contextual. In the context of the admissions system project and from the viewpoint of the programmers, key knowledge was inaccessible. Such situations are common, and tool support, along with deployment of appropriate elicitation techniques, can help (e.g. concept mapping techniques [35]).

*Challenge 4*. Our framework, as we have formulated it in Sect. 2.3.3, does not consider the time dimension. This is acceptable when considering the situation at a given point in time, or during a short time interval (e.g. a round of interviews with stakeholders), but would be a serious limitation for studying the *evolution* of knowledge and abilities of stakeholders (and analyst) during a long-running process.

In fact, it is often the case that during a development effort, the analyst's knowledge of the domain is increased by the very effort of adequately eliciting requirements; at the same time, the stakeholders' ability to express their knowledge is increased by the pressure put on them by the elicitation, so that knowledge that was previously tacit could, for example, become expressible upon acquiring a sufficiently sophisticated language on both parts. Indeed, this *learning effect* is one of the positive outcomes of any development project – even of failed ones!

It is our experience that the passing of time does not have necessarily a monotonic effect. While one could expect that knowledge and communication improve with time, it may also happen that conflicts, budget or time pressure or a varying political agenda might induce stakeholders to withhold knowledge that they were willing to share previously. Another time-related phenomenon comes into play when the set of stakeholders change during the project (e.g. because a key stakeholder leaves the company).

The model we have presented in this chapter is essentially static (in keeping with our explicit focus on modelling single communication acts, rather than processes) but constitutes groundwork for a more in-depth future analysis of the dynamics of tacit knowledge.

## 2.5 Conclusions

Evidence suggests that tacit knowledge of various kinds appears to be a significant factor in systems failure and loss of business opportunity. Incorrect identification of requirements is a significant source of both customer dissatisfaction and system failures. However, the identification of tacit knowledge is a difficult process for which there is no known all-encompassing methodology, and consequently elicitation of tacit knowledge is rarely pursued in any organised or systematic fashion. Developing a systematic means to discover and to make explicit, tacit knowledge, and to ameliorate its possible effects or at least, to recognise that there is tacit knowledge at work, poses an important and challenging research problem. This challenge is twofold: conceptual and methodological. It is methodological in the need to develop empirical techniques for eliciting tacit knowledge in the requirements process. It is conceptual in that first we need to understand exactly what it is that we are looking for when seeking to unearth tacit knowledge as part of the requirements process.

This chapter has outlined some of the important conceptual issues in understanding tacit knowledge, in order to arrive at a workable and useful definition – useful, that is, in terms of shaping or influencing the requirements engineering process. We have mapped out the different ways in which the notion of tacit knowledge is used in a range of disciplines, particularly those linked to requirements engineering, and worked to develop a logical and, importantly, workable definition.

The aim of our current work is to improve our understanding of how tacit knowledge is manifested in requirements engineering and to develop tools and techniques to provide effective support for mitigating its negative effects. Currently we are investigating the practical impact of tacit knowledge and ambiguity in a number of case studies. Armed with this empirical knowledge of the types of tacit knowledge that manifest themselves in requirements and the extent to which they can be made explicit, we will develop a set of tools that can mitigate the effects of TK in the requirements engineering process. Such tools will not, of course, eliminate tacit knowledge, since tacit knowledge is essentially unavoidable and since requirements elicitation is anyway a 'satisficing' process performed by a stressed and careworn requirements engineer and the various stakeholders, rather than a potentially never ending story performed by analysts with unbounded command of domains, languages and notations. In the end, we must learn to live with some aspects of tacit knowledge – the goal of this chapter has been to make some inroads into understanding what exactly it is that we might have to live with.

**Acknowledgements** The work described in this chapter formed part of the MaTREx (*Ma*king *T*acit *R*equirements *Ex*plicit) project, which was funded by EPSRC grants EP/F069227/1 and EP/F068859/1

## References

1. Polanyi M (1966) The tacit dimension. RKP, London
2. Collins H (2007) Bicycling on the moon: collective tacit knowledge and somatic-limit tacit knowledge. Organ Stud 28(2):257–262
3. Nelson R, Winter S (1982) An evolutionary theory of economic change. Belknap, Cambridge
4. Kogut B, Zander U (1992) Knowledge of the firm: combinative capabilities, and the replication of technology. Organ Sci 3(3):383–397
5. Spender J (1994) Organizational knowledge, collective practice and Penrose rents. Int Bus Rev 3(4):353–367
6. Hu Y (1995) The international transferability of the firm's advantages. Calif Manag Rev 37 (4):73–88
7. Sobol M, Lei D (1994) Environment, manufacturing technology and embedded knowledge. Int J Hum Factor Manufact 4(2):167–189

8. Bentley R, Hughes J, Randall D, Rodden T, Sawyer P, Shapiro D, Sommerville I (1992) Ethnographically-informed systems design for air traffic control. In: Proceedings of CSCW'92, Toronto, Canada, pp 123–129
9. Harper R, Randall D, Rouncefield M (2000) Organizational change in retail finance: an ethnographic perspective (Routledge studies in money and banking). Routledge, London/New York
10. Hartswood M, Procter R, Rouncefield M, Slack R, Voss A (2003) 'Repairing' the machine: a case study of evaluating computer aided detection tools in breast screening. In: Proceedings of the ECSCW, 2003, Helsinki, Finland, pp 375–394
11. Grunbacher P, Briggs R (2001) Surfacing tacit knowledge in requirements negotiation: experiences using EasyWinWin. In: Proceedings of the HICSS'01, Hawaii
12. Collins H (2010) Tacit and explicit knowledge. University of Chicago Press, Chicago/London
13. Cowan P, David P, Foray D (2000) The explicit economics of knowledge codification and tacitness. Ind Corp Chang 9:211–253
14. Nonaka I (1991) The knowledge-creating company. Harv Bus Rev 69(6):96–104
15. Grant J, Gnyawali D (1996) Strategic process improvement through organizational learning. Strategy Leadersh 24(3):28–33
16. Nightingale P (2003) If Nelson and Winter are only half right about tacit knowledge, which half? A Searlean critique of 'codification'. Ind Corp Chang 12(2):149–183
17. Johnson B, Lorenz E, Lundvall B (2002) Why all this fuss about codified and tacit knowledge? Ind Corp Chang 11(2):245–262
18. Sternberg R (1994) Tacit knowledge and job success. In: Anderson N, Herriot P (eds) Assessment and selection in organizations. Wiley, London, pp 27–39
19. Brockmann E, Anthony W (1998) The influence of tacit knowledge and collective mind on strategic planning. J Manag Issues 10:204
20. Walsh JP, Ungson GI (1991) Organizational memory. Acad Manag Rev 16(1):57–91
21. Ackerman M (1996) Definitional and contextual issues in organizational and group memories. Inform Technol People 9(1):10–24
22. Maiden N, Rugg G (1996) ACRE: selecting methods for requirements acquisition. Softw Eng J 11(3):183–192
23. Finkelstein A (2005) Unsolved problems in requirements engineering – a presentation to the British Computer Society's Requirements Engineering Specialist Group Imperial College, London
24. Clark H (1996) Using language. University Press, Cambridge
25. Sutcliffe A (2010) Collaborative requirements engineering: bridging the gulfs between worlds. In: Nurcan S et al (eds) Intentional perspectives on information systems engineering. Springer, Berlin/Heidelberg, pp 353–374
26. Clark H, Brennan S (1991) Perspectives on socially shared cognition. In: Resnick L, Levine J, Teasley S (eds) Perspectives on socially shared cognition, American Psychological Association, Washington, DC, USA, pp 127–149
27. Rugg G, McGeorge P, Maiden N (2000) Method fragments. Expert Syst 17(5):248–257
28. Stallinger F, Grunbacher P (2001) System dynamics modelling and simulation of collaborative requirements engineering. J Syst Softw 59(3):311–321
29. Eraut M (2000) Non – formal learning and tacit knowledge in professional work. Br J Educ Psychol 70(1):113–136
30. Sommerville I, Rodden T, Sawyer P, Bentley R (1993) Sociologists can be surprisingly useful in interactive systems design. In: Proceedings of the conference on people and computers VII, pp 342–354, York
31. Maalej W, Happel H-J, Rashid A (2009) When users become collaborators: towards continuous and context-aware user input. In: Proceedings of the 24th ACM SIGPLAN conference companion on object oriented programming systems languages and applications (OOPSLA 09), Orlando
32. Ali R, Solis C, Salehie M, Omoronyia I, Nuseibeh B, Maalej W (2011) Social sensing: when users become monitors. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th

European conference on foundations of software engineering(ESEC/FSE'11), Szeged, pp 476–479
33. Gacitua R, Sawyer P, Gervasi V (2010) On the effectiveness of abstraction identification in requirements engineering. In: Proceedings of the 18th IEEE international conference on requirements engineering (RE 10), Sydney, pp 5–14
34. Yang H, de Roeck A, Gervasi V, Willis A, Nuseibeh B (2010) Extending nocuous ambiguity analysis for anaphora in natural language requirements. In: Proceedings of the 18th IEEE international conference on requirements engineering (RE 10), Sydney, pp 25–34
35. Kof L, Gacitua R, Rouncefield M, Sawyer P (2010) Concept mapping as a means of requirements tracing. In: Proceedings of the third international workshop on managing requirements knowledge (MaRK'10), Sydney, Australia, pp 22–31
36. Huffman Hayes J, Dekhtyar A, Sundaram SK (2006) Advancing candidate link generation for requirements tracing: the study of methods. IEEE Trans Softw Eng January:4–19

# Chapter 3
# Mining Requirements Knowledge from Operational Experience

**R. Lutz, M. Lavin, J. Lux, K. Peters, and N.F. Rouquette**

**Abstract** This chapter reports results from two recent studies of how operational experience with mission-critical product lines can enhance knowledge management for use with their future products. The challenge was how to propagate new requirements knowledge forward in a product line in ways that projects will use. In the first product line, the concern was capture and retention of requirements knowledge exposed by defects that occurred during operations. This led to two mechanisms not traditionally associated with requirements management – feature models extended with assumption specifications (formal) and structured anecdotes of paradigmatic product-line defects (informal). In the second product line, the traditional notion of binding time in a product line did not accurately reflect the timing of project decisions. This led to a definition of product-line binding times that better accommodates the varying requirements of the different missions using the product line. It appears that the practical techniques reported here to build requirements knowledge into software product lines in the spacecraft domain also are useful in other product-line developments.

## 3.1 Introduction

This chapter reports results from recent studies of two mission-critical product lines: a spacecraft flight software product line and a family of software-defined radios for spacecraft. In both cases certain types of knowledge gained while

R. Lutz (✉)
Iowa State University, Ames, IA, USA
e-mail: rlutz@iastate.edu

M. Lavin • J. Lux • K. Peters • N.F. Rouquette
Jet Propulsion Lab/Caltech, Pasadena, CA, USA
e-mail: milton.l.lavin@jpl.nasa.gov; james.p.lux@jpl.nasa.gov; kenneth.peters@jpl.nasa.gov; nicolas.rouquette@jpl.nasa.gov

operating in-flight or in-flight-like test beds (here, jointly called operational experience) were not being captured for use with future products. In both cases changing this so that the missing information was available to the projects was likely to improve quality and efficiency of their future products. The goal of our studies was to find ways to better propagate this additional requirements knowledge forward to new products in the product line in forms that the projects considered to be convenient.

An earlier description of the first study was published in the MaRK'09 workshop [1]. This chapter revises it and describes it in the context of more recent product-line experience, as reported in the second study.

The first study considered a product line of flight software for spacecraft, previously used on seven missions managed by Jet Propulsion Lab with two more missions in development. We examined flight software defects reported during integration and testing for two earlier members of the product line that were considered by the project to be most relevant to a new member.

We found four types of requirements knowledge revealed by the software defect reports. Store-and-retrieve-based requirements management is insufficient to avoid recurrence of these types of defects on upcoming members of the product line. We thus propose the use of two mechanisms not traditionally associated with requirements management, one formal and one informal, to improve communication of these types of requirements knowledge to developers of future products in the product line. We show how the two proposed mechanisms, namely, feature models extended with assumption specifications (formal) and structured anecdotes of paradigmatic product-line defects (informal), can together improve propagation of the requirements knowledge exposed by these defects to future products in the product line.

We then selected for in-depth analysis defects associated with the Electra family of software-defined radios (SDRs). Discussions with the lead developer of the initial Electra revealed that subsequent versions shared many components – both design and code – and were in effect an emerging product line. To support definition and design of new Electra radios, we conducted a second study to identify product line commonalities/variabilities and examine the associated binding times.

In this second study we found that the traditional notion of binding time in a product line did not accurately reflect the timing of actual project decisions for the Electra SDRs. These SDRs are used on a wide variety of spacecraft. Traditionally, binding time is the point in time at which a decision is made as to which option, among the alternatives available in the product line, will be selected for a new product. We found that we needed to explicitly extend product-line binding times to the requirements phase to describe when the requirements decisions for a new Electra should be made. We show how, by defining binding times as intervals rather than points in time, we can identify and relax some overly stringent binding times for the Electra product-line requirements. This allows different Electra projects to decide which features they need at somewhat different points in time and better accommodates the varying mission requirements of the different spacecraft using Electras. We also added information to the specification of the binding times so that constraints that temporally order the binding times can be better understood and checked later during application engineering. More generally, it appears that the practical techniques reported here to build requirements knowledge in the spacecraft domain also may be useful in other product-line developments.

In describing each study, we start with a problem definition (Sects. 3.2.1 and 3.3.1), then review previous work or context (Sects. 3.2.2 and 3.3.2), outline the study approach (Sects. 3.2.3 and 3.3.3), describe the results (Sects. 3.2.4 and 3.3.4) and examples of using the results (Sects. 3.2.5 and 3.3.5), and end with related work (Sects. 3.2.6 and 3.3.6). A final conclusion summarizes the key ideas from both studies (Sect. 3.4).

## 3.2 Using Operational Defect Reports to Build Requirements Knowledge in Product Lines

Product-line engineering is successful due to systematic reuse of product-line assets for a family of similar products as indicated, for example, by the hall of fame for product lines [2]. The cost advantages of adopting a product-line approach have been widely reported [3–5]. Claims of higher quality are harder to demonstrate but appear to have merit [5]. As a product line matures, the developers' increasing familiarity with the set of product-line artifacts and their growing understanding of the product-line domain assist in the construction of high-quality products. With careful management, requirements knowledge in a product line thus promises to be both incremental and cumulative. We are interested in how this accumulating requirements knowledge can be used to prevent requirements-related defects in product lines. The intent is to avoid repeating in future products the mistakes made in previous products. Thus, two projects that plan to use the product line have encouraged this investigation and helped out with additional domain expertise.

The development of a product line is typically divided into two phases, *domain engineering* and *application engineering*. In the first phase, domain engineering, the product-line assets are developed. In the second phase, application engineering, the product-line assets are reused to build the new product. The most important of these assets are the software requirements, the shared software architecture, and the reusable components and test suites. The requirements are defined by means of a commonality and variability analysis (CVA) that identifies the common requirements, called *commonalities*, that are shared by all members of the product line and the variable requirements, called *variabilities*, that some but not all products have. An example of a commonality from the spacecraft domain is the requirement that all spacecraft have software fault protection to respond to power loss. An example of a variability is the selection of the reaction wheels used to control spacecraft attitude (i.e., position) for a specific mission. The specific option or alternative selected is a *variation*.

### *3.2.1 Using Operational Defect Reports: Problem Definition*

We report experience from an effort to use defect reports from previous product-line members to improve the quality of future members by reducing their requirements-related defects. We show that the requirements knowledge available in defect reports of previous products goes far beyond simply avoiding similar mistakes in the future. Capture of tacit requirements knowledge implicated in such reports supports incremental improvement of the product line as described below. To this end, we suggest adding defect reports as another key product-line asset.

We focus on requirements-related defect reports because requirements defects are difficult to detect and costly to correct in the completed product [6]. This is primarily because requirements-related defect reports often involve missing or incorrect requirements discovered during operational experience about the environment or subtle interactions among the subsystems (often timing-related or resource dependencies). These missing or incorrect requirements often stem from undocumented assumptions.

We describe results from an analysis of defects reported during integration and testing for two earlier members of a spacecraft flight software product line to uncover tacit requirements knowledge for the benefit of an upcoming new product. Our analysis suggests opportunities for the defect reports to serve as a richer source of information for mining tacit requirements knowledge for future product-line members. Although these data are available, their role in the product line is yet to be defined. Historically, the focus has been on avoiding recurrence of defects in the current product rather than on using this knowledge to improve future products.

In pursuing this investigation, we faced two challenges:

1. *How to capture relevant product-line requirements knowledge from the defect reports*. This involves acquiring access to members' defect reports from previous product-line applications (nontrivial, as some of these belonged to other organizations), analyzing them and deciding how to filter out information that was not relevant to future products (e.g., one-time variations). We describe below how we identify relevant defect reports and how we use Orthogonal Defect Classification (ODC) [7] to identify requirements-related patterns in the defect data. We also describe an extension to the feature model to capture tacit assumptions.
2. *How to proactively communicate the new requirements-related information to developers of future product-line members*. Initially, we concentrated on determining how to specify and store the information so that it could be retrieved readily by future projects. However, after reviewing our preliminary results with a domain expert who had many years of project experience [8], we realized that making the requirements knowledge available to future product-line developers took us only halfway toward our objective of propagating information forward to new product-line members. It is insufficient merely to make previous defects available for retrieval by future developers (a "pull" mechanism for querying stored information). In addition, it is necessary to provide a "push" mechanism

so that the requirements knowledge that could have prevented the defects in previous product-line applications will be remembered in future product-line applications.

We introduce below the concept of a Product-Line Analysis Defect Paradigm (PLA-DP) (extending Petroski's design paradigm [9]) to propagate forward requirements knowledge gained through product-line analysis of recurring patterns of requirements-related defects. Essentially, we identify a small set of representative anecdotes where a serious defect involving a failure or near failure occurred that could have been prevented by additional requirements knowledge unavailable at the time.

This anecdotal approach to "remembering old lessons learned" (as the domain expert put it) is compatible with the culture of an organization such as ours that builds high-dependability systems. Such anecdotes command the attention of future developers because of their concrete descriptions of subtle interactions and dependencies of the software requirements on hardware idiosyncrasies and environmental rare events.

Addressing the first challenge involves an activity to incorporate defect reports from previous products into the set of product-line assets. Addressing the second challenge involves an activity to make it easy to reuse the defect-report information for the next product in the product line. We hope that this combination of "pull" mechanisms (storing and querying defect reports as product-line assets) and "push" mechanisms (telling stories via PLA-DPs) will help improve the management of requirements knowledge derived from defect reports in product lines.

## *3.2.2 Using Operational Defect Reports: Previous Work*

To the best of our knowledge, this work represents the first attempt at using defect analysis to incrementally improve the requirements knowledge for a product line. There have been multiple efforts to use defect reports to measure quality (e.g., bugs remaining) [10, 11], improve the organization's development process [6, 12–15], and predict future fault occurrences [7, 16], but these efforts have not addressed product-line improvement.

By capturing the missing or tacit requirements knowledge in prior systems, we seek to reduce requirements-related defects in future systems. We use Kruchten, Lago, and van Vliet's definition of tacit knowledge as knowledge that is essential but not documented [17]. Historically, requirements-related defects (e.g., missing requirements, incorrect requirements, or misunderstood requirements) have caused the most problems in terms of time-consuming debugging and nonlocalized fixes.

### *3.2.3 Using Operational Defect Reports: Analysis*

The operational defect data used in this analysis to inform the GRAIL project came from two earlier spacecraft –MRO (Mars Reconnaissance Orbiter) and ODY (Mars Odyssey). The GRAIL project launched twin spacecraft in 2011 to perform gravity mapping of the moon. GRAIL has approximately 3,000 low-level software requirements textually documented in the DOORS requirements management toolset. A subcontractor developed the GRAIL flight software by using the assets of a mature product line. GRAIL has significant reuse of the MRO flight software, a previous instance of the subcontractor's product line. MRO launched in 2005 and is currently orbiting Mars on a 5-year mission. We analyzed software defect reports from MRO and from an earlier spacecraft in the flight software product line, ODY. ODY launched in 2001 and is currently operational on an extended science mission. It is also well known for its role in conveying transmissions from the two Martian rovers to Earth.

Because GRAIL was in development at the time this analysis was done, GRAIL had the opportunity to benefit from insights and concerns from systematic study of the MRO and ODY defect reports. Figure 3.1 shows an overview of this process. On the left we see a sequence of spacecraft in the product line going from top to bottom down the page. The curved arrows indicate that each inherits the shared (common) software base. In the middle, the horizontal arrows show how each of these projects produces a set of Problem/Failure Reports (PFRs) that is recorded in the institutional, problem-reporting database. The dotted line focuses on MRO and GRAIL. The curved arrow on the right-hand side of the figure represents our goal: to build requirements knowledge from defect reports for previous product-line members (MRO and ODY) in order to reduce defects on the new product-line member (GRAIL).

The dataset for the analysis consisted of the 69 MRO and 24 ODY PFRs classified as flight software-related in the Jet Propulsion Laboratory (JPL) problem-reporting database. The online PFRs filled out by the project consist of three parts. The first part describes the problem and is filled out by the tester when the problem occurs. The second part is filled out by the analyst assigned to investigate the problem. The third part is filled in later with a description of the corrective action that was taken to close out the problem. We looked at all the MRO and ODY software PFRs, even for features that would not be used by GRAIL, since subsequent spacecraft in the product line might require those features. Many of these features involve more extensive redundancy and fault protection than GRAIL (a relatively inexpensive mission) could afford.

We analyzed the MRO and ODY PFRs using a variation of ODC (Orthogonal Defect Classification) [7]. ODC is a technique that we have previously used to analyze both PFRs and postlaunch anomalies on spacecraft [14]. It provides a way to "extract signatures from defects" [7] and to correlate the defects to attributes of the development process. ODC differs from causal analysis, another widely used

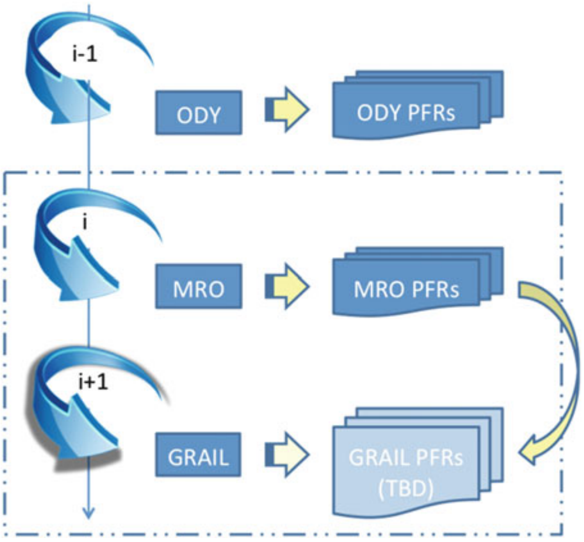


**Fig. 3.1** Product-line approach to building requirements knowledge from defect reports [1]

defect analysis technique, which employs a manual, in-depth search for root cause, usually of a subset of defects.

The ODC analysis used a script to extract the relevant fields from the PFR database and then classified each PFR in terms of Cause, Target, Problem Type, and Subsystem. The Cause was extracted from the PFR cause field, for example, "Software design" or "Support equipment (software)." The Target described the entity that was fixed or changed to avoid the problem in the future. It was extracted from the PFR disposition field. The Problem Type characterized the fix. It was manually classified from the textual description of the problem/failure in the PFR form, for example, "Algorithm" or "Timing." We also extracted and recorded the MRO subsystem for which the fix/change was made, as well as some other information to help with follow-on analyses.

## 3.2.4 *Using Operational Defect Reports: Results*

Summary results from the defect analysis are shown in Fig. 3.2. The tall bars at the back of Fig. 3.2 show Algorithms as the most frequent PFR problem for flight software (totaling 34), followed by Configuration PFRs (11), Hardware Design PFRs (8), and Timing PFRs (7) in bars near the foreground of the figure. Requirements-related PFRs are common, but requirements was rarely selected as the primary cause once code existed. Thus, although no PFRs on MRO and only 6 PFRs on ODY had requirements listed as their cause, we will see that many of the algorithm and timing problems were, in fact, caused by undocumented tacit requirements.

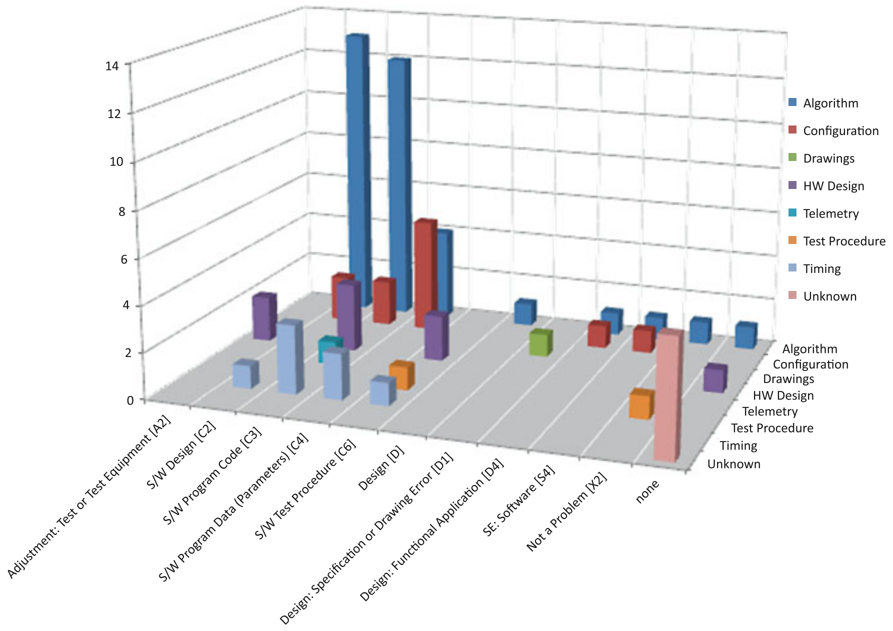


**Fig. 3.2** ODC analysis of MRO defect reports (Adapted from [1])

The investigation showed that requirements-related defects do recur in the product line. For example, ODY had problems related to managing the mode of operation for the spacecraft's attitude control system and its devices, including reaction wheels. In MRO, the problem of managing the mode of operation for the spacecraft's attitude control system became more difficult, for example, due to additional redundancy in the power system for the reaction wheels. GRAIL has common features with ODY and MRO (including attitude control via reaction wheels) but variation in the amount of redundancy and in the configuration of devices for attitude control. There is thus a need to ensure that GRAIL will not have a repeat of problems regarding the management of mode of operation for the spacecraft's attitude control system.

The analysis found four types of requirements-related knowledge in the MRO and ODY software-related PFRs:

1. *Newly discovered requirements*. These defect reports describe missing or incomplete requirements where the knowledge needed to identify the requirement surfaced only during testing. Consistent with our earlier studies on some spacecraft not in a product line [14], these new requirements often involved complicated interface issues between software variabilities or between hardware and software variabilities. Several of the incomplete requirements involved fault protection, which is of special concern in high-dependability systems such as these. Many of these newly discovered requirements will also be needed in some future systems in the product line.

2. *Unexpected requirements dependencies.* A closely related knowledge type was the uncovering during testing of unexpected dependencies among existing variability requirements. These usually involved new knowledge about coordination constraints and were often revealed as inconsistent states. In some cases the unexpected, latent requirements dependency involved an incorrect assumption. This new or corrected knowledge about requirements dependencies will be needed by future systems in the product line to avoid recurrence. A potential concern is whether some dependencies were resolved by one-time workarounds that may not carry over to the next product.
3. *Tacit requirements rationales.* These undocumented rationales, that is, "justifications of decisions" [18], contributed to defect occurrences, either because the rationale was not sufficiently documented or because it was incorrect. This information was not available to the tester regarding why a requirement had to be the way it was. These rationales often involved the hardware or environment. Sometimes the new knowledge involved an idiosyncrasy that was known but not explicitly documented. This kind of requirements knowledge is especially useful to future developers during evolution of the product line, as it captures potential, unintended impacts of changing requirements. The importance of requirements rationales was underlined in a recent NASA investigation of risks associated with the growth in complexity of flight software. This NASA investigation's second recommendation in order of importance was "emphasize requirements rationale" [19].
4. *Misunderstood requirements.* Some defect reports were caused by requirements or requirements-related information that was in some sense documented, but in such a manner that it confused the developer or the tester. Often this occurred because the documentation was partial or ambiguous. Such gaps in requirements understanding often surface when the software behavior is accurate but surprises the testers, leading them to initiate a defect report. In previous work we found similar cases where the software behaved correctly but unexpectedly [14]. It seems likely that, in a product line, a similar phenomenon will occur. Requirements-related information that confuses developers on one member of the product line, if not clarified, can confuse developers of subsequent product-line members. This suggests that in a product line, improving the communication of requirements knowledge can help preclude making the same mistake on later products.

### 3.2.5 Using Operational Defect Reports: Use of the Results

Earlier in the chapter (Sect. 3.2.1) we described two challenges to using the requirements knowledge gained from defect analysis of individual product-line members to improve the requirements of subsequent product-line members. The first challenge has to do with preserving the new requirements-related knowledge captured from defect reports so that it can be reused across the product line as part of the domain-engineered product-line assets. The second challenge has to do with conveying the new requirements knowledge to developers of a future project for use in its application engineering.

We now describe how we tried to address these two challenges through the use of mechanisms not normally associated with requirements management, namely:

1. To formally preserve new requirements-related knowledge, we *extend feature models with assumption specifications*.
2. To informally convey the new requirements knowledge, we use *structured anecdotes of paradigmatic defects*.

Together, these two mechanisms appear to help build and propagate the four types of requirements knowledge exposed by the PFRs for use in future products in the product line.

#### 3.2.5.1 Preserving New Requirements Knowledge by Extending the Feature Model

Associating new requirements knowledge with the features in a product line provides a natural way to preserve this information for future product-line applications. A feature model describes the common and variable requirements of a product line by showing the structural relationships (aggregation and generalization) and dependencies (e.g., required, excluded) among the features [3, 20]. To incorporate new requirements knowledge, we build on Lago and van Vliet's approach to modeling tacit assumptions in an architecture-based product-line feature model [21].

To document an assumption, one first identifies the features directly influenced by the assumption and then defines the dependencies between the assumption and the feature model. The feature model specifies which features are potentially impacted by an assumption, as well as on which assumptions a feature depends. Architectural modules and interfaces implement each feature.

#### 3.2.5.2 Application of the Extended Feature Model: MRO Transponder

The transponder is the spacecraft receiver/transmitter used for telecommunications. During system testing on MRO, a false assumption regarding the transponder was discovered, resulting in new requirements knowledge. It was assumed that the transponder state always reflected the state of the carrier, that is, locked or unlocked. However, it was found in system testing that these values could be temporarily out of synchronization when the carrier detection was transitioning between locked and unlocked. The consequence was that the flight software requirement for fault-protection checking of the transponder telemetry had to be revised. New timing-related software requirements arising from the asynchronous carrier lock had to be captured. Moreover, since the transponder was a product-line

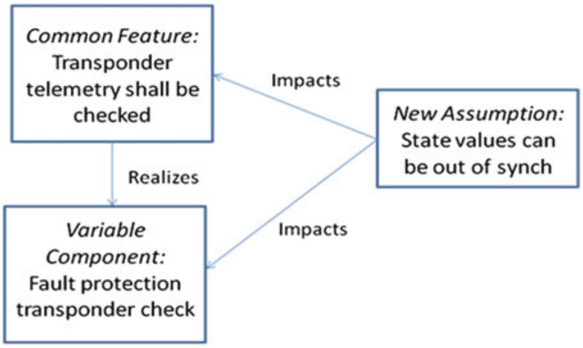

*Common Feature:* Transponder telemetry shall be checked
Impacts
*New Assumption:* State values can be out of synch
Realizes
Impacts
*Variable Component:* Fault protection transponder check

**Fig. 3.3** Feature model extended with assumption [1]

asset, the new knowledge needed to be preserved for other product-line members. Figure 3.3 shows a simplified diagram of how the corrected assumption is preserved by incorporating it into a product-line feature model.

To date we have avoided scalability concerns by proposing to record only those assumptions implicated in defect reports in the feature model. This decision is consistent with the scope of our current effort to use knowledge about defects in past systems to improve future systems in the product line. Because defect reports are systematically tracked to closure, it appears practical to maintain these defect-related assumption links in the feature model as the product line evolves. However, it is an open question and beyond the scope of this effort whether it would be feasible, or even desirable, to record in a feature model all the many assumptions for complex systems such as the spacecraft. By instead focusing on the historically troublesome assumptions, the extended feature model remains readable and compelling.

### 3.2.5.3 Capturing New Product-Line Requirements by Constructing PLA-DPs

To capture tacit requirements knowledge from past defect reports for use in future products in the product line, we use Product-Line Analysis Defect Paradigms (PLA-DPs). PLA-DPs are stories or anecdotes of failures or near failures on previous products. They are concrete instances of a pattern of defects found via the product-line analysis of the defect reports. These anecdotes make the tacit requirements knowledge manifest and more actively involve the developer in scrutinizing previous product-line experience for requirements-related information relevant to the current product. The documentation of anecdotes helps ensure that lessons learned on early instances of the product line do not become lessons lost on later instances.

The PLA-DPs extend Petroski's design paradigms [9] to requirements. A design paradigm is a case study "capable of being presented as a fresh and memorable story" that embodies "a general principle of design error" that can also arise in new situations. It both improves understanding and alerts developers to common pitfalls [9]. For example, Petroski describes the collapse of the Tacoma Narrows Bridge, subtitling it “a paradigm of the selective use of history” and describes the risk of design myopia that prolonged success can bring.

The PLA-DP is a good fit with product lines because product-line requirements have a high degree of commonality. Explicit and sustained attention to past anomalies characterizes spacecraft design projects, most especially when the spacecraft inherit components from each other or form part of a product line. For example, Bayer [22] has described 14 MRO postlaunch anomalies in detail, including the story of each problem as it unfolded (often from the perspective of the operator) together with recovery attempts, the causal chain and contributing factors, root causes, safety nets used, corrective actions, and the lessons learned. Presentations to subsequent projects and lunchtime seminars help propagate the memory of those anomalies.

One recommendation arising from our study was to explicitly include PLA-DP accounts in the domain-engineered product-line repository. By associating them with the defect reports whose lessons they generalize, PLA-DP accounts can be automatically extracted and distributed (i.e., pushed) to each new project. Since the intent is to make the PLA-DPs not only accessible but also notorious, each textual account is best accompanied by a graphic-rich slide set that visually presents the story of the failure or near-miss together with the insights for future systems in the product line. Dudley Herschbach, a Nobel Laureate in chemistry, has similarly described the use of compelling stories as a way to teach the habit of “actively scrutinizing evidence and puzzling out answers” and has urged emphasis on the “human adventure of intellectual exploration, replete with foibles and failures but ultimately achieving wondrous insights” [23]. The PLA-DPs are intended to convey a similar excitement and insight.

#### 3.2.5.4 Application of the PLA-DP: MRO Reaction Wheel

We next give a simplified version of an example PLA-DP from MRO experience, followed by an explanation of its consequence for subsequent spacecraft in the product line.

During integration testing of MRO, the electrical power subsystem reported an unexpected software state. Analysis showed that this was due to a race condition that could occur if the reaction-wheel subsystem (responsible for aiming the spacecraft) was powered off and then back on within 5 min. In this case an uplinked sequence of preprogrammed software commands had turned the reaction wheels off. A minute later the onboard fault-protection software had commanded the reaction wheels back on. This caused the power software to be “confused” about what state the wheels were in. The fix entailed a new software requirement to disable a schedule-verification function whenever the reaction wheel was being commanded.

This anecdote identifies additional requirements-related information regarding time-related constraints on the allowable interactions between the software sequence and the fault-protection software in this product line. This is important information not only for the software developers on this project but also for the developers on the other spacecraft in this product line. This anecdote also records a rationale for why the flight software sometimes disables the schedule-verification function. More generally, this anecdote alerts the developer to identify which commands in this product line can be issued (thus, can compete) by different software systems (here, fault protection and sequences) to the same hardware component. This is especially important as timing-related defects, such as this race condition, are probably more expensive to fix than function defects, according to a recent study [15].

### *3.2.6 Using Operational Defect Reports: Related Work*

The analysis presented here draws together work in requirements management of dependencies, rationales, and assumptions with work in defect analysis and applies those to product lines. Most discussion of knowledge management in software development has described architectural knowledge [17] rather than requirements knowledge. Where requirements have been considered, the authors usually address the elicitation of tacit requirements [24, 25] rather than requirements maintenance over the lifetime of a product line. An exception is Stoiber and Glinz's description of how knowledge about component dependencies in a product line is often tacit and distributed across an organization. They propose explicit modeling using decision tables and aspect modeling [26]. Savolainen and Sajaniemi described a structured feature model that provided a very detailed specification for each feature, including error behavior [27]. They described the need to capture complex feature interactions and to make feature behavior conditional on the presence and absence of other features. However, unlike our work, they did not consider information from defect reports.

Dutoit and Paech included the use of rationales in use case and scenario-based modeling [28]. They defined rationale to be the justification of system or process decisions, including description of options. Their work surveyed rationale management methods and described a process to elicit, document, and maintain rationale using web-based tool support for a two-column table, where requirements appear in the left column and rationales in the right column. However, unlike the work described here, they considered a single system rather than a product line and provided no discussion of defects.

As described above, Lago and van Vliet showed how assumptions about the execution environment can be incorporated into a graphical feature model by adding assumption nodes and links to the set of features influenced by the assumption [21]. They found that they had to add new features to the model (i.e., make implicit decisions explicit) in order to characterize the assumptions, that assumptions could crosscut features, and that the dependencies among features and assumptions could be complicated. They recommended the explicit modeling of assumptions both to enhance understanding and traceability and to explore the effect of changing assumptions.

Jirapanthong and Zisman advocated using traceability relations in product-line engineering [29]. They described the automatic generation of traceability relations among feature-based documents. However, they did not include defect reports among the eight types of documents that they considered.

Defect analysis during testing has been used to evaluate the readiness of software for release and to estimate the reliability of the software [7]. Fenton and Ohlsson have described the problems in using such defect analysis results to measure the quality of deployed software [10]. Dalal, Hamada, Matthews, and Patton have used ODC to guide prerelease process improvement [12]. Ostrand and Weyuker compared pre- and post-release faults in an investigation of module fault density and fault proneness components [11].

While many authors have described the use of defect mining to improve the quality of a single project or of the development process, there has been little work that uses defect analysis results to improve or manage the requirements knowledge needed for a product line. One exception is Maalej and Happel's suggestion of providing a hierarchical schema for classifying errors in order to enable finding similar situations for which experience already exists [30]. Mohagheghi, Conradi, Killi, and Schwarz studied reused components and found that they had lower defect density than non-reused ones but more defects with highest severity [31]. At the architectural level, Trew used root cause analysis of 900 problem reports to identify and package rules to reduce integration errors in a product line [32].

Defect analysis has shown that misunderstanding of requirements and their underlying rationales frequently cause defects. Lauesen and Vinter, for example, looked at 200 of the 800 defect reports available a few months after a product's release. They found that about half of the defect reports involved requirements defects, with missing requirements being the most frequent cause [13]. Similarly, in an early study of testing defects in the spacecraft domain, we found that the most common causes of critical software defects were misunderstanding the software's interfaces with the system and discrepancies (e.g., omissions or inaccuracies) between documented requirements and actual requirements [6]. Van Lamsweerde and Letier's classification of common information-related obstacles to achieving requirements goals (e.g., Information unavailable, Information not in time, and Wrong belief) [33] can also be used to describe requirements defects.

## 3.3 Using Operational Experience to Build Requirements Knowledge of Product-Line Binding Times

We next report experience with product-line binding times on a family of software-defined radios (SDR) called Electras, developed at Caltech's Jet Propulsion Laboratory (JPL). Electra SDRs are used for communication among spacecraft, for example, between a Mars orbiter and a rover on the surface.

Binding time in a product line is the point at which alternative or optional features are selected for a new product [5, 34–37]. In this section we describe difficulties we faced with regard to feature binding times for the Electra SDR product line; the solutions we developed, together with their advantages and shortcomings; and how we think these solutions can be useful more generally in other product lines. We illustrate the discussion with examples from Electra SDR products.

In a software product line, the products share a common set of mandatory features but are differentiated one from the other by their variant features. Each feature carries an increment of functionality for the system [38]. We recently performed a product-line commonality and variability analysis (CVA) of features for the Electra SDR product line [39]. Many of the features are commonalities shared by all Electra products. Examples of *common* features are that an Electra shall provide nonvolatile memory, that an Electra shall provide conversion between analog signals and digital data streams, and that an Electra shall provide File Storage Management. Other features are *variabilities* (optional or alternative features) that are required in some but not all Electra SDR products. Examples of Electra features that are *optional* are that a new Electra may or may not provide error detection and retransmission, may or may not support two-way Doppler ranging, and may or may not provide Kalman filtering of the signals. Examples of Electra features that are *alternative* are the telemetry encoding choices, the ranging algorithm(s) choices, and the waveform (signal processing) applications choices that are to be provided by a specific Electra SDR product.

### *3.3.1 Product-Line Binding Times: Problem Definition*

A step that gave us trouble in the CVA was determining and documenting the appropriate binding times of the features. Deciding when to bind is not an easy problem. Krueger describes the choice of the right variation binding time as “often one of the most critical, most contentious and least well understood issues for Software Product Line organizations” [40]. Related works in the literature acknowledge the difficulty of binding, but give limited, pragmatic guidance for developers. In fact, as described below, there is not even a consensus on what are the available binding times. To develop binding times for the Electra SDR product line, we needed more guidance than is available in the literature. More specifically, we discovered that project decisions impact the timing of the bindings for feature variations in a new Electra.

### *3.3.2 Product-Line Binding Times: Study Context (Software-Defined Radios (SDRs))*

Before we delve into binding times, it may be useful to describe the context in which this work was done – namely, a software-defined radio (SDR) called Electra.

An SDR is a telecommunications device that uses software to implement functionality traditionally implemented in hardware, such as signal processing. For deep space missions to other planets, the flexibility and adaptability available with software implementation is advantageous. Such spacecraft often need to communicate with different types of vehicles and radio protocols and may need to be maintained or enhanced over long lifetimes to complete their missions.

The Electra SDRs use a JPL-designed baseband processor module (BPM) containing a CPU, reconfigurable FPGAs, and the analog/digital conversion coupled to an analog RF Module, RF power amplifier, and power supplies. Unlike previous transponders [41], the hardware is developed by a vendor, but the digital design and software intellectual property are developed and tested at JPL. The term "software," as used by the Electra product line, includes the code for specifying the behavior of FPGAs.

Electras are used on a variety of current and future spacecraft, including orbiters (MRO, the Mars Reconnaissance Orbiter that launched in 2005; Maven, the Mars Atmosphere and Volatile EvolutioN orbiter to be launched in 2013; the 2016 ExoMars Trace Gas Orbiter), landers, rovers (the Electra-Lite on the Mars Science Laboratory's rover launched in 2011), and the JPL-SDR CoNNeCT (NASA's Communications, Navigation and Networking re-Configurable Testbed, to be launched to the International Space Station in 2012). A next-generation deep space transponder being built by NASA, called the UST (Universal Space Transponder), will also use an Electra SDR.

### 3.3.3 Product-Line Binding Times: Analysis

The binding times for variabilities are determined in the domain engineering phase as part of the CVA. To keep the documentation easy to read and update by the Electra SDR projects, we used a tabular decision model to record the set of decisions that have to be made to build a new product in the product line [5]. Each decision assigns a legal value to a parameter of variability. The binding time associated with each parameter of variability imposes a partial order on the set of decisions. For example, the memory size must be decided before the waveform (signal processing) applications are decided, but the memory size and the frequency bands that will be supported can be decided in any order, since they have no effect on each other. The decision model, produced in the domain engineering phase, functions as an instruction sheet for building a new product in the application engineering phase.

To help make the CVA complete and correct, we employed usage scenarios derived from use cases. These verified that the core common features and expected variant features were identified, given past experience and our knowledge of the roles that Electra SDRs are likely to play in future missions.

Discussion of the draft CVA with internal customers revealed several groups of potential users, with each group having distinct interests and needs. The project formulation team is primarily interested in high-level functionality that is responsive to mission requirements, plus rough costing rules of thumb. Those involved in requirements and preliminary design are primarily interested in trade-offs that support make-or-buy decisions, the use of legacy hardware and software, and more detailed costing rules for implementation effort. Detailed designers and implementers make decisions about such things as the amount of RAM, the configuration of FPGAs, selection of available utilities to support error detection and correction, and the extent of fault recovery. To obtain better insight into these decisions, we created an activity model of the overall design and development process and focused our attention on the choices that were being made at each stage.

## *3.3.4 Product-Line Binding Times: Results*

Experience with the Electra SDR application engineering showed that the typical decision gets made in a preliminary way in an early activity, gets rethought and refined as development proceeds, and becomes firm (committed to implementation) at a certain point in the process. In short, the binding of variabilities often does not occur at a point in time but is instead a process that extends over several development activities and may, in some instances, span two or more project phases.

This observation led us to make three enhancements to the traditional product-line binding times, as described in the next three subsections. The goal of these enhancements was to better align the documentation of feature binding times to the reality of the timing of the project decisions.

### 3.3.4.1 Extending Binding Times to the Requirements Phase

The first enhancement we made to traditional binding time documentation was to explicitly extend product-line binding times to the requirements phase. This was done to better reflect and guide the preliminary, feature binding time decisions that are made during the requirements phase of the application engineering of a new Electra SDR product.

Projects usually begin selecting the features for a new Electra SDR early, during the requirements phase. In the literature, some authors describe early-phase binding times (e.g., Clements et al. describe possible binding times as including design, compile, link, and runtime [18]), while other authors describe only binding times that occur later (implementation and runtime) or describe the binding time as the point at which the feature is “physically included” or incorporated into a product [36]. Svahnberg et al. list product architecture derivation, compilation, linking, and runtime, as possible binding times, but explicitly exclude design and implementation, as they require the selection to be done by the compiler, linker, or runtime system [37]. The problem we encountered was how to reconcile the way the projects work (requirements-time binding of features) with the implementation bias in the literature regarding product-line binding.

There are several advantages to preliminary, requirements-phase binding of features that explain why projects do it. Some features are determined early because they are essential to the mission and/or are contractual in nature. For example, many fault protection and adaptability features need to be built into an Electra for a Martian orbiter because it will be in service for many years, supporting a variety of scientific missions. The requirements for the next Electra SDR may also be driven by the availability of new technology. An example would be providing the capability to perform data compression of images. This is an innovative feature of an Electra SDR that is specified as a variation, not yet implemented, and may be needed for a future mission. If so, that project would select the data-compression feature at requirements time, in order to ensure its availability at integration time. We also found in the Electra SDR product line that variations that could cause resource contention tended to be bound early (e.g., features that involve memory, bandwidth, or power). Variations that could have an impact outside the Electra SDR box, such as external interfaces, also tended to have early binding to avoid system integration problems.

#### 3.3.4.2 Recognizing that Binding Time Can Be an Interval

The second enhancement we made was to define the binding time as an interval. This was done to avoid imposing overly stringent binding times on feature decisions. As noted above, what we saw in the Electra SDR projects was that decisions were often made in a preliminary way in the requirements phase but were reconsidered later in development. We wanted to support this flexibility where it was appropriate by capturing in the documentation both the feature decisions that were made during the requirements phase and the interval of time during which it was still possible to revise the decision.

In order to avoid overly stringent binding times, which can cause unnecessary rework, complexity, and cost, we define a time interval during which the decision is made as to which choice, among the alternative or optional features available in the product line, will be selected for a new product. We document for each parameter of variation: (1) the earliest possible time at which it makes sense to decide and (2) the last time at which the decision can be made (the project event by which point the decision had to be made). For example, several decisions can be made at any point up until the board is sent for fabrication. At that point, the discussion is closed.

For ease of use by the project, these endpoints in time are tied to standard project development-phase milestones, as defined by the organization. Binding times may be documented in a table (Table 3.1 shows a segment of such a table for illustration) where each variation is a row and each of the seven binding times (project milestones) is a column. This alignment of binding times with project milestones during application engineering provides clear guidance to a new project building an Electra SDR using a familiar vocabulary and timeline.

**Table 3.1** Sample segment of binding time table

**Legend**

**X: An initial design decision is made now, but it will/could be refined later (there may be multiple times where design decisions are commonly made/refined)**

**B: Bound – design choices can not be altered after this time (without major rework)**

**B(HW): Bound for the hardware components only**

**B(SW): Bound for software components only (Arrows added in two rows to indicate binding intervals more clearly)**

| **Binding time ➔ Feature 🡇** | **Proposal** | **Flight system design** | **Subsystem design** | **HW procure-ment and build** | **SW feature selection /buy** | **SW implemen tation** | **Operations** |
|---|---|---|---|---|---|---|---|
| Internal visibility for test and diagnosis | - | - | X ⟹ | - | - | ⟹ B (SW) | - |
| Frequency selection | X ⟹ | - | - | ⟹ B (HW) | X ⟹ | ⟹ B (SW) | - |
| Operational scheduling/ sequencing | - | X | - | - | X | - | B (SW) |
| Host spacecraft control and data interface | X | - | - | B (HW) | - | - | - |

The flexibility of representing binding time as an interval lessens some of the risks of too early binding. A recent NASA study noted that developers often do not realize the "cascade" of increased downstream complexity entailed by their local decisions [19]. Representing binding time as an interval acknowledges the reality that different Electra SDR products will bind the same variation feature in somewhat different phases.

We found that, contrary to the notion of binding at a point in time for all products in a product line, different Electra SDRs needed slightly different binding times. Many features available on future Electra radios are expected to be waveform applications for signal processing bought from third party vendors. Sometimes it is not known early in the development process exactly which waveform applications will be needed. Having a binding time interval that can accommodate this type of uncertainty helps reduce the risk of expensive rework to accommodate the desired waveform. In addition, most such decisions can appropriately be made during an interval of time. This is especially true for software, such as the different signal-processing waveform applications or ranging algorithms that will be available on different Electras.

Handling binding time as an interval is also useful for the Electra SDR product line because requirements for a particular product often evolve both during development and after launch. Indeed, delivery commitments of a new Electra SDR are sometimes made even before the requirements for the mission are finalized.

Moreover, having a binding time interval is useful in knowing whether the feature decision still can be revised when something unforeseen comes up late in development.

Cost considerations also play a role, in that it is easier to defer decisions on desirable but nonessential features until the development budget is better understood. The alternative, that is, selecting features early and later removing them, can entail costly redesign or modification. Similarly, since the cost of an unimplemented option may not be accurately predicted, making an early decision to use such an option is more likely to be reversed than later decisions. In sum, because evolution of requirements on a particular Electra is expected, having an interval during which binding is allowed is advantageous.

#### 3.3.4.3 Reflect Temporal Dependencies Among Binding Decisions

The third enhancement we made to the binding time process was to add additional information to the binding time specifications to describe temporal dependencies among the decisions. We especially wanted to give the projects an easy way to see which hardware decisions related to a feature had to be made before the software decision regarding that feature could be made. Developers identified these temporal constraints among decisions as an area that needed be better understood and described. The effort was to make the ordering of related hardware/software decisions easier to identify and check when new Electras were built.

The binding times of variations are, in accordance with reality, partially ordered in the CVA. This means that most decisions can be made independently of each other but that some decisions constrain other, subsequent decisions. This partial ordering is expressible in the decision model [5]. The dependencies among the decisions prevent inconsistent, infeasible, or undesirable combinations of choices from being made. In general, earlier decisions (e.g., hardware manufacture) are those that are likely to change less often than the later ones (e.g., software waveform applications).

## *3.3.5 Product-Line Binding Times: Use of the Results*

Table 3.1 (above) illustrates some binding time characterizations in a typical Electra development cycle. Note that for the Electra product line, the software variations that can be selected often depend on prior hardware decisions. For example, the decision as to which frequency channels will be selectable by software in a new Electra SDR product is limited by the decision as to which frequency band will be supported by the radio frequency hardware. This is why it is so useful for the CVA table to distinguish binding times for feature-related hardware decisions from binding times for software feature decisions.

When a decision is made during application engineering (i.e., when a new Electra is being built), two checks are run: that the decision was made within the prescribed time period (binding time interval) and that all decisions that should have been made before it were made (ordering constraints). More formally, the intervals and constraints impose a topological ordering on the decisions. The goal is not to find an optimal sequence of decisions, nor a schedule, nor a critical path, although the information could be used for all of these. Instead, the goal is to check that the new Electra project's sequence of decisions will not cause problems. Currently projects choose to do this manually, but automation of this check is feasible [35, 42].

To summarize, in the domain engineering phase, we represented the decisions as to which features to include in a new Electra SDR as a set of intervals. In the application engineering phase, where the decisions are made for a new Electra SDR, we use the binding times for the decisions to show (1) which decisions should be made in that interval, (2) which decisions must be made by the endpoint milestones (usually a formal review), and (3) which decisions must have been made before this decision and (4) to check that any decisions made at this point are within the intervals specified for them by the decision model.

Interestingly, we found that the single greatest value of the binding time documentation for the projects was not to verify a new Electra project's decisions but to give the project guidance ahead of time as to which decisions had to be made by what deadlines. That is, the primary use of our results is in providing a checklist for the project.

In response to suggestions by a telecommunications manager, we also added information regarding whether or not each variation feature had flown previously. This was important information because features that have flown tend to be better understood, with change and defect logs that provide additional insight into their interfaces and operational profile, and consequently are considered to be lower risk. Associating this additional information with binding times makes the feature binding documentation in the CVA more useful to the Electra SDR projects.

#### 3.3.5.1 Product-Line Binding Times: Illustration from MRO

An MRO operational anomaly described in [22] illustrates how difficult and how important it is to bind the right alternative at the right time. In this case the project had to reverse an earlier binding decision regarding the parameter-update feature for the Mars Reconnaissance Orbiter (MRO) Electra. At the time the MRO Electra SDR was built, there were two choices for updating parameter values during operations. The first alternative was parameter updating of stopped threads only. This selection meant that to change operational parameters (such as the telemetry heartbeat generation interval) during flight, the thread was stopped, new parameters were passed in, and then the thread was restarted. Parameters changed only when the thread was not running. The second alternative was to allow parameters to be updated while the threads were running.

Early in the Electra design for MRO, the decision was made to use the first alternative, allowing parameter updating only to those threads that were stopped, rather than to support parameter updating of running threads. An unintended side effect of the early binding of this decision was that it was later found to be incompatible with the selection and usage model of the real-time operating system. Specifically, when a thread was restarted after a parameter update, it could cause other threads to not release all their resources (e.g., semaphores).

MRO was launched in 2005 and had an extended and successful science mission. It also provided relay service for Phoenix, a spacecraft that landed on the surface of Mars in 2008. MRO transmitted commands from Earth ground operations down to Phoenix, and science data up from Phoenix and on to Earth scientists.

During operations, a critical anomaly occurred. The MRO Electra, after sending repeated "hail" signals to Phoenix, stopped updating the heartbeat telemetry to the spacecraft computer. In response to the absent heartbeat, the MRO spacecraft powered off its Electra as expected, and Phoenix switched to using another spacecraft for relay support. Investigation following the anomaly revealed a root cause to be inadequate semaphore analysis. The anomaly caused the project to reverse the original design decision to update only stopped threads. Instead, the other alternative (supporting parameter updates of running threads) was included in the spacecraft software. The change was made by uplinking the modified software to the spacecraft.

In this case, the side effects of the early binding included the rework and risk associated with updating operating software. If the consequences had been understood at the time that the binding decision was made, or if the decision had been made later, the second alternative would have been selected. This experience from MRO shows the importance of two of the results described above: extending the consideration of binding time to requirements definition and recognizing that binding may occur during a time interval.

### *3.3.6 Product-Line Binding Times: Related Work*

There are few studies of binding time during development. Niu, Savolainen, and Yu describe how to represent and tolerate inconsistency in viewpoints to support late binding [43]. Czarnecki, Helsen, and Eisenecker show how the choices available at each phase can be represented by separate feature models [34]. Dolstra, Florijn, and Visser describe in two case studies variation points that can be bound at more than one point in the development cycles [44]. Other studies of binding time have focused on runtime binding as an alternative to static binding. For example, Rosenmüller et al. describe flexible binding times that can occur either at compile time or runtime [45]. This differs from the binding times we describe here, which occur within a defined interval during development, but may occur at any time within the interval. In general, runtime binding is fairly rare on spacecraft since postlaunch software changes are risky.

## 3.4 Conclusion

This chapter examined two different approaches to improving the engineering of product-line requirements and illustrated how each has been used in designing and developing spacecraft. The results seem to be applicable to other product lines, as well.

In the first approach, a study of defect reports from operational product-line members, we identified four kinds of requirements knowledge that can be mined from these reports: (1) newly discovered requirements, which describe missing or incomplete requirements; (2) unexpected requirements dependencies, which often involve new knowledge about coordination constraints; (3) tacit requirements rationales, which are essential to alert future developers to possible unintended consequences of requirements changes; and (4) misunderstood requirements caused by incomplete, ambiguous, or inconsistent documentation. This study identified two enhancements to the engineering of product-line requirements:

- Portray new requirements knowledge by using an extended feature model to show structural relationships and dependencies.
- Capture new knowledge via Product-line Analysis Defect Paradigms, which comprise anecdotes or stories about prior anomalies or failures.

In the second approach, a study of binding times in a family of software-defined radios, we showed that the typical decision gets made in a preliminary way in an early phase, gets rethought and refined as development proceeds, and becomes firm at a certain point in the process. This study identified three enhancements to standard binding times in order to better align them with the development of a product line:

- Explicitly extend product-line binding times to the requirements phase to better reflect and guide the preliminary decisions.
- Define the binding time as an interval to support changes in requirements, interfaces, or component availability during the development process and bug fixes or enhancements during operations.
- Add information about ordering constraints on the decisions, most especially about hardware decisions that have to occur before software variations are selected.

**Acknowledgments** The authors thank Michel Ingham and Margaret Smith for insightful feedback on an early version of this report. The research described in this chapter was carried out in part at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautic and Space Administration and funded by NASA's OSMA Software Assurance Research Program. The first author's research is supported in part by NSF grant 0916275. Part of this work was performed while the first author was on sabbatical at Caltech and the Open University, UK.

## References

1. Lutz R, Rouquette N (2009) Using defect reports to build requirements knowledge in product lines. In: Proceedings of the 2nd international workshop on managing requirements knowledge (MaRK09), Atlanta, 1 Sept 2009, pp 12–21
2. Software Engineering Institute. Product line hall of fame. http://www.sei.cmu.edu/productlines/plp_hof.html
3. Kang KC, Lee J, Donohoe P (2002) Feature-oriented product line engineering. IEEE Softw 9(4):58–65
4. Pohl K, Bockle G, van der Linden F (2005) Software product line engineering: foundations, principles and techniques. Springer, Heidelberg
5. Weiss D, Lai C (1999) Software product line engineering: a family-based software development process. Addison-Wesley, Reading
6. Lutz R (1993) Analyzing software requirements errors in safety-critical, embedded systems. In: Proceedings of the IEEE international symposium on requirements engineering, IEEE CS Press, pp 126–133
7. Chillarege R, Bhandari IS, Chaar JK, Halliday MJ, Moebus DS, Ray BK, Wong M-Y (1992) Orthogonal defect classification – a concept for in-process measurements. IEEE Trans Softw Eng, pp 943–956
8. Rasmussen R (2009) Thinking outside the box to reduce complexity in NASA flight software, App. H. In: Dvorak D (ed) NASA Study on flight software complexity. http://oceexternal.nasa.gov/OCE_LIB/pdf/1021608main_FSWC_Final_Report.pdf
9. Petroski H (1994) Design paradigms: case histories of error and judgment in engineering. Cambridge University Press, Cambridge/New York
10. Fenton NE, Ohlsson N (2000) Quantitative analysis of faults and failures in a complex software system. IEEE Trans Softw Eng 26(8):797–814
11. Ostrand TJ, Weyuker EJ (2002) The distribution of faults in a large industrial software system. In: Proceedings of the international symposium on software testing and analysis, in Software engineering notes, Roma, Italy, pp 55–64
12. Dalal S, Hamada M, Matthews P, Patton G (1999) Using defect patterns to uncover opportunities for improvement. In: Proceedings of the international conference on applications of software measurement, San Jose, CA
13. Lauesen S, Vinter O (2001) Preventing requirements defects: an experiment in process improvement. Requir Eng J 6:37–50
14. Lutz R, Mikulski IC (2003) Requirements discovery during the testing of safety-critical software. ICSE 2003, Portland, OR, pp 578–585
15. Shull R, Basili V, Boehm B, Winsor Brown A, Costa P, Lindvall M, Port D, Rus I, Tesoriero R, Zelkowitz M (2002) What we have learned about fighting defects. In: Proceedings of the 8th IEEE symposium on software metrics, Ottawa, CA, pp 249–258
16. Leszak M, Perry DE, Stoll D (2002) Classification and evaluation of defects in a project retrospective. J Syst Softw 61(3):173–187
17. Kruchten P, Lago P, van Vliet H (2006) Building up and reasoning about architectural knowledge. In: QoSA 2006, Lecture notes in computer science, vol 4214, Västerås, Sweden, pp 43–58
18. Clements P et al (2011) Documenting software architectures, views and beyond, 2nd edn. Addison Wesley, Upper Saddle River
19. Dvorak DL (ed) (2009) Final report: NASA study on flight software complexity. http://oceexternal.nasa.gov/OCE_LIB/pdf/1021608main_FSWC_Final_Report.pdf. Accessed 5 Mar 2009
20. Cho H, Lee K, Kang KC (2008) Feature relation and dependency management: an aspect-oriented approach. In: SPLC 2008, Limerick, Ireland, pp 3–11
21. Lago P, van Vliet H (2005) Explicit assumptions enrich architectural models. ICSE'05, St. Louis, MO, pp 206–214

22. Bayer TJ (2008) Mars reconnaissance orbiter in-flight anomalies and lessons learned: an update. IEEE aerospace conference, Big Sky, pp 1–11
23. Herschbach D (2003) The impossible takes a little longer. In: Marshall SP, Scheppler JA, Palmisano MJ (eds) Science literacy for the twenty-first century. Prometheus, Amherst
24. Grunbacher P, Briggs RO (2001) Surfacing tacit knowledge in requirements negotiation: experiences using EasyWinWin. In: Proceedings of the 34th HICSS, Maui, Hawaii, pp 1062–1069
25. Stone A, Sawyer P (2006) Identifying tacit knowledge-based requirements. IEE Proc Softw 153(6):211–218
26. Stoiber R, Glinz M (2009) Modeling and managing tacit product line requirements knowledge. In: Proceedings of the 2nd international workshop on managing requirements knowledge, Atlanta, GA, pp 60–64
27. Savolainen P, Sajaniemi J (2008) Improving knowledge sharing in embedded software production line. In: Proceedings of the 1st international workshop on managing requirements knowledge (MARK'08), Barcelona, Spain, pp 68–72
28. Dutoit A, Paech B (2000) Rationale management in software engineering. In: Chang SK (ed) Handbook of software engineering and knowledge engineering. World Scientific Publishing, River Edge, pp 787–816
29. Jirapanthong W, Zisman A (2005) Supporting product line development through traceabilility. In: APSEC'05, Taipei, Taiwan, pp 506–514
30. Maalej W, Happel H-J (2008) A lightweight approach for knowledge sharing in distributed software teams. In: Proceedings of the 7th conference on practical aspects of knowledge management. Lecture notes in computer science, vol 5345, Yokohama, Japan, pp 14–25
31. Mohagheghi P, Conradi R, Killi OM, Schwarz H (2004) An empirical study of software reuse vs. defect-density and stability. In: ICSE 2004, Edinburgh, Scotland, pp 282–292
32. Trew T (2005) Enabling the smooth integration of core assets: defining and packaging architectural rules for a family of embedded products. In: SPLC 2005, Rennes, France, pp 137–149
33. van Lamsweerde A, Letier E (2000) Handling obstacles in goal-oriented requirements engineering. IEEE Trans Softw Eng 26(10):978–1005
34. Czarnecki K, Helsen S, Eisenecker UW (2005) Staged configuration through specialization and multilevel configuration of feature models. Softw Process Improv Pract 10:143–169
35. Dehlinger J, Lutz R (2011) Gaia-PL: a product-line engineering approach for efficiently designing multi-agent systems. TOSEM 20(4):1–27
36. Lee J, Kang KC (2004) Feature binding analysis for product line component development. In: van der Linden F (ed) PFE 2003, Lecture notes in computer science, vol 3014, Siena, Italy, pp 250–260
37. Svahnberg M, van Gurp J, Bosch J (2005) A taxonomy of variability realization techniques. Softw Pract Exper 35:705–754
38. Batory D, Benavides B, Ruiz-Cortes A (2006) Automated analysis of feature models: challenges ahead. CACM 49(12):45–47
39. Lux J, Lavin M, Lutz R (2010) Commonality and variability analysis of Electra product line. JPL, 30 Sept 2010
40. Krueger CW (2004) Product line binding times: what you don't know can hurt you. In: SPLC. Lecture notes in computer science, vol 3154, Boston, MA, pp 305–306
41. Koski E, Linn C (2006) The JTRS program: software-defined radios as a software product line. In: SPLC, Baltimore, MD, pp 182–191
42. Padmanabhan P, Lutz R (2005) Tool-supported verification of product line requirements. Autom Softw Eng J 12:447–465
43. Niu N, Savolainen J, Yu Y (2010) Variability modeling for product line viewpoints integration. In: COMPSAC, Seoul, South Korea, pp 337–346
44. Dolstra E, Florijn G, Visser E (2003) Timeline variability: the variability of binding time of variation points. In: van Gurp J, Bosch J (eds) Workshop on software variability management (SVM'03), Groningen, pp 119–122
45. Rosenmüller M, Siegmund N, Apel S, Saake G (2011) Flexible feature binding in software product lines. Autom Softw Eng 18(2):163–197

# Chapter 4
# *DUFICE*: Guidelines for a Lightweight Management of Requirements Knowledge

W. Maalej and A.K. Thurimella

**Abstract** Working with requirements is a knowledge-intensive task. During the elicitation, comprehension, or management of requirements, practitioners often consume and produce additional information such as domain knowledge, rationale, requirements dependencies, "who knows what", or how-to's. However, current requirements engineering processes and tools lack a systematic support for the management of knowledge about requirements. This makes it difficult for practitioners to capture and share such knowledge.

This chapter summarises our experience on implementing a lightweight, pragmatic approach to capture and share requirements knowledge. We recommend practitioners to *D*raw a knowledge landscape, *U*se lightweight tools, *F*ollow a simple iterative process, *I*nteract with external communities, *C*apture tacit knowledge, and *E*stablish a knowledge culture. We introduce these guidelines, report on motivating examples, and discuss how they can be applied successfully in practice.

## 4.1 Introduction

Engineering, managing, or implementing requirements are knowledge-intensive activities, which need diverse information from diverse sources. For example, requirement analysts need information on the application domain for defining correct and complete requirements. Change requesters need information on the

W. Maalej (✉)
University of Hamburg, Department of Informatics/MOBIS, Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
e-mail: maalej@informatik.uni-hamburg.de

A.K. Thurimella
Harman Becker Automotive Systems GmbH, Moosacher Str. 48, 80809, Munich, Germany
e-mail: anil.thurimella@gmail.com

**Table 4.1** Organisations from which we derived our guidelines

| Organisation | Size | Short description | Observation period |
|---|---|---|---|
| LC1 | >100,000 employees | Global technology provider designing transportation, telecom, medical, and energy systems | 4 years |
| LC2 | >100,000 employees | Global outsourcing and IT service provider, in the banking, health care, and insurance domains | 1.5 years |
| MC1 | 5,000–10,000 employees | Global company developing infotainment systems in particular for the automotive industry | 3.5 years |
| MC2 | 5,000–10,000 employees | European hi-tech company in the field of telecommunication, radars, and security | 2 years |
| SC1 | 50–100 employees | Small company designing and customising content management and internet marketing solutions | 4 years |
| SC2 | 50–100 people | Project-based consortium of 10 European organisations, for a large software engineering project | 3 years |

processes followed for tracking the status of their requests. Architects need information on the technologies used in order to assess the requirements feasibility [1].

Specific aspects of knowledge management such as documenting *decisions* [2] or collecting *how-to*'s [3] are partly implemented by requirements tools and used in practice. However, conventional requirements engineering (RE) processes focus on eliciting, analysing, and validating requirements but rarely on how to gather and share supplementary information [2, 4]. Organisations often set a low priority for knowledge management due to its long-term and hard-to-measure return on investment [5]. Available resources are rather spent on engineering tasks.

This chapter presents practical guidelines for a lightweight management of requirements knowledge. The guidelines are derived from our experience while working with six organisations, as well as the discussions, which took place during the last four international workshops on Managing Requirements Knowledge (MaRK) [6]. Table 4.1 briefly introduces the referred organisations: two large companies (LC), two medium-sized companies (MC), one small company, and one small consortium (SC). At least one of the authors has worked with these organisations for one or more years, either as an employee, as a consultant, or as a collaboration partner in more than two projects. During this work, we were involved in diverse RE activities and were able to observe the processes, practices, success, and failure stories. We did not intend to scientifically study the knowledge management practices in these organisations from the beginning. Instead, we synthesised in a post-mortem manner our observations into a list of guidelines, which we discussed and refined during the MARK workshop series [6].

The goal of synthesising the guidelines is twofold. First, we aim at helping practitioners to implement means for managing requirements knowledge with a minimal effort. Second, we want to increase the awareness for the importance of this issue. Our long-term vision is to build a community of practice that collects and exchanges guidelines, best practices, and tools for a lightweight management of requirements knowledge. We do *not* believe that knowledge management in software projects should become a goal on its own. Instead, it should be regarded as a means for coping with the complexity of requirements engineering work.

We structure our recommendations along six guidelines, which we call ***DUFICE***: ***D***raw a knowledge landscape (Sect. 4.2), ***U***se lightweight tools (Sect. 4.3), ***F***ollow an iterative process, (Sect. 4.4), ***I***nteract with external communities (Sect. 4.5), ***C***apture tacit knowledge (Sect. 4.6), and ***E***stablish a knowledge culture (Sect. 4.7). In each of these sections, we introduce the guideline and its rationale. Then, we give recommendations how to implement the guideline in practice with examples from our experience. We then conclude the chapter and discuss next steps (Sect. 4.8).

## 4.2 Draw a Knowledge Landscape

A knowledge landscape is an evolving collection of information types, which are needed for a particular activity. Organisations should draw their knowledge landscapes on requirements and make them explicit. Drawing includes three actions: defining, structuring, and publishing. Defining means identifying and labelling artefacts types, which contain knowledge relevant for working with requirements. Structuring means developing a simple taxonomy for the artefacts, which helps practitioners navigate and remember the landscape. Publishing means listing the artefact types in a visible, accessible place, for example, in a poster near the coffee machine or a frequently accessed intranet page.

Drawing a landscape raises awareness and creates a common understanding of requirements knowledge. It also reminds where to access relevant information. A landscape acts as an index or a navigation map for requirements knowledge and enables an informal evaluation of “what we have and what not”. For example, MC2 introduced a new requirements engineering tool, starting with similar pilot projects in three departments. After several months, the team in one department placed a poster in the kitchen with relevant information for requirements engineering and where to find it (in which tool). After 1 year, we observed that the members of this team were overall more satisfied with their requirements engineering than the other teams. The new tool was used more frequently and consistently. We also observed that this team consistently documented more knowledge types than in the other two departments.

### *4.2.1 How in Practice*

Figure 4.1 shows an example of a knowledge landscape, organising artefacts along domain knowledge, engineering, management, collaboration, and how-to knowledge.

*Domain knowledge* refers to common knowledge in a particular area or a specialised discipline. This is usually the domain, for which a system should be developed. Domain knowledge includes a vocabulary, standards used in the domain

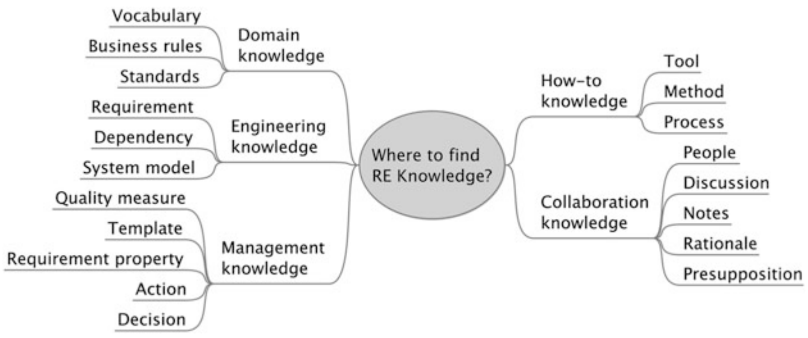


**Fig. 4.1** Landscape of requirements knowledge: an illustrative example

(e.g. telecommunication or banking standards), and business rules (i.e. domain constraints to be satisfied when designing or using the system). Vocabularies (also called glossaries) provide common understanding and definitions of the terms used in the domain. Apart from the domain terminology, a vocabulary can also define process- and project-specific terms or keywords from the solution domain, which help to understand the requirements. The lack of a common vocabulary is often a source for ambiguity [7]. For example, SC2 involved stakeholders from different backgrounds including programming, system administration, scientific computing, business consulting, and information management. For over 1 year, the stakeholders had different understandings of simple terms like “software engineering” or “scenarios”. This made the discussions and decision-making very difficult. Only clearly defining these terms in a shared Wiki with examples helped getting a more focused discussion, and later a common opinion as well as a detailed specification and implementation of the requirements.

*Engineering knowledge* includes requirements “content”, such as the requirements specifications, dependencies between requirements, as well as other artefacts needed to understand and use the requirements such as models, test cases, or system architecture. Also informal notes and personal comments [8] typically annotating artefacts such as models, requirements, or plans might include useful information. For example, in SC1 key developers often wrote source code comment such as “This feature should be supported only for the platform XX. It should be given highest priority for release YY. See customer request ZZ”. This information was useful for requirements planning and prioritisation.

*Management knowledge* includes quality measures, templates, and properties of requirements such as status and priority. Moreover, issues, decisions, and actions are part of this knowledge. During a requirements review, open issues, decisions, and action items on requirements might be identified and discussed. High-level decisions include strategic decisions on requirements, process decisions, or decisions in product management meetings and change control board meetings.

*Collaboration knowledge* includes information about people, their interactions, discussions, argumentation chains, and presuppositions. Knowing “who knows what” is an important piece of requirements knowledge, because people might have different experiences about different problems. In LC1 and LC2, we observed that getting to the right person who can answer a question to a requirement often requires asking 4–5 people. Discussions include information exchanged or shared between different stakeholders on various problems related to requirements. Discussion might also include requirements rationale or the reasoning behind the requirements, a crucial piece of knowledge to understand and implement requirements especially when people leave the projects. Finally, presuppositions are assumptions for realising a requirement. The lack of common understanding of presuppositions often leads to misunderstanding of requirements [9].

*How-to knowledge* includes information on tools, methods, and processes to be used for a particular situation while engineering and managing requirements. Organisation or vendor guidelines include information on how to perform requirements engineering activities or how to use a tool. We observed that such how-to's were collected and maintained both by a central department and the separate teams in LC1, MC2, and SC1. These how-to's were especially useful for us when joining these organisations, since tools and processes used were internally customised. Also the Capability Maturity Model Integration (CMMI) encourages guidelines [10].

The knowledge landscape is an evolving document, which depends on the project and organisation context. For example, specific projects might give a particular importance to defect reports, management reports, or other artefacts required by the customer (such as a specific database to be maintained). In LC2, large projects specified in the project contract what requirements-related information should be maintained where. Overall, the objective of such a landscape is to collect, visualise, and remember where requirements-related information can and should be found, rather than to define a requirements engineering taxonomy.

## 4.3 Use Lightweight Tools

We recommend using lightweight tools to access and share knowledge. A lightweight tool includes a *minimal* set of features, that is, searching, navigating, capturing, and sharing information. Lightweight tools should be *integrated* into main working tools, such as requirements management tools, bug trackers, email clients, and development environments. They should run in the background as much as possible and avoid interrupting the workflow of stakeholders.

Knowledge management tools should be easy to learn and to use. Complex tools dedicated for knowledge management with numerous, partly unneeded features enforce new formal processes and roles. This would increase the overhead and decrease the acceptance by stakeholders. Moreover, a tight integration decreases fragmentation of information and workflows. This reduces the barriers to, for example, annotate a requirement with a "thought" or link two requirements. We observed in LC1 and MC2 a large resistance against tools that were introduced as "separate islands" to manage requirements and related information such as release plans and release notes. Teams often tried to create workarounds to manage requirements knowledge such as action items or rationale in tools they are used to.

### *4.3.1 How in Practice*

Different companies use different requirements management tools. For example, medium and large companies such as LC1, MC2, and MC1 use commercial tools to support requirements management, whereas small companies such as SC2 and SC1 use Wikis and web-based tools. Therefore, it is difficult to give "one-size-fits-all" recommendation. However, our experience showed that the following recommendations worked well for most types of projects:

- Use Wikis and maintain its structure carefully [11]. Wikis were used in all observed organisations either officially or unofficially. Wikis enable an iterative, collective distillation of knowledge [12]. Their entry barriers are low, since (a) their syntax is simple, (b) they run in a web browser, and (c) they are informal without any imposed structure. When a concept gets mature (e.g. a decision with arguments), a new page can be easily created and linked from other pages. Moreover, formal requirements as well as other system models can be linked to Wiki pages, since most tools have a URIs for their internal resources.

  However, due to their flat and changing structures, Wikis can become overloaded and information not easily accessible. We observed, for example, in SC1 and SC2, that, when knowledge grows and the Wiki becomes bigger and bigger, people start creating subpages and move links which were previously on the start page. After a while, a Wiki becomes a deep tree, and the only way to find information is to search for keywords. New employees joining projects often compared the Wiki start pages in these organisations with "an information jungle". For this reason, it is important to define a main structure for the Wiki (e.g. as reported in [11]) and maintain it regularly and carefully.
- Use integrated search engines like Lucene [13] or Teamweaver [14]. Such engines maintain an index of different types of artefacts, including emails, requirements, requests, Wiki pages, and code. They enable stakeholders to search for information and navigate to it from a single place. These engines typically offer simple, keyword, Google-like user interfaces. We observed in all organisations that stakeholders frequently used keyword search to access requirements-related information. When the tools did not index this information, stakeholders used other tools or global search engines for the file system. Middleton and Baeza-Yates compare several open-source search engines in [15].
- Consider using intranet tools such as news, forums, or polls to publish release notes or a survey on a relevant technology. We observed that MC2 and SC1 heavily used such tools to create awareness. Employees reported to use these tools during "their low-concentration times and still learn something useful".

- Annotation tools enable the addition of semi-structured metadata. This allows for an evolutionary structuring of knowledge and advance searches to satisfy complex information needs. SC2 successfully used Semantic MediaWiki, which allows annotating Wiki pages with "semantic keywords". We observed that stakeholders who used this feature, for example, to generate all requirements pages relevant to a specific component or a release, reported less problems with coping with the complexity of the requirements.
- Integrate knowledge management tools into working tools to reduce duplication, increase reuse, and reduce capturing effort. Many open-source communities offer plug-ins for commercial tools and vice versa. For example, Eclipse and OpenOffice provide plug-ins for Enterprise Architect and vice versa. It might be a good idea to integrate two commercial tools by using a third open-source tool supported by them.
- Design lightweight tools to run as a background application rather than a main tool. When it starts, it should neither display an application window nor include an entry in the tool list. Instead, the system tray can be used as entry point, similar to wireless network tools, backup managers, or system clock tools.
- Main functionality such as searching or annotating should be accessible via a shortcut at any point in time in the workflow. This reduces the intrusiveness of such tools and enables the user to concentrate on their main working tools.
- Gadgets and widgets might inspire the interface design of lightweight knowledge management tools [16]. Views might have a uniform, distinguishable, semi-transparent background, which makes the views easy to identify. This also emphasises that such tools run "in parallel" to main applications.

Finally, intelligent tools such as recommendation systems [17] (e.g. "other stakeholders also used this requirement") as well as mining tools can extract knowledge by systematically analysing artefacts and stakeholders behaviours. For instance, Lutz and Rouquette [18] reported on their experience with continuously developing and maintaining defect reports in the Jet Propulsion Lab. The defect reports are extracted from sources such as integration and testing artefacts. The defect reports helped to identify requirements rationales, new requirements, unexpected requirements dependencies, and misunderstood requirements. Juergens et al. also describe [19] a similar experience of applying clone detection techniques on requirements, which leads to useful information on requirements quality and dependencies. Lim et al. [20] suggested to use social networking and crowdsourcing techniques to identify and prioritise stakeholders. Chapter 14 [21] and Chap. 15 [22] discuss intelligent tools for requirements engineering in more detail.

## 4.4 Follow an Iterative Process

We recommend organisations to follow a simple, iterative process to manage requirements knowledge. Figure 4.2 depicts such a process as the "dual problem of accessing (searching for and identifying) and sharing (capturing and transferring)

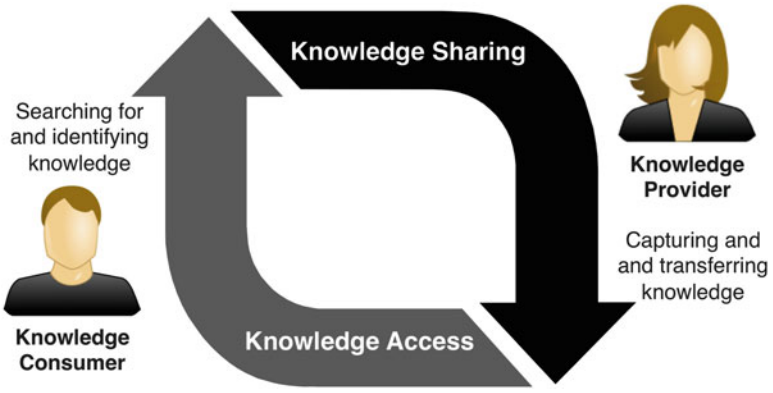

Knowledge Sharing
Searching for and identifying knowledge
Knowledge Provider
Capturing and and transferring knowledge
Knowledge Consumer
Knowledge Access

**Fig. 4.2** An iterative knowledge management process (Adopted from [23])

knowledge across organisational subunits" [23]. This process mediates between two roles: the *knowledge consumer*, who benefits in a specific situation, and the *knowledge provider*, who contributes to the teams' experience. A requirements engineering stakeholder could be a knowledge provider, a knowledge consumer, or both.

The process should be iterative because knowledge (both tacit and explicit) is continuously evolving. New experiences, people joining, and milestones achieved change the level of knowledge. The process should reflect the incomplete nature of knowledge and enable an open world assumption [24]. That is, in addition to true or false statements, the process should also explicitly allow for unknown or not yet known. Moreover, organisations should minimise the process overhead of knowledge management. Complex, formal processes would increase resistance of practitioners as we observed in LC1 and LC2, in particular since the primary work goal is to engineer requirements rather than to manage knowledge. For example, LC2 invested a large amount of effort (several man years) to develop an in-house knowledge management system for project and requirements knowledge. This system was, however, tightly coupled to a formal elaborate process. While rolling out the system into the development teams, the main feedback was about the lack of flexibility and the complexity of the process. While the teams agreed that they would like to have a systematic way to manage their requirements knowledge, they claimed that it is unrealistic to adopt formal processes given the high overhead and the time pressure to deliver work products.

### *4.4.1 How in Practice*

Knowledge should be captured, transferred, and accessed *incrementally*. This is similar to agile and lean methodologies [25] (e.g. SCRUM or Crystal Clear).

For example, we observed in MC1 that a typical project involves domain experts and a requirements manager. Both reviewed requirements over few months. The review was iterative and included regular status meetings, which covered experiences, problems encountered, and possible solution strategies. Requirements themselves and other information were documented and refined iteratively.

Further, knowledge management should be *tightly integrated* into other processes. Instead of "specifying knowledge", the goal of documenting requirements knowledge should be to provide required information to relevant stakeholders when needed as early and efficiently as possible. For example, Gallaro-Valencia described a practice where requirements knowledge is captured within various forms, for example, embedded in user stories, acceptance tests, and personal notes [26].

In addition to drawing the knowledge landscape, an iterative knowledge management process can be supported by *defining templates* for capturing the various artefacts. For example, action items obtained from a requirements review can be presented using the following template, as we observed in MC1:

Action: Requirement XX should be clarified with N. David
Priority: high
Responsible: B. Bob
Deadline: 25 March 2011

Finally, we recommend identifying users for the specific knowledge artefacts. This allows implementing a *need-driven process*. For example, at MC1 information quality was adjusted depending on the users. Reports with annotated requirements to be communicated to customers or managers had a higher quality and were more formal than information to be used within a team.

## 4.5 Interact with External Communities

We recommend a strong interaction with external communities to exchange knowledge and solve problems that occur in practice. External communities are open-source communities, requirements working groups, universities, research institutes, online communities, and other organisations.

Interacting with external community enables knowledge sharing on requirements engineering practices and simplifies the implementation of particular steps. For example, we observed twice that employees of MC2 and MC1 used web discussion forums to learn about experiences in using database management systems for projects with a very large number of requirements (e.g. > one million). Input from external communities opens new horizons, brings new ideas into the organisation, and allows for being up to date with the state of the art.

### 4.5.1 How in Practice

We observed that the small organisations SC1 and SC2 benefited much more from interacting with external communities than large organisation such as LC1 and LC2. Employees from SC1 and SC2 use to answer their questions on tool usage and how-to's in online communities. They also tried to reuse complete sets of requirements from open-source projects whenever applicable. Overall, stakeholders typically give a lower priority to such interaction due to time pressure and the focus on "everyday's business". Nevertheless, we recommend implementing at least one of the following measures:

- Learn from open source. That is, use public "open-source" community knowledge. In many cases, features that you are planning to develop have already been implemented by open-source or research communities at least prototypically. Even most innovative features with complex and unique requirements (as we observed in MC2 for military projects) include rather *common* sub-features such as manipulating data or real-time communication. Looking at how open-source projects solved particular problems and implemented particular requirements would help to reduce risks and save useful resources.
- Join social networks, including user forums and social networks such as RE-newsletter [27], Re-online [28], or DXL forum [29]. Post questions on these forums and continuously browse posts (e.g. once a week). Advise other colleague to join these networks. Examples of RE communities and social networks are Requirements Working Group, Seilevel message board [30], RE-Wissen [31], LinkedIn Group IBM Rational DOORS, RESG groups, and other groups at LinkedIn, Facebook, Yahoo, Planet, and Twitter. For example, we observed that requirements engineers working with DOORS in LC1, LC2, MC2, and MC1 have been benefiting from the DOORS eXtension Language (DXL) forum within IBM developer works. These companies extend the tool using the DXL as other companies do. Working with DXL involves dealing with undocumented APIs and limited development environment (e.g. no debugger). Several hundreds of issues have been solved in the DXL forum. Recently Q&A websites such as Stack Overflow emerged as a huge and very efficient knowledge source for software engineering projects, with questions answered after 11 min on overage [32]. These pages mostly focus on low-level development knowledge. However, they also contain a significant amount of requirements knowledge such as framework functionality, how-to's, domain-specific languages, or interoperability requirements.
- Plan sending employees to events at least once each 6 months. Events include tool conferences (e.g. Atlassian, Sparx Systems, IBM user conferences), as well as scientific conferences such as IEEE RE or REFSQ. In addition local events such as round discussion tables and democamps represent a low-cost informal initiative. Examples include the GI requirements engineering working groups, the Eclipse Democamp, and Zurich Knowledge Café. If you are unable to find

local events, consider organising one. Typically, all what you need for this is a meeting room, drinks, some snacks, and access to the "right" mailing list or a forum to announce the event.

## 4.6 Capture Tacit Knowledge

Knowledge such as decision rationale and presuppositions is often kept implicit in the head of stakeholders [4, 9]. If captured, tacit knowledge can be shared and reused. For example, captured presuppositions can be reused by future projects in the same domain. This is particularly useful for outsourcing organisations focussing on specific industries, as we observed for LC2.

Moreover, captured tacit knowledge can be referenced and allocated when people leave [33]. New stakeholders joining the project can better understand decisions on what the system does or how it does it. Therefore, we recommend capturing tacit knowledge – as much as possible. Previous contributions, for example, Ma et al. [34] and Dutoit et al. [4], discussed means to capture such tacit knowledge. Chapter 2 of this book [35] introduces a formal framework on how to capture and reason about tacit knowledge.

### *4.6.1 How in Practice*

Tacit knowledge can be documented in plain texts, in structured natural language, or in forms. Several researchers suggested using model-based approaches such as Questions, Options, and Criteria (QOC) [36], Issue-based Information Systems (IBIS) [37], and Decision Representation Language (DRL) [34] to capture decisions. These models are similar as they all define detailed information models to capture and reuse issues, alternatives, decisions, and rationale. For a detailed comparison, we refer the reader to [4].

The main problem of capturing tacit knowledge is the *justification of the effort needed*. We recommend therefore that the amount of effort and level of granularity for documenting decisions should depend on the context, the priority, and the complexity of the decision-making problem. Decisions on fine-grained requirements are locally made within small teams. In case of high-level issues, decision-making involves several key stakeholders and is performed over a long period of time. Therefore, we recommend a *non-monolithic approach*. Table 4.2 summarises four approaches and our recommendations on their applications.

The simplest approach is to document decisions in unstructured natural language. This is only recommended to document low-level decisions. IBIS is a simple approach, which requires identifying issues, proposals, argument, and decision. We recommend IBIS for documenting decisions if different proposals should

**Table 4.2** Recommendations on approaches to document decisions

| Approach | When to use | Decision process |
|---|---|---|
| Unstructured | Recommended only for documenting personal low priority decisions | No explicit process required |
| Simple structure | Recommended. A "tag-based" template should be used to document issues and decisions | No explicit process required |
| IBIS | Recommended when various proposals are evaluated | A decision-making process should be defined |
| QOC | Recommended when proposals are evaluated against goals and non-functional requirements | A decision-making process should be defined |

be evaluated. The advantage of documenting alternatives is that at long term, the alternatives could be reconsidered for future decisions. Furthermore, similar issues could appear and knowledge captured could be reused [38].

QOC differs from IBIS at the process level. While IBIS captures arguments as they occur, QOC reflects the current state of the discussion [4]. QOC has been used by scientific community to support various activities in requirements engineering such as decisions in product management meetings [39], rich traceability [40], and variability management [41]. QOC is recommended when a rich argumentation is required based on selection criteria.

QOC or IBIS can be combined with a formal decision-making process. For example, a decision-making process based on QOC includes the collection and prioritisation of issues, creation of proposals, identification of selection criteria, collection of assessments, and formal meetings (e.g. weekly) to take decisions.

With the increase of stakeholder mobility and project distribution, we observed in all organisations, in particular, in LC1, LC2, and MC1 that stakeholders exchange *emails* to discuss issues and therefore knowledge related to requirements. These emails typically externalise a significant amount of tacit knowledge. Often decisions are taken within email threads. This trend has large potentials in "digitalising" tacit knowledge but also has several disadvantages. Information in discussions often remains private, not accessible to stakeholders, and not linked to respective requirements. If stakeholders need to refer to the rationale (e.g. in case of similar issues), they have to search emails, which might have been deleted.

We recommend two lightweight alternatives for using "pure" emails to capture tacit knowledge. The first option is to capture the issues and alternatives in a Wiki or a bug tracker and to use the automatic notification features of these systems to send emails with links to the respective pages. The second option is to annotate the sentences in the emails with *tags* such as <issue>, <option>, and <decision>. Pieces of information can then be extracted from the emails and shared in a common repository.

Finally, since tacit knowledge is subjective [33], it is relevant to know who had this tacit knowledge and in which time context. We *recommend to record history*, for example, by using a version control system or simple annotation with timestamps. For example, a team in MC1 maintains requirements within different excel sheets. Each sheet is owned by an expert and is shared in a public folder. One of the experts suddenly discovers a new requirement in his sheet. If no history is recorded, the expert will encounter major difficulties for requesting clarifications on this requirement. Recording history helps solving such issues. If the tool used does not record history or if information is entered for another person, the name of the person and the date should be documented, as in the following example:

Bob, 12.01.2011: Requirement R should be also supported on LINUX.

## 4.7 Establish a Knowledge Culture

Organisation should harvest a knowledge culture by convincing practitioners to value the managing and sharing of requirements-related information. A knowledge culture encourages collecting best practices that cover tools, approaches, and processes. Such a culture also accepts incomplete, evolving knowledge and encourages collective contributions.

Knowledge management is a cross-cutting concern. It is affected by how people collaborate, work, organise themselves, and put extra effort to improve the productivity and to mitigate risks for organisations. Therefore, it can be ensured only if a culture is established within the organisation. If stakeholders get convinced with the value of this activity, they will practice it and convince others as well.

### *4.7.1 How in Practice*

This guideline is probably the hardest to implement. We observed in all six organisations, in particular in the large- and medium-sized, that it is very difficult to change the culture of the teams. This process is long lasting and requires a lot of convincing argumentation and personal experiences. In several cases, we heard “why should we change a running system?” when we, for example, suggested a particular practice to improve managing the requirements knowledge. We therefore recommend considering at least one of the following measures, whenever applicable:

- Develop *incentive* methodologies to encourage quality contributions. For example, we observed in MC2 that publishing best practices across the organisation including the names of the authors motivates contributors to ensure a high quality and creates a “healthy” competition between experts. We also observed that using requirements knowledge explicitly in requirements training programmes and citing best practices within newsletters represent valuable incentives for contributors. A good incentive is to quantify the produced and consumed knowledge, for example, with a simple metric on the number of accessed/viewed rationale, how-to’s, or presuppositions.

- Identify experts, champions, and mentors for various aspects of requirements engineering and assign them to project newcomers. This ensures informal face-to-face knowledge sharing and a feedback culture.
- Create discussion groups and workshops on sensitive issues of knowledge management. These are mainly “knowledge is power”, “not my job”, and “I don’t have time”. These issues present typical blockers for knowledge sharing in organisations. We observed that knowledge providers often assume that organisation knowledge interests are conflicting with personal knowledge interests. This theory should be discussed and arguments falsifying it should be collected and shared.
- Create a feedback culture, enabling learning from mistakes and learning by doing. This can be done, for example, by conducting brief retrospective meetings and identifying, discussing, and if possible documenting lessons learned from each project, including what worked well and what not. This is particularly valuable, for example, when new tools are piloted, a prioritisation strategy is tested, or new means for capturing rationale are used.
- If possible, identify and quantify the return on investment of lightweight knowledge sharing. For example, estimate time saved in a requirements engineering activity because information is reused easily from the Wiki, and communicate.
- Maintain your knowledge landscape *continuously*, and encourage continuous small contributions.

## 4.8 Conclusion

When applied to requirements engineering, formal, process-based knowledge management tools and practices introduce an additional overhead, which is difficult to justify to the management and to the stakeholders. The return on investment is in the long term and difficult to measure. However, systematically managing requirements knowledge allows for saving time to find information and share experiences. We suggest building a community of practice on managing requirements knowledge. We summarised six major guidelines inspired from our experience with six different organisations.

We argued how drawing a knowledge landscape helps visualising the different types of artefacts and establishing a common understanding among stakeholders on where to access and share which information. Using lightweight tools reduces the workflow interruptions and information fragmentation and therefore the overhead of knowledge management. Following an iterative process supports stakeholders dealing with the incomplete evolving nature of requirements and reducing the contribution barriers. Interacting with external communities ensures the incorporation of the state of the art and exchange of experiences. Capturing tacit knowledge allows for reusing it and for being independent from individuals. Finally, establishing a knowledge culture enables to deal with the social issues of knowledge management.

We created an online platform to share such guidelines on www1.cs.tum.edu/mark/community. We will further collect experiences and case studies and link this community to the scientific workshop International Workshop on Managing Requirements Knowledge. One successful example is *e20cases.org,* which does the same for social software in professional organisations.

**Acknowledgments** We thank Barbara Paech, Dennis Pagano, and Yang Li for the constructive and detailed reviews. We are also grateful to all MaRK'08–'10 participants and to our colleagues from the referred companies for the fruitful discussions and the feedback on early versions of this chapter. This work has been supported in part by the FastFix project and TEAM project, which are funded by the 6th and 7th Framework Programmes of the European Commission, Grant Agreement No. FP6-35111 and FP7-258109.

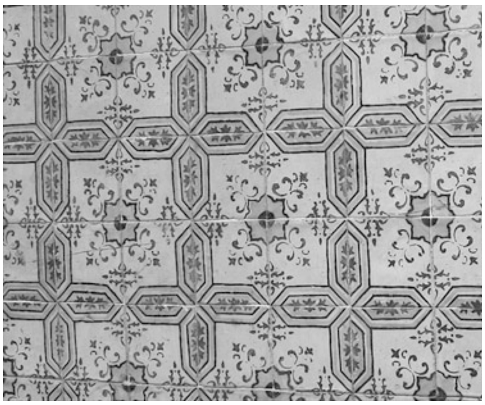
## References

1. Finkelstein A, Kramer J, Nuseibeh B, Finkelstein L, Goedicke M (1992) Viewpoints: a framework for integrating multiple perspectives in system development. Int J Softw Eng Knowl Eng 2–1:31–57
2. Alenljung B, Persson A (2005) Decision-making activities in the requirements engineering decision processes: a case study. In: ISD 2005, Karlstad, pp 707–718
3. Mavin A, Wilkinson P, Harwood A, Novak M (2009) Easy approach to requirements syntax (EARS). In: Requirements engineering conference, 2009, RE '09, 17th IEEE international, Atlanta, USA, pp 317–322
4. Dutoit AH, McCall R, Mistrik I, Paech B (2006) Rationale management in software engineering: concepts and techniques. In: Rationale management in software engineering. Springer, Berlin, pp 1–48
5. Rus I, Lindvall M, Sinha SS (2001) Knowledge management in software engineering a state-of-the-art-report. Fraunhofer Center for Experimental Software Engineering Maryland and the University of Maryland for Data and Analysis Center for Software, Department of Defence, Maryland
6. Maalej W, Thurimella AK (2010) Managing requirements knowledge. International workshop on 2008–2010, IEEE. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5611788
7. Kiyavitskaya N, Zeni N, Mich L, Berry DM (2008) Requirements for tools for ambiguity identification and measurement in natural language requirements specifications. Requir Eng 13 (3):207–239
8. IEEE Standard 1063–2001 (2001) IEEE standard for software user documentation, ISBN: 0-7381-3099-0
9. Ma L, Nuseibeh B, Piwek P, De Roeck A, Willis A (2009) On Presuppositions in requirements. In: Proceedings of 2nd international workshop on managing requirements knowledge, MaRK'09 IEEE, Atlanta, USA, pp 27–31
10. CMMI DEV, CMMI for development v1.3 (2010) http://www.sei.cmu.edu/reports/10tr033.pdf
11. Decker B, Ras E, Rech J, Jaubert P, Rieth R (2007) Wiki-based stakeholder participation in requirements engineering. In: IEEE software, vol 24-2, pp 28–35
12. Maalej W, Panagiotou D, Happel HJ (2008) Towards effective management of software knowledge exploiting the semantic wiki paradigm. In: Software Engineering, GI- LNI, Munich, Germany, vol 121, pp 183–197
13. Lucene Homepage (2012) http://jakarta.apache.org/lucene/

14. Teamweaver Homepage (2012) http://www.teamweaver.org/
15. Middleton C, Baeza-Yates R (2007) A comparison of open source search engines, TR. Universitat Pompeu Fabra, Department of Technologies. http://wrg.upf.edu/WRG/dctos/Middleton-Baeza.pdf
16. Udell S (2009) Pro web gadgets for mobile and desktop. Apress, Berkely
17. Maalej W, Thurimella AK (2009) Towards a research agenda for recommendation systems in requirements engineering. In: MaRK'09, IEEE, Atlanta, USA, pp 32–39
18. Lutz R, Rouquette N (2010) Using defect reports to build knowledge in product lines. In: Proceedings of 3rd international workshop on managing requirements knowledge, IEEE, Sydney, Australia, pp 12–21
19. Juergens E, Deissenboeck F, Feilkas M, Hummel B et al (2010) Can clone detection support quality assessments of requirements specifications? In: Proceedings of ICSE'10, ACM, Cape Town, South Africa, pp 79–88
20. Lim SL, Damian D, Ishikawa F, Finkelstein A (2013) Using web 2.0 for stakeholder analysis: StakeSource and its application in ten industrial projects. In: Managing Requirements Knowledge, Springer
21. Felfernig A, Ninaus G, Grabner H, Reinfrank F, Weninger L, Pagano D, Maalej W (2012) An overview of recommender systems in requirements engineering. In: Managing Requirements Knowledge, Springer
22. Knauss E, Meyer S (2013) Experience-based requirements engineering tools. In: Managing Requirements Knowledge, Springer
23. Hansen MT (1999) The search-transfer problem: the role of weak ties in sharing knowledge across organization subunits. Adm Sci Quart 44(1):82–111
24. Drummond N, Shearer R (2006) The Open World Assumption – or Sometimes its nice to know what we don't know, The University of Manchester. http://www.cs.man.ac.uk/~drummond/presentations/OWA.pdf
25. Poppendieck T, Poppendieck M (2003) Lean software development: an agile toolkit. Addison-Wesley Professional, Boston
26. Gallaro-Valencia R, Sim S (2009) Continuous and collaborative validation: field study of requirements knowledge in agile. In: Proceedings MaRK'09, Atlanta, USA
27. Newsletter der Fachgruppe Requirements Engineering (2012) https://mail.gi-ev.de/mailman/listinfo/re-newsletter
28. Requirements Engineering Online Discussion Forum (2012) http://discuss.it.uts.edu.au/mailman/listinfo/re-online
29. DXL forum (2012) http://www.ibm.com/developerworks/forums
30. Seilevel message board (2012) www.seilevel.com/messageboard
31. RE Wissen (2012) http://www.re-wissen.de/
32. Mamykina, L, Manoim B, Mittal M, Hripcsak G, Hartmann B (2011) Design lessons from the fastest Q&A site in the west. In: Proceedings of the SIGCHI conference on human factors in computing systems CHI '11, ACM, Vancouver, Canada, pp 2857–2866
33. Gacitua R, Ma L, Nuseibeh B, Piwek P, de Roeck AN, Rouncefield M, Sawyer P, Willis A, Yang H (2009) Making tacit requirements explicit. In: Proceedings MaRK'09, Atlanta, USA, pp 40–44
34. Lee J (1991) Extending the Potts and Bruns model for recording design rationale. In: Proceedings of the 13th international conference on software engineering (ICSE"13), IEEE Computer Society Press, Los Alamitos, pp 114–125
35. Gervasi V, Gacitua R, Rouncefield M, Sawyer P, Kof L, Ma L, Piwek P, Roeck A, Willis A, Yang H, Nuseibeh B (2013) Unpacking tacit knowledge for requirements engineering. In: Managing Requirements Knowledge, Springer
36. MacLean A, Young RM, Bellotti VME, Moran TP (1991) Questions, options, and criteria: elements of design space analysis. Hum Comput Interact 6(3–4):201–250
37. Kunz W, Rittel H (1970) Issues as elements of information systems, vol 131. University of California at Berkeley, Institute of Urban and Regional Development, Berkeley

38. Thurimella AK, Bruegge B (2007) Evolution in product line requirements engineering: a rationale management approach. In: RE 07, New Delhi, pp 254–257
39. Dutoit AH (1996) Rationale management in requirements engineering. Ph.D. dissertation, Carnegie Mellon University
40. Hull E, Jackson K, Dick J (2004) Requirements engineering. Springer, London
41. Thurimella AK, Bruegge B, Creighton O (2008) Identifying and exploiting the similarities between rationale management and variability management. In: Proceedings 12th international software product line conference (SPLC 2008), Limerick, pp 99–108

# Part II
# Representing Requirements Knowledge for Reuse



"Reuse Reduce RECYCLE!
Save the world, no need to be superman.
I bring what I can to the table."
— Margaret Irvine

# Chapter 5
# Constructing and Using Software Requirement Patterns

**X. Franch, C. Quer, S. Renault, C. Guerlain, and C. Palomares**

**Abstract** Software requirement reuse strategies are necessary to capitalize and reuse knowledge in the requirement engineering phase. The PABRE framework is designed to support requirement reuse through the use of software requirement patterns. It consists of a meta-model that describes the main concepts around the notion of pattern, a method to conduct the elicitation and documentation processes, a catalogue of patterns, and a tool that supports the catalogue's management and use. In this chapter all these elements are presented in detail making emphasis on the construction, use and evolution of software requirement patterns. Furthermore, the chapter includes the construction of a catalogue of nontechnical software requirement patterns for illustration purposes.

## 5.1 Introduction

Requirement elicitation is the process of acquiring system requirements from system stakeholders. The quality of this process is critical to make information technology (IT) projects a success.

When a company runs many elicitation processes over time, it is often the case that a significant proportion of requirements is recurrent and belongs to a relatively small number of categories, especially in the case of nonfunctional [1] and non-technical [2] requirements. Capitalizing on knowledge acquired in previous projects seems in this way an adequate strategy to improve the quality of requirements and then increase the changes of project success, as well as to increase

X. Franch (✉) • C. Quer • C. Palomares
Universitat Politècnica de Catalunya, Barcelona, Spain
e-mail: franch@essi.upc.edu; cquer@essi.upc.edu; cpalomares@essi.upc.edu

S. Renault • C. Guerlain
Centre de Recherche Public Henri Tudor, Luxembourg, Luxembourg
e-mail: samuel.renault@tudor.lu; cindy.guerlain@tudor.lu

the efficiency of the requirement elicitation process. This chapter proposes an application of the concept of *software requirement pattern* as a means to capture and capitalize requirement knowledge in the context of IT systems and services procurement projects. Specifically it presents this concept in the mark of the PABRE framework making emphasis on the construction, use, and evolution of software requirement patterns.

The chapter is structured as follows. Section 5.2 presents the context of our work. Then in Sect. 5.3, we summarize the state of the art on software requirement patterns. We present the main elements of our PABRE approach in Sect. 5.4, and in Sect. 5.5 we describe the patterns and catalogue structure as well as their construction process. In Sect. 5.6, we detail our experience in building a catalogue of patterns for nontechnical requirements. Finally, Sect. 5.7 presents some conclusions and future work.

## 5.2 Context

The work presented in this chapter stems from the needs of the Public Research Centre Henri Tudor (TUDOR) at Luxembourg when conducting IT procurement projects over time. Since 2004, TUDOR works in collaboration with freelance and independent consultants. These consultants are federated in a business network that we refer as CASSIS. They are trained to innovative methods produced by research projects, and they use these methods in industrial contexts. TUDOR monitors their activity to ensure that they do not deviate over the time. One of the main methodologies delivered to consultants is a requirement engineering method used to design software requirement specification (SRS) documents for IT procurement projects in small- and medium-size companies [3].

Consultants work in collaboration with customers to help them in identifying their needs for a new IT system supporting their business activities and then selecting the most relevant system accordingly to their needs. In this particular context, requirement engineers' consultants define SRS for external customers and not for their internal purpose. Consultants' customers are usually looking both for an IT system and for its implementation. In other words, they have requirements towards an IT system and towards additional services. For this reason, the scope of the SRS often encompasses functional, nonfunctional, and nontechnical requirements.

The initial goal of the SRS is to serve as a basis for a competitive procurement process. So their primary use is for IT sales managers to understand the needs of the customer and to propose a commercial bid. Only when this process is achieved, the SRS is used in second intend as source for the design or the customization of the selected IT system.

So far, consultants and TUDOR have performed more than 40 projects in compliance with the methodology. The initial approach for capitalizing requirement knowledge among the consultants was quite basic. It consisted in reusing fragments of a former SRS as a basis to build the new SRS. This approach was simple to use but required to be aware of the former projects, which was not easy for the consultants due to their decentralized organization in a business network.

The second TUDOR approach to capitalize requirement knowledge was to design SRS' templates based on existing SRS with similarities. This approach no longer requires the consultants to be aware of all former projects. However, the SRS' templates remained unstructured as domain experts built them both on their own knowledge and on assumptions of similarities found in existing SRS but without any underlying meta-model.

The limitations of these reuse approaches led us to the adoption of a more elaborated framework for requirement reuse.

## 5.3 Patterns in Requirement Engineering

As in any other software engineering discipline, reuse has been a matter of research in requirement engineering. Reviewing the literature, we may find different approaches for implementing a reuse program within the context described in Sect. 5.2, i.e., facilitating the process of requirement elicitation and also improving the quality of the resulting SRS. We may classify these approaches depending on the structure of capitalized knowledge, the language in which the requirements are expressed, the classification and browsing capabilities of the repository, and the existence of a method for building, evolving, and exploiting the requirement knowledge repository. From these aspects, in this chapter we focus on the first one, the structure of the capitalized information using patterns.

In the context of engineering, the term "pattern" was introduced by the architect Christopher Alexander that proposed them to improve the quality of the buildings' construction. In his view, *"each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"* [4]. This formulation is so generic that fitted well in other engineering domains and in particular, software engineers adopted it in several contexts, remarkably related with software design (being software design patterns [5] and software architectural patterns [6] the most representative approaches) but also in other software development phases. In particular, several approaches have proposed the use of patterns as a reuse strategy in the requirement engineering phase, which can be roughly classified as follows:

- Specific pattern-based approaches. We group here those approaches whose patterns cannot be applied in every project but just in those that are compliant to some property. Examples are:
  - Artifact-oriented patterns. Patterns that apply to a particular type of model or diagram. For instance, use case patterns propose use cases to be included in the specification of a system to ensure some properties or achieve some goals [7].

  – Domain-oriented patterns. Based upon the notion of variability proposed in domain engineering. While common requirements are necessary in any system of the domain, other requirements can be chosen or not for a specific system [8]. In some of these proposals, rules are provided to establish dependencies among variable parts of the requirement specifications.

- Refinement-oriented pattern-based approaches. They establish how the attainment of certain goals can be achieved in a certain system. They usually adopt a goal-oriented modeling language as *i** [9] or KAOS [10]. Requirement engineers are guided in the process of deciding which requirements are necessary to implement in a system to satisfy certain goals.
- Template-oriented pattern-based approaches. They propose templates with some additional information about when to use them. The ultimate goal of these approaches is to produce an SRS:
  – In their simplest form, they do not follow any structure, or this structure is very basic even if enriched with some search facilities [11, 12]. In these cases, they promote direct reuse (i.e., copy and paste) of templates as requirements, which are written as natural language sentences usually compliant to a language grammar [13].
  – More elaborated approaches include additional information about the context where they can be applied that guides the requirement engineer during the requirement elicitation process [14, 15]. Usually these proposals are general purpose in terms of domain although others are specific (e.g., [16, 17] for real-time patterns). Most of them still keep natural language as preferred notation for expressing the requirements, but we may find some that use other notations (e.g., UML [16]) or even combine two (this is the case of [17] that combines natural language with real-time temporal logics).

In the rest of the chapter, we present our PABRE template-oriented approach to conduct pattern-based requirement elicitation. It consists of a meta-model that describes the main concepts around our notion of pattern [18], a method to conduct the elicitation process [19], a catalogue of patterns classified according to some schema, and a tool that supports its management and use [20]. The main result of the application of PABRE is an SRS whose requirements are written in natural language.

## 5.4 Software Requirement Patterns in PABRE

In this section we describe the notion of *software requirement pattern* (SRP) as used in PABRE. We present the structure of patterns through a meta-model (see Fig. 5.1) and an example, the *economic information* pattern (see Fig. 5.2), that illustrates the SRP structure and helps to understand the meta-model behind them.

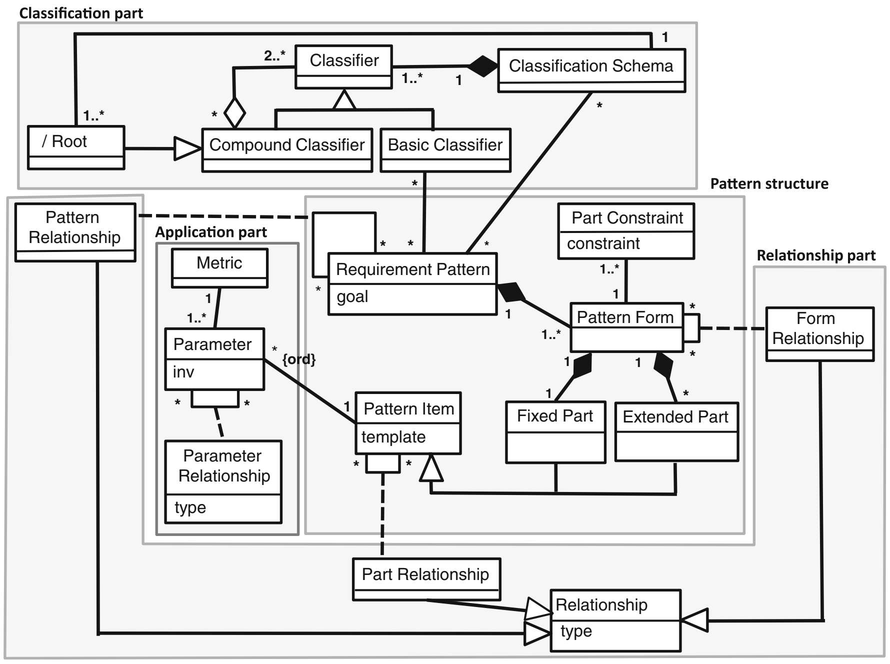


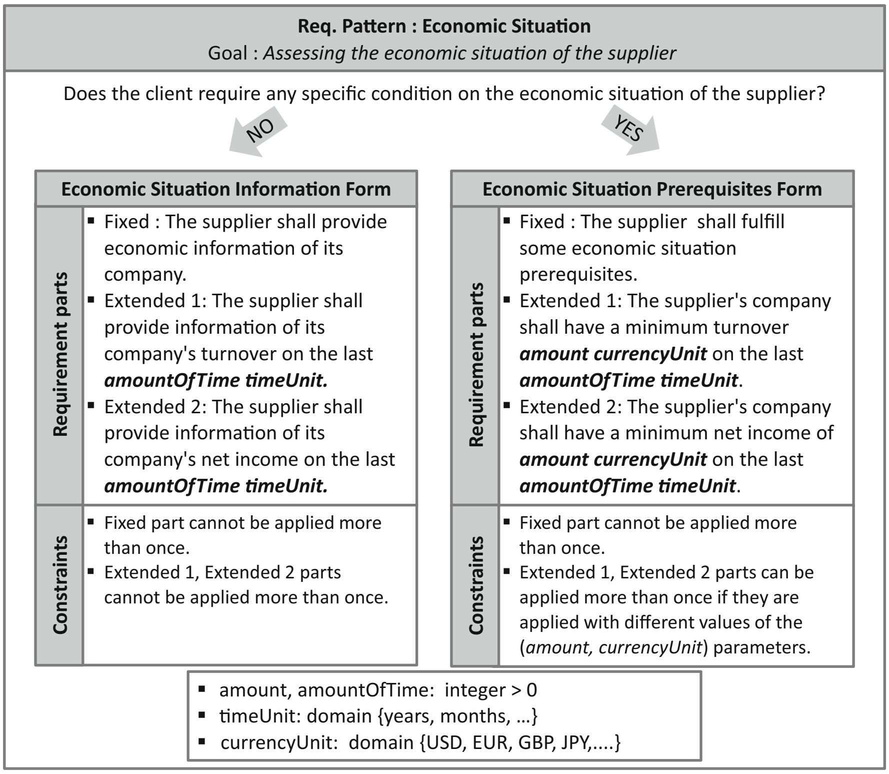
**Fig. 5.1** Meta-model for software requirement patterns

An SRP is a pattern that, when applied, produces software requirements related to the objective (goal) of that pattern. Giving an analogy with the context-problem-solution Alexander's definition of patterns, goals correspond to *problems* to be solved by applying the SRP. Applying the *economic situation* SRP, we may produce requirements related to the goal of *assessing the economic situation of the supplier* that procures a software system.

In our analysis of SRS, we have observed that a goal can be achieved in different ways. To deal with this situation, we define an SRP as consisting of several *forms*, each one representing a different *solution* for achieving the goal. In the *economic situation* SRP, its goal can be attained by asking the supplier the relevant economic information (*economic situation information* form) or by setting conditions or prerequisites on the economic situation that the supplier should have (*economic situation prerequisites* form).

Nevertheless, even considering a *form*, we may find variations in the way they are detailed in different specifications. We have therefore organized a *form* into *parts*, each of them being a template. Each *form* is characterized by a *fixed part* which states the minimal requirement that always applies when applying that form and some *extended parts* which may be applied or not in each occurrence in a project.

The *fixed part* always becomes a requirement when an SRP is applied with this *form*. *Extended parts* are only used if more precise information is required in the specification. Due to this nature, the *fixed part* is usually quite generic and hardly measurable. For instance, the first form of *economic situation* is *the supplier shall provide economic*



**Req. Pattern : Economic Situation**
Goal : *Assessing the economic situation of the supplier*

Does the client require any specific condition on the economic situation of the supplier?

**Economic Situation Information Form**

Requirement parts

- Fixed : The supplier shall provide economic information of its company.
- Extended 1: The supplier shall provide information of its company's turnover on the last ***amountOfTime timeUnit.***
- Extended 2: The supplier shall provide information of its company's net income on the last ***amountOfTime timeUnit.***

Constraints

- Fixed part cannot be applied more than once.
- Extended 1, Extended 2 parts cannot be applied more than once.

**Economic Situation Prerequisites Form**

Requirement parts

- Fixed : The supplier shall fulfill some economic situation prerequisites.
- Extended 1: The supplier's company shall have a minimum turnover ***amount currencyUnit*** on the last ***amountOfTime timeUnit***.
- Extended 2: The supplier's company shall have a minimum net income of ***amount currencyUnit*** on the last ***amountOfTime timeUnit***.

Constraints

- Fixed part cannot be applied more than once.
- Extended 1, Extended 2 parts can be applied more than once if they are applied with different values of the (*amount, currencyUnit*) parameters.

- amount, amountOfTime: integer > 0
- timeUnit: domain {years, months, …}
- currencyUnit: domain {USD, EUR, GBP, JPY,….}

**Fig. 5.2** The *economic situation* software requirement pattern

*information of its company*, while the two extended parts identify the type of information required (company's turnover or net income) and the period of time.

In general, fixed and extended parts must conform to some *part constraint* represented by means of a regular expression that may involve some predefined operators (e.g., for declaring multiplicities or dependencies among parts, as *excludes* and *requires*). In the *economic situation* SRP, each part of the forms may be used just once in a specification project, and there are neither *excludes* nor *requires* dependencies among them.

From a syntactic point of view, both fixed and extended parts are similar, therefore an abstract superclass *pattern item* is included in the meta-model. Their *templates* are composed by the text to be used as a requirement and optionally some *parameters* to be instantiated when applying the pattern. Parameters establish their *metric*, eventually a correctness condition *inv*, and also may be *related* to other parameters (belonging to other patterns) such that they must have the same value. The second form in the *economic situation* SRP declares two extended parts that identify additional conditions on this form. For example, the second extended part allows stating prerequisites on the net supplier incomes (by assigning values to the parameters *amount* and *currencyUnit*, e.g., 1M EUR) for a certain period of time (by assigning values to the parameters *number* and *timeUnit*, e.g., 2 years). The metrics of these parameters are detailed at the bottom of the figure.

SRP are not isolated units of knowledge; instead there are several types of relationships among them. In the PABRE approach, we identify three types of relationships:

– *Pattern relationship*. The most general relationship that implies all the forms and all the forms' parts of the related patterns.
– *Form relationship*. A relationship at the level of forms implies all the parts of the related forms.
– *Part relationship*. The relationship only applies to these two parts.

In any case, if *A* is related to *B* and *A* is applied in the current project, the need of applying or avoiding *B* must be explicitly addressed. The types of relationships are not predetermined in the meta-model to make it more flexible. The superclass *relationship* includes an attribute to classify each relationship.

## 5.5 A Catalogue for Software Requirement Patterns

The existence of patterns by themselves does not ensure an efficient implementation of requirement reuse. It is necessary to set up an infrastructure able to support the analyst to organize and apply them. In the PABRE framework, we are coping with this aspect through a catalogue of SRP.

### *5.5.1 Structure of the Catalogue*

PABRE's catalogue stores the collection of SRP identified so far. A fundamental issue is the need of classifying them over some criteria for supporting their search. In fact, it is important to observe that different contexts (organizations, projects, standards, etc.) may, and usually do, define or require different *classification schemas*. History shows that trying to impose a particular classification schema does not work. For this reason, PABRE decouples SRP from classification schemas (see Fig. 5.3): the latter just impose different structuring schemas on top of the former. SRP are bound to *basic classifiers*, while *compound classifiers* just impose the usual hierarchical structure of any classification schema. Several *roots* for a classification schema are allowed.

The meta-model (Fig. 5.1) shows that an SRP may be bound to several classification schemas and even to more than one basic classifier in a single classification schema. In other words, we do not impose unnecessary constraints that could lead to rigidness. For instance, a classification schema may not cover all existing SRP (i.e., some SRP may not be classified).

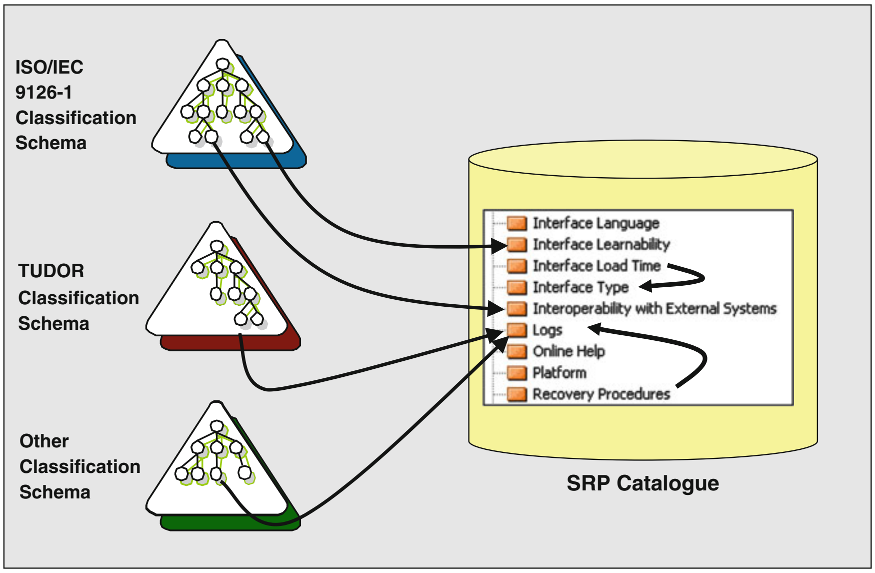

ISO/IEC 9126-1 Classification Schema
TUDOR Classification Schema
Other Classification Schema
Interface Language
Interface Learnability
Interface Load Time
Interface Type
Interoperability with External Systems
Logs
Online Help
Platform
Recovery Procedures
SRP Catalogue

**Fig. 5.3** Software requirement pattern classification schemas

### 5.5.2 *SRP Catalogue Construction*

The current PABRE SRP catalogue was built as a result of analyzing the SRS of a certain number of projects in which TUDOR was involved. These SRS are usually broken down into three distinct parts: functional requirements, nonfunctional requirements (NFR), and nontechnical requirements (NTR). Our previous experience in quality models [21] and in requirement engineering projects and the analysis of TUDOR SRS showed us that nonfunctional requirements and nontechnical requirements have higher reuse frequency than functional requirements. Then, our aim for the first version of the catalogue was to represent those SRP whose application leads to NFR that appear in the mentioned SRS [22]. From the experience gained, we recently finished the second version of the catalogue in which we added the SRP corresponding to the NTR, as presented in Sect. 5.6.

In both cases, the steps (Fig. 5.4) were:

1. Alignment. First, the requirements of the different SRS are consolidated and aligned according to their type. This corresponds to the identification of the departing requirements in the SRS. To make this alignment more reliable, it is convenient to identify the concepts addressed by requirements. As part of the process, requirements need to be leveraged, which usually requires decomposing complex requirements into simpler ones. As a result, this step delivers a set of requirement types.
2. Analysis. For each of these types, a study of their adequacy as an SRP is performed. The main criterion of course is repetition that identifies high

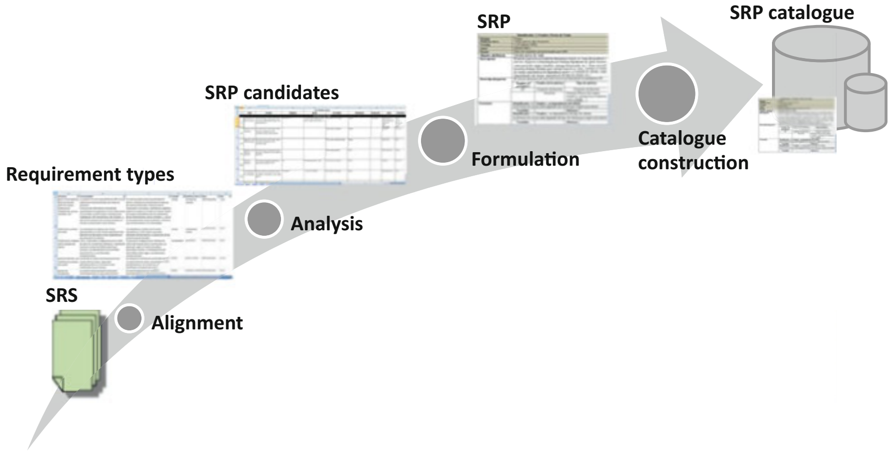


**Fig. 5.4** Software requirement pattern catalogue construction process

probability of reuse: those requirements that appear in most or all of the SRS are clear candidates. But this is not the only condition. A requirement appearing in a few, even just one, SRS may also be considered adequate as SRP. In this step, expert assessment is the cornerstone, since experts are the only ones that may say, e.g., that a requirement appearing in just one SRS could in fact have appeared in all of them, in other words that its absence is a flaw. As a result, this step restricts the former set to a subset with all the types that may be considered patterns' seed or SRP candidates. The different requirement types are converted into SRP candidates mainly by means of abstraction, but also a consistency analysis and grammatical improvement are applied.

3. Formulation. The selected SRP candidates are converted into SRP. Not every candidate is necessarily converted into a different SRP, since some of them may be considered close enough as to be integrated in the same pattern. As a result, the final structure of the patterns, their forms, their parts, and parameters, emerges. In the process, again with expert assessment, the final structure of every SRP may be slightly different than the corresponding requirements in the SRS, since experts may consider that for future projects these differences could be useful. For the templates, syntactical conventions may be enforced.
4. Catalogue construction. Finally, the patterns evolve from individual artifacts into an articulated structure of knowledge, stored in the catalogue. Two things need to be done. First, the SRP need to be classified according to the existing classification schemas. Second, the relationships among SRP are established, as well as those (less frequent) among parameters.

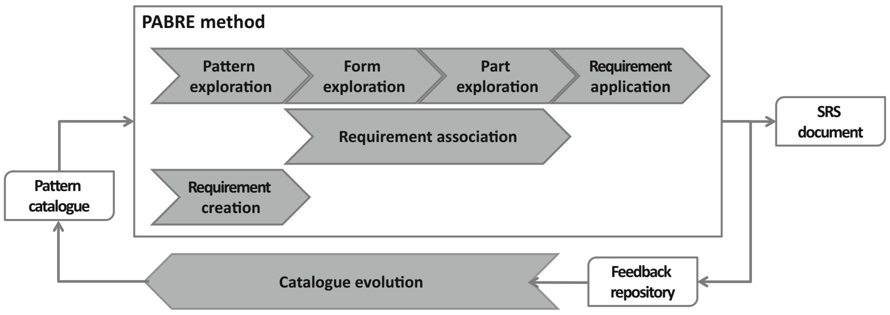


**Fig. 5.5** Overview of the PABRE method

## *5.5.3 The SRP Catalogue Use*

The SRP catalogue is used during the requirement elicitation phase of IT systems and services procurement projects. During this use, requirement engineers select SRP from the catalogue that apply to the particular project and convert them into the real requirements that finally configure the SRS. The complete PABRE method is detailed in [7]. In a nutshell, it converts requirement elicitation into a process of search in, and pick-up from, the SRP catalogue (Fig. 5.5).

During elicitation, the catalogue is explored according to the following procedure:

- Pattern exploration. The requirement engineer selects the next applicable pattern according to some criteria (e.g., the classification schema, the SRP relationships). Based on an explanation and with continuous support from the engineer, the customer decides whether the pattern applies in the project or not.
- Form exploration. For each selected pattern, the requirement engineer explains the different forms. Then the customer chooses the form that suits his/her situation and moves to the next step. If no form meets the customer requirements, the requirement engineer elaborates the requirement(s) and moves to the requirement creation step.
- Part exploration. For each selected form, the requirement engineer explains the different extended parts. If it is necessary, the consultant skims over the parameters and gives example of possible values, in order to improve understanding of the parts. The customer chooses the extended parts that considers necessary for his/her project. As well as in the previous steps, if no extension fits completely into the customer needs, it is necessary to elicit the missing bits separately.

At this point, the requirement may be defined in different ways. Figure 5.6 shows the three types or *requirement* subclasses and their relationships regarding the SRP meta-model:

- Applied pattern. For the selected parts, the requirement engineer gives more details about the parameters that apply (e.g., details on possible correctness

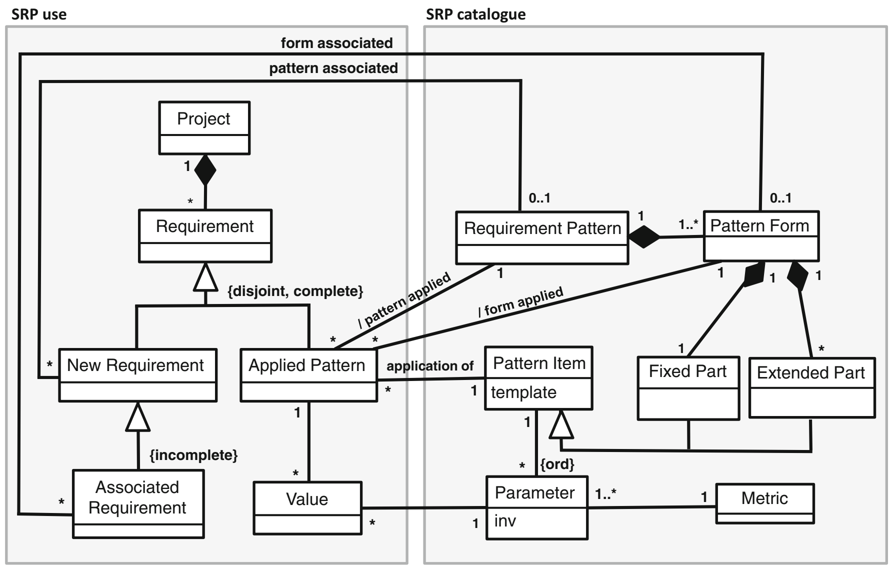


Fig. 5.6 Use of software requirement patterns

conditions, dependencies to/from other parameters) and presents the list of values for each parameter. Then the customer chooses the values for the parameters. The requirement engineer turns the customized part(s) into a requirement. The requirement engineer needs to check consistency, dependencies, and correctness of the selected parts. When the requirement engineer detects a conflict or an inconsistency, he/she warns the customer and they try to solve the conflict. The resulting requirement is represented with the *applied pattern* subclass.

- New requirement or associated requirement. Sometimes, the requirement engineer needs to create a *new requirement* from scratch because the restriction expressed by the requirement cannot be defined as application of any SRP. We distinguish one particular case: if the new requirement is related with an existent pattern, since it has its same goal, but it is not its direct application, this new requirement is an *associated requirement*. An associated requirement consists of partial and small changes of the pattern or the forms (its part's text or parameters).

### 5.5.4 The SRP Catalogue Evolution

Catalogue evolution allows capitalizing the different projects and keeping the SRP catalogue up-to-date. The requirement experts identify the patterns, forms, extended parts, and parameters which are the most and less used. According to their feedback, different actions can be undertaken to evolve the catalogue.

The feedback is obtained by having the real numbers of SRP applications, the associated requirements to patterns or forms, and the new requirements, over time:

- The number of applications of a pattern versus the number of associations to that pattern can be used by the requirement engineer as a guarantee of the validity of the SRP. If the number of applications is low regarding its associations, maybe the requirement engineer has to check the associated requirements in order to find out if there is some problem with the definition of the requirement. On the other hand, the number of applications is a confirmation of the validity of the pattern.
- The associated requirements have to be analyzed because they can correspond to forms or parts of a pattern that have never been identified before, and that would be helpful for the requirement analysts to have them as parts of the pattern.
- In the case of new requirements, it has to be analyzed if there has been an error in defining them as new or if in fact the requirement analyst is right and there is no goal corresponding to the new requirements represented by any SRP of the catalogue. In the first case, the new requirement is analyzed as an associated requirement, and in the second case, the new requirement is considered for being added as an SRP following the lasts steps presented in Sect. 5.2.

## 5.6 A Software Requirement Pattern Catalogue for Nontechnical Requirements

The goal of this section is to illustrate the process of construction of a set of SRP presented in Sect. 5.5.2. We describe the construction of the SRP catalogue part corresponding to NTR applicable to TUDOR's projects. NTR are those requirements that do not refer directly to the intrinsic quality of software, but to the context of the system under analysis. They include economic, political, and managerial issues. This type of requirements is highly independent of the software domain and, for this reason, good candidate for our work. The complete catalogue of NTR patterns (NT SRP for short) is available in the PABRE website (http://www.upc.edu/gessi/PABRE/index.html).

### *5.6.1 Preliminaries*

We used six SRS as starting point of the process, which is distilled next in terms of the different steps enumerated in Sect. 5.5.2.

In these six SRS documents, specific sections were supposed to contain separately NFR and NTR. However, when building the previous catalogue of SRP for NFR, we discovered that this separation was not clear, since some NTR were discovered in the NFR section. As a result, besides the 29 NFR patterns, we already identified three patterns that became the initial set of NT SRP.

**Table 5.1** Examples of aligned requirements

| Concept | Requirement | Keywords |
|---|---|---|
| Maintenance period | The solution should be maintained for three (3) years from the expiration of the warranty period | Maintenance period<br>Warranty |
| Maintenance period | The proposed solution must be maintained for at least 1 year from the date of expiry of the warranty period | Maintenance period<br>Warranty |
| Maintenance period | From the date of expiry of the warranty period, the contractor agrees to provide, at the explicit request of the client, ongoing maintenance services for a minimum period of 1 year | Maintenance period<br>Warranty |
| Audits | The *customer* reserves the right to conduct audits of the provider and its production during the project | Audits<br>Provider<br>Project production |
| Audits | These audits will focus on the specific development (product code, development methodology, documentation), the treatment of the reported anomalies, and quality procedures | Audits<br>Specific development<br>Reported anomalies<br>Quality procedures |

The requirements in the SRS were written in French. However, the biggest core of knowledge on requirement engineering is available mainly in English. Also, for dissemination purposes, we had the goal of producing the pattern templates in English too. Therefore, before the alignment process, we translated the requirements into English. The translation was supervised by the TUDOR team since French is their native language, but they are also fluent in English.

### 5.6.2 Alignment

Next, we undertook the alignment looking for requirements expressed differently in each of the SRS but addressing the same concept. Table 5.1, first three rows, shows three requirements appearing in different SRS but related to the concept, namely, *maintenance period.*

On the other hand, some SRS requirements were broken into several simple requirements. For instance, the two last rows of Table 5.1 show two requirements that appeared in an SRS as one single complex requirement.

### 5.6.3 Analysis

In this step it was necessary to consider the requirements that address the same concept to be joined, by means of abstraction, consistency analysis, and improvement of the grammatical form, in requirement types.

Table 5.2 shows the list of candidate SRP for the requirements presented above in Table 5.1. The first one corresponds to the abstraction of the three first

**Table 5.2** Examples of requirement types

| Concept | Requirement | Keywords |
|---|---|---|
| Maintenance period | The supplier shall maintain the system for ***number timeUnit*** from the expiration of the warranty period | Maintenance period<br>Warranty |
| Audits | The customer shall do audits of the supplier or the project deliverables if it is considered necessary | Audits<br>Supplier<br>Project deliverables |
| Audits | The audits shall focus on the ***quality aspects*** | Audits<br>Specific developments<br>Reported anomalies<br>Quality procedures |

requirements in Table 5.1. The requirement was abstracted in order to allow the statement of different periods of maintenance after the end of the warranty. This example shows a usual way to implement abstraction, namely, substituting specific aspects related to one project by parameters with some associated metric (which of course allows the generation of the abstracted requirements).

Also, some grammatical rules on the SRP templates were enforced. Examples are the following: requirements were written in an active voice; requirements were written in third person and with use of the modal verb *shall* suitable for legal requirements or statements.

To ensure catalogue consistency, we built and maintained a glossary of terms and metrics. Since we started from the previous state of the catalogue which contained nonfunctional SRP, metrics as *timeUnit* and terms as *supplier* and *system* were already therein. This last term was used to substitute the *solution* in the SRS. Also other terms were substituted for the same reasons as *project production* by *project deliverables*.

Also consistency among requirements was checked. For example, we found two requirements at the same SRS: “at each steering committee meeting, a statement of progress will be prepared and signed by the parties” and “the report will be prepared by the provider and approved by the customer, if necessary after the required updates” related to the *steering committee meetings* requirements. As can be seen, in the two requirements, a different term is used to refer the meeting reports (statement in the first requirement), and inconsistencies among the report approval process are present in them. Therefore clarification was needed to ensure consistency.

### 5.6.4 Formulation

The requirement types corresponding to SRP candidates were processed iteratively, considering at each iteration one type of candidates addressing the same concept. At each iteration, the considered types were compared to the set of the already approved SRP, in order to decide their treatment: approval as a new SRP, incorporation as parts of existing SRP, or discard.

We illustrate the formulation with a particular NT SRP. The requirement types related to *audits* (see Table 5.2) were included in the catalogue as just one SRP since they address the same concept. When all the SRP candidates aligned to *audits* were considered, we observed that there are two different groups: one constraining audits for assessing the quality of the supplier in a general way and the other that constraints audits conducted according to a certain quality standard. Therefore, the resulting SRP was structured into two alternative forms: *general quality assessment* form and *quality standard-based assessment* form (see Table 5.3). In the first form, the most general requirement type has been selected as the fixed part of the form while the other becomes an extended part, since this second type of requirement will not appear in a project without including the first one.

The process above was iterated for the rest of requirement types. Eventually, we found some special situations. On the one hand side, some requirement types were restricting the *delivered documents* of the project. When the glossary was browsed, this term was found as the name of an existing nonfunctional SRP. Therefore, these types were analyzed with respect to this SRP: some types were found redundant regarding to the existent pattern, while others were used to constitute a new pattern. The nonfunctional SRP *delivered documents* address the statement of requirements on the content of delivered documents, and the NT SRP *document characteristics* allow constraining the characteristics of the documents (i.e., their language, electronic format, metadata to include, etc.). Finally both patterns were considered as related to NT aspects, although the first one is also nonfunctional due to its relationship with the maintenance and understandability of a system and therefore may appear also classified under this perspective.

On the other hand, some of the requirement types dealt with one restriction on the concept *source code*. Specifically they were about the need of documenting the source code. In this case, they were added as extended parts of the already existent *source code* NT SRP.

As already mentioned, during this step and the previous one, expert assessment was crucial. Validation was done by requirement engineers from TUDOR with wide experience in requirement elicitation. Some relevant observations follow.

First of all, the experts provided a general observation about the focus of the forms. For instance, for those SRP referring to suppliers, most were asking for information about the supplier, instead of restricting how the supplier should be or should behave. They proposed to formulate improved forms of the SRP in a more prescriptive way. For instance, this was done in the case of the SRP *supplier workforce*, whose goal was initially formulated as "having information about the supplier workforce" and whose only form's fixed part was "the supplier shall provide workforce information about the company." After the expert's assessment, the goal was transformed into "assessing the workforce of the supplier" and a new form was added establishing a restriction of the supplier workforce with the fixed part "the supplier shall fulfill some workforce requirements." Both forms have extended parts to establish different aspects of the workforce information to obtain or to restrict respectively.

Experts also suggested restructuring some SRP while iterations progressed. Examples of actions are the following: SRP merged during the process due to redundancy, extended parts upgraded into fixed parts, and even reallocation of extended parts from one SRP to another. For instance, the *installation* SRP was

**Table 5.3** *Quality assessment* nontechnical software requirement pattern

| | |
|---|---|
| ***Quality assessment*** | |
| ***Goal**: Stating the customer's right of performing quality assessment* | |
| **Requirement form** | **Part constraints** |
| *General quality assessment* | *Fixed part* cannot be applied more than once |
| | *Review focus* cannot be applied more than once |
| | *Quality criteria agreements* cannot be applied more than once |
| | **Fixed part** |
| | **Form text** If the customer considers it necessary during the system implementation project, s/he shall be allowed to assess the quality of the process or the ***projectDeliverables*** |
| | **Parameters** |
| | **projectDeliverables** is a non-empty set of the different products delivered during the system implementation project |
| | **Metrics** |
| | ProjectDeliverables = Set(ProjectDelive-rable) |
| | ProjectDeliverable = Domain (hardware, software, data and documents provided or paid by customer as project deliverables, etc.) |
| | **Extended part** *Review Focus* |
| | **Form text** The customer shall focus the quality assessment on the ***qualityAspects*** |
| | **Parameters** |
| | **qualityAspects** is a non-empty set of the different quality aspects to be assessed |
| | **Metrics** |
| | QualityAspects = Set (QualityAspect) |
| | QualityAspect = Domain (specific development, treatment of the reported abnormalities, quality procedures, etc.) |
| | **Extended part** *Quality criteria agreement* |
| | **Form text** The customer shall agree with the supplier on the level of quality expected for the various project deliverables |
| **Requirement form** | **Part constraints** |
| *Quality standard- based assessment* | *Fixed part* cannot be applied more than once |
| | *Process quality assessment* cannot be applied more than once |
| | *Deliverables quality assessment* can be applied more than once, only if it is applied for different values of the *projectDeliverables* and *qualityStandard* parameters |
| | *Quality criteria agreement* cannot be applied more than once |
| | *Quality criteria establishment* cannot be applied more than once |
| | **Fixed part** |
| | **Form text** If the customer considers it necessary during the system implementation project, s/he shall be allowed to assess the quality of the process or project deliverables taking into account a quality standard |
| | **Extended part** Process quality assessment |
| | **Form text** The quality of the process shall be assessed taking into account the ***qualityStandard*** quality standard |

(continued)

**Table 5.3** (continued)

| | |
|---|---|
| | **Parameters** |
| | **qualityStandard**: represents the identifier of the quality standard that shall be used to assess the quality |
| | **Metrics** |
| | QualityStandard = Domain (IEEE830, IEEE829, IEEE1016, ISO/IEC9126, ISO/IEC 15504-5, etc.) |
| | **Extended part** Deliverables quality assessment |
| | **Form text** The quality of the ***projectDeliverables*** shall be assessed taking into account the ***qualityStandard*** quality standard |
| | **Parameters** |
| | **projectDeliverables** as above |
| | **qualityStandard** as above |
| | **Metrics** |
| | ProjectDeliverables as above |
| | QualityStandard as above |
| | **Extended Part** *Quality criteria agreement* |
| | **Form text** The customer shall agree with the supplier on the level of quality expected for the project deliverables |
| | **Extended part** *Quality criteria applied* |
| | **Form text** The customer shall establish the subset quality standard criteria to be applied ***timePreposition date*** |
| | **Parameters** |
| | **timePreposition** represents the relationship with respect to a date |
| | **date**: is a time point representing the date in which the quality standard criteria shall be established |
| | **Metrics** |
| | TimePreposition = Domain (on, before, after, at, by, etc.) |
| | Date = TimePoint |

subsumed by the *implementation planning* SRP, since in this SRP it is already established the planning of the different activities, being installation just a particular case. Also, changes in the vocabulary and abstraction from specific contexts of application were continuously performed. For instance, in the case of the SRP about *audits*, the experts suggested to change in the SRP body the action “audit” by “assess of the quality.”

After the validation step, we arrived to 38 NT SRP.

### 5.6.5 Catalogue Construction

The created NT SRP were stored in the PABRE catalogue. As already mentioned, the catalogue already contained three NT SRP identified in the previous version of the catalogue: *help desk*, *crash response*, and *source code documentation*.

**Table 5.4** NT SRP extended ISO/IEC 9126–1 classification

| Classifier | Basic classifier | NT SRP | NT SRP goals |
|---|---|---|---|
| 1. Supplier | 1.1. Organizational structure | Supplier administrative information | Being able to contact the supplier |
| | | Supplier organization | Understanding the supplier's organization |
| | | Supplier history | Being aware of the history of the supplier company |
| | 1.2. Positioning and strength | Supplier economic information | Assessing the economic situation of the supplier |
| | | Supplier workforce | Assessing the workforce of the supplier |
| | 1.3. Reputation | Supplier business experience | Assessing project's experience |
| | | Supplier quality certification | Assessing quality certification of the supplier |
| | 1.4. Services offered | Training | Stating the training the supplier shall provide about the implemented system |
| | 1.5. Support | Maintenance procedure | Assessing the supplier's maintenance procedures |
| | | Type of maintenance | Stating the specific types of maintenance for the system implemented the supplier shall provide |
| 2. Business | 2.1. Licensing schema | Source code licenses | Stating the source code licenses |
| | 2.2. Ownership | Intellectual property rights | Stating the rights of using assets result of the project |
| | 2.3. Guarantees | Warranty | Stating the warranty that shall be applied over the implemented system |
| | 2.4. Costs | Cost breakdown structure | Stating the structure of the global cost of the system to be implemented |
| 3. Project | 3.1. Business scheduling | System implementation scheduling | Stating the scheduling of the system implementation |
| | | Project progress control | Having or stating the indicators for assessing the progress of the project |
| | | Project management method | Stating the method used for project management |
| | | Final acceptance | Stating the time and conditions for the final acceptance of the implemented system |
| | | Release | Stating the time and conditions when the implemented system shall be released |
| | | Analysis stage activities | Stating the activities to take during analysis stage |
| | | Data migration | Stating the necessity of migrating data |

| | | | |
|---|---|---|---|
| | | Development activities | Stating the activities to take during development stage |
| | | Acceptance tests | Stating the type of tests for the system implementation acceptance |
| | 3.2. Supplier relationships | Steering committee | Stating the steering committee organization |
| | | Meetings organization | Stating system implementation meetings organization |
| | | Access to customer premises | Stating the rules for supplier access to customer premises |
| | | Privacy | Stating the privacy rules among customer and supplier |
| | | Project progress control | Having or stating the indicators for assessing the progress of the project |
| | | Quality assessment | Stating the customer's right of performing quality assessment |
| | | Payment procedure | Stating the payment schedule |
| | | Settlement of disputes | Stating how the disputes between customer and supplier shall be solved |
| | | Supplier people assigned to the project | Assessing the profile of the people assigned to the project |
| | | Help desk | Having access to a technical support service for the system for information and assistance |
| | | Crash response | Stating the required level of service for supplier support in case of crash |
| 4. Product | 4.1. History | Products history | Assessing the history of the main products that will be part of system to be implemented |
| | | Community support | Assessing the existence of a community that could give support on the implemented system |
| | 4.2. Deliverables | Delivered documents | Stating the documentation that shall be delivered |
| | | Source code documentation | Stating the source code licenses |
| | 4.3. Parameterization and customization | - - - - - - - - - - - - - - - - - - - - - - - | |

The NT SRP were classified in terms of the two classification schemas incorporated into PABRE so far: the ISO/IEC 9126–1 standard [23] and the classification schema defined by the TUDOR center. In this section we illustrate the classification using the ISO/IEC 9126–1 standard.

ISO/IEC 9126–1 does not include nontechnical features. However, in previous works we enlarged this standard with NT features [21], and we use this extension (called NT-ISO/IEC 9126) in the PABRE catalogue, which adds three characteristics (*supplier*, *business*, and *product*) and 15 subcharacteristics to the standard. Before classifying the NT SRP according to this schema, some changes had to be done to take into account some differences on the use of the catalogue.

On the one hand, during the process of classification, we found 19 patterns that did not correspond to any subcharacteristic in NT-ISO/IEC 9126. The reason is that initially that catalogue was created to include the criteria to assess the quality of a final software product, whereas the NT SRP state requisites for the procurement of a system (probably by gluing or adapting several products). This is the reason why we needed to add a new characteristic to group the SRP about the implementation project: the *project* characteristic, decomposed into two subcharacteristics – *business scheduling* and *supplier relationships*.

On the other hand, some related subcharacteristics were merged into just one. Specifically, they were those related to the cost of the business. The original subcharacteristics were too static: *licensing costs*, *platform costs*, *implement costs*, and *network costs*, but the new subcharacteristic integrates all these costs in a cost breakdown structure allowing the flexibility to add new ones.

Also relationships among the SRP were investigated. With this aim, we took into account the keywords stated for each SRP (obtained during their construction) and also the metrics of the parameters of the different SRPs. For the *quality assessment* SRP, taking into account the parameter *ProjectDeliverables* (Table 5.3), we identified a dependency with the *delivered documents* SRP that also has a parameter with the same metrics. The relationship is that the documents for which a quality assessment is done must be deliverable documents.

In Table 5.4, the 38 SRP are classified taking into account the extended NT ISO classification schema updated to include the new identified characteristics and subcharacteristics.

## 5.7 Conclusions

In this chapter we have presented the PABRE framework for reusing requirement knowledge following a pattern-based approach. The different components of PABRE have been introduced: its meta-model, the processes supported, and the catalogue of patterns. For illustration purposes, we have described the construction of the first version of a set of 38 nontechnical requirement patterns that follow the structure stated in the PABRE meta-model. Requirement engineering experts from the TUDOR research center have been collaborating in this construction.

Future work spreads over several dimensions:

- Validation of the adequacy of PABRE in other types of IT projects beyond the procurement projects targeted so far
- Adoption of clear rules and best practices for writing pattern templates (see e.g. EARS [24])
- Extension of the catalogue with functional patterns from several domains (e.g., in the context of TUDOR, ERP, and CRM procurement projects)
- Improving capabilities of tool support by introducing recommendation capabilities (e.g., "projects that used this pattern usually use this other")

In addition, more validation is needed. We have so far conducted postmortem analysis of the SRS coming from past projects to validate that the meta-model covers the features expressed in those SRS and the coverage of the catalogue is satisfactory. Still, we need to apply it to real cases in an action-research basis.

**Acknowledgements** This work has been partially supported by the Spanish project TIN-2010-19130-C02-01.

## References

1. Chung L, do Prado Leite JCS (2009) On non-functional requirements in software engineering, conceptual modeling: foundations and applications. Springer-Verlag, Berlin, Heidelberg, pp 363–379
2. Carvallo JP, Franch X, Quer C (2006) Managing non-technical requirements in COTS selection. In: IEEE international requirements engineering conference (RE), Minneapolis/St.Paul, Minnesota, USA
3. Renault S, Barafort B, Dubois E, Krystkowiak M (2007) Improving SME trust into IT consultancy: a network of certified consultants case study. In: EuroSPI, Potsdam, Germany
4. Alexander C (1979) The timeless way of building. Oxford University Press, New York
5. Gamma E, Helm R, Johnson R, Vlissides J (2000) Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA
6. Buschmann F, Meunier R, Rhonert H, Sommerlad P, Stal M (1996) Pattern-oriented software architecture: a system of patterns. John Wiley and Sons Ltd, Chichester, UK
7. Chung L, Supakkul S (2006) Capturing and reusing functional and non-functional requirements knowledge: a goal-object pattern approach. In: IEEE international conference on information reuse and integration (IRI), Waikoloa, Hawaii, USA
8. Mannion M, Kaindl H (2008) Using parameters and discriminants for product line requirements. Syst Eng J 11(1):61–80
9. Supakkul S, Hill T, Chung L, Tun TT, Leite JCSP (2010) An NFR pattern approach to dealing with NFRs. In: IEEE international requirements engineering conference (RE), Sydney, Australia
10. Darimont R, van Lamsweerde A (1996) Formal refinement patterns for goal-driven requirements elaboration. In: ACM symposium on foundations of software engineering (SIGSOFT), San Francisco, CA, USA
11. Monzon A (2008) A practical approach to requirements reuse in product families of on-board systems. In: IEEE international requirements engineering conference (RE), Barcelona, Catalunya, Spain

12. Hull E, Jackson K, Dick J (2010) Requirements engineering, 3rd edn. Springer-Verlag New York Inc
13. Watahiki K,Saeki M (2001) Scenario patterns based on case grammar approach. In: IEEE international symposium on requirements engineering (RE), Toronto, Canada
14. Withall S (2007) Software requirement patterns. Microsoft Press, Redmont, Washington
15. Toval JA, Nicolás J, Moros B, Garcia F (2002) Requirements reuse for improving information systems security: a practitioner's approach. Requir Eng 6(4):205–219
16. Konrad S, Cheng BHC (2002) Requirements patterns for embedded systems. In: IEEE joint international conference on requirements engineering (RE), Essen, Germany
17. Konrad S, Cheng BHC (2005) Real-time specification patterns. In: ACM/IEEE international conference on software engineering (ICSE), St. Louis, Missouri, USA
18. Franch X, Palomares C, Quer C, Renault S, DeLazzer F (2010) A metamodel for software requirement patterns. In: Requirements engineering: foundation for software quality (REFSQ), Essen, Germany
19. Renault S, Mendez-Bonilla O, Franch X, Quer C (2009) A pattern-based method for building requirements documents in Call-for-tender processes. Int J Comp Sci Appl 6(5):175–202
20. Palomares C, Quer C, Franch X (2011) PABRE-Man: management of a requirement patterns catalogue. In: IEEE international requirements engineering conference (RE), Trento, Italy
21. Carvallo JP, Franch X, Quer C (2007) Determining criteria for selecting software components: lessons learned. IEEE Softw 24(3):84–94
22. Renault S, Mendez-Bonilla O, Franch X, Quer C (2009) PABRE: pattern-based requirements elicitation. In: International conference on research challenges in information science (RCIS), Fès, Morocco
23. ISO Standard 9126 (2001) Software engineering – product quality, part 1. International organization for standarization
24. Mavin A, Wilkinson P, Harwood A, Novak M (2009) Easy approach to requirements syntax (EARS). In: IEEE international symposium on requirements engineering (RE), Atlanta, Georgia, USA

# Chapter 6
# Using Ontologies and Machine Learning for Hazard Identification and Safety Analysis

**O. Daramola, T. Stålhane, I. Omoronyia, and G. Sindre**

**Abstract** Safety analysis (SA) procedures, such as hazard and operability analysis (HazOp) and failure mode and effect analysis (FMEA), are generally regarded as repetitious, time consuming, costly and require a lot of human involvement. Previous efforts have targeted automated support for SA at the design stage of system development. However, studies have shown that the cost of correcting a safety error is much higher when done at the later stages than the early stages of system development. Hence, relative to previous approaches, this chapter presents an approach for hazard identification (HazId) based on requirements and reuse-oriented safety analysis. The approach offers a convenient starting point for the identification of potential system safety concerns from the RE phase of development. It ensures that knowledge contained in both the requirements document and previously documented HazOp projects can be leveraged in order to attain a reduction in the cost of SA by using established technologies such as ontology, case-based reasoning (CBR), and natural language processing (NLP). The approach is supported by a prototype tool, which was assessed by conducting a preliminary evaluation. The results indicate that the approach enables reuse of experience in conducting safety analysis, provides a sound basis for early identification of system hazards when used with a good domain ontology and is potentially suitable for application in practice by experts.

O. Daramola (✉)
Covenant University, Ota, Nigeria
e-mail: olawande.daramola@covenantuniversity.edu.ng

T. Stålhane • G. Sindre
NTNU, Trondheim, Norway
e-mail: stalhane@idi.ntnu.no; guttors@idi.ntnu.no

I. Omoronyia
University of Glasgow, Lanarkshire, Scotland
e-mail: Inah.Omoronyia@glasgow.ac.uk

## 6.1 Introduction

Safety analysis (SA) embraces all of the hazard identification (HazId), risk and safety assessment activities involved in the development of safety-critical embedded systems. The goal of SA is to influence safety-critical system design by conducting several types of safety procedures in order to identify potential system hazards and risks and to mitigate them to acceptable levels before a system is certified. Safety analysis procedures, such as hazard and operability analysis (HazOp) and failure mode and effect analysis (FMEA), are generally regarded as repetitious, time consuming, costly and require a lot of human involvement [1–3]. Although human expertise is irreplaceable in the conduct of effective SA procedures at the moment, there is a need to reduce the amount of human effort and cost of SA. Previous efforts to address this problem have been based largely on expert system approaches, which target automated support for SA from the design stage of system development [1, 4]. However, studies [5, 6] have shown that the cost of correcting a safety error is much higher when done at the later stages than the early stages of system development. Since requirements engineering (RE) precedes system design, it provides a convenient starting point for the identification of potential safety concerns of a system if the knowledge contained in requirement documents can be extracted and used as the initial basis for SA. Hence, tool support for SA at the RE phase will be more beneficial for attaining a reduction in the cost of hazard identification and hazard mitigation.

HazOp is one of the prominent safety analysis techniques [4]. HazOp is used to study hazards and operability problems by investigating the effects of deviations from prescribed design intent in order to mitigate the occurrence of adverse consequences. It involves early discovery of potential system hazards and operation problems and recommendation of appropriate safeguard mechanisms by a team of experts.

However, HazOp is a time consuming, costly and a largely human-centred process [1, 3, 6]. The HazOp process is essentially subjective, relying on the professional experience, expertise and creativity of the team members involved. Some of the crucial challenges of HazOp which are still open research issues are: (1) how to reduce the level of subjectivity, (2) how to reduce the amount of human effort, (3) how to promote reuse of valuable knowledge gained in previous HazOp studies and (4) how to facilitate transfer of HazOp experiences among HazOp teams [3, 7]. These challenges motivate the need for a framework that could enable early identification of hazards and reuse-oriented HazOp analysis. The first objective of this work is to provide a decision support tool that could assist the human expert in the process of identifying potential safety concerns that are contained in the requirements document. The second is to create a platform for the reuse of knowledge from previous HazOp studies in subsequent projects, in order to reduce the amount of human effort needed while conducting HazOp. This work would be useful in the safety analysis of product line systems or variant systems, where the systems share a significant degree of commonality. Also, the approach could be valuable in the context of system development models that are iterative or incremental in nature where there is a need to continually revise requirements and design specifications during the period of development. Our focus on HazOp stems from the interests of the CESAR project[1] that we are currently involved in.

We have adopted an approach that combines three technologies to realise the stated objectives of this work, namely:

- Case-based reasoning (CBR), which is a pattern-based problem solving paradigm that enables the reuse of previously gained knowledge in resolving a new case [8]
- Ontology, which is the semantic representation of the shared formal conceptualization of a domain that provides a platform for the standardisation of terms and vocabulary in the domain [9]
- Natural language processing (NLP) which is the processing and analysis of natural language text [10]

A prototype tool called KROSA (knowledge reuse-oriented safety analysis) that demonstrates the novel integration of these three technologies has been created to validate our approach. The unique contribution of this work is the integration of ontology and machine learning technologies into a framework that enables the identification of hazards from requirements and reduction of effort needed for HazOp through knowledge reuse. In this chapter, we present a description of the proposed framework and the evaluation of the prototype tool by an experiment and opinions provided by domain experts at ABB Norway.

The rest of this chapter is organised as follows: Sect. 6.2 presents the background for the context of this chapter, while Sect. 6.3 describes a HazOp problem example and how tool support can be provided for HazId based on requirements. In Sect. 6.4, we give a description of the KROSA framework and how it can be used for HazOp. Section 6.5 presents the evaluation procedure used for assessing the KROSA tool, while Sect. 6.6 discusses the results of the evaluation and the threats to validity of results. In Sect. 6.7, we review some closely related work, and the chapter is concluded in Sect. 6.8 with a brief note and indication of our future research plans.

## 6.2 Background

In this section, we give a brief overview of the general HazOp process and the key technologies that are relevant to this work.

### *6.2.1 Overview of the HazOp Process*

A hazard and operability study (HazOp) is a structured and semiformalised team-based procedure that focuses on the study of a system under design, in order to identify and evaluate potential hazards that may constitute a risk to personnel or

[1] http://www.cesarproject.eu

equipment or prevent efficient operation of the system. A HazOp study is undertaken by a HazOp team through a series of brainstorming sessions in order to stimulate creativity used to reveal potential hazards in the system and their cause–effect relationships [1, 4]. HazOp is based on the assumption that a problem can only arise when a system deviates from its design and operational intents.

Hence, the HazOp study entails a detailed walkthrough of the process and instrumentation diagram models of a system to spot every likely deviation from its intended operation using a set of *guidewords*. Generally, guidewords represent variations of known system parameters that may cause deviation from design intentions. They are chosen and interpreted based on particular design representation and context. Examples include no, not, more, less, before, after, late, too often and early. Examples of parameter–guidewords pairs include arrive late, arrive early, no flow, not sent and sent after. Guidewords are carefully selected to stimulate reasoning about all potential system hazards. A point of observation pertaining to a system or process that can be a source of a potential hazard is called a *study node*. As each deviation is derived, the HazOp team discusses potential causes, consequences and safeguards and recommend appropriate control actions to forestall or mitigate its occurrence.

Typically, it takes about 1–8 weeks for a HazOp team with 4–8 members to conduct a HazOp, depending on the size and complexity of the system in question. It is widely accepted that HazOp analysis is an extremely time-consuming process [1, 3, 4]. More on the procedure of HazOp study and ideals of HazOp team, membership composition can be found in [11].

### *6.2.2 Case-Based Reasoning (CBR)*

CBR is an instance-based machine learning paradigm that emulates the human reasoning process of solving problems based on past experiences. In CBR, problems are modelled as abstraction called *cases* which consist of the problem part and the solution part. The CBR life cycle [8] is a four-stage process that consists of (1) *case retrieval* – where old cases that are similar to a new case are identified by comparing the problem parts of the old cases and that of the new case using a similarity metric; (2) *case reuse* – which entails applying the solution part of the most relevant old case or group of old cases to the new case, and this may also involve adaptation of the old solutions to fit the new case; (3) *case revision* – where the reused solution is tested for appropriateness in the new case, and if need be, the reused solution is revised to fit the new case; and (4) *case retention* – which entails storing a solved case in the case base (repository) for future reuse. CBR provides a mechanism of organising, storing and reusing an organisation's memory or experiences. As such, it offers a credible model of experience-based problem solving once relevant cases exist [12]. The CBR paradigm is considered particularly relevant to the context of HazOp because of its potential to support the acquisition, retrieval, reuse and retention of knowledge, which provides a basis for documented experiences from previous HazOp studies to be leveraged in subsequent HazOp projects.

### 6.2.3 Ontology

Ontology which is a shared formal conceptualization of a domain is a key technology to shaping and exploiting information for the effective management of knowledge that pertains to specific domains [13]. Ontologies have human and machine-readable semantics that allow definition of semantic relationships between entities and inference of knowledge through reasoning at runtime. According to [14], ontologies have the capability to (1) enable knowledge reuse, (2) ensure better understanding of a knowledge area, (3) support analysis of the structure of knowledge, (4) foster understanding of available knowledge in a domain and (5) provide embedded knowledge for an application that can be used by machines. Ontology is considered relevant to the HazOp problem because of its potential to facilitate (1) formalised semantic description of relevant domain knowledge for identification of system hazards, (2) interoperable transmission of knowledge among HazOp teams and (3) knowledge reuse while conducting HazOp.

### 6.2.4 Natural Language Processing (NLP)

NLP is concerned with the process of extracting meaningful information from natural language text through the use of statistical machine learning algorithms [10]. In NLP, machine learning algorithms automatically learn rules through the analysis of large *corpora* of real-world examples. A *corpus* (plural, “corpora”) is a set of documents that have been manually annotated with the correct values to be learned. The learned rules are then used to classify words into various word categories (part of speech) following the supervised learning model. Key NLP operations include sentence tokenisation, part-of-speech tagging, coreference resolution, anaphora resolution, named-entity recognition and morphology analysis. NLP is a necessity for automated requirements analysis because requirements are mostly written as natural language text. Therefore, our approach uses NLP in combination with ontology to enable the extraction of useful knowledge from natural language requirement documents for the early identification of potential system hazards.

### 6.2.5 Knowledge Management in Requirements Engineering

In recent times, the application of knowledge management technologies such as ontologies, NLP and CBR has gained momentum in requirements engineering. In [15], the SoftWiki approach was reported as a way of semantifying requirements engineering. According to the authors, semantification of RE entails representing each requirement as a unique instance of the Semantic Web having its own URI such that spatially distributed stakeholders – including developers and users – can collect, semantically enrich, classify and aggregate requirements within the context of collaborative software development. The approach uses the SoftWiki Ontology for Requirements Engineering (SWORE) to facilitate the semantification process. Similarly, [14] gave an elaborate overview of how ontologies can be applied in collaborative software development and the vision of a software engineering Semantic Web.

In [16], a framework for requirements elicitation using ontology reasoning was proposed. NLP was used to parse initial requirements to obtain key concepts that can be mapped to functions in the domain ontology. Thereafter, the rules and relations among functions in the ontology were used to reason about errors and potential requirements. Other research efforts where ontologies have been applied for requirements elicitation and analysis include [17] where a domain ontology and requirements meta-model were used to elicit and define textual requirements; in [18], an approach for goal-oriented and ontology-driven requirements elicitation (GOORE) was proposed. In GOORE, the knowledge of a specific domain is represented as an ontology, which is then used for goal-oriented requirements analysis.

In [19], a perspective for the application of CBR for requirements engineering was provided. Also, [12] gave a detailed account of the probable applications of CBR in software engineering in the aspects of prediction and reuse. In [20], CBR was used to evaluate the requirements quality by referring to previously stored software requirements quality analysis cases (past experiences) in order to ensure that the quality of the prepared SRS is acceptable, while [21] proposed a framework for managing implicit requirements by using a combination of ontology and CBR. All of these efforts indicate an increasing interest in the application of ontology, NLP and CBR as knowledge management technologies in requirements engineering.

## 6.3 Simplified Steam Boiler Example

The steam boiler system is a simplified version of an industrial steam boiler, developed as a first pilot system for testing CESAR concepts. In order to have a simple system, important components such as the feeding tank and the blow down valve are left out.

The functional requirements of the steam boiler are as follows:

1. The steam boiler shall deliver steam at a predefined, constant pressure to an industrial process.
2. Steam is produced by heating water using an electric heating element.
3. The steam pressure is controlled by regulating the temperature setting on the heating element thermostat.
4. The water level in the tank is controlled by a feeding pump which pumps water into the tank via a non-return valve.
5. The safety of the steam boiler is taken care of by a safety valve that opens to air. The release pressure for the safety valve is fixed, based on the boiler’s strength.
6. The system shall be safety integrity level two (SIL2) certifiable.

In the CESAR project, we have embraced the notion of requirements boilerplates[2] which stems from the work in [22, 23] for writing requirements in semiformalised form. A boilerplate is a textual template for requirements specification that is based on predefined patterns, which reduces the level of inconsistency in the way requirements are expressed. We have also introduced additional requirement boilerplates patterns that are considered well suited for embedded systems requirements.

For the steam boiler example, we will now use the following predefined sample boilerplates[2]:

BP1: **The** < system > **shall** < action>
BP2: **The** < system > **shall be able to** < action > **using** < system>
BP3: **If** < condition>, **the** < system > **shall** < action>

The functional requirements of the steam boiler can then be transformed to a semiformal form as follows:

***R1: The*** *< steam boiler >* ***shall be able to*** *< deliver > [<steam >* ***to*** *< an industrial process>]* – BP1
***R2: The*** *< steam boiler >* ***shall be able to*** *< produce > [<steam >* ***using*** *(<electrical > <heating element>)]* – BP2
***R3: The*** *< steam boiler >* ***shall be able to*** *< control > [<steam pressure >* ***using*** ***(****<thermostat > of < electrical > <heating element>)]* – BP2
***R4: The*** *< steam boiler >* ***shall be able to*** *< control > [<water level >* ***using*** *(<feeding pump>)]* – BP2
***R5: The*** *< feeding pump >* ***shall be able to*** *< deliver > [<water >* ***using*** *(<non-return valve>)]* – BP2
***R6: If*** *[<steam pressure >* ***greater than*** *< critical pressure level>]****, the*** *< steam boiler >* ***shall*** *[<open > <safety valve>]* – BP3

## 6.3.1 Preliminary HazOp (PHA) for Steam Boiler

Usually, based on a concept diagram for system – say a steam boiler – a team of experts would run a PHA by brainstorming on specific requirements and components of the system in order to identify potential hazards that may arise from possible deviations from the design intent of the steam boiler. The result of such a PHA for a steam boiler system would be a manually generated preliminary HazOp table. A small part of such a table is shown in Table 6.1.

[2] www.requirementsengineering.info/boilerplates.htm

**Table 6.1** Preliminary HazOp table for a steam boiler

| Req. | Element (study node) | Guideword | Hazard | Cause | Main effect | Preventive action |
|---|---|---|---|---|---|---|
| R2 | Water tank (heating element) | Temperature too hot | The water tank is too hot | Too little water and too much heat (sensor, control, actuator, connections) | Tank gets hot/fire | Turn off the heat<br>Add water? |
| R3 | Water tank | Pressure too high | Too high pressure in the water tank (R3) | Not able to turn off the heating (sensor, control, actuator, connections) | Boiler explodes | Safety valve |
| | | | | Feeding pump failure (too strong) | Boiler rupture | Turn off the heat<br>Turn off power to the feed pump |
| R4 | Feeding pump | Water level too high | Too high water level (R4) | Water-level regulation failure (sensor, control, actuator, connections) | Water to the process | Pump emergency stop |
| R5 | Feeding pump | Non-return valve stuck | Too high pressure in the feed pipe (R5) | Non-return valve failure | Release boiling water to the water supply | Two non-return valves in series<br>Emergency valve for releasing pressure |

### *6.3.2 Tool Support for HazId Based on Requirements*

Our objective in this work is to provide tool-based support for HazId based on requirements – which is usually a costly manual procedure – such that:

1. Requirement documents can be analysed semantically using a combination of shallow NLP and domain knowledge as contained in the domain ontology, to identify potential system hazards automatically. Hence using the steam boiler ontology (see Fig. 6.2), columns 1 and 2 of Table 6.1 – a HazOp table for the steam boiler system – can be automatically generated.
2. The user is able to partially or totally reuse relevant parts of previously documented HazOp projects in order to generate causes, effect, safeguards and appropriate control actions for each system hazard that has been identified – generate data for columns 2–5 of specific hazards (study node) in Table 6.1.

With this proposed approach, we aim to provide relevant tool support for the HazOp experts so as to reduce the amount of effort needed and also to offer a good starting point for HazOp in instances where there is paucity of experts. We will now describe the architecture of our approach in the next section.

## 6.4 The KROSA Framework

The architectural framework of our proposed approach is an integration of the three core technologies NLP, CBR and ontology. A view of the architecture is presented in Fig. 6.1. The core system functionalities are depicted as rectangular boxes, while the logic, data and knowledge artefacts that enable core system functionalities are depicted using oval boxes. A detailed description of the KROSA framework is given in the following.

### *6.4.1 Knowledge Representation and Extraction*

In this section, we describe the parts of the KROSA architecture that deals with knowledge representation and extraction.

(a) Data Preprocessing

The input to the framework is a preprocessed requirements document. Preprocessing is a manual procedure that ensures that source documents are transformed into a form that is suitable for the framework. It entails extraction of requirements in form of sentences from source documents, extracting sentences that define system requirements and replacing information conveyed in figures, diagrams and tables with equivalent sentences. Also, the requirements could be expressed in

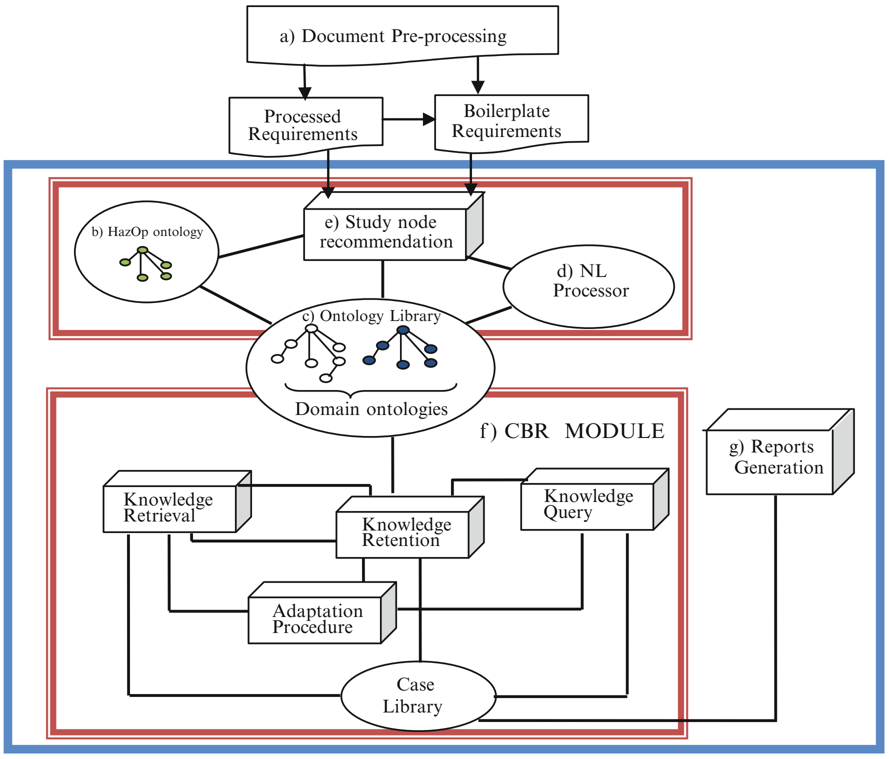


**Fig. 6.1** Architectural framework for reuse-oriented HAZOP

semiformalised way using requirement boilerplates. Boilerplate requirements will also be more susceptible to treatment by NLP algorithms (Fig. 6.1).

(b) HazOp Ontology

The HazOp ontology defines, in a generic form, the concept of a study node, its elements and the relationships between them. These are types of study node, description, guidewords, deviations, causes, consequences, risk level, safeguards and recommendation. The HazOp ontology was developed using OWL DL language and consists of 17 classes, 23 object properties and 43 restrictions. Figure 6.3 presents a schematic view of the structure of the HazOp ontology. It has two important roles: (1) helping to identify potential hazards during study nodes recommendation since its specification clearly defines which type of domain concept could be a study node and (2) validation of the structure of the HazOp information before it is stored in the case library during case retention. A HazOp study node must be one of the types defined in the HazOp ontology.

(c) Ontology Library

The ontology library is a repository of domain ontologies. The domain ontologies (.owl/.rdf) could be those that have been developed for the purpose of

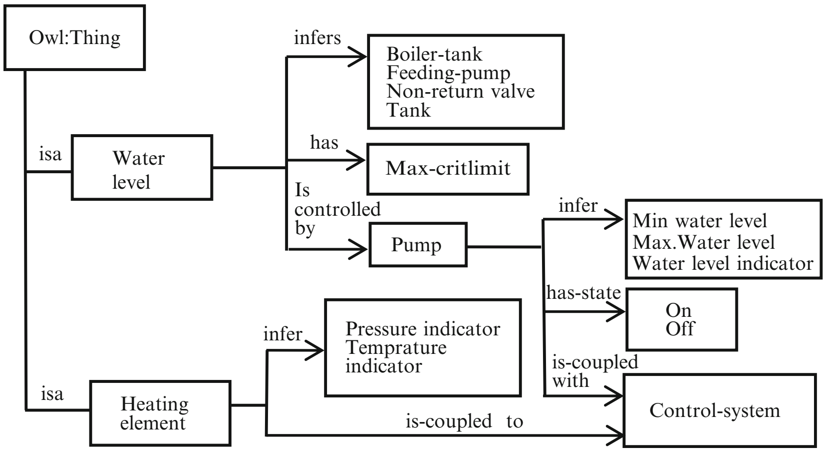


**Fig. 6.2** A view of a part of the steam boiler ontology

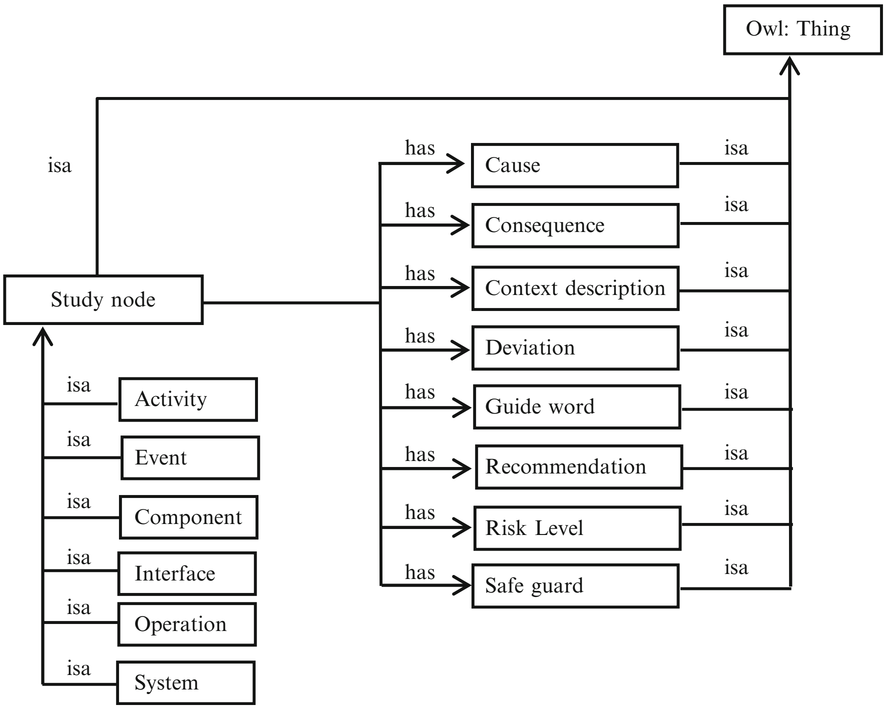


**Fig. 6.3** A view of the classes and restrictions in the HAZOP ontology

safety analysis or an existing ontology that is based on domain-specific safety standards. The domain ontology consists of all the terms in the domain and the set of relationships between terms in the domain. The domain ontology plays two roles: (1) identification of valid domain concepts that are contained in requirements document and (2) ensuring that standardised terms used in describing HazOp information during knowledge (case) retention agree with the established vocabulary of the domain. As an example, a view of part of the steam boiler ontology which describes the concepts of the steam boiler system and the interrelationships between the concepts is shown in Fig. 6.2. The ontology library, the HazOp ontology and the case library jointly constitute the knowledge model of the framework.

(d) NL Processor

The NL processor component facilitates the processing of natural language and boilerplate requirements during the process of automatic recommendation of HazOp study nodes. The core natural language processing operations implemented in the architecture are:

- *Tokenisation*: Splitting of requirements statements (sentences) into word parts.
- *Parts of speech tagging*: Classification of tokens (words) in requirements statements into parts of speech such as noun, verb, adjective and pronoun.
- *Pronominal anaphora resolution*: The process of identifying pronouns (anaphors) which have noun phrases as antecedents in requirements statements. This is essential in associating sentences that refer to the same requirement.
- *Lexical parsing*: Creating the syntax tree that represents the grammatical structure of requirements statements, in order to determine phrases, subjects, objects and predicates.

The stanford NLP toolkit[3] for natural language processing was used to implement all NLP operations.

(e) Study Node Recommendation

The procedure for automatic study node recommendation is based on a heuristic algorithm that is derived from basic knowledge of HazOp. Study node generation is not intended to replace human capability but rather to create a credible starting point for early hazard identification and to alleviate the amount human effort involved. The algorithm searches for potential study nodes in two ways:

- *Requirements level (RL)*: A requirement statement is considered a candidate if the following criteria are satisfied: (1) the requirement statement contains an *action-entity* pair such as “open valve”, “close valve”, “start pump” or “stop pump” (*action* and *entity* may not necessarily follow each other in a sentence); (2) the *action* must be an instance of a generic HazOp action word (such as: stop, close, open, send, reset, cut, receive, start or their synonyms) or one of a set of

[3] http://nlp.stanford.edu/software/lex-parser.shtml

user specified keywords, while *entity* is a valid concept in the domain ontology; and (3) the entity identified in requirement statement belongs to one of the predefined study node types (components, system, etc.) as described in the HazOp ontology.

- *Component level(CL)*: A term (word) contained in a requirement statement is considered a candidate study node if the following criteria are satisfied: (1) the term is a valid concept in the domain ontology; (2) there exists at least one axiom that pertains to the term in the domain ontology which indicates that it could be a study node (In other words, it is one of several types of study nodes as defined by the HazOp ontology) and (3) the term has failure modes or guidewords defined on it (such as stuck, omission, commission) in the domain ontology. At the CL level, terms that satisfy the criteria (1) and (3), (1), (2) and (3) or (1) and (3) are considered to be candidates. However, a term is ignored if it is same as, equivalent to or a subclass of another term that has been selected as a potential study node.

### 6.4.2 *Knowledge Reuse*

This section describes the parts of the KROSA architecture that deals with knowledge reuse and also report generation.

(f) CBR Module

The CBR component facilitates the knowledge reuse capability of the framework. It emulates the typical workflow of the CBR life cycle which is retrieve, reuse, revise and retain [8, 24]. Retrieval by the CBR module is performed by displaying a ranked list of cases similar to a target case. Two types of reuse are supported: (1) total reuse – all parts of a case are reused for a new case, and (2) partial reuse – only parts of an existing case are reused in a target case. Revision can be effected by the HazOp expert by making modifications to the selected case to suit the new target case. Retention is done through storage of study node information into the case library. The case library is implemented as a MySQL database management system (DBMS) in order to leverage its inherent capabilities for effective case organisation, case indexing, case storage and case retrieval.

(g) Report Generation

This module enables the generation of HazOp reports based on query posed by the user. HazOp reports are queried based on date and the HazOp id.

### 6.4.3 *Case Model and Case Similarity*

The case model is an abstraction of the way HazOp information is represented in the framework. A HazOp case encapsulates information attributes such as name of a study node (unique), context description and set of applicable guidewords, deviations, causes, consequences, risk levels, safeguards and recommendations. The case model is partitioned into a problem part and a solution part. The three elements of a HazOp case model that constitute the problem part are contextual description, study node type and the set of guidewords; the remaining elements of the case model make up the solution part.

At the instance of a new (target) case, an algorithm is used to compute the similarity between the problem parts of the new case and all existing relevant cases in the case library to determine suitable candidates for retrieval. The solution part of a chosen retrieved case is then used verbatim or revised as the solution part of the target case. There are several candidate similarity algorithms that can be used for case retrieval depending on the value of attributes of data elements [25]. The similarity algorithm used for comparing cases is based on the degree of intersection between two attributes of a case, which are the set of contextual descriptions and the set of guidewords, while the type of study node is used to determine relevant cases. Similarity between an attribute of the new case U and a corresponding attribute of an existing case V is determined by computing the metric:

$$Sim(U,V) = \frac{|U \cap V|}{|U|} \tag{6.1}$$

where

$$U \cap V = \{x : x \in U and x \in V\}$$

*Case Similarity*: Finally, the similarity between two cases is computed by using the weighted sum of the individual similarity metrics, where $w_i$ denotes the weight assigned to the ith attribute of a case. This is given as [26]:

$$Sim\_final = w_1 sim_{(context)} + w_2 sim_{2(guidewords)} \tag{6.2}$$

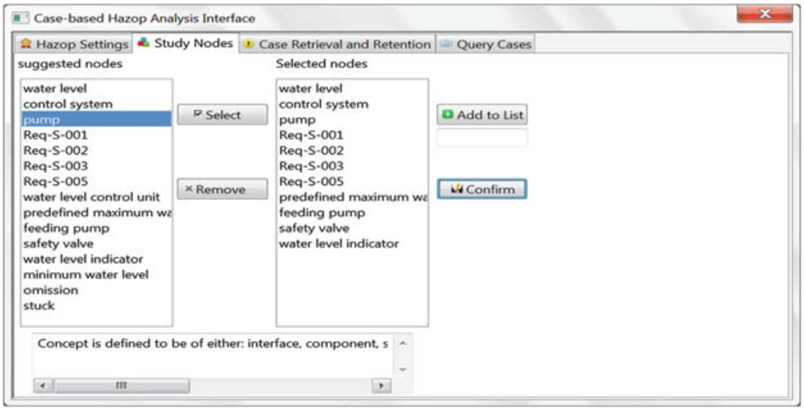
We have used equal weights (i.e., $w_1 = w_2 = 1$) since the parameters are considered as equally important.

## 6.5 Performing HazOp with KROSA

The process of using the KROSA tool for HazOp is as follows:

*Step 1*: Preprocessing of source documents to get the requirements into MS Excel or text file format and devoid of graphics, images and tables.

*Step 2*: Select existing domain ontology or create a new one to be used for the HazOp.



suggested nodes

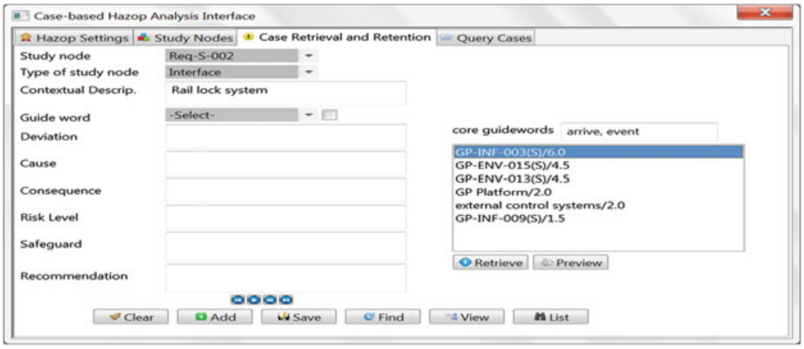
water level
control system
pump
Req-S-001
Req-S-002
Req-S-003
Req-S-005
water level control unit
predefined maximum wa
feeding pump
safety valve
water level indicator
minimum water level
omission
stuck

Selected nodes

water level
control system
pump
Req-S-001
Req-S-002
Req-S-003
Req-S-005
predefined maximum wa
feeding pump
safety valve
water level indicator

Concept is defined to be of either: interface, component, s

Fig. 6.4 A view of recommended study nodes by the KROSA tool



Study node: Req-S-002
Type of study node: Interface
Contextual Descrip.: Rail lock system
Guide word: -Select-
Deviation
Cause
Consequence
Risk Level
Safeguard
Recommendation

core guidewords: arrive, event

GP-INF-003(S)/6.0
GP-ENV-015(S)/4.5
GP-ENV-013(S)/4.5
GP Platform/2.0
external control systems/2.0
GP-INF-009(S)/1.5

Fig. 6.5 A view of ranked list of similar cased retrieved by the tool

*Step 3*: Import requirement documents and domain ontology into the KROSA environment.

*Step 4*: Supply the set of keywords that best describe the focus of the HazOp.

*Step 5*: Obtain recommended study nodes from KROSA.

*Step 6*: Expert approves a set of study nodes for the HazOp by selecting from or adding to the recommendations by KROSA.

*Step 7*: For each approved study node, expert leverages KROSA's case retrieval, reuse and retention features to generate information for specific study nodes. By doing so, the user attempts to save some effort by using content of the reuse repository to provide information for new study nodes. Figures 6.4 and 6.5 are snapshots of the interfaces for study node recommendation and case retrieval for reuse in the KROSA tool, respectively.

## 6.6 Evaluation

We have developed the KROSA[4] (knowledge reuse-oriented safety analysis) tool, a domain-independent CBR platform for ontology-supported HazOp that is based on the Eclipse plug-in architecture. In following subsections, we discuss how the KROSA tool can be integrated into the HazOp process and subsequently describe the procedure used for its evaluation.

### *6.6.1 Evaluation Procedure*

KROSA has been subjected to two kinds of evaluation: first, an in-house simulation experiment to assess the quality of its recommendation of study nodes, using requirements specifications obtained from ABB Norway, one of our partners in the CESAR project. Second, we performed a field assessment where industry experts from ABB Norway assessed the usability of KROSA for an industrial HazOp process. The objectives of the field evaluation were threefold: (1) to assess the consistency of the outcome of the tool as judged by the human experts, (2) to assess the potential of the tool to enable reuse-oriented HazOp and (3) to determine its usefulness as a support tool for safety analysis. Also, we wanted to identify areas of possible improvement of the tool.

#### 6.6.1.1 Simulation Experiment

In the simulation experiment, we worked with three sets of requirements: (1) rail lock system, (2) steam boiler control system and (3) adaptive cruise control (ACC) system. Three ontologies used for the experiment are *rail lock system ontology*, *steam boiler ontology* and *ACC ontology*. Two of the ontologies (steam boiler and ACC ontology) had existed prior to KROSA, having been used to support previous ontology-based research project in CESAR [27]. These two ontologies have a fairly wide circulation among CESAR partners. The rail lock system ontology was created for this experiment, based on information obtained from the specification of the GP rail lock system. The three ontologies have the common characteristics that they were developed to be usable for safety analysis in addition to other uses. This is because (1) safety relevant terms were used to describe ontological concepts, e.g., object properties such as *isComponent*, *isConcept*, *isFailuremode* and *isInterface* exist in the ontologies; (2) the semantic description of components included the definition of generic failure modes such as *stuck*, *omission* and *commission*. The simulation experiment compared recommendations from KROSA with those obtained from four safety experts (researchers) for the same

[4] KROSA tool can be downloaded at https://www.idi.ntnu.no/~wande/Krosa-user-guide.htm

set of requirements. We then computed the recall and precision scores for KROSA relative to the recommendations made by each of the four safety experts that participated in the experiment (see Eqs. 6.3 and 6.4):

$$precision = \frac{|\{Expert.recomm\}| \cap |\{KROSA.recomm\}|}{|\{KROSA.recomm\}|} \tag{6.3}$$

$$recall = \frac{|\{Expert.recomm\}| \cap |\{KROSA.recomm\}|}{|\{Expert.recomm\}|} \tag{6.4}$$

#### 6.6.1.2 Expert Assessment

For the field assessment, the direct method of expert systems evaluation [28, 29] was used. This method entails making qualified human experts to use a system for solving a simple benchmark problem; thereafter based on their experience, the human expert answers a set of questions about the system. The questions are quantitative and based on a 0 (completely false) to 5 (very true) numerical scales. A metric called "satisfaction level" that ranges from 0 (least satisfied user) to 5 (most satisfied user) is then computed based on the data obtained from all participants. The satisfaction level is a measure of the likelihood of the system to satisfy a prospective user.

The questions, the objective of each question and the weight associated with each question (which all the participants agreed on) are as follows:

1. *Sufficient information is provided for guidance and orientation of evaluators prior to conducting the experiment* (orientation) – (2).
2. *The KROSA tool reaches a conclusion similar to that of a human expert* (correctness of result) – (2).
3. *Does the KROSA tool provide reasonable justification for its conclusion?* (correctness of result) – (2).
4. *The KROSA tool is accurate in its suggestions of study nodes* (accuracy of result) – (2).
5. *The result is complete. The user does not need to do additional work to get a usable result* (accuracy of result) – (2).
6. *Does the result of the system change if changes are made to the system parameters?* (sensitivity) – (1).
7. *The overall usability of the KROSA tool is satisfactory* (confidence) – (1).
8. *The KROSA tool gives useful conclusions* (confidence) – (2).
9. *The KROSA tool adequately supports reuse of knowledge for HazOp* (support for reuse) – (2).
10. *The KROSA tool improves as data, or experience is inserted* (support for reuse) – (1).

11. *The limitations of the KROSA tool can be detected at this point in time* (limitation) – (1).
12. *There are still many limitations to make the KROSA tool usable* (limitation) – (1).

An evaluator gives a score between 0 and 5 per question. From the scores, a weighted score for the satisfaction level per evaluator can be calculated using the metric below:

$$\mathrm{Re}\ sult = \sum_{k=1}^{n} (weight^{*} scorevalue) / \sum_{k=1}^{n} weight \tag{6.5}$$

*where n is the number of questions.*

A one-day orientation workshop on how to use the tool was conducted for all participants, after which they had one full week to interact with the tool. The expert participants also had a detailed user manual as further guide for using the tool.

## 6.7 Evaluation Results

In this section, we give an overview of results from the two evaluations carried out.

### *6.7.1 Simulation*

Table 6.2 shows the recall and precision scores computed for KROSA relative to the four safety experts' (E1–E4) recommendations. Although the experts differed in their recommendations, confirming the subjective nature of HazOp, there exist significant agreements between study nodes recommended by KROSA and experts at the requirements level. At the component level (CL), there was a greater degree of agreement because the opinions of the safety experts generally agree that all components and interfaces between components and systems should be study nodes as recommended by KROSA. Since the experts were generally not very specific in their recommendations at the CL, recommendations at CL were not considered when arriving at the values in Table 6.2. The result – precision[5] and recall[6] values – shown in Table 6.2 is an improved version of the one reported in [27] since we have had more time to improve on the quality of the domain ontologies.

Our observation from the simulation experiment (see Figs. 6.6 and 6.7) is that the performance of the KROSA tool depends significantly on the quality of the

[5] Precision – percentage of suggested hazards that are relevant compared to expert's recommendation.

[6] Recall – percentage of relevant hazards suggested by tool compared to expert's recommendation.

**Table 6.2** Showing recall and precision values of KROSA

| Recall | E1 | E2 | E3 | E4 |
|---|---|---|---|---|
| Steam boiler system (HazOp on water level) | 0.57 | 0.67 | 0.75 | 0.60 |
| ACC system (HazOp on speed control) | 0.50 | 0.67 | 0.71 | 0.60 |
| Rail lock system (HazOp on communication) | 0.54 | 0.78 | 0.71 | 0.60 |
| **Precision** | **E1** | **E2** | **E3** | **E4** |
| Steam boiler system (HazOp on water level) | 0.67 | 0.67 | 0.50 | 0.50 |
| ACC system (HazOp on speed control) | 0.80 | 0.80 | 1.0 | 0.6 |
| Rail lock system (HazOp on communication) | 0.78 | 0.78 | 0.56 | 0.33 |

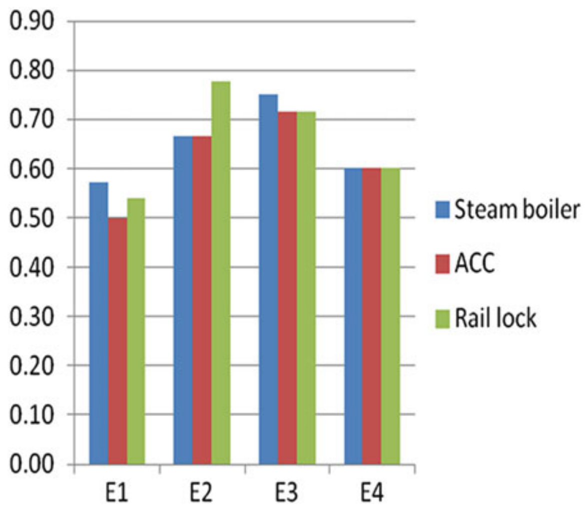


**Fig. 6.6** Recall metric for KROSA

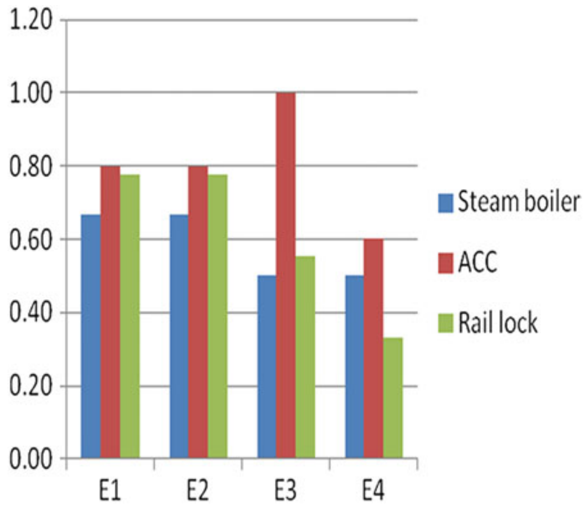


**Fig. 6.7** Precision metric for KROSA

domain ontology, even though the input of highly relevant keywords can enhance the appropriateness of the recommended study nodes. Specific ontology qualities are considered most crucial here, which are [30] (1) *syntactic quality* – the measure of the correctness of terms in the ontology and the richness of syntax used to describe terms in the ontology, (2) *semantic quality* – the measure how well the meaning of terms is defined in the ontology and (3) *pragmatic quality* – the measure of the how well it covers the scope of a domain judged by the number of classes and properties it contains and how accurate and relevant the information is that it provides. A domain ontology that contains a large number of concepts that are credible and are richly described with axioms will be more suitable for KROSA in the task of study node recommendation. Initially, we noticed that the KROSA tool had a relatively lower precision for HazOp of the steam boiler system compared to its performance in the HazOp for the ACC and rail lock systems. The reason for this was that the steam boiler ontology has a lower semantic quality than the ACC ontology and rail lock system ontology. After we improved on the quality of description of concepts and interrelationships between concepts of the steam boiler ontology, we obtained better results. This is not difficult to comprehend since the domain ontology provides the knowledge base from which inferences are made by the KROSA tool when determining what could be a potential system hazard (study node). Thus, we conclude that the overall quality of the domain ontology affects the performance of KROSA tool significantly, as it determines the extent to which inferences can be made for identification of study nodes.

### *6.7.2 Expert Evaluation*

Each of the three industry experts that took part in the assessment returned an evaluation report from which we computed a mean weighted score of 3.27 out of 5 for the KROSA tool in relation to the evaluation objectives of the field assessment. The tool obtained its highest mean score ratings in the aspects of *support for reuse* (4.08), *sensitivity* (3.67), *confidence* (3.25) *and accuracy of result* (3.25), while the lowest mean score ratings were in the aspects of: limitations (3.0) and *correctness of result* (2.7). These mean score ratings reveal the perception of the experts in terms of the strengths and weaknesses of the current version of the tool. The experts also submitted a detailed report on desired improvements needed to make the tool more usable. Key aspects mentioned as needing improvement were (1) the possibility of providing some form of guidance to users in the selection of the most appropriate keywords for study node recommendation and (2) the need to provide some form of traceability links between cases that have inherited some parts from old cases through reuse. The experts were unanimous in confirming that the tool will be a valuable support for the conduct of HazOp, with the potential to alleviate the complexity of the HazOp process by enabling reuse of experience.

The experts agreed that the existence of a domain ontology and a case library where previous knowledge is stored in a structured format would help to resolve some of the existing difficulties associated with searching, update and interoperability of knowledge during HazOp. They expressed preference for the adoption of KROSA as a support tool for HazOp over the current scenario where MS Excel software is the main tool support for their safety analysis.

## 6.7.3 Threats to Validity

Our short discussion on the validity of the preliminary evaluation will be based on the categories defined by Wohlin et al. in [31]. We consider each threat before giving a summary of validity of our results.

*Conclusion Validity*: In order to ensure reliable treatment, all participants were provided with an introduction and instructions for the experiment prior to the experiment. Also, we used standard measures – recall and precision to assess recommendations by the tool in order to avoid misunderstanding or misinterpretation of the results. Ordinarily using four participants in the experiment will translate to low statistical power, but for highly technical domain like HazOp and a preliminary evaluation, we consider this to be sufficient for a first trial.

*Internal Validity*: A key requirement is that participants have sufficient experience or knowledge of the domain. The participants had minimum master-level education in the area of systems safety. They were also provided with detailed instructions of what should be done. Therefore, there were no factors other than the treatment that influenced the outcome of the experiment.

*Construct Validity*: In order to ensure a realistic experiment, all participants had the same instruction for the experiment. Also, they performed exactly the same task which is to identify hazards (study nodes). Hence, the results obtained from participants depend only on this task (one single variable), which eliminates any mono-method bias effect.

*External Validity*: The key issue here is whether we can generalise our results from the preliminary evaluation to the system safety industry. For the simulation experiment, we used four expert researchers all affiliated with NTNU, while the industrial assessment was done by three safety experts at ABB Norway. A concern could be that possibly there would have been different results if the evaluations had been performed with a bigger group of participants with more diverse background, not only in terms of coming from different institutions and countries but also with more different educational backgrounds and covering a wider spectrum of safety-critical domains than could be achieved with only seven persons. The involved persons mainly had experience in safety analysis in the following domains: railway, automotive and industrial automation, and it is impossible to know if the tool would have been found equally promising by experts from other domains, such as nuclear power, medical technology and aviation. Our mitigation to this threat is to try to avoid including any domain-specific limitations in our general approach, but this does not entirely remove the threat. So, while we currently see no reason why the approach should not also be usable in other companies and other safety-critical domains, an interesting point for further research is to have a wider group of experts to try out the tool.

Hence, we cannot foresee any serious threats to validity for our conclusions on the simulation experiment performed. Also, the feedback for industry experts proved that the KROSA tool has sufficient merit for application in an industrial setting.

## 6.8 Related Work

Previously a number of attempts to solve some of the problems of HazOp analysis have been reported in the literature [1, 3, 7]. A significant number of HazOp expert systems and HazOp system prototypes have been reported in [4]. These include HazOpEX, Batch HazOpExpert, HazOp Diagraph Model (HDG), STOPHAZ, OptHazOp, EXPERTOP, HazOpTool and COMHazOp. A common trend for all of these attempts is that their implementation and application were focussed on the chemical process industry (CPI), the domain where HazOp originated. Also, they were essentially rule-based expert systems and were not designed to facilitate the reuse of experience [4]. Relatively few other automated tools for HazOp in other domains have been reported in the literature [4]. This situation possibly reveals the fact that the HazOp procedure in most cases is done manually but aided by the use of spreadsheet software packages such as MS Excel and Lotus 1-2-3 in many application domains.

It is only recently that case-based reasoning was introduced into HazOp and few efforts have been reported so far. Sahar et al. in [6] presents a report on development of a HazOp analysis management system with dynamic visual model aid. The system is based entirely on CBR with no ontology support for HazOp. In [7], a case-based expert system for automated HazOp analysis called PHASUITE was developed. The PHASUITE system caters to the modification of existing HazOp models and creation of new ones based on the knowledge in existing models. It is also equipped with diagnostic reasoning capability and is suitable mainly for process generic HazOp. It makes use of a suite of informally specified ontologies. PHASUITE is specialised for application in the chemical industry domain. The PetroHazOp [1] has specific application for the chemical domain and was developed to cater to both process generic and non-process generic HazOp. The system uses an integration of CBR and ontology for the automation of both process generic and non-process generic HazOp procedures.

The PetroHazOp [1] and PHASUITE [7] systems are the ones most related to our work since they are based on integration of CBR and ontology. However, none of them have the capability for HazId based on requirements nor are they designed to have any bearing or relevance to requirements engineering as conceived by our approach. Additionally, unlike the two aforementioned tools that are specialised for the chemical industry domain, our approach is a generic one that can be adapted to support several types (process, software, human or procedure) of HazOp analysis in different application domains, given the existence a relevant domain ontology. Hence, the novelty of our approach is the attempt to enable early identification of systems hazards right from the requirements engineering phase of system development and the reuse of experience in order to reduce the amount of resources needed for HazOp. The core idea of this chapter has been reported in [27] in abridged form.

## 6.9 Conclusion

This work offers support for knowledge management in systems engineering at two levels. Firstly, at the level of requirements, it facilitates the exploitation of knowledge contained in requirements documentation for early identification of potential system hazards. The novelty of this is the provision of tool-based support for safety analysis at an earlier phase of system development as compared to previous efforts that focus only on the design phase. Secondly, our approach facilitates the reuse of experience in the conduct of HazOp so that previously documented HazOp knowledge can be leveraged for reduced effort in new projects.

Specifically, we have provided a tool that can creditably assist, but not replace the human expert in the conduct of HazOp analysis so as to attain reduction in effort needed. Considering the fact that HazId is a highly creative process that depends on the experience and skill of the human domain expert, the KROSA tool would be vital as a good starting point. Also, from the results of the evaluation, KROSA has demonstrated a good potential for application in an industrial context. The tool would particularly be helpful in situations where highly skilled or experienced HazOp experts are not available, by enabling a platform whereby previously documented cases can be reused in new scenarios by a less-experienced HazOp team.

In further work, we intend to realise the objective of an extensive semantic framework for safety analysis by extending the features of KROSA to support FMEA. We will also investigate the prospects of providing diagnostic reasoning over potential hazards in order to facilitate a more elaborate automated safety analysis. In addition, we aim to conduct more extensive industrial case studies on safety analysis of systems and product lines using the tool and to report our findings subsequently.

**Acknowledgements** We appreciate the contributions of the staff of system safety research division ABB Norway in conducting trial evaluation of the KROSA tool.

## References

1. Zhao J, Cui L, Zhao L, Qui T, Chen B (2009) Learning HAZOP expert system by case-based reasoning and ontology. Comp Chem Eng 33(1):371–378
2. Dittman L, Rademacher T, Zelewski S (2004) Performing FMEA using ontologies. In: The 18th international workshop on qualitative reasoning. North-western University, Evanston

3. Smith S, Harrison M (2005) Measuring reuse in hazard analysis. Reliab Eng Syst Safe 89(1):93–104
4. Dunjo J, Fthenakis V, Vilchez J, Arnaldos J (2010) Hazard and operability analysis: a literature review. J Hazard Mater 173(1–3):19–32
5. Mokos K, Meditskos G, Katsaros P, Bassiliades N, Vasiliades V (2010) Ontology-based model driven engineering for safety verification. In: Proceedings of 36th EUROMICRO conference on software engineering and advanced applications, Lille, pp 47–54
6. Sahar B, Ardi S, Kazuhiko S, Yoshiomi M, Hirotsugu M (2010) HAZOP management system with dynamic visual model aid. Am J Appl Sci 7(7):943–948
7. Zhao C, Bhushan M, Venkatasubramanian V (2005) PHASUITE: an automated HazOp analysis tool for chemical processes Part I: knowledge engineering framework. Proc Safe Environ Prot 83(B6):509–532
8. Kolodner J (1992) An introduction to case-based reasoning. Artif Intell Rev 6(1):3–34
9. Gruber T (1993) A translation approach to portable ontologies. Knowl Acquis 5(2):199–220
10. Jurafsky D, Martin JH (2008) Speech and language processing: an introduction to natural language processing. Speech recognition and computational linguistics, 2nd edn. Prentice-Hall, Upper Saddle River
11. Redmill F, Chudleigh M, Catmur J (1999) System safety: HazOp and software HazOp. Wiley, New York
12. Shepperd M (2003) Case-based reasoning and software engineering. In: Aurum A (ed) Managing software engineering knowledge. Springer, Berlin
13. Kotis K, Vouros G (2005) Human-centered ontology engineering: the HCOME methodology. Knowl Inform Syst 10(1):109–131
14. Happel HJ, Maalej W, Seedorf S (2010) Applications of ontologies in collaborative software development. In: Mistrík I et al (eds) Collaborative software engineering. Springer, London
15. Lohmann S, Heim P, Auer S, Dietzold S, Riechert, T (2008) Semantifying requirements engineering – the softWiki approach. In: I-SEMANTICS, Graz, pp 182–185
16. Dzung DV, Ohnishi A (2009) Ontology-based reasoning in requirements elicitation. In: Proceedings of seventh international conference on software engineering and formal methods, Hanoi, pp 263–272
17. Lee Y, Zhao W (2006) An ontology-based approach for domain requirements elicitation and analysis. In: Proceedings of the first international multi-symposiums on computer and computational sciences, Hanzhou, Zhejiang
18. Shibaoka M, Kaiya H, Saeki M (2007) GOORE: Goal-oriented and ontology driven requirements elicitation method, ER workshops 2007, Auckland, Lecture notes in computer sciences 4802, Springer, Heidelberg, pp 225–234
19. Maiden N, Sutcliffe A (1993) Case-based reasoning in software engineering. In: IEEE colloquium on case-based reasoning, London, pp 2/1–2/3
20. Jani M (2010) Applying case-based reasoning to software requirements specifications quality analysis system. In: Proceeding of 2nd international conference on software engineering and data mining (SEDM), IEEE Press, Chengdu, pp 140–144
21. Daramola O, Moser T, Sindre G, Biffl S (2012) Managing implicit requirements using semantic case-based reasoning. In Regnell B, Damian D (eds) REFSQ 2012, Lecture notes in computer sciences 7195, Springer-Verlag, Berlin/Heidelberg, pp. 172–178
22. Hull E, Jackson K, Dick K (2004) Requirements engineering. Springer, London
23. Stålhane T, Omoronyia I, Reichenbach F (2010) Ontology-guided requirements and safety analysis. In: Proceedings of international conference of emerging technology for automation, Tampere
24. Aamodt A, Plaza E (1994) Case-based reasoning: foundational issues methodological variations, and system approaches. Artif Intell Commun 7(1):39–59
25. Pedersen T, Pakhomov S, Patwardhan S, Chute C (2007) Measure of semantic similarity and relatedness in biomedical domain. J Biomed Inform 40(3):288–299

26. Ferguson A, Bridge D (2000) Generalised weighting: a generic combining form for similarity metrics. In: Proceedings of the 11th Irish conference on artificial intelligence & cognitive science (AICS'2000), pp 169–179
27. Daramola O, Stålhane T, Sindre G, Omoronyia I (2011) Enabling hazard identification from requirements and reuse-oriented HAZOP analysis. In: Proceeding of 4th international workshop on managing requirements knowledge, IEEE Press, pp 3–11
28. Daramola O, Oladipupo O, Musa A (2010) A fuzzy expert system tool for personnel recruitment. Int J Bus Inform Syst 6(4):444–462
29. Salim M, Villavicencio A, Timmerman M (2002) A method for evaluating expert system shells for classroom instruction. J Indust Technol 19(1):Nov 2002–Jan 2003
30. Burton-Jones A, Storey V, Sugumaran V, Ahluwalia P (2005) A semiotic metrics suite for assessing the quality of ontologies. Data Knowl Eng 55:84–102
31. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in software engineering: an introduction. Kluwer, Norwell

# Chapter 7
# Knowledge-Assisted Ontology-Based Requirements Evolution

**S. Ghaisas and N. Ajmeri**

**Abstract** Reaching common level of understanding of a problem domain is one of the key challenges that stakeholders face during the requirements phase of a project. The stakeholders involved in requirements engineering (RE) attempt to achieve this goal through communication and knowledge sharing. The process of clarifying business problems and arriving at a specification necessitates developing a common vocabulary, assigning meanings to various business concepts, determining their interrelations, and reconciling stakeholders' viewpoints. Oftentimes, knowledge exists in organizations, but is not maintained in a reusable form. To address the knowledge and collaboration needs of RE stakeholders, we have developed a Knowledge-Assisted Ontology-Based Requirements Evolution (K-RE) method and toolset. We demonstrate creation of a knowledge repository and its reuse in two contexts: (1) to resolve change requests (CR) with better speed and accuracy and (2) to jump-start a new project. We combine the social software principles and semantic Web concepts to achieve this.

## 7.1 Introduction

The aim of requirements engineering (RE) is to collaboratively evolve the initial uncertain and ambiguous understanding of a business problem into features and attributes of a software system. Reaching a common level of understanding of a problem domain is one of the key challenges that the software vendors and customers face during requirements definition. The process of articulating and clarifying business problems

S. Ghaisas (✉)
Tata Consultancy Services, Pune, India
e-mail: smita.ghaisas@tcs.com

N. Ajmeri
North Carolina State University, Raleigh, NC, USA
e-mail: najmeri@ncsu.edu

and arriving at a specification based on a shared understanding requires exchange and transfer of knowledge. This necessitates developing a common vocabulary, assigning meanings to various business concepts, determining their interrelations, and reconciling multiple viewpoints from stakeholders. It also involves continually accommodating changes in the shared understanding by verifying it periodically. The challenge has become even more daunting of late because (1) software systems increasingly find applications in ever-widening diversity of domains, and (2) complex and conceptually nontrivial systems are developed by globally distributed teams of stakeholders.

RE stakeholders need knowledge from different perspectives. Customers need to see a tangible evidence of domain knowledge in an organization so that they feel confident that their requirements would be met. Requirement analysts need the domain knowledge to deliver good-quality requirements efficiently [1]. Subject matter experts (SMEs) would like to optimize the time they spend with requirement analysts. Oftentimes, knowledge exists in organizations, and it is estimated that more than 50 % of requirements knowledge for similar systems can be reused completely or with minimal modification [2]. But it is not visible and accessible easily because it may reside in a tacit form with people. Even if it is explicit in the form of documents or Web pages, it is difficult to refer across the multiple disparate knowledge fragments and draw useful inferences from them. IDC report shows that the Fortune 1,000 enterprises waste $5 billion annually due to intellectual rework and the inability of finding electronic resources within the enterprise [3]. In other words, knowledge is not structured to be reusable.

Due to the distributed nature of projects, there is little or no opportunity for colocated discussions among the stakeholders. This is a threat to the success of a project. Respondents to a survey hold communication as one of the top challenges. They express that it is extremely important for the distributed teams to "use the same language" while defining requirements [4]. Empirical research reported in [5] mentions "lack of common language/terminology" as one of barriers to sharing the understanding of a problem domain. The success of "social software" in achieving an effective communication has stimulated use of social principles in RE [6–8]. As a result, we see various tools weaving in collaboration into the RE process see, e.g., [9]. However, although the benefits of social platforms are valuable, they are *necessary* and not *sufficient* in themselves for an effective communication. For example, the communication using the same language quoted above is not possible with just a support for collaboration, unless requirement analysts can "see" and access domain knowledge easily and use it to discuss requirements of their new projects. They should be able to tailor the knowledge to suit their project-specific context as well.

To address the knowledge and collaboration needs of RE stakeholders, we have developed a Knowledge-Assisted Ontology-Based Requirements Evolution (K-RE) method and tool. We combine the social software principles [6–8] and semantic Web concepts to enable a knowledge-assisted RE [10].

The approach involves Knowledge-Assisted Requirements Evolution from a generic knowledge base (KB). The KB consists of requirements knowledge elements such as business constraints, features, business processes, use cases, and data models.

In collaboration with the customers and domain experts, requirement analysts can modify and enhance the knowledge elements to suit project-specific needs. Each new RE exercise thus becomes a guided evolution of a generic KB so that it meets project-specific needs. This is characteristically different from the traditional "clean slate" RE approach, hence the term Knowledge-Assisted Requirements Evolution. The just-in-time, context-sensitive assistance based on semantic Web ontologies, knowledge elements, and inference rules operating on them serves as a guidance and moderation mechanism. This complements the collaborative identification, discussion, and definition of requirements enabled by the underlying social platform. K-RE has three roles – requirement analysts who consume knowledge, knowledge contributors who capture and structure domain knowledge, and knowledge curators who review contributions made by other experts to maintain the currency and correctness of knowledge.

We have deployed K-RE in three pilots and have evaluated its usefulness. In this chapter, we present the results of applying K-RE to demonstrate its benefits in the context of knowledge reuse in a large distributed insurance project.

By weaving in domain knowledge seamlessly into an RE process, we respond directly to the call of RE community [11] for a continued research into RE process improvement. Both the well-documented high cost of requirements-related problems [12–15] and the benefits of improvements in RE processes [16] serve as a motivation for our work.

The remainder of this chapter is organized as follows. In Sect. 7.2, we describe how social platforms and semantic Web influence RE. Section 7.3 details the solution approach and underlying model. Section 7.4 describes the process of knowledge representation and reuse. Section 7.5 details K-RE tool and illustrates its usage. In Sect. 7.6, we present the results of deploying K-RE in two different industrial settings. Section 7.7 presents the conclusion, and Sect. 7.8 talks about the future work.

## 7.2 Social Platforms, Semantic Web, and RE

Social platforms and the semantic Web [17, 18] have influenced software engineering significantly [6–8]. The emerging social software engineering discipline is about enabling community-driven creation, management, and deployment of software by applying methods, processes, and tools in online environments [6, 19–21]. In this section, we highlight some relevant characteristics of the two paradigms in the context of knowledge reuse in RE. We address issues such as self-organization versus moderation, democratic voting versus weighted voting, bottom-up folksonomies versus top-down taxonomies, and semantically enriched and hence more meaningful and effective collaboration.

The social nature of Web 2.0 has been credited with democratizing knowledge content. The same democratic aspect also assigns ownership and responsibility to the content creators. RE stakeholders can use the democracy to identify, debate, and define requirements collaboratively and determine what parts of existing domain knowledge can be reused. The highly transparent communication that Web 2.0 platforms enable can contribute constructively to exchange of ideas and healthy criticisms. A wiki-like platform for entering, editing requirements, and deliberating them openly is therefore highly suitable for RE.

However, a platform meant for RE is likely to benefit from supervision and moderation by requirements experts. Lohmann et al. [6] note that supervision and moderation by requirements experts remains crucial to a project's success. They emphasize that the moderation should be unobtrusive. This seemingly defeats the purpose of a "social" platform, but the comments and discussions are for all to see, and no single heavy-weight stakeholder can unfairly overrule valid suggestions made by even junior stakeholders, or they would face pressure from other experts in the community. Such a moderation can be in the form of a context-specific semantic assistance built into the RE process. An underlying knowledge framework of semantic Web ontologies and inference rules can be employed to select relevant parts of domain knowledge for reuse and provide just-in-time context-specific suggestions to collectively evolve a requirement specification.

Social platforms provide for voting mechanisms. A social platform meant for RE however should take into account the opinions of stakeholders such as domain experts with a higher weight than less experienced stakeholders in an organization. Roles that incorporate suitable weights for respective stakeholders can be useful for this purpose. For example, a "knowledge contributor" role has a higher weight than a "requirement analyst" role but a lower weight than a "knowledge curator" role. We note here that this approach is intended to ensure a meritocratic treatment of participants' opinion. Hence, it is the responsibility of project managers to assign higher weights to the opinions expressed by knowledgeable participants such as domain experts. If instead the weights are decided based on hierarchy alone and without making knowledge the central criterion for a higher weight, the meritocratic purpose will be obviously defeated.

Folksonomies [22] that result out of free-form tagging have emerged as a way of organizing knowledge on social platforms. RE stakeholders can arrive at a shared understanding of a domain using this mechanism. Folksonomies are easy to create and hence popular, but they lack structure. Also, if some concepts are not tagged using the right term, it is hard to search and detect them. A collaborative RE platform however cannot be entirely free of structure. A foundational structure in the form of a predefined taxonomy of RE concepts and domain-specific terminology can be valuable to its stakeholders in arriving at a shared understanding of a domain. Providing a user interface that lets one use predefined concepts and add new concepts easily should strike the right balance between ease of use associated with free-form tagging and rigor associated with structure, semantic precision, and synonym control. Semantic Web ontologies [23, 24] are a way of rendering such a structure to the platform.

The social software and semantic Web complement each other and can enhance the effectiveness of knowledge reuse in an RE process. Ankolekar et al. [7] emphasize that in fact both the streams need elements from the other to overcome respective limitations. They hold that semantic technologies bear a great potential of providing a robust and extensible basis for Web 2.0 applications.

The social software platforms, through their democratic spirit, serve to connect humans (e.g., identifying experts in the RE community), whereas the semantic Web ontologies provide a mechanism to assign unambiguous meaning to vocabularies and link them. Using semantic Web ontologies, we can assign distinct, persistent URIs to each term and relationship (e.g., insurance domain-specific concepts and their relationships). Therefore, linking them with each other and with other Web resources is easy. Social platforms and semantic Web thus provide distinct and yet complementary network effects. Combining the two paradigms in the context of knowledge, reuse in RE is a compelling way to increase their total value significantly.

## 7.3 Foundations of K-RE

As discussed earlier, while adopting the social software principles to a highly specialized field such as RE and requirements knowledge reuse, we need to complement the social aspects with semantics [6, 7].

K-RE organizes knowledge along four distinct contexts: (1) environment, (2) problem domain, (3) generic requirements, and (4) RE process. The semantic assistance in K-RE comes from the inference rules operating on the four ontologies that represent these knowledge contexts. The framework also incorporates abstractions from various knowledge modeling paradigms like feature models [25], business process models [26], data models, and use case models [27], to capture and organize knowledge elements.

### *7.3.1 Solution Architecture: Ontologies as an Underpinning Framework for Requirements*

The four ontologies in K-RE – “Environmental Context Ontology,” “Generic Requirements Ontology,” “RE Process Ontology,” and “Problem Domain Ontology” – are created using RDF-OWL schema [23, 24]. Figure 7.1 shows example instances of the ontologies depicted using the UML class diagram notation.

#### 7.3.1.1 Environmental Context Ontology

This ontology is designed to capture the environment for which software requirements are to be defined. For example, a requirement analyst may want to capture requirements for a ***Business Process*** – *Member Enrolment of a Pension* application for a ***Customer*** *Europe Company1* in the *Europe Country1* ***Geography***. The abstractions ***Actor***, ***Action***, ***Domain***, ***Line of Business***, ***Customer***, and ***Geography*** are used to capture the information.

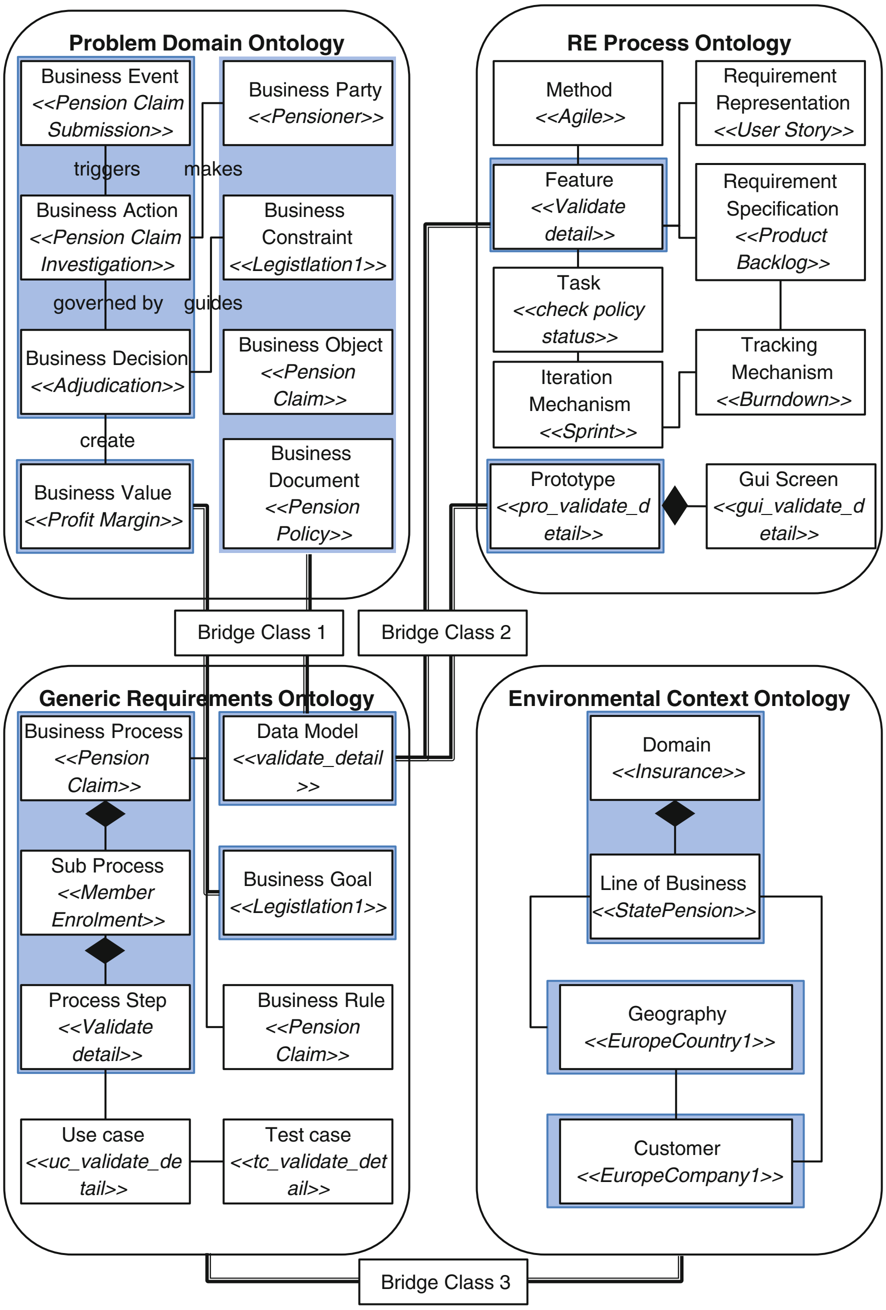

Problem Domain Ontology
Business Event <<Pension Claim Submission>>
Business Party <<Pensioner>>
triggers
makes
Business Action <<Pension Claim Investigation>>
Business Constraint <<Legistlation1>>
governed by
guides
Business Decision <<Adjudication>>
Business Object <<Pension Claim>>
create
Business Value <<Profit Margin>>
Business Document <<Pension Policy>>
RE Process Ontology
Method <<Agile>>
Requirement Representation <<User Story>>
Feature <<Validate detail>>
Requirement Specification <<Product Backlog>>
Task <<check policy status>>
Tracking Mechanism <<Burndown>>
Iteration Mechanism <<Sprint>>
Prototype <<pro_validate_detail>>
Gui Screen <<gui_validate_detail>>
Bridge Class 1
Bridge Class 2
Generic Requirements Ontology
Business Process <<Pension Claim>>
Data Model <<validate_detail>>
Sub Process <<Member Enrolment>>
Business Goal <<Legistlation1>>
Process Step <<Validate detail>>
Business Rule <<Pension Claim>>
Use case <<uc_validate_detail>>
Test case <<tc_validate_detail>>
Environmental Context Ontology
Domain <<Insurance>>
Line of Business <<StatePension>>
Geography <<EuropeCountry1>>
Customer <<EuropeCompany1>>
Bridge Class 3

**Fig. 7.1** Example KB instances and bridge classes that enable context-specific recommendations

#### 7.3.1.2 Problem Domain Ontology

This ontology provides abstractions to capture the essence of the problem domain. For example, consider the following scenario – *In event of death of a pensioner, a beneficiary may submit a claim request*. The abstractions such as ***BusinessEvent, BusinessType, Party***, and ***BusinessAction*** are used to capture this information.

#### 7.3.1.3 Generic Requirements Ontology

The KB that we present to the requirement analyst is created in terms of requirements knowledge elements such as ***business goals, features, business processes*** and ***subprocesses, business constraints*** (laws of the land, organizational policies), ***use cases,*** and ***business entities***. The Generic Requirements Ontology provides these abstractions [[28] and references therein].

#### 7.3.1.4 RE Process Ontology

This contains abstractions specific to the RE process, for, e.g., *Agile* ***Method*** has ***Requirement Representation*** in form of *User Story*, ***Iteration Mechanism*** as *Sprint,* and ***Tracking Mechanism*** as *Burndown*.

### *7.3.2 Examples of Mappings Between the Elements of Different Ontologies*

- The ***BusinessEvents*** (e.g., *Pension Claim Submission*), ***BusinessActions*** (e.g., *Investigate Pension Claim*), and ***BusinessDecisions*** (e.g., *Adjugation*) in the *Problem Domain Ontology* are represented as ***BusinessProcess*** (e.g., *Pension Handling*) in the *Generic Requirements Ontology*.
- ***BusinessConstraint*** (e.g., a *New legislation#123*) in the *Problem Domain Ontology* maps to ***Validation*** (e.g., *Verify conformance to legislation#123*) in *Generic Requirements Ontology*.
- The ***BusinessParty*** (e.g., *Member*), ***BusinessObject*** (e.g., *Pension Claim*), and ***BusinessDocument*** (e.g., *Pension Policy*) from the *Problem Domain Ontology* contribute to ***DataElement*** in the *Generic Requirements Ontology*.
- ***Feature*** (e.g., *Validate Member Details*) in *RE Process Ontology* maps to ***SubProcess*** (*Member Enrolment*) in *Generic Requirements Ontology*.

The semantic assistance is achieved by employing the "bridge classes" and inference rules written in the Semantic Web Rule Language (SWRL) [29]. The bridge classes specify semantic mappings between the four ontologies. We define rules that refer the ontology instances and provide recommendations based on the integrated inference. The recommendations may be specific to a singular ontology or may span across the four ontologies when necessary, in response to actions of the requirement analyst.

For example, if a requirement analyst selects *Europe Country1* as the ***geography*** for a *Pension application* to be developed, she would be presented with ***features*** and *user stories* relevant to *Insurance Pension* ***Processes*** from the KB. As she configures them in the context of her project, she would be presented with ***business rules***, in the given ***geography***, e.g., *Pension rules* in *Europe Country1* (*Environmental Context Ontology* and *Problem Domain Ontology*). If she selects ***features*** that complement each other but decides to associate them with different *sprints*, she would receive a recommendation to preferably rearrange them in the same *sprint* (*Problem Domain Ontology* and *RE Process Ontology*). If she adds a new ***feature*** to the *Product backlog* upon the customer's suggestion and it happens to conflict with an already selected ***feature*** in a given domain, she would be alerted about the inconsistency of her selection (singularly the *Problem Domain Ontology*). Requirement analysts can improve the completeness, correctness, and consistency of requirements using the inbuilt domain knowledge served to them in the form of just-in-time alerts.

Using the collaboration mechanisms in K-RE, dispersed teams can interact informally. Moreover, K-RE provides for a semantically enriched collaboration that would foster meaningful and focused discussions on topics in requirements engineering in general and problem domain such as insurance in particular. The stakeholders can even evolve the ontologies collaboratively. A user is at liberty to identify new concepts as and when necessary. This constitutes the folksonomy which evolves bottom-up in a community-driven way. If their usage in the community of practice (in this case, RE stakeholders) is substantial, the concepts can be absorbed into the taxonomy. This decision can be made collaboratively and with advice from domain experts.

## 7.4 Process for Knowledge Representation and Reuse

In this section, we illustrate capturing of knowledge from requirements documents and Web sites and mapping the knowledge elements onto the K-RE model.

### *7.4.1 Knowledge from Documents*

Knowledge representation is done in three steps: (1) identification of structural details of sources such as requirements documents and Web sites, (2) mapping of the document structure to the K-RE model, and (3) extraction of domain knowledge from documents and its representation in K-RE.

#### 7.4.1.1 Identification of Structural Details

Each project has its own set of templates to organize requirements knowledge. In this step, we identify the details of the document structure such as headings, sections, and subsections. For example, a document may contain a section to list business rules. We note that this section needs to be mapped to ***BusinessConstraint*** in the K-RE model in the next step. We use the K-RE model as a reference to identify missing structure details and knowledge elements in the requirements document. For example, K-RE model has a notion of ***SystemUsecase***, which may not be present in a project's requirements specification.

#### 7.4.1.2 Mapping of Document Structure to K-RE Model

Knowledge contributor defines the scope of the KB by setting the environment parameters (e.g., ***Domain***, *Insurance*; ***Line of Business***, *State Pension; **Geography**, Europe Country1*; ***Customer***, *EuopreCompany1*). The knowledge contributor maps the structural details identified in the previous step to various concepts in the K-RE model. For instance, ***subprocess*** or ***functionality*** in requirement documents maps to the concept ***feature*** in K-RE. This is a semiautomated and semantically assisted process; points of human intervention are identified explicitly. K-RE prompts for inputs from users where necessary. For example, if a subsection such as ***overrider*** in a ***use case*** cannot be mapped to any concept in the model, K-RE detects that it is unmapped and the user is asked to either map it to some existing concept or define a new concept that accommodates the detail. The structural details thus extracted correspond to the Generic Requirements Ontology (Sect. 7.3.1).

#### 7.4.1.3 Domain Knowledge Extraction and Representation

We have observed the following common problems in requirements documents:

- Granularity of domain knowledge elements such as business processes and use cases is unspecific.
- Business constraints are not explicitly documented; they are often embedded within the business processes – e.g., in alternate flows.
- Business process steps are not always associated explicitly with actors who perform them.
- Manual business process steps are not explicitly differentiated from the automatable ones.
- There is no uniformity in the usage of business terms.

K-RE aims to reduce these ambiguities by providing well-defined abstractions using the underlying ontological model and a semantic assistance to organize domain knowledge.

K-RE detects terms and key phrases specific to a given domain (such as insurance in our case). The detected key phrases are instances of concepts in the ontologies. For example, a term such as *Pension Policy* is a concept – instance of ***BusinessDocument*** in the Problem Domain Ontology. The knowledge contributor is presented with the concepts from the Problem Domain Ontology and is required to map the concept instances extracted from the document to the concepts in the ontology. The identified concept instances are then parsed to detect similarity mappings. The techniques employed are lexical similarity [30], semantic similarity [31], direct string matching, and ontological structure-based mapping [23, 24]. If concept instances that do not exactly correspond to any existing concepts in the ontology are identified, the knowledge contributor can define new concepts or reconcile them with the existing concepts if possible.

The concept instances are also parsed to detect associations between concept instances (e.g., *member participates in scheme*). Relations like subclass, super class, equivalent, part of, and concepts related with each other by minimum cardinality of one on both sides are considered. The identified concept instances and associations between them are subject to refinements by human intervention.

If a key phrase is identified as a feature, the knowledge contributor is asked to specify complementary feature(s) and conflicting feature(s) from a list of available features in the existing KB. She can also add new complementary/conflicting features to the KB (e.g., *Retirement Benefit Option to Purchase Annuity* is followed by *Decumulation (disinvestment of funds)*). The abstractions to capture complementary and conflicting features take cues from "requires" and "excludes" relationships in feature modeling [25].

If a key phrase is identified as a use case, the knowledge contributor specifies actor (s) from the available list or adds new ones to the KB. She also identified "includes" and "extends" use cases for a given use case from the KB or adds new ones. Each sentence is processed, and all sentences in passive voice are converted to active voice. We use triplet extraction to identify part of speech [32] in form of subject-predicate-object. The subject refers to the performing actor, and the predicate and the object pair in the verb phrase refers to the use case. These abstractions are captured as per the use case model [27].

Business constraints (e.g., *Member's Benefit Age should be the age notified by member to the Scheme Administrator*) typically contain terms such as ***should, must, if, if-else, and only-if***. Key phrases containing these terms are presented to the knowledge contributor so that she can map them as instances of ***BusinessConstraints***. Whenever a business constraint is identified, the features and the use cases that are affected by the constraint are specified. Complementary and conflicting constraints are specified. The constraints are then classified into ***rules*** that are domain specific, ***laws*** that are locale specific, and ***policies*** that are company specific.

The domain knowledge elements thus extracted map to the Problem Domain Ontology (Sect. 7.3.1) and other knowledge models. The knowledge curator validates the correctness and currency of the KB created using the process.

### *7.4.2 Knowledge from Domain-Rich Web Sources*

Organizations sometimes venture into new domains and need to build KBs from external sources. One of the prominent external sources today is the Web. To accumulate knowledge from the Web, K-RE employs a Web crawler developed in-house to explore various resources. We do a targeted crawling for Web sites whose patterns are known. If the pattern or template of the Web site is not known (or if the Web source does not follow a fixed template), we prune the html tags and parse only the text to extract concept instances. The mapping of the Web page pattern to K-RE model and the process of identifying the concept instances, associations, features, use cases and business constraints, and validation by knowledge curator remains the same as the one described in Sect. 7.4.1.

### *7.4.3 Knowledge Reuse and Upkeep*

The structured KB enables reuse of knowledge while defining requirements of a new application in the same domain. A suitable instance of KB is made available to requirement analyst as per the project environment.

A requirement analyst who wants to work on a new Pension project starts with selecting environmental parameters. Based on the selected environmental parameters, she is presented with a core set of features from the KB that matches the parameter selection (e.g., *State Pension*). As she proceeds to select the relevant features, she is recommended to include the complementary features and avoid the conflicting ones. Based on the feature selection, she is presented with the relevant business rules and business glossary associated with the selected feature.

If requirement analysts learn some new information about a domain, they can add/edit/remove the knowledge elements from the repository as well. K-RE identifies the usage patterns of the requirement analysts and makes them visible to the knowledge curator.

## 7.5 K-RE: A Tool for Knowledge-Assisted Collaborative Requirements Evolution

In this section, we describe the K-RE tool. We also illustrate its usage with an example.

### *7.5.1 Overview*

The tool provides a wiki-like user interface for knowledge contributor and knowledge curator to contribute domain knowledge and for requirement analyst to access and configure the knowledge:

- *Knowledge contributor:* An experienced requirement analyst who contributes generic and specific domain knowledge to the repository.
- *Knowledge curator:* As a subject matter expert, the curator monitors and ensures quality of knowledge. She ensures that the knowledge content is correct and current. Curator acts as the reviewer and monitors the contributions made by the knowledge contributor.
- *Requirement analyst:* End user of the K-RE who configures available domain knowledge as per the project environment parameters and scope.

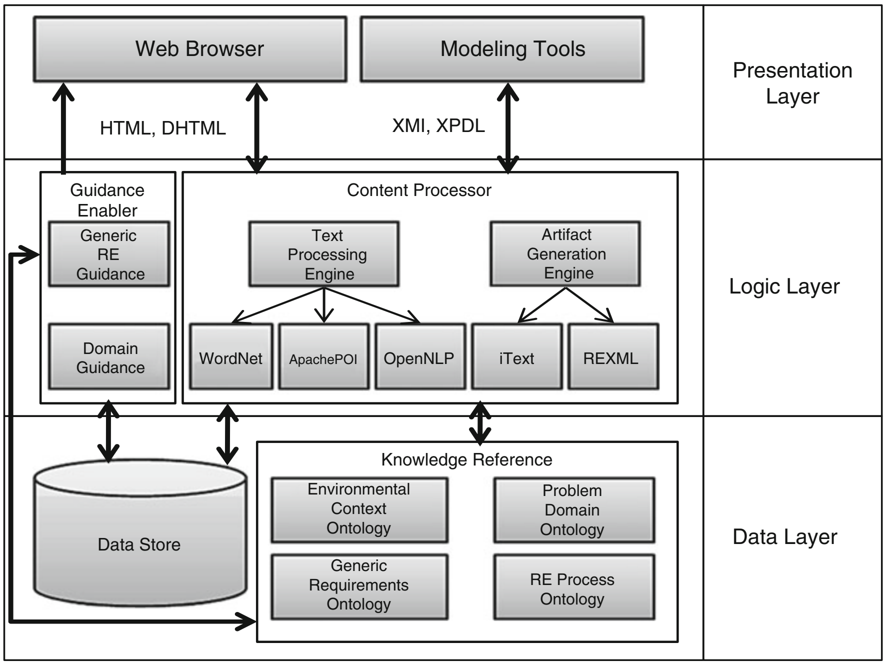


**Fig. 7.2** K-RE architecture

## 7.5.2 *Architecture*

K-RE is a Web-based tool with a centralized application server and database server accessible to multiple clients over the Internet. To store knowledge models and their instances, K-RE uses RDF-OWL [23, 24] ontologies and a relational data store [33]. The design incorporates collaborative aspects of Web 2.0 for a participatory information sharing and collaboration among the RE stakeholders. The semantic assistance provided by K-RE uses OpenNLP – NLP toolkit [34], WordNet lexical database [35], and RDF-OWL ontologies along with SWRL rules.

Figure 7.2 shows the tool architecture. The presentation layer serves as the interface between the tool and the user. The logic layer incorporates a guidance enabler and content processor. The user requests, sent using the presentation layer, are processed by the logic layer. The data layer includes the knowledge reference module to represent knowledge in form of ontology concepts and instances.

## 7.5.3 *Usage Illustration*

Table 7.1 shows K-RE activities and knowledge assistance with examples.

In addition to those illustrated in Table 7.1, K-RE enables the following:

**Table 7.1** K-RE activities and knowledge assistance

| K-RE activities | Knowledge assistance from *State Pension* KB | Example(s) | Formalized/human intervention |
|---|---|---|---|
| Select environmental parameter | A KB relevant to the selected parameters is presented | Parameters: ***Domain*** (e.g., *Insurance*), ***Line of Business*** (e.g., *State Pension*), ***Geography*** (e.g., *Europe Country1*), and ***Customer*** (e.g., *Europe Company1*)<br>KB including ***processes*** such as *Member Enrolment, Contribution, Alterations, Settlement, and Exits* presented | Formalized |
| Select feature | Domain-specific recommendation of complementary and conflicting nature of the features | Selected ***feature***: (1) *Early Retirement due to serious ill-health*, (2) *Over Maximum Retirement Age Processing*<br>Recommendation to reconsider the selection due to conflicting nature of features<br>Underlying ***business rule***: *A member is eligible for Early Retirement due to serious ill-health before the Maximum Retirement Age* | Formalized |
| Edit feature | Domain and geography-specific recommendation to include relevant business rules/policies, business terms | Selected ***feature***: *Early Retirement due to serious ill-health*<br>***Business term***: *Deferred Retirement*<br>***Rule***: *Laws of land related to Early Retirement* | Formalized |
| Edit feature step | Domain/geographic/context-specific recommendation for most agreed-upon business term | Altering ***feature step***: *Check if retirement age is less than Normal Benefit Age*<br>Most agreed-upon term: *Normal Retirement Age*<br>Synonymous term: *Normal Pension Age, Normal Benefit Age* | Formalized + human intervention |
| Specify interactions with external domain(s) | Possible interactions with other domains are presented | Requirement analyst can view possible interactions with other domains and identify | Formalized + human intervention |

(continued)

**Table 7.1** (continued)

| K-RE activities | Knowledge assistance from *State Pension* KB | Example(s) | Formalized/human intervention |
|---|---|---|---|
| | | entry points into relevant processes: *Feature*: Early Retirement due to ill-health Possible *Interfaces* with external domain(s) *Healthcare domain* (e.g., *reviewing member's medical evidences submitted for serious ill-health to confirm that life expectancy of member is less than a year*) *Banking domain* (e.g., *payout processing after Early Retirement claim is approved*) | |

### 7.5.3.1 Semantically Enabled Collaboration

While carrying out any of the activities in Table 7.1, a requirement analyst can start discussions in the form of informal chats on the selected knowledge elements with her colleagues and experts and seek opinion on selections from the KB and refinements to be made. She can post topics for discussions on semantically enabled forums and subscribe to alerts when others post their opinions on topic of her interest.

For example, if she selects the following rule to be included in her specification: *Member's Benefit Age should be the age notified by member to the Scheme Administrator*, and if she is not sure that whether the rule is valid in *Europe Country2*, she can start a forum to discuss this with experts. Upon initiating a forum, she will be presented with a set of relevant posts available on the topic. For example, she can view posts related to validity of rules for *Pension*, rules for *Europe Company1*, rules for *Europe Country2,* posts by other experts who contributed Pension rules, and rules regarding related terms such as date of commencement premium and select the most suitable thread of discussions in terms of topic, author geography, and so on.

### 7.5.3.2 Generating and Refining Artifacts Iteratively

The requirement analyst can generate structured requirements specification documents intermittently. If project follows an *agile* method, she can view *Sprints, Product backlogs,* and *Burndown charts*. She can also populate ***data models*** using modeling tools. The analyst can either work on the "text" or "diagram" and import/export to/from either format. This helps in refining artifacts incrementally.

**Table 7.2** Tasks and effort involved in creating knowledge base

| Task | Effort |
|---|---|
| Analysis and standardization of requirement specification documents | 10 person days |
| Uploading in K-RE | 1 person day |
| Review of extracted knowledge elements | 8 person days |

Starting with the KB for business process *Member Enrolment*, the requirement analyst can thus evolve a specification that suits a given project. The evolution is an assisted exercise that helps in adding to or modifying the KB by providing context-sensitive help to a requirement analyst.

It is relevant here to add that not all of the domain knowledge is formalizable in terms of ontologies and the Semantic Web Rule Language (SWRL). We therefore use a combination of formalization and human intervention to represent knowledge and enable its reuse in RE. The example presented in Table 7.1 indicates points of human intervention.

## 7.6 Evaluation in Industrial Settings

In this section, we present results of deploying K-RE in a large insurance project. The project under discussion is a large Europe Country1 workplace pension scheme. It involves providing scheme administration services to the client.

We illustrate reuse of the structured knowledge in two contexts: (1) change impact analysis in the same project (*Europe Country1*) and (2) while starting a new project (*Europe Country2*) in the same domain.

We created a *State Pension* KB using K-RE from existing requirement specifications. Table 7.2 lists the tasks and effort involved in creating the KB.

The details pertaining to the size and content of the KB are given in Table 7.3. In parentheses against the knowledge elements, we indicate the number of the respective knowledge elements identified in the documents.

The KB was found to be useful in two ways – (1) for resolution of change requests in the same project and (2) as a point of departure for a new Pension project.

### *7.6.1 Change Request Resolution in the Same Project*

For each proposed requirement change request, the requirement analysts carry out the change impact analysis and discuss with other stakeholders the effort involved in implementing the change. Changes in requirements always have a ripple effect [36]. Bohner [37] defines change impact analysis (CIA) as “the activity of identifying the potential consequences, including side effects and ripple effects, of a change, or estimating what needs to be modified to accomplish a change before it has been made.” The effort involved in implementing a change is proportional to the impact and has a direct bearing on cost of the project. The process of change resolution thus benefits from (1) a visible knowledge about impacted requirements elements as a result

**Table 7.3** Knowledge elements from project documents and their mappings to the K-RE model

| Requirement artifacts identified from project documents | Problem area | Mapping details | Corresponding knowledge elements in K-RE |
|---|---|---|---|
| Group (5) | Multiple business processes within a group | Individual processes (e.g., Settlement and Exits, Fund Administration) from a group (e.g., Group 4) were extracted and mapped to a process in K-RE | Process (16) |
| Subgroup (13) | Multiple business subprocesses within a subgroup | Individual subprocesses (e.g., Retirement, Transfers, Death, and Cessation) from a subgroup (e.g., Settlement and Exits) were extracted and mapped to a subprocess in K-RE | Subprocess (52) |
| Functional use case (48)<br>Business flows (333) | Multiple functionalities within a use case | Piece of business functionality (e.g., Early Retirement due to ill-health, Early Retirement due to incapacity) that can execute separately was extracted from the use cases and business flows and mapped to a feature in K-RE (pieces of functionalities from 48 functional use cases and 333 business flows were mapped as 220 features) | Feature (220) |
| Business flow step | Actors not associated with each business flow step | Each feature step was associated with a performing actor | Feature step |
| Business rules(218) | Many business rules embedded in the business flows | 128 additional business rules were identified from the business flows through domain analysis | Business rule (346) |
| Glossary terms (517) | Inconsistency in the usage of business terms | Additional business terms detected from documents. Business terms and the relation between them were identified | Business terms (1,210) |

of the change in a given element and (2) collaboration mechanisms that allow for discussions among requirement analysts, developers, project managers, and customers. K-RE incorporates both these aspects.

To evaluate the usefulness of K-RE, we conducted an experiment to compare the manual change request implementation routinely practiced by the project and one carried out using K-RE. Two separate groups of requirement analysts implemented 30 change requests for the project. (1) Group 1 consisting of seven requirement analysts followed the traditional approach of analyzing the requirement specification documents

**Table 7.4** Requirement change request details

| Requirement details | |
|---|---|
| *Original requirement* | Introduction of a default retirement age: member's benefit age should default to Nominal Benefit Age/State Pension Age |
| *Change request* | Introduction of two Default Retirements Age<br>1(a): Revise the rule of Nominal Benefit Age to automatically adjust the Benefit Age of member. Member's Benefit Age should default to Nominal Benefit Age/State Pension Age. In case the member does not take their benefit and does not tell when they intend to take their benefit, then after the expiration of Nominal Benefit Age/State Pension Age, the Nominal Benefit Age should automatically default to 1 day before the Maximum Retirement Age (i.e., 75)<br>1(b): Change the current definition of "Nominal Benefit Age" and replace the term with "Benefit Age" |

to identify impacted elements, and (2) Group 2 consisting eight requirement analysts implemented the changes using K-RE. The change request implementation was reviewed by five domain experts.

#### 7.6.1.1 Change Request Analysis

Table 7.4 shows an example of the original requirement and the change request.

The proposed requirement change implicitly includes change in *Nominal Benefit Age*-related ***business rules*** and change in the use of some ***business terms***.

#### 7.6.1.2 Traditional Change Request Resolution

Requirement analysts in Group 1 performed the change impact analysis and determined the related requirement artifacts. They used the traceability information available in requirement specification documents.

We noted the following:

- *Total number of impacted knowledge elements identified by domain expert ($IKE_{Total}$):* These are the total number of knowledge elements impacted by the proposed change assuming all knowledge elements have been captured.
- *Total number of impacted knowledge elements identified by requirement analyst manually ($IKE_{Manual}$):* These are actual number of impacted knowledge elements identified by the requirement analyst manually from requirement specification documents.

#### 7.6.1.3 Change Request Resolution Using K-RE

Analysts in Group 2 used K-RE to handle change requests. Using K-RE, they modified the ***business rules*** corresponding to *Nominal Benefit Age* in the ***business process*** *Settlements* and *Exits* and ***subprocess*** *Retirement*. Upon making these modifications, they received alerts to also update associated knowledge elements

**Table 7.5** Illustration of requirement change resolution using K-RE

| Requirement change resolution activities | Guidance from K-RE | Identifying impacted knowledge element |
|---|---|---|
| Update business rule | Domain-specific recommendation to update complementary business rule | Rule#1: "Member's Benefit Age should be the age notified by member to the scheme" |
| | | Complementary rules |
| | | Rule#2: "In the absence of member notification, Member's Benefit Age should be State Pension Age" |
| | | Rule#3: "In the absence of member notification, if member has already attained relevant age then Member's Benefit Age should be immediately before the member attains age 65" |
| | Generic recommendation for requirements completeness to update corresponding validations | Validation#1: Check if member age is less than 65 |
| Update validation | Generic recommendation for requirements completeness to update corresponding task | Task#1: Validate member age with Normal Minimum Retirement Age |
| Update task and feature step | Generic recommendation for requirements completeness about possible impacted knowledge elements associated with feature step | Associated actor, associated triggering feature step, associated nested feature step, associated system use case |

such as complementary ***rules*** and ***validations***. When the ***validations*** were updated, K-RE prompted the analysts to review corresponding ***task*** and ***feature step***. For each ***feature step*** altered, K-RE presented all the knowledge elements associated with it. Table 7.5 illustrates some examples for change request under consideration.

We noted the following:

- *Total number of impacted knowledge elements identified by K-RE (IKE $_{K\text{-}RE}$):* These are the total number knowledge elements identified by K-RE as impact of the proposed change.
- *Total number relevant impacted knowledge elements identified by K-RE (IKE $_{K\text{-}RE\ Rel}$):* These are the impacted knowledge elements that were identified by K-RE and considered relevant by the domain expert.

#### 7.6.1.4 Effectiveness Parameters

In order to measure the effectiveness of requirement change resolution using K-RE and compare it with to the traditional approach, we computed precision and recall values based on the data obtained from the experiment.

***Precision**:* We define precision as "percentage ratio of relevant impacted knowledge elements identified that require change to the total impacted knowledge elements identified."

***Precision ($P_M$)*** is the percentage ratio of impacted knowledge elements identified by the requirement analysts in Group 1 that were considered relevant by the domain expert (to the total number of impacted knowledge elements identified by the requirement analyst).

***Precision ($P_{K\text{-}RE}$)*** is the percentage ratio of impacted knowledge elements identified by the requirement analysts using K-RE that were considered relevant by the domain expert (to the total knowledge elements identified by K-RE).

$$\mathbf{P_{K\text{-}RE} = (IKE_{K\text{-}RE\,Rel}/IKE_{K\text{-}RE}) \times 100}$$

Here, we have assumed that precision ($P_M$) is 100 % because all the elements that the requirement analysts identify are correct. However, requirement analyst may or may not identify all the impacted knowledge elements that require change. We have verified this logical assumption with domain experts.

***Recall:*** We define recall as "percentage ratio of impacted knowledge elements identified to the total actual impacted knowledge elements."

***Recall ($R_M$)*** is the percentage ratio of the relevant knowledge elements identified by the requirement analyst to the total actual impacted knowledge elements.

***Recall ($R_{K\text{-}RE}$)*** is the percentage ratio of the relevant knowledge elements identified by K-RE to the total actual impacted knowledge elements.

$$\mathbf{R_M = (IKE_{Manual}/IKE_{Total}) \times 100}$$
$$\mathbf{R_{K\text{-}RE} = (IKE_{K\text{-}RE\,Rel}/IKE_{Total}) \times 100}$$

#### 7.6.1.5 Results

Table 7.6 lists the parameters discussed earlier.

The plot in Fig. 7.3 depicts the precision and recall value computed for each of the CR handled manually and using K-RE.

*Precision*

Average precision of the K-RE was computed as 90.79 %.

$$P_{K\text{-}RE} = \left(\sum IKE_{K\text{-}RE\ Rel}/\sum IKE_{K\text{-}RE}\right) \times 100$$

$P_{K-RE} = 90.79$ %.

*Recall*

Average recall of the K-RE was computed as 89.61 %.

**Table 7.6** Impact analysis of change requests

| Change request | IKE $_{\text{Total}}$ | IKE $_{\text{K-RE}}$ | IKE $_{\text{K-RE Rel}}$ | IKE $_{\text{Manual}}$ |
|---|---|---|---|---|
| CR 1 | 4 | 5 | 4 | 2 |
| CR 2 | 6 | 7 | 6 | 6 |
| CR 3 | 11 | 12 | 11 | 5 |
| CR 4 | 10 | 10 | 9 | 4 |
| CR 5 | 10 | 10 | 9 | 4 |
| CR 6 | 13 | 11 | 11 | 7 |
| CR 7 | 12 | 11 | 9 | 6 |
| Total | $\sum \text{IKE}_{\text{Total}} = 154$ | $\sum \text{IKE}_{\text{K-RE}} = 152$ | $\sum \text{IKE}_{\text{K-RE Rel}} = 138$ | $\sum \text{IKE}_{\text{manual}} = 80$ |

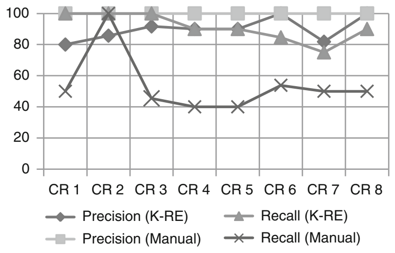


**Fig. 7.3** Precision and recall: manual and K-RE based

$$\text{R}_{\text{K-RE}} = \left(\sum \text{IKE}_{\text{K-RE Rel}} / \sum \text{IKE}_{\text{Total}}\right) \times 100$$

$\text{R}_{\text{K-RE}} = 89.61\ \%$.

Average recall of the traditional manual requirement traceability approach was computed as 51.94 %.

$$\text{R}_{\text{M}} = \left(\sum \text{IKE}_{\text{Manual}} / \sum \text{IKE}_{\text{Total}}\right) \times 100$$

$\text{R}_{\text{M}} = 51.94\ \%$

#### 7.6.1.6 Analysis

Average precision of identifying knowledge elements impacted by the change requests when using K-RE was found to be 90.79 %. If a knowledge element is being changed, K-RE uses the underlying domain knowledge ontology to identify all the related knowledge elements. For example, if it is required to change a ***feature***, the related ***rules***, complementary ***features***, ***use cases***, and ***test cases*** will be highlighted as impacted elements. Only human intervention can discern if all need to be correspondingly updated. For example, not all ***test cases*** will need to be changed if a ***use case*** is being modified. As discussed earlier, we have involved domain experts to review the results of manual identification and identification done by K-RE.

Average recall of the K-RE was computed 89.61 %. Recall using manual approach was found to be 51.94 %. K-RE thus has an inherent ability to make knowledge visible using the domain knowledge ontology it incorporates. The business analysts could not identify as many impacted elements in the absence of such a mechanism. Some of the impacted elements such as ***tasks*** and ***validations*** associated with ***use cases***; complementary features were not obvious to them. K-RE makes the impact easily visible to all stakeholders and hence easy to discuss the effort involved.

## *7.6.2 Starting a New Project Using the Knowledge Base*

A ***Pension*** application for an insurance company in ***Europe Country2*** was to be developed. The KB created for ***Europe Country1*** was reviewed for reuse. From the repository, the knowledge elements mentioned in Table 7.7 were found to be closest to the new project for ***Europe Country2***. These were imported into the new project workspace from the repository.

Eighteen of the selected ***State Pension*** processes were modified. Two hundred ***business entities*** and relations were added afresh to the ones borrowed from repository. Fifty new country-specific variants to ***business rules*** had to be included. The resultant KB was reorganized, modified, and refined in consultation with customers and domain experts, using the Web 2.0 enabled communication in addition to the assistance that K-RE provides. For example, the analyst selected the ***process*** C*ontribution* to work with from the list of available process in the KB. She received a recommendation to add the ***processe****s Customer Management*, *Finance,* and *Accounting*, as these processes are complimentary to the ***process*** *Contribution*. The analyst was also presented a list of features specific to selected process. For example, features such as *member regular contribution by new direct debit, member ad hoc contribution by new direct debit, member adhoc contribution by debit card,* and *member contribution correction* were presented to the analyst from the process *Contribution*. She selected the ***features*** *member regular contribution by new direct debit* and *member adhoc contribution by new direct debit* to add to her project. K-RE prompted her to add ***feature*** *member contribution correction* as well (as feature, *member contribution correction* is complimentary to the other selected features). The requirement analyst made changes to one of the features – *member regular contribution by new direct debit*. She entered a new feature step – *verify member contribution details with respect to present growth rate and send notification to the scheme holder*; K-RE provided an alert that *scheme holder* and *member* are synonymous terms, but *member* is the commonly accepted term. In addition to this, K-RE also parsed the text to identify new ***business terms*** and concepts (e.g., *growth rate*, *contribution*) and recommended the analyst to add them to the glossary. She was also prompted to add relevant ***business rules*** such as *growth rate must be within the range of 8–10 %* and ***validations*** such as *check if the growth rate is within the range of 8–10 %*. Corresponding ***use cases*** as well as ***screens*** were made visible to her to select from.

**Table 7.7** Knowledge elements reuse in project for Europe Country2

| Knowledge elements in knowledge base from *Europe Country1* | Knowledge elements imported from *Europe Country1* for reuse in *Europe Country2* project | Remark(s) | Example |
|---|---|---|---|
| 52 subprocess | All of 52 State Pension subprocesses and BPMN process maps | 18 out of the 52 imported State Pension subprocesses were modified | Some of the modified subprocesses are *Contribution, Joiner, Risk Management, Customer Management* |
| 220 features | All of 220 features | Feature steps modified to accommodate country-specific variations | Original step: *Perform contribution limit check as dictated by Europe Country1* Modified step: *Perform contribution limit check as dictated by Europe Country 2* |
| 346 business rules | 250 business rules | 50 new country-specific variants to business rules included | New business rule such as *in case of Europe Country 2, the member must hold a valid debit card that was added* |
| 1,210 business terms and relationships | All of 1,210 business terms and relationships | Two hundred business entities and relations added afresh to the 1,210 borrowed from repository | Example: New terms such as *growth rate and contribution* were added |

The final requirement specification for the new project consisted of 239 *State Pension* ***processes***, 3,269 ***business entities*** and relations, and 822 ***business rules*** along with exceptions and overrider scenarios. A review by domain experts reveals that 60 % of the knowledge needed to arrive at the baseline specification was reused from the repository created originally for the ***Europe Country1 Pension*** project. In subsequent phases, 20–30 % of time savings were observed.

The projects teams consider this to be a significant contribution to the requirements definition exercise, which otherwise starts from a clean slate for want of visible, accessible, and configurable knowledge. The availability of a structured KB also serves as a "thinking aid" for all RE stakeholders to brainstorm and arrive at a consensus. It is easier to review and modify (if necessary) an existing feature or a process than to come up with one afresh. Customers find it simpler to suggest changes and additions to an existing process or a rule than to narrate one from memory. Requirement analysts can leverage the domain vocabulary to have meaningful discussions with customers and optimize the time spent with SMEs.

## 7.7 Discussions and Conclusion

RE predominantly involves establishing a shared understanding of problem domain. It is estimated that more than 50 % of requirements knowledge for similar projects can be reused completely or with minimal modification [2]. However, knowledge present in tacit form is not amenable to reuse. Even explicit knowledge in the form of disparate documents may not serve the purpose of reuse because it is not structured in a way that makes it visible, accessible, and configurable. We address this need by developing a method and toolset to create, curate, and reuse knowledge for evolving requirements of large projects. K-RE facilitates extracting domain knowledge from semistructured and unstructured knowledge sources to create a structured KB consisting of generic requirement elements that can be reused in the same project as well as while starting new projects. We envisage a monitored environment where in a community of stakeholders creates and evolves the generic KB to suit project-specific needs. The just-in-time alerts to guide knowledge reuse in an RE process help achieve an improved completeness, consistency, and richness of the resulting specification. We have used a combination of the social software principles and semantic Web concepts.

We have demonstrated creation of knowledge repositories and its reuse in large projects.

In addition to knowledge reuse in RE, the concept of using active knowledge repositories can be extended to any exercise that draws on intensive knowledge services. We find that the foundation of our method and toolset is generic enough to cater to the knowledge reuse needs of stakeholders in very interesting emergent disciplines such as nanotechnology-based agriculture/food research [38, 39].

We realize however that the success of K-RE will depend largely on the quality of KB that we are able to create. Also, adopting K-RE would require a mindset change that is difficult to achieve in any organization. The upfront investment in creating a KB can also be a hindrance to adopting this approach.

## 7.8 Future Work

The work presented here attempts to combine benefits of the inference and reasoning possibilities associated with the use of semantic Web and the social aspects associated with Web 2.0 for achieving knowledge reuse in RE.

Semantic Web presents the additional possibility to link resources across the Web by assigning persistent URIs to domain concepts and their relationships. We have recently attempted to explore this possibility for understanding and visualizing the multi-domain span of requirements in a given domain [40]. Our approach tested on a few sample user stories brings out that the approach can help in explicitly visualizing the multi-domain scope of requirements and improve their completeness at the stage of specification itself [40]. We are aware that the completeness of ontologies is a precursor to this method and hence a limiting factor to its successful application. We seek opportunities to further test the applicability and scalability of the method and tool in large projects.

**Acknowledgments** We thank the anonymous referees of our previously published work in MaRK 10 and MaRK 11 workshops at previous RE conferences. One of us (SG) would like to thank Vijaya Deepti, TCS, for her solid support to this project and very many practical insights during the course of this work. The authors acknowledge inputs from Mini Jain and Preethu Rose (TCS) and reviews from V. S. Sivakumar and C. Sudheer of the Insurance unit of TCS.

## References

1. Kaiya H, Saeki M (2006) Using domain ontology as domain knowledge for requirements elicitation. In: Proceedings of the 14th IEEE international requirements engineering conference, IEEE Press, Minneapolis, pp 189–198
2. Lopez O, Laguna MA (2001) Requirements reuse for software development, in RE 01 doctoral workshop. In: Proceedings of the 5th IEEE international symposium on requirements engineering, Toronto, pp 27–31, Aug 2001
3. Feldman S, Sherman C (2004) The high cost of not finding information. Information Today, Incorporated
4. Making agile software development work for distributed teams. http://searchsoftwarequality.techtarget.com/news/article/0,289142,sid92_gci1277064,00.html. Accessed 15 Nov 2011
5. Buchan J, Ekadharmawan CH, MacDonell SG (2009) Insights into domain knowledge sharing in software development practice in SMEs. In: Proceedings of the 16th Asia-Pacific software engineering conference, IEEE CS Press, Penang, pp 93–100
6. Lohmann S, Dietzold S, Heim P, Heino N (2009) A web platform for social requirements engineering. In: Software Engineering 2009 – workshopband, Kaiserslautern, 2–6 March 2009
7. Ankolekar A, Krotzsch M, Tran T, Vrandecic D (2007) The two cultures- mashing up web 2.0 and the semantic web. In: Proceedings of the 16th international conference on World Wide Web, ACM, Banff, pp 825–834
8. Maalej W, Happel H (2008) A lightweight approach for knowledge sharing in distributed software teams. In: Yamaguchi T (ed) PAKM 2008. Lecture notes in artificial intelligence (LNAI), vol 5345. Springer, Heidelberg, pp 14–25
9. IBM -Jazz (2011) http://www-01.ibm.com/software/rational/jazz. Accessed 15 Nov 2011
10. Ajmeri N, Sejpal R, Ghaisas S (2010) A semantic and collaborative platform for agile requirements evolution. In: Proceedings of the third international workshop on managing requirements knowledge, IEEE Press, Sydney, pp 32–40
11. Cheng B, Atlee J (2007) Research directions in requirements engineering. In: Future of software engineering, Minneapolis, pp 285–303
12. Flynn DJ (1992) Information systems requirements: determination and analysis. McGraw Hill, London
13. Hofmann HF, Lehner F (2001) Requirements engineering as a success factor in software projects. IEEE Software 18(4):58–66
14. Boehm B (1981) Software engineering economics. Prentice Hall, Upper Saddle River
15. Kastanov A, Sakkinen M, Kastanov A, Sakkinen M (2006) Requirements quality control: a unifying framework. In: Requirements engineering, vol 11. Springer, New York, pp 42–57
16. Damian D, Chisan J (2006) An empirical study of complex relationships between the requirements engineering process and other processes that lead to payoffs in productivity, quality and risk management. In: IEEE transactions in software engineering, vol 32. IEEE, San Francisco, pp 433–453
17. Berners-Lee T, Hendler J, Lassila O et al (2001) The semantic web. Scientific Am 284(5): 28–37
18. Shadbolt N, Hall W, Berners-Lee T (2006) The semantic web revisited. IEEE Intell Syst 21(3):96–101

19. Hannemann A, Hocken C, Klamma R (2009) Community driven elicitation of requirements with entertaining social software. In: Software engineering 2009 workshop-band, Kaiserslautern, pp 317–328
20. Decker B, Ras E, Rech J, Jaubert P, Rieth M (2007) Wiki-based stakeholder participation in requirements engineering. IEEE Softw 24(2):28–35
21. Whitehead J (2007) Collaboration in software engineering: a roadmap. In: Future of software engineering, IEEE, Washington, pp 214–225
22. Folksonomy http://vanderwal.net/folksonomy.html. Accessed 15 Nov 2011
23. Ghazvinian A, Noy NF, Jonquet C, Shah N, Musen MA (2009) What four million mappings can tell you about two hundred ontologies? In: Proceedings of the 8th international semantic web conference, Lecturer notes in computer science, vol 5823. Springer, Heidelberg, pp 229–242
24. McGuinness DL, Van Harmelen F et al (2004) OWL web ontology language overview. W3C recommendation 10(2004-03):10
25. Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-oriented domain analysis (FODA) feasibility study. DTIC Document
26. Business Process Modeling Notation (BPMN) (2006) Specification, final adopted specification. Technical report, Object Management Group (OMG), Feb 2006
27. Cockburn A (2001) Writing effective use cases. Addison-Wesley, Boston
28. Ghaisas S (2009) A method for identifying unobvious requirements in globally distributed software projects. In: Proceedings of SENSE09, Kaiserslautern, Lecture Notes in Informatics (LNI), pp 297–308
29. Horrocks I, Patel-Schneider PF, Boley H, Tabet S, Grosof B, Dean M et al (2004) SWRL: a semantic web rule language combining OWL and RuleML. W3C Member Submission 21:79
30. Kiu CC, Lee CS (2007) Ontodna: ontology alignment results for OAEI 2007. In: Proceedings of the 2nd ontology matching workshop, Bonn, pp 196–204
31. Dao TN, Simpson T (2005) Measuring Similarity between sentences. http://opensvn.csie.org/WordNetDotNet/trunk/Projects/Thanh/Paper/WordNetDotNet_Semantic_Similarity. Accessed 24 Feb 2008
32. Rusu D, Dali L, Fortuna B, Grobelnik M, Mladenic D (2007) Triplet extraction from sentences. In: Proceedings of the 10th international multiconference on information society-IS, Ljubljana, pp 8–12
33. MySQL (2011) http://mysql.com. Accessed 15 Nov 2011
34. OpenNLP Project (2011) http://incubator.apache.org/opennlp. Accessed 15 Nov 2011
35. WordNet (2011) http://wordnet.princeton.edu. Accessed 15 Nov 2011
36. Spijkerman W (2010) Tool support for change impact analysis in requirement models: exploiting semantics of requirement relations as traceability relations, University of Twente
37. Bohner SA, Arnold RS (1996) Software change impact analysis. IEEE CS Press, Los Alamitos
38. Rose P, Bhat M, Vidhani K, Ajmeri N, Gole A, Ghaisas S (2011) Intelligent informatics platform for nano-agriculture. In: 2011 11th IEEE conference on nanotechnology (IEEE-NANO), IEEE, pp 916–919
39. Rose P, Gole A, Ghaisas S (2011) A semantic regulatory framework for nanotechnology application in agri-food domain. In: 2011 fourth international workshop on requirements engineering and law (RELAW), IEEE, pp 60–66
40. Ajmeri N, Vidhani K, Bhat M, Ghaisas S (2011) An ontology-based method and tool for cross-domain requirements visualization. In: 2011 fourth international workshop on managing requirements knowledge (MARK), IEEE, pp 20–23

# Part III
# Sharing Requirements Knowledge

"The power of human thought grows exponentially with the number of minds that share that thought."
—Dan Brown

© Walid Maalej. Printed with permission

# Chapter 8
# Reusing Requirements in Global Software Engineering

**Juan Manuel Carrillo de Gea, Joaquín Nicolás, José Luis Fernández Alemán, Ambrosio Toval, A. Vizcaíno, and Christof Ebert**

**Abstract** Knowledge sharing and reuse in global software engineering (GSE) are challenging issues. Knowledge management (KM) is specifically impacted because on top of distance, culture and language mismatches, there is also the perceived risk of sharing something which could mean that others could take over some work. Mistrust and protectionism are often the consequence, leading to insufficient reuse. This is visible specifically in requirements engineering (RE), where all reuse should start. In this chapter, we will look to reuse in RE with a detailed look on how to improve knowledge sharing and collaboration in distributed environments. We first look into the state of the practice. Then we present a lightweight, reuse-based, global RE method called PANGEA (*Process for globAl requiremeNts enGinEering and quAlity*), based on natural language requirements and software engineering standards. Based on this method, we also build a prototypical tool, called PANTALASA (*PANgea Tool And Lightweight Automated Support Architecture*) which provides automated support for PANGEA. Its features are drawn from PANGEA and the state of the practice commercially available RE tools. A prototype of PANTALASA was developed by using Semantic MediaWiki and Facebook and applied to a case study in the domain of hotel management. We could show with this method and prototype that collaboration and thus KM and reuse in RE are improved.

J.M. Carrillo de Gea (✉) • J. Nicolás • J.L. Fernández Alemán • A. Toval
Universidad de Murcia, Murcia, Spain
e-mail: jmcdg1@um.es; jnr@um.es; aleman@um.es; atoval@um.es

A. Vizcaíno
Universidad de Castilla - La Mancha, Ciudad Real, Spain
e-mail: aurora.vizcaino@uclm.es

C. Ebert
Vector Consulting Services, Stuttgart, Germany
e-mail: christof.ebert@vector.com

## 8.1 Introduction

Brooks stated more than 20 years ago [1] that there are approaches in software engineering (SE) that target the *accidental* complexity of software, while others are targeted at its *essential* complexity. As regards this difficulty, and discussing the role of requirements engineering (RE) in software development, Brooks considered the *refinement of requirements and rapid prototyping*. Among the SE strategies which attack the conceptual essence of the problem, he also mentions the idea of *buy* versus *build*: in other words, he stated the need for software reuse. Software reuse is for Meyer [2] "the ability of software elements to serve for the construction of many different applications". Meyer summarises the benefits of reusability: (1) improved timeliness, in the sense of decreased time to market; (2) reduced software maintenance efforts; (3) improved reliability, efficiency and consistency of the developed software and (4) enhanced investment, through the preservation of the know-how. Mili et al. [3] affirm that software reuse is "the (only) realistic approach to bring about the gains of productivity and quality that the software industry needs".

There is a line of thinking within software reuse that considers that every software artefact (asset) produced during the development process is reusable, including system or product specifications, design, source code, test cases, project plans, quality plans, etc. Since the mid-1990s, specifications and requirements reuse are postulated by a number of authors as a promising path towards quality and productivity in software development, a way that has been less explored than reuse of source code or designs. There is some consensus, in that the higher the abstraction level and the more not only source code but also design and specifications are reused, the greater the reusability benefits are [4, 5]. In this respect, Favaro [6] affirms that "a well-formulated, measurable, reusable requirement [. . .] is every bit as valuable as a reusable software module". In addition, Robertson and Robertson [7] claim that if the development starts with a set of requirements that were specified for other projects or domains, the accuracy of the requirements specification is improved, and the time to develop this specification is reduced. Cheng and Atlee [8] also consider the identification of sets of reusable requirements for particular domains or types of applications to be of interest. To the best of our knowledge, Rine and Nada [9] are the first authors who demonstrated empirically that the reusability level determines the effectiveness of improvements in productivity, quality and development time, concluding that greater benefits are obtained when reusability is applied during the initial processes of the software development life cycle.

Cheng and Atlee [8] highlighted *globalisation* and *requirements reuse* as two of the more urgent needs and grand challenges in RE research and expected the solutions to these issues to produce a great impact on both research and practice in SE. Globalisation arises from growing and relatively new software needs, while requirements reuse focuses on extending and maturing existing technologies. Global software engineering (GSE) implies a paradigm shift towards globally distributed development teams [10], and it has become a business need for various reasons: to decrease costs, capitalise on global resource pools owing to the scarcity of resources, locate development closer to the customers, exploit around-the-clock development to achieve cycle-time acceleration and cater to local markets [11]. In contrast, GSE leads to an increased risk of communication gaps, given the temporal, geographic, cultural and linguistic nature of the distance imposed by GSE [12], which might hinder collaborative activities that require stakeholders to share a mental model of the problem and requirements [8]. Indeed, due to its collaboration-intensive nature, RE presents several specific challenges and difficulties when the stakeholders are distributed [13, 14]. Gallardo-Valencia and Sim [15] are convinced that the success of a software product depends on the proper understanding of requirements among stakeholders. Herbsleb [10], for his part, emphasised that "getting the requirements right, and dealing with unstable requirements" are notoriously difficult problems to address, even in a traditional, collocated environment. A shared understanding of the requirements is even more difficult to achieve in a GSE context, "both because of loss of context and loss of communication". As well as this, Herbsleb pointed out that research on *eliciting and communicating requirements* has made substantial progress in addressing the issues posed by globalisation, although impediments still exist.

Knowledge is considered to be one of the major resources of an organisation, and this is further emphasised in organisations dedicated to software development. SE and RE are knowledge-intensive activities [15, 16] and hence, the growing interest of software development organisations in providing methods to help with its appropriate management [17]. By means of knowledge management (KM), software development organisations might obtain certain potential benefits: decrease the development time and cost of software projects, avoid mistakes and reduce rework, increase productivity through repetition of successful processes, increase quality and make better decisions [18]. Thus, achieving good KM is very important if competitive levels are to be maintained in an increasingly globalised and demanding world. Nevertheless, challenges to KM increase when the development activities are geographically distributed [16]. According to Ebling et al. [19], proposals related to KM challenges in the field of RE in distributed software development are needed; they encourage further research on these issues.

Both GSE and requirements reuse are therefore relevant approaches for the software industry. As far as we know, however, there are no proposals which tackle both GSE and requirements reuse together. Since one of the current problems in GSE is the existence of knowledge which is not properly shared and reused, KM and awareness in distributed settings is a challenging task [20] which might be addressed by reusing, sharing and collaboration mechanisms in GSE. The proposal which is presented here treats knowledge in the form of natural language requirements and aims at laying out the basis for a reuse-based RE method for GSE environments, named PANGEA (*Process for globAl requiremeNts enGinEering and quAlity*). It also includes PANTALASA (*PANgea Tool And Lightweight Automated Support Architecture*), the automated support for the PANGEA method.

The rest of the chapter is organised as follows: Sect. 8.2 provides an overview of the field. Current and relevant challenges for both RE and KM in GSE are studied in Sect. 8.3. Section 8.4 includes our proposal for a method to address these issues. Section 8.5 focuses on the tool architecture supporting this method. Section 8.6 reports on a preliminary evaluation of both method and tool. Finally, our conclusions and future work are presented in Sect. 8.7.

## 8.2 Foundations

At this point, and in the context of this chapter, a brief description of KM, architectural knowledge management (AKM) and KM as the basis for RE is provided.

### *8.2.1 Knowledge Management*

Al-Ani [21] cites a generally accepted definition for the term *knowledge*. It is located at the top of a hierarchical structure, and this representational structure sees *information* as standing above *data* and knowledge standing above of both. In the model described, data is processed and transformed into information; information is then interpreted and contextualised by individuals and transformed into knowledge. In addition, the term KM suggests that knowledge is something tangible which is possible to manipulate. According to Ebert and De Man [22], KM is "the process that deals with systematically eliciting, structuring and facilitating the efficient retrieval and effective use of knowledge".

In those organisations involved in GSE projects, an adequate KM is necessary to mitigate those factors derived from the geographical, temporal and sociocultural distance [23–25] that might hamper communication and relationships between stakeholders. Indeed, knowledge changes quickly during software development, and so all the knowledge generated in the project should be made as accurate, complete and updated as possible. Furthermore, if software development is distributed globally, many more people are involved in the development activities, and thus organisations tend to have problems in terms of content, location and use of knowledge that can make it difficult to take advantage of this knowledge. Moreover, Al-Ani [21] states that ineffective KM can lead to disastrous consequences, showing some examples and lessons learnt which illustrate ineffective KM practices and their aftermath.

During the activities of the software life cycle, knowledge which is important for the subsequent activities is generated [26], and these are commonly carried out by different people to those involved previously. It is important to ensure that this knowledge is accessible for them. The software development process generates documents and other software engineering artefacts. In this sense, KM plays a very important role, since this knowledge has to be captured, and it emerges from the solution to problems encountered during the course of past and current projects. However, some common issues in software development organisations are actually problems in the flow of knowledge, e.g. lack of documentation [27]. To be successful, organisations that apply KM techniques and processes should create an organisational culture that fosters and promotes the dissemination and sharing of knowledge among its employees, through encouraging them to document and store their knowledge in a KM repository [18]. This is especially true in GSE settings, where according to Ebert and De Neve [28], a relevant step towards creating such a common culture is to choose a specific, common language to be used within the organisation. However, these authors highlight that since "a common syntactical language does not necessarily mean the same semantics and pragmatics", team members should rotate across locations to live within different cultures and gradually build mutual understanding. On the other hand, this issue might be also addressed by means of *ontologies*, which formalise concepts and relationships among them as well as enable automatic reasoning with knowledge [29].

### *8.2.2 Architectural Knowledge Management*

According to Beecham et al. [30], AKM involves capturing, sharing and managing the information resulting from the software architecture process, in the form of knowledge of the problem domain, the solution domain and knowledge artefacts used throughout the whole process. It supports the creation, storage and dissemination of all the knowledge used in defining and using the architecture, including the requirements documentation [31] and the functional and non-functional requirements [30]. In fact, there are similarities between requirements and architecture [32]. We could go even further; Clerc [33] notices "a close resemblance between a set of requirements for a software system and a set of architectural decisions taken for that software system: what one person regards as requirements for a software system, another person may regard as architectural decisions".

Architectural knowledge can serve to support the collaboration needs of distributed software development organisations [31]. In addition, as was stated above, KM is even more challenging in a GSE context [16], and exactly the same situation occurs in the case of AKM [30, 31], which faces the same risks as RE in GSE settings [33] (for a detailed study of threats and safeguards in RE for GSE environments, see, e.g. [34]). There is a need to capture, share and manage architectural knowledge, in particular in the form or requirements, among different and distributed sites, but the related tasks become more demanding than in a collocated setting. Thus, the challenges of GSE have to be addressed by the members of globally distributed teams by relying on appropriate AKM practices; Beecham et al. [30] identified that one important area to achieve AKM is represented by KM practices for creating and disseminating architectural knowledge.

In this context, Clerc [33] identified seven essential KM practices to achieve effective AKM in GSE environments that build on the RE discipline. In summary, these practices are (1) *frequent interaction across sites*, encouraging team members to interact frequently with each other; (2) *cross-site delegation*, improving integration of distributed teams by means of mutual delegation of team members from a local site to a remote site; (3) *face-to-face project kickoff meetings*, assisting the establishment of initial relationships between and among teams; (4) *urgent request*, identifying expert project members to collect information on a given topic quickly; (5) *collocated high-level architecture phase*, creating a sound high-level architecture efficiently; (6) *clear organisation structure with communicating responsibilities*, maintaining clear lines of communication among stakeholders' roles; and (7) *establishing of a repository for architecture artefacts*, built to store architectural knowledge. Moreover, Ebert and De Neve [28] emphasise that customer requirements might be mapped to architectural units in order to cluster activities and split a globally distributed project into different, collocated work teams. In this way, each team assumes the responsibility for a set of functionally related customer requirements, and teamwork is therefore reinforced.

According to Desouza et al. [35], there are three distinct strategies for AKM: (1) *codification*, which refers to the use of a central repository for storing the architectural knowledge; (2) *personalisation*, which alludes to stakeholders and the interaction between them to get the knowledge when they needed it; and (3) a *hybrid approach*, in which there is a central knowledge repository shared among stakeholders, addressing questions such as "when", "how" or "who" in relation to knowledge, in order to enable personalised knowledge sharing.

### 8.2.3 Knowledge Management for Requirements Engineering

Regarding reusability, there are various studies which have centred on applying KM in software development organisations, most of which focus on the reuse of experiences, such as best practices or lessons learned, in order to improve the quality of processes and products or to facilitate the reuse of software artefacts [36–38]. In this context, Rus and Lindvall [18] consider software reuse to be a KM activity that supports software development. These authors affirm that repeated implementation by programmers of the same or very similar solutions, along with the rework resulting from this, might be avoided or reduced through establishing a reuse repository, which would contain software previously submitted by others, and "this same concept can apply to all software engineering artefacts". This approach requires a change in the software development process, since the first step would be to search the repository for reusable parts developing the solution from scratch only if nothing useful is found. Furthermore, information is often reused with high redundancies or manual overhead, eventually leading to rework or even errors in the product [22]. Thus, only reuse of information is not enough, and knowledge should be embedded into integrated workflows for specific tasks. This strategy "generates immediate returns by making engineers more flexible".

For Gallardo-Valencia and Sim [15], requirements knowledge is ideally captured in a requirements specification document using a written format, although such written knowledge is complemented by requirements knowledge that is shared in an informal manner through conversations among and between stakeholders. Moreover, Maalej et al. [39] affirm that it is necessary to capture and share tacit knowledge about requirements and make it explicit, in order to be able to manipulate it, because (1) reuse is enhanced, (2) traceability is enabled, (3) requirements evolution is supported and (4) collaboration between participants in distributed projects is improved. Besides, Ma et al. [40] have noted that the presence of tacit knowledge might have a negative impact on communication through requirements documents; such knowledge should therefore be properly managed with the intention of avoiding miscommunication, misinterpretation and inappropriate contextualisation among stakeholders, especially in the case of a GSE context [21]. Even though recent technologies and advancements have boosted KM support within distributed software development teams [21], in particular KM support to RE [39], usual technologies and infrastructures typically focus only on addressing issues related to the management of explicit knowledge, whereas they should capture, formalise and manipulate tacit or implicit knowledge [21, 39]. Maalej et al. [39] demand improvements in RE processes and tools in order to achieve better management of requirements knowledge, owing in particular to the growing tendency towards agile methods and distribution in software development, while "the major constraint is to have a lightweight, usable, intelligent and personalised capturing and sharing approach".

Another key issue for achieving successful results in global RE is requirements awareness. Informal communication is also important in global, distributed development because it contributes to project awareness [41]. In a software development setting, and in particular in RE, *awareness* happens if "a software developer working in a project has knowledge of events, such as changes to a requirement suggested by a customer, that occur in the project" [42]; this becomes even more essential in GSE [13]. Kwan et al. [42] conducted an industrial study of two GSE projects; one of them is an example of offshore outsourcing, and the other project has outsourced collocated development. These authors identified three main factors that produce certain effects which might influence awareness: (1) the distributed development reduces awareness, (2) the experience of the team members bridges awareness gaps and (3) the centralised communication structure might prevent awareness problems. All this being so, project members should keep abreast of any issues that take place in the scope of the project. The lessons learned are (1) experienced team members should be accessible; (2) a centralised communication structure can help new teams to remain aware, whereas a decentralised structure decreases communication bandwidth and improves response times; (3) frequent meetings improve awareness among distributed sites and (4) each distributed team member should be assigned to a set of stable requirements and an unstable requirement, in order to allow him or her to experience minimal downtime when there are delays. In conclusion, requirements awareness is neither a banal nor an easy issue, and a lack of awareness can lead to problems with design, quality and cost within the distributed project [42].

## 8.3 Practical Challenges

Distance hinders KM and requirements management processes in GSE. Issues concerning GSE have been given a lot of attention in literature in the last years. To begin with, Cheng and Atlee [8] affirm that new or extended techniques are needed to overcome the challenges posed to RE by globalisation; they are the following: (1) obtain proper support to outsourcing of downstream software development tasks (e.g. design, coding, testing, etc.), bearing in mind that distance complicates the collaboration between the requirements and the development teams, and (2) enable effective distributed RE activities, since analysts and geographically distributed stakeholders will work together and distributed software development teams might work with in-house customers; practitioners therefore need techniques to facilitate distributed requirements elicitation, modelling, negotiation and management of distributed teams.

Ebling et al. [19] have conducted a systematic literature review of RE in distributed software environments, concluding that the challenges identified in the field of KM issues are related to inappropriate sharing of requirements information with distributed stakeholders [10, 41], which eventually damages the interaction between them [13]. Moreover, there are no available methods, models, techniques or approaches to RE in GSE environments in relation to the KM challenges identified [19].

Process mismatches, differing technical and domain vocabularies, incompatible environments and conflicting assumptions can be particularly problematic in a GSE context [43]. Cultural differences can also pose formidable challenges for achieving a shared understanding of the requirements [44], and all these factors can hamper discovery and integration of knowledge [10]. Knowledge transfer is "the transmission of general or project specific knowledge which is needed to understand and execute the project requirements" [45]. It is also a challenging activity of KM in GSE environments, because of the significant reduction in communication frequency and speed among remote teams [46], which leads knowledge to become fixed to locations in which it is produced, hindering the transfer of such knowledge from one site to another [47].

Manteli et al. [16] classify the challenges of KM in GSE under three main categories: *communication*, *knowledge creation and storage* and *knowledge transfer*. The coordination of communication between remote teams is included in KM [20, 23], but it is a fact that distance introduces barriers to both informal and face-to-face communication in GSE. Project members have to rely on synchronous communication tools (e.g. chats, phone calls, videoconferences), or asynchronous ones (e.g. discussion forums, emails), in order to collaborate [41]. Since communication speed and frequency is relevant in a GSE context, any communication delay can slow down or even detain the project course, producing delivery delays [16], so synchronous communication is generally preferable in distributed environments [48], as it boosts real-time, interactive communication and improves collaboration among stakeholders. By means of synchronous communication tools, analysts can check each other's work, see if certain features have been implemented the right way or solve problems together and assist each other. Nevertheless, the most appropriate communication media depend on the team member's role [16]. It is also worthwhile to reduce tasks by coupling as much as possible, since the interdependencies among distributed tasks introduce important communication overload, thus affecting communication speed and frequency between distributed sites [16].

The effectiveness of knowledge capture (the process of making it explicit), that is, how knowledge is captured into software development artefacts and acquired by other team members, is a critical success factor for projects [49]. Among the distinct KM strategies for knowledge capture and management [35], *codification* is pursued when knowledge is documented and stored in a central repository, *personalisation* relies on the tacit knowledge of stakeholders and knowledge sharing through person-to-person communications and the *hybrid approach* combines the previous two, being the recommended strategy towards AKM in GSE [50]. An important challenge for KM is found in personalisation strategies, which are typical of agile development methods, in which employees are encouraged to cooperate with each other, taking initiative and responsibility without being constrained by strictly defined processes [16]. Furthermore, with this approach, much of the knowledge remains tacit, and the explicit knowledge is not always updated in the corresponding documents. However, documentation has an important role [27], and it should consequently be updated; it ought to reflect what distributed teams are working on, in order to keep requirements awareness [42]. According to Manteli et al. [16], the use of different repositories and tools for storing and sharing documents is not recommended. The knowledge search through all these resources to locate the right document and the up-to-date information is complicated, leading in turn to the adoption of local codification strategies that hamper knowledge sharing between sites. In addition, if no appropriate mechanisms for storing and sharing documentation are provided, more communication between distributed project members is needed, decreasing stakeholders' productivity.

Knowledge transfer was identified as a critical success factor for software development projects with an onsite/offshore structure [45], since some circumstances obstruct knowledge transferability across sites, including the use of different working methods [51] or the differences in skills and expertise of remote project members [30]. Manteli et al. [16] point out that when the greater part of a software development project is developed at one site, most of the *system-generic* knowledge ("the comprehensive knowledge of the entire system that teams are working on") resides only there, which causes knowledge to be fixed to that location [47]. It thus takes additional effort for that knowledge to be transferred to the remote sites, which only retain the *unit-specific* knowledge ("the particular knowledge that the individual has, for the specific unit he or she is working on"). Another challenge in knowledge transfer is "how" to locate the knowledge [16]; an effective knowledge-sharing strategy should enable project members to know "who", in addition to providing the ability to know "what" and "where" knowledge resides [52]. This approach is also known as *transactive memory* [51], and according to Kotlarsky and Oshri [53], it constitutes a means of knowledge sharing that contributes decisively to successful collaboration between and among remote teams. A personalisation strategy in which people transfer more knowledge in a person-to-person way leads to know "who" knows "what" more efficiently [16].

Tools help in managing requirements and are key success factors in GSE [28, 54]. Moreover, Ebert [54] affirms that change management is unmanageable without automated tools in a GSE environment. Traceability facilitates change management and must include horizontal and vertical dependencies – between artefacts in the same and in different abstraction levels, respectively. Heindl et al. [55] detected a lack of traceability and computer-aided requirements engineering tools in RE for GSE (from now on, these will be referred to as CARE tools or simply RE tools). Mistrík et al. [56] point out that a considerable number of recent advances in collaborative SE are related to the development of supporting tools for certain collaborative practices. Portillo-Rodríguez et al. [24] have conducted a systematic literature review of GSE tools supporting the ISO/IEC 12207 processes, concluding that the majority of the tools were developed within research groups or labs. In addition, all the tools analysed included web-based or client–server access, as well as communication and coordination features for globally distributed teams. Herbsleb [10] highlighted that *environments and tools* are important areas of research in GSE. This author affirms that awareness and communication are relevant and interrelated issues, since the reduced communication bandwidth in GSE makes it much more difficult to face the problem of understanding what other project members are doing and thus coordinate effectively with them. This means that synchronous and asynchronous communication features should be integrated into RE tools, avoiding practices that might lead to low efficiency and productivity, such as using e-mail instead of web-based tools or shared repositories to manage requirements [57]. Besides the awareness and communication features, Herbsleb [10] detected a need for collaborative capabilities to be integrated into the development environment, as well as more "interoperable tools with standard data formats and interaction protocols". In this regard, software requirements should be suitable for being imported from, or interfaced to, users, hardware and other software systems. This being so, standard file formats are interesting features which should be offered by RE tools. Since companies rarely work on the same requirements repository and they usually do not work with the same RE tools, a generic format for requirements information is needed. In this context, the Object Management Group (OMG) standard ReqIF [58] is an emerging open, non-proprietary exchange format that is a successful step towards satisfying the urgent industry need for exchanging requirement information between different companies, without losing the advantage of requirements management at the organisations' borders and allowing them to interact and collaborate efficiently.

## 8.4 The Method

Although GSE has recently attracted great interest, to the best of our knowledge, an RE method that specifically addresses GSE and knowledge reuse is lacking. In response to this problem, a global, reuse-based RE method called PANGEA is presented in the following paragraphs. PANGEA allows the sharing and reuse of knowledge between distributed teams through a shared repository of requirements. The PANGEA repository contains both reusable knowledge from earlier projects and the requirements under development in the current one. A proper requirements management is critical to maintaining awareness in any kind of development. For the sake of applicability, PANGEA encodes knowledge in the form of natural language requirements, which are the most widely used requirements in industry.

Industry experience demonstrates that a process model based on accepted best practices that allows tailoring processes for the specific needs of a project contributes to support GSE [28]. Therefore, to propose an RE method addressing reuse and GSE, we have done the following: firstly, we have studied the state of the art on the threats and their solutions identified in literature regarding RE and GSE, through a systematic literature review [34]. Secondly, a repository of risks and safeguards for the global RE has been compiled. Finally, based on this repository, a global RE method called PANGEA, which encompasses all the proposed safeguards, is put forward in the following pages.

PANGEA extends the SIREN (*SImple Reuse of softwarE requiremeNts*) requirements reuse method [59], a practical way of dealing with requirements reuse. SIREN is a method which is simple enough to be adopted easily by organisations that are currently immature in relation to RE. It can improve productivity and quality of software processes and products, as well as affecting the business positively – indeed, Sommerville and Ransom [60] have conducted an empirical study which revealed that improvements in the RE process led to business improvements. SIREN can be used on its own, as the first RE method adopted by a software development organisation, but in addition, SIREN can be considered as a kind of *add-in* whose goal is to extend, with reusability concerns, any RE method based on natural language requirements. Moreover, Toval et al. [61] state eight key issues that should be taken into account to achieve a practical reuse-based RE method. These are based on the lessons learned from the application of SIREN in the security and data protection fields [59, 62], as well as on other related experiences closer to the domain analysis or audit [63, 64], together with the analysis of related research and current RE tools. However, SIREN was conceived as an RE method for collocated settings right from the beginning; in consequence, it does not address the issues that should be considered when working in a global environment. That makes it necessary to extend and adapt the SIREN method.

The PANGEA method is based on SIREN to a great extent, owing to the success of the latter in practice. Nevertheless, the process model (new activities and new tasks, along with a new set of roles responsible for carrying them out) and part of the techniques (in particular, the requirements reference model, with new requirements attributes and new attribute values, as well as new traceability relations) have undergone modifications. As well as all this, another part of the techniques, namely, the hierarchy for the requirements documents and the reusability bases (the repository of requirements arranged into catalogues and the requirements reuse guidelines), have been inherited unchanged from SIREN. Thus, the remaining part of this section is devoted to the method resulting from the process explained above, i.e. the method known as PANGEA. Furthermore, having presented PANGEA, Sect. 8.5 goes on to explain the architecture of PANTALASA, the automated support for PANGEA, given that it has evolved along with the process model and the techniques for addressing the GSE issues.

## *8.4.1 Process Model*

The process model proposed for PANGEA combines a set of initial sequential tasks with other cyclical, iterative tasks. This approach therefore includes a spiral model of software development (Fig. 8.1).

In Fig. 8.1, IRD 6, IRD 7, IRD 9 and IRD 10 tasks move from the original model of SIREN, while IRD 1–5 and IRD 8 are brand new. The first five tasks serve as a preparation for making the globally distributed method successful. For this reason, they only have to be performed in the first iteration. The last five tasks make up the effective execution of the RE method, and they are conducted on a cyclical basis for as many iterations as necessary.

Before examining each task, it is necessary to introduce the roles involved in the process model: *coordinator* (coordinates the work of all the project's participants), *moderator* (moderates the requirements negotiation meetings), *team leaders* (represent their work team and speak in their name with the *coordinator* and other *team leaders*), *analysts* (or requirements engineers), *key users* (know the whole system or a part of it and give the necessary knowledge for the production of the requirements documents) and *users* (know part of the system and give the knowledge that is needed for the creation of the requirements documents). Each role has specific responsibilities and a given participation in the tasks and subtasks.

#### **8.4.1.1 IRD 1: Cultural Analysis**

This task analyses the different cultures of the participants in the project, using cultural indicators. The role responsible for conducting this study is the *coordinator*, and the reports obtained become part of a catalogue of cultural descriptions, so that they can be reused. The subtasks within this task are:

- IRD 1.1: Nationality identification. The nationalities of all the work teams involved in the project are registered, considering the term *nationality* as the

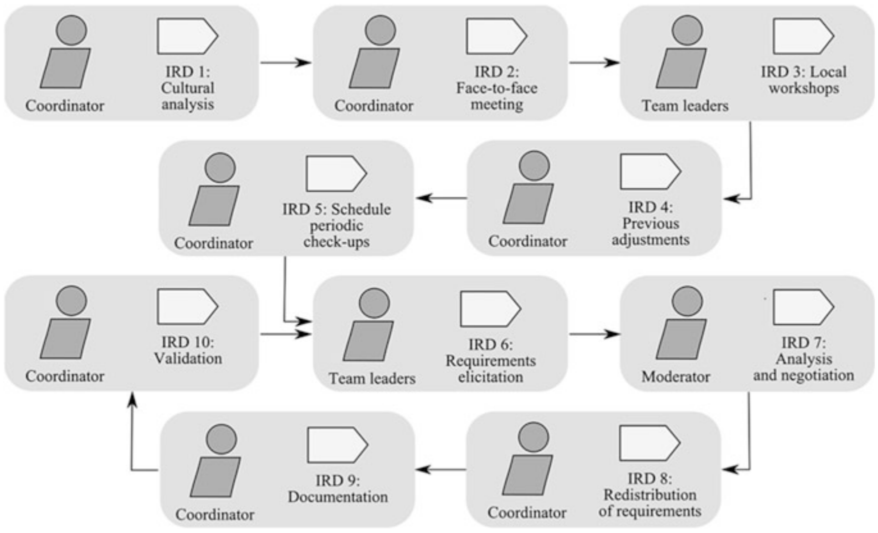


**Fig. 8.1** PANGEA process model with SPEM notation

national culture in which people have grown and acquired their values scale. If the work team is heterogeneous, individuals should be analysed.

- IRD 1.2: Report retrieval. The catalogue of cultural descriptions is consulted, and existing reports on the cultures involved are obtained.
- IRD 1.3: Cultural reporting. Preparation of reports on the cultures that are not in the catalogue of cultural descriptions. If they are already in it, review them with the purpose of refining its descriptions and finding errors.
- IRD 1.4: Improvement of the catalogue of cultural descriptions. The reports produced for the first time, along with those already existing and which were improved, are included in the catalogue.

#### 8.4.1.2 IRD 2: Face-to-Face Meeting

It is essential to hold at least one face-to-face meeting at the beginning of the project, not only because it is a richer and more efficient form of communication than any other but because it is needed if a trust relationship is to be established between the participants in a distributed project.

- IRD 2.1: Face-to-face meeting. A meeting for all participants in the project is organised by the *coordinator*. Attendance of at least one representative on behalf of the customer is recommended. All members of all work teams should take part in this meeting; if this is not possible, it will involve at least the *team leaders*. The event should make it possible both to discuss the initial questions about the project and to allow the participants to get to know each other a little, at least.

#### 8.4.1.3 IRD 3: Local Workshops

Cultural understanding is critical to establishing trust relationships.

- IRD 3.1: Local workshops. The *team leaders* organise workshops locally to study the differences between the cultures of the other teams and their own, by means of the indicators established in IRD 1. They are responsible for disseminating the information within their team, the other *team leaders* and the *coordinator*. When the *analysts* and the *team leader* of a team are aware of the major cultural differences, the *team leader* has to report to the *coordinator*. The task ends when the *coordinator* has received confirmation from all the *team leaders*.

#### 8.4.1.4 IRD 4: Previous Adjustments

This task aims to make final adjustments prior to elicitation of requirements.

- IRD 4.1: Key user identification. This problem is complicated in GSE, since the distance and the inability to observe the users performing their regular work may hinder proper identification of people playing the *user* and *key user* roles.
- IRD 4.2: Assignment of key users to work teams. A set of *key users* is assigned to each work team to elicit requirements. They should be collocated whenever possible. Otherwise, the elicitation techniques selected must be compatible with electronic communication media. Only *key users* should collaborate in the elicitation activity, but if this is not possible, then *users* can also be assigned.
- IRD 4.3: Selection of an official language. Although all roles can use the language with which they feel most comfortable in informal social interactions, the *coordinator* must establish an official language for the project. This is to be used in formal requirements negotiation meetings or any situation involving participants from different native languages.
- IRD 4.4: Compilation of a common vocabulary. The terms used in the project-specific domain should be documented, but in GSE the language differences may be substantial. Ontologies are thus useful to alleviate these problems. At this point, a catalogue of ontologies is consulted, and a relevant ontology is retrieved or built from scratch if there is not yet an ontology for the domain.

#### 8.4.1.5 IRD 5: Schedule Periodic Check-Ups

The distance in GSE projects entails many problems, but also enables a follow-the-sun or around-the-clock working model, achieving 24-h global workdays.

- IRD 5.1: Schedule periodic check-ups. The *coordinator* examines the location of the work teams and their time differences, with the intention of scheduling daily meetings. These meetings might be uncomfortable sometimes, but are necessary for tracking the work of the other teams.

#### 8.4.1.6 IRD 6: Requirements Elicitation

Each work team can reuse and extract requirements locally or in a distributed manner. In both cases, different groups of *analysts* extract requirements from a repository of requirements and from different *users*. As a result, a mostly coherent requirements documents hierarchy is obtained.

- IRD 6.1: Requirements reuse. Firstly, each work team selects the catalogues of requirements related to the project from the repository of requirements. Secondly, the team instantiates the parametrised requirements. Finally, the team adds the requirements to the current requirements document. The automated tool support should avoid most of the problems related to the multiple sources of requirements (see Sect. 8.5). Inconsistencies that cannot be confronted automatically should be added to the agenda of the next analysis and negotiation meeting (IRD 7).
- IRD 6.2: Project requirements elicitation. The *analysts* within each work team develop the requirements obtained from the *users* who are working with them and add these to the current requirements document. The *moderator* reviews the requirements included by the different work teams and adds all the inconsistencies found to the agenda of the next analysis and negotiation meeting. In addition, the *analysts* can add any other issue to the agenda that they need to address, by creating a discussion thread.

#### 8.4.1.7 IRD 7: Analysis and Negotiation

Starting from a requirements document with inconsistencies, in which there are issues to discuss, the goal is to obtain a refined version of it in which all the inconsistencies are resolved.

- IRD 7.1: Preparation of the meeting. The agenda, which should be available in the automated tool support, should be read by all the participants involved in the meeting.
- IRD 7.2: Development of the meeting. The discussion is initially carried out by means of a structured, synchronous and textual communication system. The *moderator* takes part in the discussion to impose order, to maintain the progress of the meeting on a virtual blackboard and to use a vote utility if needed. Only the *team leaders* are allowed to participate in representation of their teams. A videoconference system can only be used later on; the reasons for doing it this way are the following: (1) the participants have had enough time to study all the information concerning the agenda, without the pressure of a synchronous communication; (2) the discussion has been conducted so far by means of a textual synchronous tool controlled by the *moderator*, which means that the written interventions have had more chance of being well thought out; and (3) the face-to-face communication allows the reactions of the participants to be evaluated better and helps to solve any issue that still remains open.

- IRD 7.3: Extraction of conclusions. The *moderator* publishes the agreements of the meeting in the automated tool support, including the conclusions reached and the results of any voting carried out. The agreements will be given a unique code to identify them in the project. A discussion thread will be linked to the agreements, for the purpose of storing all the written discussions, the results of the eventual votes and the changes in the requirements documents.

#### 8.4.1.8 IRD 8: Redistribution of Requirements

The requirements documents contain information about the team which has elicited each requirement. In this task, these assignments of requirements to work teams are revised so that requirements can be allocated to different teams.

- IRD 8.1: Assignment proposal. The *coordinator* prepares a proposal for assigning requirements to each work team, based on their most prominent skills, current or expected workload, etc. Such a proposal is made available to all teams through the supporting tool, and a meeting is convened with them to discuss it.
- IRD 8.2: Validation of the assignment. A virtual meeting involving all the *team leaders* and the *coordinator* takes place to discuss the issues related to the assignments. An analysis and negotiation meeting (IRD 7) is performed if needed. Eventually, a validated requirements redistribution is obtained and published in the supporting tool.

#### 8.4.1.9 IRD 9: Documentation

We start from a requirements document that is coherent and without inconsistencies, ideally, and which can be more or less detailed depending on the current iteration of the method; the result is the establishment of such a document as a baseline.

- IRD 9.1: Formalisation of documentation. The *coordinator* creates a baseline from a stable version of the documentation, stores it in a catalogue of releases and exports it to a standard document format by means of the supporting tool.

#### 8.4.1.10 IRD 10: Validation

This task is identical to that carried out in collocated development of software.

- IRD 10.1: Validation of requirements. The *coordinator* provides the document output of IRD 9 to the customer, negotiates all the change requests and forwards the information to the *team leaders*, through a list of changes published in the supporting tool.

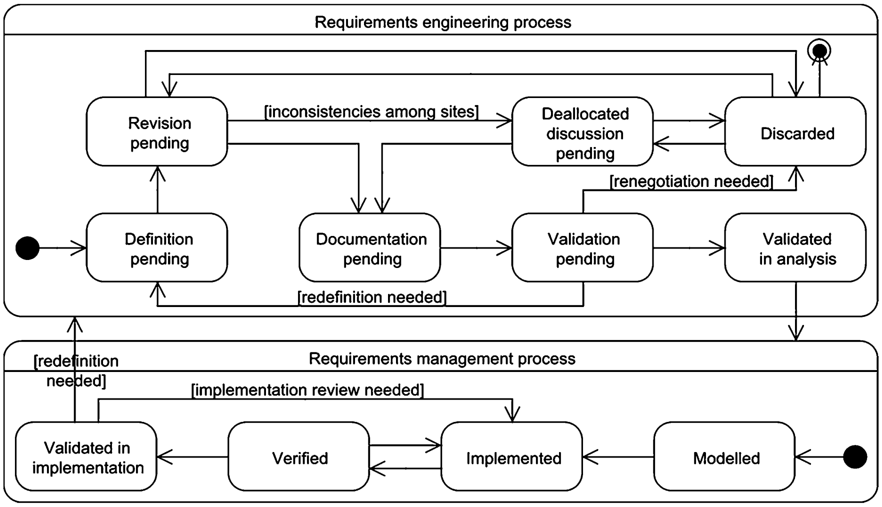


**Fig. 8.2** State diagram of a requirement

## *8.4.2 Requirements Reference Model*

There are different types of requirements in PANGEA, but all have the minimum set of associated attributes shown below (the requirements that must be initialised when created are marked as "compulsory"):

- *Text* (compulsory): natural language sentence that specifies the requirement.
- *UniqueIdentificator* (compulsory): identifier of the requirement in the project.
- *Risk*: relative risk of the requirement compared with the rest.
- *Criticality*: relative importance of the requirement for the customer.
- *Priority*: helps to establish an order for the development (set by the analyst).
- *Rationale*: reason why the requirement is included in the project.
- *State*: situation of the requirement (see Fig. 8.2).
- *Source*: origin of the requirement (if it was reused from the repository, then the attribute reflects the catalogue and reusable requirement it comes from).
- *ValidationCriteria*: validation criteria needed to test the requirement (included in the STS document).
- *Responsible*: person responsible for the implementation of the requirement.
- *Section*: document section in which the requirement is specified.
- *VersionLog*: historical record of all versions of the requirement (including author, date, version number and text).
- *RequestedBy* (compulsory): user asking for the inclusion of the requirement (particularly useful when the person who elicited the requirement is not in the team in charge of implementing it).
- *SourceTeam* (compulsory): identifier of the team that elicited or reused the requirement.

- *SourceAnalyst* (compulsory): analyst who drew up or reused the requirement.
- *DiscussionThread*: link to the identifier of the agreements of the meetings in which the requirement was discussed (empty until IRD 7).
- *DestinationTeam*: identifier of the team which has been assigned the requirement for refinement (empty until IRD 8).

PANGEA includes the concept of parametrised requirement. The parameter allows us to specify a variation point in the requirements specification, i.e. when it is necessary to choose between various alternatives in order to configure a specific product. For instance, "the system shall make it possible to export to external files for report generation in [Format]", where the value of the parameter "Format" would be comma-separated values (CSV) or Excel.

The traceability model defined in PANGEA includes a set of traceability relations that enable different links to be established between requirements. Such traces are described below:

- *Parent–child*: relationship describing a more general requirement by a sequence of more specific requirements.
- *Requires*: directional dependency relationship between two requirements. A *Requires* B means that the reuse of A necessarily involves the reuse of B.
- *RelatedTo*: bidirectional dependency relationship between two requirements. A is *RelatedTo* B if (1) B refines or supplements A in some way, so that when A is reused, then B should also be considered for reuse; or (2) A and B belong to the same cluster of requirements.
- *MutuallyExclusive*: mutually exclusive relationship between two requirements. A *MutuallyExclusive* B means that if A is present in the specification, then B cannot be, and vice versa.
- *Reifies*: relationship between a requirement and an artefact of the development process in which it materialises (e.g. class, module, component, etc.).
- *DiscussionThread*: relationship between a requirement and the negotiation meeting in which it was discussed, recorded in the minutes of the meeting and the object *DiscussionThread* that contains the complete record of the discussion, together with the results of any vote taken.

A repository of reusable requirements arranged into catalogues for managing requirements knowledge is included in PANGEA. These catalogues can be (1) *domains*, "vertical" application domains (e.g. insurance or banking), or (2) *profiles*, "horizontal" application domains (e.g. security or personal data protection). Moreover, they are organised in a hierarchy of requirements documents, which, in turn, are structured according to standards (IEEE Std. 1233, IEEE Std. 12207.1, and IEEE Std. 830).

With regard to the reuse of requirements, once the scope of the project has been established, the requirements repository should be searched to find a domain catalogue within that area. If so, the project probably corresponds with the development of a particular product in the domain specified in the catalogue. In this case, the searches in the repository might begin with the requirements with the highest value in the attribute *Criticality* in the domain catalogue. Since these requirements are mandatory, they are part of any product for that domain, and as such all of them should be reused. Their traces should be analysed to determine other non-mandatory requirements which ought also to be part of the specification of the current project. After the common specification of the product is established, new searches should be defined, guided by the business requirements and the features or system objectives identified. These guidelines may involve searching the same domain catalogue or different profile catalogues. If, as a result of any of the searches, a requirement is reused that has trace relations of type *RelatedTo* other requirements, then a cluster of requirements is found. In this case, we should consider the selection of the requirements within that cluster. In summary, reusing the requirements selected involves the resolution of the variation points found in them: (1) instantiation of the parameters of the parametrised requirements with values appropriate to the current project; (2) resolution of *MutuallyExclusive* traces; (3) resolution of *RelatedTo* traces, which are optional; and (4) resolution of *Requires* and *Parent–child* traces, which should normally be included.

## 8.5 The Tool Architecture

As is shown in Sect. 8.3, literature reflects the need for RE tools that support GSE. In this respect, the PANGEA method is supported by PANTALASA (*PANgea Tool And Lightweight Automated Support Architecture*), which is the underlying tool architecture for the global RE processes and models. Current RE tools' capabilities [65] and the ISO TR 24766 [66] have been taken into account in its conception.

PANTALASA is responsible for managing the requirements knowledge in a coherent way, giving a boost to reuse, automatising some repetitive tasks and, in short, facilitating the distributed stakeholders' activities carried out within the framework of PANGEA. Among its features, PANTALASA allows multiple users to edit the same requirements document simultaneously, so when a user reuses a requirement from a catalogue of the requirements repository, the tool automatically has to keep track of all the existing relationships, i.e. the tool will check if the parent requirement has to be included, if other requirements are involved or if the reused requirement or its dependencies violate any exclusive relationship with any requirement previously added.

By means of these automated verification mechanisms, it is ensured that the requirements are consistent at all times and that there is a single shared working document for all stakeholders, promoting proper control and coordination of the team members' work. Moreover, if different analysts introduce the same parametrised requirement and assign different values to the same parameter, when the second analyst attempts to insert the troublesome requirement in the requirements document, the tool will detect the inconsistency and will notify the analysts involved about such a situation. These analysts can then discuss what the correct value of the parameter is, and, if necessary, they can take the matter to the agenda of the next requirements analysis and negotiation meeting (IRD 7).

With regard to specific technologies that might be particularly useful for PANTALASA, we considered the use of semantic wikis and social networks, because we believe that they turn out to be complementary. Semantic wikis are organised around an ontology of requirements, so that the requirements repository is structured more consistently than in a *plain* wiki, whereas social networks leverage the communication strategies between project members. In this regard, wikis were originally conceived for distributed collaborative content creation [67], but it is possible to use them to capture requirements and domain knowledge [67, 68] or even to support AKM in GSE [50], improving domain knowledge reuse and tacit knowledge acquisition [68]. Furthermore, as a result of their underlying information models, semantic wikis can provide support to reasoning with the requirements knowledge [50, 69]. This is especially relevant for traceability approaches and also enables automated information retrieval [50]. On the other hand, Whitehead et al. [70] raised the possible application of new trends in networking and social networks to improve formal and informal communication. Following this trend, Lim et al. [71] at first proposed a social network system for stakeholder analysis and later, an extension of this tool was developed to identify and prioritise software requirements [72].

## 8.6 Prototype Implementation and Validation

We have recently developed an automated tool support proposal for PANGEA by means of the integration of (1) a well-known social network like Facebook,[1] which integrates both synchronous and asynchronous communication, and that serves to establish and strengthen trust relationships between distributed software development teams, and (2) a semantic wiki like Semantic MediaWiki (SMW),[2] which is suitable for supporting the PANGEA requirements reference model and the reusable-requirements repository, taking into account issues such as security, concurrency and discussion threads. Some of the KM challenges previously identified in Sect. 8.3, such as communication, knowledge capture and knowledge transfer issues, are therefore addressed by relying on such collaborative and open source technologies.

A block diagram of the prototype is shown in Fig. 8.3. Users connect with the application in an automatic and transparent manner through Facebook. The application serves as a connection bridge to SMW, where the requirements repository is located. Figure 8.4 shows a SMW page in the Facebook interface for enabling distributed stakeholders' work.

[1] http://www.facebook.com.

[2] http://semantic-mediawiki.org.

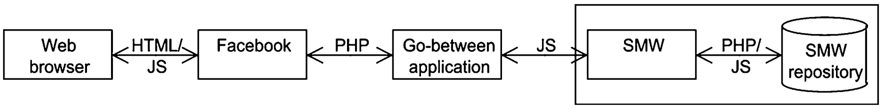
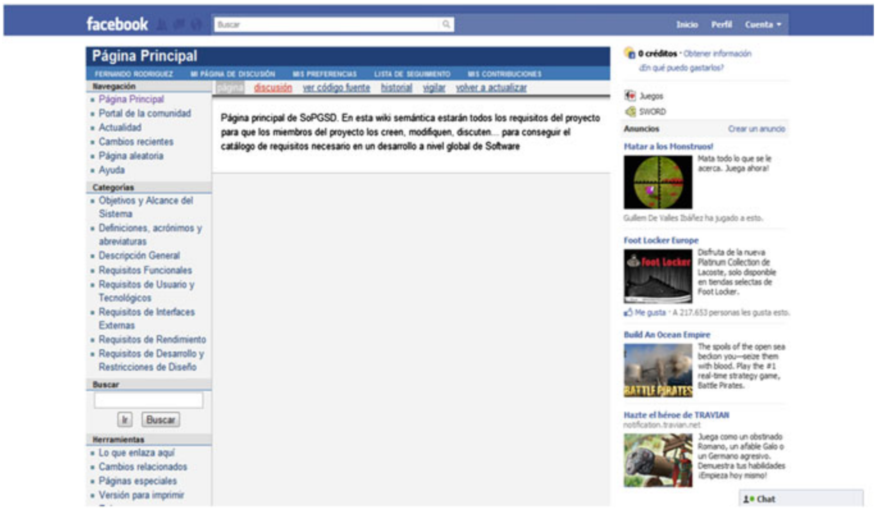


**Fig. 8.3** Free-form architecture diagram of the prototype



Página principal de SoPGSD. En esta wiki semántica estarán todos los requisitos del proyecto para que los miembros del proyecto los creen, modifiquen, discuten... para conseguir el catálogo de requisitos necesario en un desarrollo a nivel global de Software

Matar a los Monstruos!
Mata todo lo que se le acerca. Juega ahora!
Guillem De Valles Ibáñez ha jugado a esto.

Foot Locker Europe
Disfruta de la nueva Platinum Collection de Lacoste, solo disponible en tiendas selectas de Foot Locker.
Me gusta · A 217.653 personas les gusta esto.

Build An Ocean Empire
The spoils of the open sea beckon you—seize them with blood. Play the #1 real-time strategy game, Battle Pirates.

Hazte el héroe de TRAVIAN
notification.travian.net
Juega como un obstinado Romano, un afable Galo o un Germano agresivo. Demuestra tus habilidades ¡Empieza hoy mismo!

**Fig. 8.4** Graphical user interface (GUI) of the prototype

Facebook offers an application programming interface (API) to support the execution of external applications. In addition, these applications have authentication mechanisms at their disposal, so it is possible to access personal data such as name, email, friends list or even wall comments. If the user gives permission for the external application, Facebook is able to send to our application the user information needed. The aim is to provide a personalised experience, but a user authorisation process must be carried out to make that possible.

This implementation of PANTALASA includes an external, go-between application, whose main task is to communicate Facebook and SMW (see Fig. 8.3). Hence, this piece of software allows us to perform the following duties in a transparent manner, automatically: (1) gather the data of a Facebook user, (2) register Facebook users in the SMW database, (3) authenticate users in SMW, (4) configure SMW in order to adapt it to the use requested by PANGEA and (5) execute SMW under the Facebook applications interface.

It is important to note that the validation of the prototype was conducted in an academic environment by college students who worked with the application in a distributed and collaborative way, encoding requirements catalogues in the prototype, reusing the requirements in a new project and taking advantage of the Facebook capabilities for supporting communication between them. More detailed information about the validation is provided below.

Two students participated in the study with the intention of validating the prototype; this lasted for 2 weeks. They introduced three requirements catalogues in the requirements repository. The catalogues were made up of both functional and non-functional requirements. Two of these catalogues were *profiles*, as they codified security requirements and personal data protection requirements based on the respective original catalogues developed for SIREN [59, 62]. The other catalogue, on the other hand, was a *domain*, since it included domain-specific requirements extracted from a case study on the field of hotel management, which was presented in a course on RE belonging to a degree programme on computer science and engineering. The full case study was composed of approximately 150 requirements, and about 50 of these were inserted in the system.

Note that the mentioned catalogues are located in the SMW requirements repository. All the requirements that make up these three catalogues are thus available for reuse in new projects that can be created with the tool, following the proposed method. In fact, the students created a new project in which specific requirements were extracted from the catalogues listed above in order to check the reuse functionality of the tool. Such a project was aimed at the development of a software application for a particular hotel. In this project, the different sections of the SRS were generated, including the functional requirements of the project, which were mostly reused from the functional requirements of the hotel industry catalogue.

The feedback after using the system was mainly satisfactory. However, some difficulties in reusing the requirements from the catalogues in the new project were reported; these problems come about from technical concerns related to SMW, since the tool is still in an incipient stage. In addition, the students worked in a distributed manner, but not in a global environment, since they were located in different cities about 100 km apart. Moreover, they have reported neither communication nor concurrency problems.

From our point of view, the main limitation of the validation of the prototype is the fact that it was conducted by students. Nevertheless, students play a very important role in software engineering experimentation, because before performing studies in industrial environments, which requires a lot of resources and time, it is generally useful for researchers to carry out pilot studies with students in academic environments [73, 74]. In addition, students are the next generation of professionals [75], and under some conditions, there is not a great difference between students and professionals. In situations in which the tasks which are to be performed do not require industrial experience, experimentation with students is viable [76, 77]. Another relevant threat to the validity of the study is that the students were not globally distributed. Nonetheless, this was to some extent mitigated by the fact that they were not collocated, and even relatively small geographic distances between offices profoundly affect the ability to collaborate [10]. Indeed, Allen [78] found that there is a strong negative correlation between physical distance and frequency of communication between sites and any distance greater than the critical threshold of about 50 m led to a dramatic drop in spontaneous communication and collaboration between individuals.

## 8.7 Summary

Since one of the current problems in GSE is the existence of knowledge which is not properly shared and reused, thus hampering awareness, KM in global, distributed settings is a challenging task. It can be faced by improving reuse, sharing and collaboration in global RE. To the best of our knowledge, there is no proposal which tackles both GSE and requirements reuse. In this chapter, we have presented PANGEA, a reuse-based RE method for GSE that specifies knowledge in the form of natural language requirements. PANGEA encompasses a process model, a requirements reference model and PANTALASA, the supporting tool architecture. PANTALASA has been developed by means of (1) a semantic wiki, in an effort to implement a reusable-requirements repository, and (2) a social network, to improve communication issues. Before describing PANGEA and PANTALASA, this chapter provides brief insights into the relationship between KM and (global) RE, as well as the practical challenges concerning RE and KM in GSE.

Ebert and De Man [22] state that a software company or department is confronted with many challenges that must be mastered through continuous improvement, along the following axes: (1) *consolidating*, focusing on a few essential products and maximising their business value; (2) *industrialising*, mastering projects, processes and knowledge by intelligent collaboration to improve predictability, repeatability and affordability; and (3) *globalising*, depending on the needs of the target market and the size of the company, but its success relies on the other two axes. An approach has been presented in this chapter that leverages this continuous improvement strategy by means of proper management of requirements knowledge. Firstly, an organisation that usually develops products in a domain eventually has enough expertise to generate high-quality requirements catalogues dealing with common issues in that domain (consolidating). Secondly, once such an organisation manages domain knowledge appropriately by means of requirements catalogues, the entire software development process benefits and is greatly improved in terms of cost, time and effort (industrialising). Finally, the particularities of globalisation are taken into account and its demands materialised in the RE process in order to be successful in a global environment (globalising).

Future work includes research on data mining techniques that can be applied to requirements, the aim being to build prediction models and help developers make better decisions on the subsequent stages of the software development process. We are also interested in supporting project management issues by relying on RE, so that project management and decision making processes within the organisation could take advantage of explicit or derived requirements knowledge. Finally, an academic case study between the University of Murcia (Murcia, Spain), the University of Castilla-La Mancha (Ciudad Real, Spain) and the University Mohammed V – Souissi (Rabat, Morocco) is planned, with the intention of validating our proposal in a nearshore environment. We are also planning to conduct a case study in a real industry environment later on, in a subsequent stage of the validation.

**Acknowledgments** This work has been funded by the PEGASO/PANGEA project (TIN2009-13718-C02-02), the ORIGIN Integrated Project (IDI-2010043 (1–5)) and the ENGLOBAS Project (PII2I09-0147-8235).

## References

1. Brooks FP Jr (1987) No silver bullet: essence and accidents of software engineering. IEEE Comp 20:10–19
2. Meyer B (1997) Object-oriented software construction, 2nd edn. Prentice-Hall, New York
3. Mili H, Mili F, Mili A (1995) Reusing software: issues and research directions. IEEE Trans Softw Eng 21:528–562
4. Cybulski JL, Reed K (2000) Requirements classification and reuse: crossing domain boundaries. In: Proceedings of the 6th international conference on software reuse: advances in software reusability, Vienna, pp 190–210
5. Sommerville I (2004) Software engineering, 7th edn. Pearson Addison Wesley, Boston
6. Favaro J (2002) Managing requirements for business value. IEEE Softw 19:15–17
7. Robertson S, Robertson J (2006) Mastering the requirements process, 2nd edn. Addison-Wesley, Upper Saddle River
8. Cheng BHC, Atlee JM (2007) Research directions in requirements engineering. In: Future of software engineering, IEEE Computer Society, Minneapolis, USA, pp 285–303
9. Rine DC, Nada N (2000) An empirical study of a software reuse reference model. Inf Softw Technol 42(1):47–65
10. Herbsleb JD (2007) Global software engineering: the future of socio-technical coordination. In: Future of software engineering, IEEE Computer Society, Minneapolis, USA, pp 188–198
11. Damian D, Moitra D (2006) Global software development: how far have we come? IEEE Softw 23:17–19
12. Noll J, Beecham S, Richardson I (2010) Global software development and collaboration: barriers and solutions. ACM Inroads 1(3):66–78
13. Damian D (2007) Stakeholders in global requirements engineering: lessons learned from practice. IEEE Softw 24:21–27
14. Sinha V, Sengupta B, Chandra S (2006) Enabling collaboration in distributed requirements management. IEEE Softw 23:52–61
15. Gallardo-Valencia RE, Sim SE (2009) Continuous and collaborative validation: a field study of requirements knowledge in agile. In: Proceedings of the 2nd international workshop on managing requirements knowledge, IEEE Computer Society, Atlanta, USA, pp 65–74
16. Manteli C, van den Hooff B, Tang A, van Vliet H (2011) The impact of multi-site software governance on knowledge management. In: Proceedings of the 6th IEEE international conference on global software engineering, IEEE Computer Society, Helsinki, Finland, pp 40–49
17. Aurum A, Jeffery R, Wohlin C, Handzic M (eds) (2003) Managing software engineering knowledge. Springer, Berlin
18. Rus I, Lindvall M (2002) Knowledge management in software engineering. IEEE Softw 19:26–38
19. Ebling T, Nicolas Audy JL, Prikladnicki R (2009) A systematic literature review of requirements engineering in distributed software development environments. In: Proceedings of the 11th international conference on enterprise information systems, Milan, Italy, pp 363–366
20. Berenbach B (2006) Impact of organizational structure on distributed requirements engineering processes: lessons learned. In: Proceedings of the international workshop on global software development for the practitioner, ACM, Shanghai, China, pp 15–19

21. Al-Ani B (2010) Questions regarding knowledge engineering and management. In: Proceedings of the 5th IEEE International conference on global software engineering, IEEE Computer Society, Princeton, USA, pp 324–329
22. Ebert C, De Man J (2008) Effectively utilizing project, product and process knowledge. Inf Softw Technol 50(6):579–594
23. Ågerfalk PJ, Fitzgerald B, Holmström H, Lings B, Lundell B, Conchúir EO (2005) A framework for considering opportunities and threats in distributed software development. In: Proceedings of the international workshop on distributed software development, Austrian Computer Society, Paris, France, pp 47–61
24. Portillo Rodríguez J, Ebert C, Vizcaíno A (2010) Technologies and tools for distributed teams. IEEE Softw 27:10–14
25. Portillo Rodríguez J, Vizcaíno A, Ebert C, Piattini M (2010) Tools to support global software development processes: a survey. In: Proceedings of the 5th IEEE international conference on global software engineering, IEEE Computer Society, Princeton, USA, pp 13–22
26. Edwards JS (2003) Managing software engineers and their knowledge. In: Aurum A, Jeffery R, Wohlin C, Handzic M (eds) Managing software engineering knowledge. Springer, Berlin, pp 5–27
27. Lethbridge TC, Singer J, Forward A (2003) How software engineers use documentation: the state of the practice. IEEE Softw 20:35–39
28. Ebert C, De Neve P (2001) Surviving global software development. IEEE Softw 18(2):62–69
29. Mika P (2007) Social networks and the semantic web, Semantic web and beyond. Springer, New York
30. Beecham S, Noll J, Richardson I, Ali N (2010) Crafting a global teaming model for architectural knowledge. In: Proceedings of the 5th IEEE international conference on global software engineering, IEEE Computer Society, Princeton, USA, pp 55–63
31. Ali N, Beecham S, Mistrík I (2010) Architectural knowledge management in global software development: a review. In: Proceedings of the 5th IEEE international conference on global software engineering, IEEE Computer Society, Princeton, USA, pp 347–352
32. Hall JG, Jackson M, Laney RC, Nuseibeh B, Rapanotti L (2002) Relating software requirements and architectures using problem frames. In: Proceedings of the 10th anniversary IEEE joint international conference on requirement engineering, IEEE Computer Society, Essen, Germany, pp 137–144
33. Clerc V (2008) Towards architectural knowledge management practices for global software development. In: Proceedings of the 3rd international workshop on sharing and reusing architectural knowledge, ACM, Leipzig, Germany, pp 23–28
34. López A, Nicolás J, Toval A (2009) Risks and safeguards for the requirements engineering process in global software development. In: Proceedings of the 4th IEEE international conference on global software engineering, IEEE Computer Society, Limerick, Ireland, pp 394–399
35. Desouza KC, Awazu Y, Baloh P (2006) Managing knowledge in global software development efforts: issues and practices. IEEE Softw 23:30–37
36. Kucza T, Nättinen M, Parviainen P (2001) Improving knowledge management in software reuse process. In: Proceedings of the 3rd international conference on product focused software process improved, Kalserslautern, pp 141–152
37. Schneider K, von Hunnius JP, Basili V (2002) Experience in implementing a learning software organization. IEEE Softw 19:46–49
38. Seaman CB, Mendonça MG, Basili VR, Kim YM (2003) User interface evaluation and empirically-based evolution of a prototype experience management tool. IEEE Trans Softw Eng 29:838–850
39. Maalej W, Thurimella AK, Happel HJ, Decker B (2008) Managing requirements knowledge (MaRK'08). In: Proceedings of the 1st international workshop on managing requirement knowledge, IEEE Computer Society, Barcelona, Spain, pp i–ii
40. Ma L, Nuseibeh B, Piwek P, Roeck AD, Willis A (2009) On presuppositions in requirements. In: Proceedings of the 2nd international workshop on managing requirement knowledge, IEEE Computer Society, Atlanta, USA, pp 27–31

41. Damian D, Zowghi D (2002) The impact of stakeholders' geographical distribution on managing requirements in a multi-site organization. In: Proceedings of the 10th anniversary IEEE joint international conference on requirements engineering, IEEE Computer Society, Essen, Germany, pp 319–330
42. Kwan I, Damian D, Marczak S (2007) The effects of distance, experience, and communication structure on requirements awareness in two distributed industrial software projects. In: Proceedings of the 1st international global requirements engineering workshop, Munich, Germany, pp 29–35
43. Bhat JM, Gupta M, Murthy SN (2006) Overcoming requirements engineering challenges: lessons from offshore outsourcing. IEEE Softw 23:38–44
44. Hsieh Y (2006) Culture and shared understanding in distributed requirements engineering. In: Proceedings of the IEEE international conference on global software engineering, IEEE Computer Society, Florianopolis, Brazil, pp 101–108
45. Betz S, Oberweis A, Stephan R (2010) Knowledge transfer in IT offshore outsourcing projects: an analysis of the current state and best practices. In: Proceedings of the 5th IEEE international conference on global software engineering, IEEE Computer Society, Princeton, USA, pp 330–335
46. Herbsleb JD, Mockus A (2003) An empirical study of speed and communication in globally distributed software development. IEEE Trans Softw Eng 29:481–494
47. Szulanski G, Winter S, Grant R, Spender JC, Kogut B, Miner A, Ghoshal S (2000) The process of knowledge transfer: a diachronic analysis of stickiness. Organ Behav Hum Dec Proc 82:9–27
48. Carmel E, Agarwal R (2001) Tactical approaches for alleviating distance in global software development. IEEE Softw 18:22–29
49. Correia FF, Aguiar A (2009) Software knowledge capture and acquisition: tool support for agile settings. In: Proceedings of the 4th international conference on software engineering advanced, IEEE Computer Society, Limerick, Ireland, pp 542–547
50. Clerc V, de Vries E, Lago P (2010) Using wikis to support architectural knowledge management in global software development. In: Proceedings of the ICSE workshop on sharing and reusing architectural knowledge, ACM, Cape Town, South Africa, pp 37–43
51. Oshri I, van Fenema PC, Kotlarsky J (2008) Knowledge transfer in globally distributed teams: the role of transactive memory. Inf Syst J 18(6):593–616
52. Lee SB, Shiva SG (2010) An approach to overcoming knowledge sharing challenges in a corporate IT environment. In: Proceedings of the 5th IEEE international conference on global software engineering, IEEE Computer Society, Princeton, USA, pp 342–346
53. Kotlarsky J, Oshri I (2005) Social ties, knowledge sharing and successful collaboration in globally distributed system development projects. Eur J Inf Syst 14:37–48
54. Ebert C (2012) Global software and IT: a guide to distributed development, projects, and outsourcing. Wiley, Hoboken
55. Heindl M, Reinisch F, Biffl S (2007) Requirements management infrastructures in global software development – Towards application lifecycle management with role-based in-time notification. In: Proceedings of the international conference on global software engineering (ICGSE), workshop on tool-supported requirements management in distributed projects (REMIDI), Munich, Germany
56. Mistrík I, Grundy J, Van der Hoek A, Whitehead J (2010) Collaborative software engineering: challenges and prospects. In: Mistrík I, Grundy J, Van der Hoek A, Whitehead J (eds) Collaborative software engineering. Springer, Berlin/Heidelberg, pp 389–403
57. Laurent P (2010) Globally distributed requirements engineering. In: Proceedings of the 5th IEEE international conference on global software engineering, IEEE Computer Society, Princeton, USA, pp 361–362
58. Monteiro MR, Ebert C, Recknagel M (2009) Improving the exchange of requirements and specifications between business partners. In: Proceedings of the 17th IEEE international requirements engineering conference, IEEE Computer Society, Atlanta, USA, pp 253–260

59. Toval A, Nicolás J, Moros B, García F (2002) Requirements reuse for improving information systems security: a practitioner's approach. Requir Eng 6:205–219
60. Sommerville I, Ransom J (2005) An empirical study of industrial requirements engineering process assessment and improvement. ACM Trans Softw Eng Methodol 14:85–117
61. Toval A, Moros B, Nicolás J, Lasheras J (2008) Eight key issues for an effective reuse-based requirements process. Comp Syst Sci Eng 23:1–13
62. Toval A, Olmos A, Piattini M (2002) Legal requirements reuse: a critical success factor for requirements quality and personal data protection. In: Proceedings of the 10th anniversary IEEE joint international conference on requirement engineering, IEEE Computer Society, Essen, Germany, pp 95–103
63. Martínez MA, Lasheras J, Fernández-Medina E, Toval A, Piattini M (2010) A personal data audit method through requirements engineering. Comp Stand Interf 32:166–178
64. Nicolás J, Lasheras J, Toval A, Ortiz FJ, Álvarez B (2009) An integrated domain analysis approach for teleoperated systems. Requir Eng 14:27–46
65. Carrillo de Gea JM, Nicolás J, Fernández Alemán JL, Toval A, Ebert C, Vizcaíno A (2011) Requirements engineering tools. IEEE Softw 28(4):86–91
66. ISO/IEC JTC 1 SC 7: ISO/IEC TR 24766 (2009) Information technology – systems and software engineering – guide for requirements engineering tool capabilities, 1st edn. ISO, Geneva
67. Uenalan O, Riegel N, Weber S, Doerr J (2009) Using enhanced wiki-based solutions for managing requirements. In: Proceedings of the 2nd international workshop on managing requirement knowledge, IEEE Computer Society, Atlanta, USA, pp 63–67
68. Ugai T, Aoyama K (2009) Domain knowledge wiki for eliciting requirements. In: Proceedings of the 2nd international workshop on managing requirement knowledge, IEEE Computer Society, Atlanta, USA, pp 4–6
69. Liang P, Avgeriou P, Clerc V (2009) Requirements reasoning for distributed requirements analysis using semantic wiki. In: Proceedings of the 4th IEEE international conference on global software engineering, IEEE Computer Society, Limerick, Ireland, pp 388–393
70. Whitehead J, Mistrík I, Grundy J, Van der Hoek A (2010) Collaborative software engineering: concepts and techniques. In: Mistrík I, Grundy J, Van der Hoek A, Whitehead J (eds) Collaborative software engineering. Springer, Berlin/Heidelberg, pp 1–30
71. Lim SL, Quercia D, Finkelstein A (2010) StakeSource: harnessing the power of crowdsourcing and social networks in stakeholder analysis. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, ACM, Cape Town, South Africa, pp 239–242
72. Lim SL, Damian D, Finkelstein A (2011) StakeSource2.0: using social networks of stakeholders to identify and prioritise requirements. In: Proceedings of the 33rd international conference on software engineering, Waikiki, pp 1022–1024
73. Carver J, Jaccheri L, Morasca S, Shull F (2003) Issues in using students in empirical studies in software engineering education. In: Proceedings of the 9th IEEE international symposium on software metrics, IEEE Computer Society, Sydney, Australia, pp 239–249
74. Carver J, Jaccheri L, Morasca S, Shull F (2003) Using empirical studies during software courses. In: Conradi R, Wang A (eds) Empirical methods and studies in software engineering, Lecturer notes in computer science, vol 2765, Springer, Berlin/Heidelberg, pp 81–103
75. Kitchenham BA, Pfleeger SL, Pickard LM, Jones PW, Hoaglin DC, Emam KE, Rosenberg J (2002) Preliminary guidelines for empirical research in software engineering. IEEE Trans Softw Eng 28:721–734
76. Basili VR, Shull F, Lanubile F (1999) Building knowledge through families of experiments. IEEE Trans Softw Eng 25:456–473
77. Svahnberg M, Aurum A, Wohlin C (2008) Using students as subjects – An empirical evaluation. In: Proceedings of the 2nd ACM-IEEE international symposium on empirical software engineering & measurement, ACM, Kaiserslautern, Germany, pp 288–290
78. Allen T (1977) Managing the flow of technology. MIT Press, Cambridge, MA

# Chapter 9
# Performative and Lexical Knowledge Sharing in Agile Requirements

**S.E. Sim and R.E. Gallardo-Valencia**

**Abstract** We present the results of our field study that describe how requirements knowledge was shared at an industrial software company using agile software practices. As is common in agile processes, the team did not capture requirements knowledge in a comprehensive specification document. Instead, requirements knowledge was captured in user stories, automated acceptance tests, personal notes, and conversations. We identified two modes of knowledge sharing: performative and lexical. Performative knowledge, which occurs through actions such as question-asking, gestures, and informal speeches, was observed in conversations and at the Scrum board. Lexical knowledge sharing, which occurs through inscribed texts, was observed in testing wiki and the software release documents. We found that the software team relied mainly on performative knowledge sharing. Although team members shared few written documents, they were able to effectively develop software that satisfied customer requirements. Results from our field study have implications for both agile practitioners and knowledge management. The former could encourage question-asking to provide opportunities for performative knowledge sharing. The latter could pay attention to personal management so that users can more effectively engage in performative knowledge sharing.

## 9.1 Introduction

Socrates is well known for many contributions to ancient Greek philosophy. However, he is perhaps less well known as an early theorist of knowledge management. The dialogue *Phaedrus* is an account of a conversation between him and the

S.E. Sim (✉) • R.E. Gallardo-Valencia
University of California, Irvine, CA, USA
e-mail: ses@manyroadsstudios.com; gallardo.re@gmail.com

man for whom the text is named [12]. The ostensive topic of the dialogue is the nature of love and friendship. At a deeper level, the dialogue is about the merits of the various ways to impart knowledge: oration, dialectic, and written texts. In Ancient Greece, the conventional means to acquire knowledge was to listen to a prepared, rehearsed speech, or oration, by a learned man and to memorise the speech word for word so that it can be repeated at a later occasion. The dialogue, as a method of sharing knowledge, is Socrates's contribution; we now know this as the Socratic method. Written texts were growing in quantity and adoption, enabled by a new technology called the alphabet.

Socrates was not fond of oration, because speeches are static and more suitable for persuasion and entertainment rather than the search for ultimate truth. But he reserves his most scathing criticism for written texts, saying 'Then [a philosopher] will not seriously incline to "write" his thoughts "in water" with pen and ink, sowing words which can neither speak for themselves nor teach the truth adequately to others?' The dialectic, he argues, is a far superior method for imparting knowledge:

> But nobler far is the serious pursuit of the dialectician, who, finding a congenial soul, by the help of science sows and plants therein words which are able to help themselves and him who planted them, and are not unfruitful, but have in them a seed which others brought up in different soils render immortal...

In other words, only the dialectic, or dialogue, is the only method of teaching that ensures the received knowledge will be useful and productive. This is a central concern of knowledge management even today. Not content with the sharing of preprocessed data (information), his goal is the sharing of information in context to produce an actionable understanding (knowledge).

Socrates's mistrust of written texts seems quaint in our modern age where we are surrounded by documents, post-it notes, and web pages. Nevertheless, it prompts us to consider the differences between the written word and dialogues as means for sharing requirements knowledge. Our insights come from a field study of a team using agile software development techniques. In this context, agile is significant, because the creation of 'comprehensive documentation' is not a priority and as a result there is a greater reliance on other forms of knowledge sharing.

We undertook the study with the aim of better understanding how and why agile software development teams worked. We were particularly interested in requirements techniques, specifically user stories, because they seemed too simple and under-specified to work effectively. Yet our initial research indicated that developers, customers, and executives were able to use these and were highly satisfied with them.

The team in our study used a variety of agile techniques, in particular Scrum [13], user stories [4], and automated acceptance testing [1, 11]. They captured requirements knowledge in user stories, improvised checklists, and automated acceptance test cases. The written component of these artefacts and practices was minimal; they served only as reminders. Knowledge sharing occurred primarily through ongoing conversations, meetings, and feedback.

From our observations, we identified two forms of knowledge sharing: lexical and performative. Lexical knowledge sharing occurs through inscribed texts. This label was derived from semiotics, where the lexicon is the set of things in an ontology [3]. Performative knowledge sharing occurs through actions, such as question-asking and gestures, as well as informal speeches. This label follows from work in the humanities and social sciences that analysed human behaviour using the heuristic of performance [10].

We found that software development relied much more on performative sharing of requirements knowledge than lexical. In other words, developers relied almost entirely on conversations to share knowledge. The few documents that did exist were personal and generally not shared with the team. Our analysis is based on careful study of four requirements activities (elicitation, modelling, communication, and validation) during three stages of the iteration (pre-iteration, iteration planning, and intra-iteration). We examined two cultural sites for performative knowledge sharing, conversations and the Scrum (project status) board, and contrast these with current and past sites for lexical knowledge sharing. In our discussion, we identify implications for both agilists and researchers, including directions for future research.

## 9.2 Method

This research applied qualitative field methods to understand how a team of software developers manage requirements knowledge. We were interested in how the various stakeholders made use of user stories in requirements engineering. Consequently, our data collection was focused on the practices and artefacts around this software development activity. We were also interested in comparing the current practices and artefacts with ones that the team used to use or were recommended by software engineering best practices.

Our field site was ‘Easy Retirement’, an Internet-based service provider for self-directed retirement investment plans. The company’s main product is a web application that allows individuals to manage their own retirement investment plans and is sold as a service to companies that are legally required to provide this benefit to employees. Easy Retirement has a total of 26 employees, and the software team consists of ten members. We observed the software team for 2 days in December 2007. This included observations of stand-up and iteration planning meetings. We also conducted six semi-structured interviews that lasted between 30 and 68 min. Our interview participants had various roles in the company, including the Scrum Master, Product Owner, two programmers, a tester, and the owner of the company.

We also collected some samples of artefacts by taking photographs of artefacts in use, as well as collecting older artefacts (user story cards, checklists, forms, and test cases) that were no longer in use. We used pseudonyms for names of people and places to protect the privacy of individuals who have participated in this research.

After completing data collection, we transcribed our field notes and audio recordings of the interviews. We analysed the data inductively and iteratively [9]. We used open coding to identify categories, sometimes revisiting data to apply new categories. We used axial coding to relate different categories to each other to create descriptions.

## 9.3 Field Site: Easy Retirement

We conducted a field study at ‘Easy Retirement’ (a pseudonym), an Internet-based 401 k service provider. The company develops a web application which helps users to manage their own retirement investment plans.

The software team follows a number of agile methods closely, including Scrum, daily stand-up meetings, user stories, continuous integration, on-site customer, and automated acceptance testing. Each sprint or iteration lasts 2 weeks and starts and ends on a Friday. On the first day of an iteration, the team holds a sprint planning meeting when user stories are broken down into tasks and estimated. During the sprint, programmers and testers work closely to complete the user stories and to ensure that all the user stories are accepted by the end of the sprint. It is a challenge to get everything done in an orderly fashion because testing and development are mutually dependent.

In this section, we give some background on agile and describe the agile techniques used at Easy Retirement.

### *9.3.1 Agile*

Agile software development is a family of modern software techniques that have been developed by practitioners. Their distinguishing feature is an emphasis on adapting to change by working in an incremental and iterative fashion, that is, taking many small steps repeatedly in order to grow a software system. The agile approach to software development contrasts with phased, sequential processes, as typified by the waterfall model, which seeks to minimise change through careful study and planning. The two best-known agile process models are Extreme Programming (XP) [2], which is more concerned with how software ought to be written, and Scrum [13], which is more concerned with how software projects ought to be managed. This array of techniques hold in common an adherence to the Manifesto for Agile Software Development [5], which states:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> Individuals and interactions over processes and tools
> Working software over comprehensive documentation
> Customer collaboration over contract negotiation
> Responding to change over following a plan
> That is, while there is value in the items on the right, we value the items on the left more.

Iterative and incremental development models have been around for decades [7], but users tended to be isolated, and techniques were invented locally or adopted piecemeal. The current move towards agile is marked by a large and growing community of users and advocates who are all focused on promoting and applying agile techniques. The flagship conference in 2007 was sold out with approximately 1,100 attendees, while the Agile 2011 Conference had over 1,600 attendees from around the world.

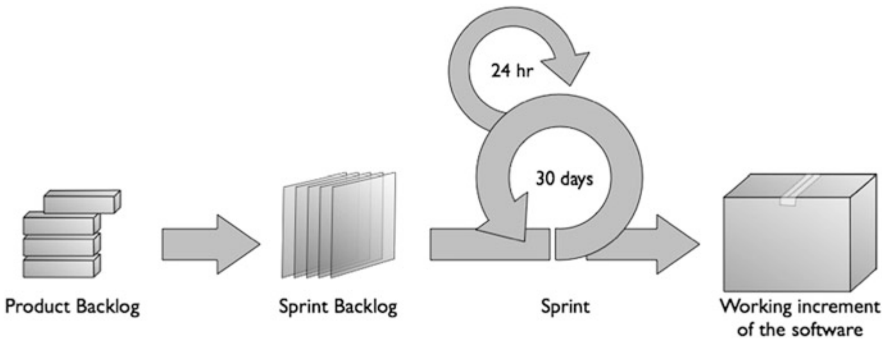


**Fig. 9.1** Diagram of the Scrum software development process model (Adapted from Wikipedia)

### *9.3.2 Scrum*

Easy Retirement uses Scrum to do planning and management of work. Rather than dividing the project into phases corresponding to activities, time is divided into increments, called a 'sprint' or iteration, where the objective is to produce working software that is another step closer to completion. The Scrum model is depicted in Fig. 9.1 above.

Agile seeks to have working but incomplete software from the first sprint. Sprints can vary in length from 1 to 4 weeks. In the figure, they are represented as 30 days. At Easy Retirement, the iterations lasted 2 weeks or ten working days, beginning and ending on a Friday.

Several sprints grouped together is called a 'release' and can correspond to a time window (e.g. a fiscal quarter) or a logical group of features indicating a level of achievement. A sprint begins with a sprint planning meeting in which items from the product backlog (called user stories) are selected and estimated for the upcoming time increment. This set of items becomes the sprint backlog. Every 24 h, the team synchronises their work in a 'daily stand-up meeting', which is a short meeting lasting no more than 15 min and is usually held at the beginning of the work day. A sprint concludes with a sprint review meeting to look back on the tactics that were supportive or detrimental to progress. Easy Retirement used all of these practices.

There are five primary roles on the Scrum team: Scrum Master, Product Owner, Team Members, Stakeholders, and Users [13]. These roles do not necessarily align with conventional job titles, for instance, a software developer could be a Scrum Master or Team Member. We will describe each of the roles and the people in the roles at Easy Retirement.

*Scrum Master*. This person facilitates software development by removing impediments and tracking the progress of the team. Typical tasks for the Scrum Master are convening the daily stand-up meeting, keeping a 'burn down chart' to record the tasks completed, maintaining the product backlog, and preparing for sprint planning meetings. Project managers are often Scrum Masters, but not always. There may be a technical manager as well as a Scrum Master. A software developer could be a Scrum Master, but a Scrum Master does not necessarily have coding skills. Usually there is one Scrum Master per project, but with very large projects, there can be a Scrum of Scrums with multiple levels of Scrum Masters.

Leanne is the Scrum Master at Easy Retirement, having served as an office manager before the switch to agile. While they tried out different people as the Scrum Master at first, she quickly settled into the role because, as she explained, she had always been a kind of liaison between software development and business.

*Product Owner*. This person represents business concerns and can be literally the person paying for the project or a figurative surrogate for customers. The Product Owner's main responsibilities are writing user stories and prioritising the product backlog. This role can be fulfilled by a senior manager, a product manager, someone involved in sales and marketing, a business analyst, or a user experience designer.

Sam is the regulatory compliance officer at Easy Retirement and also serves as the Product Owner. He has a background in law and business. His office is near the open area where the software developers work (and away from his peers in the company), so he can be consulted easily and frequently as the software is developed.

*Team Members*. This group of people is engaged in the work of creating the working software. Team members can include software developers, software testers, database analysts, system administrators, technical writers, and multimedia artists.

The software team at Easy Retirement consists of ten members including the Scrum Master, the Product Owner, a technical manager, four programmers, two testers, and a database/system administrator. Some agile methods, such as Extreme Programming, recommend that all 'developers' engage in programming and testing. Our field site does not fully embrace this practice. At Easy Retirement, both the programmers and testers call themselves developers. Programmers are responsible for writing code for features and for unit tests of those features. Testers are responsible for integration testing and verification and validation of requirements.

*Stakeholders*. Anyone with an interest in the software product is a stakeholder, but in practice, stakeholders are limited to those with business interests, such as customers and vendors. They tend to be involved only at release or sprint planning.

The stakeholders who typically attended the release planning meetings at Easy Retirement included the President, Vice Presidents, and Director of Marketing and Sales.

*Users*. A subset of the stakeholders are the Users of the software. Business stakeholders often are not Users. This role is included to remind the team that software should be built for someone to use. Individual users are usually not participants in the process like the roles but are consulted regularly to obtain feedback. This role can be fulfilled by someone who uses or will use the software or by a surrogate for them.

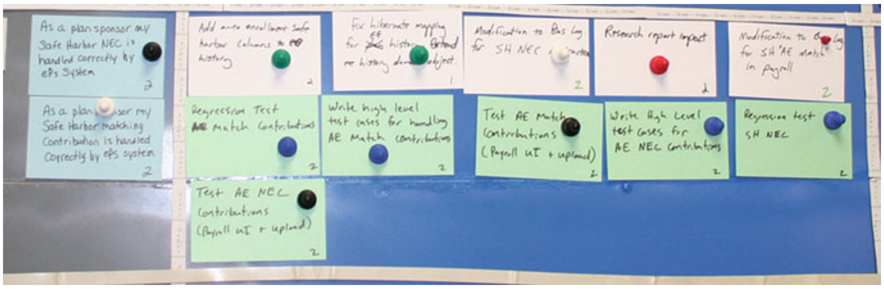

As a plan sponsor my Safe Harbor NEC is handled correctly by ePs System
As a plan sponsor my Safe Harbor matching Contribution is handled Correctly by ePs system
Research report impact
Regression Test AE Match Contributions
Write high level test cases for handling AE Match contributions
Test AE Match Contributions (Payroll UT + upload)
Write High Level test cases for AE NEC Contributions
Regression test SH NEC
Test AE NEC Contributions (Payroll UT + upload)

**Fig. 9.2** Example user story cards

### 9.3.3 *User Stories*

A user story has three parts: a written description of work to be done, conversations with the customer about the work, and the test cases [4]. A common format for the written description is ‘As a <role>, I can <action>, so that <goal>’. They are partially a software requirement and partially a to-do item. They should be small, so they can fit on a 3″ × 5″ index card and can be completed within a single sprint. In this formulation, user stories include both a lexical aspect (the written description) and a performative aspect (the conversations). It also includes a computational aspect (automated tests). Fig. 9.2 shows examples of user stories and associated tasks and tests cards.

The user stories were written on index cards and posted on a Scrum board for everyone to see. The board was large, covering an entire wall from floor to ceiling, and it was divided into four columns (to do, work in progress, to verify, and done). The placement of the index card indicates the status of the task.

In addition to the index cards, individuals also have their own representations that they use in their work. The Product Owner and Scrum Master each maintain a personal checklist in a spreadsheet. The testers store additional details in a wiki [8].

### 9.3.4 *Test-Driven Development*

Test-driven development (TDD) is an agile technique that involves creating test cases early, even before source code has been written [1]. Using an automated testing framework and running the tests often are key parts of this technique. A newly created test case should fail but pass after new application code is added.

During the sprint at Easy Retirement, testers start creating the test cases while programmers are adding new functionality. Testers use Fitnesse [11], an automated acceptance testing tool, to create the test cases. On the first Monday of the sprint, testers meet with the Product Owner to ask questions about the high-level requirements and to create acceptance tests. The programmers also use these tests to guide their design work. Programmers are expected to write their own unit tests and to write fixtures for Fitnesse.

Although the programmers were primarily responsible for unit tests, all team members were involved in creating test cases, especially for acceptance testing.

## 9.4 Requirements Process at Easy Retirement

In agile, requirements knowledge is both lexical and performative. While this coupling between physical artefacts and actions in time makes our data analysis more complicated, it is one of the strengths of agile. In other words, there is a web of collaborative and interlocking practices that provide a safety net for adaptive software development.

Consequently, we describe the requirements process at Easy Retirement using sequence diagrams, as shown in Figs. 9.3–9.5. By modelling the process in this manner, we are able to see both the lexical knowledge (artefacts) and performative knowledge (conversations and meetings).

These diagrams also facilitate our examination of knowledge sharing activities. The stick figures along the top of the diagrams represent the different roles involved in requirements. Horizontal arrows show knowledge being exchanged through questions and answers. Large boxes that are bordered with a dashed line depict meetings. Smaller rectangular boxes represent artefacts that are created or used. Down the left-hand side of the diagrams is a shorthand notation for the kinds of requirements activities taking place. The first letter shows whether the interaction is performed through a conversation (‘C’) or through other means (‘–’). The last four characters correspond to the four canonical activities in requirements engineering, elicitation (‘E’), modelling (‘M’), communication (‘C’), and validation (‘V’).

In our data analysis, we found that there were three time periods that correspond to different opportunities for requirements engineering. These stages were (1) pre-iteration shown in Fig. 9.3, (2) iteration planning shown in Fig. 9.4, and (3) intra-iteration shown in Fig. 9.5.

### *9.4.1 Pre-iteration*

Pre-iteration is the time leading up to the start of an iteration. During this stage, Sam, the Product Owner, received requirements from business people and clarified their expectations (arrows 1.1 and 1.2 in Fig. 9.3).

Everyone at Easy Retirement writes user stories, though some more than others. Leanne, the Scrum Master, focuses on how the user stories allow people in the company to feel that they are participating in the development of the software:

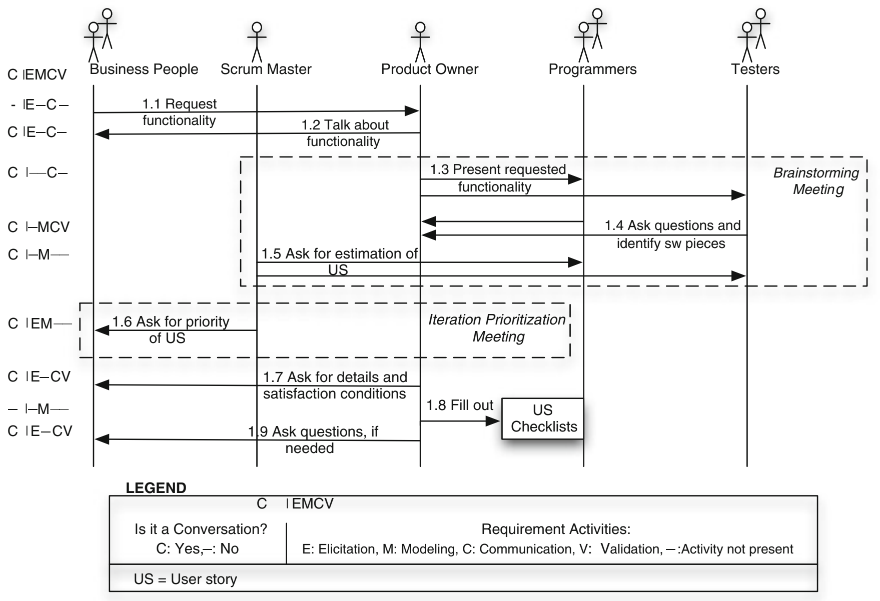

C I EMCV
Business People
Scrum Master
Product Owner
Programmers
Testers
- I E–C–
1.1 Request functionality
C I E–C–
1.2 Talk about functionality
C I —C–
1.3 Present requested functionality
Brainstorming Meeting
C I –MCV
1.4 Ask questions and identify sw pieces
C I –M—
1.5 Ask for estimation of US
C I EM—
1.6 Ask for priority of US
Iteration Prioritization Meeting
C I E–CV
1.7 Ask for details and satisfaction conditions
– I –M—
1.8 Fill out
US Checklists
C I E–CV
1.9 Ask questions, if needed
LEGEND
C I EMCV
Is it a Conversation? C: Yes, –: No
Requirement Activities: E: Elicitation, M: Modeling, C: Communication, V: Validation, – :Activity not present
US = User story

**Fig. 9.3** Pre-iteration flow of requirements at Easy Retirement

> We encourage everyone to do the stories as everyone has his or her view of how the system works or should work and makes them feel they are part of the product and feel involved. [This applies to] the plan administrator, the sales people, even the accountant [who] has actually written some stories for the report data that she needs. And this is one of things the people in this company like about it here is that they feel they are part of the things that are built and the process.

Sam records what he has learned from these interactions with the business people to create a personal user story checklist in a spreadsheet. Sam uses this checklist to help him think of questions to ask the businessperson regarding details and test cases, manage details of the user story, understand scope, manage risks, and document conversations and decisions. The checklist also helps him to participate more effectively in iteration planning. The user story checklist includes the satisfaction conditions that are needed to consider the user story completed, as well as a story description, its assigned story points, expected delivery date, the iteration it belongs to, output of the story, and related user stories .

Then, the software team held a brainstorming meeting (1.3–1.5 in Fig. 9.3) to make preliminary estimates of user stories, which are prioritised in a later meeting (1.6 in Fig. 9.3).

Jerry, the owner of Easy Retirement, explained that at first, some of their business partners did not understand their process but that the user story estimates help. He would use the prioritisation and effort estimates that were assigned to each request to negotiate with customers:

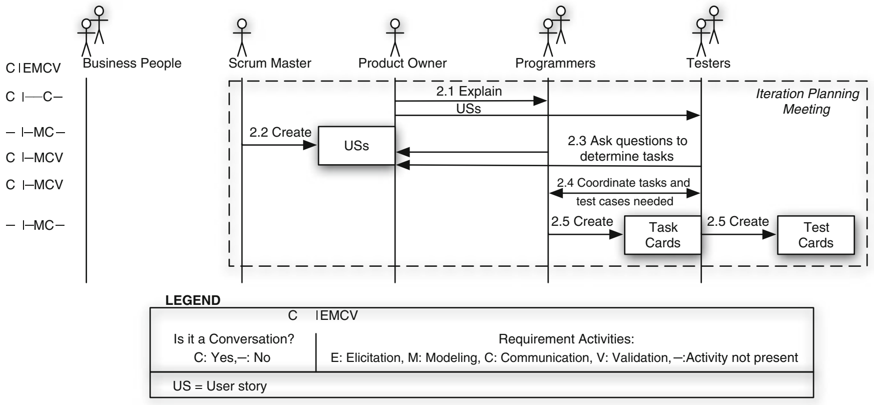

C I EMCV
Business People
Scrum Master
Product Owner
Programmers
Testers
C I—C–
– I–MC–
C I–MCV
C I–MCV
– I–MC–
Iteration Planning Meeting
2.1 Explain USs
2.2 Create
USs
2.3 Ask questions to determine tasks
2.4 Coordinate tasks and test cases needed
2.5 Create
Task Cards
2.5 Create
Test Cards
LEGEND
C IEMCV
Is it a Conversation? C: Yes, –: No
Requirement Activities: E: Elicitation, M: Modeling, C: Communication, V: Validation, –: Activity not present
US = User story

**Fig. 9.4** Iteration planning flow of requirements

> If we have an existing business requirement, you fully explain our process to anybody who said that they want something [new]. And I told them if you want it to be prioritised, here is what it means to me. So it is number five under 122 others, (laughs). You want it be in the top five? Tell me you will pay me $25,000; otherwise, just understand that there is no return on the investment. [We use] the user points to assign a value to it.

While this kind of negotiation with large customers or business partners may be rare, the same kind of negotiation can also take place internally as well on a smaller scale. Leanne explained that sometimes 'the business people are asking for the world' because they care more about new sales than anything else. They might not understand that a request is very big because, for example, it requires changes to the legacy code. Leanne said that she can go to the owner and explain that this is a 'very expensive' feature, relying on the estimates to convince him.

### 9.4.2 *Iteration Planning*

Every Friday, an iteration planning meeting (2.1–2.5 in Fig. 9.4) is held. At this time, user stories, tasks cards, and test cards are written on an index card and placed on the Scrum board. All the team members, including the Product Owner, Scrum Master, technical manager, programmers, and testers, work collaboratively to create these cards. The index cards do not have much information on them, just a couple of sentences. However, the requirements that they represent are validated when programmers and testers ask questions about details.

Observing a sprint planning meeting reveals just how much participation and collaboration takes place. The meetings we observed had strict turn-taking procedures, and holding the marker used to write the user story was an indication of who is speaking. While team members have many artefacts at their disposal for the meeting, such as privately used lists of stories or notes from conversations, what is included on a user story card is public and agreed upon by everyone. As people disagree, user story cards are frequently thrown away and replaced by a new version. The participants did comment that one of the advantages of the small card is that it is easily thrown away.

The user story is a form of performative knowledge. This can be seen in the way that people keep their personal artefacts separate from the user stories even when they contain similar information. The user story is the only standardised and accepted reference to a requirement. The user story card, while the outcome of the negotiation, still does not have all the information on it that was decided upon during the discussion. It is merely the token that refers to performative knowledge.

Several times, participants made note of how important it is that the user story card is written while everyone is present. The writing of the card in front of others is a part of the collaboration. Taking the card in front of everybody is also significant as an agreement that a particular programmer is taking ownership of that story. By writing the story together, there is 'group ownership of all the tasks'. And by taking the card in front of everybody, there is individual ownership of the task.

The user story also has a strong connection to participation because of the way that estimates of user stories create a valuing system for features of the system. The final decision of what is on the card is an agreement, not unlike a contract, of work for that sprint. The team members associated with a user story, the person who wrote it and the person who will implement it, are considered to be owners of the story.

### *9.4.3 Intra-iteration*

During the iteration, as showed in Fig. 9.5, testers write tests, programmers implement new functionality, and testers run the acceptance test cases. Additionally, the Product Owner checks that the software works as expected.

We asked Greg, a programmer, what he did to start a task and he replied:

> It depends. There are some things that are on the wiki. Some things that are new development so there is nothing really written for it yet. And other things there is past history on that feature that you are adding to. So I might talk to Carol, you know, she might point me to either the wiki or the Fitnesse that have background information on it so I can use that as a reference. I will say, well how did it work before? (laughs). And what is the delta between how it worked before and what I need to make it do now?

Testing plays a particularly important role during the intra-iteration stage. For the first time, requirements are expressed as test cases. Carol, a tester, explained, '[The testers] always meet with Sam on Monday and go over all the stories again to make sure that we understand everything and we can write a high-level test cases'. This information is stored on a wiki and coded into acceptance test cases in Fitnesse. Although Carol did not know if the programmers looked at the wiki, Ryan reported that he did:

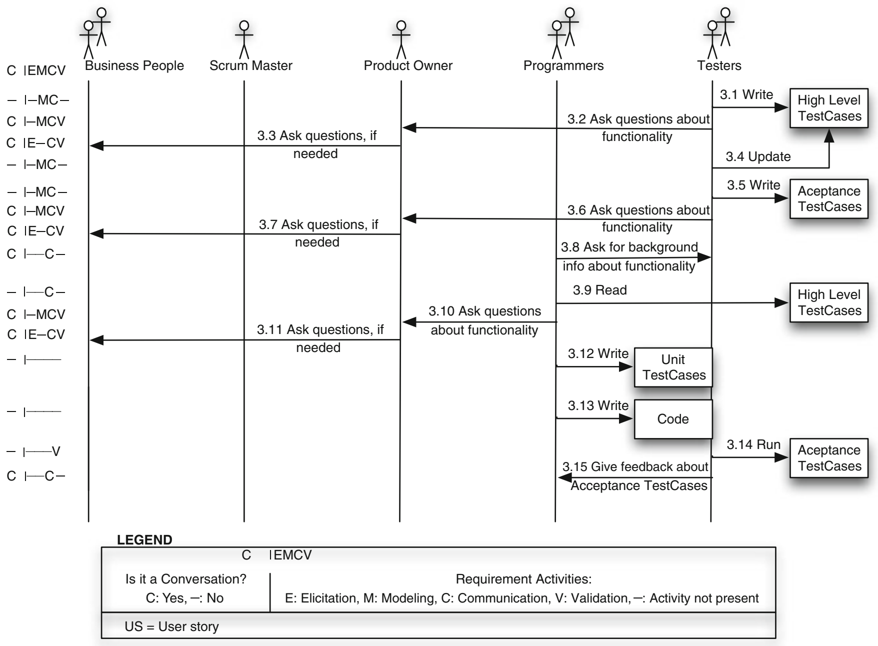

C IEMCV
Business People
Scrum Master
Product Owner
Programmers
Testers
– I–MC–
C I–MCV
C IE–CV
– I–MC–
– I–MC–
C I–MCV
C IE–CV
C I—C–
– I—C–
C I–MCV
C IE–CV
– I——
– I——
– I—–V
C I—C–
3.1 Write
High Level TestCases
3.2 Ask questions about functionality
3.3 Ask questions, if needed
3.4 Update
3.5 Write
Aceptance TestCases
3.6 Ask questions about functionality
3.7 Ask questions, if needed
3.8 Ask for background info about functionality
3.9 Read
High Level TestCases
3.10 Ask questions about functionality
3.11 Ask questions, if needed
3.12 Write
Unit TestCases
3.13 Write
Code
3.14 Run
Aceptance TestCases
3.15 Give feedback about Acceptance TestCases
LEGEND
C IEMCV
Is it a Conversation? C: Yes, –: No
Requirement Activities: E: Elicitation, M: Modeling, C: Communication, V: Validation, –: Activity not present
US = User story

**Fig. 9.5** Intra-iteration flow of requirements at Easy Retirement

> So that is pretty nice because, knowing what story I am working on, if there is any question, the first place to look is the wiki and see if it there. If there is nothing there, then we have to go ask somebody.

When the programmers add a feature to the software, they are also responsible for doing their own unit testing and for creating a 'fixture' for the Fitnesse testing framework. The testers add the fixture to the test harness and program a set of acceptance test cases.

## 9.5 Performative and Lexical Knowledge Sharing

Consistent with the values in the Manifesto for Agile Development [5], the requirements process at Easy Retirement does not use comprehensive documentation, instead relies on collaboration, and emphasises individuals and interactions. In order to better understand the forms and modes of knowledge sharing in agile requirements, we draw our attention to the categories of 'lexical' and 'performative'.

By lexical, we denote knowledge sharing through written documents. Because we do not wish to include informal jottings or notes, this label is derived from semiotics, where the lexicon is the set of things in a language or ontology [3]. In turn, lexical knowledge sharing happens in inscribed texts that are passed from one to another. Digital texts can also be lexical, provided that they are written documents, travel between stakeholders, and carry their weight of authority.

**Table 9.1** Performative and lexical sharing of requirements knowledge at Easy Retirement

| Stage | Performative knowledge sharing | Lexical knowledge sharing |
|---|---|---|
| Pre-iteration | Conversations between | Checklists |
| | Programmers and PO | |
| | Testers and PO | |
| | PO and business people | |
| Iteration planning | Conversations between | User story and task cards |
| | Programmers and PO | |
| | Testers and PO | |
| | Programmers and testers scrum board | |
| Intra-iteration | Conversations between | User story cards |
| | Programmers and PO | Wiki |
| | Testers and PO | Unit test cases |
| | PO and business people | Acceptance test cases |

By performative, we denote knowledge sharing that is acted out in the presence of others and can include question-asking and gestures, as well as informal speeches. This label follows from work in the humanities and social sciences, which sought to use performance as a heuristic for understanding human behaviour [10]. Goffman's work, which used the metaphor of a theatre (actors, stage, back-stage, audience, etc.) to analyse the presentation of the self in daily life, is highly influential in the area [6]. In the category of performative, we do not include all actions, but rather, we focus specifically on actions that are performed in order to be witnessed by an audience.

In many situations, knowledge sharing is neither entirely performative nor entirely lexical. However, one will figure more prominently than the other based on the context. We define these categories carefully to help us adjudicate the corner cases, such as the action of writing a requirements document and performances that include texts.

A written requirements specification is primarily lexical knowledge sharing. Stakeholders can sign off on the specification, and the document will serve as a set of agreements about the software to be delivered. It is a snapshot of requirements knowledge at a particular time, and as such, it can date quickly, lose accuracy, and decrease in value. Although authoring a requirements document is an action, it is not the performance of writing that completes the action of knowledge sharing but rather the reading of that writing by the receiver.

Moving an index card containing a user story from one column of the Scrum board to another is primarily performative knowledge sharing, not only because it is an action but also because it is an action that is intended to be seen by others. Although the index card contains text, it is not the words that cause knowledge to be shared but rather the interaction with or movement of the physical artefact.

In this section, we will look at the various sites for lexical and performative knowledge sharing in agile. The two types are summarised in Table 9.1.

## *9.5.1 Performative Knowledge Sharing*

Performative sharing of requirements knowledge occurred whenever a conversation took place. In the activity diagrams in the previous section, these are indicated by the letter 'C' in the first column of each row, which is the case for many of the interactions. There were two primary sites for performative knowledge sharing: conversations and the Scrum board.

### 9.5.1.1 Conversations

Conversations are a key site for performative knowledge sharing because they involve people in dialogue. This trait is not unique to software development. The decision to have a conversation, the selection of participants, and the questions that are asked are all part of the staging of the performance.

In agile, conversations are built into the process and the basic units of work. User stories are meant to include conversations between programmers, testers, and customers. Knowledge may reside in the minds of the participants, but it does not become part of the user story until it has been shared. The transmission takes place through a performance.

Performative knowledge takes place when knowledge in the minds of the participants is presented and others acquire an actionable understanding. Product Owner, Sam, described the user stories as 'triggers that help me to ask questions'. This question-asking is important, because it creates knowledge that did not previously exist. This performance is so important that Sam even writes out ideas for user stories before the planning meeting to remind him to ask questions.

Even though the performances are ephemeral, the knowledge persists beyond the moment when the action is taken. The agreement to a requirement is public and made when programmers are looking into the eyes of the customer. Developers cannot help but recall this performance when they are back at their desks, alone and working on a user story. For Greg, there is a big difference between

> sitting down with the requirements on your desk versus actually taking a piece of the requirement in front of everybody else. Now you have opened up the door to conversation about it, and now you have perspectives from multiple people and not just yourself and that piece of paper that you think that you are interpreting correctly.

As Jerry reiterated, the user stories would be 'meaningless without that interplay'. The meaning of the user story comes from the interactions between those present at the planning meetings when it is created as well as among the people whose roles are encoded in the user story.

### 9.5.1.2 Scrum Board

The Scrum board is an important example, because the performative knowledge sharing is non-verbal. Knowledge is shared by the arrangement of the index cards and other notes and actors moving, adding, or removing index cards. It contains

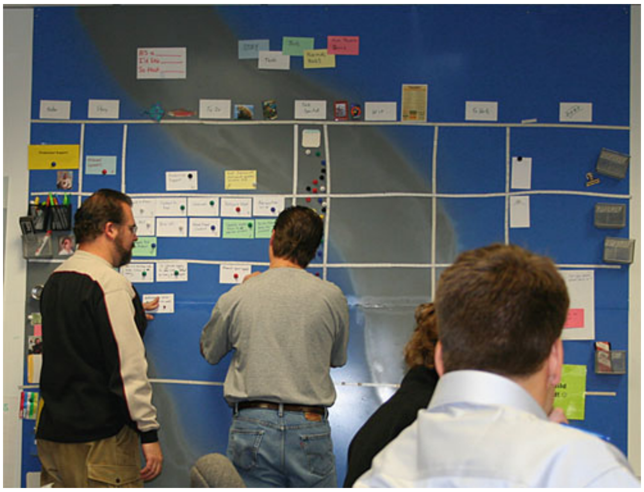


**Fig. 9.6** The Scrum board at Easy Retirement

*hand*written notes. Team members can see each other moving the cards. It is a tableau of knowledge creation, like a frame in a video, and it is a static representation of a dynamic process (Fig. 9.6).

After the cards are written during the sprint planning meeting, they are placed on the Scrum board to signify that they have officially become agreed upon units of work. During the sprint, developers move the cards from one column to another. Sometimes, a developer will even take a card to his desk while working on it. The board provides valuable information about how work is progressing. 'You can just look at the board and know how things are going', Leanne said. As the sprint progresses and cards are not being moved out of the 'to-do' column at a good pace, it gives the team a sense of how hard they will have to work to finish the sprint successfully. Similarly, a growing number of red cards indicating high severity bugs in need of attention create performative knowledge that things may be going awry.

In addition to being a representation of project status, the Scrum board can also help team members locate knowledge. The system under development is in flux, evolving, so the most current information is in the artefacts, which is most effectively accessed via the team members. As Greg explained:

> So you can look at the task and go, "well so-and-so is working out this task at the same time or just before." So you can have some context there where you might be able to use some of what he did.

### 9.5.2 Lexical Knowledge Sharing

In the activity diagrams in the previous section, lexical sharing of requirements knowledge occurred when the letter 'C' did not appear in the first column of each row. This happened only in a small number of interactions, as depicted by edge 1.1, 1.8, 2.2, 2.5, 3.1, 3.4, 3.5, 3.9, 3.12, 3.13, and 3.14. In this subsection, we discuss two sites for lexical knowledge sharing and make some comparisons to comprehensive requirements documents.

#### 9.5.2.1 Fitnesse Testing Wiki

The Fitnesse testing wiki is a lexical artefact that is built collaboratively between the programmers, testers, and the computer. The programmers provide fixtures for performing unit tests on their code. The tests use these fixtures to write system and acceptance tests. The results from running these tests are displayed on the Fitnesse testing wiki.

More than a static web page, the wiki is something that all the developers contribute to. Ryan, a programmer, explained:

> We mostly write JUnits, but we also provide [Fitnesse] fixtures for the testers, when we think that is necessary. I think that Fitnesse it is a nice testing tool because it [is] a test harness that nonprogrammers can maintain. You can add test cases to it, have your harness run them, without bothering other developers.

As can be seen from the quotation, Fitnesse 'runs' and keeps the 'nonprogrammers' from 'bothering other developers', so it is clearly an actor in Easy Retirement's software process.

The wiki is not merely a vessel for test cases and a display for results. Instead, there is a back and forth regarding whether the code is 'red' (failing) or 'green' (passing) and what new test cases are added. As Carol said enthusiastically, 'And then we will get that thing turning green, and we will start to write more and more test cases, and we always think of test cases'. Fitnesse also encourages dialogue between team members. Carol again said:

> They look at the Fitnesse and they will go "Wow, I did not understand how that was supposed to work." Or, you know, a lot of times we will sit down together, and I will say, "Here is how I am thinking on doing the Fitnesse test. Does this makes sense and line up with how you write code?" And again, as I said this, one advantage of the tool is that get us to collaborate, you know, all that communication is good.

#### 9.5.2.2 Software Release Documents

Due to the success and prominence of performative knowledge transfer at Easy Retirement, there is a paucity of inscribed knowledge or documents. This deficiency became a problem when the company was audited. Legal and regulatory compliance are important because they are providing financial service software. Leanne explained that an auditor had a difficult time dealing with their process, because they did not have documents with signatures and approvals:

> [They] were very upset that we did not have any signatures confirming what we were going to make in a sprint. And as they do not really understand agile, we told them in the next sprint, we are going to give you a release document showing that it was approved, tested on this release date, with the build number, etc.. . .so now we have documents called Software Release Documents.

This new document is an example of a lexical knowledge transfer created from undocumented knowledge in use at Easy Retirement. It was a kind of compromise to make their process look more like those that the auditor was familiar with.

#### 9.5.2.3 Past Experiences with Documents

Although the team did not rely on documents for knowledge sharing, they did have experience in the past with comprehensive requirements documents. Prior to switching to agile, Easy Retirement did rely significantly on lexical knowledge sharing. Most participants in the study did not recall these times fondly.

Jerry, Sam, and Ryan all mentioned the prior effort that they put into creating a comprehensive requirements document. Jerry said, 'I would literally build the entire application in PowerPoint and just say when you push this button, you are supposed to go to this web page and this is what you are supposed to end with'.

Despite the effort put into the documents, the process and end results were unsatisfactory. Jerry explained, 'It was taking too long [to deliver] and [the software team] would not get you what you asked for.... Whatever was written on the page and how they [the engineers] interpreted it was what was developed'. Here, Jerry alluded to the communications problem that occurs when using only lexical knowledge sharing. The 'page' is the primary mode of communication, and as is reflected in our conversations with many of our participants, the pre-agile process precluded any communication beyond this document.

Ryan offered the following critique:

> In the old days, we used to do waterfall, where we spent like 3 months just building documents. That was not fun. And nobody would get to try out a bit of code for about 3 months. And then by the time your document was done, your document was out of date, typically. It did not really serve its purpose because when you started doing development it was like, you kind of have this document that kind of sort of shows what you were shooting for, but because things had changed since you designed that, you sort of end with another approach anyways, where you build pieces, do it piece by piece.

## 9.6 Discussion

In this section, we discuss the implications of lexical and performative requirement knowledge sharing for both practitioners and researchers. Based on our field study, we have a number of insights and observations relevant to the application of agile and knowledge management.

### *9.6.1 Implications for Practitioners of Agile*

Lessons can be learned from the Easy Retirement experience, whether or not one intends to adopt agile validation or even agile itself. Some of the practices at the field site suggest improvements to knowledge sharing that are generally applicable.

*Encourage Knowledge Sharing and Question-Asking.* Team members should be encouraged to share knowledge and ask questions about requirements throughout the software lifecycle. Questions can be helpful even when using the waterfall model, especially in the early phases where detecting and fixing a misunderstood requirement could be less expensive and time-consuming. Asking questions about requirements will help team members gain a better understanding of high-level requirements and share a common understanding of requirements. This common understanding is key to have all team members participating in a collaborative validation where every member can trigger validation of requirements or validate them.

*Write Test Cases Even for the Obvious Success Scenarios.* Agile processes depend on continuous feedback and frequent conversations. Additionally, comprehensive documentation is avoided, sometimes at the risk of not having enough documentation. As a result, there is a great deal of tacit knowledge on agile projects, including requirements that seem obvious at the time. However, these requirements may not be obvious to team members as the project ages or to newcomers to the project. Consequently, tests should be written for the simple, obvious scenarios too. Test cases are actually part of user stories and provide high-level details.

### 9.6.2 Implications for Knowledge Management

Our findings are relevant to knowledge management practice and research as well.

*Pay Attention to Personal Knowledge Management.* Team members could be asked a question at any time, and when this happens, they need to have an answer. Consequently, each person needs to perform their own personal knowledge management to record information and to keep track of it, so it can be found again. At Easy Retirement, many people created documents for their own purposes that other people generally did not look at. Sam creates a checklist for gathering user stories and requirements. Leanne used a spreadsheet to track user stories completed in each iteration. Carol put details about test cases on a wiki. Each of these records is highly context specific – they are meant to be used by one person for a particular task for a constrained time period.

*The Boundary Between Performative and Lexical Requires Further Study.* This chapter draws an analytical line between performative and lexical knowledge sharing. However, there are many questions still to be answered. Our findings suggest that performative knowledge sharing is more effective for requirements, but there are many open questions. The information sharing that we witnessed was specific and tailored. Answers were given in response to a particular question posed by a team member. The participants had already established common ground, and each had a good sense of the other's perspective and starting point for the question. It is entirely possible that sensitivity to the audience and customisation of the answer were the most significant aspects of the knowledge sharing rather than performativity. Further study is required to tease out these differences, and here are some possible directions. On the web, there are question-and-answer web sites, such as Stack Overflow[1], Quora[2], and ask.metafilter.com. Users post questions and others provide answers and comments. Would this count as performative or lexical? In another vein, would we be able to achieve the same relevance, personalisation, and timeliness with written documentation? This is a topic of much research in knowledge management.

## 9.7 Limitations

We are aware that our study has some limitations. The data reported in this chapter is based on observations and interviews conducted in one company. Our findings cannot be generalised statistically to other companies due to the specific settings of Easy Retirement, the nature of the software product, and the organisational culture of the company, among other factors.

This chapter reports on data collected and analysed using qualitative methods. This work would be improved by future development of quantitative metrics to supplement the qualitative ones.

Although our study has some limitations, the results obtained represent the findings of an initial study and provide us with some empirical data on requirements knowledge in agile. We expect that our observations will be valuable for agile practitioners that use similar agile methods and our insights into lexical and performative knowledge will be valuable for researchers working in requirements.

## 9.8 Conclusion

In this chapter, we reported on a field study of the requirements practices of an agile software development team. Our analysis focused on requirements knowledge sharing because we were struck by the gap between conventional, academic concepts of requirements engineering and what we found the team was doing. In the software engineering literature, requirements knowledge is captured in specification documents, and requirements validation is an activity that relies on this detailed document. Such an artefact is rare on agile projects, which do not value comprehensive documentation. Instead, we saw a set of artefact activities that were centred around the basic unit of work in agile, the user story. A user story consists of a written reminder, test cases, and conversations, which means that it is both something that is created and something that is done. At Easy Retirement, the user story was also the basic unit for requirements. The conversations, test cases, and written reminders served to help team members share their knowledge and understanding of the requirements.

[1] http://www.stackoverflow.com

[2] http://www.quora.com

These observations led us to identify two modes of knowledge sharing: performative and lexical. Performative knowledge sharing is acted out for others and can include question-asking and gestures, as well as informal speeches. Lexical knowledge sharing happens in inscribed texts that are passed from one to another. Informal, ephemeral reminders are not considered lexical knowledge sharing, because they do not travel between stakeholders and do not carry their weight of authority. We found the software team relied almost entirely on performative knowledge sharing; there were no official documents for the entire team other than user story cards. Despite having few shared written documents, the team was able to effectively develop software that satisfied customer requirements.

Returning to Socrates, he was an advocate of performative knowledge sharing through dialogue but strongly disliked lexical knowledge sharing through written texts. He claimed that the dialectician did not impart knowledge by giving a prepared speech but rather he/she prepared the audience to receive knowledge before planting a specially selected seed of knowledge that would grow into wisdom. He had even less regard for written texts than for oration and said:

> . . .you give your disciples not truth, but only the semblance of truth; they will be hearers of many things and will have learned nothing; they will appear to be omniscient and will generally know nothing; they will be tiresome company, having the show of wisdom without the reality.

Although the rhetoric is strong, it is astonishingly similar to Ryan's criticism of comprehensive requirements documents, which came nearly 2,400 years later. In our modern age, it is unlikely that we would want to or even be able to rid ourselves of written documents entirely. But it is worthwhile to consider how we can bring more dialogue or performative knowledge sharing into requirements engineering.

**Acknowledgements** Thanks to all the participants and supporters at Easy Retirement. This research was possible due to their generosity in sharing their time and work with us. Thanks to Marisa Cohn for her help in data collection and to Anahita Fazl for helping us with transcriptions. This research was supported in part by a grant from the Agile Alliance Academic Research Program.

## References

1. Astels D (2003) Test-driven development: a practical guide. Prentice Hall PTR, London
2. Beck K (2000) Extreme programming explained: embrace change. Addison Wesley, Reading
3. Chandler D (2007) Semiotics: the basics. Taylor & Francis, New York
4. Cohn M (2004) User stories applied: for agile software development. Addison-Wesley Professional, Boston
5. Fowler M, Highsmith J (2001) The agile manifesto updation: Dr. Dobbs, 1 Aug 2001, http://www.drdobbs.com/the-agile-manifesto/184414755
6. Goffman E (2002) The presentation of self in everyday life. Doubleday, Garden City
7. Larman C, Basili VR (2003) Iterative and incremental developments: a brief history. IEEE Comput 36(6):47–56

8. Leuf B, Cunningham W (2001) The wiki way: quick collaboration on the web. Book, whole. Addison-Wesley Professional, Boston
9. Lofland J, Snow DA, Anderson L, Lofland LH (2006) Analyzing social settings: a guide to qualitative observation and analysis. Wadsworth/Thomson Learning, Belmont
10. Madison DS, Hamera J (2006) The Sage handbook of performance studies. Sage, Thousand Oaks
11. Mugridge R, Cunningham W (2005) Fit for developing software: framework for integrated tests. Prentice Hall, Upper Saddle River
12. Plato (2008) Phaedrus. Forgotten Books, Charleston
13. Schwaber K (2004) Agile project management with scrum. Microsoft Press, Redmond

# Chapter 10
# Using Web 2.0 for Stakeholder Analysis: StakeSource and Its Application in Ten Industrial Projects

**S.L. Lim, D. Damian, F. Ishikawa, and A. Finkelstein**

**Abstract** Software projects often fail because stakeholders are omitted. Existing stakeholder analysis methods rely on practitioners to manually identify and prioritise stakeholders, which is time consuming, especially in large projects with many stakeholders. This chapter investigates the use of Web 2.0 technologies, such as crowdsourcing and social networking, to identify and prioritise stakeholders. The investigation is based on the application of StakeSource in practice. StakeSource is a Web 2.0 tool that uses social networking and crowdsourcing techniques to identify and prioritise stakeholders. This chapter describes our experiences of and lessons learnt from applying StakeSource in ten real-world projects from six organisations in UK, Japan, Australia, and Canada, involving more than 600 stakeholders. We find that StakeSource can yield significant benefits, but its effectiveness depends on the stakeholders' incentives to share information. In some projects, StakeSource elicited valuable stakeholder information; in other projects, the stakeholder responses were insufficient to add value. We conclude with a description of factors that influence stakeholder engagement via the use of Web 2.0 tools such as StakeSource. If collaborative tools such as StakeSource were to find a place in requirements engineering, we would need to understand what motivates stakeholders to contribute.

S.L. Lim (✉) • A. Finkelstein
University College London, London, United Kingdom
e-mail: s.lim@cs.ucl.ac.uk; a.finkelstein@cs.ucl.ac.uk

D. Damian
University of Victoria, Victoria, BC, Canada
e-mail: danielad@cs.uvic.ca

F. Ishikawa
National Institute of Informatics, Tokyo, Japan
e-mail: f-ishikawa@nii.ac.jp

## 10.1 Introduction

Stakeholder analysis, which involves the identification and prioritisation of stakeholders, is a critical step in requirements elicitation. Stakeholders are individuals or groups who can influence or be influenced by the software project [1]. These people include customers who pay for the software system, users who interact with the system to get their work done, developers who build and maintain the system, and legislators who impose rules on the development and operation of the system [1, 2]. These people have diverse backgrounds, expertise, interests, and personal goals [3]. Projects with higher stakeholder engagement tend to have higher success [4–6], but omitting stakeholders is a common problem in software development [7]. As stakeholders are the source of requirements, they have to be identified before requirements can be elicited [1, 8]. As a result, missing stakeholders gives rise to missing requirements, causing projects to fail [9, 10].

This work investigates the application of Web 2.0 technologies for stakeholder analysis using StakeSource, a Web 2.0 stakeholder analysis tool developed in previous work [11]. StakeSource uses several Web 2.0 technologies including crowdsourcing, social networking, and tagging and aims at engaging a large set of stakeholders [11]. StakeSource elicits information about other stakeholders from the stakeholders without requiring the practitioner[1] to be present [11]. Then, it builds a social network of stakeholders and prioritises the stakeholders using the elicited information. The stakeholders can provide information anytime and anywhere, and the information they provide can reduce missing requirements and improve the quality of the elicited requirements [9, 10]. As such, StakeSource has the potential to effectively manage stakeholder information in projects with a large number of stakeholders, even when the stakeholders are in different locations [12–14]. In addition, it is one of the first Web 2.0 requirements elicitation tool to be widely available to practitioners, providing valuable data for our study. It is anticipated that Web 2.0 technologies will be increasingly used in requirements elicitation, and thus empirical studies of such tools in real projects are needed to assess their viability [15].

In this chapter we describe the application of StakeSource in ten real-world projects. These projects are based in six organisations in United Kingdom, Japan, Australia, and Canada, involving more than 600 stakeholders. The number of stakeholders per project ranges from 10 to more than 200. The effectiveness of StakeSource in engaging with stakeholders is investigated in terms of response rate, timing of response, quantity, and quality of the response. The practitioners and stakeholders are interviewed, and data is analysed to reveal the factors that influence stakeholder engagement using StakeSource.

[1] In this chapter, practitioners refer to the requirements engineers, project managers, system analysts, business analysts or developers who are responsible for stakeholder analysis in their projects.

The evaluation of StakeSource in real projects is a significant contribution to software engineering research. Web 2.0 applications are difficult to evaluate due to their collaborative nature [16]. Despite the widespread use of Web 2.0 in software development, there are few empirical studies to investigate the adoption and implications of their use [15]. Our lessons learnt and experiences can benefit practitioners and researchers by highlighting the benefits and limitations associated with using Web 2.0 technologies to support software engineering activities [15]. For example, in StakeSource, the use of Web 2.0 technologies enables stakeholders to provide information without the presence of the practitioner. Nevertheless, the participation of many stakeholders, such as users and legislators, is largely voluntary. Stakeholders with low incentives may not respond, fail to provide a timely response, or provide a low quality response [17].

The rest of this chapter is organised as follows. Section 10.2 describes existing stakeholder analysis methods and tools. Section 10.3 describes StakeSource and Sect. 10.4 introduces the projects and our methodology. Section 10.5 describes our experiences and lessons learnt, Sect. 10.6 discusses threats to validity and Sect. 10.7 concludes.

## 10.2 Background

Existing stakeholder analysis methods rely on the practitioners to manually identify stakeholders. For example, in the *semi-structured approaches* that form the basis of existing practices, the practitioner manually identifies stakeholders by considering broad stakeholder categories, such as stakeholders who interact directly with the system and stakeholders who have interests in the project [7]. In the *interview method* proposed by Pouloudi and Whitley [18], the practitioner manually identifies generic stakeholder roles and stakeholders, then interviews each stakeholder to learn about other stakeholders or stakeholder roles, and repeats the interviews for each newly identified stakeholders. In the *search method* proposed by Sharp et al. [2], the practitioner manually identifies initial stakeholders from project documentation or interviews. Then for each stakeholder, the practitioner identifies other stakeholders who interact with the stakeholder and repeats this process for each newly identified stakeholder.

Traditional stakeholder analysis tools hold and process the data provided by the practitioners but provide little support in the actual identification and prioritisation of stakeholders. The practitioners manually elicit information from the stakeholders via face-to-face meetings, workshops, or focus groups and then populate the information in the tools [1, 7, 19]. The main purpose of the tools is to hold information. Except for the project team and possibly the key clients, few stakeholders interact directly with these tools. For example, in Stakeholder Analysis Matrix,[2] the practitioner manually compiles a list of stakeholders and plots them

[2] http://www.mindtools.com/pages/article/newPPM_07.htm.

against two variables on a matrix, such as power and interest or importance and influence. The Onion Model developed by Alexander [8] and the Volere Stakeholder Analysis Template developed by Alexander and Robertson [20] consist of a set of generic stakeholder roles. The practitioner refers to the generic roles to manually derive specific roles for the project. Stakeholder Circle[3] is a software package that enables practitioner to enter the stakeholders' information after they have been manually identified, and the tool generates reports based on the information provided.

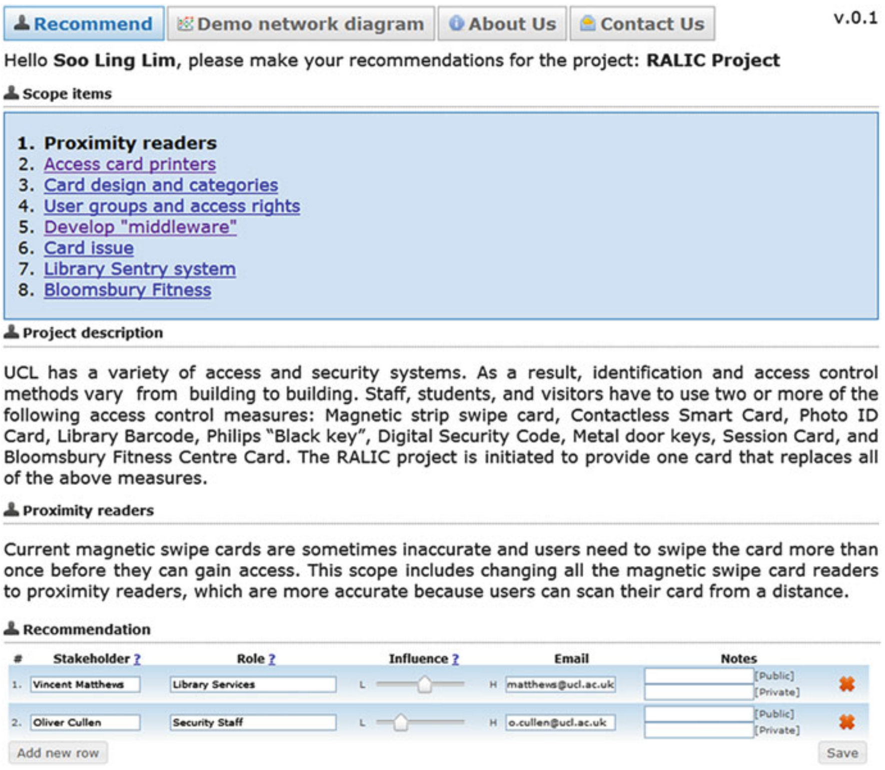
Software projects are becoming more global and involving more stakeholders. As a result, Web 2.0 tools are increasingly used to augment existing development tools, with the aim to support collaboration and increase awareness among stakeholders. Using Web 2.0 tools, emerging forms of software development, such as distributed development, can benefit from access to a large pool of stakeholders [15]. In our previous work, we have developed StakeNet, a method that uses social networks for stakeholder analysis in large projects [9, 10]. In StakeNet, the practitioner prepares an initial list of stakeholders. Then, the practitioner manually identifies stakeholders by asking the initial stakeholders to recommend other stakeholders, builds a social network of stakeholders from the recommendations, and prioritises the stakeholders using social network measures. StakeNet was applied in a substantial real-world project and shown to identify a comprehensive set of stakeholders and prioritise them accurately [9, 10]. Nevertheless, the method is time consuming. The practitioner has to approach each stakeholder to elicit recommendations, convert the recommendations into the appropriate format for the social network measures, compute the stakeholders' priorities using social network measures, and convert the output from the social network measures into a prioritised list of stakeholders [9]. Changes to the recommendations (additions, modifications, removal) require the practitioner to repeat the process [9]. In the previous application of StakeNet [10], more than 150 person hours was spent manually eliciting and processing the recommendations from 68 stakeholders.

## 10.3 StakeSource

StakeSource is a Web 2.0 tool developed to automate the StakeNet method [9, 11]. To use StakeSource, the practitioner provides StakeSource with the initial stakeholders. StakeSource automatically contacts the stakeholders and asks them to recommend other stakeholders via a web interface. Then, StakeSource converts the recommendations into the appropriate format, applies the social network measures, visualises the network of stakeholders, and produces a prioritised list of stakeholders. The remainder of this section describes the features of StakeSource.[4]

[3] http://www.stakeholder-management.com/.

[4] StakeSource tool demo is available at http://vimeo.com/18250588. For further details about StakeSource, refer to the previous work [9, 11].



Hello **Soo Ling Lim**, please make your recommendations for the project: **RALIC Project**

**Scope items**

1. **Proximity readers**
2. Access card printers
3. Card design and categories
4. User groups and access rights
5. Develop "middleware"
6. Card issue
7. Library Sentry system
8. Bloomsbury Fitness

**Project description**

UCL has a variety of access and security systems. As a result, identification and access control methods vary from building to building. Staff, students, and visitors have to use two or more of the following access control measures: Magnetic strip swipe card, Contactless Smart Card, Photo ID Card, Library Barcode, Philips "Black key", Digital Security Code, Metal door keys, Session Card, and Bloomsbury Fitness Centre Card. The RALIC project is initiated to provide one card that replaces all of the above measures.

**Proximity readers**

Current magnetic swipe cards are sometimes inaccurate and users need to swipe the card more than once before they can gain access. This scope includes changing all the magnetic swipe card readers to proximity readers, which are more accurate because users can scan their card from a distance.

**Recommendation**

| # | Stakeholder ? | Role ? | Influence ? | Email | Notes |
|---|---|---|---|---|---|
| 1. | Vincent Matthews | Library Services | L H | matthews@ucl.ac.uk | [Public] [Private] |
| 2. | Oliver Cullen | Security Staff | L H | o.cullen@ucl.ac.uk | [Public] [Private] |

**Fig. 10.1** StakeSource web-based recommendation form

StakeSource is a web-based application. The practitioners access StakeSource via a web interface and create their project by entering project details such as name, description, and scope definition. They also provide an initial set of stakeholders from the categories of users, developers, legislators, and decision-makers. Each stakeholder in this initial list has the following information: name, the role in the project, and email address. The practitioners can also customise the email template that StakeSource uses to contact stakeholders.

StakeSource contacts the initial stakeholders via email. The email provides a link that brings the stakeholders to the web-based form that enables them to recommend other stakeholders (Fig. 10.1). The recommendation form consists of the project name and scope description as provided by the practitioners. Each recommendation consists of the stakeholder's name, their role in the project, their influence in the project (from low to high), and their email address. If a stakeholder is aware of a role but is not aware of the individual stakeholders, he can recommend only the role. Stakeholders can also comment on the stakeholders they recommend. Public comments can be viewed by anyone who can access the stakeholder analysis user interface; private comments are only available to the practitioners.

Each time a new stakeholder is identified, StakeSource contacts the stakeholder to invite them to recommend other stakeholders. This technique is also known as the snowballing technique [21], where the set of stakeholders build up like a snowball rolled down a hill. People who are recommended may be "non-stakeholders" or stakeholders who lack time or interest to be involved in the project. StakeSource provides an option for these people to unsubscribe from the project and nominate other stakeholders.

Once the recommendations are collected from the stakeholders, StakeSource provides the following support for stakeholder analysis via a web interface. This interface is accessible to the practitioners as well as stakeholders who have made recommendations.

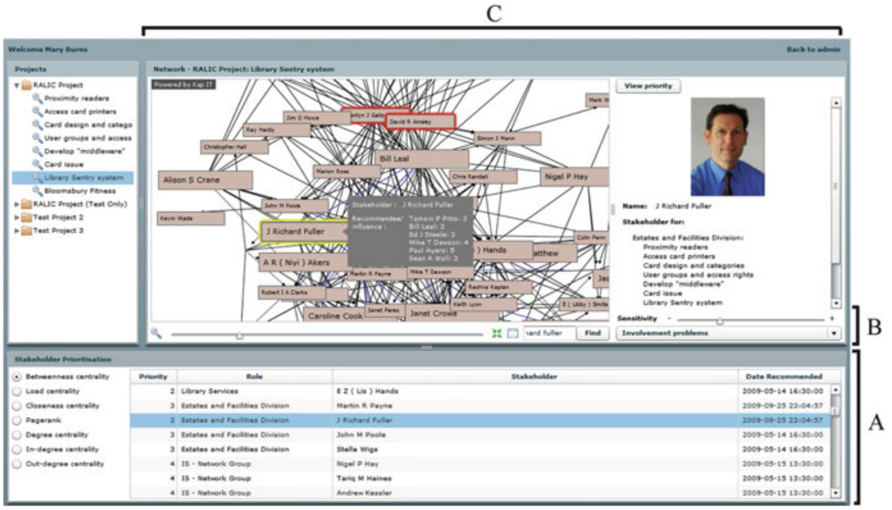
*Feature 1: Identify and prioritise stakeholders*. StakeSource compiles the initial and recommended stakeholders to form the list of stakeholders in the project. To prioritise stakeholders, StakeSource builds a social network of stakeholders with the stakeholders as nodes and their recommendations as directed links: *S1* links to *S2* if *S1* believes *S2* to be a stakeholder. Then, it prioritises the stakeholders using various social network measures. For example, in-degree centrality prioritises stakeholders who receive the most recommendations, out-degree centrality prioritises stakeholders who make the most recommendations, and betweenness centrality prioritises stakeholders who are widely recommended by disparate groups of stakeholders [11]. Each time a measure is selected, StakeSource applies the measure and displays the prioritised list of stakeholders and their roles in the stakeholder analysis user interface (Fig. 10.2 Panel A). To improve the accuracy of prioritisation, stakeholders are unaware of existing recommendations when they recommend other stakeholders.

*Feature 2: Identify stakeholders with potential problems*. StakeSource identifies potential involvement or communication problems a stakeholder may have based on the stakeholder's position on the social network (Fig 10.2 Panel B) [11]. When one of the problems is selected, StakeSource highlights stakeholders in the network who may potentially have the problem during the project. StakeSource provides a slider to change the sensitivity of problem detection. This helps the practitioners to decide the right level of problem detection for the project, which is a trade-off between the risk of the problem affecting the project and the cost to rectify the problem [11].

*Feature 3: Display stakeholder social network and details*. The stakeholders' recommendations are visualised as a social network (Fig 10.2 Panel C). StakeSource enables practitioners and stakeholders to study a stakeholder's position in the social network, the stakeholder's details, priority, and the stakeholders they recommend. For each stakeholder, StakeSource displays their name, role, photo, the scope items they are recommended for, the stakeholders who recommended them, the stakeholders they recommended, and comments from other stakeholders [11].

## 10.4 Using StakeSource in Practice

Industrial practitioners were made aware of StakeSource through the demonstration of the tool at seminars and conferences. A StakeSource website (www.stakesource.co.uk) was set up. Practitioners who were interested to use the tool could request an account



Projects:
- RALIC Project
  - Proximity readers
  - Access card printers
  - Card design and catego
  - User groups and access
  - Develop "middleware"
  - Card issue
  - Library Sentry system
  - Bloomsbury Fitness
- RALIC Project (Test Only)
- Test Project 2
- Test Project 3

Name: J Richard Fuller

Stakeholder for:

Estates and Facilities Division:
- Proximity readers
- Access card printers
- Card design and categories
- User groups and access rights
- Develop "middleware"
- Card issue
- Library Sentry system

(●) Betweenness centrality
( ) Load centrality
( ) Closeness centrality
( ) Pagerank
( ) Degree centrality
( ) In-degree centrality
( ) Out-degree centrality

| Priority | Role | Stakeholder | Date Recommended |
|---|---|---|---|
| 2 | Library Services | E Z ( Lis ) Hands | 2009-05-14 16:30:00 |
| 3 | Estates and Facilities Division | Martin R Payne | 2009-09-25 23:04:57 |
| 3 | Estates and Facilities Division | J Richard Fuller | 2009-09-25 23:04:57 |
| 3 | Estates and Facilities Division | John M Poole | 2009-05-14 16:30:00 |
| 3 | Estates and Facilities Division | Stella Wigs | 2009-05-14 16:30:00 |
| 4 | IS - Network Group | Nigel P Hay | 2009-05-15 13:30:00 |
| 4 | IS - Network Group | Tariq M Haines | 2009-05-15 13:30:00 |
| 4 | IS - Network Group | Andrew Kessler | 2009-05-15 13:30:00 |

**Fig. 10.2** The three panels (*A*, *B*, and *C*) of the StakeSource web-based user interface (refer to [11] for enlarged figure)

from the website or contact the authors. The majority of the practitioners who adopted StakeSource did so after the tool was demonstrated to them in seminars. Others adopted StakeSource through word of mouth. For example, two projects used StakeSource after being recommended by higher-level management, and another two projects used it after their colleagues (practitioners or stakeholders) who have used the tool in other projects recommended the tool to them.

## 10.4.1 Projects

Table 10.1 summarises the projects included in the study. In addition to the ten real-world projects, a student project was included to investigate the differences between industrial and student projects. These projects have completed their use of StakeSource in the 1-year period after the tool was developed (December 2009 to December 2010). In these projects, the project details were entered, initial stakeholders were provided, and invitation emails were sent to stakeholders. Small projects have about 10 stakeholders; larger projects have more than 70 stakeholder groups and 200 stakeholders. For reasons of privacy, the projects, organisations, practitioners, and stakeholders are anonymised in this chapter.

The project characteristics are summarised as follows.

- Projects P1 to P10 were industrial projects led by practitioners. P11 was a Masters project led by students.

**Table 10.1** Projects (*P* for project, *O* for organisation, *D* for department)

| ID | Short description | Application area | Identifying | Organisation | Country | Size |
|---|---|---|---|---|---|---|
| P1 | Develop an enterprise software system | Software | Stakeholders | O1.D1 | UK | L |
| P2 | Identify experts in health problems in urban environments | Non-software | Experts | O1.D2 | UK | S |
| P3 | Same project as P2 with random initial stakeholders | Non-software | Experts | O1.D2 | UK | S |
| P4 | Examine and improve organisational structures, processes, people practices, culture, and values and change management | Non-software | Stakeholders | O1.D1 | UK | L |
| P5 | Investigate the adequacy of an existing role in the organisation and create a new role if necessary | Non-software | Stakeholders | O2 | UK | M |
| P6 | Identify people who hold information that can help new employees become productive members of the organisation | Software | Experts | O3 | UK | S |
| P7 | Develop an enterprise software system | Software | Stakeholders | O1.D1 | UK | S |
| P8 | Develop a cloud computing facility | Software | Stakeholders | O4 | Japan | M |
| P9 | Identify potential investors and users of an innovative software application | Software | Stakeholders | O5 | Australia | S |
| P10 | Increase awareness about an existing access grid system | Software | Stakeholders | O6 | Canada | M |
| P11 | Develop a Web 2.0 system | Software | Stakeholders | Student | UK | S |

- All projects were conducted in English except for P8. In P8, the project description and invitation email were worded in Japanese in StakeSource, and stakeholders were able to provide recommendations in either English or Japanese.
- All the projects are software projects except the following. P2, P3, and P4 were policy-related projects. P5 was a project to investigate the adequacy of an existing role in the organisation. P8 is a software project but a major part of the project included the design and configuration of hardware devices.
- Most projects used StakeSource to find stakeholders, but P2, P3, and P6 used it to find experts. In these projects, the recommendation form comes with an extra field for the experts to enter their own expertise as tags. These projects were included in our study to investigate if stakeholders are motivated to provide recommendations when it conflicts with their own benefits. In P5 and P8,

stakeholders were already identified, StakeSource was used to validate the list of stakeholders and uncover missing stakeholders.

- In all projects, the stakeholders were unaware that their responses were being studied, in order to study their natural response or lack of response. One exception was P8, where the email to the stakeholders stated that the tool was being studied as part of a research project, and the data provided by the stakeholders will be analysed empirically for research purposes and used to improve the software system.
- All projects were set up by the practitioners themselves, except for P11 and P8. P11 was a student project set up by the students; P8 was set up by the second author on behalf of the project manager. In all projects, the first author provided technical support.
- P3 was the same project as P2, but instead of the initial set of stakeholders determined by the practitioner, P3 used the same number of initial stakeholders but randomly selected from the organisation people directory. This project was created to investigate if "non-stakeholders" were equally motivated to engage in the project by recommending stakeholders.
- All the projects were managed by different practitioners, except P2 and P3 (see previous point). P1 and P4 had different project managers but was set up by the same practitioner whose role in both projects was the communications manager.

## 10.4.2 Methodology

We interviewed the practitioners and stakeholders and analysed the data captured by StakeSource for each project. A total of eight practitioners from the real-world projects were interviewed. The practitioner for P6 was unavailable for the interview; P2 and P3 shared the same practitioner. The following questions guided the report of our experiences and lessons learnt:

Q1. Were stakeholders motivated to respond and how timely were their responses?
Q2. What were the stakeholders' responses?
Q3. How useful were the responses to the project?
Q4. What were the factors that influence the stakeholders' responses?

To analyse the stakeholder data elicited by StakeSource, the StakeSource database was accessed and the following information was extracted for each project.

- Initial stakeholders
- Customised email content
- Reminder email content (if available)
- Stakeholders who responded
- Stakeholders who provided recommendations
- Stakeholders who unsubscribed and rationale behind it (if available)

- Recommendations and date of recommendations
- Public and private notes
- Expertise description (if available)

The practitioners responsible for setting up their projects in StakeSource were invited for a face-to-face interview. The interviews were semi-structured, allowing the questions to be modified and new questions to be brought up depending on their response [22]. Some of the questions include:

- What is your previous experience in using stakeholder analysis tools?
- How were the initial stakeholders identified?
- How useful are the stakeholders identified by StakeSource?
  - How do you use StakeSource's output?
  - What is the stakeholders' contribution to the project?
  - Who should not be on the stakeholder list?
  - Who are unexpected stakeholders?
- How useful are the descriptions about stakeholders?
- How do you use the tool? Network view? List view?
- What is the progress of the project after the tool is used? Did you contact the stakeholders identified by the tool?
- What do you think about the response rate? Lower than expected, ok, more than expected? Why?
- Which part of the tool is most useful? Which is the least useful? Do you have any suggestions for improving the tool?

For P1, the first author also attended the board meeting where the practitioners reported their use of StakeSource to their directors. In addition, the practitioner for P2 and P3 allowed the stakeholders to be interviewed. In those projects, phone interviews were conducted with stakeholders who did not respond. The questions include:

- Did you receive an email about the project?
- If so, why didn't you respond? Is it because the tool is difficult to use?

## 10.5 Experiences with StakeSource and Lessons Learnt

### *10.5.1 Timeliness and Motivation to Respond*

In projects where stakeholders provided recommendations, StakeSource was able to build the social network and produce a prioritised list of stakeholders. However, in some projects, there were little incentives for stakeholders to recommend other stakeholders. In these projects, StakeSource failed to elicit information from the stakeholders.

More than 600 stakeholders were identified, but only about 150 responded, giving an overall response rate of 25 %. The response rate for each project was calculated as the number of stakeholders who responded over the total number of stakeholders identified. According to Table 10.2, the response rate for all the projects in this study ranged from 0 % to 39 %, which was similar to the online survey response rate by Deutskens et al.[5] [23]. The response rate was consistent with the results from survey research: face-to-face and phone interviews have about 40 percentage points higher response rate than online surveys [24]. This indicates a trade-off between manual and automated approaches for eliciting information from stakeholders. Manual approaches may be more time consuming but are likely to elicit more comprehensive information as compared to web-based approaches.

In this study, only two projects had higher response rate than that of online surveys, i.e., P11 (39 %) and P8 (35 %). P11 was a student project, and the high response rate may be due to the motivation to do well in their requirements engineering course. In P8, the stakeholders were aware they were being studied, which may have led to a higher response rate. In addition, many of the stakeholders in P8 are researchers or research-oriented students, so they may have been interested to participate in experiments.

Four projects have response rates of less than 10 %. Among these projects, P4 and P6 had no response. Nevertheless, P6 started with only 2–3 initial stakeholders, which is the main reason for no responses, as 2–3 stakeholders are too few for an effective snowballing process. The practitioner for P4 reported that the project manager decided against using StakeSource as the use of new technologies might be risky for the project. Nevertheless, the same practitioner continued to use StakeSource for P1 and recommended StakeSource to practitioner of P7. P3 and P10 had a very low response rate of 3 %. P3 was the project with a random set of initial stakeholders. Interviews with the stakeholders revealed that they ignored or deleted the invitation emails, as it was not relevant to them. The purpose of P10 was to ask existing users of an access grid system to recommend other users of an access grid system. It may be easier for the users to directly ask their collaborators to adopt the access grid system, rather than to recommend them using StakeSource.

In some projects, the stakeholders were already communicating before StakeSource was used. For example, in P9, three out of nine initial stakeholders provided recommendations via email or face-to-face communication before StakeSource was used. As such, StakeSource only managed to engage two other initial stakeholders. Most of the stakeholders who responded did so within the first week receiving the email. For P8, all the stakeholders who responded did so in 2 days, and it also has a very high response rate. Interviews revealed that in P8, employees tend to respond to email within a short timeframe. 10 projects were completed within 2 weeks. Only 1 project was completed in 20 days. The delay was

[5] In Deutskens et al.'s study of the response rate of online surveys with different configurations (e.g., short vs. long, donation to charity vs. lottery incentive, early vs. late reminder), they found that the response rate ranged from 9.4 % to 31.4 %.

**Table 10.2** Summary of data collected by StakeSource and in interviews

| | P1 | P2 | P3 | P4[a] | P5 | P6[b] | P7 | P8 | P9 | P10 | P11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| No. of initial stakeholders | 64 | 30 | 30 | 3 | 13 | 2 | 32 | 12 | 9 | 30 | 6 |
| Total no. of stakeholders identified[c] | 193 | 39 | 31 | 3 | 35 | 2 | 81 | 24 | 11 | 31 | 14 |
| No. of stakeholders who responded | 37 | 25 | 6 | 0 | 10 | 0 | 20 | 8 | 2 | 1 | 7 |
| No. of stakeholders who made recommendations | 32 | 10 | 1 | 0 | 10 | 0 | 20 | 8 | 2 | 1 | 7 |
| No. of experts who provided expertise | N/A | 22 | 4 | N/A | N/A | 0 | N/A | N/A | N/A | N/A | N/A |
| No. of people unsubscribed | 7 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Avg. no. of recommendations per stakeholder who recommended | 5.34 | 1.30 | 1.00 | 0.00 | 3.80 | 0.00 | 6.10 | 7.13 | 1.00 | 1 | 4.71 |
| No. of emails bounced | 0 | 0 | 0 | 0 | 1 | 0 | 6 | 1 | 0 | 0 | 0 |
| Response rate (%) | 19 | 26 | 3 | 0 | 29 | 0 | 27 | 35 | 18 | 3 | 39 |
| Factor increase in stakeholders (total – initial)/total | 2.02 | 0.30 | 0.03 | 0.00 | 1.69 | 0.00 | 1.53 | 1.00 | 0.22 | 0.03 | 1.33 |
| No. of snowballing rounds (round 1 is initial seed) | 4 | 3 | 2 | 1 | 3 | 1 | 4 | 3 | 2 | 1 | 3 |
| Responses completed in (days) | 15 | 15 | 15 | N/A | 10 | N/A | 20 | 2 | 4 | 7 | 4 |
| Did StakeSource identify stakeholders or stakeholder roles that practitioners were unaware of? | Yes | Yes | No | N/A | Yes | No | Yes | Yes | Yes | Yes | Yes |
| Was StakeSource useful to the project? | Yes | Not very | No | N/A | Yes | N/A | Yes | Yes | Yes | Not very | Yes |
| Mentioned using StakeSource in future project? | Yes | No | No | Yes | Yes | N/A | Yes | Yes | Yes | Yes | Yes |
| Recommended StakeSource to other practitioners? | Yes | No | No | Yes | Yes | N/A | No | No | Yes | Yes | No |

[a]The project manager decided to stop using StakeSource before the snowballing process started
[b]The practitioner was not interviewed due to unavailability
[c]A stakeholder consists of a name and a role; and if the name is not provided, only the role

caused by technical issues: the StakeSource server was down and stakeholders were unable to make recommendations. The recommendations restarted when the issue was fixed and an email reminder was sent.

## *10.5.2 Types of Stakeholder Response*

The responses from stakeholders consisted of recommendations about other stakeholders with optional comments about the stakeholders and unsubscription with optional rationale for unsubscription. Through the use of StakeSource technologies, the stakeholders were able to voice their opinions early in the project. As a result, the projects uncovered missing stakeholders, negative stakeholders (stakeholders who are unfriendly toward the product being developed), and stakeholders who lack time or interest to be involved.

Among the stakeholders who responded, 92 % of them provided recommendations, and the average number of recommendations ranges between 1 and 7 (Table 10.2). In the majority of the projects, the stakeholders were able to identify other stakeholders that the practitioners were unaware of. The number of new stakeholders identified also depends on the initial list. For P2, only a few new stakeholders were identified. However, this was because the project manager had used an open call to find the initial stakeholders by sending emails to mailing lists.

Most recommendations were valid. Errors were caused by misunderstanding, rather than by malicious intent. For example, a stakeholder entered the stakeholder's surname rather than their role in the Role field for all her recommendations. A few emails bounced (Table 10.2) due to the stakeholders entering incorrect email addresses. Some stakeholders wrote in the public notes field the phrase "visible to everyone," as they misunderstood that doing so would make their recommendations public.

Only 8 % of the responses were unsubscription. Unsubscription responses were informative, especially when stakeholders provided the reason for unsubscription. These responses uncovered negative stakeholders and stakeholders who lacked time to be involved in the project. For example, the objective of P1 was to replace the existing paper-based system with a software system. A stakeholder unsubscribed to the project with the reason "I find that the 'paper' system as it is, works extremely well. I think that online applications will be more time- consuming than the present system. I feel that I want to avoid anything which suggests sitting at a computer for even longer than I do already." Another stakeholder unsubscribed and provided the reason "Sorry but I don't think I can spare the time (to attend project meetings)." Uncovering negative and unavailable stakeholders early in the project can help the practitioners mitigate the risks because they can monitor negative stakeholders and find other available stakeholders.

Some unsubscriptions revealed "non-stakeholders" or stakeholders who have changed their roles. These people nominated other more suitable stakeholders in their unsubscription. For example, a non-stakeholder wrote "I am no longer the

[role] of [department name]. Please could you send your invitation to [name] [email] and [name] [email].” Another non-stakeholder wrote, “I don’t have any involvement with the xxx process. This is dealt with by [name].”

Stakeholders’ comments about other stakeholders provided the practitioners with the rationale of their recommendation and additional information about the stakeholders. For example, the following comments described the stakeholder’s expertise, “[Name] has been contributing to [another project] on children, disabilities and well-being in informal settlements in India,” and “[Name]’s influence is very strong in assisting with the more complex proposals.” Another comment revealed suitable representatives of a stakeholder group, “As the central contact point for [group], [name] could be the point of information.” Private comments were informative but generally less positive. For example, “In our case this role is not very effective. . .However, if the role were more pro-active. . .it might form the basis of a more comprehensive liaison service. . .”

### *10.5.3 Value of Responses to Project*

The study of StakeSource in real-world projects reveals benefits, limitations, and risks as follows.

*Benefits*. In general, in projects with more than 10 % response rate, the practitioners found the use of StakeSource to benefit their project. An indicative factor of success is continued use of the list of stakeholders identified by StakeSource. For example, P1 and P5 used the stakeholder list as a “contact list” to organise future workshops and meetings with the stakeholders. Additional features to print the network diagram and export stakeholder list were requested by three projects for visualisation and reporting purposes.

The practitioners found the network diagram and the stakeholder list to be useful. According to the practitioners for P1, “StakeSource identified some unexpected stakeholders, and the stakeholder network highlighted the need for communication among clusters of stakeholders for the project to be successful.” According to the practitioner for P2, “The experts identified by StakeSource were already involved in the project or haven’t been involved anyway. One exception was [name]. He came to meetings after StakeSource identified him. And since then, he has been absolutely invaluable.” Practitioner for P7 also compared the importance of stakeholders reported by StakeSource with his own perception. The ratings that did not agree were double-checked, and the comparison helped the practitioner view the priorities from the stakeholders’ perspective. For P8, StakeSource identified relationships between stakeholders, which the practitioner were unaware of. According to the practitioner, “In a sense it makes us aware of relationships we don’t recognise. However, it is not good to accept the results as they are, because some indirect relationships are presented as direct relationships, and some present relationships are lacking. For example, there is no link between NASA and the person who is responsible for collaboration with NASA.” The practitioner continued to use

StakeSource in the project. He explains, "if you accumulate information, trustfulness of the relationships will be improved. Then it is useful to catch the overall picture of the project."

Interviews revealed that the practitioners were keen to use StakeSource, as the tool was simple to use requiring little time and training from both the practitioners and stakeholders. In addition, the practitioners understand how and why the social networking concept works in the context. Most projects took 2 h or less for the practitioners to set up on StakeSource. The practitioner for P1 regarded the automated elicitation of stakeholder information as a significant timesaving and reported that she had, in a previous project, spent weeks to manually compile a list of stakeholders of a similar size. According to four practitioners, the majority of time was spent customising the email, as the content of the email is crucial to encourage the stakeholders' response.

The usefulness of StakeSource was also reflected in the practitioners' intention of using the tool in future projects and their recommendation of the tool to their colleagues. In the interviews, eight practitioners mentioned the use of StakeSource for future projects, and two projects were already in progress. In addition, 5 practitioners recommended StakeSource to their colleagues (Table 10.2). Some recommendations are learnt from the interviews. For example, a practitioner mentioned, "I showed this tool to xxx, his xxx project starts soon", another asked, "How long will this tool be available for? My managers may want to use it." Others are learnt from enquires by the practitioner. For example, enquiries from a defence organisation revealed that practitioner in P9 recommended the tool to them.

Finally, as reported in the previous section, StakeSource enables stakeholders to voice their opinions early in the project. As a result, it detected non-stakeholders and positive and negative stakeholders at the start of the project.

*Limitations.* As StakeSource automatically elicits information from stakeholders, its use is limited in projects where stakeholders are not incentivised to contribute [16]. Although manual approaches are time consuming for the practitioners, the presence of the practitioners can encourage reluctant stakeholders to provide information [10]. In addition, collaborative software applications such as StakeSource provide different levels of benefit to different stakeholders [16]; hence, those who do not see the benefit in contributing would be less inclined to contribute, especially when the elicitation is done by an automated tool.

The usefulness of StakeSource's output was dependent on number of stakeholder responses. In projects with higher response rate and higher number of recommendations per stakeholder, StakeSource's output was deemed to be more useful to the practitioners. The practitioners in projects with less than 10 % response rate were disappointed with the information elicited by StakeSource. As commented by the practitioner in P3 with only 1 recommendation, "it (StakeSource) didn't do much, did it?"

A practitioner posed a broader concern. If an organisation uses StakeSource for many projects, some individuals may receive an increasing number of emails asking for recommendations in various projects, which will start to take time to complete. Eventually, these individuals may start to ignore recommendation requests.

*Risks*. The automatic contact of stakeholders was not always favourable to the practitioners. In addition, although StakeSource is open and inclusive, many projects have private information that should not be shared with all stakeholders. In one of the projects, stakeholders recommended potential vendors that were bidding for the project. As the vendors were recommended, StakeSource automatically invited them to make recommendations and provided them with access to all the stakeholders in the network. This threatened to lead to an unfair bidding process. The practitioners reported, "We deleted potential vendors from the list – they should not be able to see internal stakeholder information. Can we 'approve' the stakeholders before they receive the invitation?" Other practitioners who expressed interest in StakeSource highlighted similar privacy issues. For example, a practitioner from the defence domain requested a feature to restrain StakeSource to only send emails within their organisation. Another requested for StakeSource to allow private objectives with a classified list of stakeholders and information that are only available to certain stakeholders.

According to a practitioner, mistakes are more open using the crowdsourcing technique in StakeSource. If the project is not interesting and there is no response, everyone knows about it. The same goes for mistakes in the invitation email. The practitioner was referring to an incident where a bug in StakeSource caused it to send garble html emails to the stakeholders. In addition, if practitioners continuously ignore the information provided by stakeholders, then the stakeholders may stop contributing in future projects.

### *10.5.4 Factors that Influence Stakeholder Engagement*

The application of StakeSource to multiple projects highlighted the following factors that influence the stakeholders' engagement. (Some factors are common in requirements elicitation regardless of the technology.) We conclude our discussion with a description of these factors.

*Factor 1. Number of stakeholders and location*. The automated crowdsourcing in StakeSource works best when there are many users. If there are only a few stakeholders and they are co-located, all stakeholders can communicate without needing to use StakeSource. For instance, in P9, informal recommendations were already in progress. In P4 and P6, there were too few stakeholders (2–3) to start the snowballing process. Projects with higher response rate have more stakeholders, and the stakeholders are not available at the same time, in different departments, or in distributed locations.

*Factor 2. Stake and benefit*. Stakeholders are more likely to respond when they have more stake in the project or when there is direct benefit associated with their response. For example, in P2, experts were more motivated to provide their expertise description, than to recommend other experts. Although the response rate was high, the majority of responses were description about the person's expertise in the field, rather than recommendations of other experts. Providing their expertise could get them involved in the project, but recommending other experts may mean the other experts would obtain the funding. This observation was confirmed by the practitioner, "The context is very important for StakeSource to work: stakeholders must be incentivised to make recommendations. In this project, most respondents may have been more incentivised to provide their expertise than recommend other experts." In P3, the random people in the organisation have very low stake in the project. As such, they may know parties interested in the project but did not recommend due to lack of benefit for them to do so.

In all projects, the practitioners customised the email that stakeholders will receive, in order to motivate the stakeholders to make recommendations. For example, in P1, the email started with "This project will affect you." In P5, the practitioners realised that recommendations were unlikely to benefit the recommender, hence started their email with "Help! We need your input into…"

*Factor 3. Culture*. Culture and social conventions affected the stakeholders' recommendations, which in turn influenced the effectiveness of StakeSource. Interviews revealed that the stakeholders in P8 (Japanese project) were polite and more private. The stakeholders were aware that their recommendations were not anonymous as the invitation email from StakeSource reveals the recommenders' identity. Hence, they only recommended stakeholders that they were familiar and interacted with, as they do not want to "disturb important people." As a result, stakeholders with higher positions in the organisation hierarchy were not recommended and hence omitted in the stakeholder list, although they were crucial to the project. For example, the director was not recommended but was responsible to promote the project and make budget decisions. The stakeholders might be more "free in their recommendations" had they been anonymous.

In addition, stakeholders in P8 only provided private comments, even when their comments were positive. Two stakeholders from different projects (one UK project and one Japanese project) provided the same comment about not remembering the exact details of the stakeholder they recommended. But the stakeholder in the UK project put the comment as public, while the stakeholder in the Japanese project put the comment as private.

P8 was also the only real-world project with a connected stakeholder network (Fig. 10.3). A social network is connected when there are no disconnected components. This indicates that the stakeholders who were involved in P8 were aware of one another. All the other real-world projects had disconnected components. For example, P5 had four disconnected components (Fig. 10.3). This finding is contrary to the assumption in the previous work that the stakeholder network is connected [10]. Interviews revealed that stakeholders in disconnected components are responsible for different subsystems or work for different departments.

*Factor 4. Availability*. StakeSource promises to automatically elicit information from stakeholders even when they are not physically present. Nevertheless, the stakeholders' response depended on their availability when the invitation email was received. If they were away or on holiday, they tended not to respond despite having access to the Internet. For example, P1 had a tight schedule to complete stakeholder

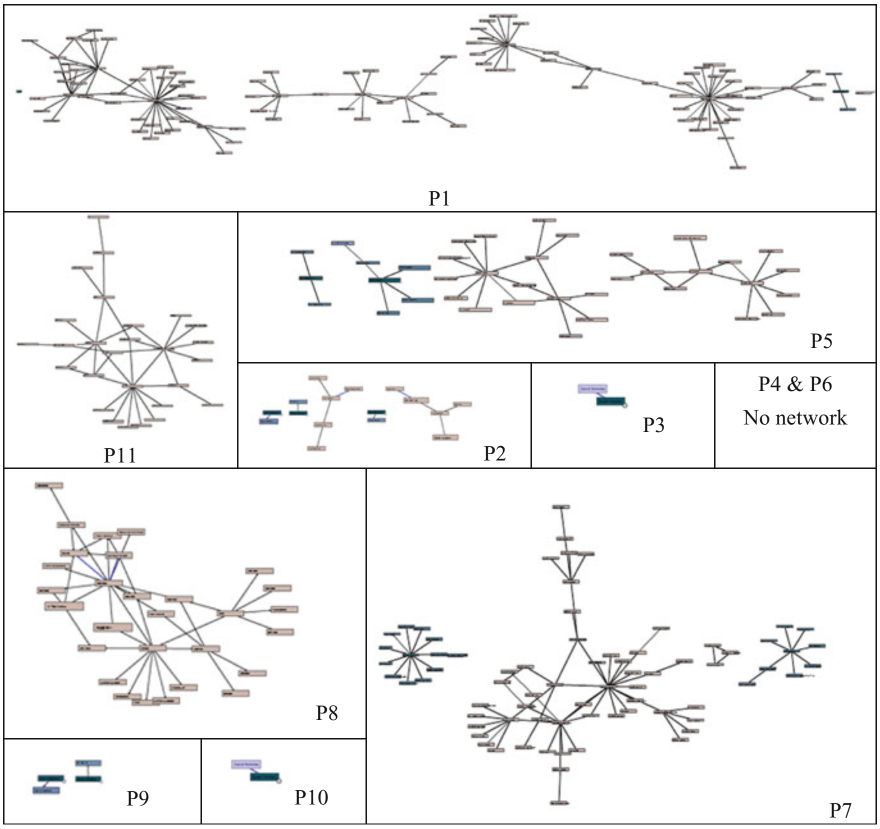


**Fig. 10.3** Social networks of stakeholders. Names are blurred for reasons of privacy

analysis by 15th Jan. The response rate was skewed by the time and duration StakeSource was used. According to the practitioners, "it was the time of year where people were particularly busy. We were warned to do it another time, but we had no choice as the meetings started in January." The duration given for stakeholders to make recommendations was just 2 weeks, excluding the Christmas break. Many stakeholders were on holiday during the recommendation period. According to the practitioners, "There is never a quiet time especially for large projects involving many departments or organisations because different departments have their own busy time. Project managers need to be cautious about the best time to run StakeSource to increase the number of recommendations."

In most projects, reminder emails were used by the practitioners a week after the initial email was sent, to remind stakeholders to respond. The reminder emails were able to encourage more stakeholders to respond. In addition, interviews with some stakeholders revealed that they did not respond because they were waiting for someone else to do it.

*Factor 5. Clarity of instructions*. The content and clarity of the project description and the invitation email is crucial to encourage response. This finding is consistent with existing literature for manual requirements elicitation approaches [25]. But because StakeSource is web-based, the importance of the clarity of instructions is increased, as the practitioners are not available to clarify or explain their intentions. In some projects, lack of response came from the stakeholders not knowing what the practitioner was looking for. For example, P5 initially received low response. The practitioner contacted some stakeholders asking if the lack of response was due to the tool being difficult to use. The stakeholders responded that they know how to use the tool, but the email description was vague and they did not understand what the project manager was looking for, hence did not make recommendations. Once the email description was clarified, the stakeholders made their recommendations.

*Factor 6. Politics*. Politics and conflicts of interest can affect the success of collaborative software applications, StakeSource or not [16, 26]. For StakeSource, political issues affected the recommendations stakeholders made and whether they made recommendations at all. In some projects, none of the stakeholders from higher-level management made recommendations, despite being influential stakeholders. A practitioner explained, "Although StakeSource is open and inclusive, some stakeholders are not. They may refrain from recommending a stakeholder to exclude their involvement in the project." Another practitioner mentioned, "They (the stakeholders) know who the other stakeholders are, but they want us to find it out ourselves." These stakeholders refuse to be engaged as their input benefits the practitioner but brings little benefit to themselves.

The practitioners also mentioned that some stakeholders are reluctant to respond to an automated tool. For these stakeholders, manual approaches using phone calls or face-to-face interviews may still be required to elicit responses. In addition, the recommendations can be biased. Some stakeholders may recommend people who are important to them in their work, regardless of whether these people will be useful for the project. Others may choose not to recommend rather than to omit recommending a stakeholder who is important to them.

## 10.6 Threats to Validity

This study is based on ten real-world projects that have used StakeSource during the first year of its deployment. Due to the variation in project size, location, and application area, there must be some caution generalising the lessons learnt to other projects. For example, cultural effects on recommendations were observed in one project P8. As more practitioners use StakeSource in their projects, additional studies should be conducted to gain further insights, and the factors that influence stakeholder engagement can be studied in more detail.

In this work, the quality of the stakeholder list returned by StakeSource was evaluated qualitatively by interviewing the practitioners. Future work should follow up with the projects when they have completed, to compare the list of stakeholders identified by StakeSource against the actual list of stakeholders in the project in terms of their pertinence. Future work should also conduct more in-depth analysis of the findings, such as analysing the relationship between project size and effectiveness of StakeSource, the effect of using StakeSource on the quality of the final product, and the properties of the different stakeholder networks.

Finally, the authors of this chapter were involved in the development and deployment of StakeSource. Due to social niceties, the practitioners' feedback on StakeSource may be positively biased. Nevertheless, these practitioners have little incentive to make socially desirable remarks, and they have been quite frank (e.g., the practitioner in P3 said that StakeSource did not do much). In addition, it was made clear to the practitioners and stakeholders that the main objective of their feedback was to improve the work. Also, their interview comments were corroborated with quantitative data and evidence. For example, we considered a practitioner X to have recommended StakeSource to practitioner Y only if practitioner Y enquired about or adopted StakeSource.

## 10.7 Conclusions

Web 2.0 collaborative tools such as StakeSource are likely to play an increasingly important role in supporting requirements elicitation, especially for emerging forms of development such as distributed development.

This chapter reports our experiences of and lessons learnt from the use of StakeSource in ten real-world projects. We learnt that the effectiveness of StakeSource in semi-automating stakeholder analysis is dependent on the stakeholders' engagement. In projects with large number of stakeholders who are motivated to contribute, StakeSource was able to elicit useful stakeholder information with little support from the practitioners. For example, StakeSource was able to uncover missing stakeholders, negative stakeholders, and the stakeholders' opinion about other stakeholders at the start of the project. Yet, it failed to elicit information when stakeholders were not incentivised enough to contribute. The main factors that influence stakeholder engagement via StakeSource include the number of stakeholders and their location, the stakeholders' motivation to be engaged, and their stake in the project. The stakeholders' culture, availability, clarity of instructions from the practitioners, and politics in the organisation also affect stakeholder engagement.

Stakeholder engagement is crucial for the success of Web 2.0 collaborative tools such as StakeSource. Future work should address the critical issue of incentives to increase stakeholder response.

**Acknowledgments** We thank the SEGAL and CHISEL research groups at University of Victoria, Peggy Storey, and Peter Bentley for their feedback on the work and the practitioners and stakeholders for their feedback on StakeSource.

## References

1. Nuseibeh B, Easterbrook S (2000) Requirements engineering: a roadmap. In: Proceedings of the conference on the future of software engineering, Limerick, Ireland, pp 35–46
2. Sharp H, Galal GH, Finkelstein A (1999) Stakeholder identification in the requirements engineering process. In: Proceedings of the database and expert system applications workshop (DEXA), Florence, Italy, pp 387–391
3. Zave P (1997) Classification of research efforts in requirements engineering. ACM Comput Surv 29:315–321
4. Macaulay L (1996) Requirements engineering. Springer Verlag, New York
5. Maiden N, Ncube C, Robertson S (2007) Can requirements be creative? Experiences with an enhanced air space management system. In: Proceedings of the 29th international conference on software engineering (ICSE), Minneapolis, MN, USA, pp 632–641
6. Gottesdiener E (2002) Requirements by collaboration: workshops for defining needs. Addison-Wesley Longman, Boston
7. Gause DC, Weinberg GM (1989) Exploring requirements: quality before design. Dorset House, New York
8. Alexander I (2005) A taxonomy of stakeholders: human roles in system development. Int J Technol Hum Interact 1:23–59
9. Lim SL (2010) Social networks and collaborative filtering for large-scale requirements elicitation. Ph.D. thesis, University of New South Wales
10. Lim SL, Quercia D, Finkelstein A (2010) StakeNet: using social networks to analyse the stakeholders of large-scale software projects. In: Proceedings of the 32nd international conference on software engineering (ICSE), vol 1, Cape Town, South Africa, pp 295–304
11. Lim SL, Quercia D, Finkelstein A (2010) StakeSource: harnessing the power of crowdsourcing and social networks in stakeholder analysis. In: Proceedings of the 32nd IEEE international conference on software engineering (ICSE), vol 2, Cape Town, South Africa, pp 239–242
12. Cleland-Huang J, Mobasher B (2008) Using data mining and recommender systems to scale up the requirements process. In: Proceedings of the 2nd international workshop on ultra-large-scale software-intensive systems, Leipzig, Germany, pp 3–6
13. Serrano N, Torres JM (2010) Web 2.0 for practitioners. IEEE Softw 27:11–15
14. Damian D (2007) Stakeholders in global requirements engineering: lessons learned from practice. IEEE Softw 24:21–27
15. Storey M, Treude C, van Deursen A, Cheng L (2010) The impact of social media on software engineering practices and tools. In: Proceedings of the FSE/SDP workshop on the future of software engineering research, Santa Fe, New Mexico
16. Grudin J (1994) Groupware and social dynamics: eight challenges for developers. Commun ACM 37:92–105
17. Oreilly T (2007) What is web 2.0: design patterns and business models for the next generation of software
18. Pouloudi A, Whitley EA (1997) Stakeholder identification in inter-organizational systems: gaining insights for drug use management systems. Eur J Inform Syst 6:1–14
19. Cheng BHC, Atlee JM (2007) Research directions in requirements engineering. In: Proceedings of the conference on the future of software engineering, Minneapolis, MN, USA, pp 285–303
20. Alexander I, Robertson S (2004) Understanding project sociology by modeling stakeholders. IEEE Softw 21:23–27
21. Scott J (2000) Social network analysis: a handbook. Sage, Thousand Oaks
22. Lindlof TR, Taylor BC (2002) Qualitative communication research methods. Sage, Thousand Oaks
23. Deutskens E, De Ruyter K, Wetzels M, Oosterveld P (2004) Response rate and response quality of internet-based surveys: an experimental study. Mark Lett 15:21–36

24. Yu J, Cooper H (1983) A quantitative review of research design effects on response rates to questionnaires. J Mark Res 20:36–44
25. Thayer RH, Dorfman M (1997) Software requirements engineering. Wiley-IEEE Computer Society Press, Los Alamitos
26. Dobson J, Blyth A, Chudge J, Strens R, Dobson J, Blyth A, Chudge J, Strens R (1994) The ORDIT approach to organisational requirements. In: Requirements engineering: social and technical issues. Academic Press Professional, San Diego, pp 87–106

# Part IV
# Reasoning About Requirements

“There are also two kinds of truths: truth of reasoning and truths of fact. Truths of reasoning are necessary and their opposite is impossible; those of fact are contingent and their opposite is possible.”
—Gottfried Leibniz
© Walid Maalej. Printed with permission

# Chapter 11
# Resolving Inconsistency and Incompleteness Issues in Software Requirements

**R. Sharma and K.K. Biswas**

**Abstract** In this chapter, we present an approach toward better understanding and analyzing significant aspect of software – the requirements. Comprehending the semantics of requirements is crucial to the success of the intended software. In order that the software requirements be well understood, it is imperative that these must be complete and consistent. But, the elicited requirements are often incomplete, inconsistent, and, consequently, ambiguous in nature. Requirements engineer is presented with the task of examining and analyzing such requirements and detecting these issues, i.e., instances of incompleteness and inconsistency. We present here courteous logic-based representations of requirements as an approach toward resolving the issues of incompleteness, inconsistency, and ambiguity in the elicited requirements and assisting improved understanding of elicited requirements. We explain how courteous logic can be an effective solution to requirements interpretation in terms of observable behavior of the system and can be a useful tool for requirements engineer toward improving the quality of software requirements. We will be more concerned toward inconsistency and incompleteness issues in this chapter.

## 11.1 Introduction

Requirements engineering (RE hereafter) is the basis for subsequent phases of software development cycle. Software design and development take off once the requirements are well understood and agreed upon by the developers, clients, and the involved stakeholders. The question of well or better understanding of the requirements calls for an adequate representation of requirements. IEEE's recommended guidelines [1] for good requirements specification suggest that

R. Sharma (✉) • K.K. Biswas
IIT Delhi, New Delhi, India
e-mail: sricha@gmail.com; kkb@cse.iitd.ernet.in

requirements should be correct, complete, consistent, unambiguous, verifiable, and traceable. The guidelines precisely define these characteristics in terms of requirements specified or mentioned in the document, referred to as software requirements specification. We however stress that these attributes are equally applicable to the requirements representation in context of the domain under study, albeit the interpretations differ. Correctness, completeness, and consistency of requirements have been described in detail in [2]. Zowghi and Gervasi suggest that correctness has at least two different perspectives:

1. From a formal point of view, correctness is usually meant to be the combination of consistency and completeness. Consistency refers to situations where a specification contains no internal contradictions, whereas completeness refers to situations where a specification entails everything that is known to be "true" in a certain context. Consistency is an internal property of a certain body of knowledge, whereas completeness is defined with respect to an external body of knowledge.
2. From a practical point of view, however, correctness is often more pragmatically defined as satisfaction of certain business goals. This indeed is the kind of correctness which is more relevant to the customer, whose goal in having a new system developed is to meet his overall business needs.

Our focus in this chapter is on business requirements pertaining to enterprise business applications. Such applications are data-intensive and are driven by the constraints, properties, and rules of the business domain under consideration. Therefore, our approach toward correctness of requirements is not merely from documentation point of view but also more from the point of view of the business domain. We, therefore, prefer to follow the perspectives drawn out in [2]. We argue here that it is the practical viewpoint that must translate to formal point of view so that requirements defined in terms of business goals hold good and can be well understood by the developers too at the time of implementation. The ultimate objective of software development is to develop the software that satisfies business goals. The software design and development activities, that are more of formal tasks in terms of the corresponding language specification, have to meet the objective of software development. Therefore, it would be wise to say that practical viewpoint of notion of correctness needs to strike chord with the formal viewpoint. Requirements engineer who is supposed to establish correct, complete, consistent, unambiguous, and verifiable requirements must take into account formal viewpoint to achieve his objective. We will explore how the formal notion of correctness of requirements leads to the question of an "adequate representation" of requirements and how both these ideas are related in following sections. The motive behind determining adequate representation is that it is the central problem in requirements engineering. An adequate representation of requirements can be helpful in well-understanding of requirements and establishing high quality of requirements. Though there is no precise definition to the "adequate representation," yet numerous authors ([3, 4, 5, 6]) have proposed various desirable features of requirements representations independently. We will discuss these features in the sections to be followed.

Business requirements are usually represented using natural languages and, therefore, often suffer from the drawback of formal analysis, verification, and validation in the absence of reasoning engine for natural language. The requirements gathered through client interactions, observations, or interviews tend to be incomplete and ambiguous as clients might fail to express their expectation from the software system clearly and precisely. This kind of informal communication results in vague and ambiguous requirements in first place. This concern is further aggravated by the fact that natural language itself is ambiguous and can have multiple interpretations given varying contexts. Natural language representation is, therefore, not an apt choice to represent requirements and establish their correctness. Some degree of formalism is required to overcome the defects of inconsistency and incompleteness – together which contribute to the notion of incorrectness of requirements. Of the various formal approaches to express business requirements, logical representations have been widely acknowledged with the increasing complexity and expectations of software systems. Real-world business requirements correspond to human way of thinking and commonsense (non-monotonic) reasoning. These aspects require that any logical specification of requirements should be able to handle non-monotonic reasoning. We have made use of courteous form of non-monotonic logic to express requirements. Requirements are usually captured in natural language; hence, it makes sense to translate these natural language expressions to logical expressions to a certain extent. We are prototyping a tool that can build domain model from the given set of requirements. This chapter contributes toward formal representation of requirements in terms of following points:

1. We present courteous logical representation of requirements as an effective solution for requirements representation. We show that these representations have the desirable features for establishing correct and verifiable requirements.
2. We show how correctness of the requirements can be verified at an early stage, i.e., how courteous logic-based representations help in resolving inconsistency and incompleteness in requirements during RE phase. This would minimize the number of defects that would otherwise permeate further phases of software development.
3. We present prototype of the tool that we are working upon for generating domain model from requirements.

Section 11.2 presents the requirements engineering problem followed by a discussion of contributions from numerous authors on desirable features of requirements representation in Sect. 11.3. This section also presents a consolidated list of desirable features for requirements representation that can be referred to as an "adequate representation" – one that can be said to be effective toward detecting and resolving inconsistency and incompleteness. Section 11.4 then presents our contribution in this chapter. We have made use of courteous logic to represent requirements following the recommendations of inference mechanism [4] and non-monotonic logic [5]. We present how courteous logic semantics are helpful in exploring and detecting instances of inconsistency and incompleteness in software requirements in case studies in Sects. 11.5 and 11.6. We also found that these representations satisfy most of the desirable features of requirements illustrated in Sect. 11.3. The related work is discussed in Sect. 11.7 followed by discussion and conclusion in Sect. 11.8.

## 11.2 The RE Problem

The introduction section has spawned sufficient background for the issue of correctness of business requirements and an adequate representation of the requirements. We have seen that the motivation behind discussing the requirements representation is that it has remained the *central problem in RE*, and an adequate representation only can offer an approach toward handling inconsistency and incompleteness issues. Determining presence of inconsistency and incompleteness requires reasoning the given set of requirements and drawing inferences from it. The representation of requirements should have some features to serve this purpose. We will discuss these features in next section, but it suffices to make the point clear that the concern of proving correctness of requirements is closely related to the concern of requirements representation. We will first discuss the RE problem in detail before moving on to consistency and completeness concerns.

Software development practices so far have varied from structured programming to object-oriented programming to aspect-oriented programming. The RE approaches have also varied in accordance to the programming practices followed, and consequently, a number of requirements model and representation practices surfaced. Structured programming resulted in structured analysis of the requirements, i.e., activity-flow-oriented analysis. The deliverables of structured analysis include context diagrams, data flow diagram, and structure charts [7]. Object-oriented analysis [8] results in a number of UML diagrams including class and object diagrams and activity diagrams and interaction diagrams. Aspect-oriented analysis [9] results in representation of aspects, cut points, and joins.

Despite several approaches in place, the *central requirements problem* remained same across these approaches: *what* is to be represented and *how* it is to be represented. As suggested in [10], the requirements problem amounts to finding the representation $S$, that for given domain assumption K satisfies the given requirements $R$. If $K$, $S$, and $R$ are represented in mathematical logic, then the requirements problem is solved once the requirements engineer finds $S$ such that $K, S \vdash R$.

The questions relevant to the requirements problem become important from the point of view of how the representations are going to be utilized. The analyst is concerned with representation as well as the ability of the representation to yield itself to validation and verification [3]. Tsai and Weigert [4] have proposed that if an inference mechanism is available for requirement specification, then it can be used to validate the requirements. Logical representations offer a solution to understanding the semantics of requirements and validating them. Most of the formalism for requirements representation is often reducible to first-order logic. RML [11] and Telos [12] representations find their well-formedness and semantics in the roots of first-order logic. KAOS [13] and i* [14] representations are goal-oriented in nature. The advantage of formal representation of requirements lies in reasoning with representations for inferencing purpose.

*Consistency Concern* – Consistency concern among requirements requires that no two or more requirements in the given requirement set contradict each other.

There can be various reasons for contradictions to arise. A simple reason could be that requirements specified in natural language are interpreted differently by different people involved during RE phase. Another possible reason can be because of the fact that the terminology used by clients and stakeholders might differ from the terminology familiar to developers. Inconsistency also arises when multiple stakeholders have varying views and appropriate prioritization on a conflicting scenario has not been considered. Consistency has been defined and reviewed by various authors in number of ways independently like [15, 16, 17], and many more. Multiple stakeholders' view has been taken into account in [15]. Balzer has proposed approach for tolerating and minimizing inconsistency in [17].

With growing complexity of the software applications being developed, it becomes imperative to effectively manage inconsistency before commencing the design and development phase. The complex nature of software with lot many inconsistency will eventually result in budget and schedule overruns.

*Completeness Concern* – Completeness of the requirements is the most difficult to define and determine. The IEEE guidelines state completeness in terms of the information covered in software requirements specification that there should not be any information undefined or "to be determined." But, there is another dimension to completeness in context of the business domain under consideration. This aspect considers that the requirements should cover all the necessary information required for problem definition and processing. The notion of completeness has more to do with the scope and goals of the system to be developed. Goal-oriented RE therefore focuses more on goals. Letier and Lamsweerde [18] have also suggested specifying the goals explicitly and that goals offer a criterion to measure requirements completeness.

## 11.3 Desirable Features for Representation of Requirements

Section 11.2 has laid out the foundation for the expectations from requirements representation. In order that these expectations should be satisfied, requirements representations must have some desirable features. As introduced in Sect. 11.1, several authors have suggested desirable features for representations of requirements independently at different times. We will discuss some of these features in this section.

In [3], Zualkernan and Tsai have suggested following criteria for a language to qualify for problem specification and consequently requirements specification:

1. *Epistemological Adequacy* – Also called as expressive power in context of knowledge representation and refers to the adequacy of a formal system to represent concepts in real-world.
2. *Synthesis Adequacy* – It refers to the criteria of maintainability from representation perspective and availability of procedures for constructing specification from methodology point of view.

3. *Analysis Adequacy* – It refers to the adequacy of representation to yield itself to validation and verification.
4. *Contractual Adequacy* – This is the adequacy of the representation to support interaction with domain experts or clients.
5. *Blueprint Adequacy* – This is the adequacy of problem specification to serve as blueprint for next stages of development.

Tsai and Weigert [4] have proposed that if the requirements can be operationally interpreted, then the user would be exposed to a "working model" of the system at an early stage of development. Secondly, if an inference mechanism is available for the requirement specification, then it can be used to validate the requirements. They propose following demands on a language to express requirements:

1. Free from implementation concerns
2. Ability to check the validity of the requirements
3. Ability to express both the objects and the relations of the domain, as well as the constraints on them and rules about them in a straightforward manner
4. Should be sound and complete
5. Should allow for operational interpretation of the system behavior

Tse and Pong in [6] have recommended following points of consideration for requirements specification:

1. *Abstraction of the real-world* – The requirements language must allow abstracting out the important issues from nonessentials. This will help in improving the conceptual clarity of the problem.
2. *Manipulation of the requirements* – Requirements specification must be structured in such a way that parts of it can be modified as requirements gradually evolve. Requirements should be expressible in a precise notation with a unique interpretation to avoid ambiguity.
3. *Construction of real-world system* – Requirements specification should be independent of the design and implementation issues. The elements in specification should be traceable to the final design and implementation.

In [5], the recommendation is to deal with over-abstraction and ensuring consistency of the model. It is further proposed that AI research on non-monotonic logics may provide insight on how to make final specification logically consistent.

A careful look at above discussed features too shows overlaps. Since we are looking for an adequate representation of requirements that can help us expressing the requirements correctly, we can say that adequacy of representation can be looked at in terms of the observable behavior of the system. We agree with the recommendations in [2] and [3] that availability of inference mechanism for software requirements can support validating the requirements against the expected observable behavior of the system and that non-monotonic logic may provide insight to make final specification logically consistent. The motivation for non-monotonic logic comes from the fact that software requirements correspond to some real-world system where there can be conflicts as well as exceptional scenarios occurring quite often and these are resolved by commonsense reasoning on a day-to-day basis. We have consolidated the desirable features in our case as:

1. The requirements specification should be an *abstract representation* or model of the real-world.
2. The requirements specification *should not be affected by the design and development* of the information system.
3. It can be *subjected to validation and verification* in terms of the observable behavior of the real-world, i.e., an inference mechanism should exist.
4. It should be *sound and complete*. Any modification to existing requirements should not result in an inconsistent set of requirements, i.e., it should support non-monotonic reasoning.
5. The requirements specification should be *maintainable and traceable* to design and development artifacts.
6. It should be able *to act as a bridge between the user or client and the development team*, i.e., it should be well understood by both parties

## 11.4 Our Approach

In this chapter, we present courteous logic-based representation of requirements as a solution toward representing requirements in a way that assists in detecting and resolving inconsistency and incompleteness in the elicited requirements. Courteous logical representation (CLP) is based on non-monotonic reasoning. It is an expressive subclass of ordinary logical representation (OLP) having procedural attachments to it for prioritized conflict handling. Having the ability to resolve conflicts, CLP representations can keep the existing knowledge base consistent while resolving conflicting opinion or exception by adding some new facts or information or removing some existing details.

*Non-monotonic Logic* – Non-monotonic logic refers to formal logic where consequence relation is not monotonic. An important goal of knowledge representation and reasoning is to reach only the true conclusions, and we also require that our conclusions are justified. These two constraints are not same. Justification preserving is not always monotonic, and human reasoning too is not monotonic and, therefore, the need for non-monotonic logic. There are several forms of non-monotonic logics. Default logic, proposed by Raymond Reiter [19], is one of the oldest and most studied non-monotonic logic. Default logic has found its application for inconsistency handling in requirements in [20].

Defeasible logic, proposed by Donald Nute [21], is based on strict rules, defeasible rules, and defeaters. Conclusions are tentative in the sense that a conclusion can be withdrawn when there is a new piece of information.

Courteous logic [22] is based on prioritization of the rules. This is in contrast to defeasible logic where rules are marked as strict and defeasible rules. We will discuss features of courteous logic in next subsection.

*Courteous Logic* – Courteous logical representation is an expressive subclass of ordinary logical representation with which we are familiar, and it has got procedural attachments for prioritized conflict handling. First-order logic beyond logic programming (LP) has not become widely used for two main reasons: it is pure belief language, it cannot represent procedural attachments for querying and actions, and it is logically monotonic; it cannot specify prioritized conflict handling which are logically non-monotonic. The courteous logic programming (CLP) extension of LP is equipped with classical negation and prioritized conflict handling. CLP features disciplined form of conflict handling that guarantees a consistent and unique set of conclusions [22]. CLP provides a method to resolve conflicts that arise in specifying, updating, and merging rules. Our CLP representations are based on IBM's CommonRules, available under free trial license from IBM alpha works [23].

Syntactically, courteous logic program is defined as a restricted class of extended logic programs, in which, additionally, rules have labels. In an extended logic program, each rule is of the form:

$$L(0) \leftarrow L(1) \wedge L(2) \wedge \ldots . \wedge Lm \wedge {\sim} L(m+1) \wedge \ldots . \wedge {\sim} L(n)$$

Here, each L(i) is a literal of the form A or $\neg$A, where A is an atom and $\neg$ stands for classical negation and ~ stands for negation-as-failure operator. L(0) is the head of the rule, and L(1), L(2), etc., constitute the body of the rule. The rule can be interpreted as L(0) holds true if the body of the rule evaluates to true. Such a rule will optionally have a label before L(0) in CLP. The label is used as a handle for specifying prioritization information. Each label represents a logical term, e.g., a logical 0-ary function constant.

In addition to label, there is binary predicate "overrides" to specify prioritization. The "overrides" predicate is used to specify prioritization. "Overrides (lab1, lab2)" means that any rule having label "lab1" is higher priority than any other rule having label "lab2." The scope of what is conflict is specified by pair-wise mutual exclusion statements called "mutex's." For example, a mutex (or set of mutex's) might specify that there is at most one amount of discount granted to any particular customer. Any literal may be classically negated. There is an implicit mutex between p and classical-negation-of-p, for each p, where p is a ground atom, atom, or predicate.

The answer set is defined incrementally and constructively by means of partial answer sets S(i) that are built up iteratively by generating conclusions for each ground atom along the way taking into account the partial order relation specified in overrides predicate. It has been proved in [22] that every courteous logic program has exactly one answer set which is consistent.

*An example illustrating the expressive and reasoning power of CLP*: Consider following rules for giving discount to customer:

- If a customer has Loyal Spending History, then give him 5 % Discount.
- If a customer was Slow to pay last year, then grant him No Discount.
- Slow Payer rule overrides Steady Spender.
- The amount of discount given to a customer is unique.

These rules are represented in CLP as following set of rulebase:

```
<steadySpender>
 if    shopper(?Cust) and spendingHistory(?Cust, loyal)
 then  giveDiscount(percent5, ?Cust);
<slowPayer>
 if    slowToPay(?Cust, last1year)
 then  giveDiscount(percent0, ?Cust);
overrides(slowPayer, steadySpender);
```

As discussed above, there are two types of customers labeled as <steadySpender> and <slowPayer> each allowed different discounts on purchase. The conflict arises when a customer qualifies to be both steadyspender as well as slowPayer. In such a conflicting situation, the predicate "overrides" is used to prioritize the discount attributed to slowpayer over the discount to be given to steadyspender.

*Advantages of Courteous Logic* – We found that courteous logic representation is more suitable for our cause for two main reasons: first, it supports non-monotonic reasoning and, second, the representation can be well understood by the involved parties during the requirements phase of software development. Courteous logic representations are closer to natural language representation of business rules in terms of commonly used "if-then" rules and can be interpreted by both the users and the developers. An experience with CLP shows that it is especially useful for creating rule-based systems by nontechnical authors too [22]. We present a case study in the following section with the idea of showing that courteous logic-based requirements specification satisfies the desirable features of requirements specification language. We were able to uncover ambiguities and resolve inconsistency and presuppositions in the studied requirements set using courteous logic-based requirements specifications.

Another advantage of CLP is computational scalability: inferencing is tractable (worst-case polynomial time) for a broad expressive case. By contrast, classical logic inferencing is NP-hard for this case. This is an additional advantage to the two main reasons for which we chose courteous logic. Key to courteous programs' computational and conceptual simplicity is that conflicts are resolved locally: by refutation and skepticism among rules that mention (positively or negatively) the same head atom. There is a unique answer set which facilitates understandability, especially by nontechnical end users. We found tractability and uniqueness of answer set features useful and attractive in context of resolving inconsistency and incompleteness in requirements.

*Processing the Natural Language Requirements* – As discussed in introduction section, normally the stakeholders and clients involved in RE phase of software development are not comfortable with formal logical language; hence, natural language has been the most preferred means to express requirements. The requirements engineer or analyst is posed with the task of analyzing the gathered requirements; generate models or representations from the given specification; detect any errors (incorrectness, ambiguity, inconsistency, and incompleteness); and then, finally validate the requirements against the expected behavior of the system. It would be an additional advantage if models or representations can be

| | |
|---|---|
| Original Requirement (Scenario: Direct Entry of grades by course coordinator) | Student can be assigned grades that are validated by grade rules. User initiates the 'Direct grade entry' Process. User selects the academic year, semester and course of which he wants to submit grade. Student list is displayed based on the above selection along with the possible valid grades. User selects the grades for all the students and submits the information to save. |
| After Morphosyntactic analysis | Student/NN can/MD be/VB assigned/VBN grades/NNS that/WDT are/VBP validated/VBN by/IN grade/NN rules/NNS./.User/NN initiates/VBZ the/DT `/`` Direct/JJ grade/NN entry/NN '/'' Process/NNP ./. User/NN selects/VBZ the/DT academic/JJ year/NN ,/, semester/NN and/CC course/NN of/IN which/WDT he/PRP wants/VBZ to/TO submit/VB grade/NN ./. Student/NN list/NN is/VBZ displayed/VBN based/VBN on/IN the/DT above/JJ selection/NN along/IN with/IN the/DT possible/JJ valid/JJ grades/NNS ./. User/NN selects/VBZ the/DT grades/NNS for/IN all/PDT the/DT students/NNS and/CC submits/VBZ the/DT information/NN to/TO save/VB ./. |
| Entities Presented to analyst | Student grades grade rules User entry Process year semester course list selection information |
| Finalized Entities | Student grades year semester course |
| Equivalent logical formula | Student(?X) grades(?X)<br>year(?X) semester(?X)<br>course(?X) |

**Fig. 11.1** Generating domain entity model from NL requirements

auto-generated from the given specifications. We are currently working on tool intended to generate domain model from the natural language specifications and, then, to allow analysts to design processing rules for the domain model. Our idea is to manually capture not just rules but also the constraints and exceptional scenarios for the generated domain model. Analyst would be prompted to ensure that he has covered all these aspects relevant to a scenario being modeled.

Generating domain model requires parsing process in place to translate NL sentences to logical formulae. The parsing process consists of typographical adjustments, tokenization, and morphosyntactic analysis. The processed text is then parsed to produce a parse tree corresponding to original statement. Morphosyntactic analysis is done with the help of Stanford-postagger [24]. We are refining the results of morphosyntactic analysis by presenting them to requirements analyst. We present the analyst the nouns (that would be marked as entities in the domain) from the result of morphosyntactic analysis, and he is supposed to remove, add, or update any entity if required. The refined collection of nouns is then translated to domain entity model. The steps discussed are exemplified in Fig. 11.1:

## 11.5 Case Study

We studied the registration, grade-processing, and graduation part of an educational institute to carry out our study. The reason behind choosing these two processing parts was that these are rule-intensive, and we took the challenge of writing the given use-cases using courteous logic representations and, then, ensure that consistency between rules is preserved. We were presented processing information in the form of use-cases and here we will take the running example of grade-processing part.

Our aim in conducting this experiment was to determine if courteous logic-based specifications meet the desirable features as discussed in Sect. 11.3, especially in the fourth feature that is relevant to consistency and completeness. Our aim in conducting this experiment was to determine if courteous logic-based specifications help in detecting and resolving inconsistencies and incompleteness, if any. The experiment started with the identification of entities, relevant attributes, and relations in the given use-cases. We identified a total of eight entities along with their attributes in this subsystem. We translated the processing part of the use-cases under study to courteous logic representation. Much to our satisfaction, it didn't turn out to be an arduous task. The information obtained was already structured in the form of use-cases and secondly, the translation part didn't require much experience with CommonRules. Before proceeding to discuss each point from desirable features of requirements specification language, we illustrate a sample from our representations so that observations become clearer:

```
  <new>
    if    assignGrades(?Regno, ?Year, ?Sem, ?Group, ?Sub,
?Point)
   then  valStatus(new, ?Regno, ?Year, ?Sem, ?Group, ?Sub);
```

The representation indicates that when grades are assigned first time to a student with registration Id, *Regno;* year as *Year*; semester and group as *Sem* and *Group* respectively; with subject and grade point as *Sub* and *Point* respectively, then the status of that grade would be assigned as 'new'.

We conducted one more experiment in Corporate Action Event domain to verify that courteous logic representations can scale to a complex domain and still ensure consistency among the requirements. We present findings from this example-study while discussing the consistency feature in Sect. 11.6.

## 11.6 Observations from Case study

Our observations for the case studies conducted can be summarized in terms of the desirable features of requirements specifications as compiled below:

1. *Abstract representation of real-world* – Within the periphery of given use-cases, identifying the entities; their attributes and relations was not ambiguous.

We could represent the behavior of the system by capturing the governing rules and the constraints imposed at a high level without delving into any sort of implementation details. This representation itself worked as a 'working model' for the subsystem under study. This point would be clearer with the scenario presented in next point.

2. *Not affected by design and development details* – The requirements are modified only when something new is encountered at a later stage. In our case, we didn't have to tweak our requirements specifications later as the development details started pouring in. An interesting and relevant scenario was with grade conversion use-case:

The Grade Conversion use-case was defined with *preconditions* stating:

Grade rules must be defined in the system and grades should be submitted and approved.

The *processing part* of the use-case stated:

User initiates 'Grade Conversion' process. User selects academic year, semester and course of which he wants to change the grade. User then selects the student names and can optionally give remarks while converting the grade. The updated information is saved.

The *post-condition part* stated:

Grade status gets modified and entire workflow of grade approval is initiated.

We represented this use-case as:
*<gradeMod>*
*if convertGrades (?Regno, ?Year, ?Sem, ?Group, ?Sub, ?Status, ?OldPoint, ?NewPoint)*
*then valStatus (new, ?Regno, ?Year, ?Sem, ?Group,?Sub);*

The representation indicates that any grade conversion request for a student with registration Id, *Regno*; year as *Year*; semester and group as *Sem* and *Group*, respectively; with subject and current grade status as *Sub* and *Status* respectively would change the status of grades to *new*, which calls for grade approval process as is the case with new grade assignment:
*<new>*
*if assignGrades(?Regno, ?Year, ?Sem, ?Group,?Sub, ?Point)*
*then valStatus(new, ?Regno, ?Year, ?Sem, ?Group,?Sub);*

The design team preferred to have grade approval part as a component reusable at the time of regular/new (without conversion) grade approval and at the time of grade conversion. The development team, on the other hand, couldn't use that component in its entirety owing to technical reasons and had to go by the choice of separate components. We have two observations here to make:

(a) The representation remained unaffected by the choices design team and development made. It just represented the abstracted underlying idea that grade change would entail invoking the grade approval processing right from the very beginning.
(b) The abstract idea had a suitable representation in the use-case too, but courteous logic representation was quite compact as compared to the use-case one.

Secondly, we could verify the observable behavior of grade change as courteous logic specifications represented working model of the system.

3. *Validation in terms of observable behavior of the system* – We have already made a reference to this observation in the above point that courteous logic-based specifications represent working model of the system. We would like to explore this observation in terms for requirements analysis for two main issues that analysts encounter, namely, *resolving ambiguity* and *finding incompleteness*.

   Direct grade change use-case was defined with administrator as actor and *preconditions* stating:

   Grade rules must be defined in the system, and grades should be submitted and approved.

   The *processing part* stated:

   User initiates "direct grade change." User selects academic year, semester, and student whose grades need to be changed. A list is displayed showing the student's grades in the respective courses he enrolled. User submits the information to save.

   The *post-condition* part stated:

   Grade submitted are finalized.

   A quick look at the use-case description apparently doesn't pose any problem. But, when this use-case was studied with other use-cases where three different levels of approval are defined, we were immediately posed with these questions:

   (a) When can admin intervene for direct grade change?
   (b) What is meant by grade finalization?
   (c) Do finalization and approval refer to same grade status? What happens if there is no direct grade change request? How would finalization of grades happen?
   (d) Can this use-case be executed again after finalization?

   It's quite possible that different people are working on different use-case and may not uncover incompleteness in their use-case in context of the environment. This use-case had ambiguity in terms of approval and finalization status, and this was overlooked in the textual use-case. When we approached subject matter expert (SME hereafter), then we found that there was a presupposition in the grade-processing use-cases. It was assumed that approval by dean refers to finalization of grades and that the above use-case would be executed in exceptional scenario only when grades are finalized (that called for a change in precondition too). In such a scenario the system administrator will do the needful as mentioned in this use-case.

   We could successfully find above-mentioned ambiguity and incompleteness at the time of requirements specification only.

4. *Ensuring consistency* – We will present two case studies in this context:

   *Grade-Processing Example:*

   This is the main focus point that motivated us to make use of courteous logic-based representation of requirements. It's important while representing and

analyzing the requirements that any modification to current requirements should not result in any sort of inconsistency.
We would like to bring out this point following the same use-case as discussed in point 3. Our representation had already captured grade approval part and the facts whose approval would be given priority as:
When we filled for the gaps in the use-case as discussed in point 3, we could easily modify the requirements without any inconsistency creeping in as; courteous logic representation has already provided means of prioritized conflict handling. We could also validate the observable behavior of the system by executing different scenarios. Existing requirement specifications were:

*<**new**>*
*if assignGrades(?Regno, ?Year, ?Sem, ?Group,?Sub, ?Point)*
*then valStatus(new, ?Regno, ?Year, ?Sem, ?Group,?Sub);*
*<**cdn**>*
*if approvedby(?Regno, ?Year, ?Sem, ?Group,?Sub, ?Point, ?Status, coordinator)*
*then valStatus(coordApproved, ?Regno, ?Year, ?Sem,?Group, ?Sub);*
*<**hod**>*
*if approvedby(?Regno, ?Year, ?Sem, ?Group,?Sub, ?Point, coordApproved, hod)*
*then valStatus(hodApproved, ?Regno, ?Year, ?Sem,?Group, ?Sub);*
*<**dean**>*
*if approvedby(?Regno, ?Year, ?Sem, ?Group,?Sub, ?Point, hodApproved, dean)*
*then valStatus(deanApproved, ?Regno, ?Year, ?Sem,?Group, ?Sub);*
*<**gradeMod**>*
*if convertGrades(?Regno, ?Year, ?Sem, ?Group,?Sub, ?Status, ?OldPoint, ?NewPoint)*
*then valStatus(new, ?Regno, ?Year, ?Sem, ?Group,?Sub);*
*overrides(cdn, new);*
*overrides(hod, new);*
*overrides(dean, new);*
*overrides(hod, cdn);*
*overrides(dean, cdn);*
*overrides(dean, hod);*
*overrides(gradeMod, cdn);*
*overrides(gradeMod, hod);*
*overrides(gradeMod, dean);*

Adding updated information for 'Direct Grade Change' use-case entailed specifying:

*<**adminMod**>*
*if convertGrades(?Regno, ?Year, ?Sem, ?Group,?Sub, deanApproved, ?OldPoint, ?NewPoint)*
*then valStatus(deanApproved, ?Regno, ?Year, ?Sem, ?Group, ?Sub);*
*Corporate Action Event Processing Example*

Consider a corporate action event announced on a security. If a client is holding the security on which event is announced, then that client is eligible to get the announced benefits of the event. These benefits can either be in the form of cash or stock or both. The types of benefits disbursed to the clients vary from one event type to another; it also depends on various other factors like base country of the security on which event is announced, the country of the customer, client opting for an option, etc. Then, there can be multiple stakeholders having differing views like one particular stock market has rules that do not allow client to opt any option announced on event; whereas, clients from some other market can opt for event's announced operations, so on and so forth. We took a small subset of this large set of rules and gradually scaled the rules as well as the data to find that results are consistent with the actual observable expectations. This particular example served toward claiming scalability of courteous logic-based requirements specifications. Our expressions could not only be easily validated against the real-world expected behavior, but also these were small and compact making them easy to comprehend and verify against multiple real-world scenarios as shown below:

***<cash>***
*if event(?EventId, ?Type, ?Security) and holds(?Client, ?Security) and opts (?Client, cash)*
*then distribute(?Client, ?EventId, cash);*
***<stock>***
*if event(?EventId, ?Type, ?Security) and holds(?Client, ?Security) and opts (?Client, stock)*
*then distribute(?Client, ?EventId, stock);*
***<both>***
*if event(?EventId, ?Type, ?Security) and holds(?Client, ?Security) and opts (?Client, both)*
*then distribute(?Client, ?EventId, both);*
***<divMtk1>***
*if event(?EventId, dividend, ?Security) and holds(?Client, ?Security) and baseCntry(?Security, Mkt1)*
*then distribute(?Client, ?EventId, stock);*
***<divMkt2>***
*if event(?EventId, dividend, ?Security) and holds(?Client, ?Security) and clientCntry(?Client, Mkt2)*
*then distribute(?Client, ?EventId, nothing);*
***<divMkt1Mkt5>***
*if event(?EventId, dividend, ?Security) and holds(?Client, ?Security) and baseCntry(?Security, Mkt1) and clientCntry(?Client, Mkt5)*
*then distribute(?Client, ?EventId, cash);*

The rule with label as < *cash* > indicates that if an event, ?Event of some type, ?Type is announced on a stock, ?Security and a client, ?Client is holding that stock and he opt for *cash* option then he will receive the benefit of event in the form of cash as per the announced rules of the event. Similarly, use-cases with

stock and both types of disbursements are represented through rules labeled as *stock* and *both,* respectively. These are generic rules. Next, we have considered a hypothetical scenario where in stakeholders from stock market, *Mkt1* are of the view that if "dividend" type of event is announced on the stock belonging to their nation, then all customers shall get event's benefits as stock only. This is represented in the rule labeled as < *divMkt1*>. The rule with label < *divMk2* > indicates that dividend event announced will not entail any benefits to clients from stock market *Mkt2*. The last rule is an exception to rule < *divMkt1* > – it says that if client hails from the stock market, *Mkt5,* then he is eligible for benefit in the form of cash rather than stock. The above-mentioned rules were then verified against facts from real-world as below:
*event(11, dividend, samsung);*
*event(22, dividend, dell);*
*baseCntry(dell, US);*
*holds(abc, samsung);*
*holds(abc, dell);*
*holds(xyz, dell);*
*holds(pqr, dell);*
*clientCntry(xyz,Mkt2);*
*clientCntry(pqr, Mkt5);*
*opts(abc, both);*
In the absence of any kind of prioritization among multiple views, we got the *validation results* as:
*distribute(pqr, 22, cash);*
*distribute(pqr, 22, stock);*
*distribute(xyz, 22, nothing);*
*distribute(xyz, 22, stock);*
*distribute(abc, 11, both);*
*distribute(abc, 22, both);*
*distribute(abc, 22, stock);*
These results are *not in line with what actual happens* in the stock market as one conclusion indicates no benefit to xyz for event 22; whereas next conclusion points out stock benefit to the same client on the same event. When the multiple views from stakeholders of different stock market were assigned priorities (that can be easily modified or updated later on too), the results obtained were as per the expected benefits disbursed to the client in stock market abiding terms and conditions:
*overrides(divMkt1Mkt5,divMkt2)*
*overrides(divMkt1Mkt5,divMkt1)*
*overrides(divMkt1Mkt5,cash)*
*overrides(divMkt1Mkt5,stock)*
*overrides(divMkt1Mkt5,both)*
*and similar rules for rest of the markets including the generic ones:*
*overrides(both,stock);*
*overrides(both,cash);*

*overrides(stock, cash);*
*MUTEX*
  *distribute(?Client, ?EventId, ?Value1) AND*
  *distribute(?Client, ?EventId, ?Value2)*
*GIVEN*
  *notEquals( ?Value1, ?Value2 );*

Validating the facts gathered earlier against the set of labeled rules and the prioritized information, consistent and expected results were obtained as:
*distribute(abc, 22, stock);*
*distribute(pqr, 22, cash);*
*distribute(abc, 11, both);*
*distribute(xyz, 22, nothing);*

5. *Maintainability and Traceability* – As we've made our observation while discussing point 2 that courteous logic-based requirements representation is quite compact and preserves consistency (refer point 4), it can be said that this representation is maintainable in nature. We're further planning to refine the proposed representations to take care of traceability as well.
6. *Well understood by the involved parties* – While carrying out our case study, we worked in close collaboration with the analysts and the developers of the system. We had to check with SMEs on account of any gap or ambiguity and, then, we took our representation to developers. We found that none of the parties had much difficulty in making sense of these representations. Developers found these quite helpful as it was easier for them to relate observable behavior of the system to the expected behavior. This observation led us to believe that our approach should turn out to be practical, though we have to further test the representations for practicability.

## 11.7 Related work

The use of use of logic for requirements representation has been acknowledged earlier too and has found its place in several authors' work. RML [11] is one of the earliest logic-based formal requirements modeling language. HCLIE language [4] is a predicate logic-based requirement representation language. It makes use of Horn clause logic, augmented with multiple inheritances and exceptions. In case of exceptions and contradictions, HCLIE refrains from generating any conclusion. Description logic has also been used for representing requirements in [25]. Description logic is an expressive fragment of predicate logic and is good for capturing ontology, whereas Horn clause logic is useful for capturing rules. Description logic alone is not sufficient to express business requirements as explored in [26]. Ordinary logic programming (OLP) lacks in conflict-handling or non-monotonic reasoning. Business rules do present themselves with many instances of conflict handling along with constraints and related events. To capture business rules successfully, Horn clause logic needs to be augmented with non-monotonic (/commonsense) reasoning as has been discussed in [5]. Default logic has also been used for expressing requirements and resolving inconsistency in requirements in [20]. The computational complexity of default logic is quite high, and the default logic expressions are not simpler to comprehend by end users. We have presented courteous logic-based representations toward the cause

## 11.8 Discussion and Conclusion

This chapter has addressed the challenge to establish correct business requirements, i.e., free from inconsistency and incompleteness concerns. This is one of the important and crucial issues in RE as incorrect and inconsistent requirements are major contributors to project budget overruns and schedule slippage. Our approach has been to find a suitable representation of business requirements that can help us meet the challenge rather than finding some methodology to combat the challenge. Having presented the case studies and their observations, we can now go back and review our contributions. Our preliminary work has shown that courteous logic-based requirements representation meets the consolidated set of desirable features for a requirements specification language. We have presented scenarios from two case studies here in domains varying in complexities and number of involved processes. This reinforces our belief that courteous logic-based expressions of requirements are scalable too. We have been able to derive domain model from natural language sentences, and the model is verified during generation process by analyst's intervention. We have been able to demonstrate how this representation can prove helpful in ensuring consistency and resolving conflicts and incompleteness in the elicited requirements. We are now developing the framework based on courteous logic representation for formal analysis of requirements as part of our future work.

## References

1. IEEE Recommended Practice for Software Requirements Specifications, IEEE Std 830-1998, pp 1–40, 20 Oct 1998, doi: 10.1109/IEEESTD.1998.88286
2. Zowghi D, Gervasi V (2003) On the interplay between consistency, completeness, and correctness in requirements evolution. Inform Technol 45(14):993–1009
3. Zualkernan IA, Tsai WT (1988) Are knowledge representations the answer to requirements analysis. In: Proceedings of the IEEE conference on computer languages (ICCL 1988), pp 437–443. doi:10.1109/ICCL.1988.13094
4. Jeffrey J-PT, Weigert T (1991) HCLIE: a logic-based requirement language for new software engineering paradigms. Softw Eng 6(4):137–151
5. Borgida A, Greenspan S, Mylopoulos J (1985) Knowledge representation as the basis for requirements specifications. Computer 18(4):82–91. doi:10.1109/MC.1985.1662870
6. Tse TH, Pong L (1991) An examination of requirements specification languages. Comput J 34(2):doi:10.1093/com/jnl/34.2.143

7. Svoboda CP (1990) Tutorial on structured analysis. In: Thayer RH, Dorfman M (eds) System and software requirements engineering. IEEE Computer Press Society, Los Alamitos
8. Grady Booch Object-oriented analysis and design with applications, 2nd edn, Pearson Education
9. Weston N, Chitchyan R, Rashid A (2008) A formal approach to semantic composition of aspect-oriented requirements. In: Proceedings of the 16th IEEE international requirements engineering conference, Barcelona, Spain, pp 173–182
10. Zave P, Jackson M (1997) Four dark corners of requirements engineering. ACM Trans Softw Eng Methodol 6(1):1–30
11. Greenspan S, Borgida A, Mylopoulos J (1986) A requirements modeling language and its logic. Inform Syst 11(1):9–23
12. Mylopoulos J, Borgida A, Jarke M, Koubarakis M (1990) Telos: representing knowledge about information systems. ACM Trans Inform Syst 8(4):325–362
13. Dardenne A, van Lamsweerde A, Fickas S (1996) Goal directed requirements acquisition. Sci Comput Program 28(4):623–643
14. Yu E (1997) Towards modelling and reasoning support for early-phase requirements engineering. In: Proceedings of the 5th IEEE international requirements engineering conference, Maryland, US, pp 226–235
15. Nuseibeh B (1996) To be and not to be: on managing inconsistency in software development. In: Proceedings of the 8th IEEE international workshop on software specifications and design, Germany, pp 164–169
16. Easterbook S, Nuseibeh B (1995) Managing inconsistencies in an evolving specification. In: Proceedings of the 2nd international symposium on requirements engineering, UK, pp 48–55
17. Balzer R (1991) Tolerating inconsistency. In: Proceedings of the 13th international conference on software engineering, IEEE Computer Press, Texas, US, pp 158–165
18. Letier E, van Lamsweerde A (2002) Requirements analysis: deriving operational software specifications from system goals. In: Proceedings of the 10th ACM SIGSOFT symposium on foundations of software engineering. ACM Press, South Carolina, US, pp 119–128
19. Reiter R (1980) A logic for default reasoning. Artif Intell 13:81–132
20. Gervasi V, Zowghi D (2005) Reasoning about inconsistencies in natural language requirements. ACM Trans Softw Eng Methodol 14(3):277–330
21. Nute D (2001) Defeasible logic. In: Proceedings of the international conference on applications of prolog (INAP 2001), Tokyo, Japan, pp 87–114
22. Grosof, BN (1997) Courteous logic programs: prioritized conflict handling for rules. IBM research report RC20836, IBM Research Division, T.J. Watson Research Centre
23. Grosof BN (2004) Representing E-commerce rules via situated courteous logic programs in RuleML. Electron Comm Res Appl 3(1):2–20. doi:10.1016/j.elerap. 2003.09.005
24. http://nlp.stanford.edu/software/tagger.shtml
25. Zhang Y, Zhang W (2007) Description logic representation for requirement specification. In: Proceedings of the international conference on computational science (ICCS 2007), Part II, Springer-Verlag, Beijing, China, pp 1147–1154
26. Sharma R, Biswas KK (2011) Can ontologies be sufficient solution to requirements engineering problem? In: Proceedings of the international conference on knowledge engineering and ontology development (KEOD 2011)

# Chapter 12
# Automated Verification of Variability Model Using First-Order Logic

**A.O. Elfaki**

**Abstract** Verification of the domain engineering is motivated by two reasons: (1) the huge size of the software assets and (2) the possibility of changes in business rules or in stakeholders' needs which affect the structure of the domain engineering. To solve this problem of verifying software product line (SPL), we propose set of rules to verify four operations: inconsistency detection, inconsistency prevention, dead feature detection, and false-optional feature detection. Scalability is a key factor in measuring the applicability of the methods dealing with the domain engineering. We generated experiments for testing the scalability of our approach. Our experiments results show that our approach is scalable.

## 12.1 Introduction

Software product line consists of two processes: the domain engineering and the application engineering. Application engineering is responsible for configuring new software products. Domain engineering is responsible for collecting all the software assets (such as requirements, specification reports, functions, procedures, components, and test cases) in a specific business domain. In addition, data in domain engineering are classified and grouped in a way that allows the development of a specific software product based on the reuse of these software assets. The development of the domain engineering is a continuous process in which the adding or updating of software assets could happen at any time.

A.O. Elfaki (✉)
Management and Science University, Shah Alam, Malaysia
e-mail: abdelrahmanelfaki@gmail.com

In SPL, a successful software product is highly dependent on the validity of an SPL. Hence, verification is a significant process within SPL. Recently, verification of SPL has been discussed as an important issue, and research has focused on the maturity of SPL [1–5]. Mannion [6] defines verification in SPL as a mechanism that is used to ensure that an SPL can produce at least one product that can satisfy the constraint dependency rules. Lan et al. [7] define verification in SPL as a mechanism to check if the configuration output satisfies corresponding variability constraints (in a specific domain) or not. In this chapter, we define verification as a method used to ensure the correctness of assets in the domain engineering.

Usually, a medium-sized domain engineering contains thousands of software assets with constraint dependency rules among them. Therefore, validating domain engineering represents a challenge. The verification of the feature model (FM) (one of the accepted SPL modeling techniques) has already been identified as a critical task in Batory et al. [8], Massen and Litcher [9, 10], and Czarnecki and Eisenecker[11].

Domain engineering is a continuous process; when there are new assets, these are added to the existing assets. Cumulative aggregation for the software assets may produce some errors. The grouping of assets may be made at different times and by different groups of people. In some cases, there is a parallel development process, that is, several people add assets to develop domain engineering at the same time. The verification of SPL is a vital process, and it is not feasible that it be done manually. For all the above reasons, verification of domain engineering is a necessity. In this chapter, a set of rules is introduced for validating domain engineering. The proposed rules guide SPL engineers by detecting errors and anomalies in the domain-engineering process.

In Elfaki et al. [12], a new variability modeling technique is suggested as a prerequisite process for our approach. The proposed variability modeling technique is based on two layers. The first layer is a graphical representation, which consists of merging the feature model (FM) [13] with the orthogonal variability model (OVM) [14]. The lower layer is a mathematical representation, in which variability is modeled using first-order logic predicates. Modeling SPL completely using the proposed notations is the condition for implementing the verification rules.

Software requirements are the main and major component in the domain engineering. Modeling the domain engineering is the main concern of variability modeling technique. Therefore, this chapter introduces a novel approach to verify software requirements in a variability model.

In this chapter, four verification operations are discussed. These operations are as follows: inconsistency detection, inconsistency prevention, dead features detection, and false-optional features detection.

This chapter is organized as follows; introducing the verification operations is taken place in Sect. 12.2. Scalability results are presented in Sect. 12.3. In Sect. 12.4, we compare our work with the previous works. The chapter is concluded in Sect. 12.5.

## 12.2 Automated Verification of the Variability Using First-Order Logic Rules

### *12.2.1 The Operations*

In an SPL, software assets are also known as features. In the proposed approach, a feature has only two possibilities: variation point or variant.

**Definition.** Let fi denote a feature, VPi denote a variation point, and Vi denote a variant, where $i \in I+$.

Let us now examine the four verification operations in turn.

### *12.2.2 Inconsistency Detection*

Inconsistency detection is introduced in Batory et al. [8] as a research challenge. Inconsistency occurs because of contradictions in constraint dependency rules. Inconsistency is the existence of relations between features that cannot be true at the same time. For example, (A requires B) and (B excludes A), which means selection of A must be followed by selection of B, but selection of B prevents selection of A. Therefore, these relations cannot be true at the same time. An SPL can contain complicated forms of inconsistency such as (f1 requires f2) and (f2 requires f3) and (f3 requires f4) and (f4 excludes f1). Another example is VP1 excludes VP2 and VP1 require V2, where VP1 and VP2 are variation points and V2 is a variant belongs to VP2. Inconsistency is very complicated because it takes different forms. Inconsistency can occur between groups of features, individual features, or between a group features and an individual feature. In the following, we define three types of inconsistency, and logic rules (based on FOL) are developed to detect all the defined types of inconsistency.

#### 12.2.2.1 Forms of Inconsistency

We categorize inconsistency into three forms: direct inconsistency, indirect inconsistency, and inconsistency related to a common feature. In the following, these forms of inconsistency are discussed, and the rules that detect each form are illustrated.

1. Direct Inconsistency

In direct inconsistency, all features are of the same type: variation point or variant. The relation ($f_1$, $f_2$) requires ($f_3$, $f_4$, $f_5$) means the existence of both $f_1$ and $f_2$ requires the existence of $f_3$, $f_4$, and $f_5$ together. Direct inconsistency can be divided in four groups:

**Table 12.1** Rule 1 and its three implementations

| | |
|---|---|
| $\mathbf{F}_A \mathcal{R}\, \mathbf{F}_B \wedge \text{select } (\mathbf{F}_A) \Rightarrow \{f_i\, \mathcal{R}\, f_j \mid f_i \in \mathbf{F}_A \text{ and } f_j \in \mathbf{F}_B\}$ | (1) |
| $\lvert\mathbf{F}_A\rvert > 1 \wedge \lvert\mathbf{F}_B\rvert > 1$ : many-to-many | (i) |
| $\lvert\mathbf{F}_A\rvert = 1 \wedge \lvert\mathbf{F}_B\rvert > 1$ : one-to-many | (ii) |
| $\lvert\mathbf{F}_A\rvert > 1 \wedge \lvert\mathbf{F}_B\rvert = 1$ : many-to-one | (iii) |

*Many-to-Many Inconsistency*: Here, a set requires another set, while the required set excludes the first one, for example, (($f_1$, $f_2$, $f_3$) requires ($f_4$, $f_5$, $f_6$)) and (($f_4$, $f_5$, $f_6$) excludes ($f_1$, $f_2$, $f_3$)).

*Many-to-One Inconsistency*: A set of features has a constraint dependency relation (require/exclude) with one feature, while this feature has a contradictory relation to that set, for example, (($f_1$, $f_2$, $f_3$) requires $f_4$) and ($f_4$ excludes ($f_1$, $f_2$, $f_3$)).

*One-to-Many Inconsistency*: One feature has a constraint dependency relation (require/exclude) with a set of features, while this set has a contradictory relation to that feature, for example, ($f_4$ requires ($f_1$, $f_2$, $f_3$)) and (($f_1$, $f_2$, $f_3$) excludes $f_4$).

*One-to-One Inconsistency*: One feature has a constraint dependency with one feature, while the second feature has a contradictory relation to the first feature, for example, ($f_1$ requires $f_2$) and ($f_2$ excludes $f_1$).

Each type of direct inconsistency at the same time represents the type of constraint relation. There are four types of constraint relation: many-to-many, many-to-one, one-to-many, and one-to-one relations. All forms of constraint relation must be converted to a one-to-one relation as a prerequisite process for direct inconsistency detection. In order to implement the conversion of the other forms of constraint relation to a one-to-one relation, we use rule 1 which converts complex constraints to direct constraints:

Let $\mathbf{F}_A$ and $\mathbf{F}_B$ denote two sets of features where $f_i \in \mathbf{F}_A$, $f_j \in \mathbf{F}_B$, (i, j) $\in$ I+, $\mathbf{F}_A \cap \mathbf{F}_B = \emptyset$, and $\mathcal{R} \in \{\text{require, exclude}\}$.

Table 12.1 demonstrates rule 1 and its three implementations. By using the implementations of rule 1, all forms of the constraint relations will be converted to one-to-one constraint relations.

There are three cases ((i), (ii), and (iii)) for the two sets: $\mathbf{F}_A$ and $\mathbf{F}_B$. In case (i), the number of elements in set $\mathbf{F}_A$ is greater than 1, and the number of elements in $\mathbf{F}_B$ is also greater than 1. In case (ii), the number of elements in set $\mathbf{F}_A$ is equal to 1, and the number of elements in $\mathbf{F}_B$ is greater than 1. In case (iii), the number of elements in set $\mathbf{F}_A$ is greater than 1, and the number of elements in set $\mathbf{F}_B$ is equal to 1.

Case (i) converts many-to-many, case (ii) converts one-to-many, and case (iii) converts many-to-one inconsistency.

*Example 12.1* This example shows how the many-to-many constraint relation can be converted to a one-to-one constraint relation. Suppose that an SPL has this form of constraint: (($f_1$, $f_2$, $f_3$) requires ($f_4$, $f_5$)) and (($f_4$, $f_5$) excludes ($f_1$, $f_2$)). This constraint denotes that the existence of ($f_1$, $f_2$, $f_3$) requires the existence of ($f_4$, $f_5$) and the existence of ($f_4$, $f_5$) excludes the existence of ($f_1$, $f_2$). To convert this complex constraint to a one-to-one constraint relation, first, this constraint must

**Table 12.2** Many-to-many relation constraint and its equivalent one-to-one relation constraint

| | |
|---|---|
| (($f_1$, $f_2$, $f_3$) requires ($f_4$, $f_5$)) and (($f_4$, $f_5$) excludes ($f_1$, $f_2$)). | *many-to-many* |
| (($f_1$ requires $f_4$), ($f_1$ requires $f_5$), ($f_2$ requires $f_4$), ($f_2$ requires $f_5$), ($f_3$ requires $f_4$), and ($f_3$ requires $f_5$), ($f_4$ excludes $f_1$), ($f_4$ excludes $f_2$), ($f_5$ excludes $f_1$) and ($f_5$ excludes $f_2$)). | *one-to-one* |

**Table 12.3** Rules for detecting direct inconsistency

| Definitions | |
|---|---|
| type($VP_1$,variationpoint),type($VP_2$,variationpoint),type($V_1$,variant),and type($V_2$,variant) | |
| $\forall V_1, V_2$ : requires_v_v$(V_1, V_2) \wedge$ excludes_v_v$(V_2, V_1) \Rightarrow$ error | (2) |
| $\forall VP_1, VP_2$ : requires_vp_vp$(VP_1, VP_2) \wedge$ excludes_vp_vp$(VP_2, VP_1) \Rightarrow$ error | (3) |

be restructured to be suitable for applying rule 1. Rule 1 must be applied for each constraint relation, and this example has two constraint relations. Therefore, rule 1 must be applied to each one separately:

The first constraint is (($f_1$, $f_2$, $f_3$) requires ($f_4$, $f_5$)). Let $F_A$ represents ($f_1$, $f_2$, $f_3$) and $F_B$ represents ($f_4$, $f_5$). By searching for these predicates, select($f_1$), select($f_2$), and select($f_3$), the existence of all elements of the set $F_A$ in the configuration is checked. This searching implements the predicate select (**F**$_A$) from rule 1. If all the elements of $F_A$ are found, then the many-to-many constraint will be converted to a one-to-one constraint. The new constraint is (($f_1$ requires $f_4$), ($f_1$requires $f_5$), ($f_2$ requires $f_4$), ($f_2$ requires $f_5$), ($f_3$ requires $f_4$), and ($f_3$ requires $f_5$)).

The second constraint is (($f_4$, $f_5$) excludes ($f_1$, $f_2$)). Let $F_A$ represents ($f_4$, $f_5$) and $F_B$ represents (f1, f2). By searching for these predicates, select ($f_4$), and select ($f_5$), the existence of all elements of the set $F_A$ in the configuration is checked. If all the elements of $F_A$ are found, then the many-to-many constraint will be converted to a one-to-one constraint. The new constraint is ($f_4$ excludes $f_1$), ($f_4$ excludes $f_2$), ($f_5$ excludes $f_1$), and ($f_5$ excludes $f_2$).

Table 12.2 shows the many-to-many relation constraint (from Example 1) and its equivalent one-to-one relation constraint. The many-to-many inconsistency (case (i)) is converted to a one-to-one inconsistency (case (ii)).

After converting all constraint relations to one-to-one relations, the rules in Table 12.3 are used for detecting direct inconsistency. In Table 12.3, rule 2 is used for detecting inconsistency between two variants, whereas rule 3 is used for detecting inconsistency between two variation points.

There is no representation for the variation point-to-variant relation, because it could be divided into variant-to-variant relations. As an example, imagine an SPL has the following inconsistency: variation point VP requires variant V, and variant V excludes variation point VP. The relation variant point VP requires variant V cannot be implemented in our notations. Therefore, all features in the direct inconsistency detection process are limited to being of the same type: variation point or variant.

Rules 2 and 3 trigger an error message that is sent to the user announcing an error in the relations. The conversion of complex constraint relations to one-to-one relations happens during configuration time. Hence, rules 2 and 3 detect all types of direct inconsistency during configuration time. Rules 2 and 3 can also be used to detect direct inconsistency in domain engineering. In this case, rules 2 and 3 detect direct inconsistency in domain engineering between two variants or two variation points only.

2. Indirect Inconsistency

We describe indirect inconsistency in general as the relationship between two variation points and a contradictory relation (at the same time) between a variant belonging to the first variation point and the second variation point or one of its variants. The general patterns of indirect inconsistency are as follows:

2. $VP_1$ requires $VP_2$, and $V_1$ excludes $V_2$ where $V_1$ and $V_2$ are common.
3. $VP_1$ requires $VP_2$, and the common variant $V_1$ excludes $VP_2$.
4. $VP_1$ excludes $VP_2$, and $V_1$ requires $V_2$.
5. $VP_1$ excludes $VP_2$, and $V_1$ requires $VP_2$.

where type($V_1$,variant), type($V_2$,variant), type($VP_1$,variationpoint), type($VP_2$, variationpoint), variants($VP_1$,$V_1$), and variants($VP_2$,$V_2$).

Table 12.4 shows the rules that are used for detecting indirect inconsistency. Rules 4, 5, 6, and 7 detect indirect inconsistency denoted by the general pattern numbers 1, 2, 3, and 4, respectively.

The rules in Table 12.4 can be applied directly to domain engineering for detecting indirect inconsistency. Remember that domain engineering has already been modeled as a knowledge base [12]. The output of these rules is an error message which is sent to the user. In addition, these rules can be used to define the source of inconsistency. For instance, rule 4 detects the indirect inconsistency that is denoted by the general pattern number 1. Hence, modifying the output from rule 4 to be, for example, error_4 can lead to the detection of the incorrect relations that have caused the indirect inconsistency.

3. Inconsistency Related to a Common Feature

In general, we define inconsistency related to a common feature as *a feature excludes a common feature*. This inconsistency has two types: (1) full inconsistency and (2) conditional inconsistency.

1. *Full Inconsistency*: The general pattern of full inconsistency is *feature $f_1$ is common and feature $f_2$ is common, and $f_1$ excludes $f_2$.*

Using our proposed approach for modeling an SPL, there are three possibilities for the exclusion relation: (1) variant excludes variant, (2) variant excludes variation point, and (3) variation point excludes variation point. Moreover, there are two possibilities for a feature to be common: common variation point and common variant belong to common variation point. Hence, the common possibilities must be applied into both sides of the exclude relation to implement the general pattern of full inconsistency (a common feature excludes another common feature). As an example, in the exclude relation, “variant excludes variation point,” the variant must be common variant belongs to common variation point and the variation point must be common. Table 12.5 shows the rules that are used for detecting full inconsistency.

**Table 12.4** Rules for detecting indirect inconsistency

| Definitions | |
|---|---|
| type($VP_1$,variationpoint), type($VP_2$,variationpoint),type($V_1$,variant), type($V_2$,variant), varints($VP_1$,$V_1$),and variants($VP_2$,$V_2$) | |
| $\forall VP_1, VP_2, V_1, V_2$ : requires_vp_vp$(VP_1, VP_2) \wedge$ common$(V_1,$ yes$) \wedge$ common $(V_2,$ yes$) \wedge$ excludes_v_v$(V_1, V_2) \Rightarrow$ error | (4) |
| $\forall VP_1, VP_2, V_1$ : requires_vp_vp$(VP_1, VP_2) \wedge$ common$(V_1,$ yes$)\wedge$ excludes_v_vp$(V_1, VP_2) \Rightarrow$ error | (5) |
| $\forall VP_1, VP_2, V_1$ : excludes_vp_vp$(VP_1, VP_2) \wedge$ requires_v_vp$(V_1, VP_2) \Rightarrow$ error | (6) |
| $\forall VP_1, VP_2, V_1, V_2$ : excludes_vp_vp$(VP_1, VP_2) \wedge$ requires_v_v$(V_1, V_2) \Rightarrow$ error, | (7) |

**Table 12.5** Rules for detecting the full inconsistency

| Definitions | |
|---|---|
| type($VP_1$,variationpoint),type($VP_2$,variationpoint),type($V_1$,variant),type($V_2$,variant) ,variants($VP_1$,$V_1$), and variants($VP_2$,$V_2$) | |
| $\forall VP_1, VP_2 : \text{excludes\_vp\_vp}(VP_1, VP_2) \wedge \text{common}(VP_1, \text{yes}) \wedge \text{common}(VP_2, \text{yes}) \Rightarrow \text{error}$ | (8) |
| $\forall VP_1, VP_2, V_1 : \text{common}(VP_1, \text{yes}) \wedge \text{common}(V_1, \text{yes}) \wedge \text{common}(VP_2, \text{yes}) \wedge \text{excludes\_v\_vp}(V_1, VP_2) \Rightarrow \text{error}$ | (9) |
| $\forall VP_1, VP_2, V_1, V_2 : \text{common}(VP_1, \text{yes}) \wedge \text{common}(V_1, \text{yes}) \wedge \text{common}(VP_2, \text{yes}) \wedge \wedge \text{common}(VP_2, \text{yes}) \wedge \text{excludes\_v\_v}(V_1, V_2) \Rightarrow \text{error}$ | (10) |

In Table 12.5, rule 8 illustrates a common variation point excludes another common variation point, rule 9 illustrates a common variant belongs to a common variation point excludes common variation point, and rule 10 illustrates a common variant belongs to a common variation point excludes another common variant that belongs to another common variation point.

2. Conditional Inconsistency

The general pattern of conditional inconsistency is *feature $f_1$ is not common and feature $f_2$ is common, and $f_1$ excludes $f_2$*. The only difference between this type and full inconsistency is that in conditional inconsistency, the first feature is not common. This type of inconsistency happens only if the first feature is selected. Table 12.6 shows the rules that are used for detecting conditional inconsistency.

Rule 10 illustrates a not common variation point excludes a common variation point, rule 12 illustrates a common variant belongs to a not common variation point excludes common variation point, and rule 13 illustrates a common variant belongs to a not common variation point excludes another common variant that belongs to another common variation point. The rules in Tables 12.3, 12.4, 12.5, and 12.6 could be applied directly to check inconsistency in domain engineering. As mentioned earlier, these rules can also be used to find the source of inconsistency, that is, the incorrect constraint relations.

### *12.2.3 Inconsistency Prevention*

In this operation, new constraint dependency rules are developed to avoid or prevent direct inconsistency. The general patterns of inconsistency prevention are as follows:

1. ($f_1$ require $f_2$) and ($f_2$ require $f_3$) $\Rightarrow$ $f_1$ require $f_3$
2. ($f_1$ require $f_2$) and ($f_2$ exclude $f_3$) $\Rightarrow$ $f_1$ exclude $f_3$

*Example 12.2* This example is provided in order to clarify the advantage of general pattern number 1. In general pattern number 1, if feature $f_1$ requires feature $f_2$ and $f_2$ requires feature $f_3$, then the new constraint rule ($f_1$ requires $f_3$) will be added to the domain-engineering process. This new rule solves the following inconsistency: imagine that an SPL also has this relation – $f_3$ excludes $f_1$. Before applying our rules, the constraint relations in this SPL are $f_1$ requires $f_2$, $f_2$ requires $f_3$, and $f_3$ excludes $f_1$. It is very difficult to detect this inconsistency. After adding the new rule ($f_1$ requires $f_3$), the constraint relations in this SPL become $f_1$ requires $f_2$, $f_2$ requires $f_3$, $f_3$ excludes $f_1$, and $f_1$ requires $f_3$. The inconsistency now becomes a direct inconsistency between features: $f_1$ and $f_3$, and it could be detected using rule 2 or rule 3. If the feature type is a variant, then rule 1 will detect the direct inconsistency. If the feature type is a variation point, then rule 2 will detect the direct inconsistency.

**Table 12.6** Rules for detecting conditional inconsistency

| Definitions | |
|---|---|
| type($VP_1$,variationpoint),type($VP_2$,variationpoint),type($V_1$,variant), type($V_2$,variant) ,variants($VP_1$,$V_1$), and variants($VP_2$,$V_2$) | |
| $\forall VP_1, VP_2 : \text{excludes\_VP\_VP}(VP_1, VP_2) \wedge \text{common}(VP_1, \text{no})\ \ \text{common}(VP_2, \text{yes}) \Rightarrow \text{error}$ | (11) |
| $\forall VP_1, VP_2, V_1 : \text{common}(V_1, \text{yes}) \wedge \text{common}(VP_2, \text{yes}) \wedge \text{common}(VP_1, \text{no}) \wedge \text{excludes\_v\_vp}(V_1, VP_2) \Rightarrow \text{error}$ | (12) |
| $\forall VP_1, VP_2, V_1, V_2 : \text{common}(VP_1, \text{no}) \wedge \text{common}(V_1, \text{yes}) \wedge \text{common}(VP_2, \text{yes}) \wedge \text{common}(V_2, \text{yes}) \wedge \text{excludes\_v\_v}(V_1, V_2) \Rightarrow \text{error}$ | (13) |

**Table 12.7** All possibilities for patterns 1 and 2

| | $f_1$ | $f_2$ | $f_3$ | Applicability |
|---|---|---|---|---|
| 1 | V | V | V | Yes |
| 2 | VP | VP | VP | Yes |
| 3 | V | VP | VP | Yes |
| 4 | VP | V | VP | No |
| 5 | VP | VP | V | No |
| 6 | VP | V | V | No |
| 7 | V | VP | V | No |
| 8 | V | V | VP | Yes |

*Example 12.3* This example is provided to clarify the advantage of general pattern number 2. In general pattern number 2, if the feature f1 requires the feature $f_2$ and $f_2$ excludes $f_3$, then the new constraint rule ($f_1$ excludes $f_3$) will be added to the domain-engineering process. The new rule is added to avoid direct inconsistency. As an example, suppose an SPL also has this constraint relation: $f_3$ requires $f_1$. Before applying our rules, the constraint relations in this SPL are $f_1$ requires $f_2$, $f_2$ excludes $f_3$, and $f_3$ requires $f_1$. This inconsistency is not easy to detect. After adding the new constraint rule, the constraint relations become $f_1$ requires $f_2$, $f_2$ excludes $f_3$, $f_3$ requires $f_1$, and $f_1$ excludes $f_3$. Rules 1 and 2 could then be used to detect the following direct inconsistency: ($f_3$ requires $f_1$, and $f_1$ excludes $f_3$).

Table 12.7 explains all the possibilities of general patterns 1 and 2. The term "V" represents variant, and the term "VP" represents variation point; the word "yes" denotes that this possibility can be implemented, and the word "no" denotes that this possibility cannot be implemented. The relation (variation point-variant) has no place in our proposed approach. Therefore, general patterns 1 and 2 could be implemented using only four possibilities (1, 2, 3, 8).

Applying our notations, the first general pattern becomes:

$V_1$ requires $V_2$ and $V_2$ requires $V_3 \Rightarrow V_1$ req $V_3$
$VP_1$ requires $VP_2$ and $VP_2$ requires $VP_3 \Rightarrow VP_1$ req $V_3$
$V_1$ requires $VP_1$ and $VP_1$ requires $VP_2 \Rightarrow V_1$ req $VP_2$
$V_1$ requires $V_2$ and $V_2$ requires VP1$\Rightarrow V_1$ req $VP_1$

Applying our notations, the second general pattern becomes:

$V_1$ requires $V_2$ and $V_2$ excludes $V_3 \Rightarrow V_1$ exc $V_3$
$VP_1$ requires $VP_2$ and $VP_2$ excludes $VP_3 \Rightarrow$ VP1 exc $VP_3$
$V_1$ requires $VP_1$ and $VP_1$ excludes $VP_2 \Rightarrow V_1$ exc $VP_2$
$V_1$ requires $V_2$ and $V_2$ excludes $VP_1 \Rightarrow V_1$ exc $VP_1$

Table 12.8 shows the rules that are used for implementing the first pattern, and Table 12.9 shows the rules that are used for implementing the second pattern in the inconsistency prevention operation.

The rules in Tables 12.8 and 12.9 are applied directly to the domain-engineering process to prevent direct inconsistency.

**Table 12.8** Implementation of the first pattern in inconsistency prevention

| Definitions | |
|---|---|
| type($VP_1$,variationpoint),type($VP_2$,variationpoint),type($VP_3$,variationpoint),type($V_1$,variant), type($V_2$,variant), and type($V_3$,variant) | |
| $\forall\, V_1, V_2, V_3$ : requires_v_v$(V_1, V_2) \wedge$ requires_v_v$(V_2, V_3) \Rightarrow$ requires_v_v$(V_1, V_3)$ | (14) |
| $\forall VP_1, VP_2, VP_3$ : requires_vp_vp$(VP_1, VP_2) \wedge$ requires_vp_vp$(VP_2, VP_3) \Rightarrow$ requires_vp_vp$(VP_1, VP_3)$ | (15) |
| $\forall V_1, VP_1, VP_2$ : requires_v_vp$(V_1, VP_1) \wedge$ requires_vp_vp$(VP_1, VP_2) \Rightarrow$ requires_v_vp$(V_1, VP_2)$ | (16) |
| $\forall V_1, V_2, VP_1$ : requires_v_v$(V_1, V_2) \wedge$ requires_v_vp$(V_2, VP_1) \Rightarrow$ requires_v_vp$(V_1, VP_1)$ | (17) |

**Table 12.9** Implementation of the second pattern in inconsistency prevention

| Definitions | |
|---|---|
| type($VP_1$,variationpoint),type($VP_2$,variationpoint),type($VP_3$,variationpoint), type($V_1$,variant), type($V_2$,variant), and type($V_3$,variant) | |
| $\forall V_1, V_2, V_3$ : requires_v_v($V_1, V_2$) $\wedge$ excludes_v_v($V_2, V_3$) $\Rightarrow$ excludes_v_v($V_1, V_3$) | (18) |
| $\forall VP_1, VP_2, VP_3$ : requires_vp_vp($VP_1, VP_2$) $\wedge$ excludes_vp_vp($VP_2, VP_3$) $\Rightarrow$ excludes_vp_vp($VP_1, VP_3$) | (19) |
| $\forall V_1, V_2, VP_1$ : requires_v_v($V_1, V_2$) $\wedge$ excludes_v_vp($V_2, VP_1$) $\Rightarrow$ exludes_v_vp($V_1, VP_1$) | (20) |
| $\forall V_1, VP_1, VP_2$ : require_v_vp($V_1, VP_1$) $\wedge$ excludes_vp_vp($VP_1, VP_2$) $\Rightarrow$ excludes_v_vp($V_1, VP_2$) | (21) |

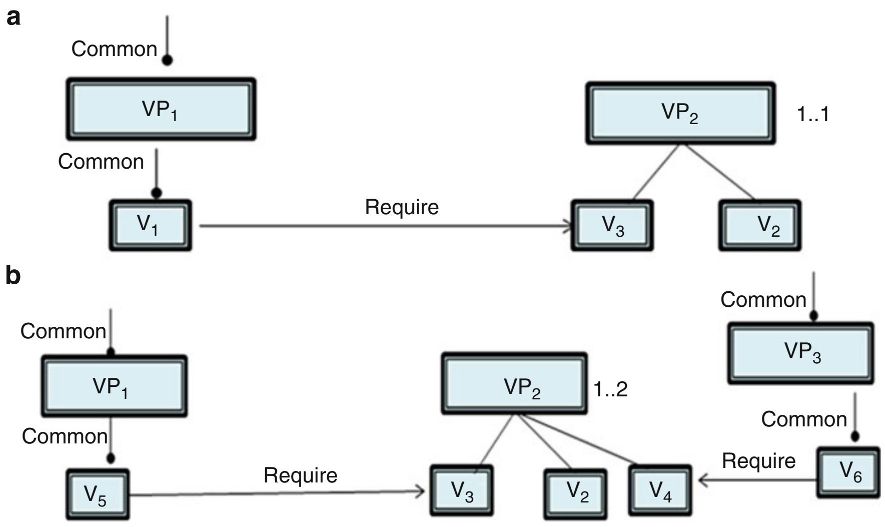


**Fig. 12.1** Examples of dead features caused by indirect exclusion

### *12.2.4 Dead Feature Detection*

A dead feature is defined in Czarnecki and Kim [15] as a frequent case of error in the FM. A dead feature is a feature that never appears in any legal product of an SPL. The only reason to prevent a feature from being included in any product is that there is a common feature that excludes this feature. As previously stated, in our proposed notations, the term "feature" has only two possibilities: variation point or variant. Therefore, this operation detects dead variants or dead variation points.

We define the general pattern for describing a dead feature as *feature $f_1$ excludes feature $f_2$, and feature $f_1$ is common, and then feature $f_2$ is a dead feature*. This exclusion could be indirect or direct.

1. Dead Feature Caused by Indirect Exclusion

In indirect exclusion, a common feature requires another feature, and the required feature is the only choice that is allowed to be selected from its variation point. From this definition, it is clear that the type of required feature must be a variant because it belongs to a variation point. In this case, where the required feature is a variant, the requiring feature must be also a variant because there is no variation point-to-variant relation. Now, the definition of indirect exclusion becomes a common variant requires another variant which is the only variant that is allowed to be selected from its variation point. The other variants that share the variation point with the required variant are dead variants because they cannot be included in any product.

Figure 12.1 shows examples of indirect exclusion. In Fig. 12.1 a, there are three facts: (1) the variant V$_1$ is a common variant belonging to a common variation point VP$_1$, which means V$_1$ must be included in all products; (2) the maximum number of variants allowed to be selected from $VP_2$ is 1; and (3) the common variant $V_1$ requires the variant $V_3$ which belongs to the variation point $VP_2$. Hence, $V_3$ will be selected in all products, whereas variant $V_2$ will never be included in any product. Therefore, variant $V_2$ is a dead variant caused by indirect exclusion. In Fig. 12.1 b, the maximum number of variants allowed to be selected from $VP_2$ is 2. The variants $V_3$ and $V_4$ are required by common variants $V_5$ and $V_6$ which belong to common variation points $VP_1$ and $VP_3$. Therefore, $V_3$ and $V_4$ must be included in any product. Again, variant $V_2$ is a dead variant caused by indirect exclusion.

Table 12.10 shows the rule that is used for detecting a dead variant caused by indirect exclusion. Rule 22 states that if there is a variation point ($VP_2$) and a number of its variants are required by common variants that belong to common variation points and this number is equal to the maximum number allowed to be selected from this variation point ($VP_2$), then the rest of variants of the variation point ($VP_2$) are dead variants.

2. Dead Feature Caused by Direct Exclusion

This type of dead feature is caused by direct exclusion between a common feature and a dead feature. There are three possibilities for the exclusion relations: variant excludes another variant, variant excludes variation point, and variation point excludes another variation point. Table 12.11 shows the rules that are used for detecting dead variants caused by direct exclusion. Rule 23 states that if the common variation point $VP_1$ excludes the variation point $VP_2$, then $VP_2$ is a dead variation point. All variants belonging to a dead variation point are dead variants. Therefore, $V_2$ in rule 23 is a dead variant. In rule 24, a common variant ($V_1$) belonging to a common variation point ($VP_1$) excludes another variants ($V_2$). Hence, $V_2$ is a dead variant. In rule 25, if a common variant belongs to a common variation point that excludes a variation point ($VP_2$), then this variation point is a dead variation point. All variants belonging to a dead variation point are dead variants.

### 12.2.5 False-Optional Feature Detection

A feature is false optional if it is included in all the products of an SPL despite not being modeled as a common feature [16]. The general pattern to describe false-optional feature detection is *feature $f_1$ is common, and requires feature $f_2$, and feature $f_2$ is not common.*

As mentioned earlier, there are three implementations for the require relation: variant requires another variant, variant requires variation point, and variation point requires another variation point. Table 12.12 shows the rules for detecting the false-optional variants and false-optional variation points. Rule 26 shows that if a common variant which belongs to common variation requires a non-common variant, then the required variant is a false-optional variant. Rules 27 and 28 detect false-optional variation points. In this case, it is clear that all the common variants belonging to the

**Table 12.10** Rule for detecting dead features caused by indirect exclusion

| Definitions | |
|---|---|
| ($V_1$,variant),($V_2$,variant),($V_3$,variant),type($VP_1$,varitionpoint),type($VP_2$,variationpoint), variants($VP_1$,$V_1$), variants($VP_2$,$V_2$) and variants($VP_2$,$V_3$) | |
| $\forall V1, V2, VP1, VP2 : \mathrm{common}(VP1, yes) \wedge \mathrm{common}(V1, yes) \wedge (\mathrm{requires\_v\_v}(V1, V2) = \max(VP2, n)) \Rightarrow \mathrm{dead\_variant}(V3)$ | (22) |

Table 12.11 Rules for detecting dead variants caused by direct exclusion

| Definitions | |
|---|---|
| type(VP$_1$,variationpoint),type(VP$_2$,variationpoint),type(V$_1$,variant), type(V$_2$,variant), variants(VP$_1$,V$_1$), and variants(VP$_2$,V$_2$) | |
| $\forall \mathrm{VP_1, VP_2, V_2}$ : common(VP$_1$, yes) $\wedge$ excludes_vp_vp(VP$_1$, VP$_2$) $\wedge$ variants(VP$_2$, V$_2$) $\Rightarrow$ dead_variant(V$_2$) | (23) |
| $\forall \mathrm{VP_1, V_1, V_2}$ : common(VP$_1$, yes) $\wedge$ common(V$_1$, yes) $\wedge$ excludes_v_v(V$_1$, V$_2$) $\Rightarrow$ dead_variant(V$_2$) | (24) |
| $\forall \mathrm{VP_1, VP_2, V_1, V_2}$ : common(VP$_1$, yes) $\wedge$ common(V$_1$, yes) $\wedge$ excludes_v_vp(V$_1$, VP$_2$) $\wedge$ variants(VP$_2$, V$_2$) $\Rightarrow$ dead_variant(V$_2$) | (25) |

Table 12.12 Rules for detecting false-optional features

| Definitions | |
|---|---|
| type(V$_1$,variant),type(V$_2$,variant),type(VP$_1$,variationpoint),type(VP$_2$,variationpoint) variants (VP$_1$,V$_1$), variants(VP$_2$,V$_2$), common(V$_1$,yes), and common(VP$_1$,yes) | |
| $\forall \mathrm{VP_2, V_1, V_2}$ : requires_v_v(V$_1$, V$_2$) $\wedge$ common(V$_2$, no) $\Rightarrow$ false_option(V$_2$) | (26) |
| $\forall \mathrm{VP_2, VP_1}$ : requires_vp_vp(VP$_1$, VP$_2$) $\wedge$ common(VP$_2$, no) $\Rightarrow$ false_option(VP$_2$) | (27) |
| $\forall \mathrm{VP_2, V_1}$ : requires_v_vp(V$_1$, VP$_2$) $\wedge$ common(VP$_2$, no) $\Rightarrow$ false_option(VP$_2$) | (28) |
| $\forall \mathrm{VP_2, V_2}$ : false_option(VP$_2$) $\wedge$ common(V$_2$, no) $\Rightarrow$ false_option(V$_2$) | (29) |

false-optional variation point are false-optional variants. Rule 29 explains how to detect the common variants belonging to the false-optional variation point.

In this chapter, we have discussed four problems and explained how they can be detected and handled by the FOL rules. These problems can be divided in two groups: severe and light issues. Inconsistency is a severe problem. Dead feature and false-optional are light problems. The problems classified under severe issues are critical problems and must be handled to ensure generation of valid products. Although valid software products with light issue problems can be generated, handling and removing these problems ensures the maturity and increases the maintainability of the domain-engineering process.

Each operation could be applied alone. For example, if we want to detect only inconsistency in domain engineering, rules 1 to 13 could be used to find the inconsistency. In addition, all these rules could also be applied together. There is no direct relation between these rules, that is, each rule is completely independent.

## 12.3 Scalability Testing

Usually, a medium-sized SPL has a huge amount of software assets, and there are complex dependency relations between these assets. Therefore, scalability is a key factor in measuring the applicability of the modeling techniques that deal with variability in an SPL [17, 18].

In general, a system is considered scalable for a specific problem if it can solve that problem in a reasonable time. The scalability test is an important criterion for testing performance because even a small SPL will grow in size over time.

In order to test the scalability of our approach, we implement our proposed verification operations in SPLs of different sizes. Our experiments start with domain engineering containing 1,000 assets and end with domain engineering containing 20,000 assets. Compared with the ranges reported in the literature, to the best of our knowledge, our range is the biggest tested. The principal aim of this section is to demonstrate that our approach could be applicable for industrial SPLs.

### *12.3.1 Experiment: The Method*

SPL data are considered to be business secrets; therefore, many researchers evaluate their work using synthetic data. There are many methods that are used empirical results to test scalability by generating artificial data, such as Trinidad et al. [19] and White et al. [20]. According to these methods, the output time is considered a key measurement for scalability experiments. In this chapter, we follow the same concept and test the scalability of our approach using artificial data.

In the following, the method of our experiments is described:

*Define the Assumptions*: We have one assumption: each variation point and variant has a unique name.

*Generate the Domain Engineering as a Dataset:* Domain engineering is generated in terms of predicates (variation points and variants). We generated four sets containing 1,000, 5,000, 15,000, and 20,000 variants. Variants are defined as numbers represented in sequential order, for example, in the first set (1,000 variants), the variants are 1, 2, 3,. . ., 1,000, and in the last set (20,000 variants), the variants are 1, 2, 3,. . ., 20,000. The number of variation points in each set is equal to the number of variants divided by five, which means each variation point has five variants. As an example, in the first set (5,000 variants), the number of variation points equals 1,000. Each variation point is defined as a sequence number having the term VP as a prefix, for example, VP12.

*Set the Parameters*: The main parameters that are used in our experiments are the number of variants and the number of variation points. The remaining parameters are the following: common variants, common variation points, variant requires variant, variant excludes variant, variation point requires variation point, variation point excludes variation points, variant requires variation point, and variant excludes variation point. These eight parameters are defined randomly as a percentage of the number of variants or variation points. Three ratios are defined: 10 %, 25 %, and 50 %. The number of parameters related to the variant, such as common variant, variant requires variant, variant excludes variant, variant requires variation point, and variant excludes variation point, is defined as a percentage of the number of variants. The number of parameters

**Table 12.13** Snapshot of the experimental dataset

| |
|---|
| type(vp1,variationpoint) |
| type(1,variant) |
| variants(vp1,1) |
| common(570,yes) |
| Common(vp123,yes) |
| requires_v_v(7552,2517) |
| requires_vp_vp(vp1572,vp1011) |
| excludes_vp_vp(vp759,vp134) |
| excludes_v_v(219,2740) |
| requires_v_vp(3067,vp46) |
| excludes_v_vp(5654,vp1673) |

related to the variation point, such as variation point requires variation point, is defined as a percentage of the number of variation points. As an example, in the dataset that contains 1,000 variants (note that in the dataset that contains 1,000 variants, there are 200 variation points) and in the ratio 10 %, the number of each parameter related to the variant is 100 and number of each parameter related to the variation point is 20, that is, 100 variants require 100 variants, 100 variants exclude 100 variants, 20 variants require 20 variation points, 20 variants exclude 20 variation points, 20 variation points require 20 variation points, 20 variation points exclude 20 variation points, 100 common variants, and 20 common variation points. The variants and variation points that are included in these constraint relations are selected randomly. The variants are selected randomly from 1 to 1,000, and the variation points are selected randomly from 1 to 200. Table 12.13 represents snapshots of the experimental dataset, that is, the domain engineering used in our experiments.

*Calculate Output*: For each set, we conducted 30 experiments and calculated the execution time as an average. The experiments were performed for the range of 1,000–20,000 variants and the percentage ratio of 10 %, 25 %, and **50 %.**

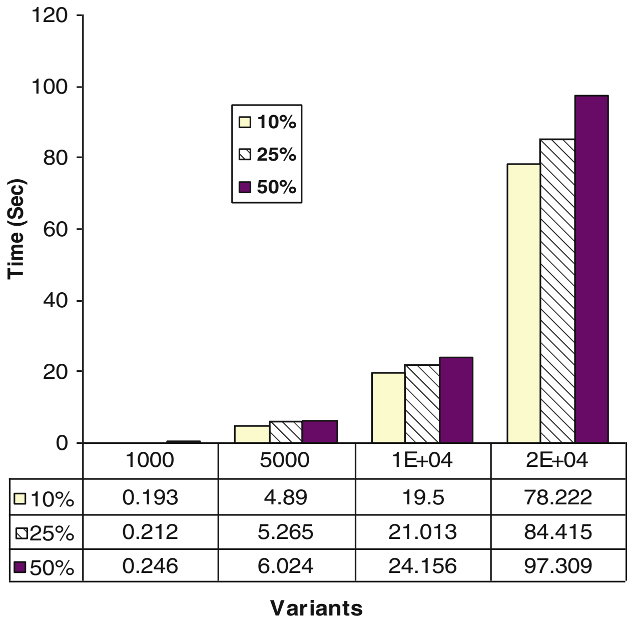
## 12.3.2 Scalability Result of Inconsistency Detection Experiments

Inconsistency in the SPL is a relationship between features that cannot be true at the same time. In Sect. 12.2, we have defined three types of inconsistency: direct, indirect, and inconsistency between common feature. The rules of detecting inconsistency are combined in one program to calculate the scalability of inconsistency detection. Table 12.14 shows a snapshot of the inconsistency detection program. The outputs of this program are messages that explain the type of inconsistency and variants or variation points that create the inconsistency.

Figure 12.2 illustrates the average execution time to detect inconsistency in an SPL with a range of 1,000–20,000 variants. The other parameters are defined as three ratios: 10 %, 25 %, and 50 %.

**Table 12.14** Snapshot of inconsistency detection program

```
direct_inconsistecy_rule4.6
type(X,variant), type(Y,variant), requires_v_v(X,Y), excludes_v_v(Y,X)
write('direct_inconsistecy_rule4.6'),nl,
write('inconsistency between ....'),write(X),write('. . .and....'),write(Y),
nl, fail, main
direct_inconsistecy_case_one:- true
```



| | 1000 | 5000 | 1E+04 | 2E+04 |
|---|---|---|---|---|
| □ 10% | 0.193 | 4.89 | 19.5 | 78.222 |
| ▨ 25% | 0.212 | 5.265 | 21.013 | 84.415 |
| ■ 50% | 0.246 | 6.024 | 24.156 | 97.309 |

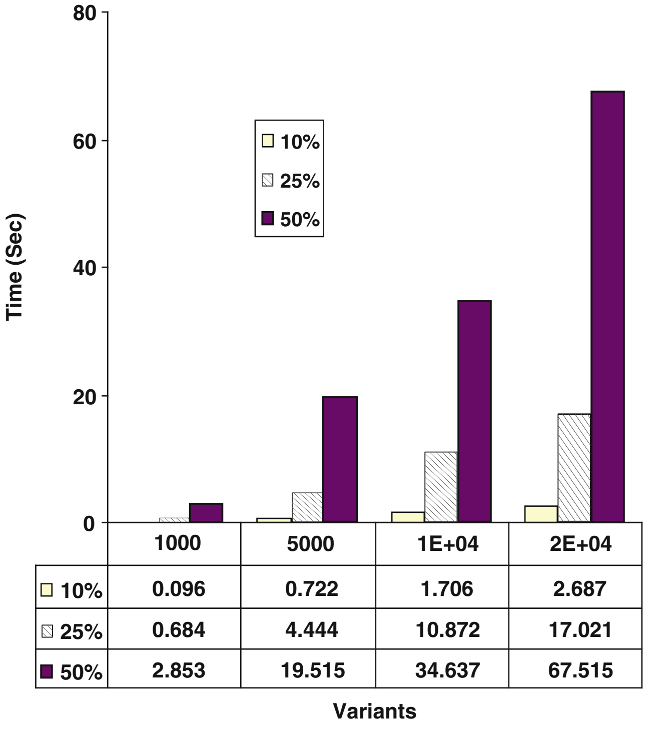
**Fig. 12.2** Inconsistency detection: scalability results

### *12.3.3 Scalability Result of Inconsistency Prevention Experiments*

According to the specific constraint dependency rules that are defined in Sect. 12.3, new dependency rules (require/exclude) should be added to the domain engineering to prevent inconsistency. The rules in Tables 12.8 and 12.9 are combined together in one program to prevent inconsistency. The output of this program is new constraint rules which update the domain engineering directly. Figure 12.3 illustrates the average execution time to prevent inconsistency in an SPL with a range of 1,000–20,000 variants.

### *12.3.4 Scalability Result of Dead Feature Detection Experiments*

A dead feature is a feature that never appears in any legal product of an SPL. The rules of detecting dead features are combined in one program to detect dead



| | 1000 | 5000 | 1E+04 | 2E+04 |
|---|---|---|---|---|
| □ 10% | 0.096 | 0.722 | 1.706 | 2.687 |
| ▨ 25% | 0.684 | 4.444 | 10.872 | 17.021 |
| ■ 50% | 2.853 | 19.515 | 34.637 | 67.515 |

**Fig. 12.3** Inconsistency prevention: scalability results

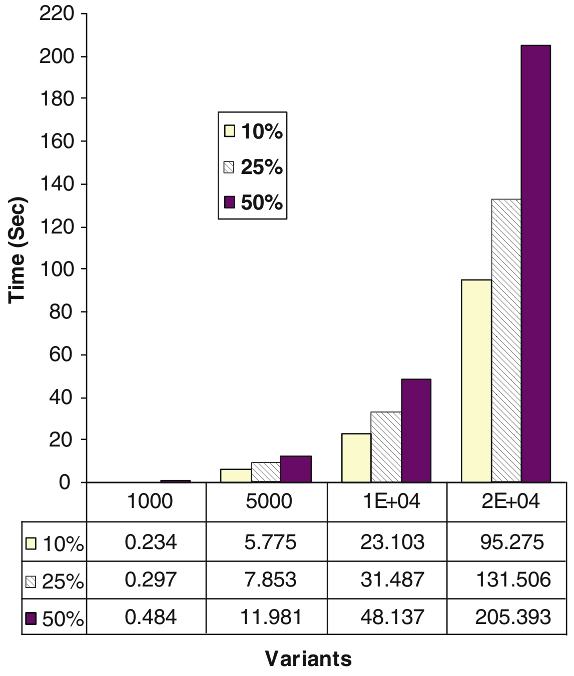
features. This program searches for dead features within the domain engineering. Figure 12.4 illustrates the average execution time. The output of each experiment is a result file containing the dead variants.

### *12.3.5 Scalability Result of False-Optional Feature Detection Experiments*

Figure 12.5 illustrates the average execution time. The output of each experiment is a result file containing the false-optional features.

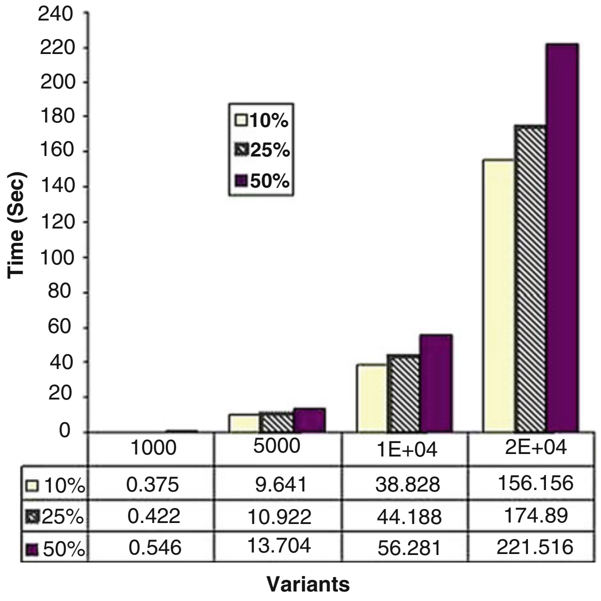
## 12.4 Contributions of the Research and Comparison with Previous Works

Our methodology defines a general pattern for each verification operation in order to handle verification of an SPL. These definitions of the general patterns formulate the problems and allow the FOL rules to deduce the results from predefined cases.



| | 1000 | 5000 | 1E+04 | 2E+04 |
|---|---|---|---|---|
| □ 10% | 0.234 | 5.775 | 23.103 | 95.275 |
| ▨ 25% | 0.297 | 7.853 | 31.487 | 131.506 |
| ■ 50% | 0.484 | 11.981 | 48.137 | 205.393 |

**Fig. 12.4** Dead feature detection: scalability results



| | 1000 | 5000 | 1E+04 | 2E+04 |
|---|---|---|---|---|
| □ 10% | 0.375 | 9.641 | 38.828 | 156.156 |
| ▨ 25% | 0.422 | 10.922 | 44.188 | 174.89 |
| ■ 50% | 0.546 | 13.704 | 56.281 | 221.516 |

**Fig. 12.5** False-optional feature detection: scalability results

In the following, we discuss each operation and highlight our contributions this area of research:

*Inconsistency Detection*: In this chapter, we analyze the inconsistency problem and define three types of inconsistency. First-order logic rules are developed in order to detect inconsistency in the domain-engineering process. The definition of the three types of inconsistency and the detection of inconsistency in the domain-engineering process are our contributions to the literature in respect of this operation. In the literature, only direct inconsistency is detected at the configuration stage [20, 21].

*Inconsistency Prevention*: To the best of our knowledge, this is the first time this operation has been implemented with the aim of preventing direct inconsistency in the domain-engineering process. This is another contribution to this field of research.

*Dead Features Detection*: Trinidad et al. [22, 23] developed a method to detect dead features based on finding all products and searching for the not used features. Trinidad and Ruiz-Cortes [24] used abductive reasoning to explain the dead features using the same concept of Trinidad et al. [22, 23]. Finding all products from the SPL is very tough operation, and it is not feasible in big SPL [16]. Thus, Trinidad et al.'s [22, 23] method is not a practical solution. Our approach detects dead features by searching only for predefined cases, which means defining dead features in the domain-engineering process without implementing the configuration process. Our approach needs less cost.

*False-Optional Features Detection*: In the literature, detecting false-optional features is based on finding all products and searching for common features that are not assigned as common [22]. As it mentioned before, finding all products is not practical solution. Using our approach, false-optional features could be detected in the domain-engineering process based on predefined cases. Thurimella and Janzen [25] are verifying the configuration process and the constraint relation between the features during the configuration (selecting features of specific product). Our work is focusing to verify the variability model in the domain engineering.

## 12.5 Conclusion

The benefits of using an SPL have been proved in both the academic and industrial domains. Although a huge amount of research work in the area of the SPL has already been undertaken, the automated analysis of the SPL is still a "hot" area of research [4]. In this research, we introduce an approach for the automated verification of domain-engineering process using FOL rules.

The problems that are discussed in this research could be found in any SPL regardless of the technique used for modeling variability. Inconsistency, dead features, and false-optional features could occur in an SPL due to the wrong usage of constraint dependency rules. Although these problems are very clear in both the FM and the OVM, it still could occur in all types of variability modeling techniques. Since any SPL has a group of features (by which we mean software assets) that are collected in the domain-engineering process, wrong usage of the dependency rules leads to these problems.

In respect of the scalability results of the domain engineering, we conducted experiments for SPLs with ranges of up to 20,000 variants and up to 50 % of constraint dependency rules, and we were able to obtain results in a good time. White et al. [20] scaled their work by 5,000 features in 1 min. In Segura [26], the execution time for 200–300 features is 20 min after applying atomic sets to enhance the scalability. When compared to the literature, it can be seen that our proposed method is scalable. The scalability of our approach is good enough when compared with the literature because we first define special patterns and later the system searches only for these patterns. As a consequence, the searching time is acceptable.

Our approach is limited to work only in a certain environment, that is, where constraint dependency rules are well known in all cases. In some SPL, constraint dependency rules are different from product to product. We called these types of SPL as uncertain SPL environments. For future work, our approach could be extended to handle uncertain SPL environments using case-based reasoning. In domain engineering, our approach is used to detect inconsistency, dead features, and false-optional features. As a future extension of this work, some new rules could be developed for auto-correction of these errors.

## References

1. Benavides D, Ruiz-Cortés A, Batory D, Heymans P (2008) Opining introduction. In: First international workshop on analyses of software product lines (ASPL'08), Limerick
2. Benavides D, Metzger A, Eisenecker U (2009) Opining introduction. In: Third international workshop on variability modelling of software-intensive systems, ICB-research report No. 29, University of Duisburg Essen, Duisburg
3. Benavides D, Batory D, Grünbacher P (2010) Opining introduction. In: Fourth international workshop on variability modelling of software-intensive systems, ICB-research report No. 37, University of Duisburg Essen, Duisburg
4. Eisenecke U, Apel S, Gnesi S (2012) Opining introduction. In: Sixth international workshop on variability modelling of software-intensive systems, ACM, Leipzig
5. Heymans P, Czarnecki K, Eisenecker U (2011) Opining introduction. In: Fifth international workshop on variability modelling of software-intensive systems, Namur
6. Mannion M (2002) Using first-order logic for product line model validation. In: the second software product line conference SPLC2, San Diego
7. Lan Q, Liu S, Li B, ChenY, Pang S, Yin J (2006) Research on variability metamodeling method. In: The first international symposium on pervasive computing and applications (SPCA06), Urumchi
8. Batory D, Benavides D, Ruiz-Cortés A (2006) Automated analysis of feature models: challenges ahead. Commun ACM 49(12): 45–47
9. Massen T von der, Litcher H (2004) Deficiencies in feature models, workshop on software variability management for product derivation- towards tool support, collocated with SPLC 2004, Boston
10. Massen T von der, Litcher H (2005) Determining the variation degree of feature models. In: Software product lines conference, Lecturer notes in computer science, vol 3714. Rennes, pp 82–88

11. Czarnecki K, Eisenecker U (2002) Generative programming: methods, tools, and applications. Addison-Wesley, Boston
12. Elfaki A, Phon-Amnuaisuk S, Ho CK (2009) Modeling variability in software product line using first order logic. In: Proceedings of 7th the international conference on software engineering research, management and applications (SERA2009), Haikou
13. Kang K, Cohen S, Hess J, Novak W, Peterson S (1990) Feature oriented domain analysis (FODA) feasibility study, Technical report no. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh
14. Pohl K, Böckle G, van der Linden F (2005) Software product line engineering foundations principles and techniques. Springer, Heidelberg
15. Czarnecki K, Kim C (2005) Cardinality-based feature modeling and constraints: a progress report. In: Proceedings of the international workshop on software factories at OOPSLA05, San Diego California
16. Benavides D, Segura S, Ruiz-Cort´es A (2010) Automated analysis of feature models 20 years later: a literature review. Inform Syst j 35(6), Elsevier: 615–636
17. Kasikci BC, Bilgen S (2009) Scalable modeling of software product line variability. In: 13th international software product line conference, SPLC 2009, San Francisco
18. Wang H, Li YF, Sun J, Zhang H, Pan J (2007) Verifying feature models using OWL. J Web Semantics 5(2):117–129, Elsevier
19. Trinidad P, Benavides D, Dur´an A,Ruiz-Cort´es A, Toro M (2008) Automated error analysis for the agilization of feature modeling. J Syst Softw 81(6): 883–896
20. White J, Dougherty B, Schmidt D, Benavides D (2009) Automated reasoning for multi-step feature model configuration problems. In: 13th international software product line conference table of contents San Francisco, California, pp 11–20
21. Hemakumar A (2008) Finding contradictions in feature models. In: First international workshop on analyses of software product lines (ASPL'08)', collocated with SPLC08. Limerick
22. Trinidad P, Benavides D, Ruiz-Cort´es A (2006) Isolated features detection in feature models. In: Advanced information systems engineering, 18th international conference, CAiSE2006, short paper proceedings, Luxembourg
23. Trinidad P, Benavides D, Ruiz-Cort´es A (2006) Explanations for agile feature modeling. In the first workshop on agile product line engineering (APLE'06), Baltimore
24. Trinidad P, Ruiz Cort´es A (2009) Abductive reasoning and automated analysis of feature models: how are they connected? In: Third international workshop on variability modelling of software-intensive systems. Proceedings, pp 145–153
25. Thurimella AK, Janzen D (2011) on the metadoc feature modeler. In: SPLC2011 proceedings
26. Segura S (2008) Automated analysis of feature models using atomic sets. In: Proceedings of first international workshop on analyses of software product lines (ASPL'08)', collocated with SPLC08, Limerick

# Chapter 13
# Model-Based Requirements Engineering Framework for Systems Life-Cycle Support

**A. Soffer and D. Dori**

**Abstract** The recent migration from traditional sequential development process models to the more modern iterative and evolutionary process models has brought about an evolution in the scope of the requirements engineering process, along with new challenges of managing the requirements knowledge. In parallel, conceptual modeling throughout the development process has been receiving growing attention and wide acceptance.

Working under the premise that effective requirements knowledge management is a key factor in developing quality software that meets customer needs, the main contribution introduced in this chapter is creation and study of a new requirements engineering and management (REM) framework that is tightly coupled with the evolving conceptual model of the developed system. The integration of the proposed REM process into an Object-Process Methodology (OPM)-based systems development and modeling environment is demonstrated via a case study, followed by an evaluation.

The work presented in this chapter shows that coupling the requirements knowledge management activities with the development methodology and a tool-supported modeling environment creates a comprehensive approach for the production of high-quality software.

## 13.1 Introduction

Close examination of the typical outcome of software-intensive systems development projects introduces the challenge of narrowing the gap between the required system and the resulting implementation. In spite of efforts to improve the flow of

A. Soffer (✉)
ORT-Braude College of Engineering, Karmiel, Israel
e-mail: asoffer@braude.ac.il

D. Dori
Technion, Israel Institute of Technology, Haifa, Israel
e-mail: dori@ie.technion.ac.il

engineering information throughout the development process, often times the implemented system does not fully match the required one, nor does it meet the customer's needs and expectations [1]. In this setting, an effective requirements engineering and management process should be used throughout the development life cycle to handle new and evolving requirements. System requirements are expected to reflect stakeholder needs by describing the system's externally perceived functionality as well as certain properties at the desired granularity. Although the field of requirements engineering provides means for managing this need based on establishing and maintaining traces between the requirements and their implementation, the effectiveness of these means remains a significant factor in quality and productivity of the software development process.

Requirements engineering (RE) is the branch of software engineering that focuses on the processes of handling and management of requirements in any software development effort [2]. Regardless of the applied life-cycle development model or the software engineering methodology, RE is a critical component of any comprehensive engineering process employed for systems development or maintenance. Numerous conceivable scenarios (e.g., the typical process of change management) demonstrate that proper management of the requirements knowledge plays a pivotal role in system evolution, as it is embedded within the core of the system's development process throughout its life cycle and may take place at any point during this process.

The main contribution introduced in this chapter is creating, analyzing, and evaluating a life-cycle requirements engineering and management (REM) environment that is tightly coupled with the conceptual model of the evolving system. The proposed RE process is integrated into the Object-Process Methodology (OPM) system development process. This new REM environment supports requirements documentation, tracing, testing, and management and enables reasoning about requirementsduring the entire system's life span. Analysis and evaluation of this approach and the developed process automation show that coupling the requirements knowledge management activities with the software development methodology and a tool-supported modeling environment creates a comprehensive approach for the production of high-quality software.

To present this comprehensive framework, the necessary concepts are explored in this chapter as follows: In Sect. 13.2 some requirements engineering challenges are presented as motivation for this work. In Sect. 13.3, the concept of life-cycle REM process is introduced, along with the value of integrating it with the model-driven approach to systems development. Section 13.4 includes an introduction to OPM and argues for the choice of OPCAT (Object-Process CASE Tool) as the supporting environment for REM implementation. Focusing on implementation of these ideas, Sect. 13.5 describes an extension to OPM's capability which enables linking textual requirements to the system's models. It then demonstrates how the requirements engineering activities are integrated into the OPM-based conceptual system model which evolves throughout the system its life cycle. An evaluation of this work is also presented in this section. Section 13.6 concludes the chapter with some observations on the utility of this mechanism.

## 13.2 Background: Requirements Engineering Challenges

Requirements should be complete, consistent, comprehensible, unambiguous, well documented, traceable, and testable [2]. Proper communication of requirements knowledge to all the software developers in the project and across their organization helps to ensure that the requirements, as well as changes in them, are handled correctly throughout the project life cycle and that all stakeholders maintain a shared vision [3–5].As requirements evolve during the development or maintenance phases, it becomes necessary to modify and manage the system specification and design. Although the field of requirements engineering provides means for managing this need based on establishing and maintaining traces between the requirements and their implementation, this need remains a significant challenge in productivity of the software process.

The term "software product quality" is used as a qualifier of the extent to which the implementation satisfies stated and implied needs. The quality of a software product is directly linked to the characteristics of the process that was used to create it [3, 4]. Consequently, enhancing the development process and methodology is expected to result in higher product quality, which would manifest itself in a better match between the envisioned system, as expressed in the requirements specification, and its realization at the end of the development process.

Requirements management (RM) is the activity concerned with the effective control of information related to system requirements, in particular the preservation of the integrity of that information with respect to changes in the system and its environment [2]. The recent trend of migration from traditional sequential software development models to evolutionary process models has brought about the evolution of the scope of requirements management. In traditional development approaches, requirements were formulated and documented (often by the acquirer without the involvement of the developer) prior to any development activities. This set of textually expressed requirements was frozen prior to its validation for the entire duration of the system development effort. Often, the requirements list is a contractually binding document, serving as a "technical appendix" to the legal engagement between the system acquirer and provider, so deviating from it poses problems of various kinds. This may still be the case in formal, very-large-scale projects, where organizations such as governments are involved. In most modern systems R&D environments, however, where RE is at the heart of the development process, such requirements freezing is inconceivable. Focused on change management, the evolving requirements set serves as the glue that ties the other engineering processes together over time, ensuring that the resulting system or software-intensive product satisfies the needs and expectations of the users, the customers, and the beneficiaries.

A functional requirement is a specific need or desired behavior as seen by an external user of the system. The required capability or function must be delivered by a system through one or more of its components. Achieving high-level stakeholders' satisfaction strongly depends on maintaining the system's life-cycle conceptual integrity, that is, faithfully reflecting the stakeholders' views and needs throughout the system life cycle. Model-driven development (MDD) [6], which has received growing attention and wide acceptance in recent years, is an adequate response to this need. High-quality requirements should give information on what a software-intensive system is and what it does, rather than how to implement the system [7]. This means that the requirements ("what") need to be linked to the implementation model ("how") in order to facilitate the adequate connection between them.

The requirements phase is a critical part of software development, yet difficult to enhance [8]. Brooks attests on the difficulty of establishing the requirements and their importance throughout the entire development process: "the hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work so cripples the resulting system if done wrong" [9].

Since requirements engineering in general and the traceability function in particular are quality-enabling techniques [10], improvements in RE capability are essential for developing higher-quality software systems. A major expectation of RE is the quality (i.e., effectiveness, efficiency) of the requirements traceability, which is aimed at improving error detection and correction, leading to a better environment for the production of high-quality software [11]. Typically, systems are tested to verify the satisfaction of all of the requirements that were deemed feasible. This way, quality is guaranteed with respect to compliance with the requirements.

In summary, the objectives of an integrated requirements engineering and management process include the continuous management of new and changed requirements in a way that ensures competitiveness in the marketplace. Successful implementation of RE depends primarily on (1) controlling requirements and changes in them, (2) managing requirement attributes for better decision making, (3) establishing and maintaining traceability throughout the life span of the project, and (4) ensuring the availability of flexible frameworks for the necessary communication channels among stakeholders. The integration of requirements engineering processes into the model-driven development paradigm (MDD) has motivated this work as a promising approach for supporting these needs.

## 13.3 Model-Based Requirements Engineering and Management

Combining requirements engineering processes and artifacts with model-driven development approach may offer effective solutions to some of the critical difficulties in software development. In this section, we first review some basic terms relevant to this approach and challenges they present. Then we describe the integration of RE into the system model-based development and discuss how it improves many factors that affect the system's quality, notably change management.

### *13.3.1 Life-Cycle Requirements Management*

The objectives of requirements management activities include collecting, documenting and organizing the requirements, linking requirements to software items, tracing requirements to all development artifacts, and tracking and communicating this information to all stakeholders. This is necessary to ensure that the basic requirements and their evolution are properly handled throughout the project life cycle. Some process models such as the SEI's CMMI [4] associate some of the requirements-related activities also to other process areas, such as product engineering, project management, and configuration management.

The requirements document is the basic source for communication among customers, end users, system designers, and implementers of the software. It also serves as a validation device for the stated requirements. In many circumstances, it is used as the basis for user's manuals or other documents. Another important role that this document plays is serving as the basis for project planning and project management activities.

### *13.3.2 Model-Based Development*

Model-driven development (MDD) technologies [12] have been recognized by the software engineering community as potential vehicles to alleviate the cognitive effort required for creating a software system that matches its requirements [13, 14]. In this context, a model is a domain-specific abstraction which defines in appropriate terms the structure and behavior of the modeled concepts. According to the MDD paradigm, the requirements model (specification), which is problem domain oriented, evolves into another representation of the contemplated system in a new and different context – the implementation, which belongs to the solution domain. In the solution domain, the approach to implementing the required system is described by means of system architecture, design, and code [15].

Requirements modeling is the process of constructing abstract, formal representation of the initial textually described system requirements in a way that is amenable to unambiguous interpretation. This process ends with a requirements model, which is expected to capture as much of the relevant real-world semantics as possible. Based on the need to clearly express and communicate the engineering knowledge (particularly pertaining to requirements), the conceptual model of the system has become an essential and effective means of communication among all stakeholders of the system's development process. Not less importantly, the conceptual model is used for the system validation and documentation needs.

### 13.3.3 Model-Based Requirements Process

Creation of the requirements model is the first stage in model-driven development. The model aims to capture the real-world semantics pertaining to the system [16]. The requirements representation and documentation serve interests of the different system's stakeholders: customers, developers, and managers [17]. The requirements model focuses primarily on functional requirements, that is, specific business needs or behaviors as seen by an external user of the system. The required capability or function must be delivered by some subset of the system, so the functional requirements set captured in this model is a basis for subsequent system development and later on possibly also for its maintenance activities.

In the subsequent phases of the development process, the requirements model is elaborated and transformed into the design model. This transition emphasizes the critical need for creating a formal, accurate, and complete requirement model from the outset, as it designed serve as the foundation of the entire software developing process.

Modeling is targeted at clear and accurate representation of the concepts that comprise the system. An important benefit of requirements modeling is that since the resulting model is available at an early stage in the system's life cycle, model analysis and simulation may be used to validate the requirements and reduce conceptual design errors. Later on, as requirements modeling is integrated into the flow of activities in the development process, it fosters collaboration between requirements acquirers, system engineers, and developers.

A domain model is an abstraction which defines generically the structure and behavior of the problem domain. Two types of concepts are involved in life-cycle requirements modeling: (1) concepts related to customers' objectives, which represent the problem domain and are emphasized in the requirements model, and (2) engineering concepts, which are emphasized in the design model(s). A conceptual gap, which is often very wide, exists between these two model types, since one faces the acquirer with her problem domain, while the other faces the solution domain and the technological solution provider. Inevitably, the problems created due to this dissonance cast dark shadows over the remainder of the systems development process. To attenuate the adverse impact of switching from requirements to design, it is desirable to minimize the additional shake caused by the need not just to switch the attention from the problem to the solution but to also switch between a modeling approach or diagram type used in the requirements engineering phase to the one(s) used in the design phase. In this regard, if one uses UML, for example, the problem of switching from use case diagrams, applied at the requirements phase, to object-class diagrams and the rest of the UML diagram types exacerbates the difficulties involved in this transition.

Our goal is therefore to adopt a single modeling approach, language, and methodology, which can be used across this chasm and help bridge it. Such a language must have the traits of simplicity, clarity, and dynamic aspects that use cases possess, combined with formality, the object orientation of object-class

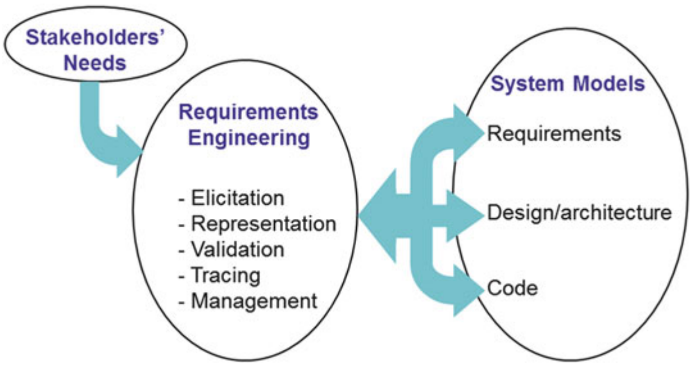


**Fig. 13.1** Model-integrated requirements engineering

diagrams. In addition, it should have the state transition aspects of state charts and the workflow and time sequence of activity, collaboration, and time sequence diagrams in a single modeling framework.

The use of domain knowledge can significantly reduce the requirements engineering effort and the amount and severity of errors [16]. Using domain ontologies that include abstractions of typical structure and behavior as modeling patterns facilitates reuse and error reduction. Here too, the ability to apply the same modeling language and diagram type for domain knowledge modeling across the gap between requirements engineering and design can significantly reduce the cognitive load that impedes comprehension of the two sides of this canyon.

## *13.3.4 Integrating the Requirements Process*

The requirements engineering processes constitute a fundamental life-cycle chain of engineering activities, which start early on with the establishment of positive relationships among success-critical stakeholders, and continue throughout the life span of the system. Modeling the requirements and integrating the requirements model into the system's life cycle enable extending the benefits of a solid REM process from the early stage to all the remaining stages of the development process, as depicted in Fig. 13.1.

A natural way of connecting requirements to a model is to represent the RE information and activities according to the adopted product and process models [18]. System model integration adds value to the knowledge captured in the RE process by enabling validation and better understanding of the causal relations underlying traces.

A good requirements specification is one in which requirements are arranged hierarchically. Few high-level, broadly defined requirements are specified in increasing levels of detail, where each level contains a set of requirements that elaborate upon one or more requirements at the level above it. Current commercial tools indeed support such structure for requirements management. A hierarchical structure of requirements may also facilitate the process of their modeling. In general, high-level requirements correspond to abstract processes and aggregate objects or agents (actors) and interactions between them at lower levels.

How are requirements integrated into a model? Each requirement is a specification of an individual system function. Thus, a model component designed to meet this requirement is created if it does not yet exist in the model, and it is linked to other model elements to express its relation to the rest of the model. This component should also be associated with the corresponding requirement (or requirements set) with which it is related, creating the traceability information. This traceability information is embedded into the model and is built on the fly during the modeling process, eliminating the need for an external tracing mechanism, such as a traceability matrix. This approach is becoming particularly valuable as traditional phase-oriented (waterfall-type) system development processes are being replaced by concurrent and iterative approaches, such as the spiral process and RUP [19]. In such nonlinear development processes, it is difficult to maintain accurate and reliable traceability information over time due to their concurrent and repetitive nature.

Model-integrated REM helps bring down the communication barriers between the various stakeholders involved, including system engineers, analysts, developers, and customers. Such barriers, misunderstandings, and potential contradictions typically exist in requirements that are based on text documents. Linking the requirements to the system model facilitates forward and backward traceability to and from the model (possibly including its test plan and model) to the requirements set and vice versa. This two-way traceability enables impact assessment of a proposed change in a requirement using the design model.

The requirements engineering and management (REM) process relates to the information captured in the specification and implementation models through requirements traceability checks, validation, error detection and analysis, and change management. Since the required capability or function must be delivered by the system to be, the information captured in the implementation model is used later for driving system development or maintenance activities.

A complete and effective model-integrated REM process includes the following steps:

1. Elicitation, acquisition, and text-based documentation of domain-dependent knowledge that pertains to the requirements from the system to be developed
2. Modeling: formal, semantics-conveying graphic representation of the contents of the text-based requirements documentation elicited in (1)
3. Mapping of the text-based requirements description to the conceptual requirements-level system model (the problem domain model), to the system's architectural model (the solution domain model), and to the system implementation in software and/or hardware

### *13.3.5 Model-Integrated Requirements Traceability*

Requirements traceability (RT) is the ability to describe and follow information about the life of a requirement in both forward and backward directions [20]. RT is aimed at improving error detection and correction, leading to a better environment for the production of high-quality software [11]. A major activity in the REM process, RT helps stakeholders understand the relationships between software artifacts created during the software development life cycle. RT is fundamental to change management, as it enables one to assess the impact of a change in some part of the system on the rest of the system by analyzing dependencies between the associated requirements and the system design and implementation.

The scope of RT is evolving as the migration from traditional sequential software development models to evolutionary development approach is taking place. Consequently, a number of issues should be considered: when should the trace be created, what is the detail level (granularity) of traces, how to relate traces to the hierarchical structure of requirements and to designed system artifacts, and how to report traces.

#### 13.3.5.1 Requirements Traceability Challenges

Requirements traceability (RT) is a facilitator of system quality, as it provides means to keep track of the relationships between stakeholders' requirements and artifacts produced during the software development life cycle. Despite being introduced as a mandatory activity in development standards [4], RT is highlighted as an area in need of improvement [16, 21].

Management of change is an essential part in the development and maintenance of systems in general and software-intensive systems in particular. RT is fundamental to the management of change, and its scope is evolving due to the migration from traditional sequential development models to evolutionary ones. This transition raises a number of issues, including the timing of creating each trace, the granularity (detail level) of a trace, and how to relate a trace to the hierarchical structure of the requirements on one hand and to the system conceptual model on the other hand.

Studies that examined the nature of the relations between requirements, artifacts, and people [20] propose that a more detailed semantic relation than statements such as "X contributed to Y" is needed. Research based on empirical investigations with practitioners and extensive surveys of techniques and tools for requirements engineering [8, 20] suggests that documenting and preserving information generated during early stages of a project is of crucial importance. This is so because this information is potentially relevant to later stages of the system under development, but it cannot be reconstructed at those stages.

A significant practical problem underlying RT is the difficulty to maintain the trace information up-to-date as development artifacts change over time [22]. The challenge here is to provide a low-effort method to continuously update the traceability picture. Performing the REM activities efficiently and effectively cannot be achieved without adequate tool support for automating tedious and error-prone REM tasks. The desired properties of such tools and methods of using them are discussed next.

### *13.3.6 Tool Support for Model-Based Requirements Engineering*

Years of attempts to achieve progress in RE have focused primarily on establishing procedures and developing practice-oriented methodologies [23] and standards, such as CMMI [4]. However, the high expectations for quality gains cannot be completely met by using methodologies and tools that address the needs of the requirements engineering phase alone. These specialized tools focus mainly on traceability between requirements and code, that is, relating sections of the systems code to the relevant requirements [20, 24]. However, the amount of knowledge captured by means of traceability can be used in many more ways to improve the gamut of software engineering activities, including requirements validation, change management, complexity and cost measurement and estimation, and test management. Unfortunately, most RE tools fall short of exploiting this full potential of the knowledge represented by traceability, since their methods of operation do not reflect the complexity of the software development scenarios and process [6].

Requirements are the first artifact in the development process, while code is the last in line, the most obscure and least accessible to most stakeholders. To achieve effective traceability, we need to fill the gap between requirements and code. Relating requirements to precode artifacts, such as conceptual model elements, test plans, and test outcomes is far more effective than connecting requirements directly to the code.

While moving from high to lower levels of abstraction in system modeling, the complexity of the associated traceability structure grows. The challenge then is to maintain the accuracy and effectiveness of the trace information. Model-based REM is therefore most suitable for RT, as it can exploit the potential for automation through proper tools [25]. Many automation solutions employ an array of tools to support a variety of functions, creating the challenge of consolidation and unification of these tools.

The proposed approach suggests embedding the requirements engineering and management process as an integral part of the system engineering process based on Object-Process Methodology (OPM) through the use of OPCAT – Object-Process CASE Tool [26]. Spanning across the entire system life cycle and supported by automation, REM, including RT, is thus fully integrated into the modeling and development methodology.

## 13.4 The Choice of OPM

This section includes an introduction to Object-Process Methodology (OPM) and analysis of principles to establish the theoretical foundations for considering OPM as a basis for implementing the required solution.

An effective requirements engineering process that yields a successful system and high-level stakeholders' satisfaction should be supported by a model that clearly and accurately expresses the system's requirements. The requirements model is a critical artifact in the development life cycle, since it serves as the basis for the system's architecture, detailed design, testing, and change management. Following the premise that effective modeling of complex systems should combine structure and behavior representations in the same model [27], OPCAT is a comprehensive system modeling, engineering, and life-cycle support environment, based on Object-Process Methodology (OPM) [28, 29].

OPM integrates the object-oriented and process-oriented paradigms into a single frame of reference. OPM's core entities are objects, representing things that exist, and processes, which are things that transform objects. A process transforms an object by creating it, consuming it, or changing its state. Processes are connected to the involved objects through procedural links, which describe the function and behavior of a system. In addition, a set of structural relations provides for modeling the system's structure. The premise of the OPM paradigm is that objects and processes are two types of equally significant entities. Contrary to the object-oriented approach, processes in OPM are not necessarily properties of objects or operations owned by objects but can rather stand alone. A set of interrelated Object-Process Diagrams (OPDs) constitutes the graphical OPM formalism. The same set of symbols is used in all the Object-Process Diagrams, which is the only diagram type. This way, both the static (structure) and dynamic (function, behavior) aspects that each system exhibits coexist as integrated and equivalent components in the same OPM model.

One of the most progressive ideas of OPM is the approach that promotes integration of the various system life-cycle perspectives into a unified continuous model that may reflect the complete system's life cycle. The OPM philosophy is that the same system model that started in the requirements stage as an analysis model of the requirements is continuously developed, evolved, and refined to represent the system all the way to implementation and deployment.

### *13.4.1 Bimodal Representation*

An OPM model is jointly expressed by two semantically equivalent modalities, one graphic and the other textual. A set of interrelated Object-Process Diagrams (OPDs) constitute the graphical, visual OPM formalism. The graphical OPD set has a textual counterpart modality named Object-Process Language (OPL). OPL is a dual-purpose language, oriented towards humans as well as machines. Catering to human needs, OPL is designed as a constrained subset of English, which serves domain experts and system architects engaged in analyzing and designing a system. Every OPD construct is expressed by a semantically equivalent OPL sentence or phrase. Designed also for machine interpretation through a well-defined set of production rules, OPL has an XML-based notation that provides a solid basis for further automated processing, such as information extraction or code generation.

### *13.4.2 Complexity Management*

OPM mitigates the inherent complexity of model-based system development by supporting gradual refinement/abstraction of information and smooth traversal of a hierarchy of diagrams. This capability is enabled by a multidimensional complexity management mechanism. The three built-in refinement/abstraction mechanisms are as follows: (1) unfolding/folding, which is used for refining/abstracting the structural hierarchy of a thing and is applied by default to objects; (2) in-zooming/out-zooming, which exposes/hides the inner details of a thing within its frame and is applied primarily to processes; and (3) state expressing/suppressing, which exposes/hides the states of an object. Flexible combinations of these three mechanisms enable consistent system modeling at different levels of detail without losing the overall comprehension of the representation.

The unified, bimodal, and flexible representation of a system increases the user's comprehension and processing capability, which is further enhanced through automation by CASE Tool support.

### *13.4.3 OPM Tool Support*

OPM facilitates clear and concise graphical formalism along with semantically equivalent and formally structured text. Both representations are automatically kept consistent at all times by OPCAT – the OPM CASE Tool. The unified, bimodal representation of an OPM system makes use of the same methodology-supported language, which could be universal or domain-specific, throughout the system life cycle. In addition, OPM facilitates continuous life-cycle modeling; thus, life-cycle requirements engineering can be naturally and easily achieved through OPM-based system development.

OPCAT can provide a number of functions and features to support various activities in the RE process, such as domain modeling, requirements modeling and analysis, and traceability. Based on unified modeling, combining OPCAT's modeling capabilities and the RE process needs will bring better results for system development challenges such as effective alignment of stakeholders' requirements with system evolution, and enhanced traceability and process measurements (complexity, cost, and quality).

### *13.4.4 A Comparison to UML-Based Systems*

The ability to trace the requirements throughout the entire life cycle is a critical part of the requirements management process [30]. Many tools enable linking requirements to a variety of life-cycle artifacts, such as source code files, change requests, test case descriptions, or project tasks. However, the significant advantage of model-driven development is the ability to link requirements to the system model for various needs, such as verification and certification, impact analysis, or test case design. Tools designed for UML environments provide the capability to link requirements to use cases and to activity diagrams. However, they do not support fully integrated model traceability, since none of them have provided the ability to trace a requirement to the internals of the design model, notably to a specific class. Furthermore, since a UML representation of a particular system includes a collection of models, an attempt to achieve full requirements traceability in UML will require a very complex association mechanism in order to link the requirements to the various components included in the representation.

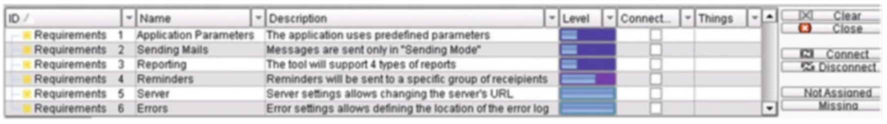
In contrast, the unique ability of OPCAT/REM, described below, to trace requirements to elements of the design model is highly valuable. OPM facilitates clear and concise formalism. Furthermore, OPM enables integration of the various system perspectives into a single unified model that may reflect the complete system's life cycle, based on its complexity management mechanisms that enable continuous refinement of the model [10]. Thus, a highly effective integration of requirement engineering capabilities into a modeling environment would be with OPM.

## 13.5 The OPM Requirements Engineering and Management Framework

Having established the value of model-integrated requirement engineering and management, as well as the main concepts of OPM, we now review and demonstrate the implementation of the proposed approach to connecting these two technologies. The *OPCAT Requirements Engineering Module* (REM) has been developed as an extension of OPCAT, the OPM-based system modeling environment. This integrated requirements engineering and management environment is tightly coupled with the model of the evolving system. It supports requirements modeling, documentation, analyzing, validating, tracing, and management during the entire system's life cycle, including its development and maintenance phases.

OPCAT/REM supports the following necessary requirements engineering activities:

1. Requirements acquisition and modeling: obtaining and maintaining a text-based requirements database
2. Linking requirements to the system model: producing and maintaining the requirements traceability information by linking requirements and model elements



| ID / | Name | Description | Level | Connect... | Things |
|---|---|---|---|---|---|
| Requirements 1 | Application Parameters | The application uses predefined parameters | | ☐ | |
| Requirements 2 | Sending Mails | Messages are sent only in "Sending Mode" | | ☐ | |
| Requirements 3 | Reporting | The tool will support 4 types of reports | | ☐ | |
| Requirements 4 | Reminders | Reminders will be sent to a specific group of receipients | | ☐ | |
| Requirements 5 | Server | Server settings allows changing the server's URL | | ☐ | |
| Requirements 6 | Errors | Error settings allows defining the location of the error log | | ☐ | |

Fig. 13.2 The OPCAT/REM requirements database view

3. Reasoning about requirements: analyzing and validating requirements traceability and reporting deficiencies, including omissions, conflicts, and ambiguity

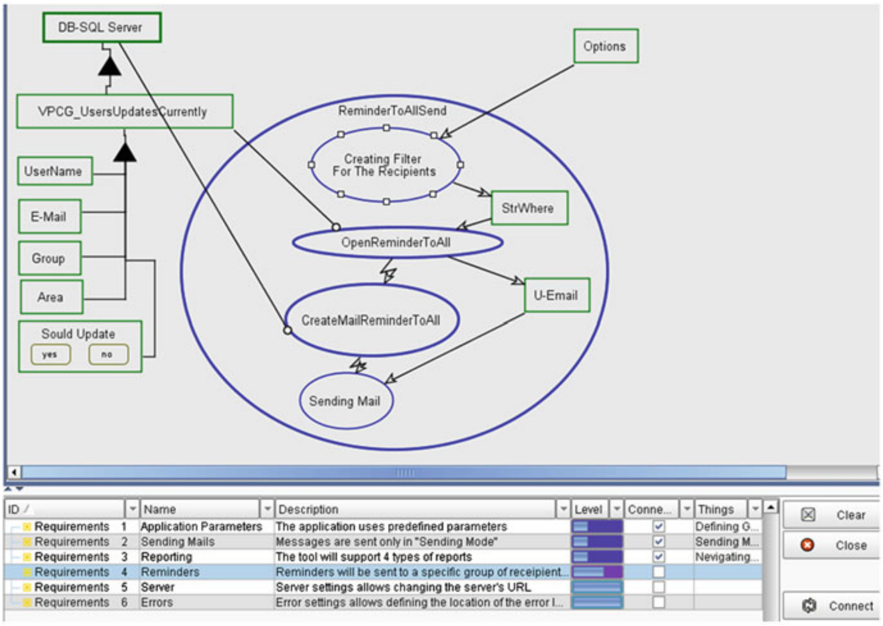
Application of the OPM-based technical solution for integrated requirements management is described in detail next, using a few usage scenarios which represent common engineering tasks during the course of the development cycle. Typically, the OPM unified system model is built incrementally, including the detailed connection between the requirements database and the system specification model.

### 13.5.1 Requirements Acquisition and Modeling

The system's requirements database, which includes textual information concerning each requirements such as ID, description, level (hierarchy), originator, status, priority, owner, and permissions (to modify), may be created and maintained by a dedicated text-based requirements management tool. These tools (such as DOORS [31], RequisitePro [32], or Cradle [33]) typically support exporting the requirements data in CSV (comma-separated value) format, which OPCAT/REM is designed to accept. Alternatively, the requirements table may be created manually by the requirements engineer via a designated user interface as plain text in CSV format. At this point, the requirements database is acquired and incorporated in the OPCAT modeling environment. This step sets the foundation for further processing, which includes (1) creating the traceability links by mapping requirements to model elements and (2) analyzing and validating the integrated (text/model) requirements data.

The OPCAT/REM interface is depicted in Fig. 13.2. This example shows the textual requirements in a grid structure, which is part of OPCAT's Configurable Management View Console, which, in this case, is configured for requirements management. The representation allows flexibility in the level of detail, as each hierarchical level can be collapsed or expanded. The basic view shown in this example includes the unique identifier of the requirement (ID), its title, description, and hierarchical level. The description field enables editing free text description of the requirement. Additional attributes may be included as needed.

Once the requirements database is imported and integrated into the model, OPCAT/REM supports modifications and updates of the requirements set. In order to secure the integrity of the requirements database, which is primarily maintained through an external dedicated tool (e.g., DOORS [31]), all the changes



| ID | Name | Description | Level | Conne... | Things |
|---|---|---|---|---|---|
| Requirements 1 | Application Parameters | The application uses predefined parameters | | ✓ | Defining G... |
| Requirements 2 | Sending Mails | Messages are sent only in "Sending Mode" | | ✓ | Sending M... |
| Requirements 3 | Reporting | The tool will support 4 types of reports | | ✓ | Nevigating... |
| Requirements 4 | Reminders | Reminders will be sent to a specific group of receipient... | | | |
| Requirements 5 | Server | Server settings allows changing the server's URL | | | |
| Requirements 6 | Errors | Error settings allows defining the location of the error l... | | | |

**Fig. 13.3** The requirements linking process

(addition, removal, modification) made to the requirements set through the external tool are reloaded into the OPCAT environment by updating the CSV file. The update is then completed by applying additional processing, mapping, and analyzing, as described next.

### 13.5.2 *Linking Requirements to the System Model*

The novelty of this integrated RE tool is its ability to bind OPM model elements (mainly objects and processes, collectively called "things") from any of the system life-cycle phases (e.g., conceptual modeling, requirements specification, architecture, and design) to the related system requirements. The requirements linking process is described below using Fig. 13.3.

Linking a requirement to a corresponding model element is done by first selecting the desired requirement from the available requirements database (e.g., Requirement 4 – highlighted), then the target model element is selected (e.g., the "Creating Filter for the Recipients" process). The link is made by clicking the "connect" button and indicated by setting a check mark in the "connected?" field (Fig. 13.3 depicts the OPCAT screen just before making the "connection"). Once a requirement has been linked to one or more model elements, the requirement links become an integral part of the OPM model engine. The requirement links are used by the tracing and validation mechanisms described next.

### *13.5.3 Reasoning About Requirements*

Tracing, analyzing, and reasoning are done using the analysis and reporting capability of the tool. Through this capability, OPCAT/REM facilitates efficient and accurate access to the relevant model sections based on the requirement of interest, thus providing the ability to trace requirements directly to the system model. A number of viewing, analysis, and validation functions are available, as follows:

#### 13.5.3.1 Thing: Requirement Matching

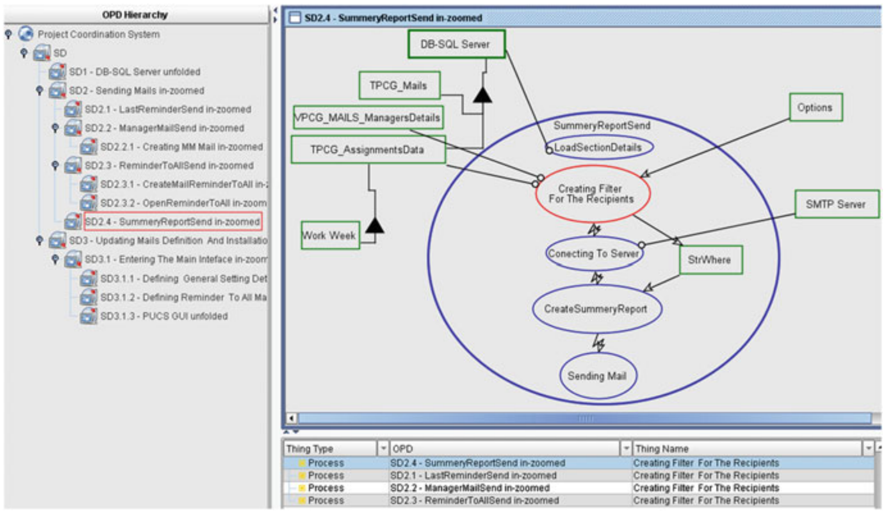
Indicating which requirements are linked to a certain Thing "T" (e.g., "Sending Mails," as depicted in Fig. 13.3). This information can be obtained by using the thing's name "Sending Mails" (="T") as a reduction filter on the things list in the requirements database view (shown in Fig. 13.2). The result will be a new list that includes just the requirements that are associated with "T" (in the same format as depicted in Fig. 13.3).

#### 13.5.3.2 Requirement: Thing Matching

Indicating which model elements ("Things") are linked to a certain requirement "R." The "Things" field in the "R" row in the requirements table (shown in Fig. 13.3) includes a list of the things associated with "R" (the Things list is horizontal, partly visible in Fig. 13.3 due to space constraints). Double-clicking on the "R" row will bring up a list of all the appearances of these things in the model, including the Object-Process Diagram (OPD) in which each thing appears, as shown in Fig. 13.4. Furthermore, double-clicking on the OPD name will bring up that diagram with the relevant thing highlighted in red. For example, the requirement-model association view is depicted in Fig. 13.4. In this example, there are four occurrences of the "Creating Filter for the Recipients" process which are associated with Requirement 4. The desired row is then selected from the list. For example, the first row may be selected, and as a result of double-clicking on the diagram title ("SD2.4 – Summary Report") in that row, the SD2.4 diagram is displayed with the "Creating Filter for the Recipients" process highlighted in red.

The integrated REM module also enables checking for gaps and inconsistencies in the designed system. For example:

- What requirements in the database are not linked to any element of the model? This information can be obtained by using the "not connected" flag (i.e., the "connected?" check mark is not sown) to apply a reduction filter on the "connected?" list.
- Which elements of the model are not linked to any requirement? This information is obtained by clicking the "Not Assigned" button.


OPD Hierarchy
Project Coordination System
SD
SD1 - DB-SQL Server unfolded
SD2 - Sending Mails in-zoomed
SD2.1 - LastReminderSend in-zoomed
SD2.2 - ManagerMailSend in-zoomed
SD2.2.1 - Creating MM Mail in-zoomed
SD2.3 - ReminderToAllSend in-zoomed
SD2.3.1 - CreateMailReminderToAll in-
SD2.3.2 - OpenReminderToAll in-zoom
SD2.4 - SummeryReportSend in-zoomed
SD3 - Updating Mails Definition And Installatio
SD3.1 - Entering The Main Inteface in-zoom
SD3.1.1 - Defining General Setting Det
SD3.1.2 - Defining Reminder To All Ma
SD3.1.3 - PUCS GUI unfolded
SD2.4 - SummeryReportSend in-zoomed
DB-SQL Server
TPCG_Mails
VPCG_MAILS_ManagersDetails
TPCG_AssignmentsData
Work Week
Options
SummeryReportSend
LoadSectionDetails
Creating Filter For The Recipients
SMTP Server
Conecting To Server
StrWhere
CreateSummeryReport
Sending Mail

| Thing Type | OPD | Thing Name |
|---|---|---|
| Process | SD2.4 - SummeryReportSend in-zoomed | Creating Filter For The Recipients |
| Process | SD2.1 - LastReminderSend in-zoomed | Creating Filter For The Recipients |
| Process | SD2.2 - ManagerMailSend in-zoomed | Creating Filter For The Recipients |
| Process | SD2.3 - ReminderToAllSend in-zoomed | Creating Filter For The Recipients |

**Fig. 13.4** The requirement-model association view

- Where are the trace inconsistencies (broken links)? Trace inconsistencies might appear as a result of a scenario in which a requirement R is linked to a thing T such that if the requirement R is deleted (or even modified), when the requirements database is reloaded, the link to T gets broken. This situation may be detected by clicking the "Missing" button, in response to which a list of all broken links will be displayed.

## *13.5.4 Evaluation*

Evaluation and validation of the results of this work were done based on a modeling case study and an experienced users' survey.

The modeling case study was based on a commercial home management system. The purpose of this exercise was to gain experience with the proposed framework and methodology on a larger scale and with a real-world system and to evaluate its usefulness in terms of the quality of the resulting model and the OPM-based model-integrated requirements engineering process. Analysis of these factors enables drawing some conclusions on the effectiveness of the OPM-based REM environment, as follows:

The process of creating, binding, and reasoning about requirements is convenient and user-friendly. It is easy to navigate through the diagram hierarchy and quickly get to where is needed. The concurrent display of the diagram tree structure, the details of a selected component, and the requirements link table in one view was instrumental in seeing the big picture, while working on the details of one particular part of the system. The automation provided by the OPCAT tool (e.g., finding the connected things) is very helpful in increasing the productivity of the development process.

In order to further evaluate and validate the impact of this development with experienced practitioners, we conducted a survey among several software system engineers who have years of experience developing large-scale systems of different types using various modeling techniques. The objective was to evaluate three main aspects of the underlying problem:

1. Corroborating the significance of the requirements-implementation traceability problem and its impact on the quality of resulting systems
2. Assessing the effectiveness of the OPM-based requirements engineering and management process as it is integrated in the development life cycle
3. Evaluating the viability and usefulness of the OPM-based implementation and its potential to be applied in large-scale system development tasks

Overall, the results of the survey provided useful insights that corroborated most of the assumptions set out to examine:

1. The participants generally agreed with our analysis and conclusion that the requirements-implementation information gap is a significant problem that impacts the quality of the developed system.
2. The participants were comfortable with the assertion that integrated model-based requirements engineering process helps to obtain better, less disruptive information flow. However, they were slightly less certain that achieving model integration will contribute to the overall quality of the software product.
3. The participants found the system stronger in some aspects compared to others. In particular, the criteria that were ranked high were clarity, level of detail, completeness, and effectiveness. The participants were less convinced about the required learning curve, the ability of the system to scale, and the support for developers' collaboration while using this system.

## 13.6 Summary and Conclusions

The primary objective of the work presented here was to develop an engineering solution that can reduce the undesired mismatch that often exists between the required system and its implementation. The fundamental approach to achieving this objective is based on the premise that coupling core software engineering concepts with innovative model-driven development methodology enables creation of the desired solution.

While traditional knowledge management techniques focus on creation, organization, and management of knowledge based on the properties and attributes of the data, the approach presented here is primarily focused on the utility of the requirements knowledge. The system development processed is enhanced by integration of two engineering process: requirements engineering and model-based development.

After establishing the value of model-integrated requirement engineering and management and presenting the main concepts of Object-Process Methodology (OPM), the chapter includes a description of the implementation of the proposed approach to connecting these two technologies. OPCAT – the OPM – supporting system modeling tool with its integrated requirements engineering capabilities, supports a requirements engineering process that spans across the entire software life cycle, including design, maintenance, and evolution. System engineers who use OPCAT can quickly and accurately model and specify requirements, architecture, and any structural and behavioral aspects of the system. Furthermore, they can promptly and correctly establish and obtain the traceability between requirements and their implementation, thereby contributing to improving the system quality. These conclusions were also corroborated by the user survey presented in Sect. 13.5.4.

In summary, following this analysis and findings, this chapter exhibits the following contributions of this work:

- Establishing that coupling the requirements engineering process with a conceptual modeling environment and development methodology facilitates production of high-quality software. By using model-integrated requirements-trace information, error detection and analysis, as well as change management can be done more efficiently and correctly. Consequently, the value and impact of effective management of requirements knowledge is extended to the entire system life cycle.
- Development of the OPCAT Requirements Engineering Module (REM) as an extension of OPCAT, the OPM-based system modeling environment. This integrated requirements engineering and management environment is tightly coupled with the model of the evolving system and enables creating, linking, and reasoning about requirements throughout the system's life cycle, including its initiation, development, and maintenance phases.
- Analyzing and evaluating the role of automation and CASE Tool support, including knowledge management and reasoning capability, aimed at improving the effectiveness of the requirements engineering process across the software development and maintenance cycle. As has been shown by the example system and by the user survey, the result can indeed solve the system development problem stated above.

While the tip of the iceberg has been touched here with some basic analytical functions that can be performed on the requirements-integrated model, the opportunity for additional analyses is vast. Specifically, an interesting future research direction is to extend this basic capability to enable the system analyst to answer questions such as the following: To what extent does the requirements model faithfully represent the operational concepts of the system? Does the design fulfill the requirements?

More future work may relate to automation of RE activities which are supported by OPCAT. One possible enhancement is automatically generating an OPM requirements model from text-based requirements specification documentation, a capability that was proven feasible [34]. Other possible enhancements are in the area of requirements trace management: addressing the need to encapsulate the engineering rationale for the link between the requirement and its design within the traceability structure. The challenge is to maintain the effectiveness of the trace information in many development-cycle situations.

## References

1. Gibbs W (1994) Software's chronic crisis. Sci Am 271(3):86–95
2. Sommerville I, Sawyer P (1999) Requirements engineering: a good practice guide. Wiley, Chichester/New York
3. Robertson S, Robertson J (1999) Mastering the requirements process. Addison-Wesley, Reading
4. Capability Maturity Model Integration-SE/SW, Ver. 1.2, Technical Report CMU/SEI-2006-TR-008, 2006
5. Marschall F, Schoenmakers M (2003) Towards model-based requirements engineering for web-enabled B2B applications. In: Proceedings of the 10th IEEE international conference and workshop on the engineering of computer-based systems (ECBS'03), digitally available from IEEExplore, pp 312–320
6. OMG Model Driven Architecture. http://www.omg.org/mda/
7. Jackson MA (1994) The role of architecture in requirements engineering. In: Proceedings, 1st IEEE international conference on requirements engineering, Colorado Springs, April 1994, p 241
8. Ramesh B (1998) Factors influencing requirements traceability practice. Commun ACM 41(2):37–44
9. Brooks F (1975–1995) The mythical man-month. Addison-Wesley, Reading
10. Gotel O, Finkelstein A (1994) Modeling the contribution structure underlying requirements. In: Proceedings of the first international workshop on requirements engineering: foundation of software quality (REFSQ '94), Utrecht, June 1994, p 21
11. Ramesh B, Stubbs C, Powers T, Edwards M (1997) Requirements traceability: theory and practice. Ann Softw Eng 3:397–415
12. Kleppe A, Warmer J, Bast W (2003) MDA explained, the model driven architecture: practice and promise. Addison-Wesley, Boston
13. Brooks F (1987) No silver bullet – essence and accident in software engineering. IEEE Comp 20(4):10–19
14. Harel D (1992) Biting the silver bullet: towards a brighter future for system development. IEEE Comp 25(1):8–20
15. Boehm B, Port D (1999) Escaping the software tar pit: model clashes and how to avoid them. Softw Eng Notes Assoc ComputMach 24(1):36–48
16. Rolland C, Prakash N (2000) From conceptual modeling to requirements engineering. Ann Softw Eng 10:151–176
17. Nuseibeh B, Easterbrook S (2000) Requirements engineering: a roadmap. In: Proceedings, conference on the future of software engineering (ICSE), Limerick, June 2000, pp 35–46
18. Lavazza L, Valetto G (2000) Enhancing requirements and change management through process modeling and measurement, In: ICRE2000 fourth IEEE international conference on requirements engineering, Schaumburg, 19–23 June 2000
19. IBM, the Rational Unified Process. http://www-01.ibm.com/software/awdtools/rup/
20. Gotel O, Finkelstein A (1994) An analysis of the requirements traceability problem. In: Proceedings of the first international conference on requirements engineering, Colorado Springs, April 1994, pp 94–101

21. Ramesh B, Jarke M (2001) Toward reference models for requirements traceability. IEEE Trans Softw Eng 27(1):58–93
22. Egyed A, GrunbacherP (2002) Automating requirements traceability: beyond the record & replay paradigm. In: Proceedings of the 17th international conference on automated software engineering, IEEE CS, Edinburgh
23. Boehm B, Egyed A (1999) Optimizing software product integrity through life-cycle process integration. J Comp Stand Inter 21:63–75
24. The International Council on Systems Engineering (INCOSE) (1999) Tools survey: requirements management tools. http://www.incose.org/tools/tooltax.html, April 1999
25. Badar M, Zowghi D Developing a requirement toolset: lessons learned. In: Proceedings of the Australian software engineering conference (ASWEC'04), Melbourne, April 2004
26. Dori D, Reinhartz-Berger I, Sturm A (2003)OPCAT – a bimodal case tool for object-process based system development. In: Proceedings IEEE/ACM 5th international conference on enterprise information systems (ICEIS 2003), Angers, pp 286–291
27. Dori D (1995) Object-process analysis: maintaining the balance between structure and behavior. J Logic Comput 5(2):227–249
28. Dori D (2002) Object-process methodology – a holistic systems paradigm. Springer, Berlin/New York
29. OPM – The Official Web Site. http://www.objectprocess.org
30. Kotonya G, Sommervile I (1998) Requirements engineering: processes and techniques. Wiley, Chichester/New York
31. DOORS – IBM Rational. http://www-01.ibm.com/software/awdtools/doors/
32. IBM Software – Rational RequisitePro. http://www-01.ibm.com/software/awdtools/reqpro/
33. Cradle: requirements management tools. http://www.threesl.com/pages/index.php
34. Dori D, Korda N, Soffer A, Cohen S (2004) SMART: system model acquisition from requirements text.In: Proceedings: international conference on business process management, Potsdam June 2004

# Part V
# Intelligent Tool Support

"The expectations of life depend upon diligence; the mechanic that would perfect his work must first sharpen his tools."
—Confucius

# Chapter 14
# An Overview of Recommender Systems in Requirements Engineering

**A. Felfernig, G. Ninaus, H. Grabner, F. Reinfrank, L. Weninger, D. Pagano, and W. Maalej**

**Abstract** Requirements engineering (RE) is considered as one of the most critical phases in software development. Poorly implemented RE processes are still one of the major risks for project failure. As a consequence, we can observe an increasing demand for intelligent software components that support stakeholders in the completion of RE tasks. In this chapter, we give an overview of the research dedicated to the application of recommendation technologies in RE. On the basis of a literature analysis, we exemplify the application of recommendation technologies in different scenarios. In this context, the approaches of collaborative filtering, content-based filtering, clustering, knowledge-based recommendation, group-based recommendation, and social network analysis are discussed. With the goal to stimulate further related research, we conclude the chapter with a discussion of issues for future work.

A. Felfernig (✉) • G. Ninaus • H. Grabner • F. Reinfrank
Graz University of Technology, Graz, Austria
e-mail: alexander.felfernig@ist.tugraz.at; gerald.ninaus@gmail.com; hgrabn@ist.tugraz.at; freinfra@ist.tugraz.at1

L. Weninger
wsop, Vienna, Austria
e-mail: LWeninger@wsop.at

D. Pagano
Technische Universität München, Munich, Germany
e-mail: pagano@cs.tum.edu

W. Maalej
University of Hamburg, Hamburg, Germany
e-mail: maalej@informatik.uni-hamburg.de

## 14.1 Introduction

Requirements engineering (RE) is considered as one of the most critical phases of a software development project. Poorly implemented RE is one of the major reasons for the failure of a project [1]. Core activities of an RE process are elicitation and definition, quality assurance, negotiation, and release planning [2]. Due to the increasing size and complexity of software systems, we can observe a growing demand for intelligent methods, techniques, and tools that can help to improve the overall quality of RE processes [3–5]. In this chapter, we focus on the aspect of how different types of recommendation technologies [6] can be applied to support stakeholders in the completion of their RE tasks.

A recommender system can be defined as *any system that guides a user in a personalized way to interesting or useful objects in a large space of possible options or that produces such objects as output* [6, 7]. Recommender systems are intensively applied for the purpose of recommending products and services such as movies, books, digital cameras, and financial services. Such systems support users in the identification of relevant items in situations where the amount and/or complexity of an assortment outstrips their capability to survey it and to reach a decision [8]. *Low-involvement items* such as movies and books are often recommended by analyzing the preferences of users with a similar rating behavior. The corresponding recommendation approach is *collaborative filtering* [9] which is a basic implementation of word-of-mouth promotion where purchase decisions are influenced by the opinion of relatives and friends: if two users rated similar items in a similar fashion in the past, a collaborative filtering-based recommender system would propose new items to one user that the other one has already rated positively. For example, the online selling platform amazon.com recommends books which have already been purchased by customers with a similar rating behavior [10]. An alternative approach to the recommendation of low-involvement items is *content-based filtering* [11]. It is an approach to information filtering where features of items a user liked in the past are exploited for determining new recommendations for the same user. For example, if a customer of amazon.com bought books related to the Linux operating system, similar books (related to Linux) will be proposed in future recommendation sessions. *High-involvement items* such as digital cameras or financial services are recommended on the basis of *knowledge-based recommender applications* where predefined recommendation rules are exploited by the recommendation engine to determine a set of candidate items [12]. Typically, rating-based recommendation approaches are not applicable to high-involvement items since such items are not purchased frequently and therefore no up-to-date rating data is available.

The major contributions of this chapter are the following. First, we provide an overview of the research related to the application of recommendation technologies in RE. Second, we show in detail how different types of recommendation techniques can be applied to proactively support users in RE scenarios. Third, we want to stimulate new ideas and research by discussing a couple of issues for future work.

**Table 14.1** Overview of recommendation approaches for requirements engineering

| Activity | Scenario *Si* | Recommendation approach | |
|---|---|---|---|
| Elicitation and definition | Recommending stakeholders ($S_1$) | Social network analysis | Lim et al. [17] |
| | | Content-based filtering | Cleland-Huang et al. [5], Castro et al. [18] |
| | Recommending requirements ($S_2$) | Content-based filtering | Dumitru et al. [19] |
| | | Social network analysis | Lim and Finkelstein [20] |
| | | Collaborative filtering | Cleland-Huang et al. [5], Castro et al. [18] |
| Quality assurance | Managing feature requests ($S_3$) | Clustering | Cleland-Huang et al. [22] |
| | | Machine learning | Fitzgerald et al. [21] |
| | Consistency management ($S_4$) | Knowledge-based recommendation | Felfernig et al. [23] |
| | Dependency detection ($S_5$) | Clustering | Cleland-Huang et al. [22] |
| Negotiation and planning | Triage ($S_6$) | Clustering, Utility theory | Duan et al. [24] |
| | Release planning ($S_7$) | Group recommendation | Felfernig et al. [25] |
| | | Utility theory | Felfernig et al. [23], Ruhe et al. [26, 27] |

The remainder of this chapter is organized as follows. In Sect. 14.2, we provide an overview of research on the application of recommendation technologies in RE. In the following, we discuss application scenarios for recommendation technologies with a focus on collaborative filtering [13], content-based filtering [11], clustering [14], knowledge-based recommendation [7, 12], group-based recommendation [15], and social network analysis [16] (Sect. 14.3). Relevant issues for future research are discussed in Sect. 14.4. With Sect. 14.5, we conclude this chapter.

## 14.2 Research on Recommender Systems in Requirements Engineering

In this section, we discuss existing research dedicated to the application of recommendation technologies in requirements engineering (RE). Our discussion of related research is organized along the typical activities in an RE process. In this context, we take into account the activities of *requirements elicitation and definition*, *quality assurance*, and *negotiation and release planning*. For each activity, we discuss relevant application scenarios for recommendation techniques. Table 14.1 provides an overview of these scenarios. Further technical insights into recommendation techniques are provided in Sect. 14.3.

### 14.2.1 Requirements Elicitation and Definition

Requirements elicitation and definition focuses on the collection of requirements from different stakeholders. Typical resulting artifacts are, for example, textual requirement descriptions, scenario descriptions, use cases, and sketches of prototypical user interfaces. The following application scenarios for recommendation techniques are related to activities in the context of requirements elicitation and definition.

*Recommending Stakeholders* ($S_1$). This is an important task in the early phase of an RE process since, for example, a low degree of user involvement in most cases leads to project failure [17]. The major goal of stakeholder identification is to identify a set of persons who are capable of providing a complete and accurate description of the software requirements. Identifying a set of authorized, collaborative, responsible, committed, and knowledgeable stakeholders is an important and challenging task [5, 17]. A common mistake is that wrong representatives of groups are integrated into a project or that important stakeholders are simply omitted.

*StakeNet* [17] is an approach to stakeholder identification which is based on the concepts of social network analysis [16, 28]. In *StakeNet*, an initial set of stakeholders and recommendation information provided by these stakeholders is applied for the construction of a social network (SN). The included nodes represent the stakeholders, and connections between nodes represent recommendations articulated by the stakeholders, that is, if stakeholder $s_i$ recommends stakeholders $s_j$ with a certain rating, then this information is included in the corresponding social network. This process of stakeholder recommendation is repeated in order to exploit a kind of snowball effect. On the basis of the constructed social network, different SN analysis measures are used for stakeholder prioritization. An example of such a measure is *betweenness centrality* which determines for a specific stakeholder $s_i$ the number of shortest paths between other stakeholders in which $s_i$ is contained. A high value of this measure indicates a person's (stakeholder $s_i$'s) capability of acting as a broker between different groups of stakeholders.

Especially in large-scale and distributed software projects, it is infeasible to organize personal meetings on a regular basis. In such scenarios, requirements are often defined in wiki-based forums which are receptive to the problems of information overload, redundancy, incompleteness of information, and diverging opinions of different stakeholders. In their approach to improve the stakeholder support in *ultra-large-scale software systems* development (ULS software systems), Cleland-Huang et al. [5] and Castro-Herrera et al. [18] show how to exploit clustering techniques for grouping user requirements and in the following to assign (recommend) stakeholders to clusters on the basis of content-based filtering [11]. One major motivation for such an assignment of stakeholders to requirement clusters is to achieve a representative coverage, that is, each requirement should be discussed and evaluated by a sufficient number of stakeholders.

*Recommending Requirements* ($S_2$). A systematic reuse of already existing software requirements has the potential of significantly reducing the overall costs of a software project. A recommendation-based approach to requirements reuse is presented by Dumitru et al. [19]. The basic idea is to analyze requirements which are accessible in software project repositories and to apply clustering techniques for the intelligent grouping of such requirements. The identified requirement groups can be analyzed in future software projects for the purpose of reuse and also for the purpose of completeness checking (are all relevant requirements contained in the current requirements model). The proposed recommendation approach is content-based filtering, where a vector of keywords (derived from the description of the new software project) is matched with the keywords extracted from requirements artifacts from the repository of already completed software projects.

Lim and Finkelstein [20] introduce the *StakeRare* approach which supports the identification and reuse of requirements. *StakeRare* [20] is based on the aforementioned *StakeNet* approach [17]. In *StakeRare*, stakeholders are rating initial sets of requirements. Additional (new) requirements currently not contained in the list are then recommended using the concepts of collaborative filtering [13]. On the basis of the rating information (weighted with the stakeholders' weight (influence) in the current project), requirements are prioritized.

Similar to their approach of recommending (assigning) stakeholders to requirement clusters (topics), the approaches discussed in Cleland-Huang et al. [5] and Castro-Herrera et al. [18] also support the recommendation of requirements to stakeholders, for example, based on the concepts of collaborative filtering. A major motivation for the application of collaborative filtering in this scenario was to achieve serendipity effects which help to increase requirements quality (stakeholders receiving recommendations regarding requirements they are interested in have a higher probability of analyzing these requirements). Another motivation for the application of collaborative filtering is to improve requirement model understanding since it generates personalized navigation paths for stakeholders.

## *14.2.2 Quality Assurance*

A set of requirements has to be evaluated regarding properties such as consistency (requirements are not contradictory), completeness (all relevant requirements should be part of the requirements model), feasibility (technical feasibility as well as economic feasibility), understandability (does the description fulfill the quality standards), and reusability (are the requirements reusable in future projects). Currently, recommenders are applied to support the following quality assurance scenarios.

*Managing Feature Requests* ($S_3$). The major goal of feature request management is to support the effective management of large sets of software features. Unstructured request management can lead to suboptimal communication between stakeholders and to the selection of irrelevant features [5]. An approach to support effective feature management has been introduced by Cleland-Huang et al. [22] where clusters of similar requirements are exploited for the identification of redundancies and the prioritization of feature requests. Fitzgerald et al. [21] introduce an approach to feature request management which is based on the idea of predicting software failures (e.g., stopped implementation of a feature) by analyzing the communication threads in feature management systems. Their approach to failure identification is based on the idea of applying different machine learning techniques for the construction of a prediction model for failures. The basis for learning this prediction model is logged feature requests and their related positive or negative outcomes. Prediction models are derived on the basis of parameters that are assumed to be important for specifying the quality of a feature request, for example, *involvement of the right stakeholders* or *sufficient engagement of stakeholders* in terms of contributing to a feature-related discussion thread.

*Consistency Management* $(S_4)$. Inconsistencies between requirements are resulting from factors such as not enough time for consistency checking, different perceptions and goals, or different granularity of knowledge [29]. Especially for informally defined requirements, the complete automation of consistency management is unrealistic [29], but semiautomated tools help to keep the efforts acceptable. Assuming the existence of a formal description of the requirements model (on the basis of a constraint satisfaction problem [30]) and the stakeholder preferences (priorities) regarding the defined set of requirements, Felfernig et al. [23] introduce an approach to the automated diagnosis of inconsistent requirement models and inconsistent stakeholder preferences. In this context, a diagnosis is interpreted as a minimal set of stakeholder preferences (or requirements) that have to be adapted or deleted in order to restore consistency. A detailed introduction to the concepts of model-based diagnosis can be found, for example, in the work of Reiter [31].

*Dependency Detection* $(S_5)$. Due to the fact that requirements are often represented on an informal level, the analysis regarding properties such as model consistency and completeness is challenging. Relationships between requirements are typically expressed in terms of dependencies (e.g., requirement $A$ requires requirement $B$ or requirement $A$ is incompatible with requirement $B$) which are defined by stakeholders. Recommender systems allow the provision of additional information which proactively supports stakeholders in the identification of dependencies. Dependency detection between requirements can be based, for example, on clustering techniques where requirements are grouped into clusters of similar topics (see, e.g., Cleland-Huang et al. [22]). The basic underlying assumption is that requirements which are assigned to the same cluster are depending on each other. Although helpful, this approach does not result in a complete specification of the type of dependency but serves as a basis for a further analysis by stakeholders. Some preliminary work regarding requirements and inconsistency discovery in *Open Source Software Development (OSSD)* which is based on the methods of Natural Language Processing (NLP) is presented by Fantechi and Spinicci [32].

### 14.2.3 Requirements Negotiation and Release Planning

Requirements negotiation is the process of identifying conflicts between stakeholder preferences and to facilitate efficient stakeholder decision-making regarding priorities and acceptance (this process is also denoted as *requirements triage*). The major goal of release planning is the development of a schedule which specifies in which development (release) period which requirement should be implemented [26].

*Requirements Triage* ($S_6$). Restrictions regarding the available resources (e.g., budget and employees) and defined deadlines for the completion of a software system in many cases require decisions regarding the set of requirements which should be implemented. Requirements have to be prioritized in order to take aside unimportant requirements and to support project managers in conflict resolution and making tradeoffs. Prioritization of requirements is often a complex and iterative communication and decision process [33] which has to take into account different soft factors such as company policies, personal preferences, and social relationships between stakeholders. The term *requirements triage* stems from medical decision-making [34]. In disaster scenarios, victims are categorized into three types: those who will die (independent of the medication), those who will survive (independent of the medication), and those whose survival depends on the given medication. Requirements prioritization has to deal with a similar task: identify the requirements which must not be included in the next release, requirements that are optional for the next release, and the requirements that must be included in the next release.

The lack of efficient triage processes in large software projects with hundreds of stakeholders and thousands of (sometimes conflicting) requirements led to the development of intelligent technologies supporting the semiautomated requirements prioritization. The approach presented by Duan et al. [24] focuses on the generation of clusters which are derived from different clustering criteria. The weight of clustering criteria is specified by stakeholders, and an initial prioritization is generated on the basis of a utility function. This utility function is based on the number of clusters a requirement is included in and the weights of these clusters.

*Recommendation of Release Plans* ($S_7$). Ruhe et al. [27] introduce an approach to release planning that is based on the concept of linear programming [35]. The basic idea is to define a linear program that should calculate a sequence of assignments of features to a corresponding release taking into account the dependencies between the different features. Ruhe et al. [26] show how to apply AHP (Analytical Hierarchy Process) for determining a set of preferred requirements. Felfernig et al. [23, 36] extend the work of Ruhe et al. [26] by introducing automated diagnosis and repair mechanisms which effectively help to figure out minimal sets of acceptable changes in situations where release plan preferences of stakeholders become inconsistent. The contributions of [23, 26, 27, 36] are important steps towards improving the quality of requirements selection. However, these approaches depend on the assumption that stakeholders know their preferences and that *preferences remain stable*. In the line of traditional models of human decision-making, it is assumed that humans are taking decisions on the basis of rational thinking [37]. Following such models, a human would take the optimal decision following a formal evaluation process. In contradiction to these models, research has clearly pointed out the fact that preference stability in decision processes does not exist and can also be easily manipulated [38]. A customer who wants to purchase a digital camera could first define a strict upper limit for the price. But due to additional technical information about the camera, the customer could change her mind and significantly increase the upper limit of the price. This typical example of preference reversal [39] indicates the nonexistence of stable preferences. Instead, the model of preference construction [38] should be used, in which decision-making processes are more characterized by a process of iterative refinement and adaptation of the current preferences in the face of new alternatives and as well in the face of opinions of other users that are visible to the decision maker.

The idea of applying group decision-making techniques in requirements engineering is to exploit basic *decision heuristics* [40] such as *majority voting* (the decision is taken conform to the majority of the votes of the engaged stakeholders) or the *fairness heuristic* which guarantees that none of the stakeholders will be disadvantaged in the group decision process. Group decision heuristics already play an important role in application scenarios outside software engineering [40]. Felfernig et al. [25] applied group decision heuristics in the context of RE scenarios. They introduce the *IntelliReq* environment which can be used for supporting group decision processes in distributed settings (e.g., open-source platforms or large and distributed software projects). The authors present the results of an empirical study which show that group recommendation technologies can help to improve the perceived quality of decision support. A further insight was that stakeholders should not be confronted with the preferences of other group members at the beginning of prioritization – the reason is that knowledge about preferences automatically triggers insufficient information exchange between group members.

## 14.3 Recommendation Techniques for Requirements Engineering

In order to show how recommendation techniques can be exploited in the RE context, we will now discuss basic application scenarios. These scenarios should help to develop an understanding of potential applications of recommendation techniques and show how different recommendation approaches have to be tailored in order to be applicable. Note that we interpret recommendation technologies as supportive technologies; we do not claim that information gaps in general can be closed by the application of recommendation technologies. Information does not substitute communication, that is, effective RE processes still heavily rely on personal stakeholder interaction. Furthermore, the quality of recommendations depends on the quality of information provided by stakeholders, that is, the successful application of recommendation technologies is only possible on the

**Table 14.2** Example of a content-based filtering recommendation problem

| Requirement | Category | Planned release | Efforts (person days) | Description |
|---|---|---|---|---|
| $r_1$ | Database | 1 | 150 | Store component configuration in DB |
| $r_2$ | User interface | 2 | 60 | User interface with online help available |
| $r_3$ | Database | 1 | 300 | Separate tier for DB independence |
| $r_4$ | User interface | 1 | 30 | User interface with corporate identity |

basis of motivated and proactive stakeholders. Finally, successful RE depends on process quality, which cannot be achieved and guaranteed only by the application of recommendation technologies. In the following, we discuss basic RE application scenarios for the major types of recommendation technologies which are *content-based filtering* (CBF) [11], *clustering* [14], *collaborative filtering* (CF) [13], *group recommendation* (GR) [41], *social network analysis* [16], and *knowledge-based recommendation* (KBR) [7, 12].

*Content-Based Filtering.* Content-based filtering (CBF) [11] exploits the similarities between the preferences of the current user and descriptions of items the user did not notice up to now. User preferences can be, for example, represented by frequent keywords extracted from artifacts previously processed by the user. Another alternative are predefined categories assigned to items as meta-information. Typical recommendations derived by CBF recommenders are of the form *item C is recommended since you were also interested in item A* (which is similar to item *C*).

When *defining requirements*, a recommender can support stakeholders, for example, by indicating similar requirements or point out requirements already defined in previous projects. Let us assume the active stakeholder ($s_1$) has already investigated the requirement $r_1$ which has the assigned category *database* (see Table 14.2). Now, CBF would recommend requirement $r_3$ if $r_3$ has not been investigated up to now by the active stakeholder. If no such categorization of requirements is available, the detailed textual description of requirements can be used: keywords have to be extracted [42], and the determination of similar requirements can then be based on the similarity of the extracted keywords – a simple corresponding similarity metric is shown in Formula 14.1.[1] For example, $sim(r_1, r_3) = 0.17$, if we assume $keywords(r_1) = \{store, component, configuration, DB\}$ and $keywords(r_3) = \{tier, DB, independence\}$:

$$sim(s, r) = \frac{|keywords(s) \cap keywords(r)|}{|keywords(s) \cup keywords(r)|} \tag{14.1}$$

[1] Note that the parameter *s* in Formula 14.1 represents a *user profile*; however, this approach can as well be applied to calculate the similarities between different requirements, that is, $sim(r_i, r_j)$.

**Table 14.3** Keywords extracted from the textual requirement descriptions in Table 14.2

| Requirement | Extracted keywords |
|---|---|
| $r_1$ | Store component configuration DB |
| $r_2$ | User interface help |
| $r_3$ | Tier DB independence |
| $r_4$ | User interface corporate |

**Table 14.4** Example of a collaborative recommendation problem. A table entry *ij* with value 1 (0) denotes that fact that stakeholder $s_i$ has (has not) inspected the requirement $r_j$

| | $r_1$ | $r_2$ | $r_3$ | $r_4$ |
|---|---|---|---|---|
| $s_1$ | 1 | 0 | 1 | 0 |
| $s_2$ | 1 | 0 | 1 | 1 |
| $s_3$ | 1 | 1 | 0 | 1 |

*k-Means Clustering*. A basic method for determining clusters is *k-means clustering* [43] where *k* specifies the number of clusters being sought. In the initial iteration, *k* requirements can be chosen as *cluster centers*, and the other requirements are assigned to their closest cluster. Different distance metrics can be applied [14] – for the purposes of our example, we apply the similarity between keywords (see Table 14.3) extracted from the textual description of our example requirements ($\{r_1, r_2, r_3, r_4\}$ in Table 14.2). Thereafter, the centroid (mean) per cluster is determined for each cluster, and an assignment of requirements to clusters takes place again. For our example, we assume that after one step, the two clusters $c_1{:}\{r_1, r_3\}$ and $c_2{:}\{r_2, r_4\}$ have been identified where $sim(r_1, r_3) = 0.17$ and $sim(r_2, r_4) = 0.5$. The algorithm terminates if a maximum iteration depth has been reached or all clusters remain stable.

*Collaborative Filtering*. Collaborative filtering (CF) [13] is perhaps the most widespread recommendation approach where information about the rating behavior of *nearest neighbors* (i.e., users with similar ratings compared to the current user) is exploited for predicting the current user's ratings for items not known to her/him yet. Typical recommendations derived by CF recommenders are of the form *users who were interested in item A were also interested in item C*.

When stakeholders try to *understand a given set of requirements* (e.g., new stakeholders in the project), recommender systems can provide support in terms of showing related artifacts or showing those artifacts stakeholders have investigated when working on the current or a similar requirement. In the setting of Table 14.4, the requirements $\{r_1, r_2, r_3, r_4\}$ have already partially been investigated by the stakeholders $\{s_1, s_2, s_3\}$. For example, stakeholder $s_1$ has already investigated the requirements $r_1$ and $r_3$. The main idea of collaborative filtering (CF) is to exploit user ratings (in our context, the rating $= 1$ if a stakeholder has already investigated a certain requirement and the rating $= 0$ if the stakeholder did not investigate the requirement up to now) in order to identify additional requirements the stakeholder may be interested in. *User-based* CF is a basic variant which is often used in industrial contexts [13]. User-based CF tries to identify the k-nearest neighbors (stakeholders interested in a similar set of requirements) of the current user

**Table 14.5** Example of a decision problem: deciding about the group evaluation of requirement *r*

| Requirement: *r* | $s_1$ | $s_2$ | $s_3$ | $s_4$ |
|---|---|---|---|---|
| Quality | Medium | Medium | Medium | High |
| Effort (person days) | 10 | 7 | 14 | 8 |
| Decision | Accept | Revision | Accept | Accept |

(stakeholder) and calculates a prediction for the rating of an item the stakeholder has not investigated up to now. Such a rating can be defined, for example, as the weighted majority of the k-nearest neighbors. In our example, stakeholder $s_2$ can be identified as the nearest neighbor (if we set $k = 1$) since $s_2$ has investigated all the requirements investigated by stakeholder $s_1$. Vice versa, stakeholder $s_1$ did not investigate the requirement $r_4$ up to now – in this context, collaborative filtering would recommend requirement $r_4$ to stakeholder $s_1$ since the nearest neighbor of $s_1$ has already investigated $r_4$.

*Group Recommendation.* The major goal of group recommendation (GR) technologies [41] is to support/achieve consensus among group members. GR can support groups in their decision process by taking into account the fact that individual decisions depend on various factors, such as own evaluation of a solution alternative, beliefs about the opinions of group members, and information about the individual motivation (e.g., egocentric or cooperative motivation [41]). GR includes heuristics that can be exploited for identifying solution alternatives that are (with a high probability) accepted by all or at least the majority of group members. Typical recommendations derived by group recommenders are of the form *this recommendation tries to take into account the preferences of all group members*.

Requirements evaluation and negotiation have a clear need of group decision support: a group of stakeholders has to decide about the quality of requirements and in the following to figure out which requirements should be accepted without a change. Let us assume that the requirement *r* has to be evaluated by the stakeholders $\{s_1, s_2, s_3, s_4\}$ – the individual evaluations of *r* are depicted in Table 14.5.

In this context, group recommendation concepts can be applied which propose alternatives to be further evaluated by the group. Different strategies for determining such a group recommendation are possible [15], for example, the *least-misery strategy* would propose evaluations that are stable in the sense that none of the evaluation dimensions have been overestimated (or underestimated, e.g., in the case of *person days*). Applying this strategy in our context would mean to propose the evaluation (*quality* = *medium*, *effort* = *14*, *decision* = *revision*) as first alternative for the overall group decision. On the basis of this and further proposals, each individual stakeholder enters the next review round with the goal to achieve (if possible) a consensus regarding the evaluation. A detailed discussion of further strategies for determining group recommendations can be found in Masthoff [15].

*Social Network Analysis.* With the concepts of social network analysis, different properties of a network of stakeholders engaged in an RE process can be identified. In order to sketch the analysis of *betweenness centrality* of stakeholders, we introduce the communication patterns between the stakeholders $\{s_1, s_2, s_3, s_4, s_5\}$ in Table 14.6. For simplicity, we assume that each discussion thread (related to one

**Table 14.6** Communication patterns (e.g., in a discussion forum) between stakeholders $\{s_1, s_2, s_3, s_4, s_5\}$ regarding requirements $\{r_1, r_2, r_3, r_4\}$

| Requirement: $r$ | Comment 1 | Comment 2 | Comment 3 | Comment 4 |
|---|---|---|---|---|
| $r_1$ | $s_1$ | $s_2$ | $s_1$ | $s_2$ |
| $r_2$ | $s_3$ | $s_4$ | $s_1$ | $s_3$ |
| $r_3$ | $s_3$ | $s_5$ | $s_3$ | $s_5$ |
| $r_4$ | $s_3$ | $s_1$ | $s_3$ | $s_1$ |

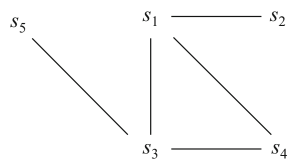


**Fig. 14.1** Example social network (sn) derived from communication patterns of Table 14.6

**Table 14.7** *Betweenness centrality* values for stakeholders $\{s_1, s_2, s_3, s_4, s_5\}$. For example, stakeholder $s_1$ has a betweenness centrality of three since he/she is included in three shortest paths between stakeholders $s_j$ ($s_j \neq s_i$); these shortest paths are $\{s_2 - s_3,\ s_2 - s_4,\ s_2 - s_5\}$

| Stakeholder | Shortest paths between $s_j$ | Betweenness centrality |
|---|---|---|
| $s_1$ | $s_2 - s_3$, $s_2 - s_4$, $s_2 - s_5$ | 3.0 |
| $s_2$ | – | 0.0 |
| $s_3$ | $s_1 - s_5$, $s_2 - s_5$, $s_4 - s_5$ | 3.0 |
| $s_4$ | – | 0.0 |
| $s_5$ | – | 0.0 |

requirement) includes at most four comments and stakeholder $s_i$ is connected to stakeholder $s_j$ in a social network (see Fig. 14.1) if both are in at least one common discussion thread. *Betweenness centrality* measures for each stakeholder $s_i$ the number of shortest paths between pairs of other stakeholders $s_j(s_i \neq s_j)$ in which $s_i$ is included. Table 14.7 depicts the results of the *betweenness centrality* evaluation in our working example; the stakeholders $s_1$ and $s_3$ have a centrality measure of 3.0 (both are part of three shortest paths between stakeholders $s_j(s_j \neq s_i)$) whereas the other ones have a betweenness centrality of 0.0. This way we are able to analyze the role of a person in communication processes (with regard to certain topics). As such, this measure can be exploited as a first basic selection criterion for stakeholders who should participate in a project.

*Knowledge-Based Recommendation.* Knowledge-based recommendation (KBR) [7, 12] exploits formal knowledge about the offered item assortment, knowledge about user preferences, and knowledge about which items should be recommended in which context. The explicit form of knowledge representation in terms of rules (constraints) allows the generation of deep explanations as to why a certain item has been recommended or why no solution exists in a certain recommendation context [36]. Typical recommendations derived by KBR are of the form *you specified the item properties I={x,y,z} therefore we recommend* C which supports all the properties of *I*.

**Table 14.8** Example of inconsistent stakeholder preferences: each table entry represents a constraint $c_{ij}$, where $c_{ij} = 1$ (0) denotes the fact that stakeholder *i* wants to include (exclude) requirement *j*

| | $s_1$ | $s_2$ | $s_3$ |
|---|---|---|---|
| $r_1$ | 1 | 1 | 1 |
| $r_2$ | 1 | 0 | 1 |
| $r_3$ | 0 | 0 | 1 |
| $r_4$ | 1 | 1 | 1 |

**Table 14.9** Example importance values for the stakeholder preferences shown in Table 14.8

| | $s_1$ | $s_2$ | $s_3$ |
|---|---|---|---|
| $r_1$ | $imp(c_{11}) = 0.5$ | $imp(c_{21}) = 0.3$ | $imp(c_{31}) = 0.4$ |
| $r_2$ | $imp(c_{12}) = 0.2$ | $imp(c_{22}) = 0.3$ | $imp(c_{32}) = 0.2$ |
| $r_3$ | $imp(c_{13}) = 0.2$ | $imp(c_{23}) = 0.2$ | $imp(c_{33}) = 0.2$ |
| $r_4$ | $imp(c_{14}) = 0.1$ | $imp(c_{24}) = 0.2$ | $imp(c_{34}) = 0.2$ |

**Table 14.10** Utility values of repair actions $\{repc_1, repc_2, repc_3, repc_4\}$

| $repc_k \in REP_c$ | $Utility(repc_k)$ |
|---|---|
| $repc_1$ | 2 |
| $repc_2$ | 1.42 |
| $repc_3$ | 1.66 |
| $repc_4$ | 1.25 |

Knowledge-based recommendation technologies can support consistency management as well as intelligent explanations in situations where no solution (e.g., release plan) can be identified due to contradicting stakeholder preferences [23, 36]. Table 14.8 depicts a set of requirements $R = \{r_1, r_2, r_3, r_4\}$ and a set of stakeholders $S = \{s_1, s_2, s_3\}$. For each requirement $r_i \in R$, each stakeholder specifies his/her preferences which can be 1 (*include*) or 0 (*exclude*), for example, $c_{12} = 1$ denotes the fact that stakeholder $s_1$ wants to include requirement $r_2$ in the next software release. The set of stakeholder preferences is denoted as $C = \cup c_{ij}$. Inclusion and exclusion are example constraints (preferences). Further types of constraints are possible (see, e.g., the RE ontology proposed by Lohmann et al. [44]) but not used in this example. For the preferences shown in Table 4, no solution exists, that is, the stakeholder preferences are inconsistent (Tables 14.9 and 14.10).

The first step to resolve this inconsistency is to figure out combinations of constraints (preferences) that are causes for the inconsistency, for example, the stakeholder preference $c_{12}$ is inconsistent with the preference $c_{22}$. The complete set of such (minimal [45]) inconsistencies is $CON = \{con_1:\{c_{12}, c_{22}\}, con_2:\{c_{22}, c_{32}\}, con_3:\{c_{13}, c_{33}\}, con_4:\{c_{23}, c_{33}\}\}$. Such sets can be determined using the algorithm presented by Junker [45]. We can now determine all possible repairs for the given set C of stakeholder preferences by simply deleting at least one element from each subset of CON (see [31]). The possible repair constraint sets $rep_k$ for CON are elements of $REP = \{rep_1:\{c_{22}, c_{33}\}, rep_2:\{c_{22}, c_{13}, c_{23}\}, rep_3:\{c_{12}, c_{32}, c_{33}\}, rep_4:\{c_{12}, c_{32}, c_{13}, c_{23}\}\}$ where a repair constraint set $rep_k$ is defined as a *minimal set of stakeholder preferences* (see [46]) that have to be changed in order to make the stakeholder preferences consistent.

For the given set REP, we can identify the following set of concrete repair actions: $\text{REP}_c = \{\text{repc}_1{:}\{c_{22} = 1,\ c_{33} = 0\},\ \text{repc}_2{:}\{c_{22} = 1,\ c_{13} = 1,\ c_{23} = 1\},\ \text{repc}_3{:}\{c_{12} = 0,\ c_{32} = 0,\ c_{33} = 0\},\ \text{repc}_4{:}\{c_{12} = 0,\ c_{32} = 0,\ c_{13} = 1,\ c_{23} = 1\}$. $\text{REP}_c$ can now be considered as a set of alternative and minimal repairs for the original set of stakeholder preferences such that consistency between the preferences can be restored.

## 14.4 Issues for Future Research

Based on our analysis of existing research on the application of recommendation technologies in requirements engineering (RE), we now focus on a discussion of relevant *issues for future research.*

*Decision Support and Preference Construction.* Existing RE approaches often rely on the assumption of stable stakeholder preferences (e.g., in the context of requirements negotiation). The assumption of stable preferences is not applicable for RE scenarios; in fact, related decision-making follows an incremental preference construction process [25, 47]. In order to better integrate recommendation technologies into RE processes, we are in the need of deep knowledge about human decision strategies. Such a knowledge will help us to improve the decision support quality. The integration of human decision strategies into recommendation systems research is a new and challenging field of research which requires a strongly interdisciplinary research approach [25].

*Recommendation Approaches.* We exemplified how conventional recommendation approaches can be exploited in RE scenarios. However, there are settings with complex interdependencies between requirements and a large number of inconsistent stakeholder preferences. These settings require to adapt, combine, and extend existing recommendation approaches. One possible direction is to adapt knowledge-based recommendation functionality for group-based recommendation scenarios, for example, critiquing-based recommendation approaches [48] have to be extended to support different types of group-based recommendation and diagnosis functionalities (for determining repair actions for inconsistent stakeholder preferences).

*Quality of Recommendations.* Stakeholders are often skeptical regarding a new form of automated tool support. As a consequence, recommendation technologies will only succeed if they deliver high-quality recommendations. To this end, we have to design and conduct empirical studies to (a) learn about stakeholder needs and (b) evaluate recommendation systems. The goal is to figure out how existing recommendation approaches have to be adapted for an optimal performance in RE scenarios. Empirical studies should deliver *grounded theories* about the behavior of stakeholders in particular situations, which are needed to train and optimize related recommendation algorithms.

*Social Networks in Recommender Systems.* The position of stakeholders in a social network often has an enormous impact on RE-related decision processes. Social network analysis is an important supportive technology for different types of recommenders. For example, collaborative filtering recommenders can use trust information to improve the quality of item predictions. Group-based recommenders can exploit trust information for determining group recommendations.

*Semiautomated Dependency Detection.* Effective dependency management is crucial for efficient requirements engineering processes. Existing recommendation support is focused on the analysis of similarities between requirements (using, e.g., clustering and content-based filtering methods). An important issue for future research is to make dependency detection more intelligent in terms of making it possible to predict, for example, the type of dependency (e.g., refinement or incompatibility dependency). Such new approaches can rely on concepts from the areas of natural language processing [32] and text mining [14].

*Requirements Discovery in Open Source Software Development.* Open-source platforms include different types of communication channels and types of communication. As a consequence, the filtering of requirement-relevant information is a challenge but a prerequisite for improving the quality of recommendation support. An issue for future research is the development of methods which allow to isolate requirement-relevant artifacts before recommendation algorithms are applied.

*Recommendation Beyond Textual Requirements.* Existing RE recommendation approaches focus on the analysis of textual requirements specifications which are represented, for example, in a completely informal fashion or in terms of use case scenarios. Future recommendation techniques for RE should be able to deal with graphical data sources such as, for example, class diagrams, sequence diagrams, state charts, as well as formal requirements specifications and system models. The inclusion and analysis of such artifacts has the potential to improve the prediction quality of recommendation algorithms and – as a consequence – also to improve the overall efficiency of RE processes.

*Context Awareness.* There are two basic recommendation modes: *pull* and *push*. *Pull* means that stakeholders are actively triggering recommendation functionality when needed. *Push* means that the recommender application proactively detects situations (contexts) in which a stakeholder needs a particular support [49]. In order to deliver push recommendations, the context of a stakeholder has to be observed and discovered [49]. An approach to tackle this challenge is to continuously collect users' context and reason about user needs. Contextual recommendation is an emerging field [50] and will also play a major role in the development of recommendation solutions for RE.

## 14.5 Conclusions

Due to the increasing size and complexity of software systems as well as the growing degree of distributedness in software projects, recommendation technologies are becoming more and more popular as an intelligent technology for requirements engineering (RE). In this chapter, we focused on a discussion of existing research related to the application of recommendation technologies in different requirements engineering scenarios. In order to demonstrate the application of recommendation technologies, we came up with examples such as the analysis of social networks for stakeholder identification and the clustering of requirements for the detection of dependencies. With our outlook on relevant topics for future research, we hope to stimulate fruitful further research focused on the development and application of recommendation technologies in RE.

**Acknowledgements** The work presented in this chapter has been conducted in the IntelliReq (829626) research project funded by the Austrian Research Promotion Agency.

## References

1. Hofmann H, Lehner F (2001) Requirements engineering as a success factor in software projects. IEEE Softw 18(4):58–66
2. Sommerville I (2007) Software engineering. Pearson, Munich
3. Felfernig A, Maalej W, Mandl M, Schubert M, Ricci F (2010) Recommendation and decision technologies for requirements engineering. In: ICSE 2010 workshop on recommender systems in software engineering, Cape Town, pp 1–5
4. Maalej W, Thurimella A (2009) Towards a research agenda for recommendation systems in requirements engineering. In: Proceedings of 2nd international workshop on managing requirements knowledge, Atlanta
5. Mobasher B, Cleland-Huang J (2011) Recommender systems in requirements engineering. AI Mag 32(3):81–89
6. Felfernig A, Burke R, Goeker M (2011) Recommender systems: an overview. AI Mag 32(3):13–18
7. Burke R (2000) Knowledge-based recommender systems. Encycl Libr Inf Syst 69(32):180–200
8. Burke R (2002) Hybrid recommender systems: survey and experiments. UMUAI J 12(4):331–370
9. Terveen L, Herlocker J, Konstan J, Riedl J (2004) Evaluating collaborative filtering recommender systems. ACM Trans Inf Syst 22(1):5–53
10. Linden G, Smith B, York J (2003) Amazon.com recommendations: item-to-item collaborative filtering. IEEE Inter Comput 7(1):76–80
11. Pazzani M, Billsus D (1997) Learning and revising user profiles: the identification of interesting web sites. Mach Learn 27:313–331
12. Felfernig A, Burke R (2008) Constraint-based recommender systems: technologies and research issues. In: Proceedings of IEEE ICEC'08, Innsbruck, pp 17–26
13. Konstan J, Miller B, Maltz D, Herlocker J, Gordon L, Riedl J (1997) Grouplens: applying collaborative filtering to usenet news full text. Commun ACM 40(3):77–87
14. Witten I, Frank E (2005) Data mining. Elsevier, San Francisco
15. Masthoff J (2004) Group modeling: selecting a sequence of television items to suit a group of viewers. UMUAI 14(1):37–85
16. Golbeck J (2009) Computing with social trust. Springer, London
17. Lim S, Quercia D, Finkelstein A (2010) Stakenet: using social networks to analyse the stakeholders of large-scale software projects. In: Proceedings of ACM/IEEE, Cape Town, pp 295–304

18. Castro-Herrera C, Duan C, Cleland-Huang J, Mobasher B (2008) Using data mining and recommender systems to facilitate large-scale, open, and inclusive requirements elicitation processes. In: Proceeding of the 16th IEEE international conference on requirements engineering (RE'08), Barcelona, pp 165–168
19. Dumitru H, Gibiec M, Hariri N, Cleland-Huang J, Mobasher B, Castro-Herrera C (2011) On-demand feature recommendations derived from mining public product descriptions. In: Proceedings of ACM/IEEE, Waikiki/Honolulu, pp 181–190
20. Lim S, Finkelstein A (2012) Stakerare: using social networks and collaborative filtering for large-scale requirements elicitation. IEEE Transactions on Software Engineering 38(3):707–735
21. Fitzgerald C, Letier E, Finkelstein A (2011) Early failure prediction in feature request management systems. In: 19th IEEE requirements engineering conference, Trento, pp 229–238
22. Cleland-Huang J, Dumitru H, Duan C, Castro-Herrera C (2009) Automated support for managing feature requests in open forums. Communications of the ACM 52(11):68–74
23. Felfernig A, Schubert M, Mandl M, Ghirardini P (2010) Diagnosing inconsistent requirements preferences in distributed software projects. In: Proceedings of 3rd International workshop on social software engineering, Paderborn, pp 1–8
24. Duan C, Laurent P, Cleland-Huang J, Kwiatkowski C (2009) Towards automated requirements prioritization and triage. Requir Eng 14(2):73–89
25. Felfernig A, Zehentner C, Ninaus G, Grabner H, Maalej W, Pagano D, Weninger L, Reinfrank F (2011) Group decision support for requirements negotiation. Springer Lect Notes Comput Sci 7138:1–12
26. Ruhe G, Eberlein A, Pfahl D (2003) Trade-off analysis for requirements selection. J Softw Eng Knowl Eng (IJSEKE) 13(4):354–366
27. Ruhe G, Saliu M (2005) The art and science of software release planning. IEEE Softw 22(6):47–53
28. Marczak S, Kwan I, Damian D (2007) Social networks in the study of collaboration in global software teams. In: Proceedings of ICGSE'07, Munich
29. Iyer J, Richards D (2004) Evaluation framework for tools that manage requirements inconsistency. In: 9th Australian workshop on requirements engineering, Adelaide, pp 1.1–1.8
30. Tsang E (1993) Foundations of constraint satisfaction. Academic, London
31. Reiter R (1987) A theory of diagnosis from first principles. AI J 23(1):57–95
32. Fantechi A, Spinicci E (2005) A content analysis technique for inconsistency detection in software requirements documents. In: WER05 – workshop em Engenharia de Requisitos, Porto, pp 245–256
33. Aurum A, Wohlin C (2003) The fundamental nature of requirements engineering activities as a decision-making process. Inf Soft Technol 45(14):945–954
34. Davis A (2003) The art of requirements triage. IEEE Comput 36(3):42–49
35. Schrijver A (1998) Theory of linear and integer programming. Wiley, New York
36. Felfernig A, Friedrich G, Schubert M, Mandl M, Mairitsch M, Teppan E (2009) Plausible repairs for inconsistent requirements. In: Proceedings of IJCAI'09, Pasadena, pp 791–796
37. McFadden D (1999) Rationality for economists. J Risk Uncertain 19(1):73–105
38. Bettman J, Luce M, Payne J (1998) Constructive consumer choice. J Consum Res 25(3):187–217
39. Lichtenstein S, Slovic P (2006) The construction of preference. Cambridge University Press, New York
40. Masthoff J (2011) Group recommender systems. In: Recommender systems handbook. Springer, Boston, pp 677–702
41. Jameson A, Baldes S, Kleinbauer T (2004) Two methods for enhancing mutual awareness in a group recommender system. In: ACM international working conference on advanced visual interfaces, Gallipoli, pp 48–54
42. Roy L, Mooney R (2004) Content-based book recommending using learning for text categorization. User Model User-Adapt Interact 14(1):37–85

43. Can F, Ozkarahan A (1990) Concepts and effectiveness of the clustering methodology for text databases. ACM Trans Database Syst 15(4):483–517
44. Lohmann S, Riechert T, Auer S (2008) Collaborative development of knowledge bases in distributed requirements elicitation. In: Software engineering (workshops): agile knowledge sharing for distributed software teams, Munich, pp 22–28
45. Junker U (2004) Quickxplain: preferred explanations and relaxations for over-constrained problems. In: Proceedings of 19th national conference on AI (AAAI04), San Jose, pp 167–172
46. Felfernig A, Schubert M, Mandl M, Ghirardini P (2010) Diagnosing inconsistent requirements preferences in distributed software projects. In: Proceedings of 3rd international workshop on social software engineering, Paderborn, pp 1–8
47. Felfernig A, Chen L, Mandl M (2005) Recsys'11 workshop on human decision making in recommender systems, Chicago, pp 389–390
48. Burke R, Felfernig A, Goeker M (2011) Recommender systems – an overview. AI Mag 32(3):13–18
49. Hans-Jörg H, Maalej W (2008) Potentials and challenges of recommendation systems for software development. In: RSSE '08: proceedings of the 2008 international workshop on recommendation systems for software engineering, ACM, Atlanta
50. Anand S, Mobasher B (2007) Contextual recommendation. In: Discovering and deploying user and content profiles, Springer Berlin/Heidelberg, pp 142–160

# Chapter 15
# Experience-Based Requirements Engineering Tools

**E. Knauss and S. Meyer**

**Abstract** Writing a good software requirement specification is a complex task. Many different aspects must be taken into account; most of them can only be learned through experience. Being aware of experiences and distilled best practices at the right time when writing a specification is another challenge. Experience-based requirements engineering tools make sharing and reuse of experience feasible. In this chapter, we present design principles for such tools, define a learning model to describe how organisations and individuals can learn new experiences by using them, and sketch a strategy for evaluating experience-based requirements engineering tools. We highlight these concepts with an example.

## 15.1 Introduction

Authors of software requirement specifications (SRS) have to adopt the increasing complexity of today's software systems. This leads to more comprehensive and complicated requirements. In these situations, authors find it increasingly difficult to avoid ambiguities and contradictions in order to create a high-quality SRS.

There are many guidelines and best practices about how to create a good specification. Authors are expected to know them and to be able to apply them when needed. This leads to an information overflow, especially if the author is not a full-time requirements analyst. Furthermore, writing good requirements needs a lot of experience. Gaining and sharing experiences in writing SRS is a challenging task. In general, software developing organisations try to address these challenges by systematic knowledge and experience management [1], thus becoming Learning Software Organisations (LSO) [2]. However, there is limited support for organisational learning in requirements engineering.

E. Knauss (✉) • S. Meyer
Leibniz Universität Hannover, Hanover, Germany
e-mail: eric.knauss@inf.uni-hannover.de; sebastian.meyer@inf.uni-hannover.de

When trying to support analysts in this situation, one has to face two problems that are intertwined with each other: On the one hand, an author of a SRS is not aware of guidelines or best practices that could help to create a good specification. On the other hand, experienced requirements engineers that know these best practices do not know how to share them with others in a form that supports efficient reuse.

In this chapter, we discuss how tools can address these two problems and offer experience-based support. Based on a learning model, we address the following research question:

> Can experience-based tools support learning requirements engineering and do they lead to better requirements documentation?

We describe important design principles for such tools in Sect. 15.2 and illustrate them at the hand of the example of our HeRA tool (Sect. 15.3). In Sect. 15.4, we relate other works to these concepts. In addition, we discuss how our concepts can be applied and evaluated in Sect. 15.5. Our concepts are specific to requirements engineering activities in allowing dealing with inconsistent and incomplete documents with limited structure and formality, that is, the kind of documents typically encountered during the requirements analysis.

## 15.2 Design Principles

This section gives design principles for requirements engineering tools based on principles of experience and knowledge management. Based on Schneider, we define experience as a specific kind of knowledge being acquired by a person by being involved [1]:

**Definition 1.** *Experience* (*Schneider* [1]). An experience is defined as a three-tuple consisting of:

(a) An observation
(b) An emotion (with respect to the observed event)
(c) A conclusion or hypothesis (derived from the observed event and emotion)

We call a tool that supports requirements engineering based on externalised experience an *experience-based requirements tool.*

**Definition 2.** *Experience-based requirements tool.* An experience-based requirements tool supports one or more of the following aspects of organisational learning during requirements engineering activities:

(a) *Learning*: the creation of new experience
(b) *Evaluation*: the evaluation of existing experience
(c) *Management*: the distribution, updating, and refinement of existing experience
(d) *Application*: the usage of existing experience in a given context

An example of a simple experience-based requirements engineering tool is a tool allowing tailoring (and *manage*) requirements templates. Based on experience, the template can be improved. A better template (i.e. tailored to a specific project context) supports requirements engineering. More sophisticated examples include automatic requirements checkers, because they support the *application* of existing experience about quality problems that should be removed from requirements documentation.

Often, the discussion of tools that automatically analyse requirements documentation is limited to the discussion of their recall and precision (e.g. in [3]). Here we give a broader model for requirements analysis tools allowing us to describe their usefulness for supporting continuous improvement and organisational learning in requirements engineering activities.

Kiyavitskaya et al. argue that tools used for identifying problems in requirements documentation should have 100 % recall, that is, they should find all problematic requirements [3]. Only then, requirements analysts can focus on reading only the problematic requirements returned by the tool. With lower recall, analysts have to read all requirements again to find all problems. In addition, many false positives have a similar effect: If the tool reports almost every requirement to be problematic, the analyst does not gain much. Thus, a high precision is also important. We agree that recall and precision are important properties and subsume them in the property *reliability*. Note that even low reliability might be acceptable, if other properties of the tool add enough value.

**Definition 3.** *Reliability*. The reliability of a requirements checking tool is defined inversely proportional to the number of *type one errors* (the checking tool reports a problem, but there is none; metric: precision) and *type two errors* (the checking tool reports no problem, but there is one; metric: recall). The reliability can be:

(a) *Low:* Recommendations of tool are often incorrect.
(b) *Medium:* F-measure $> 0.7$ (f-measure is a combined metric based on the harmonic mean of precision and recall).
(c) *High:* F-measure $> 0.85$.

Especially, when focusing on learning aspects of such tools, it is important to switch the perspective and analyse how such tools affect the work of requirements analysts. For this, we add another definition: *authority* of tools.

**Definition 4.** *Authority*. The degree to that users rely on feedback of automatic requirements checkers. The authority can be:

(a) *Low*: Recommendations of tool are seldom adhered to.
(b) *Medium:* Recommendations of tool are more often adhered to than not.
(c) *High:* Recommendations of tool are mandatory.

This definition has two facets:

1. In an extreme case, the requirements engineering workflow of an organisation could dictate analysts to react on all findings from an experience-based requirements engineering tool, no matter whether the finding stands for an actual problem or not. This makes sense, if changing 100 requirements without problems

is cheaper than missing one requirement with a problem. We conclude: *Authority* can be independent from *reliability*.
2. Otherwise, the experience-based tool has to earn authority by being useful. We conclude: *Reliability* (as expressed by precision and recall) has a strong impact on usefulness.

Note that inexperienced users could rely on bad feedback. Therefore, it is important to distinguish between the concepts of reliability and authority. Beyond these concepts, the usefulness of computer-based feedback is strongly influenced by the *proactivity* and by the *degree of interpretation*.

**Definition 5.** *Proactivity*. The degree to that an automatic requirements checker defines the time of feedback. The proactivity can be:

(a) *Low (=reactive):* Feedback is only given if the user requests it.
(b) *Medium:* Feedback is triggered by the user's specific actions.
(c) *High:* Time of feedback is determined based on more complex rules.

*Proactivity* defines the trigger of the feedback. Often, it is easier to directly improve a problematic requirement. When using a reactive tool, the analyst needs to understand that feedback will be useful in a given situation. Because of the high time pressure during requirements analysis, this often leads to the problem that feedback is only given very late. Then, it is much more difficult to repair problematic requirements: Other requirements might depend on the problematic one. In addition, the analyst needs to understand the exact circumstances of a given requirement.

**Definition 6.** *Degree of interpretation.* The degree to that an automatic requirements checker gathers and interprets data about a given situation beyond concrete available data. The degree of interpretation can be:

(a) *Low:* The tool reproduces existing data and lets the user interpret it.
(b) *Medium:* The tool sorts and filters data, thus suggesting a specific interpretation.
(c) *High:* The tool refines and interprets data.

Finally, an important property of experience-based tools is the ability to learn.

**Definition 7.** *Learning ability*. The ability to integrate new experiences or refine existing experiences in the experience-based requirements tool. Learning ability can be:

(a) *Low:* New experience has to be hard coded into the tool by experts.
(b) *Medium:* Special facilities exist in the tool to allow users to add new experience.
(c) *High:* The tool can automatically adopt, for example, based on observing its user.

The learning ability is important to allow adopting the experience-based tool to a specific situation. Furthermore, the learning ability is the primary facility to support continuous improvement and organisational learning: If new knowledge is encoded into the experience-based tool, the organisation owning the experience-based tool has learnt.

In requirements engineering, we encounter a large variety of document types. We aim to define our design principles independent from the specific document type, but of course the properties of documents handled by a tool have a certain impact on the performance of the tool. For our purposes, it is sufficient to distinguish two properties of requirements documents: (a) *formality* and (b) *maturity*. The more formal the syntax of a requirements document (or model) is, the easier it is to analyse for requirements checkers. Formal requirements models even allow simulation and other mechanisms for quality insurance. However, in many projects, requirements are specified without too much formalism, especially in the beginning of a project. Experience-based tools that offer support early during requirements engineering activities should be able to handle informal requirements descriptions as well as incomplete documents (with low maturity).

### *15.2.1 Learning Through Experience: Heuristic Critiques*

In this section, we describe how the presented design principles affect learning and experience management. First, we introduce the concept of *heuristic critiques*:

**Definition 8.** *Heuristic critique*. Computer based feedback to an activity or work product (e.g. requirements documentation) based on experience. A heuristic critique consists of:

(a) A *heuristic rule* that can be evaluated by a computer
(b) A *criticality*
(c) A meaningful and constructive *message*

A heuristic critique depicts a single automatic requirements check. Furthermore, it supports Learning Software Organisations (LSO). Such an LSO focuses on developing software while supporting the following aspects of organisational learning [1]:

(a) Learning of the individuals in the organisation
(b) Organisation-wide collection of knowledge and experience
(c) Cultivation of an organisation-wide infrastructure for exchanging knowledge and experience

When integrated in requirements engineering tools, heuristic critiques offer support for a LSO. Table 15.1 gives an example of the relation between heuristic critique and experience (see Definition 1). While implementing a requirement, a developer makes an experience (left hand of Table 15.1). A heuristic could be created that covers part of this experience (right hand of Table 15.1).

The heuristic rule gives a good solution without guarantee for optimality or feasibility. In the example, passive voice can be found by computers and often (not always) leads to the detection of unclear responsibilities. Examples of other heuristic critiques include:

- *Inconsistencies:* If two similar requirements (e.g. use cases with similar title) are detected, give a warning and suggest that the user merges both requirements or clarifies them to avoid inconsistencies.

**Table 15.1** Example: a heuristic critique encodes an experience (cf. [33])

| *Part of exp.* | Experience | Heuristic critique | *Part of heur. crit.* |
|---|---|---|---|
| (*i*) *Observation* | Req. was misunderstood, because we had not specified who was responsible | If passive voice is detected… | (*a*) *Heuristic rule* |
| (*ii*) *Emotion* | It took a week to rework the module – just for sloppy writing! | …give a warning | (*b*) *Notion of severity* |
| (*iii*) *Conclusion/ hypothesis* | It should always be spelled out who is responsible for an action. Avoid passive voice! | …asking the user to use active voice and state responsibility | (*c*) *Meaningful and constructive msg.* |

- *Ambiguities:* If a weak word (e.g. always, sometimes) is encountered, give a warning and ask the user to specify the circumstances of the requirement more exactly.
- *Incompleteness:* If a use case on user goal level is found that is not included or does not extend a use case on business level, give a warning and ask the user to add the missing link or to specify the missing business goal.

## *15.2.2 Learning Model*

Figure 15.1 shows two important areas of learning, supported by heuristic critiques. Learning occurs on individual and organisational level during the activity of writing requirements.

### 15.2.2.1 Individual Learning: Reflect and Apply

Under pressure, analysts write bad requirements, even if they know how to do it better. It is good to have a mentor who gives gentle reminders until knowledge develops to skills. Heuristic critiques like in Table 15.1 could give just this type of feedback.

*Reflect*: If a heuristic detects passive voice in a requirement and fires a warning, the requirements engineer is interrupted in his task. This gives him the chance to reflect about the action currently taken out; a breakdown occurs [4, 5]. This facilitates learning through reflection, if evaluated directly on the requirements engineers' input. Note that we depict this information flow as irreproducible, because the reflexion depends not only on the heuristic critique but also on the mental context and state of mind of the analyst.

*Apply*: The requirements engineer might already know how to write good requirements in general. Nevertheless, passive sentences may slip into a specification

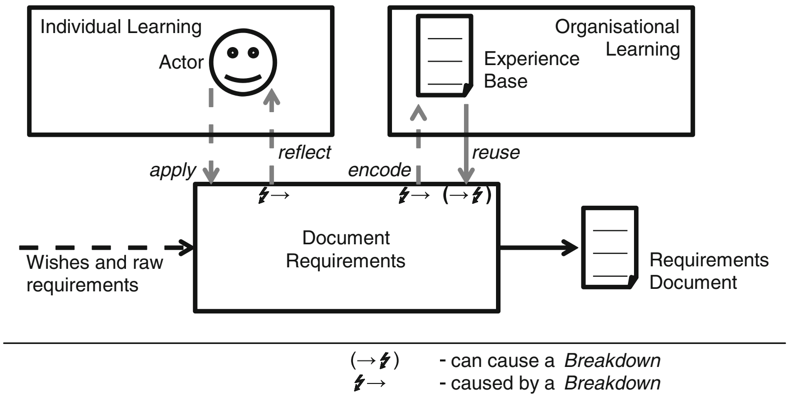


**Fig. 15.1** Learning model (cf. [32, 33]): information flows (*arrows*; *dashed*, irreproducible; *grey*, experiences) are the foundation of *individual* and *organisational learning*. *Heuristic critiques* can stimulate the information flows by causing *breakdowns*

during periods of intense writing. Reminders and warnings help to apply and repeat the knowledge. Thus, they support turning abstract knowledge into internalised skills and help writing good requirements. Again, this information flow is irreproducible, as the analyst might choose different solutions, even if he makes the same mistake again.

#### 15.2.2.2 Organisational Learning: Reuse and Encode

Heuristic critiques allow codifying experiences in a useful way.

*Reuse*: Based on heuristics, computers can be used to find situations matching the observation that led to the original experience. A more or less disruptive message points to potential improvements that are inferred from reported emotions.

*Encode*: Heuristic warnings are not always correct, for example, an actor could be specified even in a passive sentence. Furthermore, they are not always applicable. For a condition, which is stated in requirements documentation, use of passive voice is unproblematic. If such a situation is observed during a breakdown, the requirements engineer can refine the heuristic warning and specify that it should not be applied to conditions. Thus, experience is added to the organisations' knowledge base. As a by-product, the growing body of codified experience adopts a manageable granularity for a LSO's knowledge base. This information flow is irreproducible, as solutions differ based on the context that leaded to the encoding of the new experience.

### *15.2.3 Research Questions*

We derive our research questions from the learning model in Fig. 15.1. Each (group of) experience and knowledge flow leads to a specific question:

- *RQ 1*. Can experience-based tools support individual learning? We need to investigate the impact on individuals (*a) applying* and (*b) reflecting* as depicted in Fig. 15.1.

- *RQ 2*. Can experience-based tools support organisational learning? We need to investigate organisation-wide (*a*) *reuse* and (*b*) *encoding* of experience (c.f. Fig. 15.1.).
- *RQ 3*. Do experience-based tools lead to better requirements documentation? We need to investigate if (*a*) *quality* and (*b*) *costs* of requirements documentation are improved with experience-based tools.

## 15.3 Example: The Heuristic Requirements Assistant (HeRA)

This section gives an example of how to integrate the described design principles into an experience-based requirements engineering tool. While we demonstrate the principles and their implementation on our own HeRA tool, there are of course other tools. We give a short overview of them at the end of this section.

HeRA is our implementation of an experience-based requirements engineering tool. It is basically a smart requirements editor with several heuristic feedback facilities [6]. While HeRA is designed to be extensible, we focus in HeRA's original critique system for describing the key concepts.

HeRA is based on Fischer's architecture for domain-oriented design environments (DODE) [5]. The central part of this architecture is a construction component. In the case of HeRA, requirements are "constructed" using a general-purpose requirements editor, a use case editor, and a glossary editor. These editors allow constructing specific artefacts (i.e. requirements, use cases, and a glossary). HeRA offers two other DODE components, namely, the argumentation component and the simulation component. Figure 15.2 shows the structure of the HeRA tool.

We describe the interplay of construction, argumentation, and simulation components based on use cases. Use cases are used to describe the interactions of a user with the system to construct (based on templates suggested by Cockburn [7]). HeRA was designed to support the requirements engineer with heuristic feedback: It analyses the input and warns the user if it detects ambiguities or incomplete specifications. Furthermore, HeRA can generate diagrams like use case diagrams from the Unified Modelling Language (UML) or event-driven process chain (EPC) models that show how the current user goals relate to the business goal or the global process. If needed, a glossary assistant can be used to ensure consistent use of important terms. On demand, HeRA computes use case points and displays an effort estimation associated with the use cases. Use case points are calculated based on the actors and transaction between them. The HeRA module implements the use case points method as described by Kusumoto et al. [8]. All of these perspectives are derived while the use case is being written. In this way, the author gets immediate feedback on the input and may improve it. We have successfully applied HeRA during interviews and workshops (see Sect. 5). Supported by HeRA's feedback facilities, we aim to achieve the following:

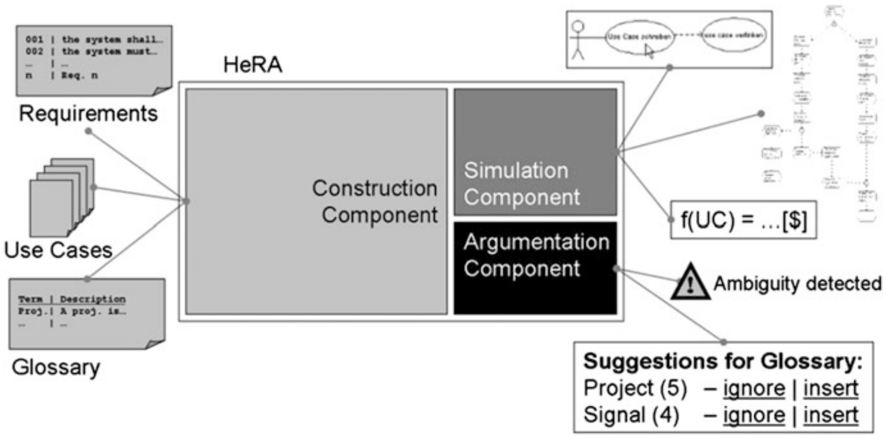


**Fig. 15.2** Structure of the HeRA tool

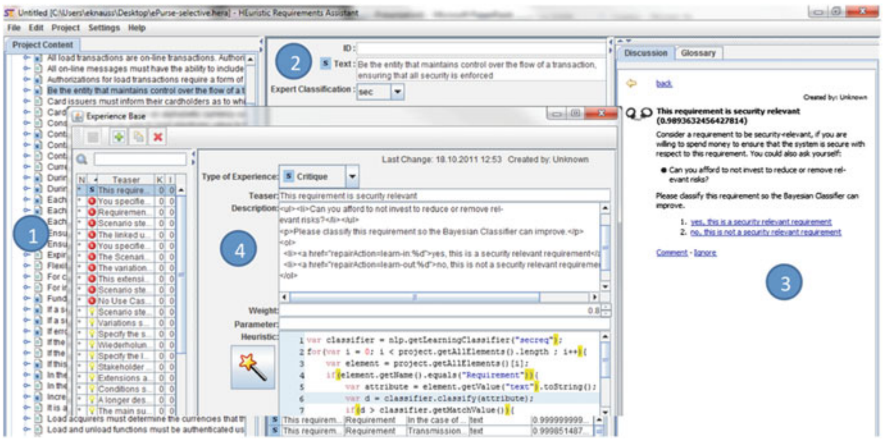
1. Elicitation of user goals on a level of detail that allows for the identification of conflicts
2. Discussion whether the current user goals fit into the underlying business goal and the other user goals already documented (based on visualisation as UML use case and process models)
3. Discussion of important terms and identification of conflicting interpretations of these terms
4. Discussion of prioritisation and project constraints based on the use case points

The direct feedback of HeRA allows starting with elicitation of user goals and detection of inconsistencies and conflicts very early by applying computer-generated feedback.

HeRA's capability of applying direct feedback while writing the document is achieved by automatically evaluating heuristic critiques in the background, whenever a user works on a use case in HeRA. If a heuristic rule fires, the message is displayed besides the use case form and a warning symbol is displayed at the specific field of the use case where the potential problem was detected (see Fig. 15.3, points (1) and (2)).

Additionally, the argumentation component shows a short description for each fired heuristic rule. The user can decide to get a more detailed description for the experience that is encoded through this rule (see Fig. 15.3, point (3)). He can comment on this rule. This feedback can be used to re-evaluate the heuristics and maybe fine-tune them by hand. The user can also decide to ignore this rule if he finds it inappropriate for his current document.

Heuristic critiques can be directly changed in HeRA (see Fig. 15.3, point (4)). This allows rapid prototyping of new critiques. All users can change the message of the critique or parameters (e.g. keyword lists). In addition, the heuristic rule can be adjusted. This rule is encoded in JavaScript. In the scope of the script, all use cases written in HeRA can be accessed.



Text: Be the entity that maintains control over the flow of a transaction, ensuring that all security is enforced

Expert Classification : sec

Last Change: 18.10.2011 12:53 Created by: Unknown

Type of Experience: Critique

Teaser: This requirement is security relevant

Description:
```
<ul><li>Can you afford to not invest to reduce or remove rel-
evant risks?</li></ul>
<p>Please classify this requirement so the Bayesian Classifier can improve.</p>
<ol>
  <li><a href="repairAction=learn-in:%d">yes, this is a security relevant requirement</
  <li><a href="repairAction=learn-out:%d">no, this is not a security relevant requireme
</ol>
```

Weight: 0.8

Parameter:

Heuristic:
```
var classifier = nlp.getLearningClassifier("secreq");
for(var i = 0; i < project.getAllElements().length ; i++){
    var element = project.getAllElements()[i];
    if(element.getName().equals("Requirement")){
        var attribute = element.getValue("text").toString();
        var d = classifier.classify(attribute);
        if(d > classifier.getMatchValue()){
```

back

Created by: Unknown

**This requirement is security relevant (0.9893632456427814)**

Consider a requirement to be security-relevant, if you are willing to spend money to ensure that the system is secure with respect to this requirement. You could also ask yourself:

- Can you afford to not invest to reduce or remove relevant risks?

Please classify this requirement so the Bayesian Classifier can improve.

1. yes, this is a security relevant requirement
2. no, this is not a security relevant requirement

Comment - Ignore

**Fig. 15.3** The HeRA critique environment

## 15.3.1 Argumentation Component: Glossaries

Glossaries are widely accepted as a method to define a common ground for communication in software projects. Their goal is to define terms used in the project-specific domain and gain a common understanding. The usefulness of glossaries has often been reported [9, 10].

We often observe that complicated terms are gladly introduced into the glossary of a given project. However, problems arise when common terms are used differently in the context of a project. An example for this is the term *application*. Most developers think they know how the term *application* is defined (according to Wikipedia as *computer software designed to help the user to perform specific tasks*). However, in a given software project, this term was used as "*the task of a student applying for his thesis*".

While this is of course a domain-specific definition of the term at hand, this scenario is especially dangerous, because developers and customers think they have a common understanding about a given term. Customers know the domain-specific definition of this term for their domain, while developers might just know the common definition.

The problem is that neither customers nor developers are aware of these conflicting interpretations of terms. Therefore, they cannot be expected to solve it without help. The only way of solving this hidden ambiguity is to confront both parties with the inconsistency of their assumed meanings.

In the context of experience-based RE tools, there are two possible ways in which the tool can help. Firstly, it can help to build the glossary by suggesting terms to put into it, and it can help maintaining the glossary by giving feedback on its quality. Secondly, it can increase the awareness that there is a glossary and which terms are defined in it. This is important since a reader may not look up these terms because he is not aware of the conflict.

#### 15.3.1.1 Suggesting Terms for a Glossary

To recommend potentially interesting words to the requirements author, we have to define what makes a term interesting. For a glossary tool in the context of experience-based RE tools, we identified two useful heuristics for identifying terms that should be suggested:

1. *Occurrence matters*: A term that is used frequently is probably relevant to the glossary because it can be misused more often.
2. *Experience matters*: If a term was added to a glossary in another project in the same domain, it probably should be defined in the current glossary, too.

There exists empirical evidence that people write important terms more often in order to clarify them or stress their importance [11]. This makes the occurrence heuristic – although simple – an important and strong method to identify terms that are important for a project. These terms are candidates for being added to a glossary.

However, simply counting the occurrence of each word does not lead to satisfying results: A term can be used in different forms over a document. If these terms were not normalised, the algorithm would suggest both spellings separately. For example, the terms project and projects have the same meaning, but if they are compared literally, they do not match. Hence, the algorithm would list each of them with an occurrence count of 1. Therefore, each term has first to be converted into a defined base form. Methods for this are stemming (e.g. Porter stemming [12]) or lemmatisation.

Because SRS are often written in natural language, each term in the glossary should be a correct term according to grammar rules. This requirement makes stemming unusable for getting a base form and leaves us with lemmatisation. We use a modified spellchecking engine together with the OpenOffice.org[1] dictionaries to get base forms of inflected words.

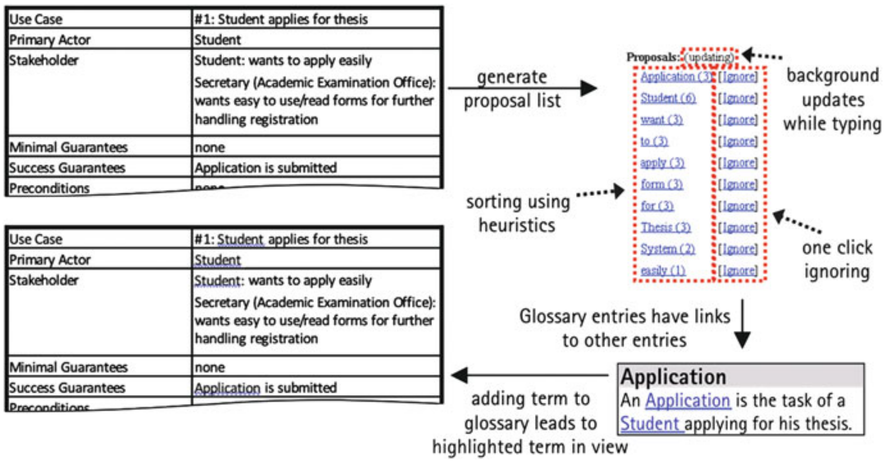
This yields another favourable effect: Since the lemmatisation is language-agnostic, the SRS can be written in every language for which a dictionary exists. We can even have more than one dictionary activated simultaneously. This is, for example, useful for non-English texts containing technical terms in English.

Having solved the problem of different forms of the same term strengthens the occurrence count heuristic. In the above example, project and projects would be suggested as project with an occurrence count of 2.

The second heuristic (*experience matters*) works by remembering the terms, which have already been added to a glossary in another project in the same domain. The experience circle has an "activate" step. This is where tacit knowledge is seeded into the experience circle. The most difficult part is recognising the terms to put into the glossary. Afterwards we can profit from previous experiences and suggest those terms again whenever they are used.

Therefore, a term, which has already been added to a glossary, is not tacit any longer but has successfully been identified to be added to a glossary. Such a term is

[1] http://www.openoffice.org.



| | |
|---|---|
| Use Case | #1: Student applies for thesis |
| Primary Actor | Student |
| Stakeholder | Student: wants to apply easily<br>Secretary (Academic Examination Office): wants easy to use/read forms for further handling registration |
| Minimal Guarantees | none |
| Success Guarantees | Application is submitted |
| Preconditions | none |

Proposals: (updating)

| | |
|---|---|
| Application (3) | [Ignore] |
| Student (6) | [Ignore] |
| want (3) | [Ignore] |
| to (3) | [Ignore] |
| apply (3) | [Ignore] |
| form (3) | [Ignore] |
| for (3) | [Ignore] |
| Thesis (3) | [Ignore] |
| System (2) | [Ignore] |
| easily (1) | [Ignore] |

**Application**

An Application is the task of a Student applying for his thesis.

generate proposal list
background updates while typing
sorting using heuristics
one click ignoring
Glossary entries have links to other entries
adding term to glossary leads to highlighted term in view

**Fig. 15.4** The overall process of working with glossaries in HeRA

more valuable than a term that is frequently used. Thus, we put those terms on top of the suggestion list, regardless if there are other terms that are used more often, but have not been added to a glossary before.

After identifying the terms that qualify for recommendation, they are shown to the requirements author. For the specification to be helpful, it is necessary to limit the number of simultaneous recommendations. If there are too many recommendations, the user is rather distracted than supported. So we decided to initially show only the ten terms with the highest priority.

Not all words that have been identified as possible suggestions through the two heuristics should be presented to the author. For example, words like *and* or *that* can be filtered out through a stopword list. Furthermore, the author may decide to ignore some words because he does not want them to be part of the glossary, and therefore, they should no longer be suggested. For this case, another filter list is maintained, consisting of all the ignored words. This list is editable by the author. Each author maintains his own list of ignored words. Since they are only plain text files, they can be easily reused or exchanged between different users. Further work may combine creation, maintenance, and exchange of ignored words with social elements, to allow sharing between different users of the same knowledge base. Figure 15.4 shows the overall process of working with glossaries in HeRA.

### 15.3.1.2 Awareness of Defined Terms

As discussed earlier, the crucial step is to identify ambiguity. This is important both for the requirements author and for anyone who reads the SRS. However, there is one big difference: Since the author has already added the term to the glossary, the former implicit and tacit knowledge about his specific understanding has been explicitly defined. So it is possible to share this special knowledge. That means a

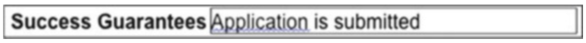

Success Guarantees Application is submitted

**Fig. 15.5** Creating awareness for terms in the glossary

reader does not have to bother whether a term is ambiguous or not because he or she can look it up in the glossary.

However, most readers do not look up critical words in the glossary. Either because it stops their natural flow of reading or because they believe a term is not ambiguous. This is the same problem the author faces when writing the document. Consequently the reader has to be supported in identifying ambiguous terms.

We use highlighting to mark terms defined in the glossary. This is a common approach that is, for example, used in almost every modern word processor for spellchecking. This method does not need much space in the text input area but can still easily be recognised by a user. We noticed that we cannot use red as the highlighting colour since this may let the user think of a spelling error.

The use of highlighting makes the recognition of defined terms very easy. The reader is made aware of an existing definition for a term without having to look it up. Therefore, the natural reading flow is not interrupted. This enhances the awareness of the reader since he notices that a word is in the glossary while looking at its context. This would not be given if he looked up the word in the glossary by himself. To further foster the context awareness, tooltips can be used to show the definition of the term. Figure 15.5 shows an underlined term (*Application*) in the HeRA use case form.

## *15.3.2 More Heuristic Feedback*

Glossaries are not the only part of HeRA that takes advantage of the heuristic feedback mechanism. Other examples are Cockburn's aforementioned rules for writing effective use cases that can also be encoded as heuristic critiques.

A more sophisticated example for applying heuristic rules constructively during the creation of a SRS is the SecReq heuristics [13]. These heuristics support the identification of security-relevant requirements. We first created a body of knowledge from three industrial-level specifications. For two of them, we classified the requirements with the help of security experts; for the third, we made use of an already existing requirements database. We then used this body of knowledge to train a Bayesian classifier to recognise security-relevant requirements. The Bayesian classifier was added to HeRA's critique mechanism. This allows HeRA's argumentation component to suggest security-relevant requirements that need further refinement. In addition, the classifier can be trained further by adding information about falsely classified requirements, similar to email spam filters (see Fig. 15.3).

This is an example of a more sophisticated mechanism. In our experience, it is important to have a high learning ability in a tool. This can be achieved by either having very easy mechanisms that can be adjusted by the user (e.g. heuristic JavaScript rules) or by sophisticated algorithms with an easy-to-use interface.

## 15.4 Overview of Related Tools

Requirements are often specified using natural language, if only as an intermediate solution before formal modelling. As natural language is inherently ambiguous [14], several approaches have been proposed to automatically analyse natural language requirements in order to support requirements engineers in creating good requirements specifications [15–19]. Typically, such approaches define a specific quality model first. Then indicators are defined for the quality aspects that can be automatically evaluated. A good example for this approach is the ARM tool by Wilson et al. [15]. Often these indicators are based on simple mechanisms, for example, keyword lists. Newer approaches leverage sophisticated analysis of natural language, for example, the search for under specification in the QuARS tool [20]. Fabbrini et al. report that the QuARS tool can effectively assess the quality of requirements documentation [20]. Fantechi et al. have applied both the QuARS and the ARMS tool on use cases [21]. They report that use cases are especially well suited for automatic analysis because of their structure. Melchisedech's work goes beyond these approaches by using input from specific workflow and information models for the automatic checks in the ADMIRE tool [22]. This allows for a closer investigation of relationships between requirements but renders the approach only useful if a given workflow is applied.

Somé describes a method and a tool (UCed) to systematically create use cases [23]. The method prescribes a strict use case metamodel and a grammar for natural language descriptions. By constricting natural language to a formalised subset, certain ambiguities and inconsistencies can be avoided.

Jang proposes a formal language for specifying requirements that is founded on mechanisms from knowledge management [24]. The language has similarities to Prolog and allows to easily checking even complex relationships between conditions. Hunter and Nuseibeh propose a related approach that also documents and checks requirements based on logical expressions [25]. They extend the classical logic to allow automatic reasoning on inconsistent requirements descriptions. Such formal and logical languages have advantages in automatic checking but disadvantages in the usability of writing and reading requirements. Thus, Gervasi et al. describe how to transform natural language requirements into a logical representation for verification [26].

Berenbach proposes an approach for automatic checks of UML models, based on heuristics that create an analysis model [27]. This analysis model allows checking, whether UML requirements models have sufficient quality (e.g. for verification or for automatic requirements extraction). The heuristics can be used interactively by the modeller or analytically by the quality assurance. Souza et al. propose to use critique systems to check software engineering models for inconsistencies [28]. Accordingly, such critique systems allow efficient and scalable consistency checking because of the rather small and local critiques.

Kof, Lee et al. work on extracting semantics from natural language texts [16, 17] by focusing on the semiautomatic extraction of an ontology from a requirements document. Their goal is identifying ambiguities in requirements specifications. Gleich et al. present

**Table 15.2** Adherence of experience-based requirement tool to design principles

| Name | Authority | Proactivity | DEG of interpr. | Learning ability |
|---|---|---|---|---|
| HeRA | ●●●○○ | ●●●●● | ●●●●○ | ●●●○○ |
| ARM [15] | ●●●●○ | ○○○○○ | ●●●○○ | ●●○○○ |
| QuARS [20] | ●●●●○ | ○○○○○ | ●●●○○ | ○○○○○ |
| ADMIRE [22] | ●●●●○ | ○○○○○ | ●●●●○ | ○○○○○ |
| Chantree [18] | ●●○○○ | ●○○○○ | ●●●●● | ●●●●○ |
| Kiyavitskaya et al. [3] | ●○○○○ | ○○○○○ | ●●●●● | ○○○○○ |
| Jang [24] | ●●●●● | ○○○○○ | ●●○○○ | ○○○○○ |
| Hunter [25] | ●●●●● | ○○○○○ | ●●○○○ | ○○○○○ |
| Gervasi et al. [26] | ●●●●● | ○○○○○ | ●●●○○ | ○○○○○ |
| Berenbach [27] | ●●●○○ | ●●●○○ | ●●●○○ | ○○○○○ |
| Souza et al. [28] | ●●●○○ | ●●●○○ | ●●○○○ | ●●○○○ |
| UCed [23] | ●●●●○ | ●○○○○ | ●●●●○ | ○○○○○ |

a tool that is able to detect a comprehensive set of ambiguities in natural language requirements [19].

Chantree et al. describe how to detect noxious ambiguities in natural language requirements [18] by using word distribution in requirements to train heuristic classifiers (i.e. how to interpret the conjunctions *and*/*or* in natural language). The reported results (recall = 0.587, precision = 0.71) are useful in the described context but are too low for more generic approaches as discussed by Kiyavitskaya et al. as discussed in Sect. 15.2.

Table 15.2 gives an assessment of related experience-based requirements tools based on the properties defined in Sect. 15.2. Based on this assessment, related work seems to concentrate on authority and degree of interpretation, thus giving sophisticated and reliable feedback. In contrast, learning ability and proactivity seem to be underrepresented. Because of the impact of these properties on individual and organisational learning, we consider this to be a gap in research that should be closed.

We feel supported in this by Gervasi's discussion on why ambiguity is not always bad [29]. He argues that our language has evolved to cope with uncertainty and missing knowledge. Thus, people are able to articulate this missing knowledge in ways that are then identified as ambiguities. Removing these ambiguities can only be beneficial, if the underlying uncertainty is removed.

Further, Adam et al. propose to approach requirements engineering in a domain-specific way, thus capturing the peculiarities of the domain as soon as possible [30]. The rationale is that for a given domain, the creation of a solution depends on domain-specific information needs. We conclude from this that generic rules have their limitations and experience-based tools, which can be adapted to a specific domain, are a relevant concept.

## 15.5 Evaluating Experience-Based RE Tools

In Sect. 15.2, we presented concepts for RE tools that go beyond the current state of automatic requirements checking by introducing a learning model. In this section, we discuss strategies for evaluating these experience-based RE tools, based on the research questions we introduced in Sect. 15.2.3. We do this by sketching the evaluation of HeRA, showing that these concepts hold, that the learning model is valid, and that experience-based requirements tools are feasible and beneficial. For details, we refer to other work.

### *15.5.1 Research Method*

For investigating the research questions, we apply the following methods: First, we investigate related work for relevant evidence. Then we find or create a suitable exemplary implementation of an experience-based requirements tool (e.g. HeRA). We let typical users use this exemplary implementation, either in a case study or in an experimental setting. Finally, we complement the observation by using questionnaires to tackle individual learning of these users.

### *15.5.2 Sketch of Evidence*

Here we will give a rough sketch of our evidence that experience-based requirements tools are feasible and beneficial.

*Step 1*. Existing automatic requirements checkers can be regarded as experience-based requirements tools, because they incorporate experience about typical problems in requirements documentation.

*Step 2*. Related work (Sect. 3.4) shows that such tools are feasible (as they exist) and beneficial: Some tools are able to cut costs for quality assurance (RQ 3.b); others help to create better requirements documentation (RQ 3.a). We can support RQ 3.a with empirical evidence that students create better requirements documentation with the HeRA (see [31]).

*Step 3*. We can conclude from this that users improved the documentation based on the heuristic feedback. Thus, they have received (RQ 2.a: reuse), understood (RQ 1.b: reflect), and applied (RQ 1.a: apply) the experience transported by the feedback. During the evaluation reported in [31], we used questionnaires to interview the students. They reported in majority that they have learnt from using HeRA (RQ 1.b: reflect).

*Step 4*. In an experiment we could show that it is possible to encode new experiences as heuristic critiques in less than 7 min and to change existing heuristic critiques in less than 2 min [32] (RQ 2.b: encode).

### *15.5.3 Discussion of Implications*

In the scope of this chapter, we do not give details about the evaluation. Instead, we show how to approach the research questions raised in this section. Thus, we complement the design principles by giving hints on how to operationalise those principles. In addition, we show that automatic requirements checkers can be even more valuable for an organisation, if they are seen as a way to capture and apply knowledge and experience. Our evaluation shows that it is possible to support the complete learning model: Experience-based requirements tools are improving the documentation of requirements (by lowering documentation cost or increasing quality) and support learning on the organisational and individual level.

## 15.6 Summary

In this chapter, we argued that experience-based tools can offer important support for requirements engineers. Such tools offer valuable experience during requirements engineering activities. Feedback based on these experiences helps analysts to cope with the information overload and complexity of modern systems.

Beyond recall and precision, we discussed additional properties of such experience-based tools. Especially the proactivity and the ability to learn new experience can be valuable to manage requirements knowledge.

We illustrated our concepts with an exemplary implementation of an experience-based requirements engineering tool. Based on this implementation, we sketched a strategy for evaluating such tools. It is important to evaluate the impact of such tools on the quality of the product (i.e. software requirements specification) and on the quality of the process (i.e. the documentation and quality assurance of requirements). Beyond that, the effect of such tools on organisational and individual learning should be assessed to fully cover the capabilities of such tools. The learning model we presented in this chapter is an important asset for this task. On the long run, improvements of these aspects of knowledge management can lead to considerable benefits.

## References

1. Schneider K (2009) Experience and knowledge management in software engineering. Springer, Berlin
2. Senge PM (1993) The fifth discipline: the art and practice of the learning organization. Century Business Random House, London
3. Kiyavitskaya N, Zeni N, Mich L, Berry DM (2008) Requirements for tools for ambiguity identification and measurement in natural language requirements specifications. Requir Eng 13:207–239
4. Schön DA (1983) The reflective practitioner: how professionals think in action. Basic Books, New York

5. Fischer G (1994) Domain-oriented design environments. Autom Sof Eng 1:177–203
6. Knauss E, Lübke D, Meyer S (2009) Feedback-driven requirements engineering: the heuristic requirements assistant. In: Proceedings of the IEEE 31st international conference on software engineering, IEEE, Vancouver, Canada, pp 587–590
7. Cockburn A (2001) Writing effective use cases. Addison-Wesley, Boston
8. Kusumoto S, Matukawa F, Inoue K, Hanabusa S, Maegawa Y (2004) Estimating effort by use case points: method, tool and case study. In: Proceedings of the 10th international symposium on software metrics, IEEE, Chicago, USA, pp 292–299
9. Maciaszek LA (2007) Requirements analysis and system design. Pearson Education Limited, Harlow
10. Berry DM, Kamsties E, Krieger MM (2003) From contract drafting to software specification: linguistic sources of ambiguity, Technical report, University of Waterloo
11. Luhn HP (1958) The automatic creation of literature abstracts. IBM J Res Develop 2:159–165
12. Willett P (2006) The Porter stemming algorithm: then and now. Program Electron Lib Inform Syst 40:219–223
13. Knauss E, Houmb S, Schneider K, Islam S, Jürjens J (2011) Supporting requirements engineers in recognising security issues. In: 17th international working conference on requirements engineering: foundation for software quality, Essen, Germany, pp 4–18
14. Berry DM, Kamsties E (2003) Ambiguity in requirements specification. In: do Leite Prado JCS, Doorn JH (eds) Perspectives of requirements engineering. Kluwer, Norwell, pp 7–44
15. Wilson WM, Rosenberg LH, Hyatt LE (1997) Automated analysis of requirement specifications. In: Proceedings of the 19th international conference on software engineering (ICSE'97. ACM, New York, pp 161–171
16. Kof L (2005) Text analysis for requirements engineering. Ph.D. thesis, Technische Universität München, Germany
17. Lee SW, Muthurajan D, Gandhi RA, Yavagal DS, Ahn G-J (2006) Building decision support problem domain ontology from natural language requirements for software assurance. Int J Softw Eng Knowl Eng 16:851–884
18. Chantree F, Nuseibeh B, de Roeck A, Willis A (2006) Identifying nocuous ambiguities in natural language requirements. In: Proceedings of the 14th IEEE international requirements engineering conference. IEEE Computer Society, Minneapolis, pp 56–65
19. Gleich B, Creighton O, Kof L (2010) Ambiguity detection: towards a tool explaining ambiguity sources. In: Wieringa R, Persson A (eds) Proceedings of requirements engineering: foundation for software quality (REFSQ). Springer, Essen, pp 218–232
20. Fabbrini F, Fusani M, Gnesi S, Lami G (2001) An automatic quality evaluation for natural language requirements. In: Proceedings of the seventh international workshop on RE: foundation for software quality (REFSQ 2001), Interlaken, pp 150–164
21. Fantechi A, Gnesi S, Lami G, Maccari A (2002) Application of linguistic techniques for use case analysis. In: Proceedings of IEEE joint international conference on requirements engineering, Essen, pp 157–164
22. Melchisedech R (2000) Verwaltung und Prüfung natürlichsprachlicher Spezifikationen. Ph.D. thesis, Fakultät Informatik, Universität and Stuttgart, Germany
23. Somé SS (2006) Supporting use case based requirements engineering. Inform Softw Technol 48:43–58
24. Jang H-C (1994) A knowledge-based analyzer for requirements specification analysis. In: Proceedings of the sixth international conference on tools with artificial intelligence, New Orleans, USA, pp 276–282
25. Hunter A, Nuseibeh B (1998) Managing inconsistent specifications: reasoning, analysis, and action. ACM Trans Softw Eng Methodol 7:335–367
26. Gervasi V, Zowghi D (2005) Reasoning about inconsistencies in natural language requirements. ACM Trans Softw Eng Methodol 14:277–330

27. Berenbach B, Borotto G (2006) Metrics for model driven requirements development. In: ICSE'06: Proceedings of the 28th international conference on software engineering. ACM, Shanghai, pp 445–451
28. de Souza CRB, Oliveira HLR, da Rocha CRP, Gonçalves KM, Redmiles DF (2003) Using critiquing systems for inconsistency detection in software engineering models. SEKE, San Francisco, USA, pp 196–203
29. Gervasi V, Zowghi D (2010) On the role of ambiguity in RE. In: Wieringa R, Persson A (eds) Proceedings of requirements engineering: foundation for software quality (REFSQ). Springer, Essen, pp 248–254
30. Adam S, Doerr J, Eisenbarth M, Gross A (2009) Using task-oriented requirements engineering in different domains – experience of application in research and industry. In: Proceedings of the 17th IEEE international requirements engineering conference (RE'09), Atlanta, pp 267–272
31. Knauss E, Flohr T (2007) Managing requirement engineering processes by adapted quality gateways and critique-based RE-tools. In: Proceedings of workshop on measuring requirements for project and product success, Palma de Mallorca, Spain
32. Knauss E, Schneider K (2012) Supporting learning organisations in writing better requirements documents based on heuristic critiques. In: Regnell B, Damian D (eds) Proceedings of requirements engineering: foundation for software quality (REFSQ'12). Springer, Heidelberg/Essen, pp 165–171
33. Knauss E, Schneider K, Stapel K (2009) Learning to write better requirements through heuristic critiques, IEEE, Atlanta, USA

# Chapter 16
# The Eclipse Requirements Modeling Framework

**M. Jastram**

**Abstract** This chapter is concerned with the Requirements Modeling Framework (RMF) (http://eclipse.org/rmf), an Eclipse-based open-source platform for requirements engineering. The core of RMF is based on the emerging Requirements Interchange Format (ReqIF), which is an OMG standard [1]. The project uses ReqIF as the central data model. At the time of this writing, RMF was the only open-source implementation of the ReqIF data model.

By being based on an open standard that is currently gaining industry support, RMF can act as an interface to existing requirements management tools. Further, by based on the Eclipse platform, integration with existing Eclipse-based offerings is possible.

In this chapter, we will describe the architecture of the RMF project, as well as the underlying ReqIF standard. Further, we give an overview of the GUI, which is called ProR. A key strength of RMF and ProR is the extensibility, and we present the integration ProR with Rodin, which allows traceability between natural language requirements and Event-B formal models.

## 16.1 Introduction

RMF may be of relevance to the reader for a number of reasons. First and foremost, RMF extends the Eclipse ecosystem with a meta-model for modeling requirements. We hope that ReqIF will have a similar effect on requirements engineering to what UML did for modeling: providing a unified data model that tools could converge on. By being open source, RMF contributes to spreading the use of ReqIF.

Second, RMF contains the stand-alone ProR application, a platform for requirements engineering. This tool uses ReqIF as the underlying data model and therefore offers sophisticated, standardized data structures for organizing

M. Jastram (✉)
Formal Mind GmbH, Düsseldorf, Germany
e-mail: michael.jastram@formalmind.com

requirements and provides interoperability with industry tools. Especially in small companies or academic projects, users until now faced the dilemma: Tools like Word and Excel have wide acceptance but limited features for requirements engineering. Professional tools like Rational DOORS[1] or IRQA[2] are not affordable. There are some free tools, like Trend/Analyst,[3] Topcased [2], or Wikis.[4] But these either use their own data structures, with their own limitations, or follow a standard with few features in respect to requirements, like SysML. ProR provides a lot of functionality out of the box and offers interoperability according to an international standard. The interest that ProR created both in academia and industry confirms this.

Third, ProR can be easily extended to provide additional functionality or for integration with other tools. ProR is implemented as Eclipse plug-ins. While it can run stand-alone, it can be installed in any existing Eclipse system. No special project type is required for ProR; therefore, any Eclipse project can contain ProR requirements files. ProR provides an extension point, which allows developers to build tool integrations. For instance, we created an integration plug-in for Rodin, a tool for formal modeling. After installing ProR into Rodin, it was possible to use drag and drop to integrate model elements into the requirements specification.

Fourth, RMF can be used as a generic platform for working with ReqIF-based requirements, independently of an Eclipse-based GUI. This can be useful for a wide range of activities from analysis, report generation, generation of requirements artifacts, product line management, etc. RMF is built on the Eclipse Modeling Framework (EMF), which supports the integration with other EMF-based offerings.

Before discussing the technical details, we will provide an overview of the current state of requirements modeling, both in industry and academia.

This chapter is structured as follows: Sect. 16.2 provides an overview of the current state of requirements modeling. Section 16.3 describes the data model of the ReqIF format. Section 16.4 introduces the architecture of the Requirements Modeling Framework (RMF), followed in Sect. 16.5 by a description and tutorial of ProR, the user interface. Section 16.6 demonstrates how the platform can be extended and integrated with other Eclipse-based software. This chapter concludes with Sect. 16.7.

## 16.2 Requirements Modeling

While this chapter is concerned with a software platform, it is useful to put it into the bigger context of requirements management and engineering, which we will do in this section. Ultimately, a tool is only useful if used properly and with a clear goal.

[1] http://www.ibm.com/software/awdtools/doors/.

[2] http://www.visuresolutions.com/irqa-requirements-tool.

[3] http://www.gebit.de/loesungen/technische-loesungen/trend-analyst-requirements.html.

[4] http://www.mediawiki.org/.

A tool must support the activities found in requirements engineering and requirements management. These include the structuring of requirements, establishing traceability, handling versions, and integrating with other processes (e.g., testing and project management), to name just a few. The specific activities depend to a degree on the process that the tool has to support.

After discussing requirements and specifications in general below, we will revisit the topic of tool support in Sect. 16.2.5.

### *16.2.1 Specifying Systems*

Everything is build twice: First, an idea forms in the mind; then the idea is realized. This is true from the smallest to the biggest projects, from hanging up a picture to building a space rocket. In the case of the space rocket, there would be a number of intermediate steps to account for the complexity and scale of the task at hand. The number of intermediate steps and types of documentation depends on the size of the project, how critical it is, how many people are involved, and many other factors. Nevertheless, requirements and specification are artifacts that are so important that they play part in all but the smallest projects.

Every project should have a *goal*. A goal typically says nothing about the "how" ("How do I achieve this?") but the "what" ("What is it that I want to achieve?"). A goal is typically very simple and high level. This does not necessarily mean that it is not precise or quantifiable.

A *requirement* puts the goal into the context of the world. A requirement for hanging a picture on a wall is that it stays there, which in turn has to take the picture's properties into account. This does not mean that it should indicate *how* the picture is mounted – a nail or two screws – because a good requirement does not provide a solution but precisely describes the problem.

For big projects, it is not practical to go directly from goal to requirements. The goal is typically broken down into subgoals; an overall architecture is established that allows partitioning of the tasks at hand. In addition, there is a lot of overhead that does not directly contribute to the development but that is crucial nevertheless. This includes artifacts for subdisciplines like project management, testing, and many other areas of interest.

It is the *specification's* job to provide a solution to the problem. This is the place that describes that a nail shall be used to put up the photo and where to put it. It is dangerous to look for solutions sooner than at this point because it is easy to miss important requirements or something crucial regarding the context.

### *16.2.2 Structuring Requirements*

A good structure of requirements can make a huge difference in their management and traceability, and quite a bit of research went into understanding this relationship better. In industrial environments, this manifests itself in standards like IEEE 830–1997 [3] or the relevant aspects of process frameworks like RUP [4].

In academia, Gunter, Jackson, and Zave [5] developed WRSPM as a *reference model* for requirements and specifications. A reference model is attractive for discussion, as it draws on what is already understood about requirements and specifications, while being general enough to be flexible. There are a number of concrete approaches that fit nicely into the WRSPM reference model, including Problem Frames [6], KAOS [7] or the functional-documentation model [8].

WRSPM distinguishes five artifacts:

*Domain knowledge* ($W$) describes how the world is expected to behave.
*Requirements* ($R$) describe how the world should behave.
*Specifications* ($S$) bridge the world and the system.
*Program* ($P$) provides an implementation of $S$.
*Programming platform* ($M$) provides an execution environment for $P$.

Inexperienced users sometimes confuse requirements and domain knowledge, but the distinction is quite important:

- Requirements describe how the world *should behave*, and the system is responsible for this.
- Domain knowledge describes how the world is *expected to behave*, and the functioning of the system depends on the domain knowledge holding.

The relationship between requirements, domain knowledge, and the specification can be expressed formally:

$$S \wedge W \Rightarrow R$$

or in words: Assuming a system that conforms to the specification $S$ and assuming that the domain properties $W$ hold, the requirements $R$ are realized. There are some subtleties (e.g., we are probably not interested in the trivial solution), but this is a central idea of WRSPM.

Note that WRSPM does not know the concept of a goal. But according to WRSPM, a goal is merely a high-level requirement. Also note that there is a whole category of approaches called goal-oriented requirements engineering (GORE) [9].

The reference model defines *phenomena*, which act as the vocabulary to formulate the artifacts. There are different types of phenomena based on their visibility. For instance, there may be phenomena that the machine is not aware of. Consider a thermostat: The controller is not aware of the temperature[5] but only of the voltage at one of its inputs.

[5] To be precise, whether the controller is aware of the temperature or not depends on where the line is drawn between system and environment. In this simple example, the sensor is not part of the system (the controller).

The reference model can be applied to any requirements or specifications, no matter whether they use natural language or a formalism. Once applied, more formal reasoning about the specification is possible.

Informal requirements rarely explicitly distinguish between requirements, domain knowledge, and even specification elements and implementation details. In the following, *artifacts* will refer to all of them.

### *16.2.3 Informal and Formal Specifications*

Artifacts can be formalized by modeling them using a formalism. Many formalisms exist, all with their respective advantages and disadvantages. Modeling can also be applied on various levels of the development process – for goals, requirements, the specification, and even for the implementation.

Some formalisms are more, others less "formal." Often, a formalism only models a certain aspect of the specification and has to be complemented with additional information. Here are a few examples:

*Context diagrams* only formalize a small aspect of a system, its boundary to the world. They help in the requirements elicitation process by forcing us to define the boundary of the system and to identify the actors that can interact with it. Using a context diagram in the elicitation process will leave its traces in the structure of the requirements (i.e., by systematically enumerating all actors and how they interact with the system). They are formal only in the sense that they allow reasoning about a tiny aspect of the system and need to be complemented with much more information.

*UML and SysML diagrams* provide modeling elements for many elements of the system and their relationship, ranging from class diagrams for object relationships to state diagrams for transitions. While they are useful, they are not formal enough to express complex functionalities and must be complemented somehow, for example, by use cases.

*Problem frames* [6] introduce *problem diagrams*, which extend the notation of context diagrams and make the problem explicit by showing the requirements in the diagram. The notation of context diagrams is also formalized by distinguishing between machine domain, designed domains, and given domains. The notation further introduces *problem frame diagrams* for concisely recording problem frames.

*Z, VDM, B*, and many others are a particular kind of mathematically based techniques for the specification of *sequential* behavior. These and similar notations are used to specify and verify systems. While formal methods do not guarantee correctness, they can greatly increase the understanding of a system and help revealing inconsistencies, ambiguities, and incompleteness that might otherwise go undetected [10].

*CSP, CSS*, and others are formal methods that are used for specifying *concurrent* behavior.

### 16.2.4 Traceability

Traceability refers to the relationships between and within the artifacts and other elements [11, 12]. These are plentiful and exist implicitly. But the implicit traceability can be made explicit. By doing so, those traces become themselves artifacts that must be maintained. Therefore, the benefits and costs of making traces explicit must be weighted carefully – as with some artifacts, the cost of stale traces may be higher than the cost of no explicit traces.

Making traces explicit can in itself provide useful information. Consider the "is realized by" relationship between requirements and specification. Such a relationship would immediately identify those requirements that are not specified yet, namely, those requirements that have no outgoing traces. Such a requirement can then be inspected and the specification extended to realize it. After the specification has been extended, a new trace is created, marking the requirement as realized.

While this approach works in principle, there are at least two problems with it. First, which elements will be traced? It would be nice if there was a one-to-one relationship between requirements and specification elements, but this is true only for the simplest toy examples. In practice, this is an n-to-m relationship, and sometimes, one end of the trace can be elusive. Just consider quality requirements that apply to the system as a whole.

Maintenance is the second issue. Creating a trace correctly is one thing, but keeping it updated is quite another. Consider again the "is realized by" relationship. All incoming traces would have to be verified to make sure that the specification element still, in fact, realizes all requirements that it traces. But this works only if all traces have been created in the first place. And when more corrections have to be done during this verification (both on requirements and specification), it may trigger another wave of verifications. Tool support can help to mark traces for verification – but how much this helps depends on the completeness and correctness of the traces.

The ease of traceability depends, amongst other things, on the structure and quality of the artifacts. For instance, one quality criterion for good requirements is the lack of redundancy. Not having redundancy also eases traceability. Further, there are many ways to structure the artifacts. A good structure can make traceability significantly easier. The structure depends on notation and approach. The approach guides the artifacts towards a certain structure, while the notation constrains how easy or difficult it is to express something. Some notations require a certain approach and may also push the artifacts in a certain structure. This is good if the notation is well suited for the problem at hand, but it can be counterproductive if this is not the case. Just imagine drawing the blueprint of a house with UML or to document an enterprise system with a mechanical drawing. Other notations are open to everything, like natural language. But the downside in this case is that the notation provides no guidance and can be ambiguous or contradicting.

### *16.2.5 The Importance of Tool Support*

The previous sections described the concerns regarding the working with requirements and specifications. We created RMF specifically to provide a platform that could be used in academia and industry to realize their ideas. We believe that there is a real need for this: Research projects often build their own tools in isolation with proprietary data structures, which vastly decreases their survival chances. In industry, we see a lot of customization of proprietary tools (for instance, there is a whole industry creating scripts for IBM Rational DOORS[6]). RMF in turn builds on the open ReqIF standard that is currently being adapted by commercial tools, and it is built on top of Eclipse EMF [13]. Specifically, here are the areas where RMF could be put to use:

*Structuring Requirements* RMF provides all the features necessary for structuring artifacts, both according to WRSPM and other approaches. In itself, the ProR tool does not put any constraints on the structure, but this can be achieved via specific plug-ins which could also provide guidance to the use, for instance, by providing wizards.

*Model Integration* As we will see in Sect. 16.6, an integration with models can be achieved via plug-ins as well, especially if the formal modeling tools are built using Eclipse EMF. In that case, referenced model elements can be seamlessly incorporated into ProR specifications.

*Traceability* ReqIF includes data structures for typed traces, and RMF can be extended for intelligent handling of the traceability. For instance, traces could be marked as suspect, as soon as the source or target element of the trace changes.

## 16.3 Requirements Interchange Format

We will provide a brief overview on the Requirements Interchange Format (ReqIF) [1] file format and data model. We are mainly concerned with the capabilities and limitations of the data model. The tool that we describe in Sect. 16.5 uses ReqIF as the underlying data model. Doing so provides interoperability with industry-strength tools and builds on top of a public standard.

ReqIF allows the structuring of natural language artifacts and supports an arbitrary number of attributes and the creation of attributed links between artifacts. It therefore provides the foundation of collecting and organizing artifacts in a way that users are comfortable with but provides additional structure for supporting a solid traceability.

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

ReqIF was created in 2004[7] by the “Hersteller Initiative Software” (HIS[8]), a body of the German automotive industry that oversees vendor-independent collaboration. At the time, the car manufacturers were concerned about the efficient exchange of requirements with their suppliers. Back then, exchange took place either with low-tech tools (Word, Excel, PDF) or with proprietary tools and their proprietary exchange mechanisms. ReqIF was meant to be an exchange format that would allow the exchange to follow an open standard, even if the tools themselves are proprietary.

The basic use case for ReqIF consists of the following steps (described in detail in the ReqIF specification [1]):

1. The manufacturer exports the subset of requirements that are relevant to the supplier, with the subset of attributes that are relevant.
2. Those attributes that the supplier is expected to modify are writable; other content is marked as readable only.
3. The supplier imports the data from the manufacturer into their system. If this not the first import, then the data may be merged into an existing requirements database.
4. The supplier can then edit the writable attributes or even create a traceability to other elements in their database (e.g., a systems specification).
5. The supplier performs an export with the data relevant to the manufacturer.
6. The manufacturer merges the data back into their requirements database.

## *16.3.1 The ReqIF Data Model*

In general terms, a ReqIF model contains attributed requirements that are connected with attributed links. The requirements can be arbitrarily grouped into document-like constructs. We will first point out a few key model features and then provide more specifics from the ReqIF specification [1].

A *SpecObject* represents a requirement. A SpecObject has a number of *AttributeValues*, which hold the actual content of the SpecObject. SpecObjects are organized in *Specifications*, which are hierarchical structures holding *SpecHierarchy* elements. Each SpecHierarchy refers to exactly one SpecObject. This way, the same SpecObject can be referenced from various SpecHierarchies.

ReqIF contains a sophisticated data model for *Datatypes*, support for permission management, facilities for grouping data, and hooks for tool extensions.

ReqIF is persisted as XML and therefore represents a tree structure. The top level element is called ReqIF. It is little more than a container for the *ReqIFHeader*, a placeholder for tool-specific data (*ReqIFToolExtension*) and the actual content

[7] At the time of its creation, the format was called RIF and only later on, renamed into ReqIF.

[8] http://www.automotive-his.de/.

(*ReqIFContent*). The ReqIFContent has no attributes but is simply a container for six elements. These are as follows:

*SpecObject* A SpecObject represents an actual requirement. The values (AttributeValue) of the SpecObject depend on its SpecType.

*SpecType* A SpecType is a data structure that serves as the template for anything that has Attributes (e.g., a SpecObject). It contains a list of Attributes, which are named entities of a certain data type and an optional default value. For example, a SpecObject of a certain type has a value for each of the SpecType's attributes.

*DatatypeDefinition* A DatatypeDefinition is an instance of one of the atomic data types that is configured to use. For instance, String is an atomic data type. A DatatypeDefinition for a String would have a name and the maximum length of the string. Each attribute of a SpecType is associated with a DatatypeDefinition.

*Specification* SpecObjects can be grouped together in a tree structure called Specification. A Specification references SpecObjects. Therefore, it is possible for the same SpecObject to appear in multiple Specifications or multiple times in the same Specification.

In addition, a Specification itself may have a SpecType and therefore AttributeValues.

*SpecRelation* A SpecRelation is a link between SpecObjects; it contains a source and a target. In addition, a SpecRelation can have a SpecType and therefore AttributeValues.

*RelationGroup* SpecRelations can be grouped together in a RelationGroup but only if the SpecRelations have the same source and target Specifications. This construct got added to accommodate certain data structures of existing, proprietary requirements tools.

We just learned that there are four element types that can have attributes: SpecObjects, Specifications, SpecRelations, and RelationGroups. These four are all *SpecElementsWithAttributes* or *SpecElements* for short. Each SpecElement has its own subclass of SpecType (SpecObjectType, SpecificationType, SpecRelationType, and RelationGroupType). A SpecType has any number of AttributeDefinitions, which ultimately determines the values of a SpecElement. Correspondingly, a SpecElement can have any number of AttributeValues. The AttributeValues of a SpecElement depend on the AttributeDefinitions of the SpeElement's SpecType. This fact cannot be deducted from the model.

The AttributeDefinition references a DatatypeDefinition that ultimately determines the value of the Attribute Value of the corresponding SpecElement. For each atomic data type of ReqIF, there is a corresponding DatatypeDefinition, AttributeDefinition, and AttributeValue each.

The ReqIF specification [1] contains a number of class diagrams that nicely visualize these relationships.

ReqIF supports the following atomic data types:

*String* A unicode text string. The maximum length can be set on the Datatype.
*Boolean* A Boolean value. No customization is possible.
*Integer* An integer value. The maximum and minimum can be set on the Datatype.
*Real* A real value. The maximum and minimum can be set on the Datatype, as well as the accuracy.
*Date* A date- and timestamp value. No customization is possible.
*Enumeration* An enumeration Datatype consists of a number of enumeration values. The AttributeDefinition determines whether the values are single value or multiple values.
*XHTML* XHTML is used as a container for a number of more specific content types. The AttributeValue has a flag to indicate whether the value is simplified, which can be used if the tool used to edit only supports a simplified version of the content. For instance, a tool that does not support rich text editing could set the flag and replace the content with plain text.

ReqIF consists of 44 element types in total. The ones we just described are important for understanding ReqIF in general and this chapter in particular. Elements we omitted concern aspects like access control and identifier management.

### *16.3.2 The Impact of ReqIF*

Even though ReqIF was initially created as a file-based exchange format, we believe that it can be much more than that. By employing ReqIF directly as the underlying data model for an application, we can take full advantage of the model's versatility. Conveniently, the OMG made the data model available in the CMOF format, thereby facilitating the process of instantiating the data model in a concrete development environment. As we will see in the next section, RMF is based on EMF [13], which can use CMOF as an input.

On the significance on ReqIF and our first clean-room implementation of the standard, we draw comparisons to model-driven software development: After the specification of UML, a lot of publications and work concentrated on this standard, paving the way for low-cost and open-source tools. We hope that our open-source clean-room implementation of the standard based on Eclipse can serve as the basis for both innovative conceptual work and new tools.

This is by no means guaranteed, and there are examples where this approach did not work. For instance, the XMI format (in model-based community) was not that successful, and XMI was also promoted by OMG.

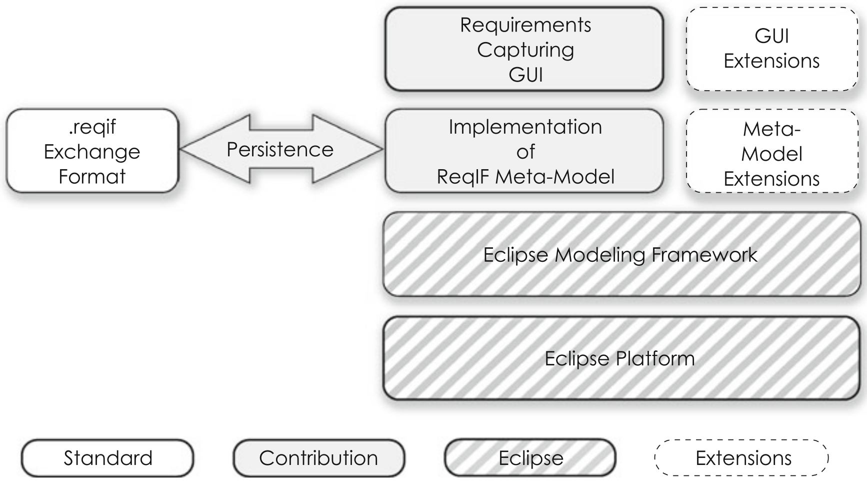


**Fig. 16.1** High-level architecture of RMF

## 16.4 The Requirements Modeling Framework (RMF)

RMF grew out of the Deploy[9] research project [14] and the VERDE[10] research project. It is an Eclipse Foundation project that unifies a generic core engine to work with RIF/ReqIF content and a GUI called ProR.

The vision of RMF is to have at least one clean-room implementation of the OMG ReqIF standard in the form of an EMF model and some rudimentary tooling to edit these models. The idea is to implement the standard so that it is compatible with Eclipse technologies like GMF, Xpand, Acceleo, Sphinx, etc. and other key technologies like CDO.

### *16.4.1 High-Level Structure*

Figure 16.1 depicts the high-level architecture of RMF. It consists of an EMF-based implementation of the ReqIF core that supports persistence using the ReqIF XML schema. The core also supports the older versions RIF 1.1a and RIF 1.2.

The GUI for capturing requirements is called ProR (see also Sect. 16.5). It operates directly on the ReqIF data model. This is an advantage compared to

[9] http://www.deploy-project.eu/.
[10] http://www.itea-verde.org/.

existing requirements tools, where a transformation between ReqIF and the tool's data model is necessary. Not all tools support all ReqIF features; therefore, information may be lost in the process.

ProR at this time only supports the current version of ReqIF 1.0.1, not the older versions.

These contributions have their origins in research projects, where they are actively used. In particular, these research projects already produced extensions, demonstrating the value of the platform. These are depicted in Fig. 16.1 as well and described in Sect. 16.6.

### *16.4.2 Extending RMF*

RMF is designed as a generic framework for requirements modeling, and the ProR GUI is designed as an extensible application. It has been used and extended in various projects, as we will describe in Sect. 16.6. It provides an extension point that allows the tailoring with plug-ins.

This is an important aspect of the project. As we have seen in industry, heavy tailoring to the processes used and integration with other tools is what makes requirements tools successful. By using Eclipse as the platform for this tool, we can provide integration with modeling tools like Rodin [15] or Topcased [2]. By providing a versatile extension point, the behavior of the application can be adapted to the process employed.

## 16.5 ProR

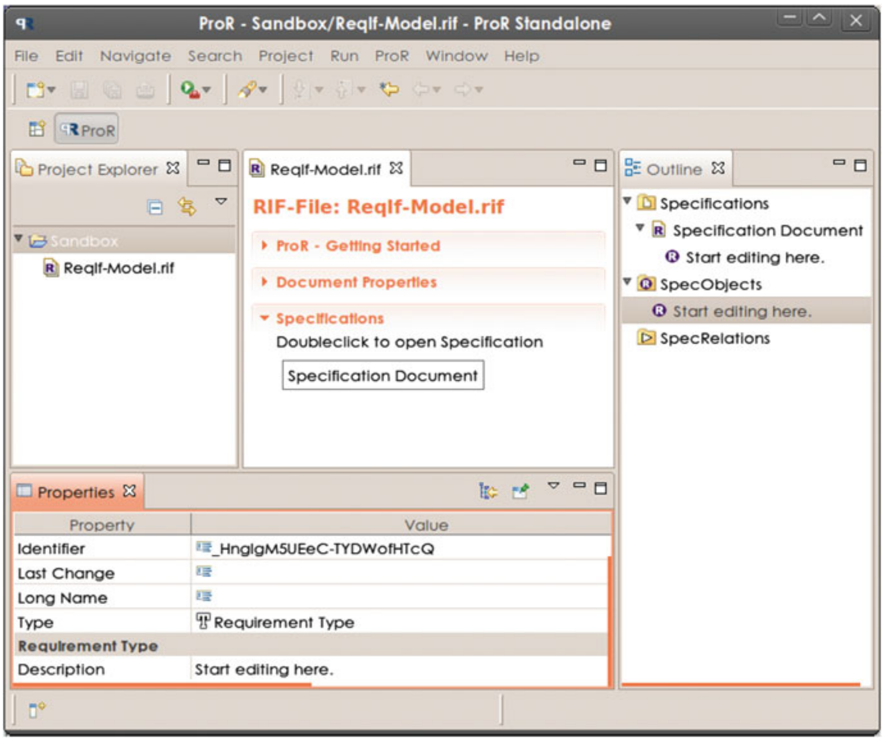
ProR is the Graphical User Interface (GUI) or RMF. ProR is available as a standalone application, and it can be integrated into existing Eclipse installations.

This section will go through the more important features of ProR to provide an impression of the tool in action. We provided a more extensive introduction to the tool in [16]. We also created a screencast[11] that demonstrates the basic features of ProR.

### *16.5.1 Installing ProR*

ProR can be downloaded stand-alone or installed into an existing application via its update site. The download is a convenient option for nontechnical people who just

[11] http://www.youtube.com/watch?v=sdfTNZduvZ4.



| Property | Value |
|---|---|
| Identifier | _HnglgM5UEeC-TYDWofHTcQ |
| Last Change | |
| Long Name | |
| Type | Requirement Type |
| **Requirement Type** | |
| Description | Start editing here. |

**Fig. 16.2** ProR with a newly created ReqIF model, as produced by the wizard

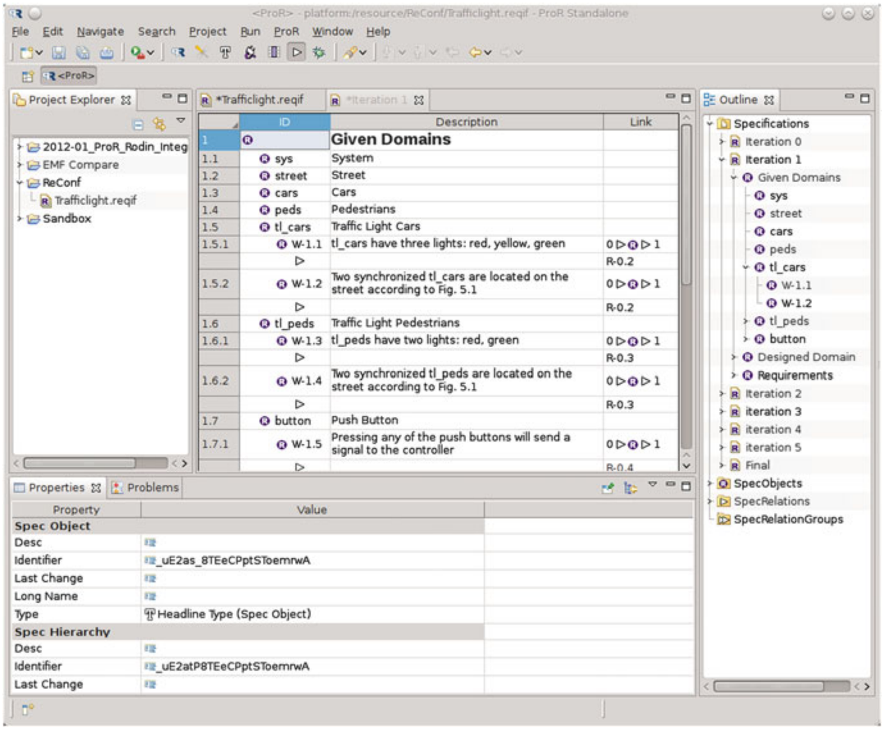
want to get started with ProR. There is no special restriction for the update site version: ProR can be installed into any reasonably new Eclipse installation.

### *16.5.2 Creating a ReqIF Model*

ReqIF models can be created in any Eclipse project and manifest themselves as a . reqif file. A user creates a new ReqIF model via the FILE | New. . . menu, where there is a wizard for a new “Reqif10 Model.” The wizard will create a new ReqIF model with a very rudimentary structure, with one Datatype, one SpecType with one Attribute, using the Datatype, and one Specification with one SpecObject that uses the SpecType.

The user can then inspect the model structure in the outline and the properties views. ProR provides its own *Perspective*, which ensures that all relevant views are shown.

The editor in Fig. 16.2 (the window in the middle) provides an overview of the model. The most important section is the one labeled “Specifications.” Users can double click on specifications to open them in their own editor, as shown in Fig. 16.3.



| | ID | Description | Link |
|---|---|---|---|
| 1 | | **Given Domains** | |
| 1.1 | sys | System | |
| 1.2 | street | Street | |
| 1.3 | cars | Cars | |
| 1.4 | peds | Pedestrians | |
| 1.5 | tl_cars | Traffic Light Cars | |
| 1.5.1 | W-1.1 | tl_cars have three lights: red, yellow, green | 0 ▷ ▷ 1 |
| | ▷ | | R-0.2 |
| 1.5.2 | W-1.2 | Two synchronized tl_cars are located on the street according to Fig. 5.1 | 0 ▷ ▷ 1 |
| | ▷ | | R-0.2 |
| 1.6 | tl_peds | Traffic Light Pedestrians | |
| 1.6.1 | W-1.3 | tl_peds have two lights: red, green | 0 ▷ ▷ 1 |
| | ▷ | | R-0.3 |
| 1.6.2 | W-1.4 | Two synchronized tl_peds are located on the street according to Fig. 5.1 | 0 ▷ ▷ 1 |
| | ▷ | | R-0.3 |
| 1.7 | button | Push Button | |
| 1.7.1 | W-1.5 | Pressing any of the push buttons will send a signal to the controller | 0 ▷ ▷ 1 |
| | ▷ | | R-0.4 |

| Property | Value |
|---|---|
| **Spec Object** | |
| Desc | |
| Identifier | _uE2as_8TEeCPptSToemrwA |
| Last Change | |
| Long Name | |
| Type | Headline Type (Spec Object) |
| **Spec Hierarchy** | |
| Desc | |
| Identifier | _uE2atP8TEeCPptSToemrwA |
| Last Change | |

**Fig. 16.3** ProR with the specification editor open. The screenshot shows some sample

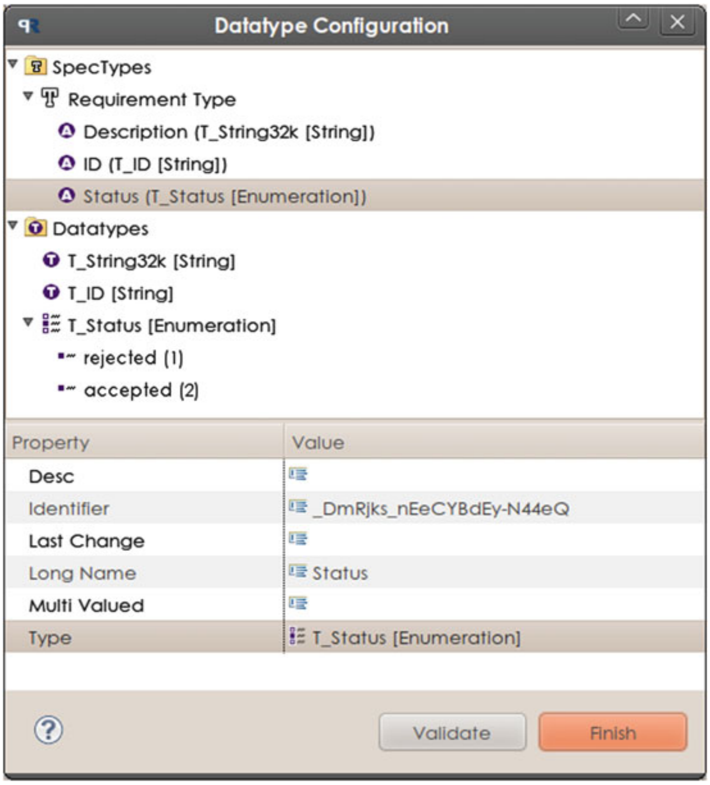
Each row represents a requirement (SpecObject), and each requirement can have an arbitrary number of attributes. Which specific attribute a requirement has depends on its type.

Users can configure the editor to show an arbitrary number of columns. Each column has a name. If an element has an attribute of that name, then the value of that attribute is shown in the corresponding column.

## 16.5.3 *New Attributes*

The actual information of requirements (SpecObjects) is stored in its attributes. Which attributes a SpecObject has depends on its type. Users can add more attributes to existing requirement types.

To do this, the user opens the dialog for the data types via ProR|Datatype Configuration... or the corresponding icon in the toolbar. The upper part of the dialog shows the data structures, while the lower part contains a property view that allows editing the properties of the element that is selected in the upper part. New child or sibling elements can be added via context menus.



SpecTypes
Requirement Type
Description (T_String32k [String])
ID (T_ID [String])
Status (T_Status [Enumeration])
Datatypes
T_String32k [String]
T_ID [String]
T_Status [Enumeration]
rejected (1)
accepted (2)

| Property | Value |
|---|---|
| Desc | |
| Identifier | _DmRjks_nEeCYBdEy-N44eQ |
| Last Change | |
| Long Name | Status |
| Multi Valued | |
| Type | T_Status [Enumeration] |

**Fig. 16.4** The data type dialog after adding some data

In this example, the user adds two attributes to the type "Requirements Type": an ID for a human readable identifier and a status field, which is an enumeration. The result is shown in Fig. 16.4.

The user just created a new data type for the ID called "T_ID." For the status field, the user created a new enumeration of type "T_Status." In the figure, we can see the properties of the selected element in the lower pane, where they can be edited.

### 16.5.4 *Configuration of the Editor*

When closing the dialog and selecting a requirement, the three properties are visible in the properties view, where the user can edit them. But the main pane of the editor still only shows one column. The user can add new columns via PROR | COLUMN CONFIGURATION. . . (or the corresponding tool bar icon), which opens a dialog for this purpose. The dialog looks and works similar to the one for the data types. In this example, the user adds one more column called "ID." The dialog also allows the reordering of columns via drag and drop, and the user uses this mechanism to make the ID column the first one.

With this setup, the user can set the status of individual requirements by selecting them and updating the status field in the properties view. Upon clicking on the field, a drop-down allows the selection of the new status value. Had the user added the status field to the editor (as just described), they could adjust the values directly in the editor as well.

Using this approach, users can add an arbitrary number of attributes to a requirement, which the user can all see and edit in the properties view. A selected number can additionally be shown in the editor. For example, the user may decide that a requirement should have a comment field to record additional information.

### *16.5.5 Generating IDs*

The ID column is now visible in the specification editor, but it is empty. While the user could add IDs simply by hand, this is error prone, and one would expect the tool to be able to handle this. ProR does not have the ability to generate IDs, but a "Presentation" can. Presentations are ProR-specific plug-ins that modify the presentation of data and inspect and modify the data. Presentations are described from a technical point of view in Sect. 16.6.

To add a presentation, the user opens the presentation dialog via ProR | Presentation Configuration... (or the tool bar). The Select Action... drop-down lists all installed presentations, and by selecting "ID Presentation," the user creates a new configuration element. In the properties, the user adjusts the prefix and counter of the ID presentation. But more important is the data type that is associated with the presentation. In this example, the user selects "T_ID" – and this is the reason why the user created a new data type for the IDs earlier.

After closing the dialog, all requirements that did not have an ID yet will have received one by the presentation.

### *16.5.6 Adding Requirements*

Finally, everything is ready for adding some data. The user does this via the context menus, but in several places, keyboard shortcuts are available as well. Upon opening the context menu for a requirement, the user adds new elements via the New Sibling and New Child submenus. A specification is a tree structure of arbitrary depth, and the left margin indicates via a corresponding numbering scheme the position in the hierarchy. In addition, the left margin of the first column is indented.

The context menu allows the creation of typed requirements – there is one entry for each user-defined type – which can save a lot of clicking. But it is also possible to add untyped requirements or even empty placeholders (*SpecHierarchies*). Adding a placeholder can be useful for referencing an existing requirement. Requirements may appear multiple times, both in the same specification and in other specifications of the same ReqIF model.

To allow the rapid addition of requirements, ProR provides the CTRL-ENTER keyboard shortcut. Upon activating the shortcut, the new requirement is inserted below the one that is currently selected and has the same type.

Last, a user can rearrange requirements via drag and drop or copy and paste.

### *16.5.7 Linking Requirements*

The user can link requirements via drag and drop. As drag and drop is also used for rearranging requirements, it has to be combined with a keyboard modifier. The key that needs to be pressed is dependent of the operating system and is the same that is used for creating file links and the like.

Once the user creates a link, the last column of the specification editor shows the number of incoming and outgoing links. The user can toggle the showing of the actual link objects (*SpecRelations*) via PROR | SPECRELATIONS. . ., which are then shown below the originating requirement (depicted in Fig. 16.3). The last column of link objects shows the destination object (selecting that column will show the target requirement's properties in the property view).

The user can assign types to link objects, resulting in them having attribute values. The values will be shown in the specification editor, if the columns are configured correspondingly.

This concludes the brief overview of the usage of ProR.

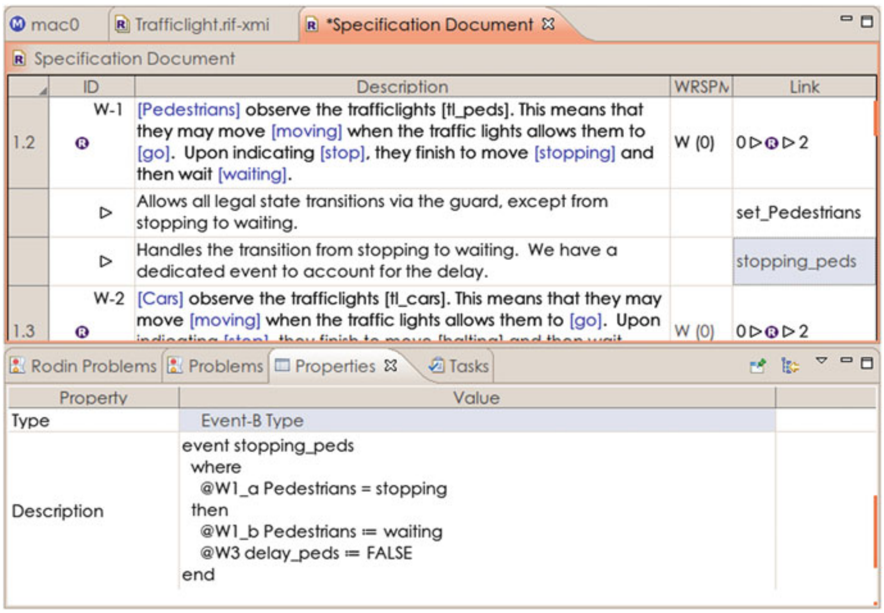
## 16.6 Extending ProR

The functionality of ProR is quite limited, but this is by design. ProR can be extended using the Eclipse plug-in mechanism. Many features that should be standard in a requirements engineering tool will not be implemented into the ProR core but could be made available via plug-ins. An example of this has been presented in Sect. 16.5.5, where a plug-in was responsible for generating user-readable IDs.

Likewise, functionality like versioning or baselining will not be integrated into the core. Versioning is already supported (albeit in a crude manner) by installing a repository plug-in like Subclipse[12] or Subversive[13] (Subversion support)

[12] http://subclipse.tigris.org/.

[13] http://www.eclipse.org/subversive/.



|  | ID | Description | WRSPM | Link |
|---|---|---|---|---|
| 1.2 | W-1 Ⓡ | [Pedestrians] observe the trafficlights [tl_peds]. This means that they may move [moving] when the traffic lights allows them to [go]. Upon indicating [stop], they finish to move [stopping] and then wait [waiting]. | W (0) | 0▷Ⓡ▷2 |
|  | ▷ | Allows all legal state transitions via the guard, except from stopping to waiting. |  | set_Pedestrians |
|  | ▷ | Handles the transition from stopping to waiting. We have a dedicated event to account for the delay. |  | stopping_peds |
| 1.3 | W-2 Ⓡ | [Cars] observe the trafficlights [tl_cars]. This means that they may move [moving] when the traffic lights allows them to [go]. Upon | W (0) | 0▷Ⓡ▷2 |

| Property | Value |
|---|---|
| Type | Event-B Type |
| Description | event stopping_peds<br>where<br>@W1_a Pedestrians = stopping<br>then<br>@W1_b Pedestrians ≔ waiting<br>@W3 delay_peds ≔ FALSE<br>end |

**Fig. 16.5** Integration of ProR with an Event-B formal model. Model elements are highlighted in the requirement text, and annotated traces to the model show the model element in the property view

or eGit[14] (git support). However, these plug-ins perform versioning on the file level. In practice, versioning on the requirement lesvel would be more desirable, and from a technical point of view, it is straightforward to realize this in the form of a plug-in.

Extensions for ProR exist – those have been driven mainly by academic needs so far. In this section, we demonstrate how a developer can integrate RMF with other Eclipse-based tools.

### *16.6.1 Traceability Between Requirements and Event-B Models*

The research project Deploy [14] is concerned with the deployment of formal methods in industry. Traceability between natural language requirements and formal models was one issue that the deployment partners were struggling with. Deploy continues to develop the Eclipse-based Rodin tool [15], which was the main deciding factor for using Eclipse for RMF. ReqIF was an attractive choice for providing interoperability with industry tools. By using EMF, we could build directly on the ReqIF data model, which allowed us to get a solid integration quickly. Figure 16.5 shows how we establish traceability between formal models

[14] http://www.eclipse.org/egit/.

and natural language requirements [17]. The formal modeling is done in Event-B (using Rodin). Integration is seamless via drag and drop, and a custom renderer supports color highlighting of model elements.

### 16.6.2 Tracepoint Approach in ProR

The general concept of traceability in VERDE led to the decision to implement a traceability that is independent of the types of artifacts that are involved. Since Eclipse-based models are usually built on EMF, VERDE implements a generic solution for the traceability of EMF-based elements called tracepoints [18]. The core data structure is a mapping table with three elements: source element, target element, and arbitrary additional information. The elements are identified by a data structure, the so-called *tracepoint*. The inner structure of a tracepoint depends on the structure of the meta-model that is being traced but is hidden from the traceability infrastructure.

We added an adapter for the tracepoint approach for ProR, which was easy to realize, as ProR is also built using EMF. This is true for all Eclipse-based offerings (and what the tracepoint plug-in is targeted at). The tracepoint application adds a new view to Eclipse. To set a tracepoint, the source and target are being selected in Eclipse and stored by clicking one button for each.

### 16.6.3 Integration of Domain-Specific Languages

The possibility to specify requirements with textual domain-specific languages (DSLs) and to trace these to development artifacts is one of the foundations of the VERDE project, which drove the DSL extension for ProR [18]. A textual DSL is a machine-processable language that is designed to express concepts of a specific domain. The concepts and notations used correspond to the way of thinking of the stakeholder concerned with these aspects while still being a formal notation.

In the Verde requirements editor, the open-source tool Xtext [19] has been used. The introduction of Xtext allows any project to define their own grammar and modeling. Users can design and evaluate new formal notations for the specification of requirements.

The editor for the DSLs integrates itself directly into the requirements tool and is activated as a presentation (as described in Sect. 16.5.5). Upon editing, a pop-up editor appears that gives immediate feedback to the user in the form of syntax highlighting and error markers and supports the user by providing auto-complete and tool tips, similar to what users are used to in modern programming editors.

## 16.7 Conclusion

In this chapter, we gave a broad overview of the current state in requirements engineering both in academia and industry. We then introduced the ReqIF data model for requirements, as well as Eclipse RMF and ProR. Last, we provided a few examples on how this ReqIF-RMF-ProR-stack has been used to solve real problems.

As an Eclipse Foundation project, RMF relies on the feedback of users and on contributors to thrive. We hope that we spurred some interest both in academia and industry to see what RMF is capable of and to use it for their projects.

## References

1. OMG (2011) Requirements interchange format (ReqIF) 1.0.1.http://www.omg.org/spec/ReqIF/
2. Jastram M, Graf A (2011) Requirement traceability in topcased with the requirements interchange format (RIF/ReqIF). First Topcased Days Toulouse
3. IEEE (1997) Recommended practice for software requirements specifications. Technical Report IEEE Std 830–1998, IEEE
4. Kruchten P (2004) The rational unified process: an introduction. Addison-Wesley, Boston
5. Gunter CA, Jackson M, Gunter EL, Zave P (2000) A reference model for requirements and specifications. IEEE Softw 17:37–43
6. Jackson M (2001) Problem frames: analysing and structuring software development problems. Addison-Wesley/ACM Press, Harlow/New York
7. Darimont R, Delor E, Massonet P, van Lamsweerde A (1997) GRAIL/KAOS: an environment for goal-driven requirements engineering. In: Proceedings of the 19th international conference on software engineering, ACM, Boston, MA, USA, pp 612–613
8. Parnas DL, Madey J (1995) Functional documents for computer systems. Sci Comp Program 25(1):41–61
9. Van Lamsweerde A et al. (2001) Goal-oriented requirements engineering: a guided tour. In: Proceedings of the 5th IEEE international symposium on requirements engineering, Toronto, Canada, vol 249, p 263
10. Clarke E, Wing J (1996) Formal methods: state of the art and future directions. ACM Comput Surv (CSUR) 28(4):626–643
11. Gotel O, Finkelstein A (1994) An analysis of the requirements traceability problem. In: Proceedings of the first international conference on requirements engineering, Colorado Springs, CO, U.S.A., p 94101
12. Jastram M, Hallerstede S, Leuschel M, Russo AG Jr (2010) An approach of requirements tracing in formal refinement. In: VSTTE. Springer, Edinburgh, Scotland
13. Steinberg D, Budinsky F, Peternostro M, Merks E (2009) EMF eclipse modeling framework, 2nd edn. Addison-Wesley, Upper Saddle River
14. Part B (2008) Deploy project
15. Coleman J, Jones C, Oliver I, Romanovsky A, Troubitsyna E (2005) RODIN (rigorous open development environment for complex systems). EDCC-5, Budapest, Supplementary Volume p 2326
16. Jastram M, Graf A (2011) Requirements modeling framework. Eclipse magazine 6.11
17. Jastram M, Hallerstede S, Ladenberger L (2011) Mixing formal and informal model elements for tracing requirements. In: Automated Verification of Critical Systems (AVoCS), Newcastle / UK
18. Jastram M, Graf A (2011) Requirements, traceability and DSLs in eclipse with the requirements interchange format (RIF/ReqIF). In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII. fortiss GmbH, Dagstuhl, Germany
19. Efftinge S, Völter M (2006) oAW xText: a framework for textual DSLs. In: Workshop on modeling symposium at eclipse summit, Esslingen, Germany, vol 32

# Chapter 17
# Managing Requirements Knowledge: Conclusion and Outlook

**A.K. Thurimella and W. Maalej**

**Abstract** This chapter summarises the Managing Requirements Knowledge book and concludes with the future work. For this purpose, we performed a case-by-case review of the book chapters as well as other relevant publications and extracted the research issues, main contributions, benefits or lessons learned, and future research directions.

## 17.1 Summary of the Motivations

Requirements engineering (RE) is a collaborative and knowledge-intensive activity, in which significant knowledge exists in a tacit form. Requirements knowledge consists of information useful for identifying, comprehending, implementing, and changing requirements.

Traditional requirements engineering does not manage this knowledge systematically, which leads to misunderstandings in specifications, delays in deliveries, problems in software quality, or inconsistencies in artefacts (e.g. mismatch between a requirement and its implementations). A systematic management of requirements knowledge brings the following advantages:

- It aids identification of new requirements from the knowledge that is captured in the previous projects of the company [1].
- It helps solving repetitive problems that occur in requirements engineering by systematising experiences and guiding stakeholders [2].
- It speeds up decision-making by sharing relevant information [3, 4].

A.K. Thurimella (✉)
Harman Becker Automotive Systems GmbH, Moosacher Str. 48, 80809, Munich, Germany
e-mail: anil.thurimella@gmail.com

W. Maalej
Department of Informatics/ MOBIS, University of Hamburg, Vogt-Kölln-Str. 30, 22527, Hamburg, Germany
e-mail: maalej@informatik.uni-hamburg.de

- It improves requirements reuse [2].
- It improves evolvability of requirements by providing rationales helpful to decide on future changes [5, 6, 61].
- It speeds up the analysis of requirements by improving understandability of requirements [4, 7] and reduces the mismatch between requirements and their implementations [8].
- It improves traceability by capturing implicit links [9] and identifying hidden interdependencies [10].

Despite these promising advantages, there are several challenges in the area of managing requirements knowledge. Perhaps the most crucial issue is the need of a deep *understanding of tacit knowledge* based on theoretical and empirical research. Fundamental questions include the following: What is tacit knowledge in requirements engineering? What are the types of tacit knowledge? Why is it tacit? And, where does it exist in a software engineering project? To answer these questions, the requirements engineering community can learn from other disciplines such as management science, linguistics, or social sciences, which have been studying tacit knowledge for a while [11]. In addition to these questions, RE methodologies, tools, and processes should be extended to give more room for capturing tacit knowledge and integrating it into requirements in an accessible, sharable, and reusable manner.

Another major challenge is to *avoid* an *additional overhead with the management of requirements knowledge* and adding additional tasks and information for stakeholders. We think that we need lightweight processes and intelligent tools as well as alternative, integrated requirements infrastructures, which automate the capture and access of useful information and the proactive sharing of knowledge. Furthermore, we need to bring the scientific concepts to the level of requirements engineers working in the industry.

To address the challenges and enable the advantages, this book introduces managing requirements knowledge and lays its foundations from a theoretical, practical, and empirical perspective. Furthermore, it presents approaches, methodologies, tools, and guidelines for managing requirements knowledge. From the inputs of the authors and the experience from the MaRK workshop series [12], we identified the following *five major topics* for the book:

1. Identifying Requirements Knowledge
2. Representing Requirements Knowledge for Reuse
3. Sharing Requirements Knowledge
4. Reasoning About Requirements
5. Intelligent Tool Support

This chapter concludes the book by a case-by-case review of the chapters along with the five topics listed above. The remainder of this chapter is organised as follows. Section 17.2 summarises the book chapters. Section 17.3 describes the future research issues and the road head of this emerging area. Furthermore, Sect. 17.4 concludes this chapter.

## 17.2 Summary of Requirements Knowledge Foundations

Overall, this book includes 15 core chapters apart from the introduction and conclusion. To summarise these chapters, we performed a systematic case-by-case review extracting the following information:

- Research issues reported
- Contributions to requirements engineering
- Benefits and lessons learned
- The future research proposed by the authors of the chapters

### *17.2.1 Identifying Requirements Knowledge*

The first part of the book focuses on the first step in requirements knowledge management: the identification of requirements knowledge. Table 17.1 summarises the chapters of this part. The main handled issues are as follows:

- What is requirements knowledge?
- How can requirements knowledge be captured?
- How can requirements knowledge be identified and extracted?

There is a tacit agreement amongst researchers that the term requirements knowledge mainly refers to tacit knowledge and that identifying requirements knowledge is mainly about identifying tacit knowledge [11]. While communicating about requirements, tacit knowledge remains in the minds of stakeholders. The lack of common understanding of tacit knowledge leads to the misunderstandings of requirements. In Chap. 2, Gervasi et al. [11] study tacit knowledge by reviewing the literature from various disciplines and proposing a theoretical, logic-based framework involving various communication cases between analysts and customers. For example, when customers talk to skilled analysts with sound experience, they barely state everything they know. In this situation, the requirements knowledge is assessable and expressible, but not articulated. To avoid the loss of requirements knowledge, analysts should proactively validate requirements derived from the domain knowledge with the customers. The theoretical framework in such cases helps stakeholders to mitigate the effects of tacit knowledge.

Gervasi et al. focus on tacit knowledge between a customer and an analyst. Similarly, identifying tacit knowledge between other stakeholders (e.g. between testers and developers) can also help capturing new requirements. Lutz et al. [1] report on two case studies in Chap. 3, from which they conclude that enriching feature models with bug report knowledge would aid developers to identify requirements for new products of a software product line. The authors also conclude that the variability binding times (or at which point of time in the software lifecycle, a variation point is resolved?) for the major decisions should be extended to the requirements engineering phase. The tacit knowledge is also useful to bind the variability early in the requirements engineering phase. For example, Stoiber and Glinz [13] propose an

**Table 17.1** Review of chapters on identifying requirements knowledge

| Authors | Research issues | Contributions | Benefits/lessons learned | Further research |
|---|---|---|---|---|
| Gervasi et al. [39] | (1) What is tacit knowledge? | (1) Multidisciplinary literature review on tacit knowledge | (1) Better understanding of tacit knowledge | (1) Evolution of knowledge over a long period of time |
| | (2) Can we build a theoretical framework for reasoning about tacit knowledge in RE? | (2) A predicate logic framework to specify tacit knowledge | (2) Potential tools that implement the framework to mitigate the effects of tacit knowledge | (2) Identifying the presence of tacit knowledge and its relevance |
| Lutz et al. [1] | (1) How to mine requirements knowledge from bug reports? | (1) Empirical evidence with two cases studies that were implemented at JPL | (1) Knowledge from bug reports helps to identify requirements for future products | (1) Capturing the requirements knowledge and propagating it forward in the development process to support design, implementations, and evolution |
| | (2) How to proactively communicate the knowledge to developers? | | (2) Variation points in SPLs are resolved late in the development. Primary decisions are to be made in RE | |
| Maalej and Thurimella [15] | (1) How can practitioners implement requirements knowledge management without introducing a bug overhead? | (1) Guidelines for a knowledge management in RE | (1) Practitioners can implement the guidelines to improve RE and increase reuse | (1) Proposing new guidelines |
| | | (2) Examples from 6 organisations on how these guidelines were used | (2) Lessons learned on what worked well and what not | (2) Broad empirical evidence for the guidelines (what works well in which context) |

approach to enrich variability at the level of product line requirements with rationale [14]. This information is used later to make decisions on binding and instantiating the variability.

While the first two chapters suggest how to formally identify and capture requirements knowledge setting the foundation for future tool support, in Chap. 4, we present a set of practical guidelines to aid practitioners manage requirements knowledge with little overhead. Our DUFICE guidelines [15] call for *D*rawing a knowledge landscape, *U*sing lightweight tools, *F*ollowing an iterative process, *I*nteracting with external communities, *C*apturing tacit knowledge and *E*stablishing a knowledge culture. We synthesise these guidelines from a long-term observation of six software companies. These guidelines enable stakeholders to identify and manage requirements knowledge despite their limited time and resources.

### *17.2.2 Representing Requirements Knowledge for Reuse*

Several requirements-related tasks are repetitive, are time-consuming, and require a lot of human involvement [7, 16]. For example, requirement analysis for safety critical systems includes Hazard and Operability Analysis or Failure Mode and Effect Analysis tasks in order to identify potential system hazards and risks and to mitigate them to acceptable levels before a system is certified. Another example of repetitive time-consuming tasks can be found in large contract-based projects. There, requirements elicitation typically includes the creation of a requirements specification document, which is used as a contractual document. Large IT service providers, which conduct similar projects in the same domain, typically spend valuable time on creating separate requirements specification documents for each project – with copy and paste as the only reuse instrument if at all.

Requirements knowledge should be captured such that it can be reused in case similar issues arise. As summarised in Table 17.2, this part involves a wide range of approaches for capturing knowledge for reuse, including patterns, case-based reasoning, natural language processing, ontologies, and social media.

A pattern is a knowledge representation scheme, which can be used to manage requirements. In Chap. 5, Franch et al. [17] propose a catalogue of patterns for eliciting, documenting, and reusing requirements. The authors also introduce a framework where a stakeholder would be able to instantiate relevant patterns for managing requirements. Other applications of patterns for requirements engineering include handling non-functional requirements (NFRs). Handling NFRs such as security, cost, or usability is difficult because they require a large body of multidisciplinary knowledge. Similarly, Supakkul et al. [18] propose patterns to guide engineers for handling NFRs. Supakkul approach also models relationships between the patterns, for example, generalisation or aggregation.

Reuse is particularly desired in critical, time-consuming, and repetitive tasks. Dararmola et al. [7] show in Chap. 6 how capturing requirements knowledge can support reuse for such tasks, in particular for safety analysis and hazard identification. To automate these activities, the authors proposes KROSA, an approach that uses case-based reasoning, ontologies, and natural language processing. The approach allows extracting and reusing experiences and identifying hazards early in the project. The approach uses ontologies to manage and resolve conflicts with other NFRs (e.g. safety vs. usability).

In Chap. 7, Ghaisas and Ajmeri [4] present an approach for capturing domain knowledge for reuse. The approach combines the advantages of ontologies (inference and simple linking of resources) with the advantages of Web 2.0 (democratic and collaborative evolution of knowledge). In two case studies, the authors demonstrate how reuse in their approach leads to (1) a faster and more accurate resolution of requests and (2) to jump-start a new project.

**Table 17.2** Review of chapters on representing requirements knowledge for reuse

| Authors | Research issues | Contributions | Benefits/lessons learned | Further research |
|---|---|---|---|---|
| Franch et al. [17] | (1) How can we increase requirements reuse by using knowledge representation patterns? | (1) The PABE framework for documenting and reusing requirements<br>(2) A catalogue of patterns | (1) Advantages from reuse including reduced efforts and improved quality | (1) Adoption of rules and best practices for writing pattern templates<br>(2) Extension of the catalogue with NFR patterns from several domains |
| Dararmola et al. [7] | (1) How can we automate repetitive and time-consuming activities in RE? | (1) KROSA an approach that enriches traditional RE with case-based reasoning, natural language processing, and ontologies | (1) Reuse of experience in conducting safety analysis<br>(2) Early identification of system hazards with good domain ontology | (1) A semantic framework for safety analysis and diagnostic reasoning for hazards |
| Ghaisas and Ajmeri [4] | (1) How can we capture domain knowledge for easy evolution and reuse? | (1) A platform for defining and reusing requirements using ontologies and Web 2.0 techniques<br>(2) Two case studies | (1) A faster and accurate resolution of change requests<br>(2) Faster start of new projects in the same domain | (1) The applicability and scalability of the method and tool in large projects |

## 17.2.3 Sharing Requirements Knowledge

Sharing requirements knowledge forms the bridge between its capture and reuse. This activity is of particular importance in large distributed projects, where the means for informal exchange "during the coffee break" or "a quick brainstorming" with relevant stakeholders are limited. Table 17.3 presents a summary of the chapters on sharing requirements knowledge, focussing on global, agile, and distributed projects.

In global projects, sharing requirements knowledge is difficult because of the geographical and cultural separation between stakeholders. In Chap. 8, Carillo De Gea et al. [2] argue how their PANGEA approach enables knowledge sharing in distributed projects by employing a reusable repository. The repository itself should be implemented using technologies such as semantic wikis and social networks.

**Table 17.3** Review of chapters on sharing requirements knowledge

| Authors | Research issues | Contributions | Benefits/lessons learned | Further research |
|---|---|---|---|---|
| Carrillo De Gea et al. [2] | (1) How to improve requirements knowledge sharing and reuse in global software engineering? | (1) PANGEA: a process for global requirements and quality that uses natural language requirements and software engineering standards | (1) Benefits from software reuse including [57] (1) improved timeliness, in the sense of decreased time to market; (2) reduced software maintenance efforts; (3) improved reliability, efficiency, and consistency of the developed software; and (4) enhanced investment, through the preservation of the know-how | (1) Mine requirements to assist developers making better decisions on subsequent software development phases (2) Reusing requirements knowledge for project management decisions |
| Sim and Gallardo-Valencia [19] | (1) How requirements knowledge is shared in agile projects? | (1) Findings from a field study of a project which uses scrum and user stories to capture requirements | (1) The agile community should encourage question asking to tighten performative knowledge sharing | (1) Research addressing the two issues from lessons learned |
| | | (2) Characterisation of performative knowledge sharing | (2) The knowledge management community should support users to engage in performative knowledge sharing | |
| Lim et al. [20] | (1) How can we identify stakeholders with a particular knowledge? | (1) StakeSource, a tool that uses social networks to identify and prioritise stakeholders | (1) The main factors that influence stakeholder engagement include the number of stakeholders, their location, motivation to be engaged, and their stake in the project | (1) Address the critical issue of incentives to increase stakeholder response |
| | (2) Which factors influence stakeholders engagement in sharing knowledge? | (2) Lessons learned from using StakeSource in an empirical study with 600 stakeholders | (2) The stakeholders' culture, availability, clarity of instructions, and organisation politics also affect stakeholder engagement | |

Agile methodologies do not emphasise the explicit maintenance of requirements documents. In contrast, "people have a higher priority than documentation". In Chap. 9, Sim and Gallardo-Valencia report on an empirical study on agile projects [19] and conclude that a question-asking culture should be introduced in order to share knowledge in agile development. Simultaneously, personnel management should pay attention to improving knowledge sharing.

Stakeholder analysis involves collecting data on stakeholders (e.g. their profiles, information added during discussions) from tools and social platforms and analysing the data to make some recommendations (e.g. suggesting relevant stakeholders). Supporting an automated analysis of stakeholders is helpful to identify relevant stakeholders to elicit and share knowledge. In Chap. 10, Lim et al. [20] report on an empirical study, in which more than 600 stakeholders of the StakeSource platform, a Web 2.0 tool based on social networking and cloud sourcing techniques, have been analysed. In an environment where stakeholders are motivated and are willing to contribute, the stakeholder analysis will be able to extract automatically useful information about stakeholders (see Table 17.3 for more details). Herrera and Cleland-Huang [21] previously raised the importance of stakeholder analysis. The authors used collaborative filtering techniques for automatically identifying the stakeholders for a given topic.

### *17.2.4 Reasoning About Requirements*

Reasoning about requirements and their *interdependencies* is essential for consistency and compatibility management as well as for requirements prioritisation and release planning [22]. Requirements planned for a certain release should be compatible, and requirements should not be treated independently. For instance, choosing a "low-cost/high-priority" requirement may also entail the need to include a "low-priority/high-cost" requirement. Inconsistencies between requirements are triggered by different factors such as a lack of time for consistency checking or stakeholders' different perceptions and goals [23].

In domains or organisations where stakeholders are able to work with formal models, there is a good chance for developing high-quality specifications by automatically verifying requirements. In Chap. 11, Sharma and Biswas [24] capture requirements in courteous logic, which allows for resolving the inconsistency and incompleteness issues. This approach leads to an overall improvement of requirements quality. Chapter 12 presents a rule-based approach for formally verifying product line requirements [25]. This approach enables detecting inconstancies, dead features, and defects in variability such as false optional features.

Reasoning about requirements and their interdependencies should also be propagated to the latter activities such as design and implementation. To support this, Soffer and Dori [8] propose in Chap. 13 the REM approach by integrating requirements process to design and implementation as well as by supporting traceability. Table 17.4 shows a summary of the chapters.

A recent promising approach uses Semantic Web technologies, enabling all stakeholders to collect and semantically annotate requirements, for example, in a

**Table 17.4** Review of chapters on reasoning about requirements

| Authors | Research issues | Contributions | Benefits/lessons learned | Further research |
|---|---|---|---|---|
| Sharma and Biswas [24] | (1) Can we use formal methods to reason about the quality of requirements? | (1) An approach to resolve consistency and completeness issues in requirement using courteous logic | (1) Formal mechanism to improve quality of requirements | (1) Formal analysis of requirements for detecting ambiguities<br>(2) A framework for the formal analysis of requirements |
| Elfaki [25] | (1) How can we reason about quality of requirements in the context of a large SPL | (1) A set of rules for verifying requirements expressed using a combination of feature modelling and OVM | (1) Inconsistency detection and prevention<br>(2) Detection of false-optional and dead features | (1) Extending the verification technique with case-based reasoning |
| Soffer and Dori [8] | (1) How reasoning about requirements reduces the undesired mismatch between the required system and its implementation? | (1) A new requirements engineering and management framework that is tightly coupled with the evolving conceptual model of the developed system | (1) System engineers can quickly and accurately model and specify requirements, architecture, and any structural and behavioural aspects of the system. | (1) Address these questions: To what extent does the requirements model faithfully represent the operational concepts of the system? Does the design fulfil the requirements? |
| | | (2) The integration of the proposed framework into an Object-process methodology | (2) Improved traceability between requirements and their implementation | (2) Adding rationales for traceability links |

semantic wiki [26, 27, 58]. Such approaches use ontologies (e.g. a requirements ontology and a domain ontology) with dependencies to reason about various properties of requirements [28, 29, 60]. Ghaisas and Ajmeri introduce in Chap. 7 a similar approach and showed its applicability in industrial settings.

Finally, data mining and machine learning approaches also seem to be very promising for supporting semiautomated reasoning about requirements [22, 30, 31], as, for example, discussed in Chap. 14 [32]. Generally speaking, however, approaches to reason about informally defined requirements are still rare and represent a major future direction for the field as discussed in Sect. 17.3.

### *17.2.5 Intelligent Tool Support*

Intelligent tool support helps engineers by automatically capturing and sharing requirements knowledge. This book contains chapters on emerging technologies such as recommendation systems, experience-based tools, as well as tools embedded into Integrated Development Environments such as the Eclipse Requirements Modelling Framework (RMF). Table 17.5 presents a short review of these chapters.

A recommender system is a system that guides a user in a personalised way to interesting or useful objects in a large space of possible options or that produces such objects as output [33]. In the context of requirements engineering, a recommender system would improve the efficiency of stakeholders' tasks by suggesting items such as related requirements, the preferences of other stakeholders, or a pattern that should be used to capture a special type of requirements knowledge [59].

In Chap. 14, Felfering et al. [32] introduce RE relevant recommendation technologies such as collaborative filtering, clustering, and group-based recommendations. The authors also discuss how these can be applied to the elicitation, negotiation, prioritisation, and planning of requirements.

In Chap. 15, Knauss and Meyer [34] discuss how tools can learn from previous experiences to help requirements engineers improve requirements documentation. The high-level design principles of an experience-based tool are reliability in reporting problematic requirements, authority for relying on feedback of automatic requirements checkers, proactivity in providing a degree for time of feedback, degree of interpretation ability of the tool to interpret data, and learnability on adding new experiences into the tool.

Based on these principles, the authors propose a learning model, a prototype, and heuristics on writing tense, inconsistencies, ambiguities, and incompleteness.

Another relevant aspect for intelligent tool support is the integration of requirements knowledge into other types of software engineering knowledge, in particular development knowledge. In Chap. 16, Jastram shows how this can be done with the Eclipse Requirements Modelling Framework [35], which extends the Open-Source Eclipse Development Environment. The framework also supports RIF (Requirements Interchange Format), which is futuristic XML-based standard for exchanging requirements-related information.

In addition to these tool possibilities, social platforms such as wikis provide economical tool support for requirements engineering. For example, Uenalan et al. [36] argue that traditional features of requirements engineering such as projects, folders, specification modules, traceability, and baselines may be provided by simple extensions of wikis.

## 17.3 Open Issues and Road Ahead

This section presents open issues and the road ahead in the area of managing requirements knowledge.

**Table 17.5** Review of chapters on intelligent tool support

| Authors | Research issues | Contributions | Benefits/lessons learned | Further research |
|---|---|---|---|---|
| Felfernig et al. [32] | (1) How could recommendation technologies be applied for various RE scenarios? | (1) Review of recommendation technologies (collaborative filtering, content-based filtering, clustering, knowledge-based recommendation, group-based recommendation, and social network analysis) for their application to requirements elicitation, definition, negotiation, and planning | (1) Improving the usability of requirements process and infrastructure.<br>(2) Effective and efficient RE | (1) Research agenda on various recommendation uses cases, including decision-support, preference construction, and context awareness |
| Knauss and Meyer [34] | (1) How can experience-based tools lead to better requirements documentation? | (1) Design principles, learning model, and a prototype for an experience-based requirements tool | (1) Improved requirements specification | (1) Deep evaluation of the experience-based tool |
| Jastram [35] | (1) A tool for state-of-the-art requirements knowledge | (1) Review of capabilities of eclipse requirements Modelling framework (RMF), which support formal and informal requirements approaches | (1) An open-source RE tool<br>(2) Support requirements interchange format | (1) Support of requirements knowledge approaches on open-source tools such as RMF |

### *17.3.1 Identification of Tacit Knowledge*

We can know more than we can tell [37]. Therefore, one of the most important research issues in knowledge management is the identification of "what we know but did not tell", that is, the identification of tacit knowledge. Non-functional requirements represent a common example of tacit knowledge in requirements engineering [38]. Unfortunately, these are often treated with lower importance than functional requirements and are neither discussed nor documented. However, non- functional requirements have a major influence on the design and evaluation of a system. For example, performance requirements impact the system design, its development, and deployment. These impacts often remain tacit in a project as well. After releasing a system and testing it, developers might then need to invest significant refactoring effort for improving the performance.

The main research challenges for identifying tacit knowledge in requirements engineering are twofold. First, we need to deeply *understand the nature of tacit knowledge* in engineering projects and why they are tacit. This can be achieved through large-scale empirical studies including experiments, surveys, and context analysis studies, as well as surveying and establishing bilateral collaboration to related fields including management, psychology, and social sciences [39]. Second, we need to develop processes, methodologies, and intelligent tools to support stakeholder in identifying and *externalising tacit knowledge*. Thereby, *the* main challenge lays in the *usability* rather than in the *power* of the approaches. Several authors have suggested different ways to capture and document tacit knowledge [40], also for requirements engineering as discussed in Chap. 2. However, these authors agree that the identification and extraction is more difficult than in the capturing and documentation. Approaches, which assume that stakeholders will "tell" tacit knowledge just by asking them or that they will capture this knowledge just by providing the right forms, will probably fail in practice, as several rationale management researchers have reported [11, p20–22]. A more promising way is to extract requirements knowledge by mining artefacts (e.g. as reported in Chap. 3) or by instrumenting the work environments of stakeholders and observing their behaviour [41, 42].

### *17.3.2 Studies of Information and Knowledge Needs in RE*

In addition to identifying requirements knowledge and capturing it in repositories and artefacts, we must also enable the efficient assessment of it, which means quantifying its relevance and usefulness for a particular context [39]. Not every piece of knowledge is relevant for every task and everyone. Therefore, researchers should focus in the future on studying, modelling, and predicting which piece of knowledge can be extracted from whom and is relevant to whom and when. The first step in this direction is to identify the *information needs* of the different stakeholders and conduct a *gap analysis* with the state-of-the-art tool support.

In recent years, researchers extensively studied knowledge needs in software development, unfortunately with a little to no attention to requirements engineering tasks. Ko et al. [43] observed 17 developers at Microsoft and identified 21 encountered questions such as "What is the program supposed to do" or "What code could have caused this behaviour". Similarly, Sillito et al. [44] identified questions specific to software evolution tasks. Robillard [45] found that "missing information" and "out-of-context documentation" are two of the obstacles that developers face when learning to use APIs. We think that the requirements engineering community should follow this movement focussing more on questions encountered when working with requirements. Thereby, the ultimate goal is twofold. First, such studies will help understanding and differentiating the knowledge needs of stakeholders and, hence, tightening the knowledge access and sharing. Second, the studies also help understanding and formalising the whole field of requirements knowledge, establishing taxonomies of knowledge and common standards across the community.

We think that the *quality of these studies* has a higher priority than the quantity. By quality, we mean in particular:

- A reliable and reproducible study design should be used. That is, the study instructions are clearly described with enough details so that researches would be able to replicate the study to come up with similar results (high agreement rates).
- Representative samples, summarising the different stakeholders, demographics, geographies, businesses, etc. It is also important that studies are conducted with real practitioners and not proxy participants
- Broad-spectrum and mixture of instruments including quantitative (e.g. surveys, content analyses, and experiments) and qualitative (e.g. case studies, interviews, and observations).

## *17.3.3 Reasoning About Requirements*

The detection of interdependencies between requirements is crucial for successful software projects. Such interdependencies must be made explicit early in the project to support decisions during the planning and the negotiation. Explicit interdependencies should be added to the traditional reasoning (e.g. rationale).

Informal requirements are hard to analyse. For informally defined requirements, the complete automation of consistency management is unrealistic [46], but semiautomated tools can help to keep the efforts acceptable. Here, text parsing and natural language techniques should be used to detect interdependencies between requirements semiautomatically.

Data mining approaches also seem to be very promising for supporting semiautomated reasoning about requirements. For example, Ruhe and Saliu [22] introduce a release planning approach, which is based on the concept of linear programming [31]. The basic idea is to define a linear program that should calculate a sequence of assignments of features to a corresponding release taking into account the dependencies between the different features. An optimisation algorithm [3] assigns features to releases. Ruhe et al. [30] show how to apply Analytical Hierarchy Process for determining a set of preferred requirements.

Representing requirements using formal languages allows engineers to automatically verify specifications for inconsistencies and missing requirements [24, 25]. Stakeholder preferences (e.g. weights/priorities of requirements) also influence the requirements reasoning. The major issue of existing approaches is their limited capability of handling *inconsistencies of preferences*. Given a formal description of the dependencies between a set of requirements, existing systems are able to detect inconsistencies but are not able to localise and repair them. Given a potentially inconsistent set of stakeholder preferences, existing approaches are unable to resolve inconsistencies amongst the preferences.

Researchers should develop novel approaches to reason about requirements by combining advantages of intelligent diagnosis and repair techniques [10, 30] with those of semantic wikis [26, 27, 58]. Some techniques have been successfully applied for diagnosing inconsistent knowledge bases [47], faulty process flows [28], or faulty utility definitions in utility-based recommendation [33] – but not to inconsistent requirements yet.

Another research direction is to replace pure preference elicitation [48] with a *preference construction* [49] in RE decision-making. This has been shown to be very beneficial in other domains [50]. New techniques for semiautomated dependency detection, for example, based on sentiment analysis [51], will help to make dependency detection more efficient.

### *17.3.4 Evolution of Requirements Knowledge*

Enhancing the longevity of software-intensive systems, which are often maintained over decades, is a real challenge. From the requirements perspective, the main difficulty is to allow for changes without reducing the maintainability and evolvability of the whole software. One of the recurrent criteria for decisions on requirements is to ensure optimal changes and minimise unanticipated changes. Techniques to reason about requirements interdependencies can help addressing this challenge.

Capturing tacit knowledge also facilitates the evolution of requirements [5]. Stakeholders are able to decide about the evolution by reviewing rationale documentation available from similar decisions. Moreover, by reviewing alternative solutions, engineers are able to identify new requirements and forecast future issues.

As requirements evolve, the knowledge around them evolves too. In this case, it becomes difficult to simply manage versions and configuration items for requirements knowledge (e.g. rationales, presuppositions, or experiences) in a lightweight manner without introducing overhead to software projects. There are also interdependencies between issues, alternative, and decisions, which can be logical (e.g. requires and conflicts) or temporal (e.g. triggers) [52]. These interdependencies also change over time – making everything changing over time. Researchers should focus on *modelling* *the requirements knowledge with time*, for example, by adopting concepts such as evolving knowledge bases [53].

Finally, prediction models, which use requirements knowledge to predict the evolution and maintenance, represent a major research direction as well. For example, a model that forecasts the number of changes that will occur in a time period will help stakeholders to plan their resources for proper reaction and change handling.

### 17.3.5 From the Information to the Recommendation Age

Managing requirements knowledge introduces new problems. Additional information is created, and additional effort is needed to identify and find information in requirements repositories. This would mean overhead of capturing, maintaining, and accessing requirements knowledge.

To address these problems, researchers should focus on data modelling and management and recommendation technologies. For example, a system that proactively recommends how similar issues were solved previously would aid engineers to automatically get relevant information. Similarly, using experience-based learning models [34] enable automatic capture of experiences. Traditional requirements databases should be enhanced such that data is modelled and stored in a way that allows learning and querying for recommendations. Furthermore, recommendation technologies [32] should extend existing requirements infrastructures and tools. A research agenda for recommendation systems in RE is available at [41].

Recommendation systems would strongly influence the future of requirements engineering. Here, the focus should be on supporting recommendation technologies for various use cases of requirements engineering [19, 32, 34]. Similarly, researchers should also focus on decision-support systems where several aspects of decision-making (e.g. decisions, related issues, etc.) could be recommended. In Chap. 14, Felfernig et al. suggest research on constructing preferences (e.g. weights) for criteria such as goals or non-functional requirements. Additionally, data mining and machine learning techniques should be applied to get prediction models from which engineers/project managers could make predictions on the subsequent stages of software development [7, 28].

### 17.3.6 Democratisation and Active Involvement

The use of social media would support collaboration, information exchange, and interoperability in requirements management. For example, a company together with its client might use social media for collecting users' feedback and elicit requirements directly from the users [39, 42]. The use of social media improves the transparency (e.g. origin of requirements?) and understandability of requirements because of additional contextual information and formalism of requirements.

Increasing the involvement of end users and "democratising" requirements activities trigger several challenges. End users want to focus on performing their tasks and not on improving the software. In an ideal case, their contribution should be at almost zero cost. In addition, collecting (usage) data, implicit feedback, and explicit feedback will lead to a *huge* amount of possibly *conflicting* data. The processing of these data should be automated as much as possible.

The next research direction is the proactive sharing of knowledge between distributed stakeholders. Here, the challenge is to identify relevant stakeholders for communicating information. Machine learning techniques (e.g. collaborative filtering [21]) can be applied for identifying relevant stakeholders. Social platforms such as StakeSource could be automatically analysed to identify expert stakeholders [40]. For enabling automated analysis, it is important that stakeholders are enthusiastic enough in making contributions on the social platform. Therefore, incentives should be provided for improving stakeholder enthusiasm. In particular, it should be researched how to motivate stakeholders to contribute and use requirements knowledge [14].

Agile methodologies are collaboration intensive, that is, tacit knowledge is involved in various collaboration patterns (e.g. between an analyst and a customer [39]). Therefore, managing requirements knowledge is an important research area in agile development as well, even though agile principles emphasise on developing working code over maintaining specifications. Here, researchers should propose and implement lightweight knowledge management principles such as promoting a question-asking culture for eliciting and sharing requirements knowledge [19]. Moreover, researchers should come up with methods and frameworks, which allow using agile methods and the tool-supported capturing, access, and sharing of knowledge.

### *17.3.7 Integration into Software Engineering Knowledge*

In the last 5 years, while organising the MaRK workshop and editing this book, we encountered the following recurrent question: "What exactly is requirements knowledge and how does it differ from other types of software engineering knowledge such as design or implementation knowledge?" In the introduction chapter, we gave a definition for requirements knowledge and for managing requirements knowledge, highlighting the impacts not only on analysts and requirements engineers but also on all stakeholders.

We think it does not really matter for the single stakeholders and the whole software projects, whether the knowledge is classified as requirements knowledge or other knowledge. We think that requirements knowledge can be identified and used in any phase and task in a software project: from the requirements elicitation to the maintenance. The captured requirements knowledge should not be localised to requirements engineering and should be transmitted forward for architects [54] and developers [1]. For example, the missing requirements that are identified by eliciting tacit knowledge are helpful for architects to identify design objects. Researchers should develop and evaluate approaches for identifying and creating design and implementation artefacts as well as traceability links from the requirements knowledge.

We think therefore that the integration of requirements knowledge with and in other types of knowledge is of a particular importance. Guidelines are helpful for practitioners to manage requirements knowledge in practical settings. Based on our experience, we proposed the DUFICE guidelines targeting industry people. These guidelines should be extended, and additional guidelines should be proposed. Furthermore, guidelines as well as experiences should be shared on online platforms and communities (e.g. [4]).

### *17.3.8 Pragmatic Need-Driven Solutions and Tools*

From the practitioners' point of view, stakeholders should be guided when managing requirements knowledge in real projects. As a first step, we have aggregated a set of guidelines [15] from previous experience and discussion in the community. These should be extended, and the experiences on these guidelines should be shared in an online community such as MaRK [55].

Pragmatic tool support also means investigating open-source solutions, which can be tightly integrated to other activities and tools in the software lifecycle, for example, the Eclipse Requirements Modelling Framework [35]. Furthermore, requirements knowledge approaches should be deeply evaluated for scalability (e.g. [17]), that is, their applicability should be tested in large projects [4].

Tool support is an important topic for the requirements engineering community. In this book, the authors described several prototypes (e.g. [28, 34]) as well as emerging tools (e.g. [35]). Tool support for managing requirements knowledge is a challenging task because there are no tools reported for practitioners.

The requirements repositories used in the industry mainly contain unstructured or structured text. Introducing intelligent functionality (e.g. recommendation systems, experience-based tools, etc.) for those repositories is another challenge for tool support. In addition, there is a gap between the tool vendors and the needs of users [56]. The tool vendors tend to negotiate with companies, and there is less interaction with the users. Furthermore, a variety of tools is available, which makes it difficult for users to select the appropriate tool for their needs. The tool community should also focus on bridging this gap.

## 17.4 Conclusion

Assuming that engineering, understanding, and implementing requirements are knowledge-intensive tasks that affect all software project stakeholders, we have identified "managing requirements knowledge" as an emerging area in software engineering. Systematically managing requirements knowledge improves the comprehension of requirements and supports the engineering, collaboration, and management activities. This book introduces theoretical concepts as well as state-of-the-art tools, methods, and practices for managing requirements knowledge.

In this chapter, we have summarised the main concepts discussed in this book by reviewing research questions, contributions, benefits, and future research directions reported across the chapters. We have organised our review based on the five fundamental topics of the book: identifying requirements knowledge, representing requirements knowledge for reuse, sharing requirements knowledge, reasoning about requirements, and intelligent tool support. We have also discussed the open issues and our vision on the long-term road ahead in the area.

**Acknowledgements** We thank Rick Rabiser and Yang Li for providing a valuable feedback on this chapter.

## References

1. Lutz R, Lavin M, Lux J, Peters K, Rouquette NF (2013) Mining requirements from operational experience. In: Managing requirements knowledge (Chapter 3 in this volume). Springer, Heidelberg, Germany
2. Carrillo de Gea JM, Nicolás J, Alemán JLF, Toval A, Vizcaíno A, Ebert C (2013) Reusing requirements in global software engineering. In: Managing requirements knowledge (Chapter 8 in this volume). Springer, Heidelberg, Germany
3. Classen A, Heymans P, Schobbens P (2008) What's in a feature: a requirements engineering perspective. FASE'08, Lecturer notes in computer science, vol 4961. Budapest, pp 16–30
4. Ghaisas S, Ajmeri N (2013) Knowledge-assisted ontology-based requirements evolution. In: Managing requirements knowledge (Chapter 7 in this volume). Springer, Heidelberg
5. Dutoit A, Paech B (2003) Eliciting and maintaining knowledge for requirements evolution. In: Aurum A, Jeffery R, Wohlin C, Handzic M (eds) Managing software engineering knowledge. Springer, New York
6. Thurimella AK, Bruegge B (2012) Issue-based variability management. Inform Softw Technol 54(9):933–950
7. Daramola O, Stålhane T, Omoronyia I, Sindre G (2013) Using ontologies and machine learning for hazard identification and analysis. In: Managing requirements knowledge (Chapter 6 in this volume). Springer, Heidelberg, Germany
8. Soffer A, Dori D (2013) Model-based requirements engineering framework for systems lifecycle support. In: Managing requirements knowledge (Chapter 13 in this volume). Springer, Heidelberg
9. Narayan N, Delater A, Paech B, Bruegge B (2011) Enhanced traceability in model-based CASE tools using ontologies and information retrieval. In: Proceedings of the 4th international workshop on managing requirements knowledge (MaRK'11), Trento, 30 Aug 2011
10. Hull E, Jackson K, Dick J (2004) Requirements engineering. Springer, London
11. Gervasi V, Gacitua R, Rouncefield M, Sawyer P, Kof L, Ma L, Piwek P, de Roeck A, Willis A, Yang H, Nuseibeh B (2013) Unpacking tacit knowledge for requirements engineering. In: Managing requirements knowledge (Chapter 2 in this volume). Springer, Heidelberg, Germany
12. Maalej W, Thurimella A (2008–2010) Managing requirements knowledge, international workshop on, 2008–2010, IEEE, Barcelona/Atlanta/Sydney/Trento
13. Stoiber R, Glinz M (2009) Modelling and managing tacit product line requirements knowledge. In: Proceedings of the 2009 second international workshop on managing requirements knowledge (MARK '09). IEEE Computer Society, Washington, DC, pp 60–64
14. Liang P, Avgeriou P, He K (2010) Rationale management challenges in requirements engineering. In: Proceedings of the third international workshop on managing requirements knowledge (MARK), Sydney, Australia, pp 16–21
15. Maalej W, Thurimella A (2013) DUFICE: practical guidelines for managing requirements knowledge. In: Managing requirements knowledge (Chapter 4 in this volume). Springer, Heidelberg, Germany
16. Smith S, Harrison M (2005) Measuring reuse in hazard analysis. Reliab Eng Syst Safe 89(1): 93–104

17. Franch X, Quer C, Renault S, Guerlain C, Palomares C (2013) Constructing and using software requirements patterns. In: Managing requirements knowledge (Chapter 5 in this volume). Springer, Heidelberg, Germany
18. Supakkul S, Hill T, Oladimeji EA, Chung L (2009) Capturing, organizing, and reusing knowledge of NFRs: an NFR pattern approach, managing requirements knowledge, Atlanta, USA, pp 75–84
19. Sim SE, Gallardo-Valencia GE (2013) Performative and lexical knowledge sharing in agile requirements. In: Managing requirements knowledge (Chapter 9 in this volume). Springer, Heidelberg, Germany
20. Lim SL, Damian D, Ishikawa F, Finkelstein A (2013) Using Web 2.0 for stakeholder analysis: StakeSource and its application in ten industrial projects. In: Managing requirements knowledge (Chapter 10 in this volume). Springer, Heidelberg, Germany
21. Castro-Herrera C, Cleland-Huang J (2009) A machine learning approach for identifying expert stakeholders, managing requirements knowledge, international workshop on, second international workshop on managing requirements knowledge, Atlanta, USA, pp 45–49
22. Ruhe G, Saliu M (2005) The art and science of software release planning. IEEE Computer Society, IEEE Softw 22(6): 47–53
23. Dardenne A, van Lamsweerde A, Fickas S (1993) Goal-directed requirements acquisition. Sci Comput Prog 20:3–50
24. Sharma R, Biswas KK (2013) Resolving inconsistency and incompleteness issues in software requirements. In Managing requirements knowledge (Chapter 11 in this volume). Springer, Heidelberg, Germany
25. Elfaki A (2013) Automated verification of variability models using first order logic. In: Managing requirements knowledge (Chapter 12 in this volume). Springer, Heidelberg, Germany
26. Lohmann S, Heim P, Auer S, Dietzold S, Riechert R (2008) Semantifying requirements engineering – the softWiki approach, I-SEMANTICS, Graz, pp 182–185
27. Lohmann S, Riechert T, Auer S (2008) Collaborative development of knowledge bases in distributed requirements elicitation. Software engineering (Workshops): agile knowledge sharing for distributed software teams, Munich, Germany pp 22–28
28. Felfernig A, Friedrich G, Jannach D, Stumptner M, Zanker M (2003) Configuration knowledge representations for semantic web applications. AIEDAM 17(2):31–50
29. Haarslev V, Möller R (2001) RACER system description. In: IJCAR 2001, LNAI, vol 2083. Siena, pp 701–705
30. Ruhe G, Eberlein A, Pfahl D (2003) Trade-off analysis for requirements selection. Int J Softw Eng Knowl Eng (IJSEKE) 13(4):354–366
31. Schrijver A (1998) Theory of linear and integer programming. Wiley, New York
32. Felfernig A, Ninaus G, Grabner H, Reinfrank F, Weninger L, Pagano D, Maalej W (2013) An overview of recommender systems in requirements engineering. In: Managing requirements knowledge (Chapter 14 in this volume). Springer, Heidelberg, Germany
33. Burke R (2000) Knowledge-based recommender systems. Encyclop Libr Inform Syst 69(32): 180–200
34. Knauss E, Meyer S (2013) Experience-based requirements engineering tools. In: managing requirements knowledge (Chapter 15 in this volume). Springer, Heidelberg, Germany
35. Jastram M (2013) The eclipse requirements modelling framework. In: Managing requirements knowledge (Chapter 16 in this volume). Springer, Heidelberg, Germany
36. Uenalan O, Riegel N, Weber S, Doerr J (2008) Using enhanced wiki-based solutions for managing requirements, first international workshop on managing requirements knowledge (MARK), Barcelona, Spain, pp 63–67
37. Polanyi M (1966) The tacit dimension. The University of Chicago Press, Garden City
38. Glinz M (2007) On non-functional requirements. In: IEEE RE2007, New Delhi, India, pp 21–26

39. Ali R, Solis C, Omoronyia I, Salehie M, Nuseibeh B (2012) Social adaptation: when software gives users a voice. In: Proceedings of the 7th international conference on evaluation of novel approaches to software engineering (ENASE 2012), Wroclaw, 29–30 June 2012
40. Dutoit A, McCall R, Mistrik I, Paech B (2006) Rationale management in software engineering. Springer, Berlin
41. Maalej W, Happel H, Rashid A (2009) When users become collaborators: towards continuous and context-aware user input. In: Proceedings of OOPSLA 2009 (Onward!), ACM, Orlando, USA
42. Maalej W Pagano D (2011) On the socialness of software. In: Proceedings of the international conference on social computing and its applications, IEEE, Sydney, Australia
43. Ko AJ, DeLine R, Venolia G (2007) Information needs in collocated software development teams. In: Proceedings of the 29th internatinal conference on software engineering, Minneapolis, USA, pp 344–353
44. Sillito J, Murphy GC, De Volder K (2008) Asking and answering questions during a programming change task. Trans Softw Eng 34:434–451
45. Robillard MP (2009) What makes APIs hard to learn? Answers from developers. IEEE Softw 26:27–34
46. Iyer J, Richards D (2004) Evaluation framework for tools that manage requirements inconsistency. In: 9th Australian workshop on requirements engineering (AWRE'04), Adelaide, Australia
47. Felfernig A, Friedrich G, Jannach D, Stumptner M (2004) Consistency-based diagnosis of configuration knowledge bases. Artif Intell 152(2):213–234
48. Grether D, Plott C (1979) Economic theory of choice and the preference reversal phenomenon. Am Econ Rev 69(4):623–638
49. Bettman J, Luce M, Payne J (1998) Constructive consumer choice. J Cons Res 25(3):187–217
50. Felfernig A, Friedrich G, Isak K, Shchekotykhin K, Jannach D, Teppan E (2009) Automated debugging of recommender user interface descriptions. J Appl Intell 31(1):1–14, Springer
51. Hu M, Liu B (2004) Mining and summarizing customer reviews. In: Proceedings of the 10th ACM SIGKDD international conference on knowledge discovery and data mining, Seattle, USA, pp 168–177
52. Zimmermann O, Koehler J, Leymann F, Polley R, Schuster N (2009) Managing architectural decision models with dependency relations, integrity constraints, and production rules. J Syst Softw 82(8):1249–1267
53. Leite JA (2002) Evolving knowledge bases, frontiers in artificial intelligence and applications, IOS Press, Amsterdam, Netherlands
54. Avgeriou P, Grundy J, Hall JG, Lago P, Mistrík I (2011) Relating software requirements and architectures. Springer, New York
55. MaRK Community www1.cs.tum.edu/mark/community
56. Jones P (2011) Can requirements tool vendors tell us about user needs? Fourth international workshop on managing requirements knowledge (MARK), Trento, Italy, pp76–81
57. Meyer B (1997) Object-oriented software construction, 2nd edn. Prentice-Hall Inc, Upper Saddle River
58. Hoenderboom B, Liang P (2009) A survey of semantic wikis for requirements engineering, technical report RUG-SEARCH-09-L03, University of Groningen
59. Maalej W, Thurimella A (2009) Towards a research agenda for recommendation systems in requirements engineering. In: Proceedings of the 2nd international workshop on managing requirements knowledge, Atlanta
60. Mairiza D, Zowghi D (2010) An ontological framework to manage the relative conflicts between security and usability requirements. In: Proceedings of the third international workshop on managing requirements knowledge (MARK), Sydney, Australia, pp 1–6
61. Schubanz M, Pleuss A, Botterweck G, Lewerentz C (2012) Modeling rationale over time to support product line evolution planning. In: VaMoS 2012, Leipzig, Germany, pp 193–199

# About the Editors

## Walid Maalej

Walid Maalej is a professor of informatics at the University of Hamburg, Germany. Previously, he has been leading a research group on context and human aspects in software engineering at the Technische Universität München (TUM), where he also received his master's and doctorate degrees. For this book, Walid brings a unique multidisciplinary and internationally renowned research profile covering the fields of knowledge management, empirical software engineering, recommender systems, and requirements engineering. Walid served as a member of the Program Committee of renowned conferences, including ICSE, RE, and ESEC/FSE. He chaired the Industrial Track of RE10 and co-organized several international workshops, e.g., on Recommendation Systems for Software Engineering and Social Software Engineering.

## Anil Kumar Thurimella

Dr. Anil Kumar Thurimella is leading the requirements engineering and management activities for the BMW Program of Harman Automotive Division. He is also an external scientist at the Technische Universität München (TUM), where he received his master's and doctorate degrees in computer science. He has several years of industrial experience on requirements engineering, software product lines, software architecture, and empirical software engineering. For this book, Anil contributes a unique blend of industry and academic experience on requirements engineering. He has published several articles in leading software engineering conferences and journals.

# Index