Documenting Implementation Decisions with Code Annotations

Tom-Michael Hesse[1], Arthur Kuehlwein[1], Barbara Paech[1], Tobias Roehm[2] and Bernd Bruegge[2]
[1]Heidelberg University, Im Neuenheimer Feld 326, 69120 Heidelberg, Germany
{hesse, kuehlwein, paech}@informatik.uni-heidelberg.de
[2]Technische Universität München, Boltzmannstr. 3, 85748 Garching b. München, Germany
{roehm, bruegge}@in.tum.de

Abstract

Software developers make various decisions when implementing software. For instance, they decide on how to implement an algorithm most efficiently or in which way to process user input. When code is revisited during maintenance, the underlying decisions need to be understood and possibly adjusted to the current situation. Common documentation approaches like JavaDoc neither cover knowledge related to decisions explicitly, nor are they integrated closely with knowledge management. In consequence, decision knowledge is rarely documented and therefore inaccessible, especially when developers have left the team. So, effective maintenance is hindered. We have developed an annotation model for decision knowledge and integrated it with the knowledge management tool UNICASE. The approach enables developers to document decisions within code without tool switches to lower their documentation effort. Afterwards, maintainers can exploit the embedded decision knowledge and follow links to external knowledge. This paper presents the approach and evaluation results of a first case study, which indicate its practicability.

1 Introduction

During the implementation of software, developers make many decisions, e.g. on how to implement an algorithm most efficiently or in which way to process user input. This means to solve a decision problem, which comprises a set of alternatives and criteria to compare them [16]. A comparison of alternatives, like using an external library instead of programming an algorithm, can be made by considering expert knowledge, personal experiences and the context of the decision. So, a complex and large amount of knowledge is required to understand a decision problem in retrospect. We will refer to this knowledge as *decision knowledge*.

Over time, decision knowledge can erode easily [12]. As a result, most information needs of developers towards implementation decisions cannot be satisfied sufficiently. This is shown in a study of Ko et al. at Mircosoft with 17 software developer teams [13]. For instance, the question "Why was the code implemented this way?" could not be answered in 44% of the cases. The major reasons are that decisions either are documented within unstructured inline comments, are not documented at all or have to be inferred from external documents without links to code [15]. These reasons imply three requirements our approach has to fulfill.

First, implementation decision are difficult to understand in retrospect, when no documentation structures are defined and unstructured inline comments are used. So, defined structures for decision documentation are required. Even frameworks like JavaDoc only provide limited capabilities for documenting decisions, as they focus on describing what was implemented, but not on the underlying decisions. But a defined documentation template for decisions often requires more than the decision knowledge, which is currently present. In consequence, a *structured, but incremental capture* of decision knowledge is required (requirement R1). Second, implementation decisions may concern code parts of different granularity levels, such as the usage of a particular operation or the purpose of an entire class. If developers cannot document decisions directly within the code, they either do not document decisions at all or have to interrupt their current implementation task and change to some external documentation tool. Therefore, decision knowledge should be *embedded within the code* for different levels of code granularity (requirement R2). Third, when decision knowledge is not linked to external documents like requirements or design diagrams, such decision-related external knowledge cannot be exploited easily. This again can cause high efforts and thereby hinders the assessment of implementation decisions by developers during system maintenance. Therefore, decision knowledge should be *linked to related external knowledge* within code (requirement R3).

The contribution of this paper is an approach, which adheres to these requirements. First, we propose a documentation approach based on code annotations, which is inte-

DOI reference number: 10.18293/SEKE2015-084

grated with knowledge management. Therefore, we derive appropriate annotations for decision knowledge from an existing knowledge model for decisions, and implement these annotations in Eclipse. The set of annotations covers many elements of decision knowledge and can be used in an incremental way without a static template. Second, we integrate our approach with the model-based knowledge management tool UNICASE [3]. This allows for links between annotations and external knowledge within UNICASE. Overall, our approach supports decision documentation within code for developers and makes decision knowledge explicit and exploitable during maintenance.

The remainder of this paper is structured as follows. Section 2 introduces background information and discusses related work. Section 3 describes our approach with a running example. In Section 4, we present results for a first evaluation of our approach. We summarize our insights in Section 5 and describe directions for future work.

2 Background and Related Work

In this section, we define *decision knowledge* in detail and introduce the *knowledge management tool UNICASE*. Then, we give a brief overview of *existing annotation approaches* for decision knowledge in code.

Decision Knowledge As defined in Section 1, *Decision knowledge* addresses all information required to understand a given decision problem with its context and rationale justifying the decision. Decision problems comprise a set of alternatives, which are compared by different criteria [16]. In practice, these criteria and the resulting rationale for the decision depend on various context aspects. For instance, constraints brought up by former design decisions or assumptions on the environment of the system shape decisions. Moreover, rationale for decisions might be influenced by time pressure in the project or personal experiences of developers [19]. As we have described in [17], many models exist that cover parts of this knowledge for different activities, for example in requirements engineering or design. However, none of these models is addressing implementation decisions, they do not support an incremental documentation and have only limited support for pre-defined links.

Due to these shortcomings, we decided to use a flexible documentation model as presented in [11]. It consists of a set of different decision knowledge elements, which may be aggregated for a decision incrementally over time by different developers. The model is depicted in Figure 1. The basic element is the *Decision*, which contains all related decision knowledge elements for one decision as *DecisionComponents*. Amongst others, DecisionComponents can be refined to a decision problem description as an *Issue*, to context information like an *Assumption*, to a solution description as an *Alternative*, or to a description of a rationale as an *Argument*. Decisions and their components can be linked to external knowledge like requirements specifications or design diagrams.

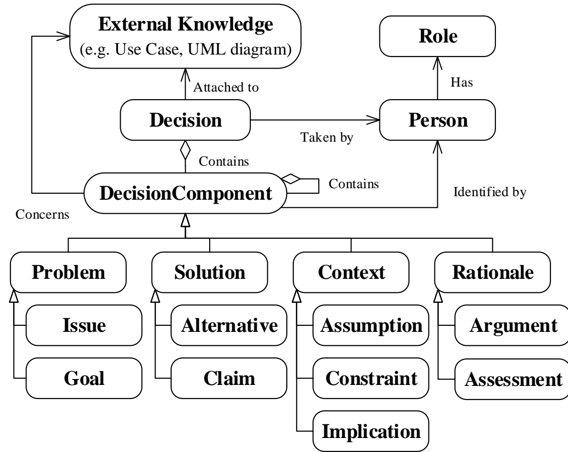

Figure 1. Documentation Model for Decisions

Knowledge Management Tool UNICASE Our annotation model is integrated with the model-based knowledge management tool UNICASE [3]. UNICASE is an Eclipse extension and provides an integrated model for system and project knowledge [5]. UNICASE offers a generic support for the collaborative editing of the underlying model elements based on the Eclipse Modeling Framework EMF [1] and the model versioning system EMFStore [2]. Moreover, it provides a variety of elements for documenting and structuring external knowledge, like use cases, UML diagrams or test protocols.

Existing Annotation Approaches In the last decades, only a few approaches were developed to document decision knowledge explicitly within code. Typically, they focus on rationale. As one of the first approaches, Lougher and Rodden introduced a system to annotate rationale in the source code of software using comments [15]. Then, a documentation is generated as a network of linked hypertext documents out of these comments. Whereas the approach claims to use a markup language for comments, no concrete proposals for well-defined structures for such a language are made. Moreover, the system is not integrated with external knowledge sources. So, this approach does not satisfy R1 and R3. Canfora et al. propose the "Cooperative Maintenance Conceptual Model" (CM^2) [7]. It also structures rationale knowledge as a network of linked comments for analysis, design and implementation. Also for this approach well-defined structures for comments are missing. Moreover, the links between analysis and design artifacts

with the comments are not specified in detail. In consequence, also this approach does not fulfill R1 and R3. Burge and Brown present the system "Software Engineering Using RATionale" (SEURAT), which is an Eclipse extension based on an ontology for rationale knowledge [6]. It can highlight existing rationale in the code via Eclipse markers as well as infer unresolved issues of inconsistencies. But the documentation of newly acquired rationale is not possible within the code, as the ontology has to be extended externally. Also, links to external knowledge are limited, as the approach focuses on linking rationale and code files. So, R2 and R3 are not satisfied. Other approaches enable developers implicitly to exploit decision knowledge by creating traceability links, e.g. between requirements and code. For instance, the approach of Cleland-Huang et al. [8] traces architectural significant requirements to code. This enables developers to reflect that a part of code realizes the linked requirements. However, such traceability links typically do not support incremental modifications and are not embedded within the code, so they do not fulfill R1 and R2. In summary, to the best of our knowledge no current approach realizes all three introduced requirements for decision documentation of implementation decisions.

3 Annotation Model for Decision Knowledge

Based on the documentation model for decision knowledge, we derived one annotation for each decision knowledge element and integrated the annotations with UNICASE. This implementation of our approach is available via an Eclipse update site [4]. In the following sections, we introduce a running example and describe how our model realizes the three requirements presented in Section1.

Running Example To explain our annotation model, we will employ the decision on implementing a wizard instead of a dialog as example. This is a typical decision point when programming plug-ins for Eclipse. On the one hand, a wizard provides multiple pages for a guided user interaction, but typically requires multiple user actions for stepping through the pages. Moreover, it often implies complex data handling, when input checks for each page are performed. On the other hand, a dialog only offers a single page, so that a step-wise user interaction is not possible directly. However, a dialog typically requires less user actions, because it just consists of one page. Depending on the actual decision context, either a wizard or a dialog are more appropriate. We will refer to this decision in the following sections and enrich it with further information to demonstrate our approach.

Annotation Structure All annotations are mapped to a corresponding decision knowledge element of the documentation model (cf. Figure 1). An annotation can be used to either create a new decision knowledge element or link to an existing one. Each annotation contains several internal attributes to deal with references to external knowledge and persistent storage. The textual content that is to be documented by the annotation can be typed directly after the annotation itself by the developer, as depicted in Figure 2 using our wizard example. We established two different kinds of annotations with different functional complexity: core annotations and augmented annotations.

Core annotations represent a decision knowledge element in the documentation model, for instance an issue as `@Issue` or an alternative as `@Alternative`. We created one core annotation for each knowledge element given by the documentation model. *Augmented annotations* represent one or more decision knowledge elements with predefined attributes or relations. This is a shortcut for developers in practice to create decision knowledge elements by patterns, so that the manual documentation effort can be reduced. For instance, `@Contra` can be used to create a new argument as a child of the nearest DecisionComponent and to link the argument to this component as an attacking argument. In our wizard example, the `@Contra`-annotation is used to argue against the "dialog"-alternative (cf. Figure 2).

This structure of annotations suits an incremental and flexible use. For instance, if a `@Decision` statement is already given, a developer can simply add another annotation like `@Alternative` to document a newly arisen alternative during re-engineering for this decision. So, developers can complete and update the given documentation in case this decision knowledge is incomplete or outdated. Also, they are not forced to document a pre-defined default set of annotations for a decision. Our approach is extensible, as developers can define their own customized augmented annotations. Overall, this enables a structured and incremental documentation of decisions and thus fulfills R1.

Annotation Embedding To implement our annotation model, we created a new annotation parser within Eclipse, which makes the annotations available for use. Decision annotations can be used in any inline code comment, including JavaDoc, to annotate class and method declarations as well as any code part within the method body. All annotations are written directly into the code file, so that their textual contents are not lost when the code is stored in a code versioning system. Whenever a developer types an annotation,

```
// @Decision Implement input UI using a wizard
// @Issue Complex user input
// @Alternative Use a dialog
// @Contra Need for step-wise user guidance
```

Figure 2. Examples of Decision Annotations

options for creating or linking related decision knowledge are displayed by hovering over the annotation. In addition, our implementation in Eclipse allows to directly create new knowledge elements as children of the nearest decision that is found in the code before, as depicted in Figure 3. Moreover, developers can annotate elements of one decision in different comments on different code parts, as long as no other decision is inserted in between. This addresses the problem of different code granularity levels and enables a documentation of implementation decisions directly within the code, so R2 is fulfilled.

Integration with Knowledge Management UNICASE allows for relating decision knowledge elements with annotations to ingrate them into knowledge management. This integration requires that for each annotation in code a corresponding decision knowledge element in the management tool is created or linked. In consequence, all decision knowledge elements were added to the UNICASE model. Then, any other UNICASE knowledge element can be linked with the decision knowledge element corresponding to the annotation. However, an explicit link is needed to relate code annotations and decision knowledge elements in UNICASE. In our model, this is done by the AnnotationLink knowledge element as the parent knowledge element for CoreAnnotation and AugmentedAnnotation, which all three were added to the UNICASE model. The AnnotationLink provides a relation to the decision knowledge element and uses the ID of the Eclipse marker for linking to an annotation. In addition, the current revision of the code file and the decision knowledge element is stored. These revisions are updated, whenever a change in code or knowledge management impacts an annotation. So, a collaborative, distributed usage of annotations is supported. This mapping is depicted in Figure 4.

In our approach, developers are enabled to create, modify or delete both annotations and knowledge elements as summarized in Table 1. For new annotations, developers decide to either create a related decision element or link the annotation to an existing one. Then, annotations can be related to any further UNICASE elements representing the related external knowledge. Modifying annotations requires the corresponding decision elements to be updated,

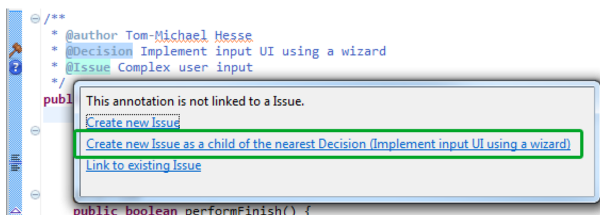

Figure 3. Create Elements using Annotations

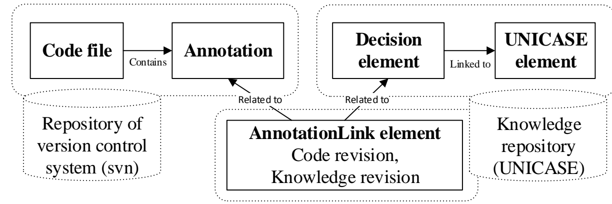

Figure 4. Relating Code Annotations and Decision Knowledge via AnnotationLink

whereas updates of decision elements in UNICASE may also require an annotation update. Considering our wizard example, a developer can document how the wizard class was embedded in the existing design. When annotations or their related decision elements are deleted, the related AnnotationLink is removed. If a deleted annotation was used to create a decision element, this corresponding element is also removed. If a decision knowledge element is updated or deleted, the related annotations also have to be updated or deleted within the code. However, this is currently not implemented due to restrictions and missing functionality in the employed Eclipse version 3.7. Through these actions, our annotation model enables developers to link any external knowledge consistently with annotations and thereby fulfills R3.

4 Evaluation

To investigate the practicability of our approach for other developers, we performed a first case study with students. We present its results in this section. However, this does not show the practicability of our approach in industry.

Context We performed a case study within a practical course for undergraduate students in computer science at Heidelberg University. Within the course, 7 participating students were grouped in two development teams in order to realize a software development project with identical project descriptions. Their task was to plan, implement and document an Eclipse plugin. We acted as the "customer"

Table 1. Impact of Developer Actions

Action	Performed on Annotations	Performed on UNICASE Elements
Create	Create new decision knowledge element or link existing one	No effect on annotation
Modify	Update decision knowledge element content, references, AnnotationLink	Update annotation, AnnotationLink
Delete	Delete decision knowledge element, AnnotationLink	Delete annotation, AnnotationLink

in both projects and provided an initial set of requirements as scenario descriptions, which were not changed during the project. Both projects were divided into three sprints lasted three weeks from mid February until the beginning of March 2015. For both teams, an initial tutorial for UNICASE and the code annotations was held to reduce the variability of competency concerning the annotations for the students. In addition, we provided textual explanations on how to use the code annotations to both teams. However, there was no mandatory rule for the students to use the annotations during implementation in order to get a realistic impression of the actual annotation usage. At the end of each sprints, the teams held a presentation to report on their current progress.

Research Questions and Method Our goal was to investigate the practicability of our approach referring to the research question: Is the annotation model and its implementation practicable to document implementation decisions? To evaluate this question, we build upon the Technology Acceptance Model (TAM) [9] to explore the actual use of our approach. TAM consists of three variables: *Ease of use* describes the degree to which a person expects the approach to be effortless, *usefulness* is defined as the subjective probability for a person to increase job or work performance and *intention to use* determines a persons' willingness to use the approach in the future. We assessed these variables with three anonymous questionnaires. Each questionnaire belonged to one sprint. They were answered by the students after each sprint presentation. We derived questions on using the annotations for each variable, as listed in Table 2. All questions were formulated as statements with defined answers in order to ensure the comparability of the students' responses. With statement #1 and #2, we distributed our investigation of ease of use on the creation and usage of annotations. Statement #3 and #4 address usefulness and intention of use for the entire approach. The answers represent a six point Likert scale [14], as this is an established approach in survey research. If the majority of subjects marks four or higher on the scale, we consider a statement to be accepted.

Table 2. Questionnaire Statements

No.	Statement	Variable
#1	It was easy to create decision elements with code annotations.	Ease of use
#2	It was easy to locate decision elements within the Eclipse Code Editor.	Ease of use
#3	Code annotations have been useful for the documentation of decisions.	Usefulness
#4	In the future I would use code annotations again to document decisions.	Intention of use

Table 3. Questionnaire Results

Sprint no.	Statement no.	Strongly disagree	Disagree	Rather disagree	Rather agree	Agree	Strongly agree	Not used, no answer	$\sum$ Disagree, Agree	Accepted
1	#1	0	0	0	0	1	2	4	0 / 3	yes
	#2	0	0	0	2	1	1	3	0 / 4	yes
	#3	0	1	0	3	1	1	1	1 / 5	yes
2	#1	0	0	0	1	2	1	3	0 / 4	yes
	#2	0	0	0	2	1	1	3	0 / 4	yes
	#3	0	1	0	0	1	3	2	1 / 4	yes
3	#1	0	0	1	0	3	2	1	1 / 5	yes
	#2	0	1	0	0	3	0	3	1 / 3	yes
	#3	0	0	1	0	2	2	2	1 / 4	yes
	#4	0	1	1	0	4	1	-	2 / 5	yes

Note, that only questionnaire 3 contained statement #4, as it addresses the overall experience with annotations during all sprints. For this statement, the "not used"-answer was not given. As the number of students does not permit to achieve statistical evidence, we also asked for rationale and comments in general and for each statement. This allowed us to collect as much individual feedback as possible.

Results The results from all questionnaires are presented in Table 3. Over time, more students used the code annotations, so that the sum of "Not used, no answer"-results slightly declines in sprint 3. Whereas the high number of "Not used, no answer"-answers especially in the first sprint provides only a limited support for the statements, no statement has to be rejected according to the number of rejecting answers. In consequence, this indicates that our approach is practicable for documenting implementation decisions with annotations. Multiple students pointed out in their feedback, that the approach was very useful to document decisions within the code in order to remember and reflect them. However, there was also a rejecting answer in the first two questionnaires concerning the usefulness of the annotations and several rejecting answers in the last questionnaire. This might be due to some errors in the integration of annotations and the code versioning system, which caused decisions to be represented at incorrect locations within the code. These errors partly are related to the employed Eclipse version 3.7 and were not entirely fixed during the course. Also, after trying our approach some students made proposals for functionality enhancements. For instance, they proposed to add keywords to annotations in order to create references to other decisions when typing the annotation.

Threats to Validity According to Runeson et al. [18], four different types of threats to validity have to be con-

sidered for our study. Concerning the *internal validity*, the students' knowledge was varying and they were not experienced in software engineering. To address this factor, we provided a tutorial for our approach and grouped the teams according to the students' subjective experience levels. However, missing experience could not be balanced completely. Concerning the *external validity*, the development projects had a rather small size regarding time, requirements, and team size. So, the evaluation of the usefulness of our approach might be affected. In addition, the results for the investigated student projects are incomparable to industry projects due to different project settings. However, Eclipse is a common tool in industry and also UNICASE has been used in an industry setting [10]. So, we believe that the usage through the students gives a first indication that our approach is useful in practice. Concerning *construct validity*, the questionnaires could have measured something different than TAM, as they were not evaluated prior to the study. However, we used typical questions for TAM. *Reliability validity* can be impacted by the fact, that the students knew we were investigating decision annotations. But this impact is unlikely to be high, as the investigators were not involved in the students' grading.

5 Conclusion and Future Work

This paper presented an approach to document implementation decisions using annotations in source code. The approach supports the structured and incremental capture of decisions within code without switching to a documentation tool. Moreover, external knowledge from knowledge management tools can be linked to annotated decisions. To the best of our knowledge no other approach addresses all of these requirements. The approach consists of an annotation model and is integrated with the knowledge management tool UNICASE. Evaluation results of a first case study were presented, which indicate the practicability of our approach.

In our future work, we will extend and improve our implementation of the annotation model. For instance, bugs with the integration of the code versioning system in the current implementation should be fixed and more code versioning systems (e.g., git) should be integrated. Moreover, we want to realize the functionality improvements acquired in the case study. Augmented annotations in our model could be extended, so that they can handle keywords as references on former or similar decisions. We also plan to execute further case studies in advanced practical courses and industry to overcome the shortcomings of the current study.

Acknowledgement This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future — Managed Software Evolution. We thank all students participating in our case study.

References

[1] EMF. http://eclipse.org/modeling/emf/ (05-2015).
[2] EMFStore. http://eclipse.org/emfstore/ (05-2015).
[3] UNICASE. http://unicase.org/ (05-2015).
[4] Update Site for Decision Annotations. http://svn.ifi.uni-heidelberg.de/unicase/0.5.2/ures/decdoc-features/ (05-2015).
[5] B. Bruegge, O. Creighton, J. Helming, and M. Koegel. Unicase - An Ecosystem for Unified Software Engineering Research Tools. In *International Conference on Global Software Engineering*, pages 1–6. IEEE, 2008.
[6] J. E. Burge and D. C. Brown. Software Engineering Using RATionale. *Journal of Systems and Software*, 81(3):395–413, 2008.
[7] G. Canfora, G. Casazza, and A. De Lucia. A Design Rationale Based Environment for Cooperative Maintenance. *International Journal of Software Engineering and Knowledge Engineering*, 10(5):627–645, 2000.
[8] J. Cleland-Huang, M. Mirakhorli, A. Czauderna, and M. Wieloch. Decision-Centric Traceability of Architectural Concerns. In *International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 5 – 11. IEEE, 2013.
[9] F. D. Davis, R. P. Bagozzi, and P. R. Warshaw. User Acceptance of Computer Technology: A Comparison of Two Theoretical Models. *Management Science*, 35(8):982 – 1002, 1989.
[10] J. Helming, J. David, M. Koegel, and H. Naughton. Integrating System Modeling with Project Management - A Case Study. In *33rd Annual IEEE International Computer Software and Applications Conference*, pages 571–578. IEEE, 2009.
[11] T.-M. Hesse and B. Paech. Supporting the Collaborative Development of Requirements and Architecture Documentation. In *3rd Int. Workshop on the Twin Peaks of Requirements and Architecture at RE2013*, pages 22 – 26. IEEE, 2013.
[12] A. Jansen and J. Bosch. Software Architecture as a Set of Architectural Design Decisions. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 109–120. IEEE, 2005.
[13] A. J. Ko, R. DeLine, and G. Venolia. Information Needs in Collocated Software Development Teams. In *29th International Conference on Software Engineering (ICSE'07)*, pages 344–353. IEEE, 2007.
[14] R. Likert. A Technique for the Measurement of Attitudes. *Archives of Psychology*, 22(140):1–55, 1932.
[15] R. Lougher and T. Rodden. Supporting Long-term Collaboration in Software Maintenance. In *Conference on Organizational Computing Systems - COCS '93*, pages 228–238. ACM Press, 1993.
[16] T. Ngo and G. Ruhe. Decision Support in Requirements Engineering. In *Engineering and Managing Software Requirements*, pages 267–286. Springer, 2005.
[17] B. Paech, A. Delater, and T.-M. Hesse. Integrating Project and System Knowledge Management. In G. Ruhe and C. Wohlin, editors, *Software Project Management in a Changing World*, pages 161–198. Springer, 2014.
[18] P. Runeson, M. Höst, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering. Guidelines and Examples*. Wiley, 1st edition, 2012.
[19] C. Zannier, M. Chiasson, and F. Maurer. A model of design decision making based on empirical results of interviews with software designers. *Information and Software Technology*, 49(6):637–653, 2007.