Michael Anders
Matriculation Number: 3292453

Comprehensive and Targeted Access to and Visualization of Decision Knowledge

Master Thesis

Winter Semester 19/20 - Summer Semester 20
Supervisors: Prof. Dr. Barbara Paech, Anja Kleebaum
Faculty of Mathematics and Computer Science
Ruprecht-Karls-University Heidelberg

June 5, 2020

Abstract

[Context & Motivation] Developers need to document decisions and related decision knowledge during the software development process. This ensures that future decisions can be assessed in the right context and past decisions can be retraced and understood. The documentation of decision knowledge encompasses all aspects that comprise a decision, including the problem, alternatives, arguments for and against these alternatives, and the selected solution. Since the value of the documentation is not immediately apparent, it is important to provide tools that allow easy documentation and coherent visualization of the documented knowledge. It also demands maintenance of the documentation to ensure consistency and completeness.
[Contributions] This thesis provides a problem investigation, a treatment design, and an evaluation concerning the management and visualization of documented decision knowledge. The problem investigation was done in the form of a literature review on current approaches towards the grouping of decision knowledge. The results of the review show that decision grouping is often merely a small part of larger frameworks. These frameworks either use predefined labels for their groups or allow the users to freely select group names. The practical contributions in this thesis are the extension of the Jira ConDec plug-in, which provides features for the documentation and visualization of decision knowledge within Jira. In particular, a grouping possibility for decisions as well as respective views and filters were added to this plug-in. To keep the necessary time spent on the documentation process as low as possible, it was decided to use a mix of fixed groups, in the form of different decision levels and custom groups, which the user is free to assign. New views were implemented which allow users to see relationships between source code and Jira issues and a dashboard is built, which can be used to assess the completeness of decision knowledge within a Jira project. The implementation was preceded by a specification of requirements and a design phase. Extensive testing, including system and component tests, were part of the quality assurance phase. Lastly, an evaluation was done by creating and analysing a gold standard of decision knowledge documentation and a survey with developers who provided feedback on the plug-in extension.
[Conclusions] The main focus of the thesis was to improve the visualization of relationships between knowledge elements. The evaluation showed that especially those views, creating connections between Jira elements and code classes were highly anticipated by ConDec users, as support for this form of visualization did not exist. The newly implemented features were almost uniformly evaluated positively. Some concerns were expressed about a need for even more information to be displayed within the views. This was a result of the compromise between a wealth of information and a possible overload in individual views. Evaluation of the responsiveness and time behaviour of the newly implemented features also showed that loading times were passable but require more focus in future works to improve the user experience thoroughly.

Zusammenfassung

Umfassender und zielgerichteter Zugriff auf und Visualisierung von Entscheidungswissen

[Kontext & Motivation] Entwickler müssen Entscheidungen und das damit verbundene Entscheidungswissen während des Software-Entwicklungsprozesses dokumentieren. Dadurch wird sichergestellt, dass zukünftige Entscheidungen im richtigen Kontext bewertet und vergangene Entscheidungen nachvollzogen und verstanden werden können. Die Dokumentation von Entscheidungswissen umfasst alle Aspekte, die eine Entscheidung ausmachen, einschließlich des Problems, der Alternativen, der Argumente für und gegen diese Alternativen und der gewählten Lösung. Da der Wert der Dokumentation nicht unmittelbar ersichtlich ist, ist es wichtig, Werkzeuge bereitzustellen, die eine einfache Dokumentation und kohärente Visualisierung des dokumentierten Wissens ermöglichen.
[Beiträge] Diese Abschlussarbeit bietet eine Problemanalyse, ein Behandlungsdesign und eine Bewertung hinsichtlich der Verwaltung und Visualisierung von dokumentiertem Entscheidungswissen. Die Problemanalyse wurde in Form einer Literaturrecherche zu aktuellen Ansätzen zur Gruppierung von Entscheidungswissen durchgeführt. Die Ergebnisse der Recherche zeigen, dass die Gruppierung von Entscheidungswissen oft nur ein kleiner Teil eines größeren Frameworks ist. Diese Frameworks verwenden entweder vordefinierte Bezeichnungen für ihre Gruppen oder erlauben es den Benutzern, Gruppennamen frei zu wählen. Die praktischen Beiträge in dieser Arbeit sind die Erweiterung des Jira ConDec-Plugins, das Funktionen für die Dokumentation und Visualisierung von Entscheidungswissen innerhalb von Jira bereitstellt. Insbesondere wurde dieses Plug-In um eine Gruppierungsmöglichkeit für Entscheidungen erweitert. Um den Zeitaufwand für den Dokumentationsprozess so gering wie möglich zu halten, wurde eine Mischung aus Gruppen in Form von Entscheidungs-Leveln und benutzerdefinierten Gruppen verwendet. Es wurden neue Ansichten implementiert, die es den Benutzern ermöglichen, Beziehungen zwischen Quellcode und Jira-Problemen zu erkennen, und es wurde ein Dashboard erstellt, mit dessen Hilfe die Vollständigkeit des Entscheidungswissens bewertet werden kann. Der Implementierung gingen eine Spezifikation der Anforderungen und eine Entwurfsphase voraus. Umfangreiche Tests, einschließlich System- und Komponententests, waren Teil der Qualitätssicherungsphase. Schließlich erfolgte eine Evaluierung durch die Erstellung und Analyse eines Goldstandards für die Dokumentation des Entscheidungswissens und eine Umfrage bei den Entwicklern, die Feedback zur Plug-in-Erweiterung gaben.
[Schlussfolgerungen] Die Evaluation zeigt, dass insbesondere die Ansichten, die Verbindungen zwischen Jira-Elementen und Code-Klassen herstellen, von den ConDec-Benutzern sehr positiv bewertet wurden, da es keine Unterstützung für diese Form der Visualisierung in Jira gab. Die neu implementierten Features wurden nahezu einheitlich positiv bewertet. Es wurden einige Bedenken geäußert, die sich auf einen Wunsch nach noch mehr Informationen in den Sichten richteten. Diese wurden gegen eine mögliche Überladung der Sichten abgewogen. Die Evaluierung des Zeitverhaltens der neu implementierten Funktionen zeigte, dass die Ladezeiten passabel sind, aber bei künftigen Arbeiten mehr Aufmerksamkeit erfordern, um das Nutzererlebnis gründlich zu verbessern.

Contents

1 Introduction

This chapter presents the motivation (Section 1.1) and overall goals (Section 1.2) of this thesis. Additionally, it will outline the structure of the thesis (1.3).

1.1 Motivation

Developers make decisions during every step of the development process. These decisions concern requirements, architectures, implementations, quality assurance and every other aspect of the engineering cycle. While these decisions are often intrinsic or even subconscious, they can have wide-ranging effects on the resulting software and its future maintainability as well as other quality aspects [6]. As such, it is of great importance to document the existing decision knowledge within a project to guarantee future understanding of the results, consequences and alternatives that every decision inherits [27][30].
The more a software project is developed over time, the greater the spread of information across different tools. For example while requirements are going to be documented in an Issue-Tracking-System (ITS) like Jira[1], the code itself is handled by a Version-Control-System (VCS) like Git[2]. To combat the spread of information, tools need to be created, which support information compilation without being a burden on the developers. As with any task, providing users with adequate tools to ease the burden of documentation and thus lower its inherent cost, can greatly increase the quantity and quality of the documentation. Decisions are no different in that matter [36]. Allowing developers to easily document their decisions in whichever way they find appropriate and convenient, makes that far more likely. However, to guarantee a future understanding of the documentation, some guidelines and documentation rules have to be created. These guidelines and rules allow automated tools to properly manage the data. They also make it easier for developers to understand each other's documentation. The result is a balancing act between presenting developers a low effort, low-cost method of documenting their decisions, while also maintaining a standard of quality for the resulting documents. Thus, the goals of a rationale management tool are to minimize the intrusiveness of the documentation process while maximizing the documentation quality.
Documenting decisions carries an additional problem. Any information that does not offer an immediate benefit is often neglected and not properly documented, because it appears to lack importance to the individual. It is only in future development that the real value of properly documented decisions appears. Trying to comprehend decisions

[1]Jira: https://www.atlassian.com/software/jira (Last access: 05.06.2020)
[2]Github: https://github.com/ (Last access: 05.06.2020)

and their effects, after the fact, can be a complex process. This is especially the case if information has been lost over time. This problem comes back to presenting the easiest way possible for developers to document their decisions so as not to discourage them from the process.
Even under perfect circumstances, with immaculate documentation, evaluating the effects of decisions towards the software or even towards each other can be challenging [15][40]. This difficulty is exacerbated as more decisions are made, as the interconnectivity between them increases. As such, users need to be supported in understanding the decisions and their effects as much as possible. Offering different visualizations becomes necessary in larger projects to make the knowledge comprehensible. However, in order to offer these visualizations a certain amount of consistency needs to be present within the documentation. Because of this need, the role of the rationale manager is introduced. Tasked with maintaining the consistency and completeness of the documented decision knowledge within software projects, this role becomes more and more important the larger a project becomes. Because the rationale manager is tasked with maintaining this consistency and overseeing the quality and coverage of the decision documentation process, they need to be provided with tools that allow them to successfully and efficiently complete this job.
The amount of decision knowledge grows during the development cycle of a software. As such features need to be provided, that allow the management, sorting and filtering of the knowledge. Developers need to be able to quickly and effectively find related decision knowledge, without being overwhelmed by the sheer amount of existing elements. Sorting the decisions into relevant groups, according to features they support, the effects they have or any other measure, allows more targeted access to relevant decisions at later dates. As such, tool support should also include such a grouping feature to aid developers in their effective usage of decision knowledge.

1.2 Goals

Broadly stated, the task of this thesis is to implement new and refactor existing methods and tools that aid developers as well as rationale managers in the creation, maintenance, and analysis of decision knowledge within a software project. Thus, this thesis adds on to the existing Jira ConDec[3] plug-in which presents decision knowledge from different documentation locations like Jira issues, comments, Git commits, pull requests, etc. in relation to other software artifacts. The collected knowledge builds up a knowledge graph and is then visualized according to users' needs.
For the rationale manager a dashboard is to be added which allows a quicker overview of the completeness and distribution of the decision knowledge across each project. Different metrics are shown here, which give indications about the status of the knowledge within the project. From here it is made possible to easily navigate to issues of interest, where consistency can be verified more precisely.
Developers gain access to views that visualize the connection between documented decisions and the code itself, as that relationship is often not easily apparent. This offers an

[3]ConDec on Git: https://github.com/cures-hub/cures-condec-jira (Last access: 05.06.2020)

easier understanding of the effects different decisions have and can simplify the process of implementing new decisions within the code.
Additionally, a detailed literature review is conducted. It is aimed at finding a possible grouping scheme for documented decisions. The groups in turn highlight relationships between the decisions and can offer insights into their importance. The results from the review are implemented into the plug-in and additional views are presented to offer management of this new feature.
All implemented views are equipped with relevant filters, which allow further customization of the views, in order to access more targeted knowledge. More precise filtering in general is an important goal of the thesis. Focus is put on maintaining a uniform filtering experience across the platform. This generates a more easily accessible and user-friendly environment, where information can be accessed and recorded efficiently.
The resulting plug-in is then tested and evaluated according to certain acceptance criteria. To ensure a satisfying level of usability, usefulness, and difficulty of use, fellow ConDec-developers are tasked with scenarios aimed at each aspect of the implemented features. Relevant changes and improvements are implemented or discussed in detail.
The resulting plug-in should be an answer to some of the problems and challenges discussed in the previous Section 1.1.

1.3 Overview

This thesis starts by introducing some fundamental techniques and tools in Chapter 2. These provide the ground works for the research and implementations provided by this thesis. The main research part is described in Chapter 3, in which a literature review was conducted to provide answers for a number of predefined research questions. Next, the requirements for the extensions of the ConDec plug-in are provided in Chapter 4. The design and implementation phase of the extension is described in Chapter 5. The chapter also provides an overview of the most important decisions during the development process. Following this, the Quality Assurance Chapter 6 presents the necessary tests to ensure proper function of the extended features. After this, Chapter 7 introduces the evaluation, including criteria, evaluation survey, and evaluation on real data, the results of which are discussed in detail. Lastly, Chapter 8 summarizes the work and provides an outlook on possible future improvements to the ConDec project.

2 Fundamentals

This chapter explains fundamental aspects and concepts used in this thesis. Section 2.1 defines the Continuous Software Development process. Afterwards Section 2.2 explains the concept of rationale management. The following Section 2.3 introduces the models used for documentation of decision knowledge. Section 2.4 then gives a short overview over the ConDec project and its capabilities. Lastly, Section 2.5 explains the concept of tagging and its connection to this thesis.

2.1 Continuous Software Development

Fitzgerald et al. [20] discuss the underlying disconnect between parts of the development process, namely documentation and implementation. Due to the often short nature of execution of these tasks, this disconnect is only exacerbated. Deadlines, expectations, customer needs, and a desire to be reactive to ever-changing requirements are part of the modern software industry. This however creates an environment in which parts of the development process, that do not directly produce code, are often discouraged. Their value is questioned in these short term production cycles as developers struggle to see a direct benefit.
In [35], it is discussed how software has to permanently adapt to an ever-changing environment. Continuous Software Development is an agile process within software engineering. Within the process, development teams are set up in a way that allows them to quickly and precisely react to these changing environments. They aim for fast deployment of their software product in regular cycles. For the fast deployment to function properly however, the quality of the software has to be on a high standard with a constantly deployable software iteration. This is were the aforementioned disconnect between documentation and implementation becomes a problem. Without proper documentation it becomes hard, if not impossible, to continuously develop software to a high standard. There might simply not be enough information available to comprehend the code of previous versions. The time needed to analyse every piece of legacy code is simply not available in a fast-paced development cycle. The same problem exists for decision documentation and management. Thus, continuous software development has to find a compromise between rapid code production and thorough documentation to guarantee that future development cycles can keep up with the high demand.
In [19], Fitzgerald et al. shift the focus away from merely focusing on continuous integration, towards a model which integrates every part of the development process. Not just the integration and deployment need to be continuous, but rather the whole process. This also includes continuous testing, continuous verification, continuous run-time

monitoring and, in case of this thesis, continuous documentation. Figure 2.1 introduces all these continuous steps in the context of the innovation process as a whole.

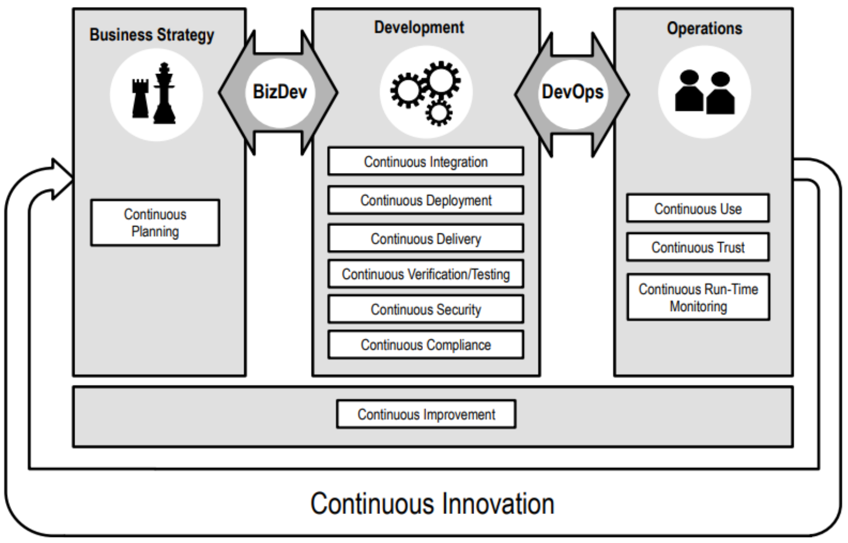


Fig. 2.1: Activities from Business, Development, Operations and Innovation [19]

2.2 Rationale Management

A decision encompasses more aspects than just the course of action that was chosen. The issue itself, alternative courses of action, arguments for and against each aspect, and the chosen course are all part of the knowledge created during the decision making process. All these individual parts together form a rationale [16] [5]. Alternatively, this collection is also called decision knowledge and is referred to in either way in this thesis. Proper documentation requires completeness of the decision knowledge, as only an encompassing overview over all aspects of a decision allows informed decisions to be made [31].
The manual and tool-supported processing and creation of decision knowledge is in turn referred to as rationale management. It is what allows developers to document and later on analyse the information effectively. The role of rationale manager, previously mentioned in Section 1.1, carries the main responsibility in ensuring the consistency of the created decision knowledge.

2.3 Decision Documentation Models

As previously discussed, decision knowledge is often neglected when writing documentation for a software as its benefit is not an immediate one and often hard to foresee [36]. Supporting developers in the documentation process increases the likelihood of frequent, high-quality documentation. Introducing models that clearly define the documentation process reduces the cost of later interpretation and understanding and thus may increase the likelihood of proper documentation. The Issue-Based Information System (IBIS) introduced by Kunz et al. [34] is one such model. It encompasses three different knowledge types, as can be seen in Figure 2.2.

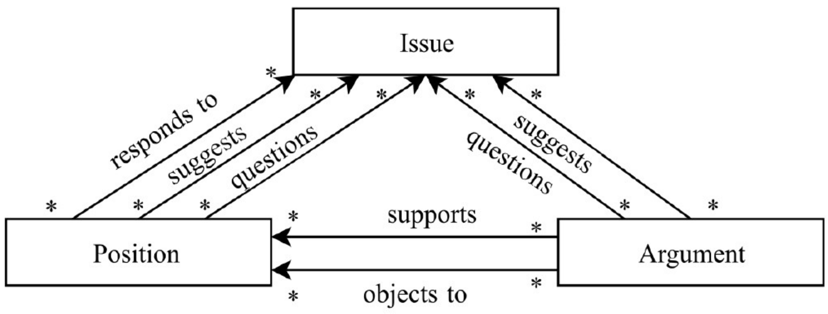


Fig. 2.2: IBIS Model [34]

Issues are defined as the underlying decision problem for which answers are to be found. Next, the *Position* is an opinion towards the connected decision problem. Lastly, *Argument* defines either a pro or con reasoning towards an *Issue* or a *Position*.
The model used by Kleebaum et al. [31] for the ConDec project is based on a more complex model by Hesse et al. [24]. The Hesse model introduces a lot more knowledge elements than the three used by the IBIS model. Figure 2.3 gives an overview of the seventeen elements that comprise a knowledge element in their model. It adds participants, in the form of persons and their roles, to the model. It also expands from mere arguments and positions to more detailed, interconnected components. The very detailed model however carries the problem of its high complexity in real use case scenarios.

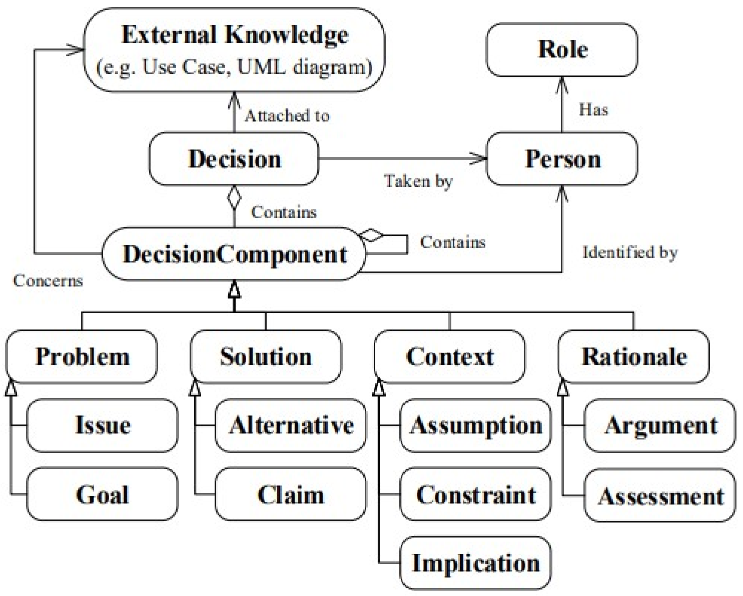


Fig. 2.3: Decision Knowledge Documentation Model according to Hesse et al.[23]

The simplified model by Kleebaum et al. [31], cuts the different elements, within a decision, down to five, settling on a model consisting of the five elements *issue, alternative, pro, con* and *decision*. Table 2.1 lists the model elements and their descriptions along with possible phrasings which can indicate the element.

Table 2.1: Knowledge Elements according to Kleebaum et al. [31]

Element	**Description**	**Indicating Phrase**
Issue	Describes the problem that is to be solved	*How should...*
Alternative	An option for solving the decision problem	*One option is...*
Pro	An argument for the connected alternative	*The advantage is...*
Con	An argument against the connected alternative	*The disadvantage is...*
Decision	The option chosen based on the give arguments	*The best option is...*

2.4 ConDec

The CURES[1] project is a joined project between the Technical University Munich and the University of Heidelberg. It focuses on support for Continuous Software Develop-

[1]CURES: http://www.dfg-spp1593.de/cures/index.html(Last access: 05.06.2020)

ment by providing tools to handle, among other things, decision knowledge management functionalities. The implemented requirements form the tools, which are collectively named ConDec [32], for Continuous decision knowledge Management. ConDec uses the documentation model previously discussed in Section 2.3.
The idea behind the ConDec Jira plug-in, which is extended upon in this thesis, is to provide views and management capabilities for rationale management within the Issue-Tracking-System (ITS) Jira. Traditionally ITS's like Jira contain very little to no capabilities to create and review decision knowledge. By creating a user-focused plug-in that is capable of providing these functionalities, developers are encouraged to pursue proper documentation standards. The main underlying data structure for the documented decision knowledge comes in the form of a knowledge tree, similar to the one introduced by Hesse et al. [24]. Knowledge is stored and displayed in these trees (Fig. 2.4), which allows interconnection between decisions and their elements. All knowledge elements and their connecting links together form the knowledge graph. Within this graph the elements form the nodes and the links the edges of the structure. ConDec puts great focus on filtering capabilities for all implemented views. This allows for easier access to targeted knowledge and guarantees functionality even for larger, ongoing projects. The focus on continuous development tackles the challenges discussed in Section 1.1 and makes the rationale management capabilities applicable to modern software engineering projects.

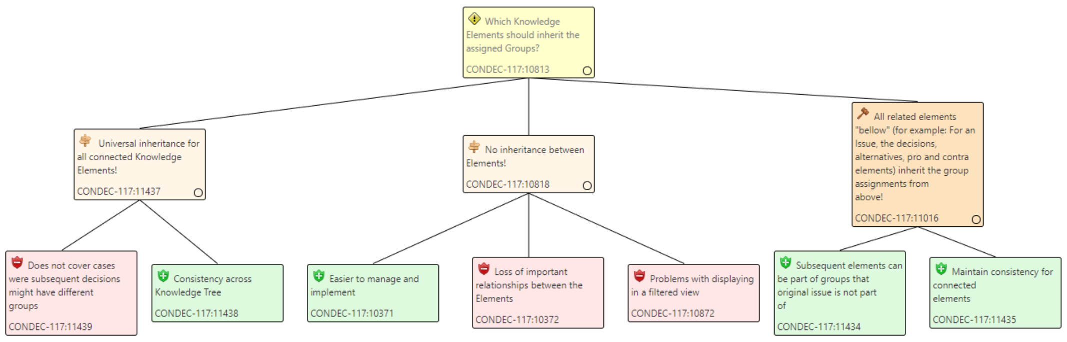
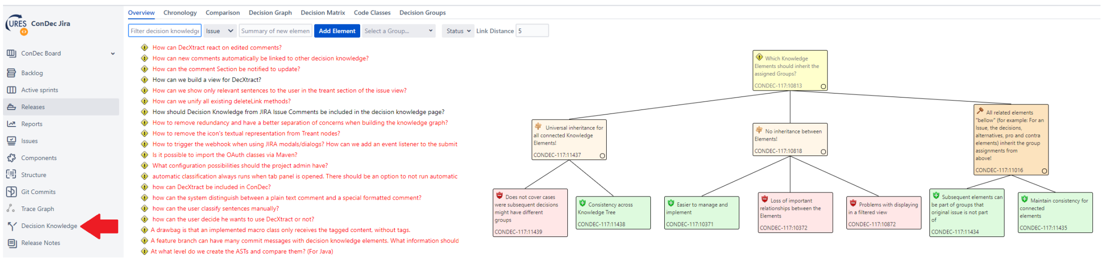

Which Knowledge Elements should inherit the assigned Groups?
CONDEC-117:10813
Universal inheritance for all connected Knowledge Elements!
CONDEC-117:11437
No inheritance between Elements!
CONDEC-117:10818
All related elements "bellow" (for example: For an Issue, the decisions, alternatives, pro and contra elements) inherit the group assignments from above!
CONDEC-117:11016
Does not cover cases were subsequent decisions might have different groups
CONDEC-117:11439
Consistency across Knowledge Tree
CONDEC-117:11438
Easier to manage and implement
CONDEC-117:10371
Loss of important relationships between the Elements
CONDEC-117:10372
Problems with displaying in a filtered view
CONDEC-117:10872
Subsequent elements can be part of groups that original issue is not part of
CONDEC-117:11434
Maintain consistency for connected elements
CONDEC-117:11435

Fig. 2.4: Decision Knowledge Tree

The knowledge visualization is supported by multiple views, including Jira issue modules, dashboards, reports, and graph visualizations. Figure 2.5 shows an example of the graph visualizations. It displays the connections between related decision knowledge artifacts in the *Decision Knowledge Overview* (WS3.1). The left section of the view presents a list of all documented decision issues but can also be changed to a list of any other decision artifact. The right section displays the immediate decision tree of the selected element, displaying all related artifacts. The view presents additional filter elements including text search, link distance filters, and the newly added group filter.
ConDec offers different capabilities of documenting decision knowledge. They differ by their documentation location. Users can create separate Jira issue of the type of decision element they want, be that issue, decision, alternative, pro or con. This possibility carries the advantage of offering the full capabilities that other Jira issues have, including attachments, descriptions, comments and the Jira issue linking system. The effort of documentation however is higher. As such, ConDec offers the documentation in other

locations such as Jira issue descriptions, comments, commits and code comments by using an annotation for the desired decision element. Connections between the documented elements are created automatically and can easily be changed by the user using simple drag and drop functionalities on the decision tree. These documented decisions are handled using the Active Object functionalities provided by Jira, which allows the processing and storing of databases containing the elements, links and all other related artifacts in accessible tables.
Source code for the ConDec project is open source and available through GitHub [2].



- How can DecXtract react on edited comments?
- How can new comments automatically be linked to other decision knowledge?
- How can the comment Section be notified to update?
- How can we build a view for DecXtract?
- How can we show only relevant sentences to the user in the treant section of the issue view?
- How can we unify all existing deleteLink methods?
- How should Decision Knowledge from JIRA Issue Comments be included in the decision knowledge page?
- How to remove redundancy and have a better separation of concerns when building the knowledge graph?
- How to remove the icon's textual representation from Treant nodes?
- How to trigger the webhook when using JIRA modals/dialogs? How can we add an event listener to the submit
- Is it possible to import the OAuth classes via Maven?
- What configuration possibilities should the project admin have?
- automatic classification always runs when tab panel is opened. There should be an option to not run automatic
- how can DecXtract be included in ConDec?
- how can the system distinguish between a plain text comment and a special formatted comment?
- how can the user classify sentences manually?
- how can the user decide he wants to use DecXtract or not?
- A drawbag is that an implemented macro class only receives the tagged content, without tags.
- A feature branch can have many commit messages with decision knowledge elements. What information should
- At what level do we create the ASTs and compare them? (For Java)

Which Knowledge Elements should inherit the assigned Groups? CONDEC-117:10813

Universal inheritance for all connected Knowledge Elements! CONDEC-117:11437

No inheritance between Elements! CONDEC-117:10818

All related elements "bellow" (for example: For an Issue, the decisions, alternatives, pro and contra elements) inherit the group assignments from above! CONDEC-117:11016

Does not cover cases were subsequent decisions might have different groups CONDEC-117:11439

Consistency across Knowledge Tree CONDEC-117:11438

Easier to manage and implement CONDEC-117:10371

Loss of important relationships between the Elements CONDEC-117:10372

Problems with displaying in a filtered view CONDEC-117:10872

Subsequent elements can be part of groups that original issue is not part of CONDEC-117:11434

Maintain consistency for connected elements CONDEC-117:11435

Fig. 2.5: Decision Knowledge Overview (WS3.1)

2.5 Tagging

Among other features, this thesis introduces a grouping functionality for decision knowledge elements. This is done by annotating (tagging) the different elements within the knowledge tree with a group in the form of a natural language identifier. The general idea behind using tags on software development artifacts is to provide additional information and express relationships between these elements. This approach is used in many different application scenarios.
For example, Seiler et al. [46], use feature tags on software artifacts in both Issue-Tracking- and Version-Control-Systems to reduce the spread of information and create deeper relationships and traceability between these two systems. By integrating tool support in both, with the addition of a recommendation system, the tagging is further simplified for developers.
Gupta et al. [21] discuss part-of-speech (POS) tagging for program identifiers. They propose tagging as a means to improve automated tool accuracy, like software search tools. By comparing multiple automated "*traditional*" POS taggers they propose improvements to increase the accuracy of the tagging process.
Additionally, the Literature Review of this thesis (Chapter 3) discusses multiple different approaches for grouping schemes using traditional tags and even some approaches deviating from the traditional natural language tags.

[2]ConDec on GitHub: https://github.com/cures-hub/cures-condec-jira (Last access: 05.06.2020)

3 Literature Review

This chapter answers a set of research questions, which were designed to aid in the implementation of this thesis. Section 3.1 introduces the predefined research questions. Section 3.2 shows the methodology used to find relevant works and which criteria were used to prune search results. Section 3.3 then introduces the found works in detail before the results are compared in Section 3.4. Lastly, Section 3.5 summarizes the chapter and draws conclusions for this thesis from the extracted papers.

3.1 Research Questions

The main research question RQ1 tries to find efficient ways to create a grouping scheme for documented decisions. This scheme can then be used to provide developers with more targeted access to the decision knowledge in a project. With the categorization of groups, developers can narrow their search for prior decisions to a more relevant set. In order to further specify the research questions several sub-questions were introduced. The questions can be seen in Table 3.1.

Table 3.1: Research Questions for the Literature Review

Name	Research Question
RQ1	**How is the grouping of decision knowledge realised and used?**
RSQ1	What grouping schemes are employed to categorize decisions?
RSQ2	How are groups of decisions documented?
RSQ3	How are these groups of decisions implemented ?
RSQ4	How is filtering implemented and used in the presented techniques ?

3.2 Methodology

3.2.1 Criteria of Relevance

To extract those works, that are relevant to the researched topic discussed in Section 3.1 criteria of relevance are introduced (CoR4, CoR5, CoR6, CoR7). Additionally, these criteria need to make sure that the approaches presented are properly researched and scientifically sound (CoR3). Their accessibility (CoR2) and comprehensibility (CoR1) needs to be guaranteed as well. As a result the criteria listed in Table 3.2 were developed.

Table 3.2: Criteria of Relevance

Designation	Description
CoR1	The paper is written in either English or German
CoR2	The paper needs to be available for free or through the University's access
CoR3	The paper has been peer reviewed
CoR4	The paper is related to the software development domain
CoR5	The title shows relevance to the researched topic
CoR6	The presented approach needs to introduce a form of grouping for documented decisions
CoR7	The paper offers sufficient content towards the relevant research questions

CoR1 - CoR5 allow a quick evaluation of the topical relevance and the accessibility of the found papers and as such were used to classify papers as *initially relevant*. Especially CoR5 was important in assessing whether a paper had to be considered for further evaluation. If the relevant search terms or related terms were not mentioned in the title, it could be assumed that the paper would also not correspond to CoR6 and CoR7. Only those works deemed to be *initially relevant* were then checked for CoR6 and CoR7 as these required a deeper look into the paper.

3.2.2 Approach

To guarantee an overview over the current state of research both a search term based and a snowballing based approach were employed during the review process.
First the search term based approach was used on the digital libraries of both the Institute of Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM). The search terms were designed to extract the relevant papers from both databases while also removing as many unrelated papers as possible. These could otherwise dilute the search results and thus risk overshadowing more related works. Because the nomenclature in this field of research is quiet broad, a total of six different search terms had to be created to allow proper coverage over the existing works. This however led to some overlapping in the created results. Additionally, *decision documentation* as a concept is not restricted to the software development domain. As such, the more specified term *architectural decision* was used in some of the searches. This specification proved to be quite important as a significant amount of research on the relevant topic focuses on these architectural decisions. In the end the following search terms yielded proper results from the search engines:

- ST1: All: "Decision Knowledge" AND All: "Grouping" AND All: "Managing"
- ST2: All: "Framework" AND All: "Architectural Decisions" AND All: "Documentation"
- ST3: All: "Framework" AND All: "Decision Knowledge" AND All: "Documentation"
- ST4: All: "Architectural Decision Knowledge" AND All: "Framework" AND All: "Software"
- ST5: Publication Title: "Framework" AND Publication Title: "Architectural" AND Publication Title: "Decision"
- ST6: All: "Software Development" AND Publication Title: "Decision" AND Abstract: "Documentation" AND Abstract: "Framework"

The use of the very general term *Framework* within most of the search terms is due to the fact that almost no papers exist, which focus solely on any form of grouping for decision knowledge. All papers found use grouping in some bigger concept. Most of them introduce a framework for decision documentation, which in turn has a grouping scheme or tag as part of that framework. Searching for any form of grouping scheme directly, rather than a more general framework, did not yield as many results as were necessary to answer the research questions. This may in part be because there is no specific naming convention for the process of grouping, as it is called in this thesis. Very few papers use the same nomenclature for the consolidation of decisions into different groups. Using *Framework* had the adverse effect of yielding more irrelevant papers, than more targeted searches. However, it allowed the finding of works, which do not follow the specific naming convention.
Table 3.3 shows the library used, found results and number of initially relevant works for each term with the duplicates removed. As mentioned before, papers were considered initially relevant if the met CoR1 - CoR5 as these could be evaluated more quickly than the remaining two criteria of relevance.

Table 3.3: Search Term Based Research results

Library	Term	Results	Initially Relevant	References
IEEE	ST1	13	1	[51]
IEEE	ST2	18	5	[37], [10], [9], [14], [17]
IEEE	ST3	41	2	[29], [2]
IEEE	ST4	47	4	[43], [44], [18], [47]
ACM	ST5	4	4	[42], [4], [8], [28]
ACM	ST6	36	2	[11], [1]

Multiple problems related to the search term based approach for the underlying research questions need to be discussed. Firstly, because of differences in the basic search engines of IEEE and ACM the used search terms had to be adapted uniquely for both platforms. Without specification for publication titles or abstracts on ACM the amount of results, which were in the thousands, was greater than could be evaluated for relevance. This problem was not present on IEEE. However, IEEE offered a different hindrance. University access for IEEE was lost before the review could take place and as such some found papers that were not accessible through other, free sources or did not show enough relevance in title and abstract, could not be fully evaluated. This meant that these works were lost in the process. Lastly, the search term based approach as such offered a problem. Unfortunately, no papers could be found that focused on the grouping of decisions. Most rather used them as part of a bigger framework or only loosely introduced them at all. This meant that a term based search did not offer the promised results and of the, initially relevant papers only three where deemed fully relevant.
These three papers allowed for the second phase, the snowballing process, to start. Snowballing is the technique of checking both the references listed by the given papers (Backward Snowballing), as well as the works that list the given papers as a reference (Forward Snowballing) [50]. This can occur in multiple iterations, in that the papers found during the snowballing process have their citations and references examined as well. IEEE, ACM and multiple other digital libraries all offer support for this technique, making the process significantly easier and improving the possibility of finding relevant works. Table 3.4 lists all papers found to be relevant in addition to their discovery process. In the case of Snowballing the last column designates the paper from which the work was found. Special note should be taken of the work by Shahin et al. [47]. Their paper collected a multitude of works related to the topic at hand and thus allowed a number relevant papers to be found, which all contained a form of grouping for decisions. All relevant papers extracted from Shahin et al.'s work are discussed individually in the review. That is why the paper itself is not discussed in more detail within the chapter.

Table 3.4: Results of Search Term Based Approach and Snowballing

Author	Reference	Title	Process	Snowb. Source
Bhat et al.	[3]	Automatic Extraction of design decisions from Issue Management Systems: A Machine Learning Based Approach	B. Snowballing	[4]
Bhat et al.(2)	[4]	Towards a framework for managing architectural design decisions	Search Term	
Capilla et al.	[7]	Modeling and Documenting the Evolution of Architectural Design Decisions	B. Snowballing	[47]
Carriere et al.	[8]	A Cost-Benefit Framework for Making Architectural Decisions in a Business Context	Search Term	
Dermevall et al.	[13]	STREAM-ADD - Supporting the Documentation of Architectural Design Decisions in an Architecture Derivation Process	F. Snowballing	[47]
van Heesch et al.	[22]	A documentation framework for architecture decisions	F. Snowballing	[47]
Jansen et al.	[26]	Architectural design decisions	B. Snowballing	[3]
Kruchten et al.	[33]	An Ontology of Architectural Design Decisions in Software-Intensive Systems	B. Snowballing	[47]
Tyree et al.	[48]	Architecture Decisions: Demystifying Architecture	B. Snowballing	[47]
Shahin et al.	[47]	Architectural Design Decision: Existing Models and Tools	Search Term	
van der Ven et al.	[49]	Making the Right Decision: Supporting Architects with Design Decision Data	B. Snowballing	[3]
Zimmermann et al.	[52]	Reusable Architectural Decision Models for Enterprise Application Development	B. Snowballing	[53]
Zimmermann et al. (2)	[53]	Managing architectural decision models with dependency relations, integrity constraints, and production rules	F. Snowballing	[48]

3.3 Review Results

One of the most influential papers on the matter of grouping of decision knowledge is the work published by Kruchten et al. [33]. This can be measured by the level of discussion and citation in other relevant papers. Being referenced and positively discussed in most the relevant works, Kruchten et al. introduce an ontology for architectural design decisions, focusing on their attributes and relationships with one another. In their model, decisions are turned into a first class entity, meaning that they are not subsumed by other design artifacts but rather stand on their own. To create these first class entities they introduce different classifications of design decisions, suggest sets of attributes and discuss possible relationships. The introduced groups show to be some of the most promising for later implementation in this thesis. In total they identify three different major groups of design decisions with additional subgroups.

- Existence decisions (ontocrises)
    - Structural decisions

    - Behavioral decisions
    - Ban / Non-Existence decisions (anticrises)
- Property decisions (diacrises)
- Executive decisions (pericrises)

Existence decisions concern those decisions that indicate some element or artifact will exist in the system due to this decision. They can be divided into three subgroups. Structural decisions are those that lead to the implementation of new components, subsystems, partitions etc. Behavioral decisions however indicate more closely how elements interact with one another and how they provide functionality or satisfy non functional requirements. Lastly, Ban decisions or Non-Existence decisions are the opposite of normal existence decisions in that they state that a certain element will not appear in the system. As such they are still categorized as a subgroup of existence decisions. Kruchten et al. note that existence decisions are the least important to capture as they are the most visible in a finished system. They however argue that it is still important to document them, as they can later be used to relate to other, less obvious decisions or alternatives.
Property decisions contain traits or quality statements of the designed system. They consider things such as design rules or general guidelines as well as design constraints. They can be subdivided between positively expressed rules and guidelines and negatively expressed constraints, this however is not expressly done by Kruchten et al. Additionally, Kruchten et al. argue that these groups of decisions can be hard to trace to specific elements because they are often cross-cutting concerns and effect too many distinct elements. Without proper documentation the decision knowledge for these types of decision stays implicit and often further design decisions are made without being traced back to appropriate properties.
Executive Decisions are those decisions that do not directly relate to any design elements or their underlying qualities. They are driven by the business environment, the development process, education and training of staff, organizational concerns and the choices of technologies or tools that are used. Kruchten et al. argue that especially these decisions are often left out of the development process because of their disconnect from the software development itself. But because of the huge implications and constraints created by these decisions their documentation seems all the more important to the authors. This is especially the case because these executive decisions often constrain existence and property decisions.
Next to the kinds of decisions introduced, Kruchten et al. also show possible Attributes in their model. Attributes such as Epitome, Rational, Scope, State, History, Cost and Risk. They also introduce a category attribute, which unlike the existence, property and executive decision groups does not have pre-existing tags. The idea behind the category is to introduce an open ended list of text-labels with which queries can be used to trace and find related decisions. They name examples such as Usability, Security and Politics as possible tags.
The relationships introduced by the model concern relationships between decisions as well as with external artifacts and aim to create inter-connectivity between design ele-

ments. They contain relationships such as constrains, forbids, enables, subsumes etc. While the model introduced by Kruchten et al. seems purely theoretical, with no studies or implementations being shown, the underlying research shows a lot of promise to allow proper documentation of decisions. Additionally, this paper is later often referenced by other works which will be discussed in this chapter.
Van Heesch et al. [22] present a decision documentation framework which is derived from the conventions of ISO/IEC/IEEE 42010. This model consists of four viewpoints designed to address general documentation concerns. The *Decision Detail Viewpoint* addresses those concerns related to the rationale behind the decisions. The *Decision Relationship Viewpoint* focuses on the relationships between decisions. Next, the *Decision Stakeholder Involvement Viewpoint* allows proper explication between Decisions and Stakeholders and lastly the *Decision Chronological Viewpoint* was developed to address temporal concerns in the decision process. Fig 3.1 presents the complete architectural framework created by Heesch et al.

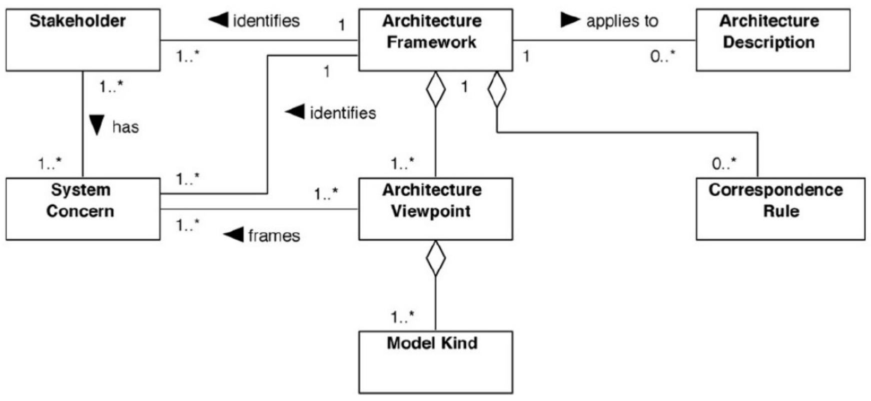


Fig. 3.1: Architecture Framework reproduced from ISO/IEC/IEEE 42010 [22]

As part of the Decision Detail Viewpoint Heesch et al. introduce description elements. Most importantly for this thesis they introduce a *Decision Group* element. They state, that in their model decisions can be part of one or multiple groups and state grouping examples. Their ideas include grouping by architecture teams, where the team making the decision is used as the respective group. They also suggest quality attributes as possible groups, stating that their Decision Group element is equal to the categories introduced by Kruchten et al. [33], without the explicit partitioning seen in Kruchten et al.'s work.
The work presented by Tyree et al. [48] aims to simplify or "demystify" the decision documentation process. They claim that some of the existing models and frameworks are too complex to allow easy and simple documentation. This then leads developers to determine that decision documentation is not worth the effort. In their opinion a simple document describing key architecture decisions is sufficient in helping to understand past and future system architectures. Deriving a template from IBM's e-Business Reference Architecture Framework, they end up with a list of necessary attributes. Table 3.5 shows an adapted, excerpt list of attributes and their explanations.

Table 3.5: Architecture decision description template [48]

Attribute	**Explanation**
Issue	Describe the architectural design issue you're addressing, leaving no questions about why you're addressing this issue now.
Decision	Clearly state the architecture's direction.
Status	The decision's status, such as pending, decided, or approved.
Group	You can use a simple grouping—such as integration, presentation, data, and so on—to help organize the set of decisions. You could also use a more sophisticated architecture ontology, which includes more abstract categories such as event, calendar, and location.
Assumptions	Clearly describe the underlying assumptions in the environment in which you're making the decision—cost, schedule, technology, and so on.
Argument	Outline why you selected a position, including items such as implementation cost and required development resources' availability.
Related decisions	It's obvious that many decisions are related; you can list them here. Metamodels are useful for showing complex relationships diagrammatically.
Related requirements	To show accountability, explicitly map your decisions to the objectives or requirements. You can enumerate these related requirements here.
Notes	Because the decision-making process can take weeks, we've found it useful to capture notes and issues that the team discusses.

For Tyree et al. the idea behind the grouping attribute is to allow for filtering based on the stakeholders' interests. As an example they use data architects reviewing prior decisions by being able to filter by a *data* group. Additionally, they mention a color-coding idea where incomplete or controversial decisions could be highlighted by distinct colors. This could in abstractions also be considered as a grouping mechanism where instead of types of decision the status of the decision is considered as a group.
Capilla et al. [7] describe the **A**rchitecture **D**esign **D**ecision **S**upport **S**ystem (ADDSS), a web-based tool aimed at recording and managing architectural design decisions. The idea behind ADDSS is to store requirements of a target system, the decisions made for requirements sets, and JPEG images that represent the architectural products generated in the architecting phase. The ADDSS tool generates PDF documents that describe decisions, architectural products, and trace links. In the tool Capilla et al. make the distinction between Mandatory attributes for the architectural design decisions and Optional attributes. The group or as they call it "*Category of Decision*" attribute is assigned to the optional attributes in their model. Similar to Tyree et al., they propose the group attribute as a way to support discovery processes. Additionally, they propose that a variability management tool for example could in theory visualize different system configurations depending on the groups in which the decisions are split. They however do not propose any concrete groups or types of decisions but rather see the possible groups as part of the application domain. Their work is still important because their approach is actually implemented in the ADDSS tool rather than being a theoretical framework as is the case with most papers discussed in this chapter.

Bhat et al. [3] introduce an approach focused on a more automatic extraction of design decisions from Issue Management Systems (IMS) such as Jira. The two-phased, machine-learning-based approach first detects design decisions from the issues in the system and then in the second step classifies these into different groups. The groups introduced are those discussed by Kruchten et al. [33] (see above), with the idea in mind to create a knowledge base from which developers can learn from the decisions in similar, past projects. Figure 3.2 again shows the used categories.

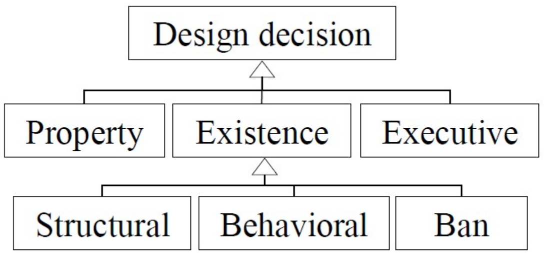


Fig. 3.2: Architecture Design Decision Groups [3]

Because of the necessity for Bhat et al. to manually annotate issues to create a dataset and because the majority of design decisions are existence decision [39], they focus their approach on the classification into the *existence* decision group with its three subgroups. According to them, the approach could however be extended by creating a labeled dataset for the remaining groups. Additionally, Bhat et al. present a framework for architecture knowledge management (AKM) [4], which contains the same categories as their automatic classification approach. They introduce a machine learning pipeline for this classification approach. The results of their testing seem promising in allowing for the implicit decision knowledge of issues to be extracted and explicitly captured in existing decision groups.
The previously mentioned framework introduced by Bhat et al. in [4], focuses on addressing different use cases. Among other components like an architectural element annotator, an architectural solutions recommender, and a rational extractor, the framework also contains a design classifier like the one discussed here. They also discuss the possibility of changing their classification approach to use different groups.
One of those possible other group strategies is the one introduced by Jansen et al. [26]. While not intended specifically as a grouping strategy for decisions, Jansen et al. introduce a hierarchy based on the abstraction level of decision making. Figure 3.3 shows the three dimensions of decision making.

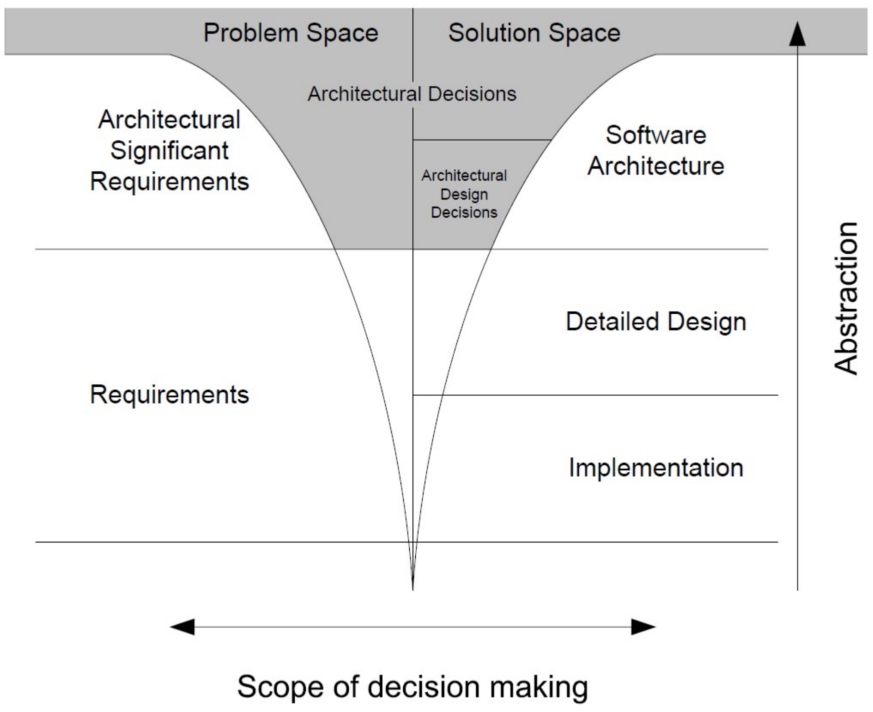


Fig. 3.3: The funnel of decision making [26]

The first distinction is the one between the problem space on the left and the solution space on the right. While the problem space contains all problems a system could address, the solution space contains all the possible system solutions.
Next, the level of abstraction forms the second dimension. In both spaces, problem and solution, the decisions can be of differing abstraction levels which are visualized on the left and right side of Figure 3.3. Lastly, the scope of the decision is what creates the funnel. The more abstract the decision is, the wider the scope becomes.
As mentioned before, this funnel model is not strictly aimed at creating groups of decisions but rather describes the decision making process. One could however easily see how the second dimension could be used to group decisions based on their abstraction level. The possible labels for those groups could then be the different stages of abstraction, namely, *Requirements, Implementation, Detailed Design, Software Architecture, and Architectural Significant Requirements* for example. This approach would allow the developer to sort Decisions into the development step they influence the most and could like other approaches presented in this chapter be used to investigate prior decisions and thus help influence similar future decisions.
Dermeval et al. [13] introduce STREAM-ADD, for **S**trategy for **T**ransition between **R**equirements and **A**rchitectural **M**odels with **A**rchitectural **D**ecisions **D**ocumentation. STREAM itself is a model-driven approach to generate architecture models from existing requirements models. The contribution of Dermeval et al. [13] is the addition of decision documentation support to the STREAM process. The idea behind the systematic documentation of decision is to aid in the refinement of architecture models based on prior decisions. Figure 3.4 shows an overview of the STREAM-ADD process introduced by the paper. The first two activities are maintained from the original STREAM

process with the third one being the addition made by Dermeval et al.

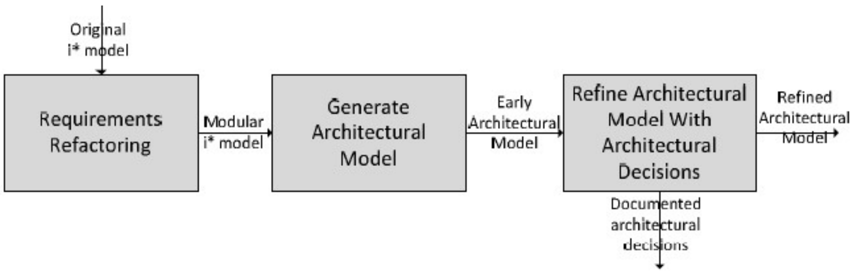

Original i* model
Requirements Refactoring
Modular i* model
Generate Architectural Model
Early Architectural Model
Refine Architectural Model With Architectural Decisions
Refined Architectural Model
Documented architectural decisions

Fig. 3.4: Overview of the STREAM-ADD process [13]

The decision documentation process is based on a template containing parameters such as Requirements, Stakeholders, Design Fragment, **Groups**, Status, Consequences, Dependencies, etc. As such the template is used in the third step of the STREAM-ADD process, *Refine Architectural Model With Architectural Decisions.* The template is filled in six steps.

1. Identify Requirements and Stakeholders Addressed by the Decision
2. Identify Architectural Alternatives
3. Perform Contribution Analysis of Alternatives
4. Perform Trade-off Analysis of Alternatives
5. Specify Architectural Decision Design Fragment
6. Fill Additional Information

Part of the Additional Information in step six is the group field which is aimed at specifying the type of architectural decision made. Dermeval et al. focus this grouping on the requirements group addressed by the architectural decision. This approach differs from the ones presented earlier in this chapter. It requires a prior, if only implicit, grouping of the requirements a system has. This then in turn would allow the decision to be assigned to the requirements and their respective groups. This approach seems to create domain-specific groups as requirements differ wildly from system to system. As such there are no proposals for group-tags made by Dermeval et al. However, the approach of binding the decision to the requirement groups is a conceivable alternative to having predefined groups of decisions. By basing the decision group on the related requirements group, an automatic link is created between the decision and the requirements which influence them or which they themselves affect.
Zimmermann et al. [52] propose a conceptually similar decision documentation framework to the ones introduced earlier in this chapter. Based on reusable decision templates they aim to capture the knowledge gained on other projects with similar architectural styles. They propose a model based on three conceptual steps, namely *Decision Identification, Decision Making* and *Decision Enforcement.* In the first step, the Decision

Identification, the initial decision model for a project team is instantiated from project-specific requirements models and the reusable decision templates. The used decision templates serve as a checklist for the architecture team, in the form of an early and informal review of the architectural work done. The second step, the decision making, uses the decision models created in the first step, in combination with a list of decision drivers for each earlier decision. For each type of decision a supporting technique, such as ADD, is selected. That allows the team to document each decision specific to its needs rather than using the same technique across all decisions whether it applies or not. This means that based on the model from the first step, the decision driver catalog and the support technique, the Decision Outcome and Justification can be created and documented accordingly in a highly specified way. Lastly, the decision enforcement step makes sure that the Architectural Decision Outcomes find their way into the resulting code. Techniques are employed at build and deployment time so that concepts such as code aspects and configuration policies can be used to express architectural intent. Additionally, the approach supports machine-readable decision models which can in turn be interpreted by model transformations and code generators to guarantee that made decisions are actually implemented in the code.
Within the Framework, closely related architecture decisions (AD) are grouped into so-called ADTopics. These can form hierarchies assigned to one of the three ADLevels of abstraction, *ConceptualLevel, TechnologyLevel, or AssetLevel.* Like in Jansen et al. [26] this grouping technique deals with abstraction levels of decisions, however on a less fine-grained level. These architecture decision levels could also be implemented within the context of this thesis as three fixed groups of abstraction for all documented decisions to be sorted under, as they are not necessarily limited to architectural decisions. Zimmermann et al. however go into further detail with their interpretation of the ADTopics. Introducing topics such as Executive Decisions (see Kruchten et al. [33]), Enterprise Architecture Decisions, Process Realization Decisions, and Service Realization Decisions, they discuss that theoretically the list of ADTopics is endless as long as a consistent naming style is implemented and comparable factors exist between the topics to guarantee proper assignment. In [53] Zimmermann et al. give a more detailed list of possible decision types. Table 3.6 shows these types along with their respective architecture design level.

Table 3.6: Decision types according to [53]

Decision Type	**ADLevel**
Executive decisions, requirements analysis decisions	Executive level
Pattern Selection Decisions (PSDs)	Conceptual level
Pattern Adoption Decisions (PADs)	Conceptual level
Technology Selection Decisions (TSDs)	Technology level
Technology Profiling Decisions (TPDs)	Technology level
Vendor Asset Selection Decisions (ASDs)	Vendor asset level
Vendor Asset Configuration Decisions (ACDs)	Vendor asset level

While these decision types offer a fine level of detail for a possible grouping strategy, they are not as well distinguished from another as the groups offered by other approaches and such require increased effort from a documentation side. This in turn makes them rather unattractive as a real live model because the effort linked with explicitly documenting decisions could be a hindrance for developers. A problem which the automation of decision knowledge creation and extraction tries to solve and not increase further.
Carriere et al. [8] introduce an approach that, while not necessarily applicable to this thesis, has interesting implications on a more business-oriented context. They introduce a framework for making architectural decisions based on cost-benefit trade-offs rather than directly software-related aspects. Their motivation behind this approach is that cost, especially in the case of refactoring operations, is often one of the deciding factors. It is however often neglected in decision making as it is difficult for developers to see their software from a more business-oriented side. Taking as input a set of prior refactoring decisions documented as tickets, they build a company-specific model that suggests the cost-benefit of future related decisions. Their method focuses on extracting the benefits of a decision made in the past since the cost is more easily calculated by the effort expended. To achieve this, the components related to each ticket are extracted and the coupling of said components is calculated. Carriere et al. hypothesize that after refactoring the average coupling of the components would decrease thus indicating the benefit of the refactoring operation. Having established this cost-benefit analysis their approach allowed developers to receive an estimate of the cost-benefit of a future refactoring decision. To allow a more precise estimate, tickets were separated into groups. While not directly decision groups, this more business-oriented grouping could nonetheless be of potential interest as a decision grouping scheme. The categories were *Price Test, Price Data, Website UI, Tool, and Price Logic.* While a group like Website UI could be too specific to certain companies, the idea of using a business, cost-oriented grouping scheme is nonetheless interesting. By sorting decisions into these types of groups it could help to improve the availability especially for stakeholders, customers, and the business end of software development companies. The results of Carriere et al.'s model show a correlation between the coupling and a concrete organizational benefit but their results are too coarse to actually predict proposed refactoring decisions.
Lastly, van der Ven et al. [49] present their approach which analyses the version management data of large open-source repositories to help architects make design decisions. Important for this thesis is their introduction of levels of decisions. They distinguish **High-Level** Decisions, **Medium-Level** Decisions and **Realization-Level** decisions. High-Level decisions are those that affect the product as a whole. They are often affected by management or enterprise architects and people in general that are not involved as much as the developers themselves. Medium-Level decisions are those that influence specific components and frameworks and lastly, Realization-Level decisions involve the structure of the code itself, for example which APIs to use. This level of decision grouping offers an interesting alternative to the other grouping schemes discussed in this chapter. It is the first one to take the impact and the relative "size" of a decision into context. Instead of binding the decision to requirements or other software artifacts, this approach would capture the scope and magnitude a decision has on the software project. This would allow developers to more easily judge how important a decision is and how big the effects on the system could be.

3.4 Synthesis

Table 3.7 compares the relevant papers based on multiple criteria. The goal is to evaluate the advantages and disadvantages of the approaches in order to find the most applicable one for the work in this thesis and to summarize the findings of the review. The criteria and their abbreviations in Table 3.7 are as follows:

- Is there a prototype or implementation of the approach? (Prototype)
- Is the approach evaluated? (Evaluated)
- Is a grouping scheme for decisions explicitly designed? (Specific design)
- How complex is the assignment of groups (i.e. are they clearly divided, are they hard to understand, how much extra knowledge is needed to assign them)? (Complexity)
- How relevant to the scope of this thesis is the grouping scheme? (Relevance)

The reasoning behind Prototype and Evaluated is the necessity to implement the grouping strategy in this thesis, thus an approach that is already implemented might have an advantage over a purely theoretical approach. A lot of papers discussed here do not offer an explicit definition of the possible decision groups but rather make vague mentions of possible grouping schemes. This motivates the Specific design question because an explicitly defined scheme would be preferable. Since decision documentation is in and of itself an additional effort for the developer, the goal should be to minimize this additional effort. Otherwise the documentation might not be carried out as carefully or at all in order to save time. Thus the complexity of assigning the groups needs to also be factored into the selection process, motivating the Complexity question. Some approaches offer interesting schemes that could find application in a business-related context. However, since this thesis aims to expand upon a plug-in used in a university context, some of these schemes might be out of scope. While ConDec is an open-source project, the amount of data and experience for business-related aspects is limited. That is why a Relevance criterion, which limits the selection to an assessable project scope, is necessary.

Table 3.7: Synthesis of Literature Review

Ref	Grouping Scheme	Criteria	Answers
[3] [4]	Structural, Behavioral, Ban	Prototype	ML-based Classifier
		Evaluated	Yes on self-created dataset
		Specific design	Explicitly designed for decision classification
		Complexity	Assignment is fairly easy if familiar with the three basic groups
		Relevance	High relevance to the area of application
[7]	Tags based on application domain	Prototype	ADDSS
		Evaluated	Modelling of real life system using ADDSS
		Specific design	No specific design but rather general group tag attribute
		Complexity	Not complex since no boundaries are given
		Relevance	Not specific enough to be relevant
[8]	Grouping by monetary aspects (e.x. Price Test, Price Data, Price Logic)	Prototype	Model implemented and trained
		Evaluated	Tested in real life project
		Specific design	Scheme not specifically designed for decisions but for tickets
		Complexity	Very complex since additional information about monetary context is necessary
		Relevance	Not applicable to scope of project since no monetary aspects are present
[13]	Assign Decision to Requirements group addressed by it	Prototype	Part of Future Work
		Evaluated	No evaluation only running example
		Specific design	No specific design but rather binding decisions to requirements groups
		Complexity	Complex since prior grouping of requirements is necessary
		Relevance	No direct grouping so not applicable
[22]	Multiple groups per decision such as by architecture team and quality attribute	Prototype	Framework doesn't require implementation but viewpoints are created in real software project
		Evaluated	Case Study
		Specific design	No specific design only some examples
		Complexity	Complexity depends on number of groups per decision and types of groups selected
		Relevance	Not specific enough to be relevant
[26]	Abstraction based (Requirements, Implementation, Detailed Design, Software Architecture and Architectural Significant Requirements)	Prototype	No implementation
		Evaluated	Not evaluated
		Specific design	Not specifically designed to group decisions but rather a model of the decision making process
		Complexity	Low complexity since decisions only have to be assigned to the relevant process step
		Relevance	High relevance as development process steps are universal
[33]	Existence (Structural, Behavioral, Ban/Non-Existence) , Property, Executive Decisions	Prototype	Theoretical Framework without Implementation
		Evaluated	Part of Future Works
		Specific design	Specifically designed for Decision Grouping
		Complexity	Some Complexity in assigning subgroups of existence decision and requires familiarization with predefined groups
		Relevance	Extremely relevant to the scope of this thesis
[48]	Stakeholder based grouping	Prototype	Theoretical Framework without Implementation
		Evaluated	Only on own example
		Specific design	Specific grouping category without fixed tags
		Complexity	Higher complexity because of the need to identify stakeholders and assign fitting tags
		Relevance	Not as relevant because of a lack of different stakeholders in the scope of ConDec
[49]	High-, Medium and Realization Level Decisions	Prototype	Conceptual model only
		Evaluated	Only proof of concept no proper evaluation
		Specific design	No specific grouping, only focusing on one level of Decision
		Complexity	Low complexity because of ease to understand and assign categories
		Relevance	High relevance to scope of thesis

[52]	Conceptual Level, Technology Level, Asset Level	Prototype	Conceptual Framework
		Evaluated	Framework applied to enterprise application development
		Specific design	Explicitly designed to group decisions
		Complexity	Low complexity because of ease of assignment
		Relevance	High relevance to thesis
[53]	Executive decisions, PSDs, PADs, TSDs, TPDs, ASDs, ACDs	Prototype	Tool Implementation: Architectural Decision Knowledge Wiki
		Evaluated	Thorough evaluation by experiments and industrial case studies
		Specific design	Specific design for assignment of decision types
		Complexity	Highly complex grouping scheme with a need for the knowledge of multiple stakeholders
		Relevance	Too complex for the scope of this thesis

3.5 Review Discussion

This chapter will summarize the answers to the proposed research question and draw conclusions for the further implementation of this thesis.

RQ1: How is the grouping of decision knowledge realised and used?

RSQ1: What grouping schemes are employed to categorize decisions?
The used grouping schemes differ vastly for the discussed approaches. RSQ1 is perhaps best answered by the first column of Table 3.7. The following paragraph servers more to highlight the overlap between the different groups.
The scheme introduced by Kruchten et al. [33] is referenced and to some extend adapted by a number of approaches [3][4], but there are some who employ very different techniques. Additionally, there is a difference between those that declare clear tags for their grouping scheme [3][4][26][33][49][52][53] and those that merely introduce a general category in which the relevant grouping tags should be created [7][13][13][22][48].

RSQ2: How are groups of decisions documented?
In general all approaches discussed in this chapter document their decision groups in the form of clear text terms. These terms are either fixed groups presented by the authors [4][3][33][49][52][53] or they are up to developers to create and use. In the latter case there are little to no constraints made by any of the authors. Their general idea is to create terms that explain the group and keep a consistent nomenclature across all of the created categories.

RSQ3: How are these groups of decisions implemented ?
These group categories either serve as part of a decision documentation framework [22][33][48][52], are used by tools to generate structures [7][8][53] or are more conceptual ideas extracted from works with a different focus [13][26].

RSQ4: How is filtering used in the presented techniques ?
Unfortunately none of the approaches offered any meaningful inside into filtering. This might be due to the fact that none of them strictly focus on the grouping of decisions and as such not enough emphasis is put onto filtering possibilities. Only Tyree et al. [48] mention filtering at all in this context.

As a result of the conducted research it has been shown that Kruchten et al. [33] is one of the most prevalent approaches in the research field (Fig 3.2). This prevalence shows good compatibility with the scope of this thesis. However, the still fairly high complexity of assignment, leads to the conclusion that, while this grouping scheme would make a fitting addition to the ConDec project, users should not be forced to choose from these groups exclusively. Merely recommending them during group assignment, seems more likely to motivate users to use the grouping system, as they can not be expected to intrinsically understand the underlying definitions behind these groups.
Thus, support for custom decision groups should be implemented, which will allow the grouping scheme introduced by Kruchten et al. to also be used. These custom groups will not have a fixed naming scheme as they are supposed to be universally usable across all types of projects. This will allow developers to sort decisions by more user-defined categories. An example would be using a *"Security"* or a *"Performance"* label to find all decisions related to those topics.
Additionally, the level of decision hierarchy presented by Ven et al. [49] is also implemented. Grouping decisions by *High-level, Medium level* and *Realization Level* introduces a clearer structure into the overall importance and effects of each decision. It also carries a lower time effort to assign the levels and can thus be used as a baseline for each decision.

4 Requirements

This chapter introduces the requirements for the extension of the ConDec plug-in during this thesis. Section 4.1 introduces the initial roughly specified requirements as a general estimate of which features were required. Section 4.2 presents the personae for different roles present in the user base of the plug-in extension. Next, Section 4.3 gives an overview over the domain data relevant to the thesis. Section 4.4 introduces the functional requirements generated from the previously gained knowledge, after which Section 4.5 introduces the relevant non-functional requirements. Section 4.6 shows the workspaces which the plug-in introduces. Lastly, Section 4.7 presents mockups of the new workspaces and features, followed by a summary of the chapter in Section 4.8.

4.1 Initial Coarse Requirements

The implementations created in this thesis follow the goal of supporting both rationale manager as well as developers in maintaining and improving the consistency between documented decisions and software artifacts such as code and requirements. To support this, multiple views are either implemented or improved upon. Additionally, existing filters are improved and new ones implemented uniformly across the platforms relevant views.
CR1: A dashboard is implemented which estimates the decision knowledge documentation quality across a whole project. The dashboard must allow a rationale manager to get a general overview of the project, including existing requirements, code classes, issues, and decisions. Metrics are introduced, which analyse the rationale completeness of the project, including metrics such as how many issues do not have decisions connected and vice versa. The dashboard will allow the rationale manager to identify incomplete decision documentation and navigate to the respective element quickly.
CR2: New views are implemented which allow an overview of the relationships between documented decision knowledge and their related code artifacts. The views visualize the decision knowledge related to each individual code class, allow management of the knowledge, and allow the users to group the decision knowledge via the use of tags. For all views expandability is a big focus, using them as interfaces to access the underlying data and designing them to be easily adaptable in the future. New filters and functionalities must be easy to add should the need arise.

4.2 Personae

The main concern of this thesis are the roles of the rationale manager described in Table 4.1 and the developer described in Table 4.2. The ConDec extension provided by this thesis aims to support these plug-in users. It focuses on providing them with the tools needed to more efficiently complete their tasks, without introducing any further frustrations or complexity to their every-day usage. These Personae are used to allow a better understanding of the user-base's needs. They help in the creation of features and influence the user interface.

Table 4.1: Persona: Rationale Manager

Job	**Rationale Manager**
Biography	Aged 31, Master's degree in Computer Science, working for a big software company. Has previously supervised projects and uses ConDec to get an overview over all issues and decision knowledge elements.
Knowledge	3 years of experience using Jira. Currently supervising multiple projects at the same time.
Needs	Assess the overall completeness of decision knowledge across a project.
Frustrations	Undocumented or incomplete decisions. Having to manually check every element for documentation consistency.
Ideal Features	Visualization of metrics, which show the completeness of the documented decisions. Direct navigation to incomplete decision knowledge elements. Visualization of general progress within a project. Configuration of whether to include code repositories in metric calculation. Filters for decision knowledge related views.

Table 4.2: Persona: Developer

Job	**Developer**
Biography	Aged 22, educated software developer, working for a big software company.
Knowledge	Familiar with IntelliJ and Eclipse as IDE's, Git as VCS and Jira as ITS. Experience in agile programming using CSE and familiar with decision knowledge documentation.
Needs	Wants to non-intrusively document decisions. Needs to understand relationships between decisions and code.
Frustrations	Documentation spread across different tools, because it creates difficulty in performing code reviews and change impact analyses. Manual identification of which task effected which code segments. Being overloaded with decisions that are not related to the current task.
Ideal Features	Visualization of code effected by a task. Visualization of relationship between all code classes and related decisions. Assignment possibility of groups for decisions to consolidate them. Manage the decision groups in bulk. Filters for decision knowledge related views.

4.3 Domain Data

Figure 4.1 presents the domain data model with the relevant relationships between the decision knowledge and the code classes, as well as the grouping feature. Each Jira issue contains an unspecified amount of knowledge elements. Each knowledge element is associated with a knowledge type and a documentation location. Depending on the combination of both a knowledge element can either be classified as a decision knowledge element or a code class. A code class itself can not have a knowledge type, which is associated with either of the five decision knowledge types and must have a commit as the documentation location. Code classes are extracted through commits. This is also where the connection between classes, Jira issues, and, in extension, the decision knowledge elements is created. Commits are attached to the Jira issue, from where the connection is extracted. All of these elements and their links between each other form the knowledge graph. Within this graph the elements form the nodes and the links the edges of the structure.
Additionally, the grouping tag is attached to the knowledge element specifically, allowing grouping of all kinds of knowledge elements.

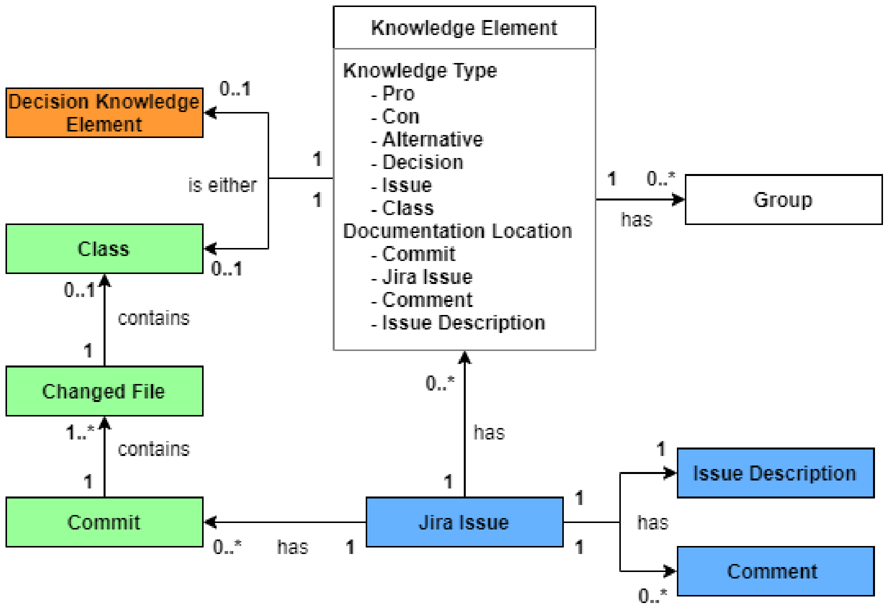


Fig. 4.1: Domain Data Model for Decision Knowledge and Code Class relationship within ConDec

4.4 Functional Requirements

This section will provide details about the functionalities that the ConDec extension, discussed in this thesis, provides. The extension supports two user tasks (4.4.1), one for each of the relevant roles. They were extracted from the initial course requirements (Section 4.1). As such, the first user task (**UT1**) is performed by the rationale manager, whose broad requirements are sketched in **BR1**. The second user task (**UT2**) is performed by developers, including roles close to the development itself, such as requirements engineers, testers, or software architects. The task originates in **BR2**. These user tasks are divided into sub-task from which the system functions (4.4.2) are derived.

4.4.1 User Tasks

UT1: Rationale Management
The rationale manager is tasked with setting up the decision knowledge management process within a project. This means deciding which issue and decision schemes are used and which knowledge types are relevant to the project. The rationale manager also has to achieve a general overview of the progress on the project and then analyse and maintain the completeness and consistency of the documented decision knowledge.
ST1.1: Configure Rationale Management process
The rationale manager defines the rationale management process for each project independently and specifies which schemes and knowledge types are relevant and thus needed for the project. Additionally, the rationale manager decides whether knowledge is to be extracted from Git repositories or not.

ST1.2: Analyse basic metrics of a project
During the lifespan of the project the rationale manager needs to assess the current status of the project by analysing some general aspects about the complexity and progress of the different tasks. This is required to quickly assess the progress within the project, as that influences the stringency of the required consistency.
ST1.3: Analyse quality of documented decision knowledge
The rationale manager checks and maintains the quality of the documented decision knowledge. This guarantees high quality and consistency across each project.

UT2: Development of Software
Developers initially elicit software requirements. They then specify these requirements and then implemented them. The decisions made during the development process are documented by the relevant parties. They either document these decisions in Jira directly or by documenting the decisions in code comments or commit-messages. During or between development cycles they perform change impact analyses and review the documented knowledge to gain insight into the decision and their effects on the code.
ST2.1: Elicit and Manage the Requirements
Before the implementation phase begins the requirements engineers need to establish the relevant requirements by analysing which features are desired by the customer. During the development phase these requirements need to be managed and adapted to changing needs.
ST2.2: Implement the code
Developers write the source code according to the elicited requirements and implement the desired features.
ST2.3: Continuously document decision knowledge in Jira
The decisions being made by requirements engineers during the elicitation and adaptation of the requirements need to be documented in Jira. Developers also need to document all decisions made during the implementation phase of the project. Their decisions can be documented in Jira or in commit messages.
ST2.4: Perform a change impact analysis
Developers need to perform analyses on the impact of changes that are made to make consistent decisions. This includes an analysis of the impact of past changes and their effects on future implementations.
ST2.5: Review the documented knowledge of a development task
Developers need to see the knowledge, which is associated with their current or past tasks. This knowledge includes code changes, documented decisions, commits, and connections between decisions and code.

4.4.2 System Functions and User Stories

The system functions (SF) describe the tasks and features of the ConDec extension from a system standpoint. They are defined by necessary inputs, outputs, exceptions, and conditions. For each system function a relevant user story is given, which explains the role and motivation of a user. Because of the size of the ConDec project, only those system functions, which are relevant to this thesis are listed here. Additionally Figure 4.2 and Figure 4.3 show example charts to better visualize the metrics discussed in the relevant system functions.

SF1: Show basic knowledge metrics for project progress

> As rationale manager, I want to see basic metrics that assess the progress of a project in order to analyse the work being done. Since managing multiple projects at once makes it difficult to keep up with all of them, these metrics allow me to estimate the stringency of the required consistency.

Table 4.3: SF: Show basic knowledge metrics for project progress

Preconditions:	Jira project exists and ConDec is activated/enabled
Input:	WS1.1: Project key, Jira issue type
Postconditions:	Nothing changed
Output:	WS1.1: Metrics displaying the basic information
Exceptions:	None
Rules:	Metrics include box plots showing the... ...average number of comments per Jira issue ...average number of decisions per Jira issue ...average number of decision problems per Jira issue Also include pie charts showing the... ...number of code classes and requirements in a project ...number of knowledge elements from different documentation locations
Supports:	Subtask ST1.2

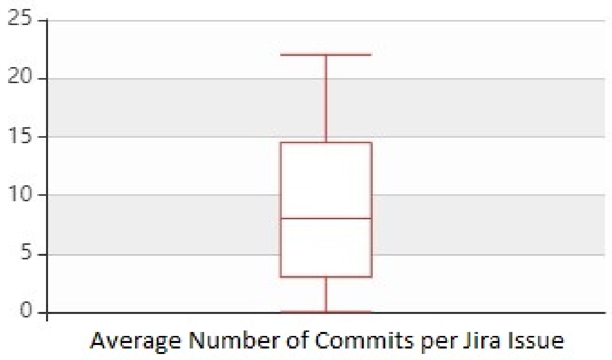


Fig. 4.2: Example box plot for basic knowledge metrics

SF2: Show rationale completeness metrics within decision knowledge

As rationale manager, I want to see inconsistencies between decision knowledge elements. This includes metrics that show issues without decisions or arguments and decisions which do not have pro's and con's.

Table 4.4: SF: Show rationale completeness metrics within decision knowledge

Preconditions:	Jira project exists and ConDec is activated/enabled
	Decision knowledge elements exist
Input:	WS1.1: All decision knowledge elements
Postconditions:	Nothing changed
Output:	WS1.1: Rationale completeness is visualized in plots
Exceptions:	None
Rules	Metrics include pie charts showing the...
	...decision problems with and without linked decisions
	...decisions with and without linked decision problems
	...decisions and alternatives with and without pro's and con's
Supports:	Subtask ST1.3

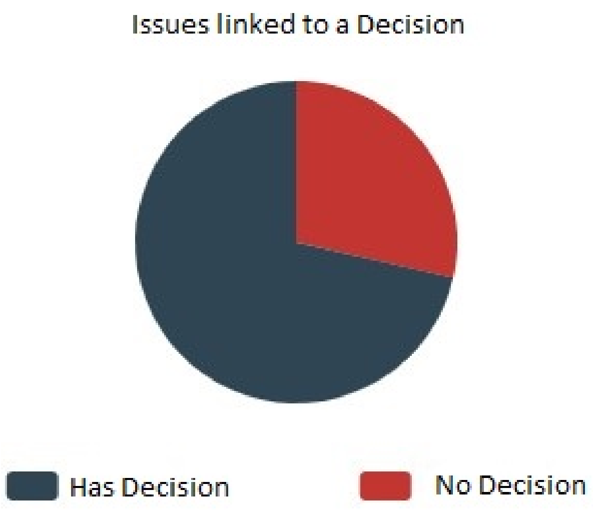


Fig. 4.3: Example pie chart for completeness metrics

SF3: Show completeness metrics of decision knowledge for requirements and tasks

As rationale manager, I want to quickly get an overview over the completeness of the documented decision knowledge within a Jira project.

Table 4.5: SF: Show completeness metrics of decision knowledge for requirements and tasks

Preconditions:	Jira project exists and ConDec is activated/enabled Jira issue(s) with relevant type exist
Input:	WS1.1: Project key and Jira issue type
Postconditions:	Nothing changed
Output:	WS1.1: Plots that show completeness metrics of decision knowledge for Jira issues with the chosen type
Exceptions:	If no repository is given completeness might not be correctly calculated as connections between elements are lost
Rules	Metrics include pie charts showing the... ...Jira issue type items having a linked decision / decision problem ...relevance of comments in Jira issues ...distribution of knowledge types in the issue type items
Supports:	Subtask ST1.3

SF4: Navigate from plots in dashboard to respective knowledge element

As rationale manager, if I find inconsistencies in a Jira issue, by analysing the metrics in the dashboard, I want to be able to navigate to that Jira issue directly.

Table 4.6: SF: Navigate from plots in dashboard to respective knowledge element

Preconditions:	Jira project exists and ConDec is activated/enabled
Input:	WS1: A datapoint in the plot (e.g. in boxplot or pie chart)
Postconditions:	Nothing changed
Output:	WS2: The Jira issue page is shown
Exceptions:	The Jira issue was deleted between the metric calculation and the navigation
Supports:	Subtask ST1.3

SF5: Configure decision knowledge extraction from Git

As a rationale manager, I want to configure whether knowledge is extracted from a Git repository or not. I want to be able to do this for each project individually.

Table 4.7: SF: Configure decision knowledge extraction from Git

Preconditions:	Jira project exists and ConDec is activated/enabled Git Repository exists
Input:	Project key, enabled/disabled extraction from git, URI of Git Repository
Postconditions:	Updated project settings, Repository is cloned
Output:	Success message that configuration has changed
Exceptions:	Repository cloning fails, Repo URI is invalid
Supports:	Subtask ST1.1

SF6: List all code classes for a project

As a developer, I want to see all code classes with their linked Jira issues and knowledge elements to see relationships and perform a change impact analysis.

Table 4.8: SF: List all code classes for a project

Preconditions:	Jira project exists and ConDec is activated/enabled Git connection is configured
Input:	WS3.6: Project Key, Git Repository and Knowledge Graph
Postconditions:	Nothing changed
Output:	WS3.6: List of all code classes
Exceptions:	None
Supports:	Subtasks ST2.4 and ST2.5

SF7: List all code classes connected to a Jira issue

As a developer, I want to see all code classes that were created or altered during the course of a task.

Table 4.9: SF: List all code classes connected to a Jira issue

Preconditions:	Jira project exists and ConDec is activated/enabled Git connection is configured and one commit exists
Input:	WS2.5: Jira issue, Related Commits
Postconditions:	Nothing changed
Output:	WS2.5: All related code classes
Exceptions:	Git Repository could not be cloned
Supports:	Subtask ST2.5

SF8: Group Knowledge Elements

As a developer, I want to be able to connect related decisions with a tag, to establish a relationship between them and see which decisions might be affected.

Table 4.10: SF: Group Knowledge Elements

Preconditions:	Jira project exists and ConDec is activated/enabled Knowledge elements exist
Input:	WS2 & WS3: Knowledge Element, Group tag to assign, rename or delete
Postconditions:	The group tag is assigned, renamed or deleted from the knowledge element and the relevant database entry is created
Output:	WS2 & WS3: Confirmation massage
Exceptions:	None
Supports:	Subtask ST2.3

SF9: Manage knowledge element groups

As a developer, I want to be able to do bulk changes to the group tags when requirements change, to keep the knowledge up to date and consistent.

Table 4.11: SF: Manage knowledge element groups

Preconditions:	Jira project exists and ConDec is activated/enabled Knowledge elements exist and at least one group exists
Input:	WS3.7: The group to delete/rename, (the new group name)
Postconditions:	The group is renamed/deleted and all relevant database entries are completed
Output:	WS3.7: Confirmation message
Exceptions:	None
Supports:	Subtask ST2.3

SF10: Filter knowledge graph

As a ConDec user, I want to be able to customize which information is shown to me in the decision knowledge views, to be able to see more specific information.

Table 4.12: SF: Filter knowledge graph

Preconditions:	Jira project exists and ConDec is activated/enabled Knowledge graph with knowledge element(s) (=nodes) and links (=edges) exists
Input:	WS1, WS2 & WS3: Filter criteria, knowledge graph
Postconditions:	Nothing changed
Output:	WS1, WS2 & WS3: Knowledge elements and links that match the filter criteria
Exceptions:	None
Rules:	Filter criteria: - Text search for summary of knowledge element - Knowledge Type - Creation date - Resolution date - Documentation Location - Status - Link distance in the knowledge graph - Link type - Group
Supports:	Subtasks ST2.4, ST2.5, ST1.2 and ST1.3

4.5 Non-Functional Requirements

The non-functional requirements (NFR) describe underlying attributes that must be present in the finished software. These can include user-oriented attributes like providing simple accessibility to the user or more system focused properties, like security measures that have to be in place. The ISO / IEC 25010 [25] quality model defines eight general types of non-functional product quality requirements.

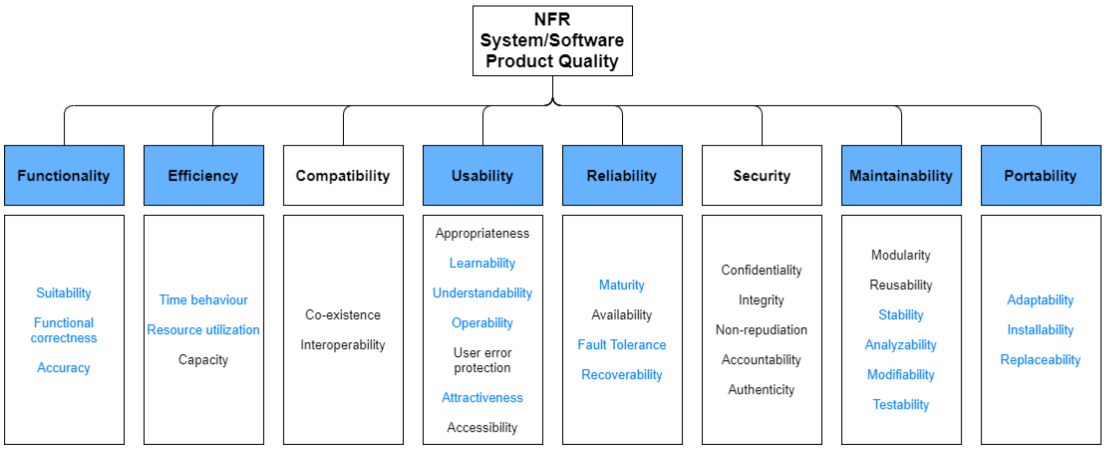


Fig. 4.4: ISO / IEC 25010 [25] Quality Model

As can be seen in Fig 4.4, these requirements are subdivided into multiple categories. This allows further specification of the requirements. The requirements highlighted in blue, show the ones which ConDec focuses on achieving across the whole plug-in. Note that Compatibility and Security are both important requirements. Security however is handled mostly by Jira itself already and thus does not require special focus within ConDec at this time. Compatibility in the case of a plug-in such as ConDec has a lot of overlap with the portability requirement and can thus be considered as one joint requirement in this case.
Sections 4.5.1, 4.5.2 and 4.5.3 detail the three NFR's this ConDec extension put its focus on and the measures that were taken to ensure their fulfilment.

4.5.1 Modifiability

A big focus of this thesis was to provide a plug-in extension that could, at a later date, easily be extended upon. Especially the dashboard added in WS1.1 could become subject to such an extension. The metrics used to measure decision documentation completeness may change at any time. Additionally, new metrics could be required as the plug-in increases in size and complexity. As such, providing a base from which further extensions could be written became paramount. To provide this base generic code must be provided which allows future metrics to be added without requiring the creation of additional classes. Merely adding the desired metric to the dashboard and providing its calculations must be enough to include it in the dashboard.

4.5.2 Time behavior

The ongoing development on ConDec and the fairly sizable Jira project, which now exists for the plug-in, made a focus on the time behavior of the plug-ins views become apparent. Loading times can be a big issue for complex projects with high amounts of interconnectivity. As the data is processed to calculate metrics or provide decision knowledge views, the user is forced to wait. The problems start when loading times become unacceptably long for the users, who as a result may choose to no longer use the affected views. The dashboard has a high risk of running into this problem, because of the amounts of calculations and iterations necessary to compute the relevant metrics. This problem is additionally exacerbated, because of the implemented filter, which in some cases requires completely new calculations. As a result boundary times were set for all newly implemented views, which were considered reasonable. Views were not allowed to surpass these loading times for the complex test project used. For further details on specific values see Section 7.3.

4.5.3 Attractiveness

Attractiveness is in no way a vanity requirement. It has specific effects on the user and their interest in the usage of the software [41]. Providing views which are both pleasing to use and look at, while also being appropriate for the context in which they are used, may encourage users to work with these functionalities. Also, by providing these views in a styling which is similar to the environment in which they are used, allows them to blend into this environment and thus also blend into the workflow of its users. As this thesis extends upon an existing plug-in, it was challenged with the task of providing views that seamlessly blend into both the plug-in's existing views and the overall styling of Jira. As such, focus was put on using the UI elements provided by Jira so as to blend into the style of Jira and with that also blend into the workflow of the user.

4.6 Workspaces

The workspaces (WS) define all the relevant environments, views, and navigations provided by the software. This section highlights those, which are added by the ConDec extensions provided by this thesis. The extension can be divided into roughly three different environments. First, the dashboard environment (WS1) which will be expanded with a rationale completeness and general statics metrics dashboard (WS1.1). Second, the Jira Issue Model View (WS2), which will feature a view for connected code classes

(WS2.4). Lastly, the decision knowledge page (WS3), which provides multiple decision knowledge related views and will be extended by a code class view (WS3.6) and a decision group management view (WS3.7). The UI-Structure diagrams in Figures 4.5, 4.6 and 4.7 show these new workspaces and which system functions are fulfilled by the extensions.

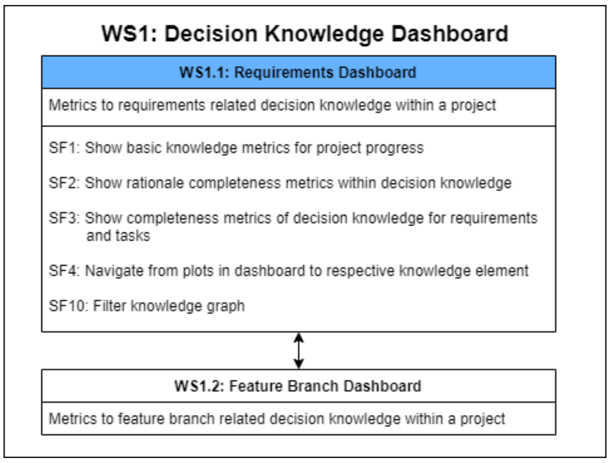

WS1: Decision Knowledge Dashboard
WS1.1: Requirements Dashboard
Metrics to requirements related decision knowledge within a project
SF1: Show basic knowledge metrics for project progress
SF2: Show rationale completeness metrics within decision knowledge
SF3: Show completeness metrics of decision knowledge for requirements and tasks
SF4: Navigate from plots in dashboard to respective knowledge element
SF10: Filter knowledge graph
WS1.2: Feature Branch Dashboard
Metrics to feature branch related decision knowledge within a project

Fig. 4.5: ConDec Workspace 1

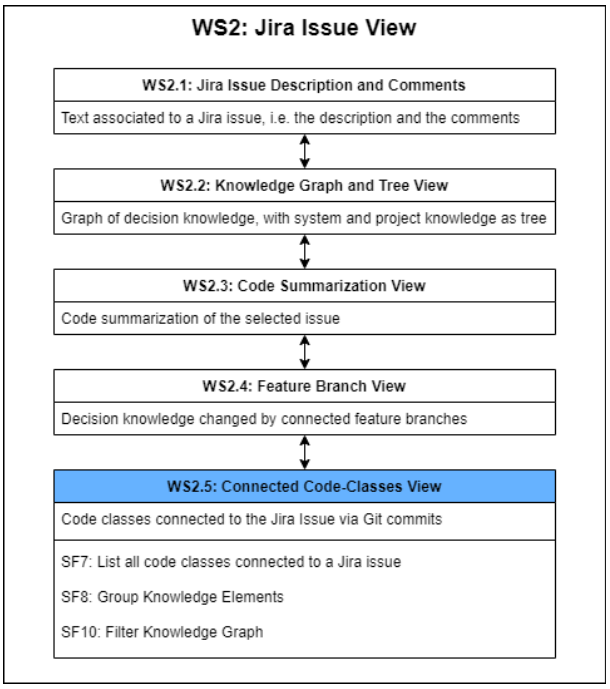

WS2: Jira Issue View
WS2.1: Jira Issue Description and Comments
Text associated to a Jira issue, i.e. the description and the comments
WS2.2: Knowledge Graph and Tree View
Graph of decision knowledge, with system and project knowledge as tree
WS2.3: Code Summarization View
Code summarization of the selected issue
WS2.4: Feature Branch View
Decision knowledge changed by connected feature branches
WS2.5: Connected Code-Classes View
Code classes connected to the Jira Issue via Git commits
SF7: List all code classes connected to a Jira issue
SF8: Group Knowledge Elements
SF10: Filter Knowledge Graph

Fig. 4.6: ConDec Workspace 2

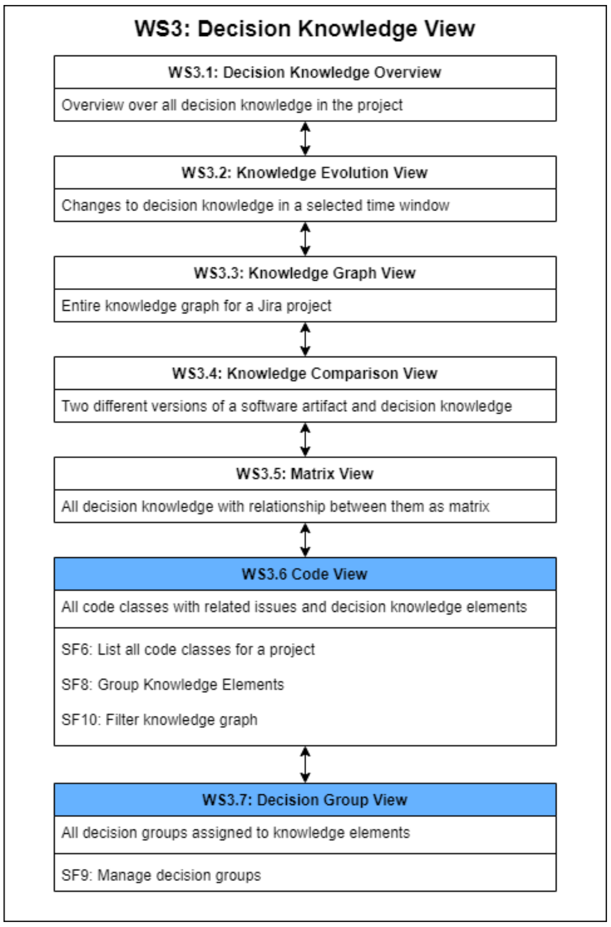

WS3: Decision Knowledge View
WS3.1: Decision Knowledge Overview
Overview over all decision knowledge in the project
WS3.2: Knowledge Evolution View
Changes to decision knowledge in a selected time window
WS3.3: Knowledge Graph View
Entire knowledge graph for a Jira project
WS3.4: Knowledge Comparison View
Two different versions of a software artifact and decision knowledge
WS3.5: Matrix View
All decision knowledge with relationship between them as matrix
WS3.6 Code View
All code classes with related issues and decision knowledge elements
SF6: List all code classes for a project
SF8: Group Knowledge Elements
SF10: Filter knowledge graph
WS3.7: Decision Group View
All decision groups assigned to knowledge elements
SF9: Manage decision groups

Fig. 4.7: ConDec Workspace 3

4.7 Mock-ups

This section provides mock-ups for the four views introduced by the ConDec extension. These are based on the previously designed workspaces 1.1, 2.5, 3.6 and 3.7 (Section 4.6).
WS1.1 introduces the Requirements Dashboard, a mock-up for which, can be seen in Figure 4.8. At the top of the dashboard a project and issue type selection menu is added to allow fast switching between these aspects. Below that, the metrics are implemented in box plots for the general statistics and pie charts in the case of the completeness metrics. By clicking on the slices in the pie charts a dialog is opened which allows the navigation to the related issues. This navigation can be seen in Figure 4.9.

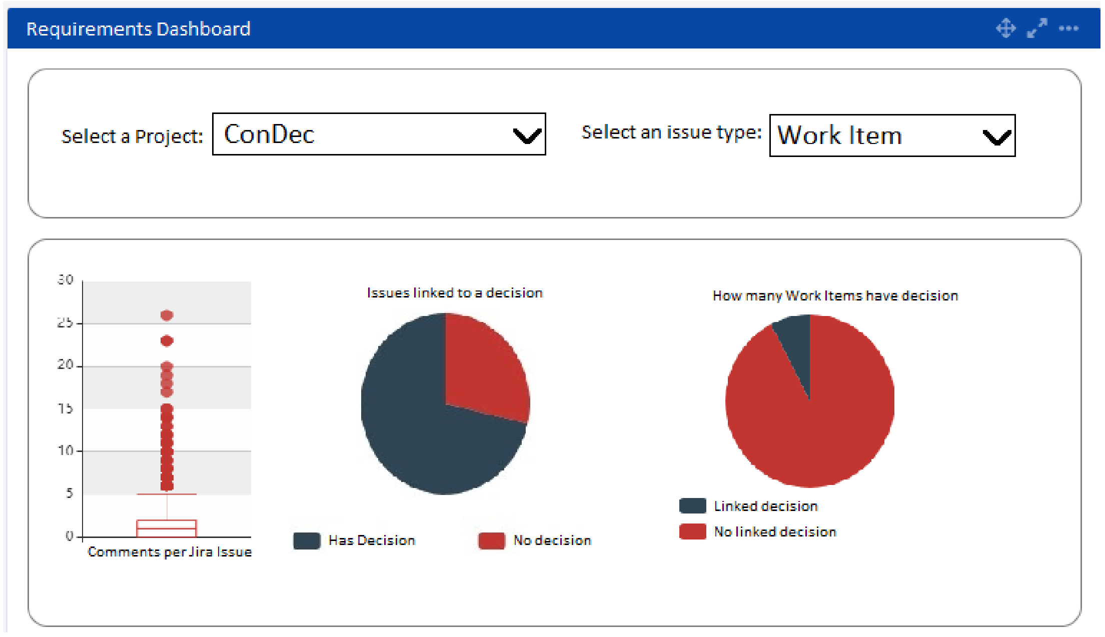


Fig. 4.8: ConDec Requirements Dashboard Mock-up

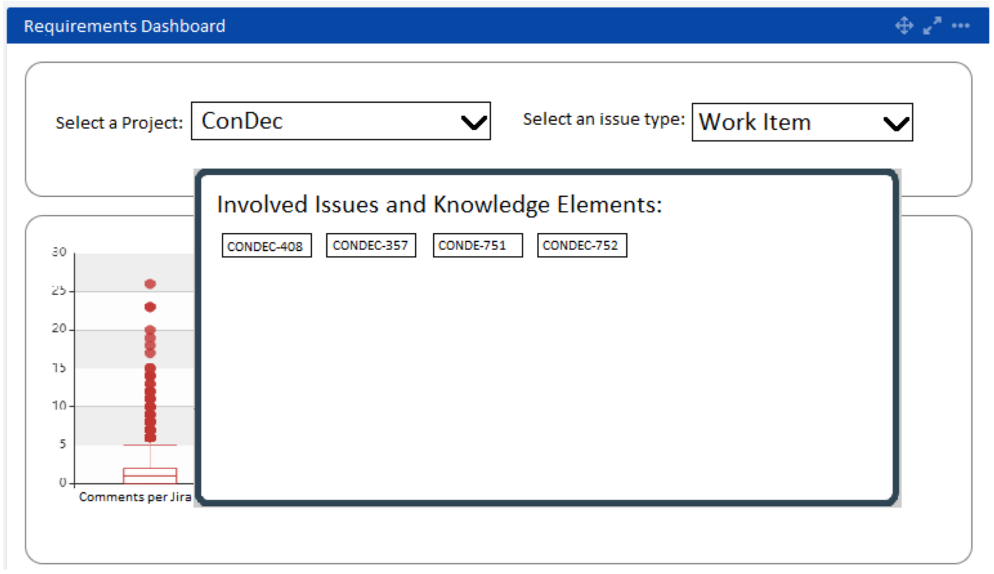


Fig. 4.9: ConDec Requirements Dashboard Navigation Mock-up

WS2.5 adds a connected code-classes view to the Jira Issue View, a mock-up for which, can be seen in Figure 4.10. The view is added on to the existing Tree Visualization, Graph Visualization, and Feature Branch(es) Views. It displays the issue itself connected to all code classes that were created or altered because of the issue, as a tree.

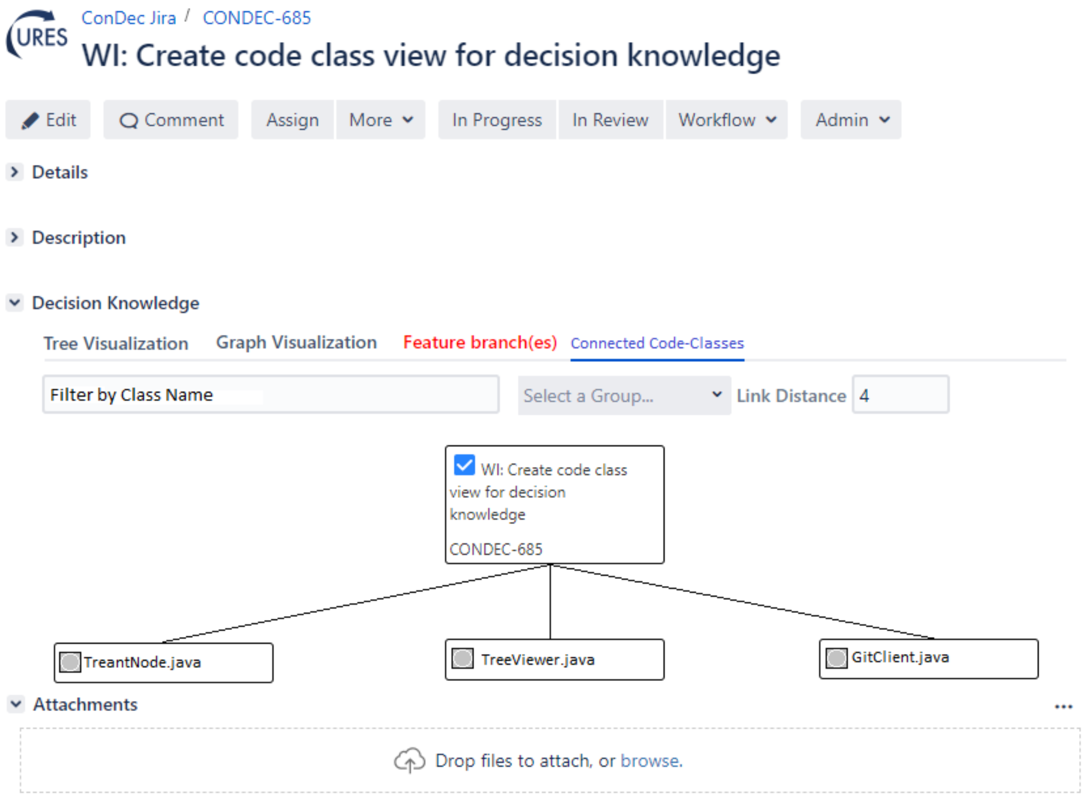


Fig. 4.10: ConDec Jira Issue Code Class View Mock-up

WS3.6 introduces a code class view, which lists all code classes present in the connected repositories and allows users, by clicking on the classes, to see the connected issues and their documented decision knowledge. This tree is presented on the right side of the class listing. Figure 4.11 presents a mock-up for this view.

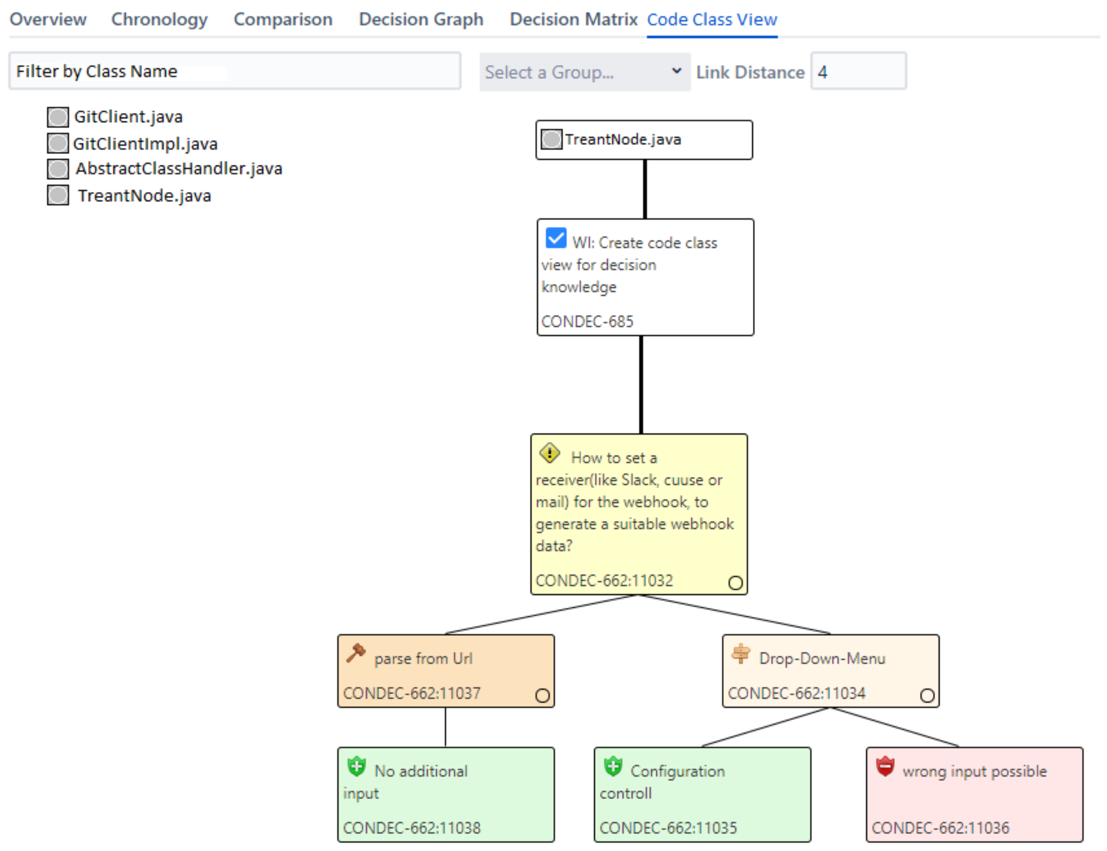
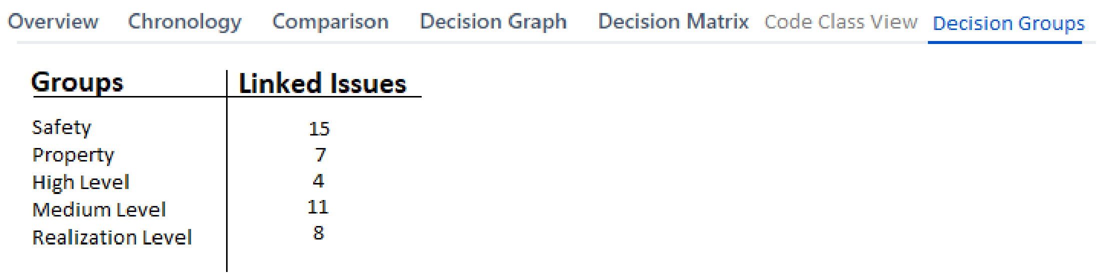


Fig. 4.11: ConDec Code Class View Mock-up

Lastly, WS3.7 introduces the decision group management view. This view shows a table that lists all decision groups present in a project together with the number of issues assigned to that group. By right-clicking a group, the user is able to select from a drop-down menu whether they want to delete the group or change its name, which will be propagated to all assigned issues. The mock-up for this view is depicted in Figure 4.12.



Groups	Linked Issues
Safety	15
Property	7
High Level	4
Medium Level	11
Realization Level	8

Fig. 4.12: ConDec Decision Group View Mock-up

4.8 Requirements Chapter Summary

The ConDec extension implemented in this thesis was based on a set of broad requirements for two roles of software stakeholders. Requirements were specified for the rationale manager, whose task it is to check and maintain consistency about all decision knowledge related documentation. These requirements specify a dashboard for the rationale manager, which provides basic project-specific metrics and completeness metrics about all decision knowledge documentation in these projects. The requirements for the second role, the developers, specify a comprehensive overview of the relationships between decision knowledge and implemented code classes. They also support for the grouping of documented decisions to create more connections between related decisions. Another important requirement is the ability to filter across all decision-related views in order to provide more targeted access to the documented data. As a result of these requirements, four new workspaces were elicited which provide the desired features. The requirements phase precedes the design and implementation phase of the project.

5 Design and Implementation

This chapter introduces the architectural decisions and design decisions made during the development of the ConDec extension both on functional and non-functional requirements. It also discusses challenges in the design and implementation phases. Section 5.1 introduces the overall architecture of the plug-in. Sections 5.2, 5.3 and 5.4 specify the architecture and design of the extension. Additionally, they explain the implemented views in detail.

5.1 Overview of ConDec Architecture

Figure 5.1 displays the overview class diagram for ConDec. The general structure of the plug-in can be separated into five main packages. Note that this diagram only visualzies the backend classes of the plug-in.
The model classes represent the domain data, i.e. different entities present within the plug-in. The decisions themselves are represented by the *KnowledgeElement* class which carries attributes such as the *DocumentationLocation*, *KnowledgeStatus* and *KnowledgeType*. As previously described in Section 2.4, documentation locations can include Jira issues, comments, descriptions, commits, and code comments. The different knowledge types are defined by decision issues, decisions, alternatives, and pro or con arguments. The knowledge status includes categories like discarded, rejected, or resolved. Knowledge elements are connected via links defined by the *Link* class. These links are directional and carry a *LinkType*. The *KnowledgeGraph* contains information about all knowledge elements and their links and is used to access and display information. Additionally, ConDec requires model classes for extracting and recognizing code changes (*Diff, ChangedFile*) and for the extraction of knowledge elements from text within Jira issues (*PartOfJiraIssueText, TextSplitter*).
Communication between frontend and backend is handled by REST-interfaces which allow the processing, accessing, and storing of the required data. The backend is written in Java-code, while the frontend uses HTML and JavaScript to visualize and display the decision knowledge. The Jira API allows the seamless blend-in of the plug-in into Jira.
View classes are used for visualization of the decision knowledge. An example for this visualization can be seen in Figure 2.5 in the Fundamentals Chapter 2. The *TreeViewer* class provides the list of all decision knowledge which is then visualized in a tree structure by the *Treant* class. The *VisGraph* is used by WS3.3: Knowledge Graph View to display all decision knowledge within a project in one complete graph, which is then specifiable through filters.

Persistence classes are used to access the settings and manage the databases of the plug-in. These allow access to stored information or configuration throughout the whole plug-in, making it easier to customize the views according to a user's preferences.
Lastly, *Config* classes are used to configure the plug-in and manage the settings. Users can edit the ConDec settings according to their needs. This includes features like deciding whether to extract knowledge from connected Git repositories or not (SF5) or editing the rationale model used for each project.

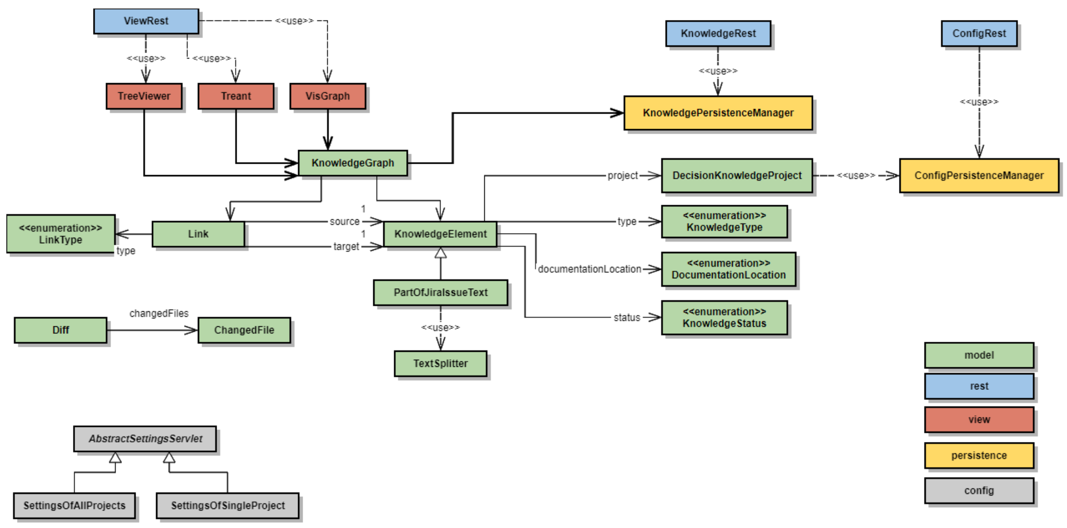


Fig. 5.1: Overview Class Diagram ConDec

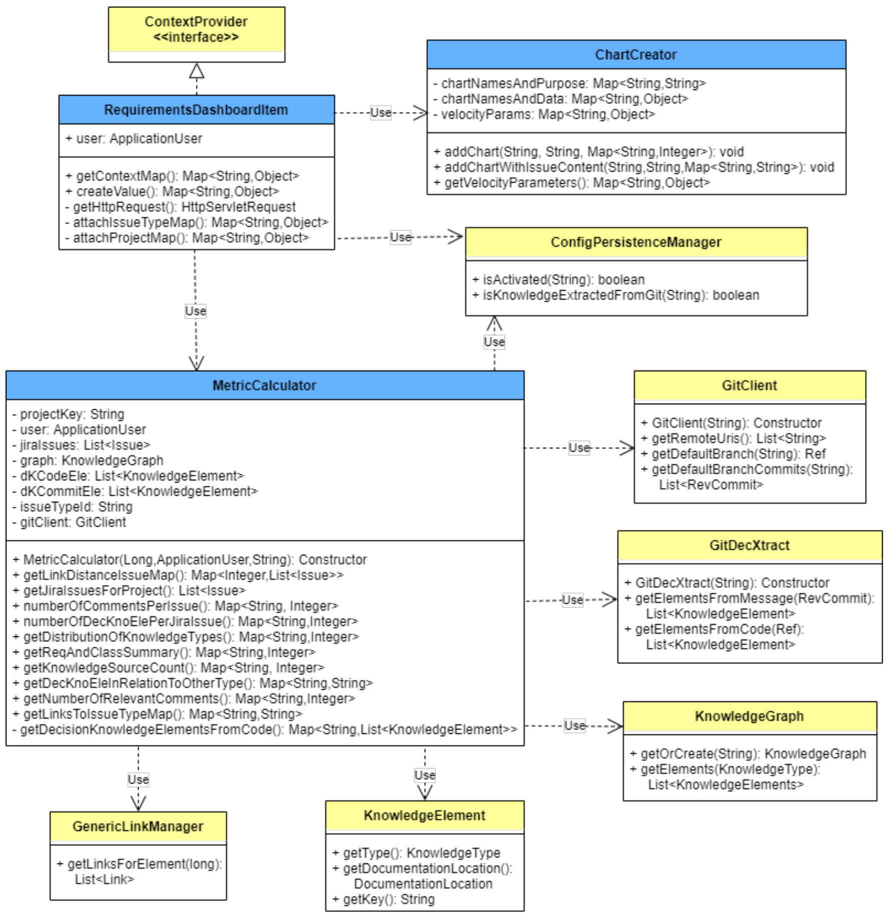
The ConDec extension made by this thesis can be split into three sections. First the creation of the requirements dashboard, then the visualization of Code-To-Work-Item relationships and lastly the addition of the decision grouping feature. The following Sections 5.2, 5.3 and 5.4 will discuss these additions in detail.

5.2 Requirements Dashboard

Section 5.2.1 introduces the architecture of the requirements dashboard, by describing the relevant parts of the class diagram. The blue classes represented those implemented by the extension, while the yellow classes are either interfaces or already existing classes within the plug-in. Section 5.2.2 describes the most important architectural decisions and Section 5.2.3 discusses decisions made in regard to the system functions introduced in Section 4.4.2. Lastly, Section 5.2.4 introduces the resulting views.

5.2.1 Architecture

The goal of the Requirements Dashboard was to provide a view for the rationale manager, which allowed a quick, encompassing overview of the completeness of the documented decision knowledge within a project. It is also supposed to provide a general overview of the progress on each project, by providing some general statistics metrics, like the amount of documented decision knowledge elements per Jira issue or the number of code classes and requirements specified in each project. Figure 5.2 shows the relevant parts of the class diagram. The *RequirementsDashboardItem* class provides the context backend for this view. It generates the information, including project lists and metric content, that is displayed on the Dashboard. The metrics are processed in the *MetricCalculator* class which accesses the other necessary parts of the plug-in, including the Git client, extraction classes, and the knowledge graph. The *ChartCreator* class uses the processed metric data to create the box plot and pie chart objects which are passed on to the frontend, to be displayed. The class separation allows easy modifiability (Section 4.5.1) in the future. To add additional metrics, developers only have to pass the calculations to the chart creator and add them to the frontend HTML code.



ContextProvider <<interface>>
RequirementsDashboardItem
+ user: ApplicationUser
+ getContextMap(): Map<String,Object>
+ createValue(): Map<String,Object>
- getHttpRequest(): HttpServletRequest
- attachIssueTypeMap(): Map<String,Object>
- attachProjectMap(): Map<String,Object>
ChartCreator
- chartNamesAndPurpose: Map<String,String>
- chartNamesAndData: Map<String,Object>
- velocityParams: Map<String,Object>
+ addChart(String, String, Map<String,Integer>): void
+ addChartWithIssueContent(String,String,Map<String,String>): void
+ getVelocityParameters(): Map<String,Object>
ConfigPersistenceManager
+ isActivated(String): boolean
+ isKnowledgeExtractedFromGit(String): boolean
Use
MetricCalculator
- projectKey: String
- user: ApplicationUser
- jiraIssues: List<Issue>
- graph: KnowledgeGraph
- dKCodeEle: List<KnowledgeElement>
- dKCommitEle: List<KnowledgeElement>
- issueTypeId: String
- gitClient: GitClient
+ MetricCalculator(Long,ApplicationUser,String): Constructor
+ getLinkDistanceIssueMap(): Map<Integer,List<Issue>>
+ getJiraIssuesForProject(): List<Issue>
+ numberOfCommentsPerIssue(): Map<String, Integer>
+ numberOfDecKnoElePerJiraIssue(): Map<String,Integer>
+ getDistributionOfKnowledgeTypes(): Map<String,Integer>
+ getReqAndClassSummary(): Map<String,Integer>
+ getKnowledgeSourceCount(): Map<String, Integer>
+ getDecKnoEleInRelationToOtherType(): Map<String,String>
+ getNumberOfRelevantComments(): Map<String,Integer>
+ getLinksToIssueTypeMap(): Map<String,String>
- getDecisionKnowledgeElementsFromCode(): Map<String,List<KnowledgeElement>>
GitClient
+ GitClient(String): Constructor
+ getRemoteUris(): List<String>
+ getDefaultBranch(String): Ref
+ getDefaultBranchCommits(String): List<RevCommit>
GitDecXtract
+ GitDecXtract(String): Constructor
+ getElementsFromMessage(RevCommit): List<KnowledgeElement>
+ getElementsFromCode(Ref): List<KnowledgeElement>
KnowledgeGraph
+ getOrCreate(String): KnowledgeGraph
+ getElements(KnowledgeType): List<KnowledgeElements>
GenericLinkManager
+ getLinksForElement(long): List<Link>
KnowledgeElement
+ getType(): KnowledgeType
+ getDocumentationLocation(): DocumentationLocation
+ getKey(): String

Fig. 5.2: Class Diagram: Requirements Dashboard

5.2.2 Architectural Decisions

Tables 5.1, 5.2 and 5.3 reference the most important architectural decision problems (DP). Table 5.1 weighs the general ideas behind using one single Dashboard to support the rationale manager rather than splitting the requirements dashboard into multiple smaller dashboards. Table 5.2 considers the implementation of the dashboard itself, as that directly affects the architecture needed. Lastly, Table 5.3 discusses the split of the *MetricCalculator* and *ChartCreator* class. Note, that decisions are also valued by their contribution to the NFR's this thesis mainly focuses on (see Section 4.5).

Table 5.1: DP: Should the metrics be split into multiple Dashboards?

Type	**Description**	**Modifiability**	**Time behavior**	**Attractiveness**
Decision	*Use a single Requirements Dashboard*			
Pros	- Quick set up		⊕	
	- One cohesive dashboard for completeness			⊕
Cons	- Longer loading times		⊖	
Alternative	*Split into a general, a rationale completeness and a Jira issue completeness dashboard*			
Pros	- Quicker loading times for each individual dashboard		⊕	
Cons	- Increased spread of information			⊖
	- Longer set up time		⊖	

Table 5.2: DP: Implement the calculations in JavaScript or Java?

Type	**Description**	**Modifiability**	**Time behavior**	**Attractiveness**
Decision	*Use Java for the backend and JavaScript for the frontend*			
Pros	- Approach similar to most ConDec features	⊕		
	- More robust programming language	⊕		
	- API support	⊕		
Cons	- Slower reaction		⊖	
	- More complicated caching	⊖		
Alternative	*Use JavaScript for the calculations as well*			
Pros	- Can use existing caching methods	⊕		
	- Better reaction to filters		⊕	
Cons	- Less robust	⊖		
	- Additional REST interface needed to access data	⊖	⊖	

Table 5.3: DP: Should metric calculation and chart creation be handled by one class?

Type	**Description**	**Modifiability**	**Time behavior**	**Attractiveness**
Decision	*Split them into two classes*			
Pros	- Less clutter	⊕		
	- Seperation between calculation and visualization	⊕		⊕
Cons	- Dependencies between classes harder to find	⊖		
Alternative	*Use one class for both*			
Pros	- No additional class calls		⊕	
	- Easier change impact analysis	⊕		
Cons	- Class becomes to big	⊖		
	- More complicated use for other dashboards	⊖		

5.2.3 System Function Related Decisions

SF1: Show basic knowledge metrics for project progress
The general metrics serve the rationale manager in quickly assessing the progress on a project, by providing basic statistics. The issue was, which metrics to include in this assessment. The decision was based on the broad requirements and a *decision knowledge report* feature which already existed in ConDec. Thus, it was decided to include three box plots showing the average number of comments, decisions, and decision issues per Jira issue. Additionally, two pie charts were added, one showing the number of code classes and requirements documented in the project and one showing the number of knowledge elements from different knowledge sources. These knowledge sources were set to be knowledge elements from commits, Jira issues, code, and the issue content, meaning Jira issue descriptions or comments.
For the box plots showing the average numbers, the issue came up of which Jira issue types to include. The decision was made to include all issue types in the calculations and not just the previously selected issue type. The reasoning behind this decision was the fact that a rationale manager would require an overview of the whole project and not just parts, in order to assess the progress. Rather, the selected issue type when setting up the dashboard would just be used for the issue type completeness metrics (see SF3).

SF2: Show rationale completeness metrics within decision knowledge
Similar to SF1, this system function required a decision about which metrics should be included to show rationale completeness. As of the writing of this thesis, ConDec contained no fully fleshed out definition of what constitutes rationale completeness. As such, metrics were constrained to basic completeness measures while leaving room for future expansion. A total of six pie chart metrics were added. They list decision issues with and without connected decisions and vice versa, as well as alternatives and decisions with and without pro and with and without con arguments.
Given the number of metrics it was an issue of how to uniformly present incompleteness. Thus, it was decided to maintain a static coloring scheme for these box plots, showing what were considered inconsistencies as red slices and complete knowledge elements as blue slices.

SF3: Show completeness metrics of decision knowledge for requirements and tasks
Again a decision had to be made about which metrics would be used to show completeness for the requirements and tasks. Similar to SF2 no definition of what constitutes completeness of the decision knowledge for the issue types existed yet. To see how prevalent the decision documentation in a project is however, metrics were introduced showing the selected issue types with and without decision issues and decisions. Additionally, an analysis of the relevance of the comments within the issue type was added. The decision on what constitutes relevance was between two alternatives. The first one being the relevance attribute of knowledge elements. This meant that all comments which contained a knowledge element, that was marked as relevant, would be considered relevant. The second one, which was then decided on, used the more basic measure of

whether the comment contained a decision knowledge element at all, to be considered relevant.
Lastly, a metric was added which showed the distribution of the knowledge types within the selected issue types. This would allow rationale managers to assess whether incompleteness existed in the project if one type was disproportionately more prevalent than another. For this metric a knowledge type could either be a decision issue, a decision, an argument, or an alternative.
Given the number of issue types it was an issue of how to specify the displayed information. Displaying all selected metrics for all issue types would prolong the calculation time and possibly be overwhelming for a single dashboard. As such, it was decided to prompt the user, to select an issue type from those available in the selected project, prior to the calculation of the metrics.
Same as with SF2, it was also decided to keep the coloring scheme for the pie charts, making possible inconsistencies red and complete element slices blue.

SF4: Navigate from plots in dashboard to respective knowledge element
The required navigation had the issue of how to implement this feature. The decision was in part dictated by the already existing *Feature Branch Rationale Quality* Dashboard. In order to maintain a uniform styling for the plug-in the same navigation was maintained. By clicking the relevant slice in a pie chart or the relevant point in the box plot an overlay listing all participating elements would be opened. Clicking the element would load the related page in a new tab. This allows quick access to the desired element without requiring the user to leave the dashboard.

SF5: Configure decision knowledge extraction from Git
Since knowledge elements can be documented in comments and code as well as in Jira, some metrics require a connection to the related Git repository. The issue was determining how to handle projects without Git connection. It was decided to implement the metrics in such a way, that these documentation locations would be ignored if no Git connection was enabled. This allows a more universal use of the dashboard, even if no repository is connected to the project. The dashboard metrics and layout are identical for the user in both cases.

SF10: Filter knowledge graph
For a list of all filter criteria see the definition of SF10 in Section 4.4.2. Implementing filters for the Dashboard, as with the other views described in Sections 5.3 and 5.4, created the decision problem, on which filters were relevant for the shown information. Table 5.4 lists, which filters were implemented and to which metrics they apply. While ConDec aims to provide filters as uniformly as possible for all views, restrictions had to be made for a multitude of reasons. Some filters were excluded due to irrelevance to the given view. The complexity of *Filtering* as a topic, and the resulting time-cost in relevance evaluation and implementation also created scenarios where not all desired filters could be implemented. However, ConDec in some cases offers these opportunities in other views. For example, filtering by documentation date is already offered in another view (WS3.2), which diminishes the loss of the filter.

Table 5.4: Requirements Dashboard Filters

Filter	**Description**	**Relevant Metrics**
Knowledge Extraction from Git	Checkbox on whether to extract knowledge elements from Git or only consider those already in Jira	#Decisions per Jira issue #Issues per Jira issue #Elements from Decision Knowledge Sources Issue Type items having at least one issue / decision documented
Link Distance	Steps from Jira issue to element in the knowledge graph, that should be considered for a metric (ex. commits have a distance of 1, code comments of 2)	#Decisions per Jira issue #Issues per Jira issue Issue Type items having at least one issue / decision documented
Knowledge Type	Specifies for which Knowledge Type (Issue, Decision, Argument, Alternative) metrics should be calculated (standard is for all types)	#Decisions per Jira issue #Issues per Jira issue #Elements from Decision Knowledge Sources Issue Type items having at least one issue / decision documented Relevant comments in Jira issues
Knowledge Status	Only consider knowledge elements with a certain status, for example idea, discarded, rejected	#Decisions per Jira issue #Issues per Jira issue #Elements from Decision Knowledge Sources Issues / Decisions with and without Decisions / Issues Alternatives / Decisions with and without pro / con arguments Issue Type items having at least one issue / decision documented Relevant comments in Jira issues Distributions of knowledge types
Decision Group	Only consider knowledge elements that are part of a certain decision group	#Decisions per Jira issue #Issues per Jira issue #Elements from Decision Knowledge Sources Issues / Decisions with and without Decisions / Issues Alternatives / Decisions with and without pro / con arguments Issue Type items having at least one issue / decision documented Relevant comments in Jira issues Distributions of knowledge types

5.2.4 Results

Figure 5.3 shows the result of the Requirements Dashboard (WS1.1) implementation and can be compared to the mockup in Figure 4.8 in Section 4.7. At the top of the dashboard the project selection is situated. By default only this section is shown until a project has been selected. On selection, another drop-down menu is shown, which allows the user to choose from all issue types present in the project. After the issue type is chosen, the metrics are calculated and the resulting dashboard is displayed. The dashboard is split into three sections according to the system functions SF1, SF2, and SF3. The general metrics are situated at the top, followed by the rationale completeness metrics, and lastly the selected issue type completeness metrics. The dashboard also offers filter capabilities in accordance with SF10, in order to specify the information for the rationale managers needs. Additionally, by clicking on the pie slices of supported metrics, users are shown a list of all relevant knowledge elements and can navigate to them by clicking on the desired element. This feature is in accordance with SF4.

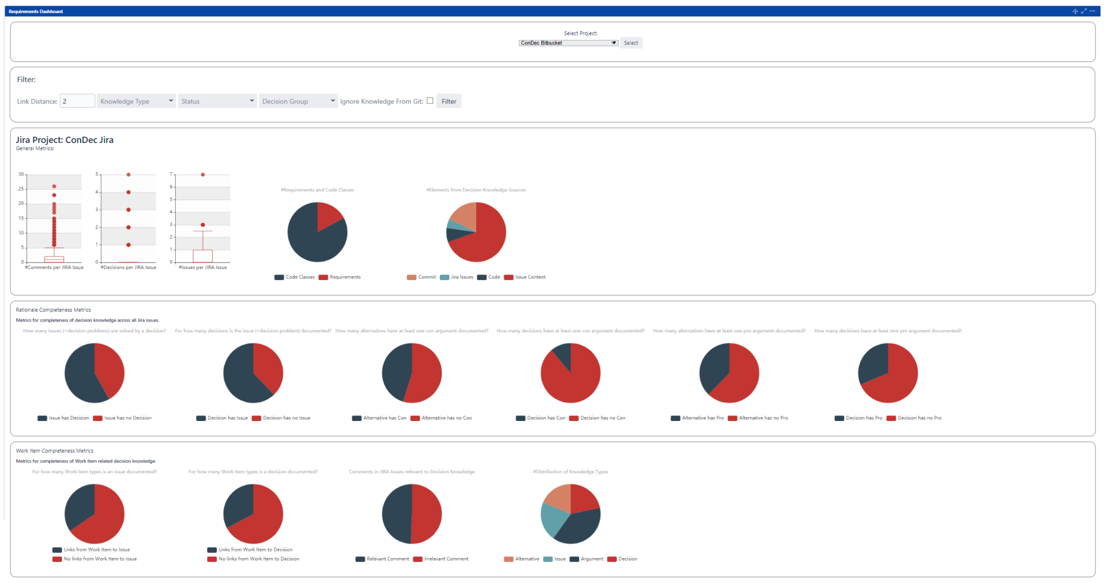

Requirements Dashboard
Select Project:
ConDec Bitbucket
Select
Filter:
Link Distance: 2
Knowledge Type
Status
Decision Group
Ignore Knowledge From Git:
Filter
Jira Project: ConDec Jira
General Metrics:
#Comments per JIRA Issue
#Decisions per JIRA Issue
#Issues per JIRA Issue
#Requirements and Code Classes
Code Classes
Requirements
#Elements from Decision Knowledge Sources
Commit
Jira Issues
Code
Issue Content
Rationale Completeness Metrics
Metrics for completeness of decision knowledge across all Jira issues.
How many issues (=decision problems) are solved by a decision?
Issue has Decision
Issue has no Decision
For how many decisions is the issue (=decision problem) documented?
Decision has Issue
Decision has no Issue
How many alternatives have at least one con argument documented?
Alternative has Con
Alternative has no Con
How many decisions have at least one con argument documented?
Decision has Con
Decision has no Con
How many alternatives have at least one pro argument documented?
Alternative has Pro
Alternative has no Pro
How many decisions have at least one pro argument documented?
Decision has Pro
Decision has no Pro
Work Item Completeness Metrics
Metrics for completeness of Work Item related decision knowledge.
For how many Work Item types is an issue documented?
Links from Work Item to Issue
No links from Work Item to Issue
For how many Work Item types is a decision documented?
Links from Work Item to Decision
No links from Work Item to Decision
Comments in JIRA Issues relevant to Decision Knowledge
Relevant Comment
Irrelevant Comment
#Distribution of Knowledge Types
Alternative
Issue
Argument
Decision

Fig. 5.3: Requirements Dashboard

5.3 Code To Work Item Relationship Visualization

Section 5.3.1 introduces the architecture of the requirements dashboard, by describing the relevant parts of the class diagram. The blue classes represented those implemented by the extension, while the yellow classes are either interfaces or already existing classes within the plug-in. Section 5.3.2 discusses the most important architectural decisions and Section 5.3.3 discusses decisions made in regard to the system functions introduced in Section 4.4.2. Lastly, Section 5.3.4 introduces the resulting views.

5.3.1 Architecture

While ConDec already contains a code change summary feature, this extension adds specific code class visualization features, in order to improve the connection between the issues, their related knowledge elements, and the implemented code classes. Figure 5.4 represents the relevant class diagram for these code class features. The code classes are stored in an active object database defined by the *CodeClassInDatabase* class (see decision in Table 5.5). This is also, how knowledge elements and their links are stored in Jira. The database and all related tasks including maintaining the classes and updating the elements in case of new commits is handled by the *CodeClassPersistenceManager*. The extraction of the classes itself from the commits and in extension to that, from the code, is handled by the *GitCodeClassExtractor*. This class uses the Git-Blame functionality to see which commit created which line of code and extracts the relevant Jira issue keys from there. Links are then created between the class and the Jira issue. The classes highlighted in green in Figure 5.4 are tasked with the visualization of this knowledge. Similar to the view shown in the Decision Knowledge Overview in Figure 2.5 within the Fundamentals Chapter 2, the *TreeViewer* provides the list of all code classes present in the project. The *Treant* displays a knowledge tree that shows the selected code class, its connected issues, and all their decision knowledge elements. Additionally, a view is provided in the Jira issue module, which displays all code classes related to that issue, as a tree.

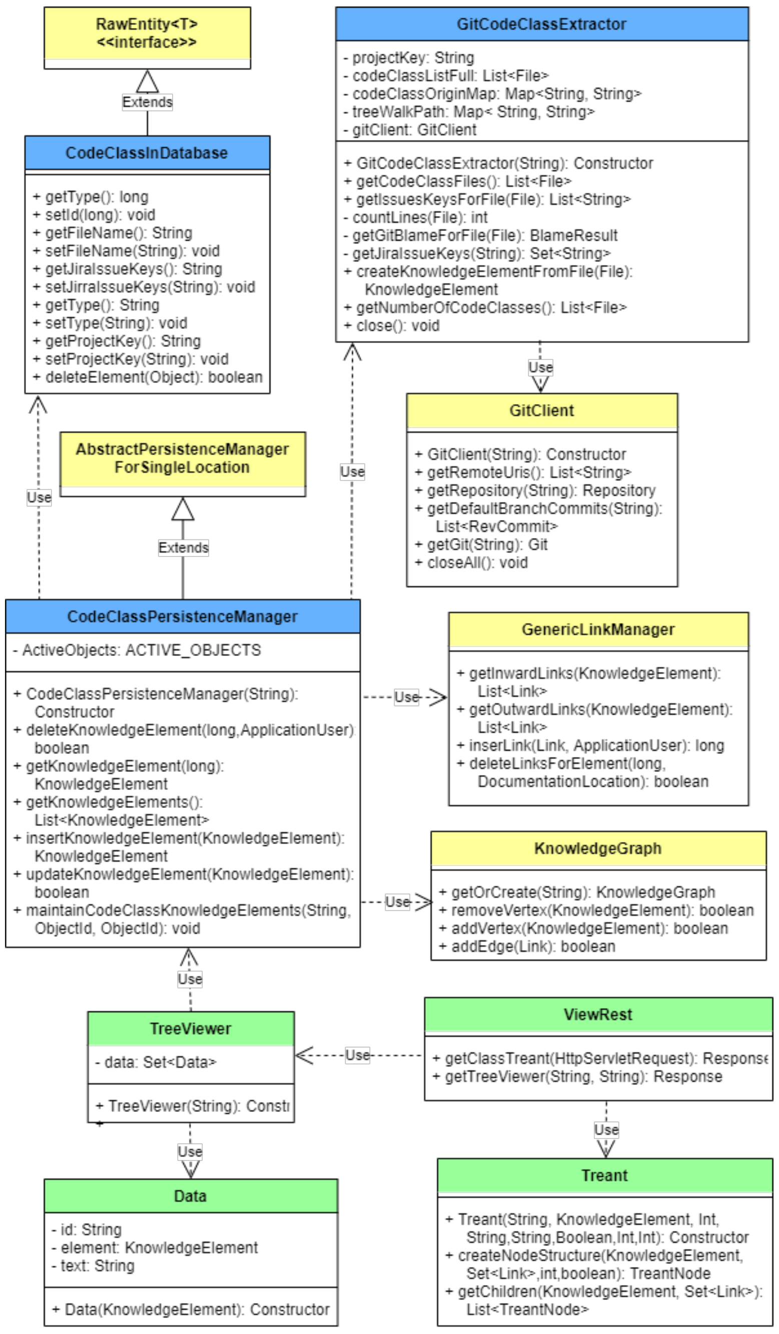

RawEntity<T>
<<interface>>
Extends
CodeClassInDatabase
+ getType(): long
+ setId(long): void
+ getFileName(): String
+ setFileName(String): void
+ getJiraIssueKeys(): String
+ setJirraIssueKeys(String): void
+ getType(): String
+ setType(String): void
+ getProjectKey(): String
+ setProjectKey(String): void
+ deleteElement(Object): boolean
GitCodeClassExtractor
- projectKey: String
- codeClassListFull: List<File>
- codeClassOriginMap: Map<String, String>
- treeWalkPath: Map< String, String>
- gitClient: GitClient
+ GitCodeClassExtractor(String): Constructor
+ getCodeClassFiles(): List<File>
+ getIssuesKeysForFile(File): List<String>
- countLines(File): int
- getGitBlameForFile(File): BlameResult
- getJiraIssueKeys(String): Set<String>
+ createKnowledgeElementFromFile(File): KnowledgeElement
+ getNumberOfCodeClasses(): List<File>
+ close(): void
Use
GitClient
+ GitClient(String): Constructor
+ getRemoteUris(): List<String>
+ getRepository(String): Repository
+ getDefaultBranchCommits(String): List<RevCommit>
+ getGit(String): Git
+ closeAll(): void
AbstractPersistenceManager ForSingleLocation
Use
Extends
Use
CodeClassPersistenceManager
- ActiveObjects: ACTIVE_OBJECTS
+ CodeClassPersistenceManager(String): Constructor
+ deleteKnowledgeElement(long,ApplicationUser) boolean
+ getKnowledgeElement(long): KnowledgeElement
+ getKnowledgeElements(): List<KnowledgeElement>
+ insertKnowledgeElement(KnowledgeElement): KnowledgeElement
+ updateKnowledgeElement(KnowledgeElement): boolean
+ maintainCodeClassKnowledgeElements(String, ObjectId, ObjectId): void
Use
GenericLinkManager
+ getInwardLinks(KnowledgeElement): List<Link>
+ getOutwardLinks(KnowledgeElement): List<Link>
+ inserLink(Link, ApplicationUser): long
+ deleteLinksForElement(long, DocumentationLocation): boolean
Use
KnowledgeGraph
+ getOrCreate(String): KnowledgeGraph
+ removeVertex(KnowledgeElement): boolean
+ addVertex(KnowledgeElement): boolean
+ addEdge(Link): boolean
Use
TreeViewer
- data: Set<Data>
+ TreeViewer(String): Constr
Use
ViewRest
+ getClassTreant(HttpServletRequest): Response
+ getTreeViewer(String, String): Response
Use
Use
Data
- id: String
- element: KnowledgeElement
- text: String
+ Data(KnowledgeElement): Constructor
Treant
+ Treant(String, KnowledgeElement, Int, String,String,Boolean,Int,Int): Constructor
+ createNodeStructure(KnowledgeElement, Set<Link>,int,boolean): TreantNode
+ getChildren(KnowledgeElement, Set<Link>): List<TreantNode>

Fig. 5.4: Class Diagram: Code Class Storage and Visualization

5.3.2 Architectural Decisions

Tables 5.5 and 5.6 reference the most important architectural decision problems (DP). Table 5.5 discusses the alternatives between the different storage options for the code classes in Jira and Table 5.6 weighs the different visualization options for the code class views. Note, that decisions are also valued by their contribution to the NFR's this thesis mainly focuses on (see Section 4.5).

Table 5.5: DP: How should the code classes be stored in Jira?

Type	**Description**	**Modifiability**	**Time behavior**	**Attractiveness**
Decision	*Store them in an Active Object Database*			
Pros	- Fast queries		⊕	
	- API support	⊕		
Cons	- No further description or comment capabilities for each class	⊖		⊖
Alternative	*Store the classes as Jira issues*			
Pros	- Description and Comment capabilities	⊕		⊕
	- Jira Link system can be used	⊕		
Cons	- Overloaded number of issues for big project		⊖	⊖
	- Slower access to all		⊖	
Alternative	*Don't store the Classes at all*			
Pros	- Requires less storage		⊕	
	- No database handling or managers required	⊕		
Cons	- Immense computation time to access data		⊖	

Table 5.6: DP: How should the visualization be implemented?

Type	Description	Modifiability	Time behavior	Attractiveness
Decision	*Use TreeViewer and Treant for visualization*			
Pros	- Already present in Project	⊕		⊕
	- Fleshed out libraries	⊕		
Cons	- More complex implementations	⊖		
Alternative	*Use simple HTML elements*			
Pros	- Easy to implement	⊕		
Cons	- Lower Attractiveness			⊖
	- Lower clarity			⊖

5.3.3 System Function Related Decisions

SF6: List all code classes for a project
The first decision that was made regarding SF6, was based on the issue of how to extract the code classes from the commits and maintain consistency. It was decided to extract all existing classes on plug-in installation an maintain them by analysing the changes of every new commit. Classes would be renamed, added and deleted according to the changes implemented by new commits, every time the plug-in made a pull request onto the repository.
As already mentioned in Table 5.6 the visualization was decided to be a list, which when a selection was made, would show a knowledge tree connecting the class, the related issue and all of its decision knowledge elements.
This also prompted the decision on the issue, which connections would be allowed in the tree. It was decided not to allow knowledge elements to directly link to a class, as deletions of classes could then result in knowledge becoming lost because of a loss of connection to a Jira artifact. Instead, all connections between code classes and knowledge elements would have to be through their connected issues. The handling of direct connections would be to difficult given the fact, that at its current state, the plug-in can extract knowledge from Git but not add it. This results in connections being fragile, with the constant moving, renaming, deleting and adding of classes.
Considering the grouping feature of SF8, a decision was made to allow users to also attach groups to code class elements in Jira. This in turn could then be used to implement the Group Filter (see SF10) to also work on code classes, allowing for more targeted access to the knowledge elements.

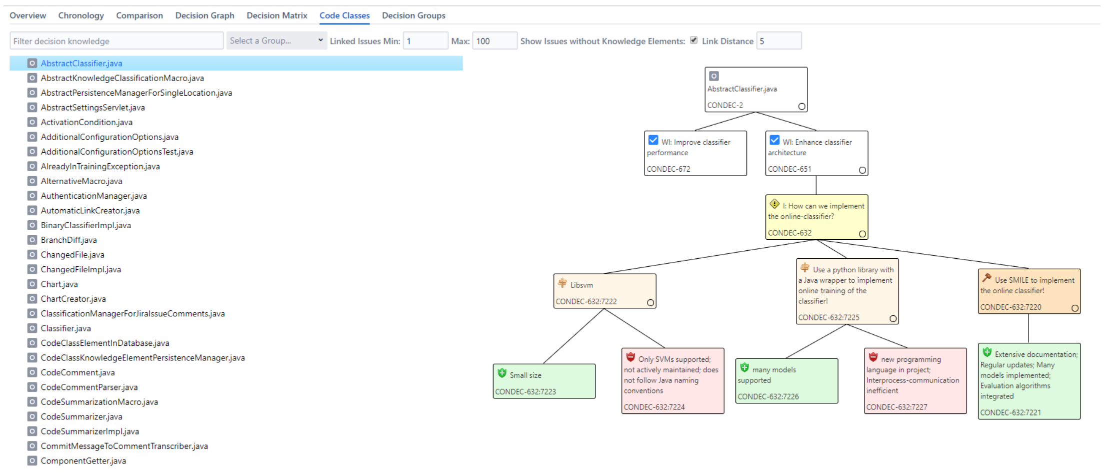
SF7: List all code classes connected to a Jira issue
The very limited space available to the required view in the Jira issue module was an issue and meant that a decision had to be made, on how much information could

be displayed in the view (WS2.5). Unfortunately, showing code classes and decision knowledge related to the selected issue would result in either a very zoomed out display of the knowledge tree or a view which could only show very limited parts of the tree. The reason behind this being, that the view is very limited in available horizontal space. As such, it was decided, that the views showing the related decision knowledge(WS2.2) and the one showing the related code classes(WS2.5) would remain separate.

SF10: Filter knowledge graph
An issue that occurred when designing the filters for the relevant views, was that space was limited and as such the user could be overwhelmed by the amount of elements shown. As such, it was decided to add a "*Show test classes*" filter for the Jira issue module Connected Code-Classes View(WS2.5), as the limited space could result in an overload of displayed classes. The filter will hide or show all classes considered to be tests, rather then implementation.
Next, a filter was added to the Code Classes View (WS3.5) allowing the user to hide and show connected Jira issues that do not have any knowledge elements attached to them. Similar to the previous decision this reduces clutter, if the information is not necessary.
Both views previously mentioned were also given a filter deciding whether to show code classes based on a minimum and maximum of connected Jira issues. Code classes having fewer connections than the minimum or more than the maximum would be hidden from view.

5.3.4 Results

Figure 5.5 shows the result of the Code View (WS3.6) and can be compared to the mockup in Figure 4.11 in Section 4.7. On the left the view presents a list of all code classes extracted from the connected Git repositories. By clicking on these their knowledge tree is shown on the right hand side. The classes are directly connected to Jira issues. These issues have their decision knowledge elements connected to them. On top of the view, the filters can be seen. The text search, group and number of linked elements filter apply to the list on the left. The link distance and "show issues without knowledge elements" filter apply to the tree on the right. The tree also offers the ConDec context menu feature. The context menu can be opened via right clicks and allows editing, group assignment and other knowledge related features.



Filter decision knowledge

Linked Issues Min: 1 Max: 100 Show Issues without Knowledge Elements: ✓ Link Distance 5

- AbstractClassifier.java
- AbstractKnowledgeClassificationMacro.java
- AbstractPersistenceManagerForSingleLocation.java
- AbstractSettingsServlet.java
- ActivationCondition.java
- AdditionalConfigurationOptions.java
- AdditionalConfigurationOptionsTest.java
- AlreadyInTrainingException.java
- AlternativeMacro.java
- AuthenticationManager.java
- AutomaticLinkCreator.java
- BinaryClassifierImpl.java
- BranchDiff.java
- ChangedFile.java
- ChangedFileImpl.java
- Chart.java
- ChartCreator.java
- ClassificationManagerForJiraIssueComments.java
- Classifier.java
- CodeClassElementInDatabase.java
- CodeClassKnowledgeElementPersistenceManager.java
- CodeComment.java
- CodeCommentParser.java
- CodeSummarizationMacro.java
- CodeSummarizer.java
- CodeSummarizerImpl.java
- CommitMessageToCommentTranscriber.java
- ComponentGetter.java

AbstractClassifier.java CONDEC-2

WI: Improve classifier performance CONDEC-672

WI: Enhance classifier architecture CONDEC-651

I: How can we implement the online-classifier? CONDEC-632

Libsvm CONDEC-632:7222

Use a python library with a Java wrapper to implement online training of the classifier! CONDEC-632:7225

Use SMILE to implement the online classifier! CONDEC-632:7220

Small size CONDEC-632:7223

Only SVMs supported; not actively maintained; does not follow Java naming conventions CONDEC-632:7224

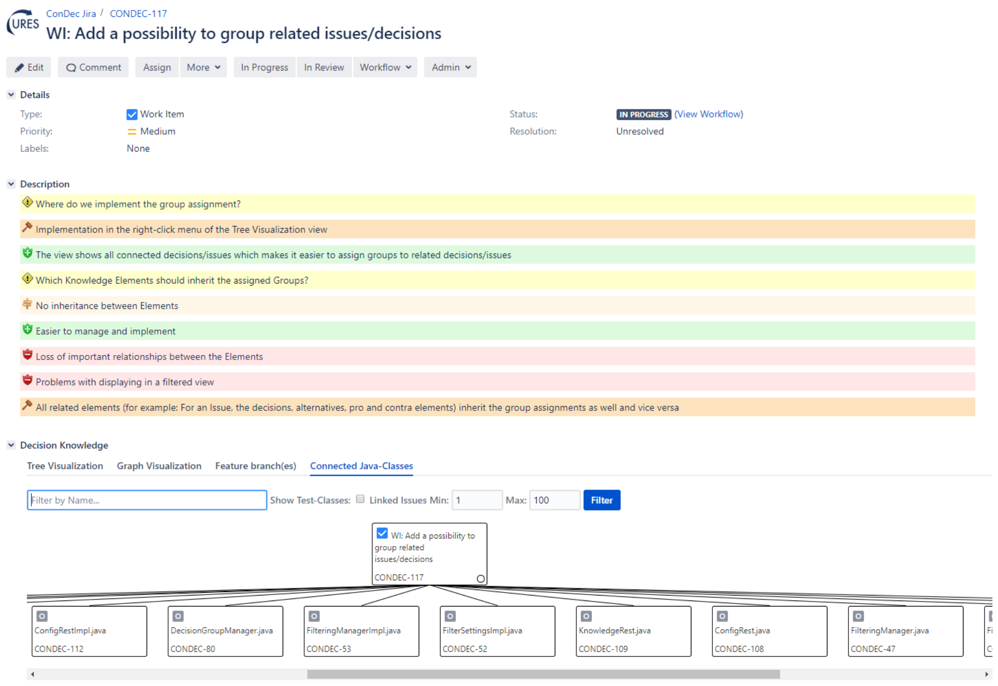
many models supported CONDEC-632:7226

new programming language in project; Interprocess-communication inefficient CONDEC-632:7227

Extensive documentation; Regular updates; Many models implemented; Evaluation algorithms integrated CONDEC-632:7221

Fig. 5.5: Code Classes View

Figure 5.6 shows the results of the Connected Code-Classes View (WS2.5) implementation and can be compared to the mockup in Figure 4.10 in Section 4.7. The issue model code class view shows all code classes that were created or edited as a result of the connected Jira issue. As of right now, the code classes are limited to Java classes. Connection is established if the class is present within a commit done in relation to the Jira issue. This view also offers a text search and a "number of linked elements" filter. Additionally, a checkbox exists which allows users to decide whether to display test classes or not.



WI: Add a possibility to group related issues/decisions

Details

Type:	Work Item	Status:	IN PROGRESS (View Workflow)
Priority:	Medium	Resolution:	Unresolved
Labels:	None		

Description

- Where do we implement the group assignment?
- Implementation in the right-click menu of the Tree Visualization view
- The view shows all connected decisions/issues which makes it easier to assign groups to related decisions/issues
- Which Knowledge Elements should inherit the assigned Groups?
- No inheritance between Elements
- Easier to manage and implement
- Loss of important relationships between the Elements
- Problems with displaying in a filtered view
- All related elements (for example: For an Issue, the decisions, alternatives, pro and contra elements) inherit the group assignments as well and vice versa

Decision Knowledge

Filter by Name... Show Test-Classes: ☐ Linked Issues Min: 1 Max: 100

WI: Add a possibility to group related issues/decisions CONDEC-117

ConfigRestImpl.java CONDEC-112

DecisionGroupManager.java CONDEC-80

FilteringManagerImpl.java CONDEC-53

FilterSettingsImpl.java CONDEC-52

KnowledgeRest.java CONDEC-109

ConfigRest.java CONDEC-108

FilteringManager.java CONDEC-47

Fig. 5.6: Connected Code-Classes view

5.4 Decision Grouping

Section 5.4.1 introduces the architecture of the requirements dashboard, by describing the relevant parts of the class diagram. The blue classes represented those implemented by the extension, while the yellow classes are either interfaces or already existing classes within the plug-in. Section 5.4.2 discusses the most important architectural decisions and Section 5.4.3 discusses decisions made in regard to the system functions introduced in Section 4.4.2. Lastly, Section 5.4.4 introduces the resulting views.

5.4.1 Architecture

Introducing decision groups into the ConDec plug-in allows users to sort their documented decisions according to any criteria they desire. These clusters of grouped decision can then easily be filtered allowing more targeted access to related decisions. This enables easier change impact analysis for the users, as related decisions are easier to find, even if they are not documented within the same issue or follow the same naming scheme. Similar to the code classes, groups are stored as database entries using the Active Object functionality of Jira. This database is defined by the *DecisionGroupInDatabase* class. The *DecisionGroupManager* class handles all the deletions, insertions and updates to the database. For visualization it is accessed by the *ConfigRest* and *KnowledgeRest* REST interfaces which supply information to the frontend, allowing filters to be used and providing information to the Decision Group View (WS3.7). Figure 5.7 shows the relevant class diagram.

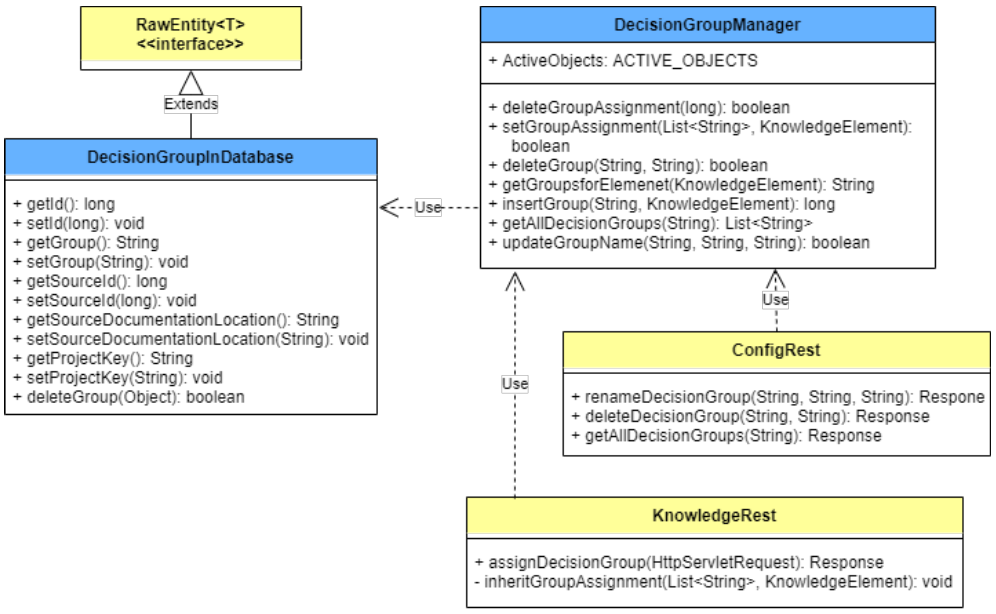

RawEntity<T>
<<interface>>
Extends
DecisionGroupInDatabase
+ getId(): long
+ setId(long): void
+ getGroup(): String
+ setGroup(String): void
+ getSourceId(): long
+ setSourceId(long): void
+ getSourceDocumentationLocation(): String
+ setSourceDocumentationLocation(String): void
+ getProjectKey(): String
+ setProjectKey(String): void
+ deleteGroup(Object): boolean
Use
DecisionGroupManager
+ ActiveObjects: ACTIVE_OBJECTS
+ deleteGroupAssignment(long): boolean
+ setGroupAssignment(List<String>, KnowledgeElement): boolean
+ deleteGroup(String, String): boolean
+ getGroupsforElemenet(KnowledgeElement): String
+ insertGroup(String, KnowledgeElement): long
+ getAllDecisionGroups(String): List<String>
+ updateGroupName(String, String, String): boolean
Use
Use
ConfigRest
+ renameDecisionGroup(String, String, String): Respone
+ deleteDecisionGroup(String, String): Response
+ getAllDecisionGroups(String): Response
KnowledgeRest
+ assignDecisionGroup(HttpServletRequest): Response
- inheritGroupAssignment(List<String>, KnowledgeElement): void

Fig. 5.7: Class Diagram: Decision Grouping

5.4.2 Architectural Decisions

Tables 5.7 and 5.8 reference the most important architectural decision problems (DP). Table 5.7 discusses the storage of decision groups within Jira. Table 5.8 discusses whether to implement a model class for decision groups. Note, that decisions are also valued by their contribution to the NFR's this thesis mainly focuses on (see Section 4.5).

Table 5.7: DP: How should the groups be stored in Jira?

Type	**Description**	**Modifiability**	**Time behavior**	**Attractiveness**
Decision	*Use a database of group entries*			
Pros	- Fast access		⊕	
	- Storage handled mostly by Jira	⊕		
Cons	- New Database required	⊖		
	- Handler for Database required	⊖		
Alternative	*Store them as attributes to knowledge elements*			
Pros	- No new classes required	⊕		
Cons	- Slower access if all groups are extracted		⊖	
	- Storage dependent on knowledge elements	⊖		

Table 5.8: DP: Should a new Group-Model-Class be created?

Type	**Description**	**Modifiability**	**Time behavior**	**Attractiveness**
Decision	*Declare Groups as Strings*			
Pros	- Simple and efficient storage	⊕	⊕	
	- Less complication	⊕		
Cons	- No additional attributes can be added			⊖
Alternative	*Create a model class for Groups*			
Pros	- Versatile	⊕		
	- Allows future expansion	⊕		
Cons	- Complex to handle	⊖		

5.4.3 System Function Related Decisions

SF8: Group Knowledge Elements
As discussed in the Review Discussion (Section 3.5) the limited amount of predetermined grouping schemes, increases the difficulty for the user to assign their decisions correctly and with a low enough effort. As such, it was decided to only implement this predetermination in a very limited manor. Users are required to select from one of three decision levels (High, Medium and Realization Level) when assigning a group but are otherwise free to assign groups with any name they desire. This allows some form of uniformity across the project, while keeping the grouping process relatively simple. It is also more versatile to assign custom groups for each project individually, rather then predetermining them universally.
Another decision that had to be made was how the inheritance of groups between related knowledge elements would work. Multiple alternatives existed for this. The first would be to not implement any inheritance what so ever and assign each element individually. This however would result in additional effort should a user require each related element to be of the same group and could lead to a loss of connection between elements as they would not automatically carry similar groups. The next alternative would be to implement universal inheritance. In this case, if one element of the knowledge tree was tagged with a group, all others would automatically be equipped as well. The problem with this approach however would be that, as individual elements are tagged with different groups, the tree could become overloaded with too many of them, reducing the value of the grouping mechanism. The third alternative, which was chosen for the project, was to implement a top-down inheritance scheme, in which all knowledge elements below another in the tree would inherit the group from the one above but not the other way around. This approach seemed to provide a good compromise between the other two alternatives.

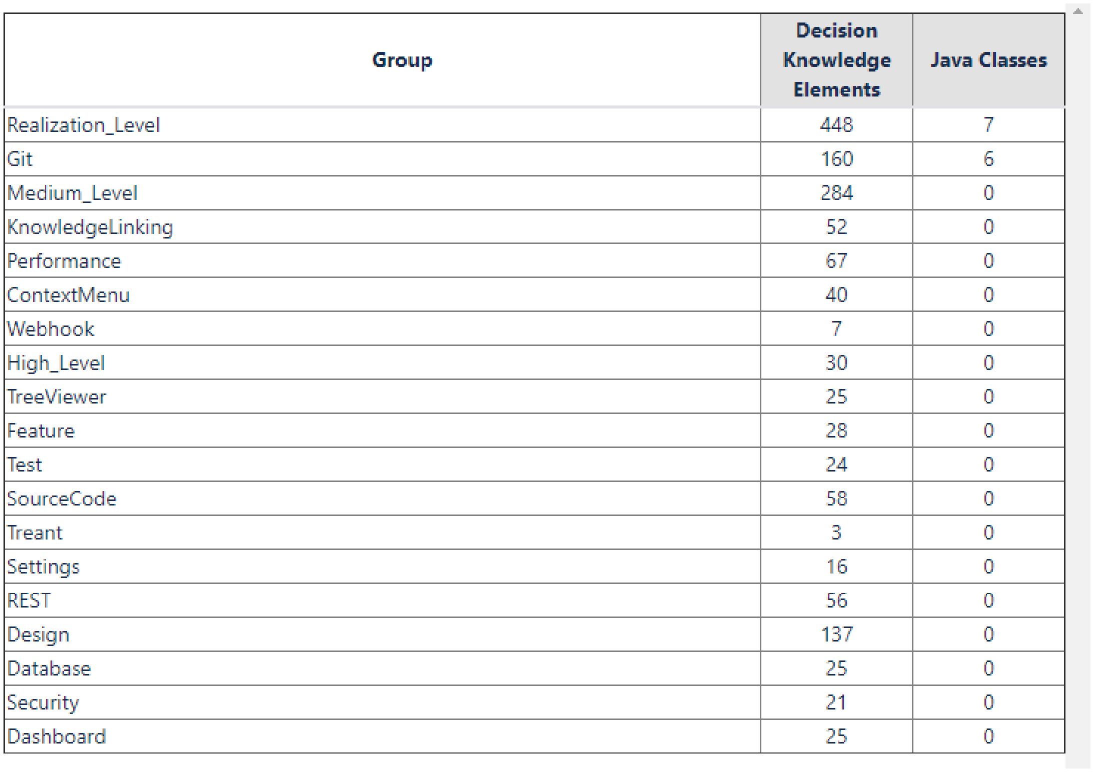
SF9: Manage knowledge element groups
The question of how to visualize the groups existing within a project was decided on based on the tasks of the related view (WS3.7). Since the views main task was the bulk deletion or renaming of existing groups, it was decided to provide a very simple clean look. The view, tasked with implementing this system function, merely presents a table listing the existing groups, with the connected decision knowledge elements and java classes. The editing of the group is done via the custom context menu already existing in ConDec.
Additionally, to keep the view as simple and clear as possible, instead of listing the keys of all connected issues, the table simply lists the number of knowledge elements and code classes connected to the group. This allows the users to easily assess how many elements are effected by their changes. The functionality of seeing all elements belonging to a group is still present in the Decision Knowledge Overview (WS3.1) and the Code Classes View (WS3.6) by using the provided group filter.

SF10: Filter knowledge graph
Lastly, the decision was made to allow the grouping filter to be combinable. This means that the user is able to search for a combination of groups at once. An example would

be all High level decision within the "Safety" group.

5.4.4 Results

Figure 5.8 shows the results of the Decision Group View (WS3.7) implementation and can be compared to the mockup in Figure 4.12 in Section 4.7. The view presents a table listing all decision groups that are currently assigned to at least one knowledge element. On the right side it shows the number of assigned decision knowledge elements and also the number of assigned Java classes. By right-clicking a group the user is presented with the possibility of either renaming the group or deleting it completely. By choosing these options, the change is automatically propagated to all assigned elements.



Group	Decision Knowledge Elements	Java Classes
Realization_Level	448	7
Git	160	6
Medium_Level	284	0
KnowledgeLinking	52	0
Performance	67	0
ContextMenu	40	0
Webhook	7	0
High_Level	30	0
TreeViewer	25	0
Feature	28	0
Test	24	0
SourceCode	58	0
Treant	3	0
Settings	16	0
REST	56	0
Design	137	0
Database	25	0
Security	21	0
Dashboard	25	0

Fig. 5.8: Decision Group View

6 Quality Assurance

This chapter describes the quality assurance process for the ConDec extension. It specifies the planing, execution and analysis of all tests done in relation to the extension. Section 6.1 specifies the general planing and structure of the different tests. Section 6.2 explains the static code testing that was part of the implementation. Next, Section 6.3 describes the component tests necessary for the ConDec extension. Lastly, Section 6.4 then introduces the system tests done in relation to the system functions of the extension.

6.1 Test Concept

To guarantee that the software is in a deployable state, ConDec uses a branching concept with the master branch being the mainline. Next to this branch, the development branch is a second branch containing new feature developments. From this branch, developers create individual feature branches. This means, that for every task tackled a developer creates a new branch in the Git repository. The branch is based on the "develop" default branch and commits are pushed to this branch instead of directly to the default branch. This simplifies the work for multiple developers working at the same time, by preventing merge conflicts. Additionally it allows individual testing of the feature branches by automated tools and code reviews, before they are merged into the develop branch at the end of implementation. The testing done on the ConDec project after the extensions implementation can be split into three groups. They consist of static code tests, component tests and system tests.
The static code tests were automated using Codacy[1], which is configured to run tests on every commit in the Git repository. Static code testing analyses the source code itself before any execution is done. This allows developers to find code smells and possible problems during execution, by using a set of code standards. The goal is to produce higher quality, more readable code and fix possible problems before more complex component and system tests are executed.
Component tests are designed to test the methods that make up ConDec's backend. For testing, JUnit[2] is used in a local environment. The Atlassian SDK used for development supports JUnit running at plug-in compilation time. Additionally, tests are executed within the Git-Repository. On commit to the relevant branch, a TravisCI[3] test runner is used to conduct the component tests and report errors. The component tests will

[1]Codacy: `https://www.codacy.com/` (Last access: 05.06.2020)
[2]JUnit: `https://junit.org/junit4/` (Last access: 05.06.2020)
[3]TravisCI: `https://travis-ci.org/` (Last access: 05.06.2020)

be measured by their success as well as the test coverage across all relevant classes. Currently, no frontend component tests are used within ConDec.
The system tests use predefined processes with fixed, expected results to assess proper function of the software on a surface level. Using step-by-step instructions, these tests are designed to arrive at a certain output. Any deviation from the expected output is noteworthy and must be assessed further. Thus, system tests will be measured by success or failure.
During this thesis, tests were conducted throughout the implementation process, mostly at the end of the relevant implementation tasks. Failing tests resulted in bug reports. These were resolved during the further implementation. Thus, all tests listed in this chapter were successful by the end of the implementation phase.

6.2 Static Code Tests

Codacy breaks the possible code issues during the static code testing down into six categories. *Security* issues can cause problems in the safety of the executed code. *Error Prone* issues are those that may hide bugs or use language keywords that should not be used. *Code style* issues do not follow the predefined styling rules designed to improve readability and comprehendability of the code. *Compatibility* issues are mostly frontend focused and consider problems like browser compatibility. *Unused Code* is dead code that can't be reached or is never actually called by the program and *Performance* issues concern drains on system resources. The most prevalent issues found during development are by far the code style issues as they are easily made and do not create problems with code execution. They are, however, also among the most easily fixed. In addition to the automated testing another measure is included in the static code testing in the form of a complexity value.
The complexity of a class can be a good indicator of the overall quality of the code. Code that is too complex can lead to multiple problems, including difficulty in the readability and thus analysability of code, as well as problems in the execution itself. Additionally, complex code is much harder to debug because of its complex traceability. This makes it important to measure the complexity of the source code. The tools used during the development of the ConDec extension provide a value for the complexity of the code in the form of the cyclomatic complexity. It is defined by McCabe et al. [38] as the minimum number of paths that can, in combination, generate all possible paths through a method. The more nested a method becomes, the higher its complexity value becomes, as more paths exist to navigate through the method. Formally, the cyclomatic complexity M of a method is based on the representation of the control flow graph of the piece of code in the form of a directed graph.

$$M = E - N + 2$$
$$\text{E = Number of Edges}$$
$$\text{N = Number of Nodes}$$

Additionally, the complexity value can serve as a measure, for how many test cases are needed in the component test-phase (Section 6.3) to ensure proper code coverage. The complexity calculation is automated by a tool called Codecov[4], which is included into the Git repository. Table 6.1 gives an overview over the lines of code and the complexity value of relevant classes.

Table 6.1: Cyclomatic Code Complexity

Class	**Lines of Code**	**Complexity**
ChartCreator	49	1
CodeClassPersistenceManager	323	17
Data	104	7
DecisionGroupManager	294	8
GitClient	1069	12
GitCodeClassExtractor	198	9
KnowledgeElement	565	4
KnowledgeRest	496	25
MetricCalculator	419	16
RequirementsDashboardItem	179	11
Treant	213	18
TreeViewer	261	9
ViewRest	430	6

6.3 Component Tests

The automatic component tests conducted by the continuous integration server Travis CI is configured to accept branches with a minimum test coverage delta of 65%. As such, the ConDec extension provides JUnit tests, that exceed this minimum coverage on all new classes. Additionally, it ensures that test coverage does not decrease on any classes that were altered or extended upon. The coverage is calculated by the percentage of code that the executed tests reach at least once. Like the complexity, the task of calculating test coverage is calculated using Codecov. Table 6.2 lists the relevant classes, the number of tests for each and their test coverage.

[4]Codecov: https://codecov.io/ (Last access: 05.06.2020)

Table 6.2: Component Test Coverage

Class	**# of Tests**	**Coverage**
ChartCreator	3	100%
CodeClassPersistenceManager	20	65.15%
Data	11	85.30%
DecisionGroupManager	28	82.84%
GitClient	31	68.35%
GitCodeClassExtractor	16	72.34%
KnowledgeElement	30	85.46%
KnowledgeRest	59	73.88%
MetricCalculator	9	65.24%
RequirementsDashboardItem	5	72.91%
Treant	23	84.62%
TreeViewer	19	82.76%
ViewRest	44	88.18%

6.4 System Tests

This section describes all system tests which relate to the system functions specified in Section 4.4.2. Please note, that in order to improve the readability of the Section, certain details were omitted. The post-conditions for the system tests were not included in the tables, as for almost all tests, these were unchanged. For more detailed conditions and additional details see Section 4.4.2 where the system functions are detailed. Additionally, for all test cases the precondition mandates, that at least one project has to exist within Jira.

SF1: Show basic knowledge metrics for project progress

Table 6.3 describes the system tests for SF1, including the preconditions, the exact test steps, and the expected result. These tests focus on the proper depiction of the basic metrics as well as the correct results within the metrics. Note that postconditions were omitted as nothing changes within the data in the presented visualizations.

Table 6.3: System Tests SF1: Show basic knowledge metrics for project progress

ID	Preconditions	Steps	Expected Result
ST1.1	At least one Jira issue exists with one comment	1. Select the existing Project 2. Select any issue type	The "# Comments per Jira issue" metric is shown and displays the issue with the comment
ST1.2	At least one Jira issue exists with one decision documented	1. Select the existing Project 2. Select any issue type	The "# Decisions per Jira issue" metric is shown and displays the issue with the decision
ST1.3	At least one Jira issue exists with one decision issue documented	1. Select the existing Project 2. Select any issue type	The "# Issues per Jira issue" metric is shown and displays the issue with the decision issue
ST1.4	At least one requirement and one code class exist in Jira	1. Select the existing Project 2. Select any issue type	The "# Requirements and Code Classes" metric is shown and displays both the requirement and the code class
ST1.5	At least one knowledge element exists from each documentation location (Commit, Jira issue, Code and Jira issue content)	1. Select the existing Project 2. Select any issue type	The "# Elements from Decision Knowledge Sources" metric is shown and displays the elements in the correct slice for their documentation location
ST1.6	No issues or knowledge elements exist	1. Select the existing Project 2. Select any issue type	All metrics are displayed with zero elements

SF2: Show rationale completeness metrics within decision knowledge

Table 6.4 describes the system tests for SF2, including the preconditions, the exact test steps, and the expected result. These tests focus on the proper depiction of the rationale completeness metrics as well as the correct results within the metrics.

Table 6.4: System Tests SF2: Show rationale completeness metrics within decision knowledge

ID	Preconditions	Steps	Expected Result
ST2.1	Two decisions issues exist; One of them has a decisions linked the other doesn't	1. Select the existing Project 2. Select any issue type	The "How many issues are solved by a decision?" metric displays the issue with the decision and the one without in blue and red respectively
ST2.2	Two decisions exist; One of them has an issue linked the other doesn't	1. Select the existing Project 2. Select any issue type	The "For how many decisions is the issue documented?" metric displays the decision with the issue and the one without in blue and red respectively
ST2.3	Two alternatives exist; One of them has a con argument linked the other doesn't	1. Select the existing Project 2. Select any issue type	The "How many alternatives have at least one con argument documented" metric displays the alternative with the con argument and the one without in blue and red respectively
ST2.4	Two decisions exist; One of them has a con argument linked the other doesn't	1. Select the existing Project 2. Select any issue type	The "How many decisions have at least one con argument documented" metric displays the decision with the con argument and the one without in blue and red respectively

ST2.5	Two alternatives exist; One of them has a pro argument linked the other doesn't	1. Select the existing Project 2. Select any issue type	The "How many alternatives have at least one pro argument documented" metric displays the alternative with the pro argument and the one without in blue and red respectively
ST2.6	Two decisions exist; One of them has a pro argument linked the other doesn't	1. Select the existing Project 2. Select any issue type	The "How many decisions have at least one pro argument documented" metric displays the alternative with the pro argument and the one without in blue and red respectively
ST2.7	No issues or knowledge elements exist	1. Select the existing Project 2. Select any issue type	All metrics are displayed with zero elements

SF3: Show completeness metrics of decision knowledge for requirements and tasks

Table 6.5 describes the system tests for SF3, including the preconditions, the exact test steps, and the expected result. These tests focus on the proper depiction of the selected issue type completeness metrics as well as the correct results within the metrics.

Table 6.5: System Tests SF3: Show completeness metrics of decision knowledge for requirements and tasks

ID	Preconditions	Steps	Expected Result
ST3.1	Two Work Item issues exist; One of them has an issue linked the other doesn't	1. Select the existing Project 2. Select the Work Item issue type	The "For how many Work Item types is an issue documented?" metric displays the Work Item with the issue and the one without in the blue and red slice respectively

ST3.2	Two Work Item issues exist; One of them has a decisions linked the other doesn't	1. Select the existing Project 2. Select the Work Item issue type	The "For how many Work Item types is a decision documented?" metric displays the Work Item with the decisions and the one without in the blue and red slice respectively
ST3.3	At least one Work Item exists with two comments; One comment is a decision knowledge element the other isn't	1. Select the existing Project 2. Select the Work Item issue type	The "Comments in JIRA issues relevant to Decision Knowledge" metric displays the Comment with the decision knowledge element and the one without in the blue and red slice respectively
ST3.4	One Work Item exists with a decision issue, a decision, an alternative and a pro or con argument linked	1. Select the existing Project 2. Select the Work Item issue type	The "Distribution of Knowledge Types" metric displays the correct amount of knowledge elements for each type
ST3.5	No issues or knowledge elements exist	1. Select the existing Project 2. Select any issue type	All metrics are displayed with zero elements

SF4: Navigate from plots in dashboard to respective knowledge element

Table 6.6 describes the system tests for SF4, including the preconditions, the exact test steps, and the expected result. These tests focus on establishing whether the navigation from both the box plots as well as the pie charts works as intended and leads to the correct issues.

Table 6.6: System Tests SF4: Navigate from plots in dashboard to respective knowledge element

ID	Preconditions	Steps	Expected Result
ST4.1	At least one Jira issue exists with one comment	1. Select the existing Project 2. Select any issue type 3. Click on the point within the "# Comments per Jira issue" metric 4. Click on the issue-key in the overlay	The browser opens a new tab with the issue module view of the selected issue
ST4.2	Two decisions exist; One of them has a con argument linked the other doesn't	1. Select the existing Project 2. Select any issue type 3. Click on one of the slices within the "How many decisions have at least one con argument documented" metric 4. Click on the knowledge element key in the overlay	The browser opens a new tab with the issue module view of the issue containing the knowledge element
ST4.3	No issues or knowledge elements exist	1. Select the existing Project 2. Select any issue type 3. Click on one of the slices within any pie chart	No overlay is opened

SF5: Configure decision knowledge extraction from Git

Table 6.7 describes the system tests for SF5, including the preconditions, the exact test steps, and the expected result. These tests focus on determining whether the metrics produce the correct results, whether the Git extraction is enabled or not.

Table 6.7: System Tests SF5: Configure decision knowledge extraction from git

ID	Preconditions	Steps	Expected Result
ST5.1	Two Work Items exist; One has an issue documented in the issue content and the other in a connected Git commit	1. Open the ConDec Git Connection Settings 2. Enable the (Extract from Git?) setting 3. Open the dashboard 4. Select the project and Work Item issue type	The "For how many Work Item types is an issue documented?" metric shows the two issues Work Items with an issue connected
ST5.2	Two Work Items exist; One has an issue documented in the issue content and the other in a connected Git commit	1. Open the ConDec Git Connection Settings 2. Disable the (Extract from Git?) setting 3. Open the dashboard 4. Select the project and Work Item issue type	The "For how many Work Item types is an issue documented?" metric shows only the Work Item with the issue in the issue content
ST5.3	Two Work Items exist; One has an issue documented in the issue content and the other in a connected Git commit	1. Open the ConDec Git Connection Settings 2. Enter an erroneous Git URI 3. Open the dashboard 4. Select the project and Work Item issue type	The metrics are correctly calculated as if the Git extraction was disabled

SF6: List all code classes for a project

Table 6.8 describes the system tests for SF6, including the preconditions, the exact test steps, and the expected result. These tests focus on the Code View (WS3.6) and the proper depiction of the extracted code classes.

Table 6.8: System Tests SF6: List all code classes for a project

ID	Preconditions	Steps	Expected Result
ST6.1	One code class exists and is connected to at least one Jira issue	1. Open the "Code Classes" view 2. Click on the code class on the left side of the view	The knowledge tree for the code class is opened, showing the connected issue
ST6.2	One code class exists in Jira and is connected to at least one Jira issue	1. Open the "Code Classes" view in the ConDec Decision Knowledge Page 2. Click on the code class on the left side of the view 3. Right-Click the code class element in the tree	The context menu opens and shows only the assign to decision group option

SF7: List all code classes connected to a Jira issue

Table 6.9 describes the system tests for SF7, including the preconditions, the exact test steps, and the expected result. These tests focus on the Connected Code-Classes View (WS2.5) and proper depiction of the extracted code classes in relation to the Jira issue.

Table 6.9: System Tests SF7: List all code classes connected to a Jira issue

ID	Preconditions	Steps	Expected Result
ST7.1	Git extraction setting is enabled One code class exists in Jira and is connected to at least one Jira issue	1. Open the "Connected Java-Classes" View	The code class is listed in the tree beneath the Jira issue
ST7.2	Git extraction setting is enabled One code class exists in Jira and is connected to at least one Jira issue	1. Open the "Connected Java-Classes" View 2. Right-click the code class in the tree	The context menu opens and shows only the assign to decision group option

SF8: Group Knowledge Elements

Table 6.10 describes the system tests for SF8, including the preconditions, the exact test steps, and the expected result. These tests focus on the grouping functionality across multiple views, making sure that the assignment and inheritance work as expected.

Table 6.10: System Tests SF8: Group Knowledge Elements

ID	Preconditions	Steps	Expected Result
ST8.1	At least one decision issue is documented with a linked decision	1. Open the Decision Knowledge Page 2. Click on the decision issue 3. Right-Click the decision issue in the tree 4. Select "Assign to Decision Group" 5. Assign a Group	The group is assigned to the decision issue and its related decision
ST8.2	At least one decision issue is documented and linked to a Jira issue; The Jira issue is connected to a code class	1. Open the "Code Classes" view in the Decision Knowledge Page 2. Click on the code class 3. Right-Click the Jira issue in the tree 4. Select "Assign to Decision Group" 5. Assign a Group	The group is assigned to the decision issue, the Jira issue and the code class
ST8.3	At least one decision issue is documented with a linked Jira issue	1. Open the "Tree Visualization" in the Jira issue module 2. Right-Click the decision issue in the tree 3. Select "Assign to Decision Group" 4. Assign a Group	The group is assigned to the decision issue

SF9: Manage knowledge element groups

Table 6.11 describes the system tests for SF9, including the preconditions, the exact test steps, and the expected result. These tests focus on the management capabilities of the Decision Groups View (WS3.7), namely the bulk renaming and deletion of groups.

Table 6.11: System Tests SF9: Manage knowledge element groups

ID	Preconditions	Steps	Expected Result
ST9.1	A decision group is assigned to atleast one knowledge element	1. Open the "Decision Groups" view in the Decision Knowledge Page 2. Right-click the decision group 3. Select the "Rename the Decision Group" feature 4. Assign a new name to the group and confirm	The new Group name is now displayed in the table while the old is gone
ST9.2	A decision group is assigned to atleast one knowledge element	1. Open the "Decision Groups" view in the Decision Knowledge Page 2. Right-click the decision group 3. Select the "Delete the Decision Group" feature Confirm the deletion	The group is gone for table and the knowledge element is no longer assigned to it

SF10: Filter knowledge graph

Table 6.12 describes the system tests for SF10, including the preconditions, the exact test steps, and the expected result. These tests focus on the new filtering capabilities introduced by the ConDec extension across different views. Note that for all filters tests with incorrect or invalid inputs were conducted. However, for special reasons and because they are all handled by HTML itself, they were omitted from this listing.

Table 6.12: System Tests SF10: Filter knowledge graph

ID	Preconditions	Steps	Expected Result
ST10.1	At least one code class exists that is assigned to two decision groups and one code class exists that has just one of the groups	1. Open the "Code Classes" View in the Decision Knowledge Page 2. Press on the "Select a Group" filter 3. Choose the groups that the class is assigned to	The code class that is part of both groups is shown on the left. The other one isn't
ST10.2	One code class exists that is connected to two Jira issues and one that is only connected to one	1. Open the "Code Classes" View in the Decision Knowledge Page 2. Set the "Linked Issue Min:" filter to 2	Only the code class with 2 connections is shown
ST10.3	One code class exists that is connected to two Jira issues and one that is only connected to one	1. Open the "Code Classes" View in the Decision Knowledge Page 2. Set the "Linked Issue Max:" filter to 1	Only the code class with 1 connection is shown
ST10.4	A code class exists with two connected Jira issues; One of the issues has connected knowledge elements, the other doesn't	1. Open the "Code Classes" View in the Decision Knowledge Page 2. Select the code class 3. Set the "Show Issues without Knowledge Elements" filter	Only the issue that has knowledge elements connected is shown in the tree
ST10.5	A Jira issue exists that has a code class and a test code class connected	1. Open the Connected Java-Classes View in the Jira issue module of the relevant issue 2. Set the "Show Test-Classes" filter 3. Press "Filter"	After pressing the button both classes are shown in the tree

7 Evaluation

This section presents and discusses the evaluation of the ConDec extension. Section 7.1 specifies the goals of the evaluation and explains its two separate parts, the survey and the evaluation on real data. Section 7.2 details the survey with feedback by fellow ConDec developers and Section 7.3 specifies the manual gold standard creation and subsequent evaluation on the created standard. Lastly, Section 7.4 discusses possible threats to the validity of the evaluation.

7.1 Evaluation Goals

The two main goals of this chapter are the evaluation of the acceptance of features implemented by the ConDec extension and the creation and evaluation of a gold standard, in particular, in terms of its time behaviour.

G1: Feature Acceptance by ConDec Users
The first part of the evaluation targeted the acceptance of the newly implemented features and views by ConDec users with different levels of user experience. To assess this acceptance a survey based on the Technology Acceptance Model (TAM) by Davis et al. [12] was used. The model split the acceptance into three subcategories. The first part is the *Perceived Ease of Use* of the software, which asks the user about the difficulty they perceived in interacting with the relevant software features. The second part focuses on the *Perceived Usefulness*, specifying how useful the feature was towards its intended use. The third part is the *Intention to Use*, which asks the user how likely future use of the feature is for them.

G2: Gold Standard Creation and Evaluation on real data
The second goal can be split into two different aspects. The creation of a gold standard dataset for the future evaluation of ConDec projects (G2a) and the evaluation of the time behaviour of the implemented features using this complex dataset, to allow proper evaluation (G2b).

G2a: Gold Standard Creation and Analysis
The gold standard creation serves two purposes. First, it is used to empirically determine the characteristics of a real project in terms of decision groups. The aim is to analyse this gold standard and describe characteristics such as how many realization-

level, medium-level and high-level decisions were documented in the project. The proper evaluation of time behaviour can only take place on a dataset that has a high enough complexity to be relevant for any performance-related concerns. Secondly, it allows future evaluations on the performance of possible features. By creating a dataset with a high level of interconnection between elements, the real performance of a plug-in and the way it handles bigger amounts of data is more realistically assessable.

G2b: Evaluation of Time Behaviour using real data
Using the dataset created by G2a the time behaviour of the implemented features and views can be assessed relatively realistically. For the assessment, time boundaries are specified, which, while subjective for every user, should allow an evaluation of the performance of the ConDec plug-in on larger datasets. Features meeting these criteria could require future improvement as their use is hindered by an excessive amount of loading time, thus possibly shying away developers from their use.

7.2 Acceptance Evaluation

In this section the acceptance of the ConDec extension's features are evaluated by fellow ConDec developers. The methodology of the evaluation and the results are presented and discussed. This fulfills the first goal (G1) of the evaluation.

Methodology

The survey used for the evaluation was split into five parts, corresponding to the new features implemented in the extension. The first part focused on the assignment of decision groups within Jira. The second part then asked users to manage these groups using the Decision Group View in WS3.7. Next users were asked to identify code classes relevant to a certain issue using WS2.5. Afterward they were tasked with extracting knowledge about code classes, in general, using the Code View WS3.6. Lastly, they were asked to use the new requirements dashboard to answer another set of questions. For all five categories users were asked questions according to the previously mentioned TAM and its three categories, *Perceived Ease of Use* (EoU), *Perceived Usefulness* (PU) and *Intention to Use* (ItU). For each TAM question answers had to be on a five scale between "complete rejection" and "total agreement". Additionally, for each question a field was provided to allow text input by the participants. This allowed them to give further explanations, report problems, or give additional feedback. Table 7.1 again lists the five categories of the survey and details the relevant tasks for the users.

Table 7.1: Evaluation Survey Categories and Tasks

ID	Category	Task
C1	Decision Group Assignment	Use the Decision Knowledge View within a Jira issue or the Decision Knowledge Page to assign groups and set a decision level for decisions documented by you.
C2	Manage Decision Groups	Use the Decision Group View to rename and delete at least one group.
C3	Identify relevant code classes	Use the Connected Java-Classes View and its filters to answer three questions about a Jira task. Q1: How many Java classes were created or changed because of the task ? Q2: How many classes are tests ? Q3: How many classes are connected to a maximum of two Jira issues?
C4	Code classes related to Jira issues	Use the Code Classes View to answer the following questions: Q1: How many code classes are connected to at least 25 Jira issues. Q2: Which Jira issue, connected to the "GitClient" code class, has no connected decision knowledge?
C5	Assess Completeness of Decision Knowledge	Use the Requirements Dashboard to answer the following questions: Q1: In which documentation location are most of the knowledge element documented ? Q2: How many decision issues are connected to a decision? Q3: How many work items do not have decisions linked to them ?

All evaluation questions that were part of the survey are listed in Table 7.2 with the relevant category they belong to and the part of the TAM they seek to answer.

Table 7.2: Evaluation Survey Questions

ID	**Cat.**	**TAM**	**Question**
EQ1	C1	EoU	It is easy to assign decision groups to elements
EQ2	C1	PU	The assigning of decision groups is useful
EQ3	C1	ItU	I will continue to assign decision groups in the future
EQ4	C1	PU	It is useful to force users to assign a decision level when assigning groups
EQ5	C2	EoU	It is easy to delete and rename groups
EQ6	C2	PU	The bulk renaming and deleting of groups is useful when working with decision groups
EQ7	C2	ItU	I will continue to use the Decision Group View in the future
EQ8	C3	-	Which of the questions were you able to answer without a problem ?
EQ9	C3	EoU	It is easy to find information about relevant code classes for each Jira issue
EQ10	C3	PU	The View is useful to visualize relationships between Jira issues and code classes
EQ11	C3	ItU	I will continue to use the view in the future
EQ12	C4	-	Which of the questions were you able to answer without a problem ?
EQ13	C4	PU	The existing filters suffice to specify the shown decision knowledge
EQ14	C4	EoU	It is easy to use the filters
EQ15	C4	EoU	It is easy to gain information about code classes and their related Jira issues and decision knowledge elements from this view.
EQ16	C4	PU	It is useful to visualize the relationships between Code, Jira issues and Knowledge elements like this
EQ17	C4	ItU	I will use this view in the future
EQ18	C5	-	Which of the questions were you able to answer without a problem ?
EQ19	C5	EoU	It is easy to find incomplete decision knowledge by using the dashboard
EQ20	C5	PU	It is useful to have the incompleteness displayed like this
EQ21	C5	ItU	I will continue to use the dashboard to find incomplete decision knowledge

Results and Discussion

The survey presented in the previous section was filled out by a total of five experienced ConDec developers over the span of a week. While an increased number of participants, especially from inexperienced users, would of course have allowed for a more thorough acceptance evaluation, the extend of the given feedback should suffice to create a general statement of the acceptance among people already familiar with the plug-in.
Figure 7.1 displays the levels of agreement to the TAM questions listed in Table 7.2. As previously mentioned acceptance was measured on a five-point scale, with a score of 1 implying total disagreement and 5 implying complete agreement with the given statement.

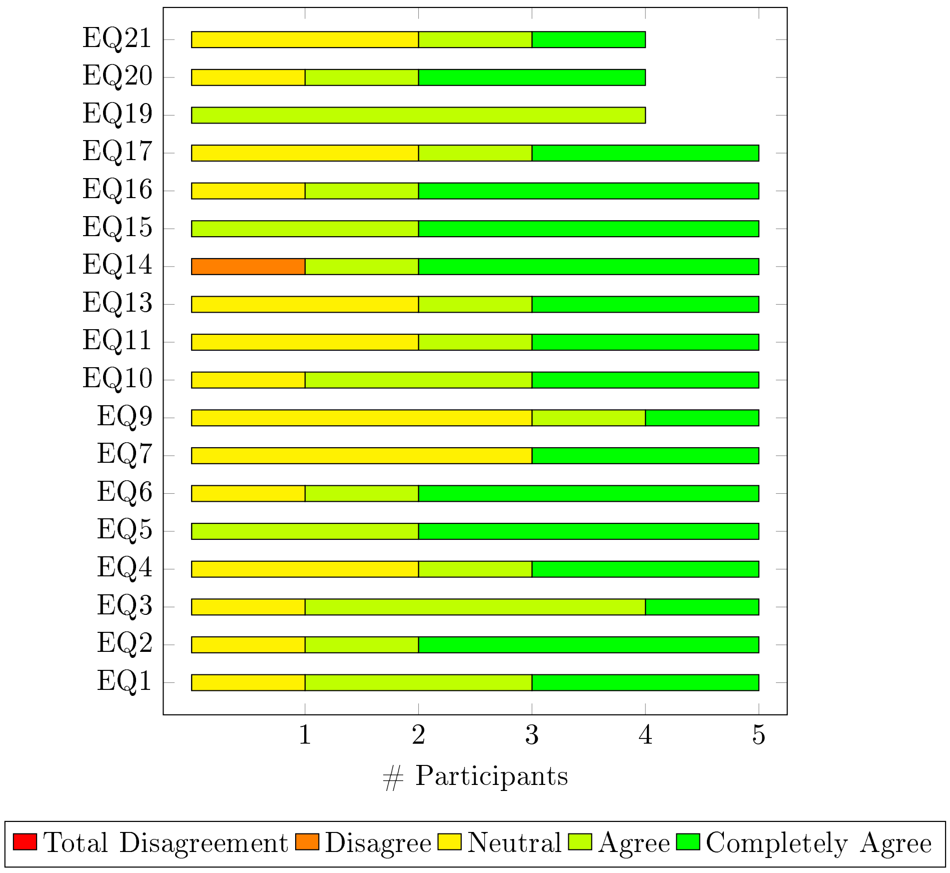


Fig. 7.1: Evaluation Survey Results

The following discussion will focus on the five different categories individually and consider the previously listed agreement scores as well as the manually written feedback by the evaluators.

C1: Decision Group Assignment (EQ1-EQ4)
The ease of use of the group assignment feature saw a generally high level of acceptance. Problems described by the evaluators included a general misunderstanding of the assignment dialogue context. The split between a text field, listing the currently

assigned decision groups, and a text field allowing new assignments seemed to not be intuitive enough. On the other hand others seemed to prefer this split. An information field next to the individual fields could help alleviate these problems while maintaining the split. Additionally, requests were made for features such as automatic substring completion and decision group recommendations which would allow easier use of the decision groups and increase consistency of the naming conventions across a project. This feature is also discussed in Section 8.2.
The usefulness of the grouping feature also gained general acceptance. However, one evaluator mentioned a possible constraint to the usefulness. Since groups can be assigned at will by developers, without any predefined groups, cases could occur in which the number of groups present within a project could become too great to offer any potential gains. This would render the relevant filters very complicated to use and thus have diminishing returns. As such, teams may need to pre-define some restraints for the assignment of decision groups.
Similar to both categories before, remarks were made about the intent of use. As a general idea the intention for future use seems to exist, however, as mentioned before, teams may need to discuss the use of decision groups and their naming schemes when multiple people are working at the same time in order to maintain consistency.
As discussed in Section 3.5 decision groups mandate the definition of a level ranging from realization to high level decisions. The idea was, to allow users a quick assignment of the level without forcing them to spend additional time looking up existing or thinking of new groups. The feedback on this mandate seems to be split between general agreement and uncertainty. It is no surprise, that a feature, which forces users into action will have split opinions. However, from the feedback it can be assessed that no strong opposition to the decision level mandate exists. As such, it is assumed that its benefits outweigh the additional work.

C2: Manage Decision Groups (EQ5-EQ7)
The ease of use for the decision group management view gathered a high level of agreement. This may in part be due to its intentionally simple design. However, some improvements were requested mostly concerning filtering options for projects with a high number of groups. As of right now, the simple display of these groups within a table should provide a certain level of clarity even with an increased amount of groups. A text filter however could guarantee that this clarity persists even as the number of decision groups rises further.
The evaluators agreed with the usefulness of the view. The only issues seemed to correspond to the group assignment feature (C1) in that recommendations for existing groups were desired to allow more consistent naming across each project.
The intent of use on the other hand received a higher level of uncertainty. This may in fact be the result of the roles developers take within a project. The view itself serves more towards the role of the rationale manager, tasked with maintaining consistency within the decision knowledge. Developers may experience a lack of interest in these administrative tasks.
Generally a desire was expressed to expand the grouping feature towards the Jira issue labels. Being able to assign groups by writing them into these labels or being able to edit them directly in the issue view, could help expand the use of the grouping feature.

A problem however occurs when the labels are used by developers for other purposes. This would require the use of prefixes to denote decision groups, which can become problematic when handling these Strings, which is why the feature was not implemented this way.

C3: Identify relevant code classes (EQ9-EQ11)
General feedback for the view in C3 was positive with some problems mostly concerning the size of the presented tree. These problems are mainly due to the limited amount of size available within the Jira issue module. As discussed before (Section 5.3.3), this limitation leads to compromises in the number of features being implementable. Allowing decision knowledge or other connected Jira issues to also be presented next to the code classes would result in a view, which was overloaded and could thus not be properly displayed. To address some of the requested features, including an expansion of the filtering system, and a general statistical overview (ex. listing the number of connected classes), a new view could be implemented in the Decision Knowledge Page, which would provide a complementary view to the already existing Code View. This new view would list Jira issues and display their complete knowledge tree, including connected issues, decision elements, and code classes. Such a feature may be part of future works on the plug-in.
The ease of use suffered from the spacial problems discussed before and as such received more uncertainty. Additional information was requested for the view, which could be presented in the possible new view discussed in the previous paragraph.
Both usefulness and intent to use received similarly high scores, despite the apparently more complex ease of use. This indicates that a desire for the provided feature exists as it provides information, that was previously missing from Jira and as such could find more popularity with future feature expansion.

C4: Code classes related to Jira issues(EQ13-EQ17)
The general feedback for the Code View evaluated by this category again shows that most problems with the previous category were a result of the limited size. Since a whole page is available to the Code View it does not suffer from these feature limitations.
The acceptance of the existing filters was generally high with an outlier in the ease of use, which unfortunately was not explained with textual feedback and as such was difficult to retrace.
The ease of use for the view as a whole was entirely positive. The main problems did lie in the desire for more filters (ex. filtering based on time), which is a valid concern across all of ConDec. While the focus on providing adequate filters is high, an expansion of the sets of filters provided could allow for even easier use and an increase in targeted knowledge.
The comments towards usefulness praise the view for providing information, which was previously not available within Jira, which is in line with the feedback given for C3.
The intent to use was constraint by two factors. The first being the necessity to use the view only in cases were actual changes to the code are required, pending a code review. The second constraint was a needed level of clarity within the view, as the used tree structure is at risk of easily being overloaded. The provided filters should help alleviate

this problem, however as discussed, expansion may be required.

C5: Assess Completeness of Decision Knowledge (EQ19-EQ21)
General feedback for the Requirements Dashboard talked about the excessive loading times of the view. This problem is discussed in detail in the Evaluation Section 7.3. The feedback regarding the asked questions (see Table 7.1), showed that a problem exists with the transparency of the features provided by the view. The questions could in part not be answered as users were unfamiliar with the features and information presented by the view. Similar to C2 this could be a result of the unfamiliarity with the role of the rationale manager. But information about available features could nonetheless be provided to the users with the help of text boxes or introductions.
According to evaluators the ease of use could be improved by providing more textual feedback on incompleteness in a project. A sort of backlog was recommended in which the inconsistencies are documented without the pie charts currently used.
The usefulness was praised as the view is currently the only way to assess the completeness across a whole project and as such provides information that was not previously present in this form.
The intent to use was once again diminished by the excessive loading times that a project as large as the used ConDec Jira Project brings for the number of calculations. Again this issue is discussed in more detail below (Section 7.3).
Lastly, note that the decreased number of participants in this category was due to an issue one of the evaluators experienced when using the dashboard for a certain type of Jira project. The problem was subsequently investigated and easily resolved.

7.3 Evaluation on real data

First, the methodology of the evaluation is explained. Afterward, the gold standard is analysed and described in detail, and experiences gained during the creation are discussed. The time behaviour of the newly implemented features is then assessed, using said gold standard.

Methodology

The evaluation on real data follows two steps. First a gold standard is created using real data in the form of the existing Jira ConDec project. Then this data is used to assess the time behaviour of the plug-ins new features.
The gold standard will use the existing ConDec Jira project as a basis. With a total of 577 Jira issues at the time of writing, the project should provide enough complexity to allow proper performance assessment later on. The project also contains all decisions documented, not only during this extension, but rather during all previous implementation phases by current and former ConDec developers. During the acceptance evaluation and the implementation of the extension's features, some decision groups were already

assigned. However, in order to properly assess the grouping feature in particular, a large amount of documented decisions will be equipped with a decision group, where that is appropriate. Lastly, some of the existing documentation will be improved upon in order to create the highest possible standard.
In order to assess the time behaviour of the newly implemented features, all views created during the ConDec extension, as well as the newly improved or implemented filters, will be tested on the created gold standard. This should provide a relatively realistic assessment of the runtime required during real, complex projects. Where appropriate, time constraints are defined, which the views or filters must not exceed. The results of this evaluation will be analysed and where possible improvements will be made in order to ensure high responsiveness of the existing views.

Gold Standard

Table 7.3 presents general statistics for the gold standard at the time of writing. These statistics may vary over time as the ConDec project is still ongoing and updates are being done constantly. However, the current state should suffice to properly assess the performance of the feature extensions. These statistics were extracted using different ConDec views, including the Requirements Dashboard, the Code View and the Decision Knowledge Overview.

Table 7.3: Gold Standard Statistics

Metric	**Number**
# of Jira issues	577
# of requirements	72
# of code classes	348
# of decision problems	187
# of decisions	190
# of knowledge elements Total	1171
# of knowledge elements from commits	208
# of knowledge elements from Jira issues	57
# of knowledge elements from code comments	90
# of knowledge elements from Jira issue content	816
# of grouped knowledge elements	762
# of decision groups	19

Table 7.4 provides a closer look at the decision groups and their distribution across the knowledge elements. It is important to note that one knowledge element may be assigned to an arbitrary number of decision groups but is only allowed to be part of one decision level. As such, the total number of assigned decision knowledge elements can be calculated from the sum of the decision levels. The Decision Groups View (WS3.7) was invaluable for these statistics as the exact information wanted is presented there.

Table 7.4: Gold Standard Decision Groups

Group Name	**# of Decision Elements**
High Level	30
Medium Level	284
Realization Level	448
ContextMenu	40
Dashboard	25
Database	25
Design	137
Feature	28
Git	160
KnowledgeLinking	52
Performance	67
REST	56
Security	21
Settings	16
SourceCode	58
Test	24
Treant	3
TreeViewer	25
Webhook	7

While creating the gold standards some additional experience was gained with the use of grouping as a whole. For instance, it quickly became clear that some elements, while not fitting into any existing groups, do not necessarily warrant their own group. These mostly include decision problems that ask questions about how to implement certain things. As such these could almost always be assigned to the Realization Level without requiring a custom group. This solidified the decision to implement the decision level aspect of the grouping feature, as even small decision problems could be assigned to an appropriate level without too much additional information being required. This could allow developers to use the grouping feature without having to spend extra time on the definition of groups when that is not necessarily warranted.
It was also almost uniquely the case that knowledge elements connected to a decision problem were fitting to the group that the problem was assigned to. The only exception to this rule were cases were the decisions for a problem were of a lower level than the problem itself. Questions about how to handle certain scenarios for example could be of a medium level with different decisions on a realization level. This enforces the decision to implement the top-down inheritance of groups (see decision in Section 5.4.3).
Experiments with using the grouping scheme by Kruchten et al., recommended in the Literature Review (Section 3.5), quickly made it apparent that these groups were too difficult to assign on a large scale. Too much information was needed to properly differentiate between the Property, Existence, or Executive Decisions. Information, that

was not easily obtainable by anyone but the original documenter. As such, the decision level system allowed a much easier assignment, while at the same time offering roughly the same sorting criteria. In order to use the Kruchten grouping scheme successfully, developers would have to be required to read the detailed explanation for every group and gather the information to assess which of the groups a knowledge element belongs to.
At the beginning of the grouping process there was some difficulty in finding appropriate names for the groups that were not too long but also self-explanatory. This always became easier as more decisions of the same group occurred. The increase in information allowed for more informed group names to be created and in this the bulk renaming allowed by the Decision Groups View (WS3.7) became invaluable. It allowed for multiple groups that were essentially the same to be merged more easily.
The main problem with the creation of groups was deciding whether a group was warranted at all. An example of this would be the "Treant" group. The group only has three attached knowledge elements, but since these elements are all part of the "Design" group they needed another tag to be able to differentiate from the other design-related decisions. Now, this could also be achieved by using the group filter in combination with the text filter, but decisions related to the "Treant" might not have the word specifically written down and would thus be lost. This struggle between finding groups that encompass enough decisions but also offer enough customization to find specific knowledge occurred multiple times. In a development cycle these groups should be discussed as a team to find the proper balance between groups that are too large to be useful, and groups that are too small to be warranted.

Time behaviour

The time behaviour evaluation of the ConDec extension was split according to the relevant system functions and their related views. Each was run on the gold standard created in the previous section. The necessary loading time is then evaluated and discussed in the context of the resulting usability of the feature. Note that SF1-3 were combined as they are present in the same Dashboard and SF4 was omitted as the only limiting factors are bandwidth and server responsiveness, while the navigation itself is almost instantaneous.

Requirements Dashboard

SF1: Show basic knowledge metrics for project progress, SF2: Show rationale completeness metrics within decision knowledge, SF3: Show completeness metrics of decision knowledge for requirements and tasks
The original idea for the requirements dashboard was to set a fixed time, that the board was not able to surpass when loading and calculating. However, the targeted time of 15 seconds, was almost immediately passed, when more than the general statistics metrics were implemented. Using the gold standard, the loading time across multiple tries averaged to about 63 seconds. This time was only achieved after multiple iterations of cost-cutting measures and efficiency improvements. There are multiple problems with

the loading times of the dashboard. The sheer amount of Jira issues within the standard and its high interconnectivity, with links between the issues, will necessarily result in longer loading times for complex metrics. Additionally, the extraction of knowledge elements from the Git repository requires a complex set of requests using the Git client, which also increases loading times. The addition of the link distance filter increases the required complexity when calculating metrics by a significant factor, as for every issue within the project, an unspecified amount of additional issues have to be considered. As a result the link distance filter had to be limited to metrics where this increase in issue considerations would not result in loading times of multiple minutes. Multiple far-reaching changes would need to be made to the storage of knowledge elements within ConDec to facilitate a faster loading time. This includes the way information extracted from Git is stored, additional information about the documentation location is captured, and a more complex connection system between knowledge elements stored in different locations. Some attempts were made at reducing the loading times further. But caching of dashboards causes new problems, including outdated information, changing settings and an inability to store all filter eventualities, since filtering and calculation are done at the backend of the plug-in. A frontend based complete rebuild of the dashboard would be required without any guarantee of improvement, because of the additional complexity of REST requests and JavaScript coding.

Requirements Dashboard without Git extraction
SF5: Configure decision knowledge extraction from Git
By disabling the Git knowledge extraction within ConDec the loading time of the dashboard can be reduced significantly to an average of about 41 seconds. This indicates, that about a third of the loading time is required to extract all needed knowledge from the Git repository. This ties in with the previous comments about a needed improvement on the information about code-related knowledge elements being stored on the Jira server.

Code Classes View
SF6: List all code classes for a project
The loading time of the Code Classes View is hard to assess by itself, as all views within the Decision Knowledge Page are loaded when opening the page. That being said, the loading time of the complete page averages to about 7 seconds. Since the code classes are stored in a database on the server itself, loading times are expected to be very short (between 1-2 seconds). Building of the visualized trees itself is also almost instantaneous. The advantages of being able to switch between the seven different views within the Decision Knowledge Page should outweigh the cost of loading all pages at the same time.

Connected Java-Classes View
SF7: List all code classes connected to a Jira issue
Similar to the previous paragraph, it is hard to estimate an exact loading time for the Connected Java-Classes View within the Jira module, as all elements are loaded when opening the page. However, the tree visualization, when opening the section takes about

1 second. A short loading time is expected, as knowledge elements are stored within the RAM storage of the server, access to which, should be almost instantaneous, leaving only the visualization to take up a noticeable amount of time.

Decision Grouping
SF8: Group Knowledge Elements
The grouping of knowledge elements requires about 6 seconds from the pressing of the assign button until the view is completely usable again. This prolonged time, however, is not due to the storage of the assignment, but rather because the relevant view is reloaded afterwards to facilitate the changed grouping and keep the filters up to date. While normally this should not be an issue, during the creation of the gold standard this refresh prolonged the needed time considerably, because hundreds of group assignments had to be made. For cases like this, a setting would have been advantageous, which allows a user to decide whether sites should refresh after an assignment or not.

Decision Groups View
SF9: Manage knowledge element groups
The bulk renaming and deletion of groups averages to about the same time, both requiring on average 5 seconds. The size of the group seemed to present no discernible difference in this case. A group with one knowledge element was renamed just as fast as a group with 137 connected knowledge elements. 5 seconds seems an appropriate time for a task which is normally not conducted all that often.

Filters within Views
SF10: Filter knowledge graph
Since filters are uniformly implemented across ConDec, evaluation will take place in a singular view. The Code Classes view provides 348 elements with differing tree complexities and should thus provide the best basis for testing.
The Group filter requires about 8 seconds on average to be applied, with the combination of groups reducing this time significantly as fewer elements have to be considered every time.
The "Linked Issues" filter takes up a little less time, averaging to about 7 seconds, without any difference to whether both min and max or only one of the two was changed.
Lastly, the "Show Issues without Knowledge Elements" filter works similarly to the previous filter, taking about 7 seconds to apply. No difference was measured between classes with few (0-1) and classes with many (10+) connected issues.
The filters in general however feel unresponsive as the view freezes for the duration of loading without any indication that work is being done in the background.

All in all, while, with the exception of the requirement dashboard, no view exceeds loading times of 8 seconds, performance as a whole needs to be a bigger priority within ConDec in general and this extension specifically. Future work should aim to not only reduce these loading times but also make the features feel more responsive. An effort needs to be made to inform the user that calculations are currently being done. As of right now, in some scenarios, inexperienced users might feel like the site has frozen, which creates frustrations when using the plug-in. Time behaviour as a whole seems passable but not in any way admirable.

7.4 Threats to Validity

Possible threats to validity are identified by using the guidelines put forward by Runeson et al. [45]. According to Runeson et al. a study must meet four criteria of validity. To guarantee a high standard studies must meet construct validity, internal validity, external validity, and reliability.

Construct Validity expresses the disparity between what the researcher has in mind and what is really investigated according to the used research questions. In the case of the ConDec extension, it represents how well the proposed survey questions provide an assessment of the implemented features. To guarantee that this criterion is met, the scientifically recognized Technology Acceptance Model was used to design the survey.

Internal Validity analyses how a researched element can be affected by other factors, the existence or effect of which the researcher is not aware of. In the context of ConDec this could for example be a differing level of knowledge of the evaluators during the survey. To combat this, only experienced ConDec developers are used and the individual tasks of the survey are explained in detail prior to execution. The downside of using experienced users is that individual opinions can already be formed on which the researcher has no influence or knowledge of.

External Validity considers to what extend third parties are interested in the findings of the study and how well the results can be generalized. Considering the gold standard, a possible threat could be the size of the provided data, because it might be dwarfed by industrial-sized projects. This criterion might also affect the relatively small size of participants in both the development of ConDec and the participation in the survey.

Reliability expresses how dependent the results of the study are on the researches conducting it. Results should always be replicable by third parties at a later date. The results in case of this thesis might be affected by two factors in this criterion. First, the fact that the survey and evaluation are conducted by its own developer, which could cause a bias towards a positive judgement. This is combated somewhat by presenting the raw unaltered results of the survey. The second factor could be that evaluators were exclusively people working or having previously worked within the ConDec project, which might influence their objectivity. Unfortunately, this somewhat stands in contrast with the *Internal Validity* as it requires a set of somewhat experienced users, to guarantee the value of the posted answers.

8 Conclusion

Section 8.1 provides a summary of the work presented in this thesis and the achieved results. Section 8.2 presents a discussion of the thesis results and provides possible future improvements to the presented ConDec extension as well as recommendations for new features.

8.1 Summary

Unfortunately, decision knowledge is often only documented informally resulting in a loss of information as a software project is continuously developed. This thesis aimed to provide adequate tool support for the explicit, formal documentation of decision knowledge. In this it considers two roles within the software development process. The rationale manager is tasked with maintaining consistency within the documented decision knowledge and managing the documentation process. Developers are tasked with documenting all aspects of the decision-making process, including the decision problem, alternatives, pro and con arguments, and alternatives.
To support these roles, the ConDec project offers, among other tools, a Jira plug-in. The plug-in provides documentation possibilities within Jira in the form of Jira issues, comments, descriptions, commit messages, and code comments. The focus of this thesis was to provide new visualizations for decision knowledge relationships, in particular, by grouping related knowledge elements and by integrating code classes into visualization. A literature review finds answers to the questions about how decision knowledge can be grouped adequately and which grouping schemes are favourable for such a task. The result of said review is that a decision level based grouping scheme in combination with the creation of custom groups by developers at the time of documentation offers the most flexibility and low time effort, while maintaining the advantages of group filtering. Implementation wise, the ConDec extension provided by this thesis offers new views, which visualize the relationships between Jira issues, decision knowledge and code classes, to support developers. Developers are also able to assign groups to knowledge elements, bulk-manage these groups, and use filters to specify their displayed decision knowledge accordingly. The rationale managers are provided with a requirements dashboard offering general overviews over project statistics, rationale completeness metrics, and Jira issue type focused decision completeness metrics.
An evaluation of the implemented features in the form of a survey filled out by fellow ConDec developers and users, provided generally positive feedback in all categories of the Technology Acceptance Model, ease of use, perceived usefulness and intention to use.

During the course of the manual evaluation a gold standard was created, using the ConDec Jira project as a basis. The gold standard contains 577 Jira issues with varying levels of interconnectivity and 1171 knowledge elements, 762 of which are equipped with any of the 19 resulting decision groups. The evaluation of the time behaviour of the provided features, on said gold standard, showed acceptable loading times and responsiveness, but also highlighted a need for more focus towards performance and user experience within the ConDec plug-in.

8.2 Discussion & Future Work

While both the survey and manual evaluation offer positive feedback, some problems remain within the implementation provided by this thesis.
Evaluators expressed a desire for additional visualization of the connection between decision knowledge and code classes, specifically within the Jira issue module view. Problems arise here because of the limited horizontal space available to this view. The used tree representation requires a lot of this space and increasing the amount of represented elements further could lead to an overloaded view, which reduces its usability. Additionally, filter opportunities could be increased even further as the number of elements present in large scale projects can become confusing. Filters offer users more customizability but can in some cases have adverse effects on time behaviour as pages need to be loaded multiple times and additional calculation can be required. Providing even more filters in all ConDec views should be part of the focus for future ConDec developments, as the feedback has made a desire for these functionalities clear.
Concerning the grouping feature, improvements are pending, which could increase the ease of use further by providing immediate quality of live improvements during the assignment process. Text completion and drop-down menus providing lists of existing groups could help with maintaining consistency across the project and make assignments easier for developers, increasing the likelihood of thorough group assignment at the time of documentation by each developer. Additionally, more complicated systems, like automated group recommendations would also provide big improvements for usability. By analysing the knowledge element and its connected elements, such an automated system could extract possible group names and list these for the users to choose from.
Because of the ever-changing requirements during a software project's lifespan, decisions are also evolving throughout the process. Decision knowledge documented at the beginning of the development process can become outdated. Such knowledge has to be modified to stay relevant. However, during this transformation, knowledge about the reasoning of past decisions might get lost. To trace the evolution of a decision throughout the process, future work could implement an evolutionary view for these changes. Comparing these different versions, could find previous mistakes or retrace the results of a change within the project.
Such a view would also help with another issue, that currently affects the decision documentation process. Whether through change or deletion of knowledge elements, in particular of code classes, existing links between elements can become erroneous. These links are hard to maintain manually and lead to possible wrong conclusions in future

analysations. Maintaining correctness within the links is another possible application for ConDec, given proper feature support. This form of tangled change recognition is currently not supported within Jira and has to be manually executed for all important changes. A view that finds possible tangled changes and wrong links between elements by analysing changes within the project would provide much-needed aid for this project maintenance task.
Overall a focus on the change impact analysis could provide a rich basis for future extensions of the ConDec project as decision documentation can and should become a part of this process.

Bibliography

[1] M. J. Albers, Decision making: A missing facet of effective documentation, in *14th Annual International Conference on Systems Documentation: Marshaling New Technological Forces: Building a Corporate, Academic, and User-Oriented Triangle*, ser. SIGDOC '96, Research Triangle Park, North Carolina, USA: ACM, 1996, pp. 57–65. DOI: 10.1145/238215.238256.

[2] C. Baudin, C. Sivard, & M. Zweben, Recovering rationale for design changes: A knowledge-based approach, in *International Conference on Systems, Man, and Cybernetics Conference Proceedings*, IEEE, 1990, pp. 745–749. DOI: 10.1109/ICSMC.1990.142220.

[3] M. Bhat, K. Shumaiev, A. Biesdorf, U. Hohenstein, & F. Matthes, Automatic extraction of design decisions from issue management systems: A machine learning based approach, in *Software Architecture*, Cham: Springer International Publishing, 2017, pp. 138–154. DOI: 10.1007/978-3-319-65831-5_10.

[4] M. Bhat, K. Shumaiev, & F. Matthes, Towards a framework for managing architectural design decisions, in *11th European Conference on Software Architecture: Companion Proceedings*, Canterbury, United Kingdom: ACM, 2017, pp. 48–51. DOI: 10.1145/3129790.3129799.

[5] B. Bruegge & A. H. Dutoit, Object-Oriented Software Engineering Using UML, Patterns, and Java, 3rd. USA: Prentice Hall Press, 2009. DOI: 10.5555/1795808.

[6] J. E. Burge & D. C. Brown, Software engineering using rationale, *Journal of Systems and Software*, vol. 81, no. 3, 395–413, ELSEVIER, 2008. DOI: https://doi.org/10.1016/j.jss.2007.05.004.

[7] R. Capilla, F. Nava, & J. C. Duenas, Modeling and documenting the evolution of architectural design decisions, in *2nd Workshop on Sharing and Reusing Architectural Knowledge Architecture, Rationale, and Design Intent*, USA: IEEE, 2007, p. 9. DOI: 10.1109/SHARK-ADI.2007.9.

[8] J. Carriere, R. Kazman, & I. Ozkaya, A cost-benefit framework for making architectural decisions in a business context, in *32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, Cape Town, South Africa: ACM, 2010, pp. 149–157. DOI: 10.1145/1810295.1810317.

[9] M. Che, An approach to documenting and evolving architectural design decisions, in *35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 1373–1376. DOI: 10.1109/ICSE.2013.6606720.

[10] M. Che & D. E. Perry, Scenario-based architectural design decisions documentation and evolution, in *18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, IEEE, 2011, pp. 216–225. DOI: 10.1109/ECBS.2011.16.

[11] B. Dagenais & M. P. Robillard, Creating and evolving developer documentation: Understanding the decisions of open source contributors, in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Santa Fe, New Mexico, USA: ACM, 2010, pp. 127–136. DOI: 10.1145/1882291.1882312.

[12] F. D. Davis, R. P. Bagozzi, & P. R. Warshaw, User acceptance of computer technology: A comparison of two theoretical models, *Management Science*, vol. 35, no. 8, 982–1003, INFORMS, 1989. DOI: 10.1287/mnsc.35.8.982.

[13] D. Dermeval, J. Pimentel, C. Silva, J. Castro, E. Santos, G. Guedes, M. Lucena, & A. Finkelstein, STREAM-ADD - supporting the documentation of architectural design decisions in an architecture derivation process, in *36th Annual Computer Software and Applications Conference*, IEEE, 2012, pp. 602–611. DOI: 10.1109/COMPSAC.2012.81.

[14] A. Dragomir, H. Lichter, & T. Budau, Systematic architectural decision management, a process-based approach, in *IEEE/IFIP Conference on Software Architecture*, IEEE, 2014, pp. 255–258. DOI: 10.1109/WICSA.2014.39.

[15] Z. Durdik, A. Koziolek, & R. H. Reussner, How the understanding of the effects of design decisions informs requirements engineering, in *2nd International Workshop on the Twin Peaks of Requirements and Architecture*, IEEE, 2013, pp. 14–18. DOI: 10.1109/TwinPeaks.2013.6614718.

[16] A. H. Dutoit, R. McCall, I. Mistrík, & B. Paech, Rationale management in software engineering: Concepts and techniques, in *Rationale Management in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–48. DOI: 10.1007/978-3-540-30998-7_1.

[17] K. D. Evensen, Reducing uncertainty in architectural decisions with AADL, in *44th Hawaii International Conference on System Sciences*, IEEE, 2011, pp. 1–9. DOI: 10.1109/HICSS.2011.358.

[18] R. Farenhorst & H. van Vliet, Understanding how to support architects in sharing knowledge, in *ICSE Workshop on Sharing and Reusing Architectural Knowledge*, IEEE, 2009, pp. 17–24. DOI: 10.1109/SHARK.2009.5069111.

[19] B. Fitzgerald & K.-J. Stol, Continuous software engineering and beyond: Trends and challenges, in *1st International Workshop on Rapid Continuous Software Engineering*, Hyderabad, India: ACM, 2014, pp. 1–9. DOI: 10.1145/2593812.2593813.

[20] ——, Continuous software engineering: A roadmap and agenda, *Journal of Systems and Software*, vol. 123, 176–189, ELSEVIER, 2017. DOI: https://doi.org/10.1016/j.jss.2015.06.063.

[21] S. Gupta, S. Malik, L. Pollock, & K. Vijay-Shanker, Part-of-speech tagging of program identifiers for improved text-based software engineering tools, in *21st International Conference on Program Comprehension (ICPC)*, IEEE, 2013, pp. 3–12. DOI: 10.1109/ICPC.2013.6613828.

[22] U. van Heesch, P. Avgeriou, & R. Hilliard, A documentation framework for architecture decisions, *Journal of Systems and Software*, vol. 85, no. 4, 795–820, ELSEVIER, 2012. DOI: https://doi.org/10.1016/j.jss.2011.10.017.

[23] T. Hesse & B. Paech, Supporting the collaborative development of requirements and architecture documentation, in *3rd International Workshop on the Twin Peaks of Requirements and Architecture*, 2013, pp. 22–26. DOI: 10.1109/TwinPeaks-2.2013.6617355.

[24] T.-M. Hesse, A. Kuehlwein, B. Paech, T. Roehm, & B. Bruegge, Documenting implementation decisions with code annotations, in *SEKE*, 2015, pp. 152–157. DOI: 10.18293/SEKE2015-084.

[25] ISO/IEC, "Iso/iec 25010 system and software quality models," Tech. Rep., 2010. DOI: 10.3403/30215101.

[26] A. Jansen, "Architectural design decisions," Ph.D. dissertation, University of Groningen, 2008.

[27] A. Jansen & J. Bosch, Software architecture as a set of architectural design decisions, in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, IEEE, 2005, pp. 109–120. DOI: 10.1109/WICSA.2005.61.

[28] M. Jiménez, L. F. Rivera, N. M. Villegas, G. Tamura, H. A. Müller, & N. Bencomo, An architectural framework for quality-driven adaptive continuous experimentation, in *Joint 4th International Workshop on Rapid Continuous Software Engineering and 1st International Workshop on Data-Driven Decisions, Experimentation and Evolution*, Montreal, Quebec, Canada: IEEE, 2019, pp. 20–23. DOI: 10.1109/RCoSE/DDrEE.2019.00012.

[29] R. A. Kamaludeen, Y. Cheah, & S. Sulaiman, Software maintenance expert base decision support (soxdes) framework, in *International Conference on Advanced Computer Science Applications and Technologies*, IEEE, 2013, pp. 25–30. DOI: 10.1109/ACSAT.2013.13.

[30] A. Kleebaum, J. O. Johanssen, B. Paech, R. Alkadhi, & B. Bruegge, Decision knowledge triggers in continuous software engineering, in *4th International Workshop on Rapid Continuous Software Engineering*, Gothenburg, Sweden: ACM, 2018, pp. 23–26. DOI: 10.1145/3194760.3194765.

[31] A. Kleebaum, J. O. Johanssen, B. Paech, & B. Bruegge, Teaching rationale management in agile project courses, *16. Workshop Software Engineering im Unterricht der Hochschulen (SEUH)*, CEUR-WS.org, 2019. DOI: https://doi.org/10.11588/heidok.00026358.

[32] ——, Continuous management of requirement decisions using the condec tools, *Co-Located Events of the 26th International Conference on Requirements Engineering (REFSQ-JP'20)*, CEUR-WS.org, 2020. DOI: 10.11588/HEIDOK.00028230.

[33] P. Kruchten, An ontology of architectural design decisions in software-intensive systems, in *2nd Groningen workshop on software variability*, University of Groningen, Johann Bernoulli Institute for Mathematics & Computer Science, 2004.

[34] Kunz & H. Rittel, Issues as elements of information systems, *Working Paper 131*, Berkely: Institute of Urban & Regional Development, University of California, 1970.

[35] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, & W. M. Turski, Metrics and laws of software evolution-the nineties view, in *4th International Software Metrics Symposium*, IEEE, 1997, pp. 20–32. DOI: 10.1109/METRIC.1997.637156.

[36] R. Lougher & T. Rodden, Supporting long-term collaboration in software maintenance, in *Conference on Organizational Computing Systems*, Milpitas, California, USA: ACM, 1993, pp. 228–238. DOI: 10.1145/168555.168581.

[37] C. Manteuffel, D. Tofan, H. Koziolek, T. Goldschmidt, & P. Avgeriou, Industrial implementation of a documentation framework for architectural decisions, in *IEEE/IFIP Conference on Software Architecture*, IEEE, 2014, pp. 225–234. DOI: 10.1109/WICSA.2014.32.

[38] T. McCabe, Cyclomatic complexity and the year 2000, *IEEE Software*, vol. 13, no. 3, 115–117, IEEE, 1996. DOI: 10.1109/52.493032.

[39] C. Miesbauer & R. Weinreich, Classification of design decisions – an expert survey in practice, in *Software Architecture*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 130–145. DOI: 10.1007/978-3-642-39031-9_12.

[40] J. A. Miller, R. Ferrari, & N. H. Madhavji, An exploratory study of architectural effects on requirements decisions, *Journal of Systems and Software*, vol. 83, no. 12, 2441–2455, ELSEVIER, 2010. DOI: 10.1016/j.jss.2010.07.006.

[41] N. Ngadiman, S. Sulaiman, & W. M. N. Wan Kadir, A systematic literature review on attractiveness and learnability factors in web applications, in *IEEE Conference on Open Systems (ICOS)*, IEEE, 2015, pp. 22–27. DOI: 10.1109/ICOS.2015.7377272.

[42] P. Petrov, R. L. Nord, & U. Buy, Probabilistic macro-architectural decision framework, in *2014 European Conference on Software Architecture Workshops*, ser. ECSAW '14, Vienna, Austria: ACM, 2014. DOI: 10.1145/2642803.2642830.

[43] N. A. Prasetyo & Y. Bandung, A design of software requirement engineering framework based on knowledge management and service-oriented architecture decision (soad) modeling framework, in *International Conference on Information Technology Systems and Innovation (ICITSI)*, IEEE, November 2015, pp. 1–6. DOI: 10.1109/ICITSI.2015.7437708.

[44] R. Raman & M. D'Souza, Learning framework for maturing architecture design decisions for evolving complex sos, in *13th Annual Conference on System of Systems Engineering (SoSE)*, IEEE, 2018, pp. 350–357. DOI: 10.1109/SYSOSE.2018.8428733.

[45] P. Runeson, M. Host, A. Rainer, & B. Regnell, Case Study Research in Software Engineering: Guidelines and Examples. Wiley Publishing, 2012. DOI: 10.5555/2361717.

[46] M. Seiler & B. Paech, Documenting and exploiting software feature knowledge through tags, in *31st International Conference on Software Engineering and Knowledge Engineering*, KSI Research Inc. & Knowledge Systems Institute Graduate School, 2019. DOI: 10.18293/seke2019-109.

[47] M. Shahin, P. Liang, & M. R. Khayyambashi, Architectural design decision: Existing models and tools, in *Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture*, IEEE, 2009, pp. 293–296. DOI: 10.1109/WICSA.2009.5290823.

[48] J. Tyree & A. Akerman, Architecture decisions: Demystifying architecture, *IEEE Software*, vol. 22, no. 2, 19–27, IEEE, 2005. DOI: 10.1109/MS.2005.27.

[49] J. S. van der Ven & J. Bosch, Making the right decision: Supporting architects with design decision data, in *Software Architecture*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 176–183. DOI: 10.1007/978-3-642-39031-9_15.

[50] J. Webster & R. T. Watson, Analyzing the past to prepare for the future: Writing a literature review, *MIS Quarterly*, vol. 26, no. 2, xiii–xxiii, Management Information Systems Research Center, University of Minnesota, 2002. DOI: 10.2307/4132319.

[51] Zhe Huang & Yun-Quan Hu, Applying ai technology and rough set theory to mine association rules for supporting knowledge management, in *International Conference on Machine Learning and Cybernetics (IEEE Cat. No.03EX693)*, vol. 3, IEEE, 2003, 1820–1825 Vol.3. DOI: 10.1109/ICMLC.2003.1259792.

[52] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, & N. Schuster, Reusable architectural decision models for enterprise application development, in *Software Architectures, Components, and Applications*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 15–32. DOI: 10.1007/978-3-540-77619-2_2.

[53] O. Zimmermann, J. Koehler, F. Leymann, R. Polley, & N. Schuster, Managing architectural decision models with dependency relations, integrity constraints, and production rules, *Journal of Systems and Software*, vol. 82, 1249–1267, ELSEVIER, 2009. DOI: 10.1016/j.jss.2009.01.039.

Glossary

Alternative A proposed possible solution for a decision problem. 3, 7, 8, 47

Architectural Decision Decision made regarding the underlying architecture of the software. 11, 47, 48, 55, 62

Backend Aspects of the software that concern calculations and server-side processes. 47, 49, 91

Backward Snowballing Using the references of a given paper to find older, related articles. 13

Bug Defect within the code, causing unwanted behaviour. 67

Code Smell Characteristic within source code indicating a deeper problem or flaw. 66

Commit Used here in the context of Git. Individual change to a file or set of files. First occurrence on p. 2

Continuous Software Development Software development process designed to deliver functional products in short development cylces with high integration of user feedback. 4, 7

Decision Knowledge All existing knowledge related to a decision problem. Includes alternatives, pro and contra arguments and the decision itself. Also called Rationale in this context. 1, 3–5, 10, 27, 47, 94

Decision Problem The problem for which a decision has to be made by weighing the pros and cons of each alternative. Alternatively called Decision Issue. 6, 7, 50, 58, 63, 89, 94

Design Decision Decision made regarding the general design of the software. 14, 47

Forward Snowballing Identifying related articles that cite a reviewed paper in their references. 13

Frontend Aspects of the software that concern elements seen and used by the user directly, like the user interface. 47, 49, 62, 67, 91

Knowledge Element A piece of documented information. In most cases in this thesis refers to either a decision problem, alternative, decision, pro-argument, con-argument or code class. First occurrence on p. 3

Knowledge Graph Encompasses all knowledge elements (as nodes) and their connecting links (as edges, relationships). 8, 29, 49

Rationale Manager Developer tasked with maintaining consistency and completeness of the documented decision knowledge. 2, 5, 27, 28, 30, 33, 46, 48, 52, 87, 94

REST Representational state transfer. System designed to allow interoperability between computers/servers on the Internet. Here it is used to connect backend and frontend of the plug-in by allowing client-server communication. 47, 51, 62, 91

Software Artifact A tangible element produced during the development of a software, i.e. source code, requirements, design documents etc. 2, 9, 22, 27

Acronyms

CSE	Continuous Software Engineering. 29
DP	Decision Problem. 50, 51, 58, 59, 63, 106
EoU	Perceived Ease of Use. 81, 83
IDE	Integrated Development Environment. 29
ITS	Issue Tracking System. 1, 8, 29
ItU	Intention to Use. 81, 83
NFR	Non-Functional Requirements. 39, 50, 58, 63
POS	Part of Speech. 9
PU	Perceived Usefulness. 81, 83
SF	System Function. 31
TAM	Technology Acceptance Model. 80–82, 84
UI	User Interface. 22, 40
URI	Uniform Resource Identifier. 75
VCS	Version Control System. 1, 29
WS	Workspace. 40

List of Figures

List of Tables

Eidesstaatliche Erklärung zur Masterarbeit

Hiermit erkläre ich, dass ich diese Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe.

Ich habe die Grundsätze und Empfehlungen „Verantwortung in der Wissenschaft“ der Universität Heidelberg gelesen und befolgt.

Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

Heidelberg, den ....