Quality Requirements in Agile as a Knowledge Management Problem: More than Just-in-Time

Eric Knauss, Grischa Liebel
Computer Science and Engineering
Chalmers | University of Gothenburg
Gothenburg, Sweden
{knauss,grischa}@chalmers.se

Kurt Schneider
Software Engineering Group
Leibniz Universität Hannover
Hannover, Germany
ks@inf.uni-hannover.de

Jennifer Horkoff, Rashidah Kasauli
Computer Science and Engineering
Chalmers | University of Gothenburg
Gothenburg, Sweden
{jenho,rashida}@chalmers.se

***Abstract*—Just-in-time (JIT) approaches have been suggested for managing non-functional requirements in agile projects. However, many non-functional requirements cannot be raised and met on the spot. In this position paper, we argue that effective JIT engineering of quality requirements depends on a solid foundation of long-term knowledge about all relevant quality requirements. We present two examples from projects related to safety and security and show that not all aspects of these quality requirements can be invented and changed just in time. Further, managing, for example, operationalization of quality requirements just in time depends on sufficient understanding of (i) customer value and (ii) the system under construction that must be shared by the engineering team. If a Learning Software Organization (LSO) intends to increase agility and speed up system development, it needs a holistic concept for managing this knowledge. We propose that a knowledge-management framework can facilitate JIT-RE by structuring, representing, and allowing updates of long-term knowledge about quality requirements. Such a knowledge-management framework should allow to map user value to system requirements and have important properties to allow JIT RE and sustainable evolution.**

***Keywords*-just-in-time RE, quality requirements, managing requirements knowledge**

I. Introduction

In this position paper, we argue based on our experience in several projects that while just-in-time management of quality requirements is important in agile development, it must be complemented by an initiative to manage long-term aspects of quality requirements and to build and use knowledge about the system under construction.

Agile software development lacks systematic approaches for managing quality requirements [10] and needs further research, even though a variety of promising practices for managing quality requirements are known [3]. Among those, some relate to just-in-time aspects of quality requirements engineering: relying on face-to-face communication and iterative emergence of requirements [3]. Others imply a long-term perspective on quality requirements, as for example product grooming, continuous integration, and test-driven development [3]. Yet, according to our experiences existing challenges of managing quality requirements in agile development show an inability to synchronize just-in-time RE activities and long-term perspectives on architecture and system verification. This is concerning: on the one hand, we need just-in-time analysis of quality requirements to operationalize them for functionality currently under development. On the other hand, quality requirements can be considered long-term business drivers that allow diversification from competitors [5]. An agile team not only needs to care about the current release or project, but also about the next [7].

Proposition 1 (JIT vs. Long-Term): *We propose that JIT management of quality requirements must be complemented by an initiative to manage long-term aspects of quality requirements.* In agile requirements engineering, a lot of emphasis is on understanding and communicating customer and end-user value [1]. This is important, since agile approaches rely on self-organized teams with some autonomy [13] and these teams need to understand what provides customer value, before they can make decisions [11]. Pre-agile approaches to requirements engineering emphasize the importance of distinguishing between user requirements and system requirements [17] and this importance has been confirmed for requirements engineering in large-scale agile system development [12]. Incremental agile development and continuous delivery do not only require an excellent understanding of customer value, but also effective knowledge about how and why the current system was built. Consequently, many challenges reported for agile management of quality requirements relate to a lack of consideration of this system perspective in agile development, e.g. focusing on delivering functionality at the cost of architecture flexibility as well as ignoring predictable architecture requirements [3].

Proposition 2 (Customer value vs. System knowledge): *We claim that both JIT and long-term management of quality requirements must consider both a user (or: market) value perspective and a system requirements perspective.* We argue that most qualities cannot be significantly improved just-in-time. While for example a single change can destroy security or safety of a system, the only way to create a system that has these properties is to grow it around a strong notion of the most important qualities. The knowledge of how this has been done must be conserved and must be made accessible for JIT quality requirements activities.

In this position paper, we revisit well established knowledge management literature with our two key propositions in mind.

II. Knowledge Management Foundations for Managing Quality Requirements

In this paper, we consider requirements engineering as a knowledge management problem. Knowledge management addresses the acquisition of knowledge, transforming it from tacit or implicit into explicit knowledge and back again, storing, disseminating, and evaluating it systematically, and applying it in new situations [16]. We see requirements as a special type of knowledge, which needs to be managed in an organization. Doing requirements engineering then relates to organizational learning, which is an approach that stimulates learning of individuals, organization-wide collection of knowledge, and cultivation of infrastructure for knowledge exchange [16].

Nonaka's and Takeuchi's theory of knowledge creation relates to tacit and explicit knowledge [14][1]. Knowledge is created by converting it from a knowledge source (either tacit or explicit) to a new knowledge store (also either tacit or explicit). This view relates very well to requirements management, especially for quality requirements. For example, conversion of tacit knowledge to tacit knowledge (*socialization* in Fig. 1), corresponds to an agile way of managing requirements, that de-emphasizes documentation and instead relies on face-to-face communication and just-in-time clarification of requirements. A long-term perspective on requirements would require explicit knowledge representations. *Combination*, for example, corresponds to tracing information derived from relating different artifacts of system engineering to each other.

Earl has developed a framework to classify studies on knowledge management according to different research directions, which he calls schools [8]. The *technocratic* school focuses on systems, maps and engineering of knowledge and resonates with a traditional approach to requirements engineering with a central requirements database (or specification) as knowledge base. In contrast, the *economic* school focuses on commercial value of knowledge and the *behavioral* school considers organizational, spatial, and strategic aspects. We note that these latter schools resonate with values of agile RE.

III. Example 1: Safety

Safety can be briefly characterized as the confidence that a software or technical system will not harm humans or cause major damage or financial loss. Airplanes or cars need to ensure functional safety according to ISE 61 508 or ISO 26262. An autonomous driving system must ensure that the intelligent breaks will not cause accidents.

In our project on Requirements Engineering for large-scale agile system development [12], many case companies develop safety critical systems and are subjected to regulation. These companies struggle to establish an effective approach to manage safety that still supports agile, incremental work. Regulations often require comprehensive tracing information that relates different system engineering artifacts to each other

[1]note that their notion of tacit knowledge differs from how this term is usually used in RE research: for them, tacit knowledge is not explicitly documented but can be shared in face-to-face communication.

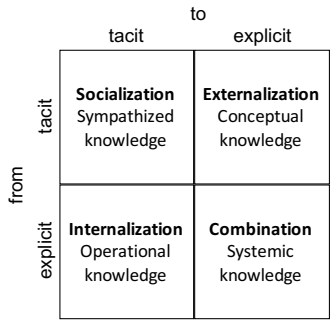

Fig. 1: Conversions between tacit and explicit knowledge [14].

and allows to show how safety was systematically build into the system. Further, in order to support incremental, agile development, it is desirable to also allow for incremental verification. For this and other reasons, safety must already be considered during creating the system architecture, e.g. by defining independent components, separating safety concerns from others, and provide redundancies for critical components. This requires long term system knowledge, mainly created through *combination* (Fig. 1) of existing system artifacts.

Yet, there is always a risk to include a change that effectively declines the safety of a component. This risk must be mitigated just-in-time, for example by doing a change-impact-analysis to understand which components will be affected and then *internalize* the existing knowledge about the system under construction. Such existing knowledge is usually provided by existing engineering artifacts (in agile: code and its tests). When a cross-functional team starts the development of a new feature for a safety critical system, one of the first steps is to do a hazard analysis. The result will inform the team about the safety criticality of the feature, which in turn defines the engineering method to be applied. In our experience, practitioners often reach out to domain experts to help them assess the system requirements efficiently, leading to emergent collaboration. This activity can be considered to be just-in-time and relies on *internalization* of existing system knowledge as well as on *socialization* to discuss how the feature will affect functional safety.

The foundation for such reasoning is long-term knowledge about the desired (or required) level of safety. While this knowledge might have been established at one time face-to-face through socialization of domain experts, it is long-lasting and reusable (i.e. the next product will relate to very similar safety concerns). Thus, an efficient way of *externalization* of this knowledge is required.

Today, this externalization is not emphasized in many agile system development approaches. Due to the long-lasting nature of safety considerations of systems, existing documentation can be reused after a company has transitioned from v-model or waterfall approaches to agile. However, we argue that updating and maintaining this knowledge must be considered in any approach to agile system development.

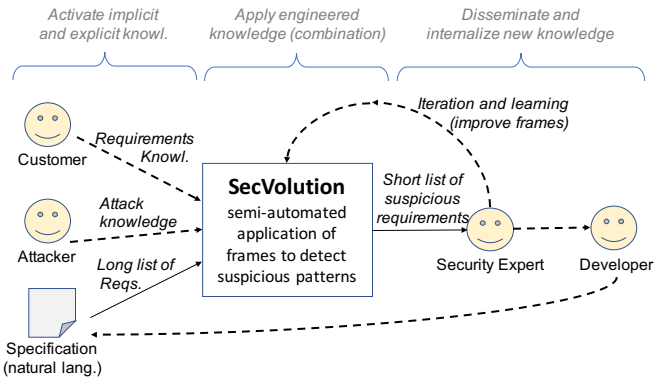

Fig. 2: Three essential activities in managing knowledge on software quality attributes.

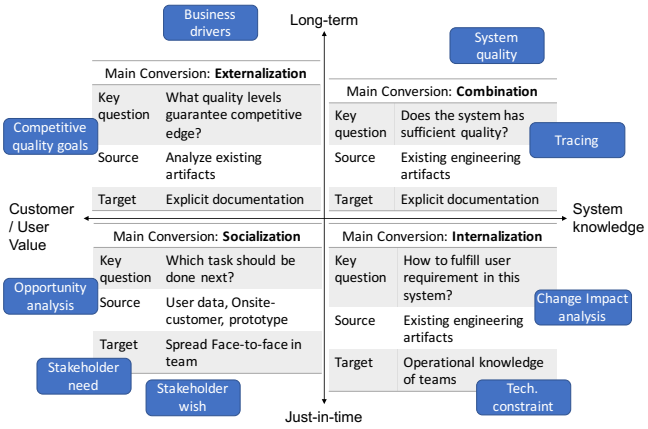

Fig. 3: Towards a Knowledge Management Framework for Agile Quality Requirements Management: Aligning knowledge conversion with our propositions.

IV. Example 2: Security

Security is defined as the ability of a system to withstand attacks. In contrast to safety, security does assume an adversary viciously attacking the software. Developers need to anticipate and consider all potential attacks; in misuse-cases [2], this antagonism between two sides is made explicit. This *externalization* is often supported by attack trees.

Knowledge about vulnerabilities, past attacks, and many other aspects of a software system is crucial for both sides. If developers want to stay ahead of attackers, they need to organize and use that knowledge. In the German DFG Priority Programme 1593 (Design for Future), we work on an approach for detecting vulnerabilities during requirements analysis. This "SecVolution" approach is fundamentally built on a knowledge-perspective of security [9, 6].

Security is a quality aspect of growing importance: Large home entertainment systems may have been initially out of scope for security, but when they collect payment information or personal data, they suddenly become very security-relevant. Supermarket management software may start out as a local and non-distributed application of moderate size and limited security relevance. When an online-store component is added, software security definitely turns into a major concern. As these examples indicate, security is far from a commodity that can be added and removed at convenience. Instead, a single known vulnerability can make the entire system insecure. SecVolution investigates changes that can cause security to suffer. The above-mentioned scenarios exemplify this type of changes. However, there is an additional type of changes that is not mentioned above but just as severe: Even if nothing in the software changes, its security can suffer when attackers discover a new security breach in the existing code, and exploit it for an attack. A long-living system does not wear out over time, but it ages in relationship to the knowledge developers and attackers have about it. A core insight in SecVolution was how crucial it is to *externalize* attacker and security knowledge from people; automate it in a tool, and help developers internalizing it when they see suspicious findings.

Highly qualified security experts are able to spot patterns of payment, data, access, and the like. However, those experts are rare and cannot check each and every document and use case. A presumably unproblematic system, such as a supermarket or smart TV, tends to be neglected in terms of security. Thus, SecVolution tries to detect as many known suspicious requirements and submit this much smaller list of requirements to the rare experts for final resolution. In order to extract knowledge from human-made natural language requirements, we use natural-language processing techniques. After parsing the sentence grammatically, we search for matching Security Frames (i.e., suspicious patterns) and use an ontology to represent the knowledge.

Techniques for soliciting and deriving explicit knowledge from people who have internalized it can be very challenging [9]. As a prerequisite, the collection of knowledge must be maintained as a long-term endeavor and the interplay of developers, attackers, and security experts must be investigated.

V. Towards a KM Framework

Safety and security are examples of two quality requirements that cannot be managed purely JIT. We believe that future research will show similar considerations to apply to all quality requirements, since they are typically related to architecture, and tend to build on knowledge as much as on software structures, although both need to come together.

In the above-mentioned SecVolution case, an ontology plays a central role of managing knowledge. Earlier experiences (of attacks), external published warnings (of vulnerabilities), and insights of security experts are encoded in the ontology, which can then be applied to natural-language requirements (Fig. 2).

There are obvious technical challenges involved in building such a knowledge infrastructure. It turned out to be at least equally challenging to solicit, interpret, and engineer the knowledge. Initially, most of that knowledge resides in people and needs to be externalized [14] before it can be encoded and stored in an ontology. Therefore, extracting implicit or even

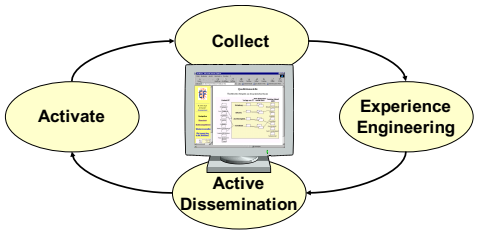

Fig. 4: Life-cycle of experiences, iterating around exp. base.

tacit knowledge from people is essential and not just a side-issue. Tapping human knowledge and experience is almost a discipline by itself [16]. Making a knowledge management infrastructure effective requires taking the social and socio-technical challenges seriously. Based on these observations, Schneider presents an experience life cycle [16] in software engineering (Fig. 4): Activation and collection are devoted to techniques and tools for attracting implicit or tacit knowledge into the system. A purely technocratic view [8] tends to neglect the left-hand input block. Along the same lines, many experiences (or knowledge items collected) are never actively disseminated. Thus, they remain useless. Experts need to be made aware of the valuable knowledge they have (*activate*); there must be support to collect that information once it surfaces (*collect*, e.g. as a by-product of other tasks that need to be conducted anyway [15]). Usually, the knowledge cannot be collected in exactly the same form that is most appropriate for reuse. Following Basili's [4] terminology, we call the activity of merging, comparing, and reformatting "*experience and knowledge engineering*". Finally, the resulting knowledge must be delivered to where it is needed, when it is needed, and in the most adequate form. In the SecVolution example, knowledge ends up in an ontology and is automatically applied to natural-language requirements. This is a very clear and technically sophisticated way of distribution. In other cases, knowledge will need to be presented to developers, requiring them to understand and apply it.

Figures 3 and 4 sketch the core of a knowledge management infrastructure for quality requirements: Fig 4 stresses the iterative nature of experience or knowledge about a quality aspect. Such an iterative process is applicable to JIT as well as to long-term requirements. Fig. 3 highlights different types of sources and the main knowledge operations conducted. JIT and long-term requirements are not a contradiction, but need to complement each other. We encourage future work to investigate implications for other qualities and on RE processes that support JIT RE based on long-term knowledge.

Acknowledgments. We thank Francisco Gomes for his feedback and support. This work was supported by Software Center (RE for Large-Scale Agile System Dev. Project) and German DFG Priority Programme 1593 (Design for Future).

References

[1] H. Alahyari, R. Berntsson Svensson, and T. Gorschek. A study of value in agile software development organizations. *Journal of Systems and Software*, 2016.

[2] Ian Alexander. Initial Industrial Experience of Misuse Cases. In *Int. Reqts. Eng. Conf.*, pages 9–13, 2002.

[3] Wasim Alsaqaf, Maya Daneva, and Roel Wieringa. Quality requirements in large-scale distributed agile projects – a systematic literature review. In *Proc. of 23rd Int. Working Conf. on Requirements Eng.: Foundation for Software Quality*, pages 219–234, Essen, Germany, 2017.

[4] V. Basili, G. Caldiera, and D.H. Rombach. *The Experience Factory*. John Wiley and Sons, 1994.

[5] Richard Berntsson Svensson and Björn Regnell. A case study evaluation of the guideline-supported quper model for elicitation of quality requirements. In *Proc. of Int. Working Conf. on Requirements Eng.: Foundation for Software Quality*, pages 230–246, 2015.

[6] Jens Bürger, Jan Jürjens, Thomas Ruhroth, Stefan Gärtner, and Kurt Schneider. Model-based Security Engineering with UML: Managed Co-Evolution of Security Knowledge and Software Models. In *Foundations of Security Analysis and Design VII: FOSAD Tutorial Lectures*, pages 34–53, 2014.

[7] Alistair Cockburn. *Agile Software Development: The Cooperative Game*. Addison-Wesley, 2nd edition, 2009.

[8] M. Earl. Knowledge management strategies: Towards a taxonomy. *Journal of Management and Information Systems*, 18(1):215–233, 2001.

[9] Stefan Gärtner, Thomas Ruhroth, Jens Bürger, Kurt Schneider, and Jan Jürjens. Maintaining Requirements for Long-Living Software Systems by Incorporating Security Knowledge. In *Proc. of 22nd Int. Reqts. Eng. Conf. (RE)*, pages 103–112, 2014.

[10] I. Inayat, S. S. Salim, S. Marczak, M. Daneva, and S. Shamshirband. A systematic literature review on agile requirements engineering practices and challenges. *Computers in human behavior*, 51:915–929, 2015.

[11] Rashidah Kasauli, Eric Knauss, Agneta Nilsson, and Sara Klug. Adding value every sprint: A case study on large-scale continuous requirements engineering. In *3rd WS on Cont. Reqts. Eng. (CRE)*, Essen, Germany, 2017.

[12] Rashidah Kasauli, Grischa Liebel, Eric Knauss, Swathi Gopakumar, and Benjamin Kanagwa. Requirements engineering challenges in large-scale agile system development. In *Proc. of 25th Int. Requirements Engineering Conf. (RE '17)*, Lisbon, Portugal, 2017.

[13] Bertrand Meyer. *Agile! The Good, the Hype and the Ugly*. Springer, 2014.

[14] I. Nonaka and H. Takeuchi. *The Knowledge Creating Company*. Oxford University Press, 17th edition, 1995.

[15] Kurt Schneider. *Rationale Management in Software Engineering*, chapter Rationale as a By-Product, pages 91–109. Springer, Berlin, Heidelberg, 2006.

[16] Kurt Schneider. *Experience and Knowledge Management in Software Engineering*. Springer, 2009.

[17] Ian Sommerville. *Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 8th edition, 2006. ISBN 0321313798.