Tool Support for Decision and Usage Knowledge in Continuous Software Engineering

Anja Kleebaum*, Jan Ole Johanssen†, Barbara Paech*, and Bernd Bruegge†

*Heidelberg University
Institute of Computer Science
Heidelberg, Germany
{kleebaum, paech}@informatik.uni-heidelberg.de

†Technical University of Munich
Department of Informatics
Munich, Germany
{jan.johanssen, bruegge}@in.tum.de

***Abstract*—Continuous software engineering copes with frequent changes and quickly evolving development projects while maintaining a high software quality. Developers require knowledge about former and ongoing decisions as well as about the users' needs to evolve software. Thus, decision and usage knowledge are two important knowledge types in continuous software engineering. Issue tracking and version control systems are widely used in continuous software engineering but lack a structured approach to integrate decision and usage knowledge. In this paper, we present ideas and requirements for a tool support to manage decision and usage knowledge in continuous software engineering. As a first step, we introduce the JIRA DecDoc plug-in for documenting decision knowledge.**

I. Introduction

In continuous software engineering (CSE), developers continuously handle change while maintaining a high software quality so that builds are releasable at any time [1]. For this purpose, developers create feature branches to implement a feature, perform code reviews prior to merging a feature branch, and write tests to ensure code quality [2].

In particular, both the individuals' and teams' knowledge are important to guarantee a high quality of the software. Every member of a software development team needs to know the major decisions made, so that their own decisions are consistent with the former ones. If developers lack such *decision knowledge* and make inconsistent decisions, they are likely to contribute to the erosion of the software architecture or introduce other quality problems. Hence, decision knowledge is important to handle change [3]. Decision knowledge covers knowledge about decisions, the problems they address, solution proposals, their context, and justifications—rationale—through arguments and assessments.

In CSE, a special focus is put on aligning software features to users' needs. The more the developers know about what the users want, the better they can develop and adopt software for them. Such *usage knowledge* can be derived from explicit and implicit user feedback. Short feedback cycles in CSE provide a great opportunity to integrate usage knowledge in the decision making process. Whenever an artifact has changed, or a new artifact increment evolved, the artifact creator can make the artifact available to the users. Then, during the artifact's usage, the users' behavior is observed and feedback is collected.

Thus, as already discussed in our previous work [4], decision and usage knowledge are two important knowledge types that need to be managed during CSE. Since this knowledge is complex, tool support is needed.

According to Burge and Brown, essential requirements for decision knowledge tool support are its integration into the development environment, direct association of knowledge with software artifacts (such as code), automatic presentation of the knowledge when needed, intuitive display of argumentation, automatic propagation and reevaluation when criteria change, and support for filtering and querying knowledge [5]. In this paper, we refine these requirements with CSE specific aspects in order to promote the paradigm shift in documentation [6].

Tool support to manage knowledge can be characterized by its intrusiveness in the software development process [3]. Tools that fit into the development context are less intrusive and will more likely be used [7]. Next to integrated development environments (IDE), issue tracking systems (ITS) and version control systems (VCS) are widely adopted [8]. We focus on integrating our tool support into these tools. As a first step, we introduce a plug-in for JIRA[1], the *JIRA DecDoc* plug-in.

This paper is structured as follows. Section II describes key ideas on how to handle decision knowledge in CSE. Section III presents the requirements for tool support for decision and usage knowledge in CSE. Section IV introduces the JIRA DecDoc plug-in. Section V discusses related work. Section VI concludes the paper and sketches ideas for further work.

II. Decision Knowledge in CSE

During CSE, developers collaboratively implement and deliver many small increments, which involves decision-making. We use the decision documentation model by Hesse & Paech [9] to represent the thereby emerging decision knowledge during CSE. This model supports incremental documentation of decisions, since it does not prescribe a complete template for decision documentation. Any part of the decision knowledge can be captured as soon as it is available. Furthermore, developers are able to collaborate while documenting decisions and contribute the part of the decision knowledge they know best.

[1]https://atlassian.com/software/jira

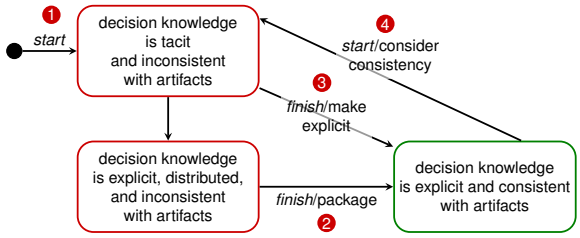

Fig. 1. Decision knowledge and artifacts shown as a state diagram. The state on the lower right side is the preferred state.

To exploit decision knowledge, it is important that decision knowledge is *consistent* with the software artifacts it concerns, e. g., with requirements, design, and code. Consistency means that decisions are documented as well as linked to and realized in the artifacts they relate to. Therefore, it is essential that decision knowledge is made explicit and checked for consistency.

However, in the beginning, decision knowledge is often *tacit* in the head of a few developers (Figure 1). If decisions are not tacit, they are often discussed informally, captured partly and in a distributed manner, such as in issue comments [10], commit messages, pull-requests [11], or chat messages [12]. We refer to this decision knowledge as *distributed knowledge*. This knowledge is hard to access later and might even be outdated. Therefore, we consider tacit and distributed knowledge as inconsistent with artifacts (cf. Figure 1, left).

Thus, tool support is needed to *explicitly capture decision knowledge consistent with artifacts* when developers *finish* some work. Practices that indicate that developers finished work are closing a feature task in the ITS and committing code or merging a feature branch in the VCS. In contrast, practices that indicate that developers *start* work are opening a feature task in the ITS—and for this purpose—the creation of a feature branch in the VCS. The integration of tool support into such short-cycled start and finish practices *triggers* developers to explicitly capture decision knowledge consistent with artifacts and exploit it afterwards, indicated by the labeled state transitions in Figure 1. The left side of these transitions indicates the type of CSE practice (*start* or *finish*), while the right side indicates the developers' tasks that we support (*package*, *make explicit*, and *consider consistency*). These transitions are further explained in the following section.

III. Tool Support

In this section, we introduce a running example, describe the requirements for tool support in CSE using this example, and map the requirements to tools.

Example Imagine the development of a software that computes three dimensional surface models of the earth from satellite stereo images. The idea behind this software is the following: An image matching algorithm detects pixels belonging to the same object—homologous points—on the stereo images. The distance between the homologous points—the disparity—is then used to calculate the relative height of the object, e. g., of a mountain. Thus, image matching is one essential feature of the software. In this example, developers decide on how to implement this image matching feature.

We derive requirements for tool support in CSE by aligning the running example to Figure 1 and evolving its decision in Figure 2. As motivated in Section II, a first requirement is:

R1 *Developers are supported in explicitly capturing decision knowledge consistent with artifacts in the tools they work with.*

A developer opens a feature task in the ITS to implement image matching, indicated by the *start* transition in Figure 1-①. Developers discuss image matching algorithms in chat messages and, thus, get into the state *decision knowledge is explicit, distributed, and inconsistent with artifacts*. One developer makes the proposal to implement a common image matching algorithm based on detecting the maximum zero normalized cross correlation. A second developer proposes to take advantage of algorithms provided by the open source computer vision library (OpenCV), which they think comes with less implementation effort. However, this also introduces third party code. Users will have a higher installation effort, since they have to make sure that the library is correctly provided by their operation system. Finally, the developers decide to implement their own image matching algorithm on a distinct feature branch in the VCS. When merging this branch back to the mainline, they perform a *finish practice*. A second requirement for our tool support is:

R2 *Developers are presented with distributed decision knowledge when performing a finish practice.*

Criteria to classify knowledge as relevant may be its creation within a specific time frame, by the same person, or textual similarity with the feature task. In this case, relevant decision knowledge is extracted from a history of chat messages [12]. The developers *package* this knowledge to make it explicit and consistent with the implemented code (Figure 1-②). Here, packaging means that knowledge is captured as in Figure 2-①. If the developers had not explicitly discussed the implementation of the image matching algorithm, this decision knowledge would be *tacit and inconsistent with the code* (Figure 1). In this case, the *finish practice* would be that they commit code. A third requirement for our tool support is:

R3 *Developers are presented with summarized artifact changes when performing a finish practice.*

As a result, the developers would be presented with the summarized change that they "inserted methods to calculate the average and the zero normalized cross correlation" inferred from the code in Figure 2-①. To achieve this, a tool parses the code before and after the change and compares the nodes of the two syntax trees [13]. This triggers developers to reconstruct decision knowledge, to *make tacit decision knowledge explicit* (Figure 1-③). Ideally, developers document decision knowledge similar to the packaged one in Figure 2-①.

The software is continuously deployed to users. After a while, multiple users provide explicit feedback in the form of written text stating that the image matching takes a fairly long time when they process new high-resolution satellite images (Figure 2-②). A forth requirement for our tool support is:

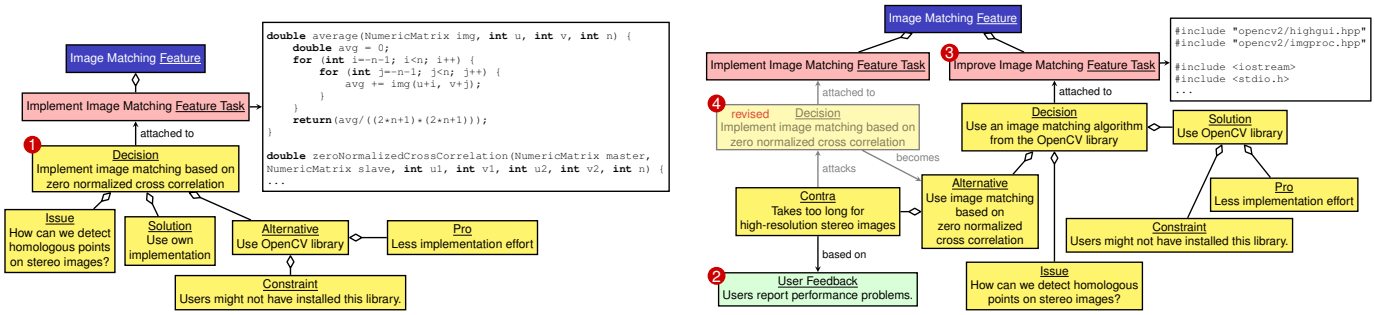

Fig. 2. Image matching decision before (left) and after (right) employing usage knowledge. Yellow items are decision knowledge.

R4 *Developers are presented with assessed user feedback, resulting in pro and contra arguments that are linked to the respective decision knowledge or to new proposals.*

Based on this insight, a developer creates the contra argument and attaches it to the image matching decision. Alternatively, the tool support could automatically assess implicit user feedback when a certain threshold is reached, such as a drop in feature usage after its internals have been changed. After that, the developers create a new feature task to improve image matching (Figure 2-③). Developers who open the feature task perform a *start practice*. A fifth requirement is:

R5 *Developers are presented with relevant knowledge and artifacts when performing a start practice.*

The decision knowledge as well as the former feature task and code in Figure 2-① are presented to the developers. This triggers the developers to *consider consistency* when changing the software (Figure 1-④). Figure 1-① indicates a start practice without any initial knowledge for presentation. By reflecting the former decision knowledge, the developers remember the alternative that they could employ the OpenCV library, which offers fast image matching methods. Therefore, they revise their former decision (Figure 2-④) and implement the new code. After a while, a new developer joins the project team and is asked to improve the image matching even further. The sixth requirement for our tool support is:

R6 *Developers are supported in accessing knowledge by filtering and searching from within the tools they use and from a dashboard, not bound to start and finish practices, also from artifacts such as code and features.*

To learn about existing knowledge, developers use a knowledge dashboard [4] and search for image matching. They are presented with the feature, the two feature tasks, code, decision and usage knowledge as depicted on the right side of Figure 2.

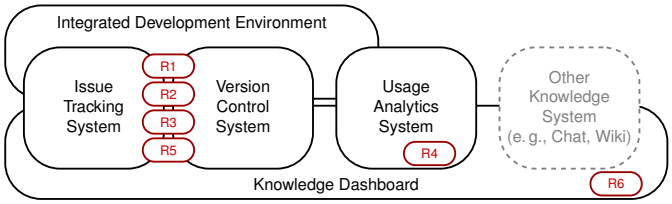

Fig. 3. Requirements mapped to tools.

These requirements could be fulfilled by multiple tools, though we prefer their implementation into that tool the developers work with. Figure 3 shows a possible mapping from the requirements to tools.

IV. JIRA DecDoc Plug-in

In this section, we present the JIRA DecDoc plug-in[2] as a first ITS extension towards our tool support. Thurimella *et al.* suggest to customize JIRA to support rationale guidelines [14], but to the best of our knowledge, this is the first work to explicitly capture decision knowledge in JIRA. The JIRA DecDoc plug-in is based on the Unicase DecDoc tool [15]. Both tools—JIRA and Unicase DecDoc—support developers in documenting decision knowledge in a structured, collaborative, and incremental way according to the decision documentation model (DDM) by Hesse & Paech [9]. JIRA DecDoc transfers the concepts from the research prototype Unicase DecDoc to the widely used commercial tool JIRA.

The JIRA DecDoc plug-in supports two strategies to implement the DDM: the *issue strategy* and the *active object strategy*. The issue strategy represents the concepts of the DDM as JIRA issues. JIRA issue links are used to link DDM elements to each other and to JIRA issues of other types such as feature tasks. The advantage of this strategy is that all features available for JIRA issues can be used to manage decision knowledge, e. g., searching for a decision in the list of issues. The disadvantage is that the dedicated issue type scheme needs to be assigned to the JIRA project. To overcome this disadvantage, the active object strategy uses distinct model classes for decision, decision components, and links. This strategy uses object-relational mapping to communicate with JIRA's internal database. The JIRA DecDoc plug-in provides a view that presents decision knowledge similar to Figure 2. Developers are supported to document decision knowledge either by using a context menu or an accordion editor. Furthermore, JIRA DecDoc enables developers to textually filter decision knowledge. The JIRA DecDoc plug-in fulfills the requirement R1. It partly fulfills the requirement R6, as it allows to filter for decision knowledge. Thus, it provides the key decision knowledge infrastructure that facilitates the implementation of the other requirements on top of it.

[2]The JIRA DecDoc plugin as well as further implementation details and screenshots are available at https://github.com/cures-hub.

V. Related Work

There are various tools to manage decision knowledge during software development. Hesse *et al*. [15] and Capilla *et al*. [16] provide a comparison between existing tools, such as SEURAT [5] or Archie [17]. Alexeeva *et al*. provide a literature overview about 56 decision documentation approaches of which 32 provide tool support [18]. The requirements for our tool support differ from other tools as follows: We use short-cycled CSE practices to integrate tool support that triggers the developers to document and exploit knowledge, which none of the existing tools address, e. g., merging of feature branches in the VCS or opening and closing a feature task in the ITS. We will integrate informal and distributed knowledge from sources such as chat messages, pull requests, or issue comments. We will employ summarization techniques to encourage the reconstruction of decision knowledge. To the best of our knowledge, none of the existing tools integrates usage knowledge into the decision-making process. Leveraging the capabilities of CSE, we will support developers in reflecting on usage knowledge.

For usage knowledge, Guzman *et al*. report on an interactive visualization to display the summarization of unstructured user feedback in the form of app reviews [19]. Maalej and Nabil describe the automatic classification of explicit application reviews into four types and derive requirements for an analytic tool [20]. We extend the presented work in the context of usage knowledge with a relation to decision knowledge, combined with a continuous analysis perspective enabled by CSE.

VI. Conclusion and Future Work

Continuous software engineering provides great opportunities for employing decision and usage knowledge—if tool support is provided. We introduced six requirements for establishing a tool support and presented the JIRA DecDoc plug-in as a first step for their implementation.

We will continue to extend tool support for decision and usage knowledge in JIRA by implementing requirements R2 to R6. We will implement an IDE-integrated Git client, which supports developers both in accessing decision knowledge from code and commits, as well as in documenting it when committing code or merging branches. It also will support changes of decision knowledge and code.

There are various kinds of usage knowledge that are not addressed in this paper. In requirement R4, we point out the relevance of explicit usage knowledge in the form of written user feedback. However, implicit usage knowledge, such as the results of A/B tests or controlled experiments, require further tool support. We are working on a usage analytics system to explore requirements and enable usage knowledge management in CSE.

To understand the developers' needs, we plan to evaluate the tool support during industrial projects that are part of a practical course at university. In particular, we will investigate which knowledge is worth capturing. Furthermore, we will clarify how to maintain the knowledge in order to keep it useful and how to access the relevant parts of knowledge.

Acknowledgement

This work was supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution (CURES project). We thank Ewald Rode and Tim Kuchenbuch for their work on the JIRA DecDoc plugin and their very helpful discussions.

References

[1] S. Krusche and B. Bruegge, "CSEPM - A continuous software engineering process metamodel," in *3rd Int. Workshop on Rapid Continuous Software Engineering*, 2017, pp. 2–8.

[2] S. Krusche, L. Alperowitz, B. Bruegge, and M. O. Wagner, "Rugby: An agile process model based on continuous delivery," in *1st Int. Workshop on Rapid Continuous Software Engineering*, 2014, pp. 42–50.

[3] A. H. Dutoit, R. McCall, I. Mistrík, and B. Paech, *Rationale Management in Softw. Engineering: Concepts and Techniques*. Springer, 2006.

[4] J. O. Johanssen, A. Kleebaum, B. Bruegge, and B. Paech, "Towards a systematic approach to integrate usage and decision knowledge in continuous software engineering," in *2nd Workshop on Continuous Software Engineering*, 2017, pp. 7–11.

[5] J. E. Burge and D. C. Brown, "Software engineering using RATionale," *Journal of Systems and Software*, vol. 81, no. 3, pp. 395–413, 2008.

[6] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez, G. C. Murphy, L. Moreno, D. Shepherd, and E. Wong, "On-demand developer documentation," in *Int. Conf. on Softw. Mainten. and Evol.*, 2017, p. 5.

[7] P. Kruchten, R. Capilla, and J. C. Dueñas, "The decision view's role in software architecture practice," *IEEE Software*, vol. 26, no. 2, pp. 36–42, 2009.

[8] S. Saito, Y. Iimura, A. K. Massey, and A. I. Antón, "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 Int. Requir. Eng. Conf.*, 2017, pp. 186–195.

[9] T.-M. Hesse and B. Paech, "Supporting the collaborative development of requirements and architecture documentation," in *3rd Int. Workshop on the Twin Peaks of Requir. and Architecture*. IEEE, 2013, pp. 22–26.

[10] T.-M. Hesse, V. Lerche, M. Seiler, K. Knoess, and B. Paech, "Documented decision-making strategies and decision knowledge in open source projects: An empirical study on firefox issue reports," *Information and Software Technology*, vol. 79, pp. 36–51, 2016.

[11] J. Brunet, G. C. Murphy, R. Terra, J. Figueiredo, and D. Serey, "Do developers discuss design?" in *11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 340–343.

[12] R. Alkadhi, T. Lata, E. Guzman, and B. Bruegge, "Rationale in development chat messages: An exploratory study," in *14th Int. Conference on Mining Software Repositories*. IEEE, 2017, pp. 436–446.

[13] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *14th Int. Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 275–284.

[14] A. K. Thurimella, M. Schubanz, A. Pleuss, and G. Botterweck, "Guidelines for Managing Requirements Rationales," *IEEE Software*, vol. 34, no. 1, pp. 82–90, 2017.

[15] T.-M. Hesse, A. Kuehlwein, and T. Roehm, "DecDoc: A tool for documenting design decisions collaboratively and incrementally," in *1st Int. Workshop on Dec. Making in Softw. ARCHitecture*, 2016, pp. 30–37.

[16] R. Capilla, A. Jansen, A. Tang, P. Avgeriou, and M. A. Babar, "10 years of software architecture knowledge management: Practice and future," *Journal of Systems and Software*, vol. 116, pp. 191–205, 2016.

[17] J. Cleland-Huang, M. Mirakhorli, A. Czauderna, and M. Wieloch, "Decision-centric traceability of architectural concerns," in *7th Int. Workshop on Traceability in Emerging Forms of Software Engineering*. IEEE, 2013, pp. 5–11.

[18] Z. Alexeeva, D. Perez-Palacin, and R. Mirandola, "Design decision documentation: A literature overview," in *Software Architecture*, ser. LNCS. Springer Berlin Heidelberg, 2016, vol. 5292, pp. 84–101.

[19] E. Guzman, P. Bhuvanagiri, and B. Bruegge, "FAVe: Visualizing user feedback for software evolution," in *2nd Working Conference on Software Visualization*. IEEE, 2014, pp. 167–171.

[20] W. Maalej and H. Nabil, "Bug report, feature request, or simply praise? On automatically classifying app reviews," in *23rd International Requirements Engineering Conference*. IEEE, 2015, pp. 116–125.