On Shared Understanding in Software Engineering

Martin Glinz[1], Samuel Fricker[2]

[1]Department of Informatics, University of Zurich, Switzerland
glinz@ifi.uzh.ch
[2]Software Engineering Research Laboratory, Blekinge Institute of Technology (BTH), Sweden
samuel.fricker@bth.se

Abstract: Shared understanding is essential for efficient communication in software development and evolution projects when the risk of unsatisfactory outcome and rework of project results shall be low. Today, however, shared understanding is used mostly in an unreflected, intuitive way. This is particularly true for implicit shared understanding.

In this paper, 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 implicit shared understanding. We discuss enablers and obstacles, compile existing practices for dealing with shared understanding, and present a roadmap for improving knowledge and practice in this area.

1. Motivation

Shared understanding between stakeholders and software engineers is a crucial prerequisite for successful development and deployment of any software system. A *stakeholder* is a person or organization that has a (direct or indirect) influence on a system's requirements [GlW07]. *Software engineers* are the persons involved in the specification, design, construction, deployment, and maintenance/evolution of software systems. In traditional development projects, eliciting requirements and producing a comprehensive requirements specification serves for establishing shared understanding both among and between stakeholders (end users, customers, operators, managers, ...) and software engineers (requirements engineers, architects, developers, coders, testers,...). In agile environments, stories, up-front test cases, and rapid validation serve the same purpose.

Shared understanding among a group of people has two facets: *explicit shared understanding (ESU)* is about interpreting explicit specifications[1] (requirements, design documents, manuals,...) in the same way by all group members. On the other hand, *implicit shared understanding (ISU)* denotes the common understanding of non-specified facts, assumptions, and values. The shared context provided by implicit shared understanding

[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.

reduces the need for explicit communication [St12] and, at the same time, lowers the risk of misunderstandings.

In daily software engineering life, we make use of shared understanding without much reflection about it. *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. In contrast, the role and value of *implicit shared understanding* is frequently neither clear nor reflected.

This paper contributes an essay on shared understanding in Software Engineering. We reflect about the role and value of shared understanding, identify enablers and obstacles for achieving shared understanding, and compile a list of practices related to shared understanding. We then focus on implicit shared understanding, reflecting about its value and risk and describing when and how we can (or even should) rely on implicit shared understanding.

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 [Gl08], and the second author's work on communicating requirements by handshaking with implementation proposals instead of writing large explicit specifications [FGB10][FG10].

The remainder of this paper reflects the role and value of shared understanding (Section 2), presents enablers, obstacles and a compilation of practices (Sections 3, 4), and then discusses how to rely on implicit shared understanding (Section 5). We conclude with a short look at explicit shared understanding (Section 6), a roadmap (Section 7), a brief look at related work (Section 8) and some concluding remarks.

2. The role and value of shared understanding

Shared understanding in Software Engineering always implies dealing with both explicit and implicit shared understanding. Figure 1 illustrates the various facets of shared understanding. It is important to note that shared understanding can be false. This means that the involved people believe to have a shared understanding of some items, while there are differences in the actual understanding. We illustrate this briefly, 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. If this mis-

understanding goes undetected, we have *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*. 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.

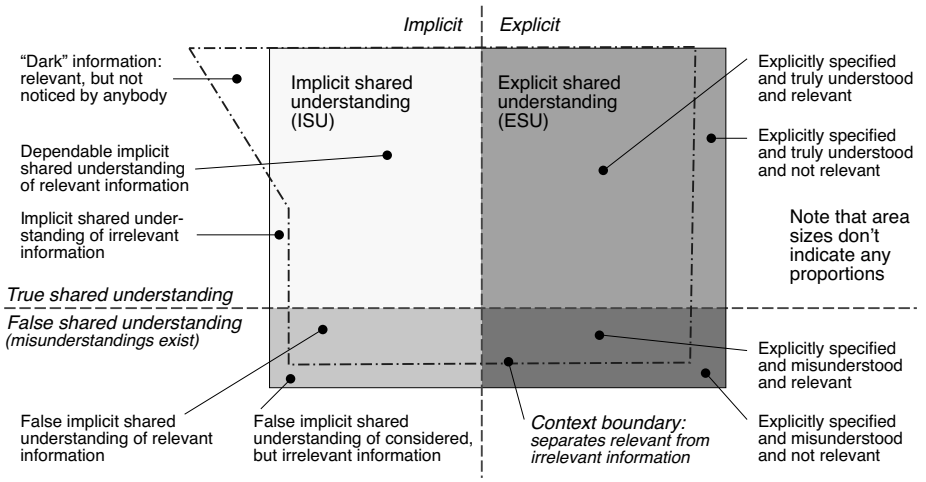

Figure 1: Forms and categories of shared understanding

When developing a system, there is a *context boundary* that separates the information which is *relevant* for the system to be built from the irrelevant rest of the world (cf. Fig. 1). However, sometimes stakeholders or software engineers also consider information (and might even achieve shared understanding about it) that is actually irrelevant. Conversely, we may have *"dark" information* that would be relevant, but has gone unnoticed by all team members. For example, imagine that in the traffic light problem presented above, there is a legal constraint in some country that 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. In this paper, we will not further elaborate the issue of relevant vs. irrelevant information and concentrate on the problem of shared understanding, regardless whether or not the underlying information actually is relevant.

Relying only on implicit shared understanding does not work because real-world software is too complex for being developed without any explicit documentation. 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 as well as evolving the deployed system by people other than the original developers can easily become a hidden knowledge nightmare. In some situations 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.

In summary, the problem of how to deal with shared understanding for ensuring successful software development can be framed 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.

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. 3) and appropriate practices for dealing with shared understanding (Sect. 4).

3. Enablers and obstacles

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

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. 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 [Be02]) 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. 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.

Geographic distance. Geographically co-located teams communicate and collaborate differently than teams where members live in different places and time zones. 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.

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, 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 no room for alternative forms such as implicit shared understanding. So regulatory constraints are an obstacle to implicit shared understanding.

Normal vs. radical design. When the development of a system is governed by the principle of "normal design" [Vi93], 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 shared understanding.

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

4. 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 (Tables 1-3): *Enabling practices* lay foundations for shared understanding or are generic procedures for achieving or analyzing shared understanding. *Building practices* are directed towards achieving shared understanding, (i) by creating explicit artifacts, or (ii) 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.

Almost all practices listed in Tables 1-3 are well-known practices in Software Engineering that don't need further explanation. The added value of Tables 1-3 lies in the classification and characterization of the listed practices with respect to shared understanding. Potential usage of the practices will be discussed in Sect. 5.

Table 1: Enabling Practices

Practice	**Goal**	**For**[1]	**Based on**
Domain scoping	Identify/narrow the domain where shared understanding has to be achieved	ESU ISU	Discussion, Documents, Models
Stakeholder/project team member selection	Identify the stakeholders and project team members who will need to achieve shared understanding	ESU ISU	Stakeholder analysis, searching, team building
Domain understanding	Achieve general understanding of important domain concepts	ESU ISU	Discussion, Documents, Models
Collaborative learning	Increase the degree of ISU by a shared discourse of learned items	ISU	Moderated or free discourse about learned subjects
Feedback[2]	Ensure shared understanding between sender(s) and recipients(s) of information	ESU ISU	Communication, Artifacts
Team building	Build teams with shared experience and cultural background	ISU	Project management processes
Negotiation and prioritization	Achieve explicit consensus on some concept or issue	ESU	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, depending on what form of understanding communication is based on. It plays a key role in many of the building and assessment practices given in Tables 2 and 3.

Table 2: Building Practices

Practice	Goal	For	Based on
Domain modeling[1]	Achieve explicit shared understanding of important domain concepts	ESU ISU	Models, Documents
Problem/ solution modeling[1]	Achieve explicit shared understanding of system requirements or architecture	ESU ISU	Models, Documents
Holding workshops	Achieve consensus about an artifact (vision, requirements spec, architecture) among the persons involved	ESU ISU	Mainly discussion, also models, documents and examples
Building and using a glossary[1]	Achieve explicit shared understanding of the relevant terminology when developing a system	ESU ISU	Glossary document
Using ontologies	Achieve a general understanding of the major terms and concepts in a given domain	ISU	Documents containing the used ontologies
Formalizing requirements or architecture[1]	Achieve explicit shared understanding of requirements and/or architectural design	ESU ISU	Requirements specifications, system architecture
Quantifying requirements	Achieve explicit shared understanding of a quality requirement by quantifying it	ESU	Quality requirements
Prototyping[2]	Build shared understanding of requirements or designs by experiencing how the final system will look and work	ESU ISU	Prototype
Reference systems	Achieve shared understanding of a system by referring to an existing system that the involved persons are familiar with	ISU	Existing reference system
Handshaking [FGB10][3]	Achieve shared understanding in a software product management context by feeding goals to the developers by the product manager and feeding back implementation proposals	ISU ESU	Goals and 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.

[3]Handshaking fosters implicit shared understanding. On the other hand, implementation proposals document the shared understanding explicitly.

Table 3: Assessment Practices

Practice	Goal	For	Based on
Creating and playing scenarios[1]	Assess and foster shared understanding about how a system will work in typical situations	ESU ISU	Scenarios, i.e., examples of system usage
Creating and (mentally) executing test cases[1]	Assess and foster shared understanding of the results that a system will produce in typical situations	ESU ISU	Test cases, i.e., examples of system usage
Model checking	Formally determine whether some understanding of a specification actually holds	ESU	Formal requirements
Checking the glossary	A good glossary lowers the probability of false shared understanding	ESU ISU	Glossary document
Prototyping or simulating systems[1]	Assess and foster shared understanding about how a system will work in typical situations	ESU ISU	Formal or semi-formal requirements
Short feedback cycles[2]	Minimize the timespan between making/ causing and detecting misunderstandings or errors	ESU ISU	Processes that enable and encourage rapid feedback
Paraphrasing[3]	Assess whether the understanding of the person(s) paraphrasing an artifact matches the understanding of the author(s) of the artifact	ESU ISU	Human-readable artifacts
Having a smart ignoramus in the team [Be02]	Uncover misunderstandings by asking all those questions that domain experts don't ask because the answers seems to be obvious to them	ESU ISU	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	ISU	Existing reference system
Measuring shared understanding	Measure the degree of shared understanding for a given artifact or set of implicit concepts	ESU ISU	An artifact such as a requirements specification or implicit concepts
Measuring ambiguity [GW89]	Assess the ambiguity of requirements with polling (see Chapter 19 in [GW89]), thus indirectly assessing shared understanding	ESU ISU	Requirements, polling questions

[1]Addresses ESU by checking whether a group of involved people is understanding an explicit specification in the same way. 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 low when relying on ISU.

[3]Mainly addresses ESU. Also useful for assessing ISU when communicating concepts orally.

5. Relying on implicit shared understanding

In this section we primarily address problem P2 posed in Section 2:

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

As mentioned in Section 2, P2 can be divided into the sub-problems of increasing the degree of implicit shared understanding and controlling the risk of implicit shared understanding, i.e., reducing both the probability for and the impact of false assumptions about shared understanding.

5.1 Reducing the probability of false implicit shared understanding

5.1.1 Enabling practices

The enabling practices which address implicit shared understanding (cf. 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.

5.1.2 Building practices

The building practices (cf. 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.

Workshops, although primarily aiming at the creation of explicit shared understanding by creating 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.

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 system can also be used as an assessment practice (see below).

With *handshaking* [FGB10], implementation proposals make the development team's 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.

5.1.3 Assessment practices

All assessment practices aiming at implicit shared understanding (cf. Table 3) contribute to the 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.

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 stakeholder tells a short story characterizing X, the requirements engineer paraphrases this story in her or his own words and then the stakeholder checks the paraphrased story against her or his original intentions.

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 fixed.

Measuring shared understanding is the only practice which is not well developed and understood today. In [FG10] 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 [GW89] 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.

The scenario and test practices are challenged with respect to assessing implicit shared understanding of *non-functional* concepts such as quality requirements or constraints, because 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 assessment of non-functional concepts.

5.2 Reducing the impact of false implicit shared understanding

The impact of false implicit shared understanding is defined as the cost for detecting and correcting the underlying misunderstandings plus the cost incurred by (i) stakeholder dissatisfaction and (ii) re-doing the work which has become invalid due to the misunderstandings.

The practice of short feedback cycles (cf. Table 3) strongly 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 (cf. Table 3) as *early* as possible also contributes to impact reduction, as early detecting and fixing problems costs considerably less than when the same problems are detected only late in the development cycle.

There are software development practices, for example, refactoring or design for change, that lower the cost of rework when errors are detected. These practices also lower the impact of false shared understanding.

The most effective way, however, of reducing the impact of false implicit shared understanding is to base the specification of high-risk concepts on *explicit* shared understanding rather than on implicit shared understanding. 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 [Gl08] 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.

5.3 Processes supporting implicit shared understanding

Traditional 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, on the other hand, strongly rely on implicit shared understanding, but without much reflection. Stories and system metaphors provide general directions. Shared understanding of the unspecified details is secured by writing up-front test cases and 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 has potential for relying on implicit shared understanding to a significant extent. However, contemporary process descriptions don't reflect on how shared understanding is achieved, which practices are used, and why they are used.

6. Some words about explicit shared understanding

We keep the discussion of explicit shared understanding rather short in this paper, because the creation and interpretation of explicit specifications, which comes with explicit shared understanding, is rather well understood today. The only crucial point that needs to be stated here is the relationship between explicit specifications and explicit shared understanding.

It is very important to know that the mere existence of explicit specifications (as well as of any other artifact) doesn't imply flawless explicit shared understanding. This is the

reason why all specifications and other artifacts need to be *validated*. Validation, typically using the practices listed in Table 3, aims at establishing explicit shared understanding between (and among) stakeholders on the one side and 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.

7. A roadmap for shared understanding

7.1 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 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.

7.2 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 [Gl08] 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).

7.3 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 [FGB10] 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.

8. Related work

There is a large body of existing work on particular problems of shared understanding. A comprehensive discussion of this work is beyond the scope of this paper. However, to the best of our knowledge, nobody so far has attempted to give an overview of the problem, summarize existing practices and shed some light on implicit shared understanding, which are the main topics of this paper.

Further there are large bodies of work in related fields such as requirements elicitation, knowledge management, ontologies, and the semantic web that we neither can survey nor summarize within the scope of this paper.

We just give a few selected pointers to related work here. McKay [MK98] proposes a technique called cognitive mapping for achieving shared understanding of requirements. Hill et al. [HSD01] try to identify shared understanding by analyzing the similarity of documents produced by team members, based on latent semantic analysis. Puntambekar [Pu06] investigates the role of collaborative interactions for building shared knowledge. Gacitua et al. [GMN09] review the role of tacit knowledge in Requirements Engineering. Zowghi and Coulin [ZC05] survey the field of requirements elicitation. This topic is also covered in almost any textbook on Requirements Engineering. Guarino et al. [GOS09] give an introduction to ontologies. Stapel [St12] contributes a theory of information flow in software development.

9. 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 assessing 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 understanding 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 for software projects. It combines a synthesis of insight and experience with concrete advice of how to build and manage shared understanding. The results provide guidance for practitioners and represent a basis for future research.

Future Work. In our own future work, we are planning to conduct a systematic survey of shared understanding in requirements engineering, including a comprehensive literature analysis. Generally, we will continue our quest for requirements specification techniques that provide optimal value in given contexts and situations.

Acknowledgements

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

[Be02] Berry, D.M.: Formal Methods: The Very Idea. Some Thoughts About Why They Work When They Work. Science of Computer Programming 42(1), 2002. 11-27.

[FG10] Fricker, S.; Glinz, M.: Comparison of Requirements Hand-Off, Analysis, and Negotiation: Case Study. In Proc. 18th IEEE International Requirements Engineering Conference (RE'10), Sydney, 2010. 167-176.

[FGB10] Fricker, S.; Gorschek, T.; Byman, C.; Schmidle, A.: Handshaking with Implementation Proposals: Negotiating Requirements Understanding. IEEE Software 27(2), 2010. 72-80.

[Gl08] Glinz, M.: A Risk-Based, Value-Oriented Approach to Quality Requirements. IEEE Software 25(2), 2008. 34-41.

[GlW07] Glinz, M.; Wieringa, R.: Stakeholders in Requirements Engineering. IEEE Software 24(2), 2007. 18-20.

[GMN09] Gacitua, R.; Ma, L.; Nuseibeh, B.; Piwek, P.; De Roeck, A.N.; Rouncefield, M.; Sawyer, P.; Willis, A.; Yang, H.: Making Tacit Requirements Explicit. In Proc. 2nd International Workshop on Managing Requirements Knowledge (MARK 2009), Atlanta, 2009. 85-88.

[GOS09] Guarino, N.; Oberle, D.; Staab, S.: What Is an Ontology? In (Staab, S.; Studer, R., eds.): Handbook on Ontologies, International Handbooks on Information Systems, 2nd edition, Berlin: Springer, 2009. 1-17.

[GW89] Gause, D.W.; Weinberg, G.M.: Exploring Requirements: Quality Before Design. New York: Dorset House, 1989.

[HSD01] Hill, A.; Song, S.; Dong, A.; Agogino, A.: Identifying Shared Understanding in Design Using Document Analysis. In Proc. 13th International Conference on Design Theory and Methodology, ASME Design Engineering Technical Conferences, Pittsburgh, 2001.

[MK98] McKay, J.: Using Cognitive Mapping to Achieve Shared Understanding in Information Requirements Determination. Australian Computer Journal 30(4), 1998. 139-145.

[Pu06] Puntambekar, S.: Analyzing Collaborative Interactions: Divergence, Shared Understanding and Construction of Knowledge. Computers and Education 47(3), 2006. 332-351.

[St12] Stapel, K.: Informationsflusstheorie der Softwareentwicklung [A Theory of Information Flow in Software Development (in German)]. PhD Thesis, University of Hannover, Germany, 2012.

[Vi93] Vincenti, W.G.: What Engineers Know and How They Know It: Analytical Studies from Aeronautical History. Baltimore: Johns Hopkins University Press, paperback edition, 1993.

[ZC05] Zowghi, D.; Coulin, C.: Requirements Elicitation: A Survey of Techniques, Approaches, and Tools. In (Aurum, A.; Wohlin, C., eds.): Engineering and Managing Software Requirements. Berlin: Springer, 2005. 19-46.