Decision Knowledge Triggers in Continuous Software Engineering

Anja Kleebaum
Heidelberg University
Institute of Computer Science
Heidelberg, Germany
kleebaum@informatik.uni-heidelberg.de

Jan Ole Johanssen
Technical University of Munich
Department of Informatics
Munich, Germany
jan.johanssen@in.tum.de

Barbara Paech
Heidelberg University
Institute of Computer Science
Heidelberg, Germany
paech@informatik.uni-heidelberg.de

Rana Alkadhi
Technical University of Munich
Department of Informatics
Munich, Germany
alkadhi@in.tum.de

Bernd Bruegge
Technical University of Munich
Department of Informatics
Munich, Germany
bruegge@in.tum.de

ABSTRACT

Decision knowledge encompasses decisions and related information such as the problems the decisions address, their rationale, or alternatives. The management of decision knowledge is considered important for software development, however, it is often not integrated, since it requires additional effort and developers do not perceive short-term benefits. Continuous software engineering offers new possibilities to overcome these drawbacks: During continuous software engineering, developers perform practices suitable to integrate the management of decision knowledge in their daily work. For example, developers regularly commit code and manage tasks to implement features. In this paper, we present ideas on how to trigger the developers to capture and use decision knowledge during these practices, in particular to 1) package distributed decision knowledge, 2) make tacit decisions explicit, and 3) consider consistency between decisions.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; *Agile software development*; *Documentation*; *Maintaining software*;

KEYWORDS

Decision Knowledge, Knowledge Management, Rationale, Continuous Software Engineering, Software Evolution, Trigger

ACM Reference Format:
Anja Kleebaum, Jan Ole Johanssen, Barbara Paech, Rana Alkadhi, and Bernd Bruegge. 2018. Decision Knowledge Triggers in Continuous Software Engineering. In *RCoSE'18: IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering, May 29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3194760.3194765

RCoSE'18, May 29, 2018, Gothenburg, Sweden

1 INTRODUCTION

Continuous software engineering (CSE) is a process in which parallel workflows are activated and interrupted through change events [17]. For example, a feedback report activates a feedback workflow and might supply developers with changed requirements or new feature requests. When developing features, developers need to make decisions and thereby solve decision problems. They either solve such decision problems by using alternatives and arguments that are weighed (rational decision making) or by using past experiences (naturalistic decision making) [11]. The knowledge that developers possess about decisions and their justifications is called *decision knowledge*. If decision knowledge is explicitly captured, it is valuable to support future changes. It supports change impact analysis, requirements validation, long-term maintenance, and keeps developers informed about underlying architectural decisions [5].

The management of decision knowledge has been a research field for more than 40 years [9]. Despite potential benefits, its formal integration is met with resistance since it requires additional effort [23]. Further, its return on investment is said to be small when decisions are captured for the first time [15]. Nonetheless, developers informally capture decision knowledge in distributed locations such as issue comments or chat messages.

Recently, various techniques emerged that try to reconstruct decision knowledge by mining written text [2, 23], which is referred to as *extractive summarization* [18]. These techniques are promising in identifying decision knowledge, however, the knowledge may be incomplete, outdated, or hard to access later. In other cases, the knowledge is not captured at all, yet it only resides in developers' heads as tacit knowledge. Research attempts to infer tacit knowledge by *abstractive summarization* of software artifacts such as source code changes [7]. However, Robillard *et al.* confirm that it is unlikely to infer complex information such as rationale "by mechanical extraction of facts from software artifacts" [21]. Thus, summarization techniques only partially help to reconstruct decision knowledge in case they are applied retrospectively.

In this paper we sketch how summarization techniques can be integrated into developers' daily work to promote the capture and use of decision knowledge.

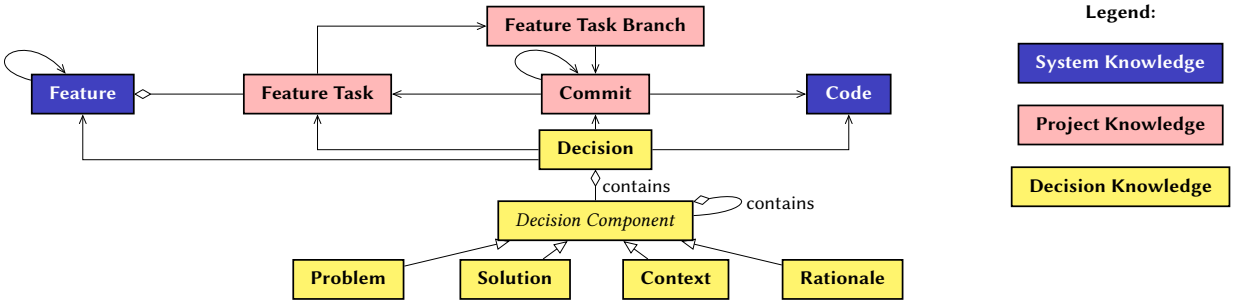

Figure 1: Relationship between features, tasks to implement features (feature tasks), code, commits, and decision knowledge.

We want to *trigger* developers to *package distributed decision knowledge* and to *make tacit decisions explicit*. We argue that the benefits of explicitly captured decision knowledge will emerge quickly during CSE when developers need to cope with change and reflect on former decisions. Therefore, we want to trigger developers to *consider consistency between decisions*. We suggest to integrate *decision knowledge triggers* into practices that relate to the version control system (VCS) and issue tracking system (ITS).

This paper is structured as follows. Section 2 introduces a knowledge metamodel. Section 3 presents ideas on how to trigger developers to capture and use decision knowledge. In Section 4, we discuss related work. Section 5 lists challenges involved in our approach. Section 6 concludes the paper.

2 KNOWLEDGE METAMODEL

Our knowledge metamodel is shown in Figure 1. Software artifacts contain knowledge that we classify into system and project knowledge [19]. *System knowledge* concerns the software itself (e. g., code, requirements, design, test cases), whereas the knowledge about its development and evolution is summarized under the term *project knowledge*. *Decision knowledge* can relate to both knowledge types.

In CSE, features are more prominent than components [3]. In this paper, we focus on features and code as essential system knowledge elements in CSE. Features address both functional and non-functional requirements. Features can be split into sub-features or grouped into bigger features.

We refer to the tasks that developers fulfill to implement a feature as *feature tasks*. Short-lived branches can be used to encapsulate the actual development work [16]. We refer to these branches as *feature task branches*. A feature task branch comprises one or more commits that bundle—refer to—code. When a feature task branch is merged into another branch, a merge commit is created. Feature tasks, feature task branches, and commits are types of project knowledge.

We use the *decision documentation model* by Hesse and Paech [12] to represent decision knowledge. According to this model, decision knowledge is captured as *decision components*, which can be nested and refer to other knowledge. We refer decision knowledge to features, feature tasks, commits, and code. In Figure 1, *decision component* is an abstract class that can only be instantiated through its subclasses. Decision components are the *problem* to be solved (issues or goals), *solution* (alternatives or claims), *rationale* (arguments), and *context* information (constraints or implications).

We assume that trace links between features, feature tasks, commits, and code are established (cf. the relationships in Figure 1). Developers can use these trace links to explore code and decisions that evolved during the implementation of a feature. Likewise, developers can see decisions relevant to a certain piece of code.

Evidently, there are other CSE artifacts which can contain relevant knowledge, e. g., user feedback, pull requests, or chat messages. We consider the artifacts in Figure 1 as the minimal set of CSE knowledge artifacts.

In the following, we introduce our implementation of this metamodel: Feature tasks are often called *tickets* and managed in an ITS [24]. For our approach, we manage both feature tasks and features in the ITS, whereas we manage code and commits in a VCS. In the ITS, we enable developers to create distinct decision knowledge elements linked to the respective features and feature tasks [14]. In the VCS, developers textually capture decision knowledge in commit messages and code. We encourage developers to mark it as such knowledge using *decision annotations* (Figure 2) as suggested by Hesse *et al.* [10]. The identifier of the feature task is added to the commit message. This satisfies the finding by Codoban *et al.* [6] that a "good commit message expresses the rationale of the change and provides a link to requirements".

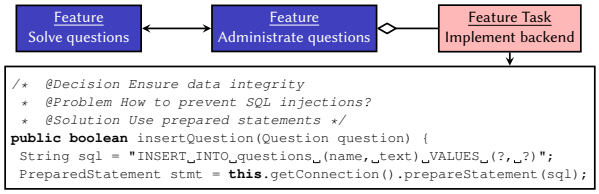

Figure 2: Decision knowledge captured in code.

Scenario. In this paper, we use the development of a mobile quiz app as a simplified scenario. A feature of this app should enable users to solve questions and a related feature should enable administrators to create, update, and delete these questions. Two feature tasks are created for each of these features: one task for the backend (database connection) and another one for the frontend (graphical user interface), respectively. First, developers implement the backend of the administrator feature. They add the *insertQuestion* method and are required to ensure data integrity. One problem is to prevent structured query language (SQL) injections. The developers decide to use *prepared statements* (Figure 2).

3 DECISION KNOWLEDGE TRIGGERS

CSE involves implementing and delivering many small increments. Practices advancing these increments are ideal to integrate *decision knowledge triggers*, i. e., techniques that trigger developers to capture and use decision knowledge. They are ideal because they are regularly performed by developers. Furthermore, they comprise practices that either indicate that developers *start* or *finish* work. A practice that indicates *start* is to open a feature task and to create a feature task branch. Practices that indicate *finish* are to commit code, merge a feature task branch, or close a feature task.

Before performing a *finish* practice, developers might have made important decisions. Thus, when developers perform a *finish* practice, we want to trigger them to explicitly capture decision knowledge. In particular, we support developers to *package distributed decision knowledge* and to *make tacit decisions explicit*. When developers perform a *start* practice, we want to trigger them to use existing decision knowledge to make sure they *consider consistency between old and new decisions*. We describe the techniques behind the decision knowledge triggers in the following.

3.1 Packaging Distributed Decision Knowledge

Decisions are most easily captured directly in the developers' working context [15]. For example, they are informally captured in descriptions and comments to feature tasks [11], pull requests [4], and in chat messages [2]. Our idea is to present developers with relevant distributed decision knowledge when they finish an implementation as indicated through a finish practice. Developers can check whether the decision knowledge really reflects the changes made. Thereby, we want to trigger them to package the most important decisions and to link them to the corresponding feature, feature task, or commits.

We want to present relevant distributed decision knowledge as *extractive summaries* using two techniques: 1) Developers could explicitly mark decision knowledge using decision annotations as presented by Hesse *et al.* for code [10] and Alkadhi *et al.* for chat messages [1]. Similarly, they could be enabled to apply such decision annotations in other CSE artifacts, e. g., in comments to feature tasks, pull requests, or wiki pages. 2) We mine the unstructured distributed decision knowledge by machine learning techniques similar to Rastkar & Murphy [20], Rogers *et al.* [23], and Alkadhi *et al.* [2]. All of these techniques require a gold standard to train a supervised classifier. It needs to be investigated to which extent such gold standards can be generalized to identify decision knowledge from different types of CSE artifacts.

Criteria for relevance for inclusion in extractive summaries could be a direct reference (e. g., decisions captured in the code to be committed) or an indirect reference (e. g., decisions mentioned in a recent chat or issue comment by the developer).

Scenario. Imagine the developers did not capture the decision knowledge as depicted in Figure 2 but discussed it in chat messages. Developers perform a finish practice when they close the respective feature task. Tool support extracts the decision knowledge from the chat messages and presents the knowledge to the developers. The developers acknowledge that *Ensure data integrity* is a decision they made. The decision knowledge is stored inside of the ITS and gets linked to the feature task.

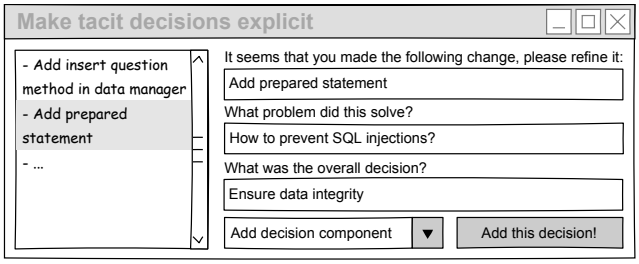

Figure 3: A summary of changes illustrated as a sketch.

3.2 Making Tacit Decisions Explicit

Many decisions remain tacit, that is, they are not captured anywhere but are already incorporated in the software. In our approach, we aim to present developers with *abstractive summaries* of changes to software artifacts when they perform a finish practice. By presenting *abstractive summaries* we want to trigger developers to make tacit decisions explicit, i. e., to reconstruct decision knowledge. Summarization of code will build on the summarization of source code changes as suggested by Cortés-Coy *et al.* [7].

Scenario. Imagine the developers did not discuss the decision *Ensure data integrity* (Figure 2) in a written form. That is, the decision resides tacit in the head of the developers. When the developers commit the changes, the summary *Add prepared statement* is suggested to them using summarization of the code changes (Figure 3). The developers can approve (or reject) that this is an important change and provide additional information on the decisions made such as the problems to be solved and rationale.

3.3 Considering Consistency Between Decisions

To ensure consistency between decisions, we focus on practices that indicate that a decision is to be taken. One example is when a developer sets the status of a feature task from *open* to *in progress*.

By presenting relevant decisions and system knowledge we want to trigger the developers to take previous decisions into account when working on the new task. Criteria for relevance are derived from the trace links in Figure 1. For example, relevant decisions and code could be derived from other feature tasks that are related to the same feature.

Scenario. The feature task to implement the backend for the feature that enables users to solve questions is assigned to a second developer. Since this feature is linked to the administrator feature, the code of the *insertQuestion* method as well as the decision *Ensure data integrity* is presented to this second developer when they set the status of the feature task from *open* to *in progress* (Figure 2). Thus, they will learn about the prevention of SQL injections and make decisions consistent with this previous one.

4 RELATED WORK

Clearly, this is not the first work that links features, relating tasks, commits, and code as shown in Figure 1. For example, Saito *et al.* [24] and Rastkar & Murphy [20] use a similar model. However, this is the first work to show how decision knowledge refers to the artifacts and to suggest decision annotations in commit messages.

We aim to support developers when they evolve a software system. Robillard *et al.* [21] identified three major categories of challenges for an *on-demand developer documentation* (O3D) that really meets the needs of developers. These challenges are: 1) information inference, 2) document request, and 3) document generation. Robillard *et al.* [21] in particular discuss rationale as part of the O3D. They state that rationale cannot automatically be inferred and that an "incentive to motivate the more systematic capture of rationale" is needed. Our approach addresses both problems by triggering developers to capture decision knowledge and by presenting this knowledge to them for an easy use.

5 CHALLENGES

To present relevant knowledge or to suggest meaningful abstractive summaries at code commit, it is important that developers only commit small changes—instead of "code bombs" [25]. This is supported by CSE, since it comprises techniques for work break-down and encourages developers to often commit changes and merge branches [16]. In addition, code changes need to be atomic (untangled) in a way that they only address the respective feature task. Tao and Kim empirically found that 29 % of commits in four open source projects were tangled [25]. Our approach can be combined with an approach to untangle code changes, e. g., [8, 25].

The combination of presenting extractive summaries of distributed decision knowledge and abstractive summaries of code changes could lead to information overload when a developer performs the *start or finish practice*. An *impact factor* as described in [7] could be used to personalize the presented decision knowledge according to the needs of individual developers.

Challenges remain in motivating developers to make use of the provided triggers; Roehm *et al.* found that developers tend to prefer face-to-face communication over documentation or use temporal notes that are for personal use only [22]. We hope that the ease of integrated, regular capture of small decision knowledge increments and the benefits of regularly using it, support the motivation.

6 CONCLUSION

CSE provides opportunities for the continuous management of decision knowledge using short-cycled practices. We presented how committing code, merging feature task branches, and changing the status of feature tasks can be used to integrate decision knowledge triggers. These start and finish practices are instances of change events during CSE, which activate or interrupt other activities [17]. The integration of decision knowledge triggers into other change events should be investigated. We will evaluate to which extent decision knowledge triggers support developers during CSE.

We started developing prototypes that implement our approach[1] and described requirements for a tool support in [14]. In particular, we are working on plug-ins for the ITS JIRA, a git-client, and an interactive dashboard for visualizing knowledge in CSE [13].

ACKNOWLEDGEMENT

This work was supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution (CURES project).

[1] https://github.com/cures-hub

REFERENCES

[1] Rana Alkadhi, Jan Ole Johanssen, Emitza Guzman, and Bernd Bruegge. 2017. REACT: An Approach for Capturing Rationale in Chat Messages. In *10th International Symposium on Empirical Software Engineering and Measurement*. 175–180.
[2] Rana Alkadhi, Teodora Lata, Emitza Guzman, and Bernd Bruegge. 2017. Rationale in Development Chat Messages: An Exploratory Study. In *14th Working Conference on Mining Software Repositories*. 436–446.
[3] Jan Bosch. 2014. *Continuous Software Engineering: An Introduction*. Springer.
[4] João Brunet, Gail C. Murphy, Ricardo Terra, Jorge Figueiredo, and Dalton Serey. 2014. Do developers discuss design?. In *11th Working Conference on Mining Software Repositories*. ACM, 340–343.
[5] Jane Cleland-Huang, Mehdi Mirakhorli, Adam Czauderna, and Mateusz Wieloch. 2013. Decision-Centric Traceability of architectural concerns. In *7th International Workshop on Traceability in Emerging Forms of Software Engineering*. IEEE, 5–11.
[6] Mihai Codoban, Sruti Srinivasa Ragavan, Danny Dig, and Brian Bailey. 2015. Software history under the lens: A study on why and how developers examine it. In *International Conference on Software Maintenance and Evolution*. 1–10.
[7] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On Automatically Generating Commit Messages via Summarization of Source Code Changes. In *14th International Working Conference on Source Code Analysis and Manipulation*. 275–284.
[8] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *22nd International Conference on Software Analysis, Evolution, and Reengineering*. 341–350.
[9] Allen H. Dutoit, Raymond McCall, Ivan Mistrík, and Barbara Paech. 2006. *Rationale Management in Software Engineering: Concepts and Techniques*. Springer.
[10] Tom-Michael Hesse, Arthur Kuehlwein, Barbara Paech, Tobias Roehm, and Bernd Bruegge. 2015. Documenting Implementation Decisions with Code Annotations. In *27th Int. Conf. on Software Engineering and Knowledge Engineering*. 152 – 157.
[11] Tom-Michael Hesse, Veronika Lerche, Marcus Seiler, Konstantin Knoess, and Barbara Paech. 2016. Documented decision-making strategies and decision knowledge in open source projects: An empirical study on Firefox issue reports. *Information and Software Technology* 79 (2016), 36–51.
[12] Tom-Michael Hesse and Barbara Paech. 2013. Supporting the Collaborative Development of Requirements and Architecture Documentation. In *3rd International Workshop on the Twin Peaks of Requirements and Architecture*. 22–26.
[13] Jan Ole Johanssen, Anja Kleebaum, Bernd Bruegge, and Barbara Paech. 2017. Towards the Visualization of Usage and Decision Knowledge in Continuous Software Engineering. In *5th Working Conference on Software Visualization*. IEEE, 104–108.
[14] Anja Kleebaum, Jan Ole Johanssen, Barbara Paech, and Bernd Bruegge. 2018. Tool Support for Decision and Usage Knowledge in Continuous Software Engineering. In *3rd Workshop on Continuous Software Engineering*. 74–77.
[15] Philippe Kruchten, Rafael Capilla, and Juan Carlos Dueñas. 2009. The Decision View's Role in Software Architecture Practice. *IEEE Software* 26, 2 (2009), 36–42.
[16] Stephan Krusche, Lukas Alperowitz, Bernd Bruegge, and Martin O. Wagner. 2014. Rugby: An Agile Process Model Based on Continuous Delivery. In *1st International Workshop on Rapid Continuous Software Engineering*. 42–50.
[17] Stephan Krusche and Bernd Bruegge. 2017. CSEPM - A Continuous Software Engineering Process Metamodel. In *3rd International Workshop on Rapid Continuous Software Engineering*. 2–8.
[18] Ani Nenkova and Kathleen McKeown. 2011. Automatic Summarization. *Foundations and Trends in Information Retrieval* 5, 2 (2011), 103–233.
[19] Barbara Paech, Alexander Delater, and Tom-Michael Hesse. 2014. Supporting Project Management Through Integrated Management of System and Project Knowledge. In *Software Project Management in a Changing World*, Günther Ruhe and Claes Wohlin (Eds.). Springer, Chapter 7, 157–192.
[20] Sarah Rastkar and Gail C. Murphy. 2013. Why did this code change?. In *35th International Conference on Software Engineering*. IEEE, 1193–1196.
[21] Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, Gail C. Murphy, Laura Moreno, David Shepherd, and Edmund Wong. 2017. On-Demand Developer Documentation. In *International Conference on Software Maintenance and Evolution*. 479–483.
[22] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software?. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 255–265.
[23] Benjamin Rogers, Yechen Qiao, James Gung, Tanmay Mathur, and Janet E. Burge. 2014. Using Text Mining Techniques to Extract Rationale from Existing Documentation. In *6th Int. Conf. on Design Computing and Cognition*. Springer, 457–474.
[24] Shinobu Saito, Yukako Iimura, Aaron K. Massey, and Annie I. Antón. 2017. How Much Undocumented Knowledge is there in Agile Software Development? Case Study on Industrial Project using Issue Tracking System and Version Control System. In *25th International Requirements Engineering Conference*.
[25] Yida Tao and Sunghun Kim. 2015. Partitioning Composite Code Changes to Facilitate Code Review. In *12th Working Conference on Mining Software Repositories*. 180–190.