Comput Sci Res Dev (2015) 30:363–376
DOI 10.1007/s00450-014-0256-x

SPECIAL ISSUE PAPER

On shared understanding in software engineering: an essay

Martin Glinz · Samuel A. Fricker

Published online: 1 July 2014
© Springer-Verlag Berlin Heidelberg 2014

Abstract Shared understanding is essential for efficient software engineering when the risk of unsatisfactory outcome and rework of project results shall be low. Today, however, shared understanding is used mostly in an unreflected, ad-hoc way. This affects the quality of the engineered software solutions and generates re-work once the quality problems are discovered. In this article, we investigate the role, value, and usage of shared understanding in software engineering. We contribute a reflected analysis of the problem, in particular of how to rely on shared understanding that is implicit, rather than explicit. After an overview of the state of the art we discuss forms and value of shared understanding in software engineering, survey enablers and obstacles, compile existing practices for dealing with shared understanding, and present a roadmap for improving knowledge and practice in this area.

Keywords Shared understanding · Software engineering · Implicit shared understanding

1 Introduction and motivation

Shared understanding between stakeholders and software engineers is a crucial prerequisite for successful development and deployment of any software system [8,17,37,55]. A *stakeholder* is a person or organization that has a (direct or indirect) influence on a system's requirements [30]. End users, customers, operators, and managers are typical examples of stakeholders. A *software engineer* is a person involved in the specification, design, construction, deployment, evolution, and maintenance of software systems. Requirements engineers, architects, developers, coders, and testers are examples of common software engineering roles.

In traditional development projects, shared understanding among and between stakeholders and software engineers is achieved by eliciting requirements and producing a comprehensive requirements specification. In agile environments, stories, up-front test cases, and early and continuous validation of an incrementally developed software system serve the same purpose.

Shared understanding among a group of people has two facets: explicit shared understanding (ESU) is about interpreting explicit specifications,[1] such as requirements, design documents, and manuals, in the same way by all group members. Implicit shared understanding (ISU) denotes the common understanding of non-specified knowledge, assumptions, opinions, and values. The shared context provided by implicit shared understanding reduces the need for explicit communication [51] and, at the same time, lowers the risk of misunderstandings.

Explicit shared understanding based on specifications is rather well understood today. In particular, research and practice in Requirements Engineering have contributed practices for eliciting requirements, documenting them in specifications and validating these specifications, see, for example, [2,3,23,25,50,60]. In contrast, the role and value of *implicit*

M. Glinz (✉)
Department of Informatics, University of Zurich, Zurich, Switzerland
e-mail: glinz@ifi.uzh.ch

S. A. Fricker
Software Engineering Research Laboratory, Blekinge Institute of Technology (BTH), Karlskrona, Sweden
e-mail: samuel.fricker@bth.se

[1] In the normal case, explicit specifications are captured in writing. Principally, however, explicit verbal communication remembered by all team members is also a form of explicit shared understanding, albeit a rather volatile one.

shared understanding is frequently neither clear nor reflected in current practice and research.

This article contributes an essay on shared understanding in software engineering. It reflects about the role, forms and value of shared understanding, identifies enablers and obstacles for achieving shared understanding, and compiles a list of practices related to shared understanding. It then focuses on implicit shared understanding, reflecting about its value and risk and describing when and how software practitioners can and should rely on implicit shared understanding. We deliberately present our research in the form of an essay; see Sect. 2 for our rationale to do so.

Our work on shared understanding is rooted in the first author's previous work on alternative forms for specifying quality requirements which includes considerations about the cost and benefit of requirements [27], and the second author's work on communicating requirements by handshaking with implementation proposals instead of writing large explicit specifications [21,22].

This article is a revised and extended version of an invited contribution presented as a keynote talk by the first author at the GI Conference "Software Engineering 2013" [29].

The remainder of the paper is structured as follows. Section 2 briefly presents our research goals and methodology. Section 3 gives an overview of existing work on shared understanding. Section 4 reflects the role, forms, and value of shared understanding. Sections 5 and 6 present enablers, obstacles, and practices. Section 7 discusses when and how to rely on *implicit* shared understanding. Section 8 briefly looks at *explicit* shared understanding. Section 9 presents a research roadmap and Sect. 10 concludes.

2 Research goals and methodology

2.1 Research goals and questions

The goal of our research that lead to this article and its predecessor [29] was to present a reflected overview of the role and value of shared understanding in the context of software engineering and to survey the current practices. From this goal we derived four research questions:

- RQ1 What is the role and importance of shared understanding in the context of software engineering?
- RQ2 How can we classify the various forms of shared understanding?
- RQ3 What are the enablers, obstacles and practices of shared understanding in software engineering?
- RQ4 How and to what extent can we rely on implicit shared understanding?

The answers to RQ1 and RQ2 provide us with definitions and a classification framework, thus framing the treatment of the remaining research questions. The answers to RQ3 and RQ4 give pragmatic meaning to the term *shared understanding* by elaborating its importance and consequences for practice. This includes approaches for building and assessing shared understanding, taking advantage of it, and dealing with problems that emerge from lack of shared understanding.

2.2 Research method

With respect to research methodology, we are using reflection and introspection by the two co-authors combined with an overview of the existing literature on shared understanding. Accordingly, we deliberately chose the form of an essay for this article: we aim at giving a grounded and reflected overview of the topic, but neither claim to present a systematic and comprehensive literature review nor contribute new empirical results. Nevertheless, we believe that a systematic, reflected presentation of shared understanding in software engineering constitutes a worthwhile contribution to our discipline.

While systematic reflection and introspection by experts is not a standard research methodology, it is nevertheless a useful and proven means for developing a body of knowledge. This form of work has a long tradition; eventually reaching back to the ancient Greek philosophers, where, in his dialogue *Theaetetus*, Plato lets Socrates state: "…why should we not calmly and patiently review our own thoughts, and thoroughly examine and see what these appearances in us really are?".[2]

Methodologically, our approach has some similarities to studies that involve elicitation and analysis of expert opinion or practice. We replace the data collection method of expert interviews by authors' introspection and reflection. Provided that the authors are true experts in the field and experienced enough both in research and practical settings, this approach provides the advantage of more systematic compilation and presentation of the knowledge than interview-based research.

To some extent, our approach is also comparable to the theory-building of grounded theory, where the researchers' insights are used to select information sources and frame questions as the research unfolds [52].

2.3 Threats to validity

Obviously, *researcher bias* is the most critical threat to the validity of our work, as our own knowledge and experience

[2] The dialogues of Plato: translated into English with analyses and introductions by Benjamin Jowett, vol. 4, 3rd edn, Oxford University Press, 1892, p 209. Retrieved 2013-07-27 at http://oll.libertyfund.org/?option=com_staticxt&staticfile=show.php%3Ftitle=768&Itemid=27#a_pdf.

frame the research and set its focus. We have contained this threat by basing our findings not just on our own experience, but also on the results of the existing literature. Also, having two co-authors and using a critical dialogue between them helped avoid researcher bias.

Reliability in qualitative research refers to being thorough, careful, and honest in carrying out the research. By critically reflecting on our research approach we aim at avoiding reliability threats.

Reactivity and respondent bias are can be excluded as a source of threats to validity. Reactivity refers to the way in which the researchers' presence interferes with the setting. Respondent bias refers to the withholding or sharing of wrong information. As our experience was collected before we undertook this study, there was no possibility that the presented research affected the sources of the presented results.

Threats to *construct validity*, i.e., the quality of the chosen variables and measurements can be excluded as our study is not based on quantitative experimentation.

Generalizability or external validity, finally, is a strength of the chosen research approach, at least in comparison to case study research. The paper synthesizes experiences made in a large number of settings from small-scale prototype development of radically new software solutions, to large-scale product line development in multi-national and cultural settings, and to maintenance of existing software. The presented findings also root in the results of a large body of literature on shared understanding. Thus the threat to external validity can be considered to be low.

3 The state of the art

Shared understanding plays a central role in many areas: ranging from fundamental issues of human communication by using language (e.g., Clark's theory of common ground [11]) over general issues of shared cognition and collaborative work up to concrete software engineering problems, particularly in requirements engineering. The areas of knowledge engineering [41] and ontologies [34] also are closely related to shared understanding.

Hence, any attempt to comprehensively survey the whole body of literature in this field would go far beyond the scope and goals of this article.

What we present in this section is a survey of selected work that we believe to be representative for characterizing the current state of the art in shared understanding. We do not aim at a systematic and comprehensive literature review of the vast amount of work on shared understanding.

3.1 Language and communication

Language and communication are fundamental enablers of reaching shared understanding. Language is the means that humans use to express information, while communication is the act and process of exchanging such information.

In the framework of a theory of language use, Clark [11] develops a theory of "common ground". According to Clark, language use is "a form of joint action" between conversing actors. Such joint action requires some basis shared by the actors as a "common ground". This is the "knowledge, beliefs, and suppositions they (the actors) believe they share". So common ground constitutes the sum of all shared understanding a group of people has, in a very universal sense. When humans interact, for example by using language or by acting jointly, they do this on the basis of their existing common ground. Through their interaction, they extend or adjust their common ground. In contrast to the universal and all-encompassing concept of common ground, our notion of shared understanding focuses on shared knowledge, beliefs and suppositions in some given software engineering context. We also object Clark's notion that "common ground is a form of self-awareness" ([11], p. 120). In our experience, many of the beliefs and suppositions that the members of a group of people actually or supposedly share are subconscious and not reflected.

Corvera Charaf et al. [13] focus on linguistic theory for achieving language-based shared understanding of requirements in information system development. They treat information system design as a linguistic communication process. For achieving shared understanding, they develop a pattern for semantic alignment, based on four categories: definition, request, reassurance, and adjustment. The approach has been developed based on data from a real world project. However, as the data come from a single project, the external validity of the approach is limited. Also, the authors focus on linguistic communication only and do not consider other factors influencing shared understanding such as context or culture.

3.2 Shared cognition

Shared cognition, the joint thinking, understanding, learning, and remembering among communicating people, is one of the foundations for building shared understanding.

Cannon-Bowers and Salas [10] investigate fundamental issues in shared cognition. They first investigate the value of the notion of shared cognition and then discuss four fundamental questions: "(1) What must be 'shared'? (2) What does 'shared' mean? (3) How should 'shared' be measured? and (4) What outcomes do we expect shared cognition to affect?"

Van den Bossche et al. [56] investigate team learning as a means for building shared mental models.

3.3 Collaborative work

Collaboration, the joint work of people to achieve an objective, depends on shared understanding.

Bittner and Leimeister [8] stress the importance of shared understanding for the performance of collaborative groups. They provide a systematic process that supports groups to converge towards shared understanding. The process is based on construction of meaning by group cognition and constructive conflict resolution. Their approach has been tested in a requirements elicitation workshop.

Piirainen et al. [45] provide a literature study about challenges in collaborative design. Creating shared understanding is one of the five main challenges that they identify in their study.

Puntambekar [46] investigates the role of collaborative interactions for building shared knowledge.

Macaulay [42] investigates the role of cooperation in understanding user needs and requirements, thus demonstrating the close relationship between the areas of collaborative work and requirements engineering.

Olson and Olson [44] use Clark's theory of common ground [11] for explaining the difficulties of building shared understanding when collaborative work is performed over geographical distance.

3.4 Software and requirements engineering

Software engineering is considered to be knowledge-intensive, thus making it particularly dependent on shared understanding. The management of knowledge to enable collaboration within the software team and between the team and its stakeholders has received much attention [9].

Tan [55] presents an empirical study on how to establish mutual understanding in systems design. She found some evidence that three processes help establish mutual understanding between system analysts and stakeholders: "(1) shifting perspective, (2) managing transaction, and (3) establishing rapport".

Whitehead [58] discusses collaboration in software engineering. He states that model-based collaboration is an important means for achieving shared meaning.

Couglan and Macredie [14] discuss the importance of achieving shared understanding for effectively eliciting and communicating requirements. They argue that an emergent and collaborative approach is crucial for successful requirements elicitation and, eventually, better shared understanding.

Cooperative requirements capture [42] and co-operative inquiry [2] stress the importance of cooperative work in requirements engineering for achieving shared understanding among stakeholders and between stakeholders and developers.

Sutcliffe [53] analyzes and frames the development of mutual understanding on the basis of Clark's theory of common ground [11]. However, he does not explicitly distinguish between explicit shared understanding based on artifacts and implicit shared understanding.

Arikoglu [4] investigates the impact of using scenarios and personas on requirements elicitation and design. In particular, in a series of four experiments, she tests the hypothesis that the usage of scenarios and personas improves shared understanding. The results are statistically inconclusive. However, it is not clear whether there is actually no measurable effect or whether inadequate criteria were used for measuring shared understanding.

Hsieh [38] identifies geographical, temporal, organizational, and cultural boundaries as obstacles for shared understanding in distributed requirements engineering. She introduces a theoretical framework for investigating the impact of culture, but does not present any concrete results.

Hadar et al. [35] investigate the effect of domain knowledge on elicitation with interviews and indirectly also on shared understanding. They found both positive and negative effects, the former ones being stronger.

McKay [43] proposes a technique called cognitive mapping for achieving shared understanding of requirements.

Hill et al. [36] try to identify shared understanding by analyzing the similarity of documents produced by team members, based on latent semantic analysis.

Stapel [51] contributes a theory of information flow in software development.

3.5 Other areas

Areas other than software engineering and collaborative work also depend on shared understanding.

Darch et al. [17,18] investigate the problem of shared understanding in collaborative e-science projects. In a study of three real world cases, they identify several obstacles to shared understanding in an e-science context, in particular cultural and organizational differences between collaborating institutions, geographically distributed projects, and different work practices and habits in the involved institutions. Darch et al. [18] also contains an extensive bibliography of work on shared understanding.

3.6 Gaps in the current literature

While there is a large body of work on shared understanding, we did not find any systematic treatment or classification of the different forms of shared understanding, neither in general nor in a software engineering context. Neither, to the best of our knowledge, is there any work that collects practices, enablers and obstacles. Finally, there is no work that investigates the role and importance of implicit shared understanding in software engineering. In the remainder of this article, we aim at filling these gaps.

4 Shared understanding in software engineering: what and why

4.1 Forms of shared understanding

To better understand the notion of shared understanding, it is useful to distinguish different forms of shared understanding as shown in Fig. 1. We already have discussed implicit and explicit shared understanding in the introduction. Next it is important to note that shared understanding can be *true* or *false*. False shared understanding means that a group of people believes to have shared understanding about some issue while in fact there are misunderstandings that may or may not have been noticed.

In any software development or evolution endeavor there is a *context boundary* that separates information which is relevant for the system to be built from information which is irrelevant [28]. Note that building and assessing shared understanding about irrelevant information constitutes a waste of effort. Also, there is typically information which is relevant, but has not yet been noticed by anybody of the persons involved. We call this *"dark" information*. Sutcliffe and Sawyer [54] speak of "unknown unknowns".

We illustrate the various forms of shared understanding by taking the problem of a road construction site with one-lane traffic controlled by two traffic lights as an example. Assume that we need to develop a traffic light control system for managing alternating one-lane traffic, with a team of stakeholders and developers. (1) If the notion that a red light is a stop signal is shared by all team members, we have *implicit shared understanding*. (2) If there exists an explicit requirement stating "The stop signal shall be represented by a red light", we have *explicit shared understanding*, provided that all team members interpret this requirement in the same way. (3) If nothing is specified about the go signal, because the stakeholders take it for granted that the go signal can be either a green light or a flashing yellow light, while the developers believe that the go signal must be implemented as a green light, we have a misunderstanding, i.e., *false implicit shared understanding*. (4) Assume there is an explicit requirement "The system shall support flashing yellow lights" without any further requirements about flashing yellow lights. In this case, a stakeholder might mean 'flashing yellow on one side and red on the opposite side' while a developer might interpret this as 'flashing yellow on both sides'. If this misunderstanding goes undetected, it results in *false explicit shared understanding*. (5) The color of the box which holds the control computer for the traffic lights is irrelevant in the given context. So explicitly specifying that this box must be painted in dark green and making sure that everybody understands this requirement yields *explicit shared understanding of irrelevant information*. (6) Finally, imagine that there is a legal constraint in some country which forbids the configuration 'flashing yellow on one side and red on the opposite side'. If nobody in the team is aware of this fact, this constraint is "dark" information.

Note that the area sizes in Fig. 1 don't indicate any proportions. We are not aware of any research investigating the percentages of information in the categories identified in Fig. 1.

4.2 The value of relying on shared understanding

When developing or evolving software, relying only on *implicit* shared understanding does not work because real-world software is too complex for being developed without any explicit documentation. Even extremely code-focused

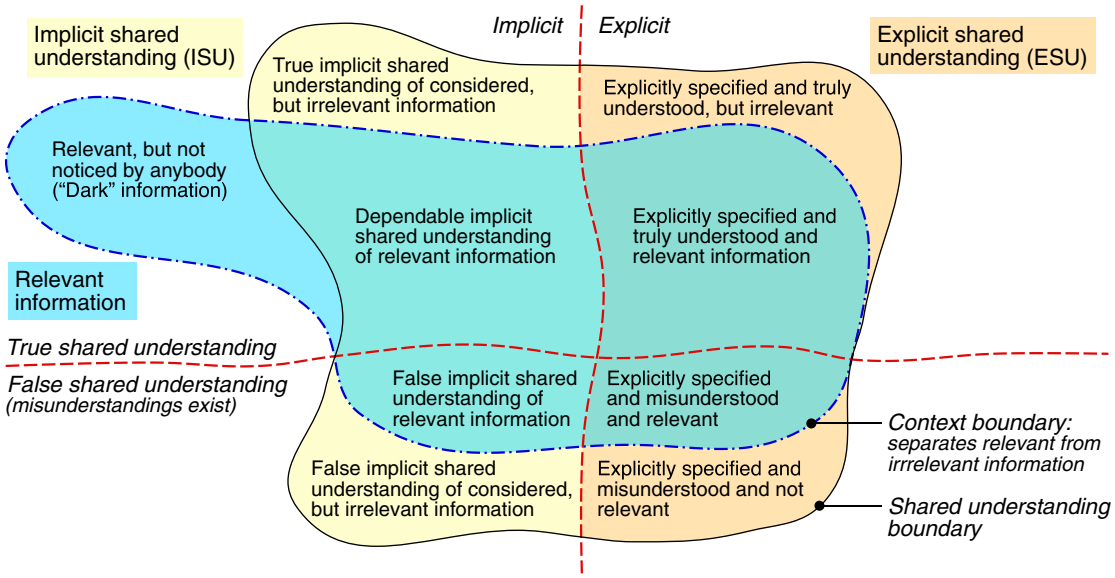

Fig. 1 Forms and categories of shared understanding. Note that area sizes do not indicate any proportions

agile methods such as XP [5] use a minimal set of explicit specifications. Conversely, relying solely on *explicit* shared understanding is both impossible and economically unreasonable for any real world software system. It is *impossible* because even within the context boundary of a system, the amount of relevant information is potentially infinite. Even if we assume that a system can be specified completely within finite time and space bounds, such a complete specification wouldn't be *economically reasonable* in most cases: the cost of creating and reading a complete specification would exceed its benefit, i.e., making sure that the deployed system meets the expectations and needs of its stakeholders.

Relying on implicit shared understanding has a strong economic impact on software development: The higher the extent of implicit shared understanding, the less resources have to be spent for explicit specifications of requirements and design, thus saving both cost and development time. However, these benefits come with a serious threat: assumptions about the existence or the degree of implicit shared understanding might be false. In this situation, omitting specifications yields systems that don't satisfy their stakeholders' needs, thus resulting in development failures or major rework for fault fixing. There is another important caveat: even if, in a given project, we manage to rely on implicit shared understanding to a major extent, reflection and explicit documentation of key concepts such as system goals, critical requirements and key architectural decisions remain necessary. Otherwise, development team fluctuation and evolution of deployed systems by people other than the original developers can easily become a nightmare because too much information is hidden. It might even be useful to document which requirements and design decisions have not been documented in detail due to reliance on implicit shared understanding so that this information is not lost when the developed system evolves.

4.3 Framing shared understanding in software engineering

As a consequence of the value considerations discussed in the previous subsection, we can frame the problem of how to deal with shared understanding for ensuring successful software development as follows:

(P1) Achieving shared understanding by explicit specifications as far as needed,
(P2) Relying on implicit shared understanding of relevant information as far as possible,
(P3) Determining the optimal amount of explicit specifications, i.e., striking a proper balance between the cost and benefit of explicit specifications.

Note that P1, P2, and P3 are not orthogonal problems, but different views of the same underlying problem: *How can we achieve specifications that create optimal value? Value* in this context means the *benefit* of an explicit specification (in terms of bringing down the probability for developing a system that doesn't satisfy its stakeholders' expectations and needs to a level that one is willing to accept), and the *cost* of writing, reading and maintaining this specification [27].

P2 can be sub-divided into three sub-problems:

(P2a) Increasing the extent of implicit shared understanding,
(P2b) Reducing the probability for false assumptions about implicit shared understanding,
(P2c) Reducing the impact of (partially or fully) false assumptions about implicit shared understanding.

Again, these sub-problems are not orthogonal: for increasing the extent of implicit shared understanding (P2a), we need to control the risk of false shared understanding, which can be framed in terms of probability (P2b) and impact (P2c).

For addressing these problems, it is important to know about the enablers and obstacles for shared understanding (Sect. 5) and appropriate practices for dealing with shared understanding (Sect. 6).

5 Enablers and obstacles

This section provides a list of enablers and obstacles for shared understanding. Knowing about enablers and obstacles helps to analyze a given project context with respect to the ease or difficulty of relying on shared understanding. In a constructive sense, careful use of these enablers helps setting up a software development project such that relying on shared understanding becomes easier and less risky than pragmatic ad-hoc work.

Domain knowledge Knowledge about the domain of the system to be built enables software engineers to understand the stakeholders' needs better, thus fostering shared understanding [35]. Domain knowledge reduces the probability that software engineers misinterpret specifications or fill gaps in the specification in an unintended way.

With respect to implicit shared understanding, domain knowledge can also be a threat: for example, implicit domain assumptions may be taken for granted by some team members, although not everybody involved is aware of them. In this situation, a smart person without domain knowledge (a "smart ignoramus" as Berry calls it [7]) can be valuable. Having a "smart ignoramus" in the team actually is an enabler for shared understanding. By asking all those questions that domain experts don't ask because the answer seems to be obvious to them, misunderstandings about domain concepts are uncovered, thus improving shared understanding.

Previous joint work or collaboration If a team of software engineers and stakeholders has collaborated successfully in previous projects, the team shares a lot of implicit understanding of the individual team members' values, habits, and preferences. In this situation, a rather coarse and high-level specification may suffice as a basis for successfully developing a system.

Existence of reference systems When a system to be developed is similar to an existing system that the involved stakeholders and engineers are familiar with, this existing system can be used as a *reference system* for the system to be built. Such a reference system constitutes a large body of implicit shared understanding.

Culture and values When the members of a team are rooted in the same (or in a similar) culture and share basic values, habits, and beliefs, building shared understanding about a problem is much easier than it is for people coming from different cultures with different value systems [38,44]. With increasing cultural distance between team members, the probability for missing or false implicit shared understanding is rising. The risk for misunderstanding explicit specifications is also higher than normal.

Common language A common language spoken by all members of a team (as their mother tongue or fluently mastered foreign language) is an enabler for shared understanding. Conversely, a language setting that requires translations between different languages spoken by team members constitutes a major obstacle.

Geographic distance Geographically co-located teams communicate and collaborate differently from teams where members live in different places and time zones [16,44]. Geographic co-location reduces the cultural distance mentioned above, thus enabling and fostering shared understanding.

Trust Mutual trust is a prerequisite for relying on implicit shared understanding. When involved parties, in particular customer and supplier, don't trust each other, explicit and detailed specifications for the system and the project must be created in writing, because everything that is not specified explicitly may not happen in the project, regardless of actual importance and needs. Vice-versa, building mutual trust requires some shared understanding [44]. Note, however, that even in situations of full mutual trust, the actual degree of implicit shared understanding must be assessed critically and carefully. Otherwise, if mutual trust leads to blind confidence in a high degree of shared understanding, such trust can actually become a threat.

Contractual situation When the relationship between customer and supplier is governed by a fixed-price contract with explicitly specified deliverables, shared understanding must be established on the basis of explicit specifications; there is not much room left for implicit shared understanding. However, even when a project is fully governed by an explicit contract, some basic implicit understanding, particularly about meanings of terms as well as cultural, political and legal issues, must exist among the involved parties. For example, if a contractual requirements specification states requirements about an order entry form of a system to be used in a European or North American context, the specification will typically not state that, for entering alphanumeric data into a field, the system has to position the cursor at the left edge of the field and display the data being typed from left to right. Instead, this is treated as a shared assumption about form editing.

Outsourcing When significant parts of a system development are outsourced, there is a high probability of non-matching cultural backgrounds (in terms of values, habits, beliefs) among team members. Team members at remote places who are assigned to outsourced work packages may also lack domain knowledge. So outsourcing is a significant obstacle to shared understanding.

Regulatory constraints If a system requires approval by a regulator, the regulator will typically require detailed, explicit specifications, thus leaving little room for alternative forms such as implicit shared understanding. So regulatory constraints are an obstacle to extensive use of implicit shared understanding.

Normal vs. radical design When the development of a system is governed by the principle of "normal design" [57], i.e., both the problem and the solution stay within an envelope of well-understood problems and solutions, the degree of implicit shared understanding is typically much higher than in "radical design", where the problem, the solution or both are new. Conversely, radical design entails a higher probability for false shared understanding.

Team size and diversity The larger and the more diverse a team, the more difficult it becomes to establish and rely on shared understanding. Hence, small teams are not only advantageous with respect to communication overhead, but also with respect to the ease of establishing and maintaining shared understanding.

Fluctuation of personnel Fluctuation of personnel is another common obstacle. This is especially problematic for implicit shared understanding, independent of whether stakeholders or software engineers change.

6 Practices for enabling, building, and assessing shared understanding

In this section, we compile a set of practices for dealing with shared understanding. We group them into three categories. *Enabling practices* lay foundations for shared understanding, but in themselves are not enough for achieving or analyzing shared understanding. *Building practices* are directed towards achieving shared understanding, (1) by creating explicit artifacts, or (2) by building a dependable body

of implicit shared understanding. *Assessment practices* aim at determining to which extent the understanding of some artifact or topic is actually shared among a group of people involved. Some practices can be used both for building and assessing shared understanding.

Tables 1, 2 and 3 classify and characterize practices for enabling, achieving, and assessing shared understanding. Almost all of the listed practices are actually well-known in software engineering. Potential usage of the practices in the context of implicit shared understanding will be discussed in Sect. 7.

7 Relying on implicit shared understanding

In this section we address the problem P2 posed in Sect. 4.3:

(P2) Relying on implicit shared understanding of relevant information as far as possible

As analyzed in Sect. 4, the two main factors that contribute to P2 are reducing (1) the probability and (2) the impact of false assumptions about implicit shared understanding. We discuss these factors in Sects. 7.1 and 7.3, respectively. In Sect. 7.3, we briefly investigate the influence of software development processes on implicit shared understanding.

7.1 Reducing the probability of false implicit shared understanding

7.1.1 Enabling practices

The enabling practices that address implicit shared understanding as outlined in Table 1 contribute to the creation of a stable and dependable basis for implicit shared understanding. *Domain scoping* and *domain understanding* narrow the amount of domain knowledge to be shared and lay the foundation for successfully communicating domain concepts. *Stakeholder selection* identifies the stakeholder roles that matter for a system to be built and helps identify proper representatives for these roles. *Team building* aims at selecting and forming teams such that members have shared experience, cultural background, and values. *Collaborative learning* helps create a common background when it is not possible to select people who already have this common background. *Feedback* is a general enabler for building and checking shared understanding.

7.1.2 Building practices

The building practices shown in Table 2 help create and improve implicit shared understanding, thus constructively lowering the probability of undetected misunderstandings.

Modeling (of domains, problems or solutions) makes the modeled concepts explicit and thus converts implicit shared understanding into explicit shared understanding. However, due to feasibility and economical reasons, models are almost never complete and frequently not detailed and/or formal enough for making everything explicit. Such models help infer and properly interpret non-modeled or only coarsely modeled concepts and increase the probability of interpreting them correctly, thus contributing to the creation of proper implicit shared understanding.

Modeling is a particular way of *formalizing requirements or architecture*. For any other form of formalization, the same arguments as for modeling apply with respect to implicit shared understanding.

Glossaries and *ontologies* provide explicit definitions of terminology for the system to be built and its domain. As this constitutes again a conversion of implicit shared understanding into explicit shared understanding, the same arguments as given for models apply: explicitly shared terminology reduces the probability of misunderstandings when concepts using this terminology are not specified or only coarsely specified.

Prototypes implement a selected subset of a system to be built. While a prototype secures explicit shared understanding of all features implemented in the prototype, it also improves implicit shared understanding of non-implemented features if the system to be built is implemented in the spirit and general directions given by the prototype.

Reference systems can serve as an anchor point for implicit shared understanding. If all persons involved are familiar with the reference system, implicit shared understanding of concepts about the system to be built can be achieved by referring to comparable or similar concepts in the reference system. Note that working with reference systems can also be used as an assessment practice (see below).

Workshops, although primarily aiming at the creation of explicit shared understanding by creating or validating artifacts, also foster implicit shared understanding and reduce the probability of misunderstandings as a by-product. Firstly, this is due to the same effect as described for modeling above. Secondly, well-moderated workshops implicitly contribute to the creation of a shared notion of goals, basic concepts, and values for the system to be built.

With *handshaking* [22], implementation proposals make the software engineers' interpretation of requirements explicit. When used early in the development process, the feedback provided by the implementation proposals allows building shared understanding among the stakeholders and the development team about the stakeholders' intentions.

7.1.3 Assessment practices

The assessment practices shown in Table 3, which aim at assessing implicit shared understanding, contribute to the

Table 1 Enabling practices

Practice	Description	ESU[1]	ISU[1]	Based on
Domain scoping [47]	Identify and narrow the domain where shared understanding has to be achieved	x	x	Discussion, documents, models
Domain understanding	Achieve general understanding of important domain concepts	x	x	Discussion, documents, models
Stakeholder and project team member selection [30,33]	Identify the stakeholders and project team members who will need to achieve shared understanding	x	x	Stakeholder analysis, searching, team building
Team building [20]	Build teams with shared experience and cultural background		x	Project management processes
Collaborative learning [19]	Increase the degree of ISU by a shared discourse of learned items		x	Moderated or free discourse about learned subjects
Feedback [15][2]	Ensure shared understanding between sender(s) and recipients(s) of information	x	x	Communication, artifacts
Negotiation and prioritization [6,32]	Achieve explicit consensus on some concept or issue	x		Artifacts and processes

[1] The form of shared understanding that the practice is useful for. ESU and ISU denote explicit shared understanding and implicit shared understanding, respectively
[2] Feedback addresses both ESU and ISU: feedback can be given on explicit artifacts (ESU) as well as on shared concepts (ISU). It plays a key role in many of the building and assessment practices given in Tables 2 and 3

Table 2 Building practices

Practice	Description	ESU	ISU	Based on
Domain modeling [24][1]	Achieve explicit shared understanding of important domain concepts by creating a model of the domain	x	x	Models, documents
Problem and solution modeling [40][1]	Achieve explicit shared understanding of system requirements or architecture by creating a model of the system	x	x	Models, documents
Building and using a glossary [28][1]	Achieve explicit shared understanding of the relevant terminology when developing a system	x	x	Glossary document
Using ontologies [34]	Achieve a general understanding of the major terms and concepts in a given domain		x	Documents containing the used ontologies
Formalizing requirements or architecture[1]	Create a formal specification for achieving explicit shared understanding of requirements or architectural design	x	x	Requirements specifications, system architecture
Quantifying requirements [27]	Achieve explicit shared understanding of a quality requirement by quantifying it	x		Quality requirements
Joint prototyping [59][2]	Build shared understanding of requirements or designs by experiencing how the final system will look and work	x	x	Prototype
Reference systems	Achieve shared understanding of a system by referring to an existing system that the involved persons are familiar with		x	Existing reference system
Holding workshops [31]	Achieve consensus about an artifact (vision, requirements specification, architecture) among the persons involved	x	x	Mainly discussion, also models, documents and examples
Handshaking [22][3]	Achieve shared understanding by aligning requirements and design for given features	x	x	Implementation proposals

[1] Building explicit shared understanding by modeling, glossaries, requirements formalization, etc. also improves implicit shared understanding of non-specified or coarsely specified concepts
[2] An approved prototype is an artifact that explicitly represents shared understanding of how a system to be shall look. On the other hand, a prototype also tests and fosters implicit shared understanding. Note that prototyping can also be used as an assessment practice, see Table 3
[3] Handshaking fosters implicit shared understanding by allowing stakeholders and engineers to agree on how requirements are operationalized into architecture and design. The resulting implementation proposals capture critical parts of that shared understanding explicitly

Table 3 Assessment practices

Practice	Description	ESU	ISU	Based on
Creating and playing scenarios [53][1]	Assess and foster shared understanding about how a system will work in typical situations	x	x	Scenarios, i.e., examples of system usage
Creating and (mentally) executing test cases[1]	Assess and foster shared understanding of results that a system will produce in typical situations	x	x	Test cases, i.e., examples of system usage
Model checking [12]	Formally determine whether some understanding of a specification actually holds	x		Formal requirements
Checking the glossary	Compare people's understanding of terminology with definitions in glossary	x	x	Glossary document
Prototyping or simulating systems [49,53][1]	Assess and foster shared understanding by comparing expectations about how a system will work in typical situations with actual system behavior	x	x	Formal or semi-formal requirements
Short feedback cycles [1][2]	Minimize the timespan between making/ causing and detecting misunderstandings or errors	x	x	Processes that enable and encourage rapid feedback
Paraphrasing[3]	Assess whether the understanding of a person paraphrasing an artifact matches the understanding of the author of the artifact	x	x	Human-readable artifacts
Having a smart ignoramus in the team [7]	Uncover misunderstandings by asking all those questions that domain experts don't ask because the answers seem to be obvious to them	x	x	Asking questions
Comparing to reference systems	Assess shared understanding of a system by comparing it to an existing system that the involved persons are familiar with		x	Existing reference system
Measuring shared understanding [21]	Measure the degree of shared understanding with subjective or objective indicators such as requirements coverage of agreed explicit design	x	x	An artifact such as a requirements specification or implicit concepts
Measuring ambiguity [23], Chapter 19	Assess the ambiguity of requirements with polling, thus indirectly assessing shared understanding	x	x	Requirements, polling questions

[1] Can also be used as a practice for building shared understanding. Addresses ESU when validating an explicit specification. Addresses ISU when there is no or only a coarse specification
[2] Short feedback cycles aim at detecting misunderstandings rapidly, as well as keeping the impact of false assumptions about ISU low
[3] Paraphrasing is originally an inspection technique [26], which also can be used for assessing shared understanding. It mainly addresses ESU, but is also useful for assessing ISU when communicating concepts orally

detection of misunderstandings, thus analytically lowering the probability of false implicit shared understanding.

Creating and playing scenarios make stakeholder intentions tangible and comprehensible by working with concrete examples. If implicit shared understanding of some concept or component can be exemplified by a representative set of scenarios, misunderstandings will be detected. Thus, the probability of false implicit shared understanding can be lowered systematically and significantly.

Creating and (mentally) executing test cases on requirements specifications or system architectures also exemplify how a system should behave in a given situation. Again, if implicit shared understanding of some concept or component can be exemplified by a representative set of test cases, misunderstandings will be detected, thus lowering the probability of false implicit shared understanding.

When a *glossary* exists, implicit shared understanding can be assessed by letting the involved people check their understanding of the terminology against the definitions in the glossary.

Prototyping or simulating systems has similar effects as playing scenarios: both make intentions tangible and comprehensible by example. Thus, the arguments given above for scenarios apply.

Short feedback cycles enable rapid detection of problems, including false implicit shared understanding. When misunderstandings are detected and corrected rapidly, the probability of undetected misunderstandings is reduced.

While *paraphrasing* is primarily a practice for assessing shared understanding of documents (i.e., explicit shared understanding), it can also be harnessed for assessing implicit shared understanding. For example, a stakeholder tells a requirements engineer that s/he needs feature X. In order to detect potential misunderstandings about what X actually is, the requirements engineer paraphrases the feature X and the impacts of that feature in her or his own words, and then the stakeholder checks the paraphrased story against her or his original intentions.

When domain experts take it for granted that everybody shares the knowledge about the basic concepts of a domain,

false implicit shared understanding can easily occur. Having a "*smart ignoramus*" in the team [7] who asks questions about all those basic concepts helps uncover such misunderstandings.

Comparing to a reference system also is a form of assessment by example. If all persons involved are familiar with the reference system, implicit shared understanding of concepts about the system to be built can be checked by comparing these concepts to corresponding concepts in the reference system. Thus misunderstandings of such concepts will become obvious and can be corrected.

Measuring shared understanding is a practice which is particularly little developed and understood today. In [21] we have described two approaches towards measuring requirements understanding: (a) *Ability to execute*, where architects estimate their confidence for developing an accepted product, (b) *R-Cov*, the coverage of requirements with explicit design.

If requirements are ambiguous, there is a high probability for misunderstanding them. Thus, *measuring ambiguity* with polling [23] indirectly assesses shared understanding. If the ambiguity of an implicit requirement or a vaguely stated requirement is measured, implicit shared understanding is assessed with respect to this requirement.

Not all of the assessment practices fit all types of requirements. For example, the scenario and test practices are challenged with respect to assessing implicit shared understanding of *non-functional* concepts such as quality requirements or constraints. These concepts are difficult to express in scenarios or to capture in test cases. In contrast, comparison to reference systems works well also for non-functional concepts. Prototyping and simulation take a middle ground with respect to how well they support the assessment of non-functional concepts.

7.2 Reducing the impact of false implicit shared understanding

We define the impact of false implicit shared understanding as the cost for detecting and correcting the underlying misunderstandings plus the cost incurred by (1) stakeholder dissatisfaction and (2) re-doing the work which has become invalid due to the misunderstandings.

Many of the assessment practices outlined in Table 3 can be used to reduce this impact. The practice of *short feedback cycles* particularly influences impact by reducing the timespan between causing and detecting misunderstandings: the less time a misunderstanding has to unfold, the lower its impact. Applying the other practices for assessing implicit shared understanding, such as using scenarios and test cases or comparing to reference systems, as *early* as possible also contributes to impact reduction. Early detection and correction of problems costs considerably less than when the same problems are detected and handled late in the development cycle.

Software development practices that increase flexibility for implementing change also contribute to lowering the impact of false shared understanding. Such practices, for example, refactoring or design for change, lower the cost of rework when errors are detected, thus reducing the cost of errors caused by false shared understanding.

The most effective and radical way of reducing the impact of false implicit shared understanding is *not* to rely on implicit shared understanding and use explicit specifications instead. However, this comes at the expense of producing and validating such specifications, so the savings from impact reduction must be weighed against this cost. Eventually, this boils down to an assessment of *risk*. Risk in this context means the risk that the system to be built does not satisfy its stakeholders' expectations and needs when it is eventually deployed. In [27] we discuss techniques for risk assessment of requirements and factors influencing the risk. By confining the reliance on implicit shared understanding to concepts with low or medium risk, both the average and the worst case impact of false implicit shared understanding are also confined.

7.3 Processes supporting implicit shared understanding

*Traditiona*l, waterfall-style software development processes strongly rely on explicit specifications, thus confining implicit shared understanding mostly to basic understanding of domain concepts and the interpretation of accidentally underspecified items.

Agile software development processes [5,48], on the other hand, strongly rely on implicit shared understanding. Stories and system metaphors provide general directions and a minimal amount of explicit shared understanding. Shared understanding of the unspecified details is secured by writing up-front test cases and by working with short feedback cycles. Other practices, for example comparison to reference systems, are not systematically used in agile development.

Any form of *incremental* or *prototype-oriented* development process [39] has potential for relying on implicit shared understanding to a significant extent. However, contemporary process descriptions do not reflect on how shared understanding is achieved, which practices are to be used, and why such practices are effective.

8 Some words about explicit shared understanding

We keep the discussion of explicit shared understanding rather short in this paper. The creation and interpretation of explicit specifications, which comes with explicit shared understanding, is rather well understood today. Massive amounts of research have been invested, and the research

results take a significant share of the scientific publications in requirements engineering.

Explicit specifications and explicit shared understanding are strongly related to each other. It is important to note, however, that they are not the same. The mere existence of explicit specifications, as well as of any other artifact, does not imply flawless explicit shared understanding. This is the reason why all specifications and other artifacts need to be *validated*. Validation is typically performed using the practices listed in Table 3. It aims at establishing explicit shared understanding between and among stakeholders on the one side and the software engineers on the other side. Only when an explicit specification has been validated thoroughly, we can say that this specification constitutes explicit shared understanding.

9 A roadmap for shared understanding

9.1 Where are we today with respect to shared understanding?

Today, as stated in Sect. 1, we make use of shared understanding in daily software engineering life without much reflection about it. Creating and validating explicit specifications where we rely on explicit shared understanding is rather well understood, particularly due to the progress made in requirements elicitation in the last 25 years. In contrast, we neither understand implicit shared understanding well nor do we handle it in a systematic and reflected way. Thus we are underusing the power of implicit shared understanding.

Also, today's development processes tend towards the extreme with respect to shared understanding: traditional sequential processes try to make all understanding explicit with extensive documentation, while agile processes aim at using as little documentation as possible, thus strongly relying on implicit shared understanding.

9.2 What can we do and where can we go with existing technology?

The notion of a risk-based, value-oriented approach to specifying quality requirements described in [27] can be extended to requirements in general. That means that for every individual requirement, we determine how to express and represent this requirement so that it yields optimal value, using an assessment of the risk as a guideline. Thus we deliberately decide where we write explicit specifications and where we rely on implicit shared understanding. A similar approach could be chosen for determining which architectural decisions should be documented explicitly and for which ones implicit shared understanding suffices.

As a general rule, we should rely on implicit shared understanding whenever we can afford it with respect to the risk involved. This requires processes that allow frequent, rapid feedback as we have it in today's agile processes. On the other hand, agile-addicts, who advocate producing code and tests as the only explicit artifacts, should note that in most real-world projects, we have high-risk requirements and architectural decisions that need to be documented explicitly in order to keep the risk under control.

As another general rule, systematic assessment of implicit shared understanding needs to be established as a standard practice in the same way as validating explicitly specified requirements is a standard practice today. With the exception of measuring implicit shared understanding, the required assessment practices exist (cf. Table 3).

9.3 Where do we need more research and insight?

The work presented in this paper is based on an analysis of our own experience accumulated over many years, as well as on experience reported in the literature. However, most of this experience is punctual and, with respect to strict scientific criteria, anecdotal. More research and investigation is needed to come up with analyses and rules that are based on dependable empirical evidence.

Measuring implicit shared understanding is an under-researched topic today. What we have today (cf. the last two rows of Table 3) is rather punctual or preliminary. Any progress in this field would be highly welcome and relevant for industrial practice.

The techniques we are currently using for assessing the risks of requirements are mainly qualitative and approximative. Any progress towards measuring or better estimating such risks would also be highly significant.

Finally, we are short of specific practices that are optimized for specific project settings. An example of such a practice is handshaking [22] which is designed for use in software product management where there is a single product or feature owner and a defined team of software engineers. Having such specific practices for other frequently occurring settings would constitute a significant progress.

10 Conclusions

Summary Shared understanding is important for efficient communication and for minimizing the risk of stakeholder dissatisfaction and rework in software projects. Achieving shared understanding between stakeholders and development team is not easy. Obstacles need to be overcome and enablers be taken advantage of. We have presented essential practices that enable and build shared understanding and practices that allow to assess it. We also have shed light on the handling of implicit shared understanding, which today is less researched and understood than dealing with explicit shared understand-

ing in the form of explicit specifications. A roadmap has been developed that describes how the current state of knowledge and practice can be improved.

Contribution Our essay represents a first focused overview of the topic of shared understanding in the development and evolution of software. It combines a synthesis of insight and experience with concrete advice on how to build and manage shared understanding. The results provide guidance for practitioners and represent a basis for future research.

Future work We hope that this essay motivates other researchers to work on the problem of shared understanding using the roadmap laid out in this article. In our own future work, we are planning empirical work on the importance and actual usage of shared understanding in practice. Generally, we will continue our quest for requirements specification techniques that provide optimal value in given contexts and situations.

Acknowledgments We thank the members of the Requirements Engineering Research Group at the University of Zurich for valuable comments on earlier versions of this paper.

References

1. Ambler S (2012) Why agile software development techniques work: improved feedback. http://www.ambysoft.com/essays/whyAgileWorksFeedback.html. Last visited 18 May 2014
2. Alexander I (1998) Engineering as a co-operative inquiry: a framework. Requir Eng 3(2):130–137
3. Aranda G, Vizcaíno A, Piattini M (2010) A framework to improve communication during the requirements elicitation process in GSD projects. Requir Eng 15(4):397–417
4. Arikoglu ES (2011) Impact des approches "scénario" et "persona" sur l'élicitation des exigences : une étude expérimentale [Impact of scenarios and personas on requirements elicitation: an experimental study (in English with title and extended abstract in French)]. PhD thesis, Université de Grenoble, France
5. Beck K (2004) Extreme programming explained: embrace change, 2nd edn. Addison-Wesley, Boston
6. Berander P, Andrews A (2005) Requirements prioritization. In: Aurum A, Wohlin C (eds) Engineering and managing software requirements. Springer, Berlin, pp 69–94
7. Berry DM (2002) Formal methods: the very idea. Some thoughts about why they work when they work. Sci Comput Progr 42(1):11–27
8. Bittner EAC, Leimeister JM (2013) Why shared understanding matters - engineering a collaboration process for shared understanding to improve collaboration effectiveness in heterogeneous teams. In: Proceedings Hawaii international conference on system sciences, pp 106–114
9. 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(11):1055–1068
10. Cannon-Bowers JA, Salas E (2001) Reflections on shared cognition. J Organ Behav 22(2):195–202
11. Clark HH (1996) Using language. Cambridge University Press, Cambridge
12. Clarke EM, Grumberg O, Peled D (1999) Model checking. MIT Press, Cambridge
13. Corvera Charaf M, Rosenkranz C, Holten R (2013) The emergence of shared language—applying functional pragmatics to study the requirements development process. Info Systems J 23(2):115–135
14. Couglan J, Macredie RD (2002) Effective communication in requirements elicitation: a comparison of methodologies. Requir Eng 7(2):47–60
15. Cramton C (2001) The mutual knowledge problem and its consequences for dispersed collaboration. Organ Sci 12(3):346–371
16. Damian D, Marczak S, Kwan I (2007) Collaboration patterns and the impact of distance on awareness in requirements-centered social networks. In: Proceedings 15th IEEE international requirements engineering conference (RE'07), pp 59–68
17. Darch P, Carusi A, Jirotka M (2009) Shared understanding of end-users' requirements in e-Science projects. In: Proceedings 5th IEEE international conference on e-Science workshops, pp 125–128
18. Darch P, Carusi A, Lloyd S, Jirotka M, de la Flor G, Schroeder R, Meyer E (2010) Shared understandings in e-science projects. Oxford e-Research Centre, University of Oxford, UK, Technical Report
19. Dillenbourg P (ed) (1999) Collaborative-learning: cognitive and computational approaches. Elsevier, Oxford
20. Edmondson A, Nembhard I (2009) Product development and learning in project teams: the challenges are the benefits. J Prod Innov Manag 26(2):123–138
21. Fricker S, Glinz M (2010), Comparison of requirements handoff, analysis, and negotiation: case study. In: Proceedings 18th IEEE international requirements engineering conference (RE'10), pp 167–176
22. Fricker S, Gorschek T, Byman C, Schmidle A (2010) Handshaking with implementation proposals: negotiating requirements understanding. IEEE Softw 27(2):72–80
23. Gause DW, Weinberg GM (1989) Exploring requirements: quality before design. Dorset House, New York
24. Gemino A, Wand Y (2005) Complexity and clarity in conceptual modeling: comparison of mandatory and optional properties. Data & Knowledge Engineering 55(3):301–326
25. 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: Maalej W, Thurimella AK (eds) Managing requirements knowledge. Springer, Berlin, pp 23–47
26. Gilb T, Graham D (1993) Software inspection. Pearson Education, Harlow
27. Glinz M (2008) A risk-based, value-oriented approach to quality requirements. IEEE Softw 25(2):34–41
28. Glinz M (2013) A glossary of requirements engineering terminology. Version 1.5. Available at http://www.ireb.org. Link "CPRE Glossary"; last visited 18 May 2014
29. Glinz M, Fricker S (2013) On shared understanding in software engineering. In: Proceedings GI conference software engineering 2013, Lecture notes in informatics, vol P-213, pp 19–35
30. Glinz M, Wieringa R (2007) Stakeholders in requirements engineering. IEEE Softw 24(2):18–20
31. Gottesdiener E (2002) Requirements by collaboration: workshops for defining needs. Addison-Wesley, Boston
32. Grünbacher P, Seyff N (2005) Requirements negotiation. In: Aurum A, Wohlin C (eds) Engineering and managing software requirements. Springer, Berlin, pp 143–162
33. Grundy T (1998) Strategy implementation and project management. Int J Project Manag 16(1):43–50
34. Guarino N, Oberle D, Staab S (2009) What is an ontology? In: Staab S, Studer R (eds) Handbook on ontologies, international handbooks on information systems, 2nd edn. Springer, Berlin, pp 1–17
35. Hadar I, Soffer P, Kenzi K (2014) The role of domain knowledge in requirements elicitation via interviews: an exploratory study. Requir Eng 19(2):143–159

36. Hill A, Song S, Dong A, Agogino A (2001) Identifying shared understanding in design using document analysis. In: Proceedings 13th international conference on design theory and methodology, ASME design engineering technical conferences, Pittsburgh, pp 1–7
37. Hoffmann A, Bittner EAC, Leimeister JM, (2013) The emergence of mutual and shared understanding in the system development process. In: Proceedings REFSQ 2013, LNCS, vol 7830. Springer, Berlin, pp 174–189
38. Hsieh Y (2006) Culture and shared understanding in distributed requirements engineering. In: Proceedings IEEE international conference on global software engineering (ICGSE 2006), pp 101–105
39. Larman C, Basili V (2003) Iterative and incremental development: a brief history. IEEE Comput 36(6):47–56
40. Lauesen S (2002) Software requirements: styles and techniques. Addison-Wesley Professional, London
41. Maalej W, Thurimella AK (eds) (2013) Managing requirements knowledge. Springer, Berlin
42. Macaulay L (1995) Cooperation in understanding user needs and requirements. Comput Integr Manuf Syst 8(2):155–165
43. McKay J (1998) Using cognitive mapping to achieve shared understanding in information requirements determination. Aust Comput J 30(4):139–145
44. Olson GM, Olson JS (2000) Distance matters. Hum-Comput Interact 15(2):139–178
45. Piirainen K, Kolfschoten G, Lukosch S (2009) Unraveling challenges in collaborative design: a literature study. In: Carriço L, Baloian N, Fonseca B (eds) Groupware: design, implementation, and use, proceedings 15th international workshop CRIWG 2009, LNCS, vol 5784. Springer, Berlin, pp 247–261
46. Puntambekar S (2006) Analyzing collaborative interactions: divergence, shared understanding and construction of knowledge. Comput Educ 47(3):332–351
47. Schmid K (2002) A comprehensive product line scoping approach and its validation. In: Proceedings 24th international conference on software engineering (ICSE 2002), pp 593–603
48. Schwaber K (2004) Agile project management with Scrum. Microsoft Press, Redmond, WA
49. Seybold C, Glinz M, Meier S (2005) Simulation-based validation and defect localization for evolving, semi-formal requirements models, In: Proceedings 12th Asia-Pacific software engineering conference (APSEC 2005), pp 408–417
50. Sommerville I, Sawyer P (1997) Requirements Engineering: a good practice guide. Wiley, Chichester
51. Stapel K (2012) Informationsflusstheorie der Softwareentwicklung [A theory of information flow in software development (in German)]. PhD Thesis, University of Hannover, Germany
52. Strauss A, Corbin J (1998) Basics of qualitative research: techniques and procedures for developing grounded theory. Sage Publications, Thousand Oaks, CA
53. Sutcliffe A (2010) Collaborative requirements engineering: bridging the gulfs between worlds. In: Nurcan S, Salinesi C, Souveyet C, Ralyté J (eds) Intentional perspectives on information systems engineering. Springer, Berlin, pp 355–376
54. Sutcliffe A, Sawyer P (2013), Requirements elicitation: towards the unknown unknowns. In: Proceedings 21st IEEE international requirements engineering conference (RE'13), pp 92–104
55. Tan M (1994) Establishing mutual understanding in systems design: an empirical study. J Manag Inf Syst 10(4):159–182
56. Van den Bossche P, Gijselaers W, Segers M, Woltjer G, Kirschner P (2011) Team learning: building shared mental models. Instr Sci 39(3):283–301
57. Vincenti WG (1993) What engineers know and how they know it: analytical studies from aeronautical history. John Hopkins University Press, paperback edition, Baltimore
58. Whitehead J (2007) Collaboration in software engineering: a roadmap. In FoSE 2007: Future of software engineering, workshop at ICSE 2007, Minneapolis, pp 214–225
59. Wood J, Silver D (1995) Joint application development. Wiley, New York
60. Zowghi D, Coulin C (2005) Requirements elicitation: a survey of techniques, approaches, and tools. In: Aurum A, Wohlin C (eds) Engineering and managing software requirements. Springer, Berlin, pp 19–46

Martin Glinz is a full professor and head of the Department of Informatics at the University of Zurich. His interests include requirements and software engineering—in particular modeling, validation, and quality—and software engineering education. He received his Dr. rer. nat. in Computer Science from RWTH Aachen University. Before joining the University of Zurich, he worked in industry for ten years.

Samuel A. Fricker is assistant professor at the Software Engineering Research Laboratory of Blekinge Institute of Technology (SERL Sweden). His interests include requirements engineering and software product management. He received his Dr. in Informatics from the University of Zurich. In addition to his work as a researcher, he has more than ten years experience in collaborating with companies at any scale, from startups to Fortune500.