INAUGURAL – DISSERTATION

submitted

to the

Combined Faculty of Mathematics, Engineering, and Natural Sciences

of

Heidelberg University, Germany

Put forward by

M.Sc. Anja Kleebaum

Born in Zwickau

Oral examination: .....

Continuous Rationale Management

Supervisors:	Prof. Dr. Barbara Paech	Heidelberg University
	Prof. Dr. Bernd Bruegge	Technical University of Munich

Abstract

Continuous Software Engineering (CSE) is a software life cycle model open to frequent changes in requirements or technology. During CSE, software developers continuously make decisions on the requirements and design of the software or the development process. They establish essential *decision knowledge*, which they need to document and share so that it supports the evolution and changes of the software. The management of decision knowledge is called *rationale management*. Rationale management provides an opportunity to support the change process during CSE.

However, rationale management is not well integrated into CSE. The overall goal of this dissertation is to provide workflows and tool support for *continuous rationale management*. The dissertation contributes an *interview study* with practitioners from the industry, which investigates rationale management problems, current practices, and features to support continuous rationale management beneficial for practitioners. Problems of rationale management in practice are threefold: First, documenting decision knowledge is intrusive in the development process and an additional effort. Second, the high amount of distributed decision knowledge documentation is difficult to access and use. Third, the documented knowledge can be of low quality, e. g., outdated, which impedes its use. The dissertation contributes a *systematic mapping study* on recommendation and classification approaches to treat the rationale management problems.

The major contribution of this dissertation is a validated approach for continuous rationale management consisting of the *ConRat* life cycle model extension and the comprehensive *ConDec* tool support. To reduce intrusiveness and additional effort, ConRat integrates rationale management activities into existing workflows, such as requirements elicitation, development, and meetings. ConDec integrates into standard development tools instead of providing a separate tool. ConDec enables lightweight capturing and use of decision knowledge from various artifacts and reduces the developers' effort through *automatic text classification*, *recommendation*, and *nudging* mechanisms for rationale management. To enable access and use of distributed decision knowledge documentation, ConRat defines a knowledge model of decision knowledge and other artifacts. ConDec instantiates the model as a *knowledge graph* and offers interactive *knowledge views* with useful tailoring, e. g., transitive linking. To operationalize high quality, ConRat introduces the *rationale backlog*, the *definition of done* for knowledge documentation, and metrics for *intra-rationale completeness* and *decision coverage* of requirements and code. ConDec implements these agile concepts for rationale management and a *knowledge dashboard*. ConDec also supports consistent changes through *change impact analysis*.

The dissertation shows the *feasibility*, *effectiveness*, and *user acceptance* of ConRat and ConDec in six case study projects in an industrial setting. Besides, it comprehensively analyses the rationale documentation created in the projects. The validation indicates that ConRat and ConDec benefit CSE projects. Based on the dissertation, continuous rationale management should become a standard part of CSE, like automated testing or continuous integration.

Zusammenfassung

Agile Softwareentwicklung muss mit häufigen Änderungen in den Anforderungen oder Technologien umgehen können. Während der Softwareentwicklung treffen die Beteiligten kontinuierlich Entscheidungen, zum Beispiel hinsichtlich der Anforderungen an die Software, ihres Entwurfs oder des Entwicklungsprozesses. Sie eignen sich wichtiges Entscheidungswissen an, das sie dokumentieren und teilen müssen, damit die Software weiterentwickelt und geändert werden kann. *Rationale-Management* beschäftigt sich mit der Verwaltung des Entscheidungswissens.

Allerdings ist Rationale-Management nicht hinreichend in den Entwicklungsprozess integriert. Das übergeordnete Ziel dieser Dissertation ist es, Arbeitsabläufe und Werkzeugunterstützung für *kontinuierliches Rationale-Management* bereitzustellen. Die Dissertation umfasst eine *Interviewstudie* mit Fachkräften aus der Industrie, die Probleme des Rationale-Management sowie aktuelle Abläufe untersucht und Anforderungen an kontinuierliches Rationale-Management erhebt. Die Studie hat drei Probleme identifiziert: 1) Die Dokumentation von Entscheidungswissen greift in den Entwicklungsprozess ein und stellt einen zusätzlichen Aufwand dar. 2) Die große Menge an verteilter Wissensdokumentation ist schwer zugänglich und nutzbar. 3) Das dokumentierte Wissen kann von geringer Qualität sein, was die Nutzung erschwert. Die Dissertation beinhaltet eine *systematische Literaturstudie* über Empfehlungs- und Klassifizierungsansätze zur Behandlung der Probleme des Rationale-Management.

Der Hauptbeitrag dieser Dissertation ist ein validierter Ansatz für das kontinuierliche Rationale-Management, bestehend aus der *ConRat*-Prozessbeschreibung und der umfassenden *ConDec*-Werkzeugunterstützung. Um den Aufwand gering zu halten, integriert ConRat Rationale-Management-Aktivitäten in bestehende Arbeitsabläufe, wie Anforderungserhebung, Implementierung und Besprechungen. ConDec erweitert bestehende Entwicklungswerkzeuge, anstatt ein separates Werkzeug bereitzustellen. ConDec ermöglicht die leichtgewichtige Erfassung und Nutzung von Entscheidungswissen ausgehend von verschiedenen Softwareartefakten. Es reduziert den Aufwand von Rationale-Management durch *automatische Textklassifikation*, *Empfehlungs-* und *Anreizmechanismen*. Um die verteilte Dokumentation zu unterstützen, definiert ConRat ein Wissensmodell mit Entscheidungswissen und anderen Softwareartefakten. ConDec instantiiert dieses Modell und bietet interaktive *Wissensansichten* mit nützlichen Anpassungsmöglichkeiten, z.B. durch transitive Verknüpfungen. Um hohe Qualität zu gewährleisten, führt ConRat das *Rationale-Backlog*, *Abnahmekriterien* für Wissensdokumentation sowie Metriken für die *Intra-Rationale-Vollständigkeit* und *Entscheidungsabdeckung* von Anforderungen und Code ein. ConDec implementiert die auf Rationale-Management übertragenen agilen Konzepte sowie ein *Wissens-Dashboard*. ConDec unterstützt konsistente Änderungen durch *Änderungsauswirkungsanalyse*.

Die Dissertation zeigt die *Einsetzbarkeit*, *Effektivität* und *Benutzerakzeptanz* von ConRat und ConDec in sechs Fallstudien in der Praxis. Sie beinhaltet eine umfassende Analyse der in den Projekten erstellten Wissensdokumentation. Ausgehend von dieser Dissertation sollte kontinuierliches Rationale-Management ein fester Bestandteil von Softwareentwicklung werden.

Acknowledgements

First and foremost, I would like to thank my first supervisor Professor Barbara Paech for making it possible for me to become a doctoral student and be a part of her *Software Engineering Group* at the Combined Faculty of Mathematics, Engineering, and Natural Sciences of Heidelberg University. I thank Barbara for providing continuous and very valuable feedback while supervising my thesis and supporting me in learning many things during my work at University. Among others, I am grateful for her selection of a very interesting seminar about sustainable digitalization called *Bits & Bäume*, for initiating collaboration with the *Heidelberg Institute for Geoinformation Technology*, and for her teaching me always to think software development from the users.

My sincere thanks go to my second supervisor Professor Bernd Bruegge, who contributed valuable feedback. I admire Bernd's mindset that everything is possible, especially how he inspires others to adopt the same attitude. I am thankful that Barbara and Bernd initiated the *Continuous Usage- and Rationale-based Evolution Decision Support (CURES)* project that shaped my thesis topic. Special thanks go to Jan Ole Johanssen, who also worked on the project, for supporting me in many situations and continuously motivating my research and thesis writing. Jan indeed showed me what it means to be an early adopter, and it would require an additional appendix to my thesis to thank him appropriately. The CURES workshops and discussions are truly unforgettable and generated many incentives for my thesis. I am grateful to have had the chance to present and validate ConRat and the ConDec tool support in the *iPraktikum* and the *Project Organization and Management* course at the Technical University of Munich.

I thank my colleagues at the Software Engineering Research Group at Heidelberg University for valuable discussions and feedback. I thank them for preparing guidelines and templates and for doing teaching work, while I could perform my research. I thank Marcus Seiler for being an excellent onboarding guide in Heidelberg and having a wonderful time in Lisbon at the *Software Engineering and Knowledge Engineering* conference. I thank Professor Christian Kücherer for always bringing a good mood to the office, sometimes even topped by bringing his dog, and for proofreading parts of my thesis. I thank Paul Hübner for motivating me at least 42 times and rescuing the world. I thank Thomas Quirchmayr for constructive criticism and for leaving his printed theses as perfect templates. I thank Astrid Rohmann for sharing her knowledge, e. g., about Android development, and outstanding traveling pictures with me. I thank Michael Anders for contributing to my research with his master's thesis and managing teaching activities and events after me. I thank Leon Radeck for daily research support, excellent team-building skills, and always being ready for lunch. I thank Rumyana Proynova for her ability for spontaneous activities and for being so open-minded. I thank Tom-Michael Hesse for laying the foundation for my work in the project on *Usage- and Rationale-based Evolution Decision Support* and for initiating the first version of the ConDec Jira and Confluence plug-ins. I thank Thorsten Merten for the yogi and sustainable lifestyle inspirations and for reflecting on supply chains.

Table 1.: Bachelor and master theses contributing to the dissertation.

Master Theses
J. Clormann (2018). "DecXtract: Dokumentation und Nutzung von Entscheidungswissen in Jira-Issue-Kommentaren". Master Thesis. Heidelberg University. DOI: 10.11588/heidok.00026059
L. Wisniowski (2019). "Quality assurance of documented decision knowledge in feature branches". Master Thesis. Heidelberg University
I. Hamma (2019). "Unterstützung der konsistenten Dokumentation von Entscheidungen im Software Engineering". Master Thesis. Heidelberg University
M. Anders (2020). "Comprehensive and Targeted Access to and Visualization of Decision Knowledge". Master Thesis. Heidelberg University. DOI: 10.11588/heidok.00029025
P. de Sombre (2020). "Verlinkungsunterstützung und Duplikaterkennung von Wissenelementen". Master Thesis. Heidelberg: Heidelberg University
K. Yan (2021). "Unterstützung der Analyse von Änderungsauswirkungen auf Graphen von Wissenelementen". Master Thesis. Heidelberg University
P. Zubrod (2021). "Vorschlagsmechanismus für Lösungsoptionen zu Entscheidungsproblemen in der Softwareentwicklung". Master Thesis. Heidelberg University
Bachelor Theses
P. Zubrod (2017). "Dokumentation und Nutzung von Entscheidungen in Git". Bachelor Thesis. Heidelberg University
M. Seiler (2017). "Dokumentation und Nutzung von Entscheidungen in Code". Bachelor Thesis. Heidelberg University
V. Aman (2019). "Unterstützung der Konsistenz zwischen Entscheidungen und ihrer Umsetzung durch Zusammenfassung von Codeänderungen". Bachelor Thesis. Heidelberg University
L. Tralle (2019). "Visualisierung und Verwaltung von Entscheidungswissen in Jira". Bachelor Thesis. Heidelberg University
K. Nizenkov (2019). "Design and implementation of a developer-centric quality web application". Bachelor Thesis. Heidelberg University
T. Kuchenbuch (2019). "Darstellung der Evolution von Entscheidungswissen". Bachelor Thesis. Heidelberg University
F. Gronert (2019). "Unterstützung der Erstellung von Release-Beschreibungen durch dokumentiertes Entscheidungswissen". Bachelor Thesis. Heidelberg University
Ö. Boz Kumru (2019). "Analyse und Klassifikation von Entscheidungswissen in Jira-Issues". Bachelor Thesis. Heidelberg University
C. Otchere (2020). "Die Rolle von Qualitätsattributen während der Dokumentation und Nutzung von Entscheidungswissen". Bachelor Thesis. Heidelberg University
R. Gerner (2020). "Entwicklung eines Rationale Backlogs". Bachelor Thesis. Heidelberg University
J. Baum (2021). "Support for Rationale Management with Nudging". Bachelor Thesis. Heidelberg University
M. Boerner (2021). "Qualitätssicherung von dokumentiertem Wissen mithilfe der Erhebung der Entscheidungsabdeckung". Bachelor Thesis. Heidelberg University
L. Bendl (2022). "Change Impact Analysis for Issue Tracking Systems". Bachelor Thesis. Heidelberg University

A big thank you goes to Doris Keidel-Müller for in-depth conversations and proofreading all our publications. A sincere thank you goes to Willi Springer for letting me learn many things about IT operations and immediately having the proper backup in an emergency. I thank Stephanie Sokoll for her support during the final sprint of my thesis and for helping organize the defense. I thank Professor Andrea Hermann, who enabled me to present my research in her course and is an admirable expert in many fields. I thank Eckhart von Hahn and Heiko Koziolek for the valuable discussions and feedback.

My gratitude goes to the members of the *Chair of Applied Software Engineering* at the Technical University of Munich. My visits to Garching were unforgettable, with a warm atmosphere making me feel like a part of the team immediately. I thank Dominic Henze and Jan for applying ConDec during the iPraktikum and collecting feedback. My profound thanks go to Rana Alkadhi, who shared her data and expertise and is a very inspiring and intelligent researcher. I thank Professor Stephan Krusche for providing Rugby as an essential life cycle modeling foundation for my work and valuable feedback. I thank Matthias Linhuber, Florian Angermeier, and Helma Schneider for installing ConDec and providing technical support.

I very much thank all the talented students who supported my dissertation through their master's and bachelor's theses (Table 1), in practical work, or as scientific assistants: Ewald Rode, Frederic Born, Tim Kuchenbuch, Paul Zubrod, Jochen Clormann, Lukasz Wisniowski, Martin Seiler, Ines Hamma, Vita Aman, Lars Tralle, Katherina Nizenkov, Edgar Brotzmann, Fabian Gronert, Zhaobin Zhu, Özlem Boz Kumru, Rafael Gerner, Colin Otchere, Philipp de Sombre, Markus Boerner, Silas Hauk, Dominik Hirsch, Klaus Yan, Julia Baum, Marvin Ruder, Maximilian Hartmann, Lukas Bendl, and Michael Anders, who kept working in the group as a doctoral student. I hope all of them will continue being open-minded researchers.

I sincerely thank my family and friends for always supporting me during my life and doctoral studies. I thank Dominik Pieper for storing me as Dr. in his address book long before I moved to Heidelberg and believing in me to write a doctoral thesis one day. I very much thank him for giving me stability in the fast-living world of academia and for all the support throughout my dissertation, for instance, by continuously delivering me coffee and food. I very much thank my parents, who endured that I moved in again for weeks when writing my thesis. Among others, I thank my dad for building my own computer and introducing me to computer science at a very young age and my mum for always caring about my work-life balance. I thank Monika Viehmann for her valuable feedback twenty-four-seven and her great personality. I thank Linda Menger and Barbara Glaser for our extraordinary hiking adventures and continuous support. I thank many people who very much supported me in the past and who I want to see more often in the future, for example, Sophie, Paula, Franzi, Florian, Stefan, Yrneh, Vreni, Franzi, Caro, Laura, Carsten, Michi, Ilka, Milan, Eva, Sina, and Maja. I thank the *Ashtanga Yoga Institute* and the *Villa Sportiva* in Heidelberg for helping me stay healthy despite working in front of the computer.

I thank the *German Research Foundation (DFG)* for granting the CURES project as part of the SPP1593 priority program. I thank the Graduate Academy of Heidelberg University, which granted me a fellowship for finishing my thesis funded through the *Baden-Württemberg Landesgraduiertenförderung*. I sincerely thank the anonymous reviewers who spent their precious time reading and commenting on our papers and helped improve this work. I thank the *Aufbau Unstrut-Finne* foundation for their support during my master's studies. My gratitude goes to Professor Klaus Bitzer, Professor Holger Lange, Professor Christina Bogner, Stefan Holzheu, and Professor Matthias Korch for having opened the way for me to become a researcher.

Contents

List of Acronyms

Part I.

Preliminaries

Chapter 1

Introduction

> *"Rationale is only useful if the developers actually use it. If the rationale support tools are integrated into tools already used by the developers then it might be possible to present the rationale exactly when it is needed without extra effort from the developer."*
>
> —Burge and D. C. Brown, 2008a

This chapter gives an introduction to the topic of continuous rationale management. Section 1.1 introduces continuous software engineering and the importance of decision knowledge and its management, i. e., rationale management. It motivates two opportunities the integration of rationale management into continuous software engineering offers: 1) a high expected benefit for the frequent changes and 2) a relatively low expected effort when integrating the rationale management into the current practices and tools. Section 1.2 describes the problems that impede rationale management, which this thesis aims to treat. Section 1.3 describes the research methodology. Section 1.4 sets the goals of this thesis, and Section 1.5 describes the respective contributions. Section 1.6 outlines the structure of the thesis. Section 1.7 lists previous scientific publications that contain parts of this thesis.

1.1. Motivation

Continuous Software Engineering (CSE) is an agile software development process model that enables developing, releasing, and learning from software in very short and rapid cycles (Bosch, 2014). CSE supports the incremental implementation of requirements and involves activities such as continuous integration, continuous delivery, and continuous deployment (Shahin et al., 2017). The term *continuous* is not interpreted in a strict mathematical sense; still, it means that an activity is done constantly, in small increments over time, rather than once in a *big bang*. CSE emerged due to a growing need for flexibility and rapid adaption in the current software environment (Fitzgerald and Stol, 2017). The CSE process model and its support are in flux; for example, Johanssen (2019) added a facility to provide timely validation through user feedback.

CSE allows developers to collaboratively implement and deliver many increments, which involves decision making, such as decisions on the requirements to be implemented, the design, implementation, and test, or the software development process. The knowledge of the developers about these decisions is called *rationale* or *decision knowledge* (Dutoit et al., 2006). Decision knowledge is the knowledge about the decision problems, i. e., issues to solve, solution alternatives, decisions, and justifications in terms of criteria, pro- and con-arguments.

Decision knowledge is communicated within a development team so that every developer knows and considers existing decisions (Brunet et al., 2014). When developers evolve software, they must reflect and build on former decisions. Otherwise, inconsistent decisions are likely to contribute to the erosion of the software architecture or introduce other quality problems (Cleland-Huang et al., 2013; Capilla et al., 2016). Reflecting on former decisions is particularly important for long-living software systems for which many decisions build on one another.

Besides decision knowledge communication, the documentation of decision knowledge, i. e., its externalization from the developers' minds and its preservation, is also essential for several reasons: First, different developers might be involved at different times. They cannot communicate directly and rely on documented decision knowledge to understand former decisions. Documenting decision knowledge prevents knowledge vaporization (Capilla et al., 2016). Decision knowledge vaporizes quickly, i. e., if developers do not capture it immediately, it is often not captured at all and thus unavailable later (Jansen and Bosch, 2005). Second, the documentation of decision knowledge makes alternatives and criteria for the decisions explicit that might otherwise be overlooked. This promotes a more rational decision-making process, better decisions, and better design (Tofan et al., 2013; Weinreich et al., 2015). Third, documented decision knowledge is valuable to support future changes. It supports change impact analysis, requirements validation and verification, architecture review, and long-term maintenance, and keeps developers informed about underlying design decisions (Bratthall et al., 2000; Cleland-Huang et al., 2013; Tang and Lau, 2014).

Recently, several summarization techniques were used to reconstruct decision knowledge by mining written text from informal sources such as chat messages (Alkadhi, 2018). These techniques for *automatic text classification*, also called *extractive summarization* (Nazar et al., 2016), help to identify decision knowledge. However, the knowledge may be incomplete, outdated, or hard to access later. In other cases, the knowledge is not captured but resides in the developers' heads as tacit knowledge. Tacit decision knowledge enlarges the risk of misunderstandings and errors during evolution or maintenance. Researchers attempt to infer tacit knowledge by *abstractive summarization* of software artifacts such as source code changes (Cortés-Coy et al., 2014). However, Robillard et al. (2017) describe that it is unlikely to infer complex information such as rationale by the mechanical extraction of facts from software artifacts. For instance, non-existence decisions, which state that an element does not appear in the design or implementation, are not traceable to and from the software artifacts (Kruchten, 2004; Kruchten, 2009). Such decisions can neither be inferred nor justified without explicit decision knowledge documentation. Therefore, summarization techniques only partially help to reconstruct decision knowledge in case they are applied retrospectively. Decision knowledge needs to be explicitly documented to preserve it. Various approaches exist to support the documentation through *recommendations* (Zimmermann and Miksovic, 2013; Miesbauer and Weinreich, 2012; Bhat et al., 2017a; Zimmermann et al., 2015). It is important to note that easy exploitation of the decision knowledge motivates developers to document it, as the developers themselves profit from the documentation (Burge and D. C. Brown, 2008a).

Rationale management demands that developers document their decision knowledge and exploit the documentation, to support explicit decision making (Dutoit et al., 2006). It is a method for justifying and supporting change (Bruegge and Dutoit, 2010). Its systematic integration is met with resistance as it requires additional effort (Rogers et al., 2015). Section 1.2 will describe the issues that complicate systematic rationale management. CSE offers two opportunities for integrating rationale management: 1) CSE and current software development tools minimize the overhead of the documentation of decision knowledge. Developers usually manage code, issues, and other development knowledge in a *version control system* and an *issue tracking system* (Saito et al., 2017). These systems offer lightweight opportunities for capturing decision knowledge, such as ticket comments, commit messages, and code comments used in established practices such

as managing requirements and development tasks or committing code. 2) Rationale management supports developers in frequently communicating, making, and changing decisions.

The term *process* is used interchangeably with the term *life cycle*. A *life cycle* describes the "evolution of a system, product, service, project, or other human-made entity from conception through retirement" (ISO/IEC/IEEE 24774, 2021). A *life cycle model* or *process model* "represents all the activities and work products necessary to develop a software system" (Bruegge and Dutoit, 2010).

To summarize, a *rationale management approach* for developers in CSE is beneficial. This thesis presents a validated approach consisting of the *Continuous Rationale Management (ConRat) life cycle model extension* and tool support. ConRat is based on a *knowledge model* and integrates *rationale management activities* into CSE: collaborative, incremental, and rational decision making, documentation, exploitation, and quality assurance of decision knowledge. ConRat is supported by the views and features of the *Continuous Management of Decision Knowledge (ConDec) plug-ins*. The long-term vision of this thesis is an *on-demand decision documentation* as part of the *on-demand developer documentation* suggested by Robillard et al. (2017), which means that developers continuously capture and reflect decision knowledge.

1.2. Problem Context

This section describes three *rationale management problems* that ConRat and its support through the views and features of ConDec aim to treat.

Tool support to manage decision knowledge can be characterized by its *intrusiveness in the software development process* (Dutoit et al., 2006). Tools that do not fit into the development context are intrusive and are less likely to be used (Kruchten et al., 2009) because they *require additional effort*, e. g., for installing or starting, and are thus not lightweight. For example, suppose developers have to capture decisions in a text document, e. g., in an architecture log file, or a particular tool. In that case, they must navigate to the document or tool and externalize the decisions and related decision knowledge. The *Agile Manifesto* values working software over comprehensive documentation (Beck et al., 2001). In agile software development, productivity is measured by the amount of working software, and documentation might even be considered a waste (Theunissen et al., 2022). Besides, the documented decision knowledge is *hard to access and exploit* for developers as it is separate from the software artifacts. We summarize the first problem as follows:

> **Intrusiveness and Effort Problem:** Developers do not document decision knowledge if the techniques and tools are *intrusive*, i. e., do not integrate into development workflows and require too much effort. It requires *additional effort for the developers to document the decision knowledge and exploit* the knowledge documented by other developers when needed, e. g., during software evolution.

Software engineering is a knowledge-intensive process (Babar et al., 2009; Robillard and Walker, 2014; Paech et al., 2014; Saito et al., 2017). The documented knowledge tends to contain many knowledge elements and links in distributed documentation locations, such as in the issue tracking system, version control system, or wiki system. It hinders the exploitation if the *distributed decision knowledge is not traceable from the software artifacts* such as requirements. The dynamic nature of CSE aggravates the second problem. There is a vast amount of data available in CSE that has the potential to support decision making (Svensson et al., 2019). Large software systems require many decisions to be made, and various stakeholders make them (Tang and Lau, 2014). The high amount of documented knowledge can cause *information overload* (Codoban et al., 2015). Making rationale accessible to all concerned parties without causing information overload is difficult (Bruegge and Dutoit, 2010). The rationale documentation is only helpful

if it is tailored to the developers' needs, i. e., if the developers can easily access the part of the documented knowledge that is useful for their current development task. We summarize the second problem as follows:

> **High Amount of Distributed Knowledge Problem:** Software development is knowledge-intensive, and the *documented knowledge is complex* with many knowledge elements in *distributed documentation locations*. Tailoring the documented knowledge to the developers' needs, preventing *information overload*, and offering knowledge access and exploitation is difficult.

To exploit documented decision knowledge, it *must be of high quality* (Thurimella et al., 2017). Quality attributes for software documentation are accessibility, traceability, completeness, consistency, correctness, up-to-dateness, uniqueness, information organization, format, spelling and grammar, readability, accuracy, trustworthiness, and author-related aspects (Zhi et al., 2015). This thesis focuses on 1) completeness and 2) consistency, correctness, and up-to-dateness as important quality aspects.

First, the *documentation must be complete* in the sense that important decision knowledge is captured. While there is a need for decision knowledge documentation, this is often not performed (Alexeeva et al., 2016). This problem is called the *capture problem* (Dutoit et al., 2006). CSE is short-term focused and, initially, short-term software development tasks can be achieved without documenting important decision knowledge. However, the knowledge disappears over the following iterations with changing context, new requirements, and new developers (Theunissen et al., 2022). In practice, decisions are often made and documented in a naturalistic way (Zannier et al., 2007; Hesse et al., 2016b). This means that only a part of the decision knowledge—often only the decision—is documented, which impairs rational decision making. Humans overlook what is missing and are subject to cognitive biases (Maule, 2010; Razavian et al., 2016). Furthermore, other developers might not understand or be convinced if the arguments for the solution decision are not documented. Practitioners are often unsure how to capture rationale and what to capture (Schubanz, 2014; Thurimella et al., 2017). They must decide whether to capture rationale in natural language text or with a rationale model as a more systematic approach. A rigorous approach to capturing rationale might lead to *analysis paralysis*, which means that the actual development is postponed or never accomplished (W. H. Brown et al., 1998). It is challenging to document the proper amount of rationale since the relevance of rationale depends on the exploitation scenario and the developers' needs (Burge, 2008). The documentation of decision knowledge often only pays off after a long time. Patterns for reusing rationale need to be clear (Thurimella et al., 2017).

Second, *the decisions must be consistent* a) with former decisions and b) with the artifacts, e. g., with the requirements, architectural software design (Tang et al., 2011), and code. Consistency means that the documented decisions are realized in the artifacts they relate to. Maintaining consistency is facilitated if the decisions are linked to these artifacts. For example, a decision to apply a design pattern should be linked to the code that implements the design pattern. Then, developers can reflect on this decision when they change the code. Developers need to reflect on former decision knowledge during decision making so that the decisions are consistent and correct. There are two types of decision knowledge: First, general decision knowledge, such as design patterns, is documented in external knowledge bases. Correctness means that the documented decision knowledge does not conflict with such factual information (Zhi et al., 2015). Second, new decisions build on previous decision knowledge specific to the software development project. Documented decision knowledge might be invalidated during software evolution and needs to be updated. The rapid change aggravates the documentation quality problem regarding inconsistency: Decisions can rapidly be changed, which might lead to inconsistent and outdated documentation. We summarize the third problem as follows:

Low Documentation Quality Problem: The decision knowledge documentation can be *incomplete* because developers do not document decision knowledge at all or only partly. If the decision knowledge is documented, it might be *inconsistent* with other decision knowledge or software artifacts because it can be *outdated*. Such decision knowledge *cannot be exploited*.

Table 1.1 summarizes how CSE aggravates the rationale management problems based on the challenges regarding software documentation reported by Theunissen et al. (2022): informal documentation is hard to understand, documentation is considered a waste, productivity is measured by the amount of working software only, documentation is out-of-sync with the software, and the short-term focus leading to knowledge vaporization.

Table 1.1.: Rationale management problems and their aggravation through CSE.

Problem	Aggravation in CSE
Intrusiveness and effort problem	Productivity is measured by the amount of working software. Documentation might even be considered a waste. ⇒ Effort for documentation needs to be particularly low. Otherwise, it is not spent.
High amount of distributed knowledge problem	Much decision knowledge accumulates in distributed documentation locations, e. g., in the issue tracking system and version control system. ⇒ Accessing the knowledge is particularly hard. The documentation is informal and, thus, hard to understand.
Low documentation quality problem	Short-term focus aggravates incomplete documentation. Rapid change aggravates inconsistent documentation that is out-of-sync with the software or with other decision knowledge, i. e., outdated.

1.3. Research Methodology

The research project conducted during this thesis follows the *design science methodology* described by Wieringa (2014). Design science deals with the design and investigation of artifacts in context. An artifact can be an algorithm, a method, a notation, a technique, a tool, or a conceptual framework. Researchers of a design science research project iterate over two activities and thereby address two categories of research goals (Figure 1.1): On the one hand, the researchers design an artifact to improve a problem context for stakeholders. Stakeholders are people in the social context who may affect the project or may be affected by it. The respective research goal of the first task is an artifact *design goal*, or, synonymously, a *technical research goal*. On the other hand, the researchers aim to achieve *knowledge goals*. A knowledge goal can be to empirically investigate one or more artifacts in a given context or describe and explain phenomena. The UML activity diagram in Figure 1.1 does not contain a start node because a design science project can start with both activities.

The design science research goals are refined to make them achievable, similar to the *goal question metric* approach (Basili et al., 1994). The technical research goal is refined into *sub-goals*, and the knowledge goals are refined into *knowledge questions*, i. e., research questions. The achievement of a knowledge goal can be supported by an *instrument design goal*, which defines that the researchers need a research instrument to answer a knowledge question (Wieringa, 2014).

The design science approach consists of five tasks: Problem investigation, treatment design, treatment validation, treatment implementation, and implementation evaluation. During the problem investigation, researchers aim to understand the current state of research and practice, refine the understanding of the problem context, and review existing solutions (i. e., treatments) for the problems. Based on the current state, artifacts are identified for the treatment design

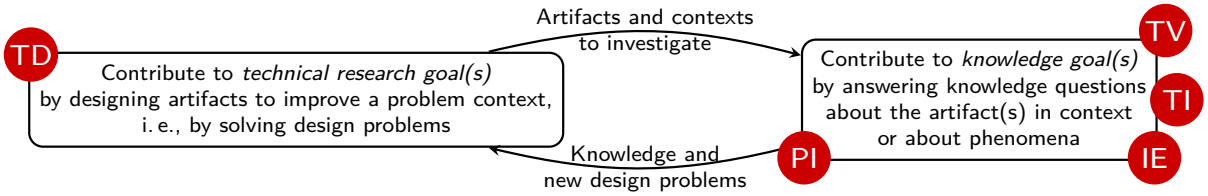

Figure 1.1.: Two major activities that researchers of a design science research project iteratively perform as a UML activity diagram based on Wieringa (2014). The activities contribute to the two categories of research goals, i. e., to the *technical research goals* and the *knowledge goals*. The tasks Problem Investigation (PI), Treatment Design (TD), Treatment Validation (TV), Treatment Implementation (TI), and Implementation Evaluation (IE) are mapped to the activities.

phase. During the treatment validation, researchers validate whether the artifacts treat the problems in a *simulated, yet realistic problem context*. In addition, during the last two tasks, treatment implementation and implementation evaluation, researchers apply and evaluate the artifacts in a real problem context, i. e., in an industrial setting. Collections of these tasks are referred to as *cycles* as researchers iterate over the tasks many times in a research project. The five tasks form the *engineering cycle*, whereas the first three tasks (problem investigation, treatment design, treatment validation) form the *design cycle*.

Figure 1.2 shows the design cycle of this thesis along with a forward reference to the respective research goal that each task aims to achieve. Section 1.4 will detail the research goals.

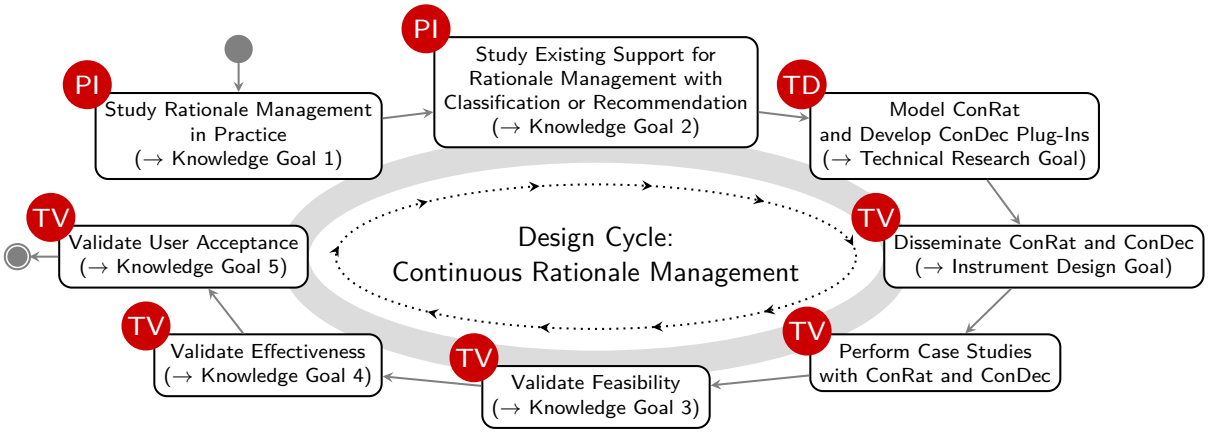

Figure 1.2.: Design cycle of the thesis as a UML activity diagram with reference to the research goals and the tasks Problem Investigation (PI), Treatment Design (TD), and Treatment Validation (TV).

The problem investigation consists of an interview study and a literature study. The interview study investigates rationale management in practice during CSE in the industry. The literature study analyzes the state of the art of rationale management support with classification and recommendation. The results of the problem investigation are the basis for the treatment design, namely the modeling of ConRat and the development of the ConDec plug-ins that support ConRat with views and features. For its validation, the treatment is applied in six case studies and improved. Before a case study, ConRat and ConDec are disseminated to the case study participants. Three different aspects are validated: 1) the feasibility through the analysis of the knowledge documentation created during the case studies, 2) the effectiveness of particular ConDec support, and 3) the acceptance of ConDec by the case study participants.

Different terms for software engineering research and validation aspects are used in the literature. Figure 1.3 groups the *validation aspects* by their reference and relates the aspects.

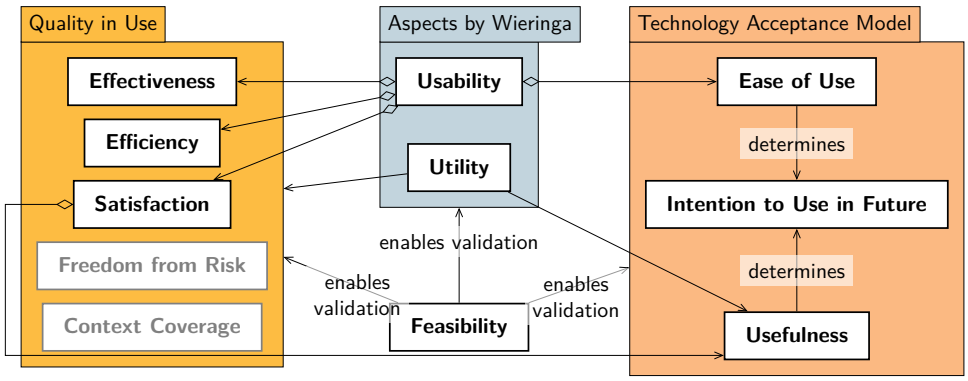

Figure 1.3.: Validation aspects examined in this thesis (UML package diagram).

Feasibility defines whether stakeholders can apply the treatment and is the primary aspect that the treatment needs to fulfill to validate the other aspects. In design science, the treatment is evaluated by its utility for the stakeholder goals and its usability (Wieringa, 2014). *Utility* is satisfied if the stakeholders can use the treatment to achieve their stakeholder goals. *Usability* is satisfied if the treatment is easy to use (and also fulfills other usability requirements such as understandability and ease of learning). The treatment validation aims to learn about the treatment's utility and usability in a real-world context (Wieringa, 2014). Similarly, the *quality in use* model of the ISO/IEC 25010 (2011) standard defines five characteristics to determine the degree to which users can use a product or system (here the treatment) to meet their needs: Effectiveness, efficiency, satisfaction, freedom from risk, and context coverage. *Effectiveness* defines the accuracy and completeness with which users achieve specified goals, i. e., how well a goal can be achieved with the product or system. *Efficiency* defines the time to complete the task, i. e., how efficiently a goal can be achieved with the product or system. *Satisfaction* defines the degree to which user needs are satisfied when a product or system is used. *Freedom from risk* defines the degree to which a product or system mitigates the potential risk to economic status, human life, health, or the environment. *Context coverage* defines whether the product or system can be used in contexts beyond those initially explicitly identified. According to the ISO/IEC 25010 (2011) standard, usability is defined as a subset of quality in use consisting of effectiveness, efficiency, and satisfaction. While the ISO/IEC 25010 (2011) standard does not use the term utility, quality in use relates to utility (Figure 1.3). This thesis does not validate freedom from risk and context coverage, so these validation aspects are grayed out in Figure 1.3. Another term used in the validation of treatments is *user acceptance* as defined by Davis et al. (1989) and Marangunić and Granić (2015) in the *Technology Acceptance Model*. The model consists of the three variables *ease of use*, *usefulness*, and the *intention to use in the future*. The ease of use and usefulness are determinants of the intention to use. *Usefulness* is also a sub-characteristic of satisfaction in the ISO/IEC 25010 (2011) standard. It defines the degree to which a user is satisfied with their perceived achievement of goals, including the results and consequences. The validation aspects are interchangeable, e. g., user acceptance with utility and usability (Figure 1.3).

The treatment can also be used to answer empirical knowledge questions, which do not depend on stakeholder goals. Answers to such knowledge questions are evaluated by truth (Wieringa, 2014). However, we can never be sure to have actually found the answer to an empirical knowledge question. The answer can be falsified as illustrated with the "all swans are white" example by Popper (1959). Just one observation of a black swan falsifies this proposition.

Stol and Fitzgerald (2018) define eight *research strategies* to conduct primary research: Field study, field experiment, experimental simulation, laboratory experiment, judgment study, sample study, formal theory, and computer simulation. This thesis applies four research strategies: sample study, field experiment, judgment study, and laboratory experiment. The research strategies are distinguished by their level of *obtrusiveness* and *generalizability*. Obtrusiveness refers to the extent to which researchers intrude on the research setting or whether they unobtrusively make observations. Generalizability refers to the extent to which researchers can draw conclusions from a study for other contexts or systems. Figure 1.4 arranges the primary empirical studies of this thesis along their level of obtrusiveness and generalizability. The literature study of the problem investigation is not included since it is a secondary study.

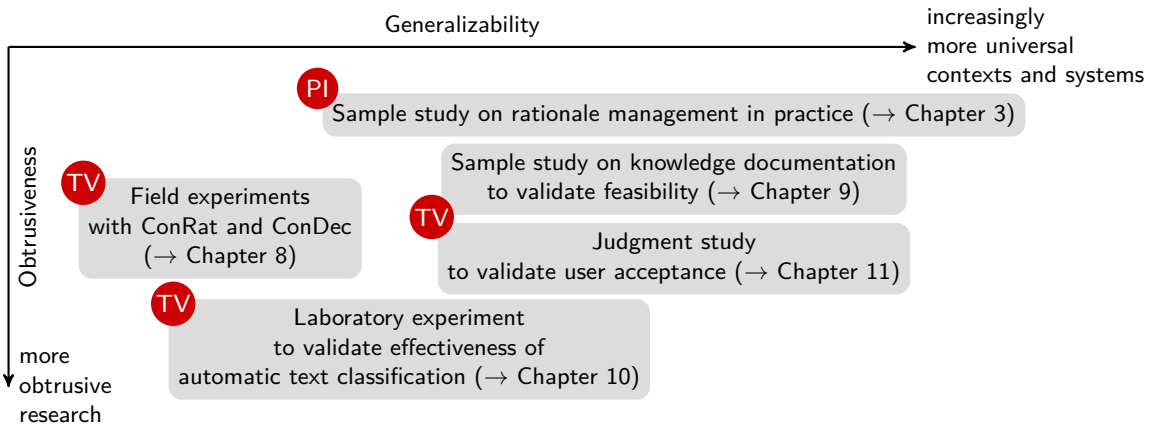

Figure 1.4.: Empirical studies along their level of *obtrusiveness* and *generalizability*. The studies are part of the Problem Investigation (PI) and Treatment Validation (TV).

In a *case study*, the researchers study a specific case, in contrast to a sample from a population. However, the term case study is used for a broad range of studies in software engineering (Runeson et al., 2012). The six case studies of this thesis are *field experiments*. A field experiment is conducted in a realistic research setting, which makes it less obtrusive than a laboratory experiment. It aims to investigate, evaluate, or compare techniques, practices, processes, or approaches. The term experiment is interpreted in a broad sense rather than in a strictly scientific sense (Stol and Fitzgerald, 2018). A typical research method applied during field experiments is *action research*. When performing action research, the researchers solve a problem and observe the effects of solving the problem (Easterbrook et al., 2008). That means that the researchers intrude into the research setting, i. e., action research is obtrusive in contrast to a purely observational case study. Wieringa and Moralı (2012) describe the usage of action research in design science, "which starts with an artifact, and then tests it under conditions of practice by solving concrete problems with them".

The thesis includes two *sample studies*. The first sample study is the interview study as part of the problem investigation. The second sample study is part of the treatment validation and analyzes the knowledge documentation created during the field experiments. When performing a sample study, researchers aim to achieve generalizability over a population of actors, e. g., practitioners, or software artifacts (Stol and Fitzgerald, 2018). Researchers do not intrude on the research setting, i. e., they do not manipulate any variables during data collection. Thus, sample studies do not allow to infer causal relationships. Sample studies involve surveys and the mining of software repositories. With *surveys*, researchers aim to "identify the characteristics of a broad population of individuals" (Easterbrook et al., 2008). In software engineering research, the term survey is also used as a synonym for literature review (Stol and Fitzgerald, 2018). The data collection is usually done with questionnaires or interviews (Ralph et al., 2020). During an

interview, the researcher asks the participant, i. e., the interviewee, a series of questions (Wohlin and Aurum, 2015). The purpose of interviews is to collect historical data from the interviewees' memories and opinions or impressions about a topic (Seaman, 1999). An advantage of interviews is that the researcher can follow up on the topic with the interviewees. There are different types of interviews: structured, unstructured, and semi-structured. In unstructured interviews, the researcher does not have a predefined questionnaire. Instead, the interviewee provides both questions and answers. In contrast, structured interviews demand a questionnaire, which does not allow to add additional questions. Semi-structured interviews involve predefined questions, but the researcher can also collect unexpected information (Seaman, 1999). Semi-structured interviews are most frequently used (Myers and Newman, 2007).

The thesis includes a *judgment study* to validate the user acceptance of ConDec. During a judgment study, the researchers are actively involved by adding a stimulus, e. g., applying a treatment. Experts, i. e., the study participants, rate or discuss a given topic of interest. When performing a judgment study, the researchers aim to achieve generalizability over the responses rather than generalizability to a population of actors (Stol and Fitzgerald, 2018). Similar to sample studies, researchers often use surveys and interviews as research methods.

This thesis includes a *laboratory experiment* to validate the effectiveness of automatic text classification as a particular ConDec support. During a laboratory experiment, the researchers conduct a *controlled experiment*. Controlled experiments enable determining a cause-effect relationship between the variables (Easterbrook et al., 2008). Laboratory experiments are conducted in a contrived (artificial) setting to minimize biases through confounding factors and extraneous conditions (Stol and Fitzgerald, 2018). The findings of a laboratory experiment are limited in their generalizability since the contrived setting is simplified compared to a realistic research setting, i. e., a real-world software development environment. Laboratory experiments are obtrusive because the researchers must set up a contrived setting.

This thesis uses the words *evaluation* and *validation* interchangeably, while Wieringa (2014) distinguishes them by the degree of the realism of the settings in which the treatment is applied.

1.4. Research Goals

This section describes two stakeholder goals as well as the goals of the researchers, i. e., the *research goals* of the thesis. Figure 1.5 depicts the goal structure of the design science research project conducted in the thesis. The research goals of this thesis include one technical research goal, five knowledge goals, and one instrument design goal.

The main stakeholders in this thesis are software developers or stakeholders of related roles who contribute to the software development, e. g., requirements engineers, product owners, architects, and testers. For simplification, software developers are used as examples of different roles, also called practitioners. The stakeholder goal of a software developer is:

> **Developer Goal:** Develop and evolve software consistent with former decisions by making the best new decisions.

Other stakeholders are the software users since this thesis aims to improve the product for the software users by improving the software development process through ConRat. However, software users are stakeholders with rather no awareness of the rationale management problems and their possible treatments. The stakeholder goal of a software user is:

> **User Goal:** Use software that fulfills functional and quality requirements.

This thesis aims to support the developer goal and, thus, indirectly support the user goal.

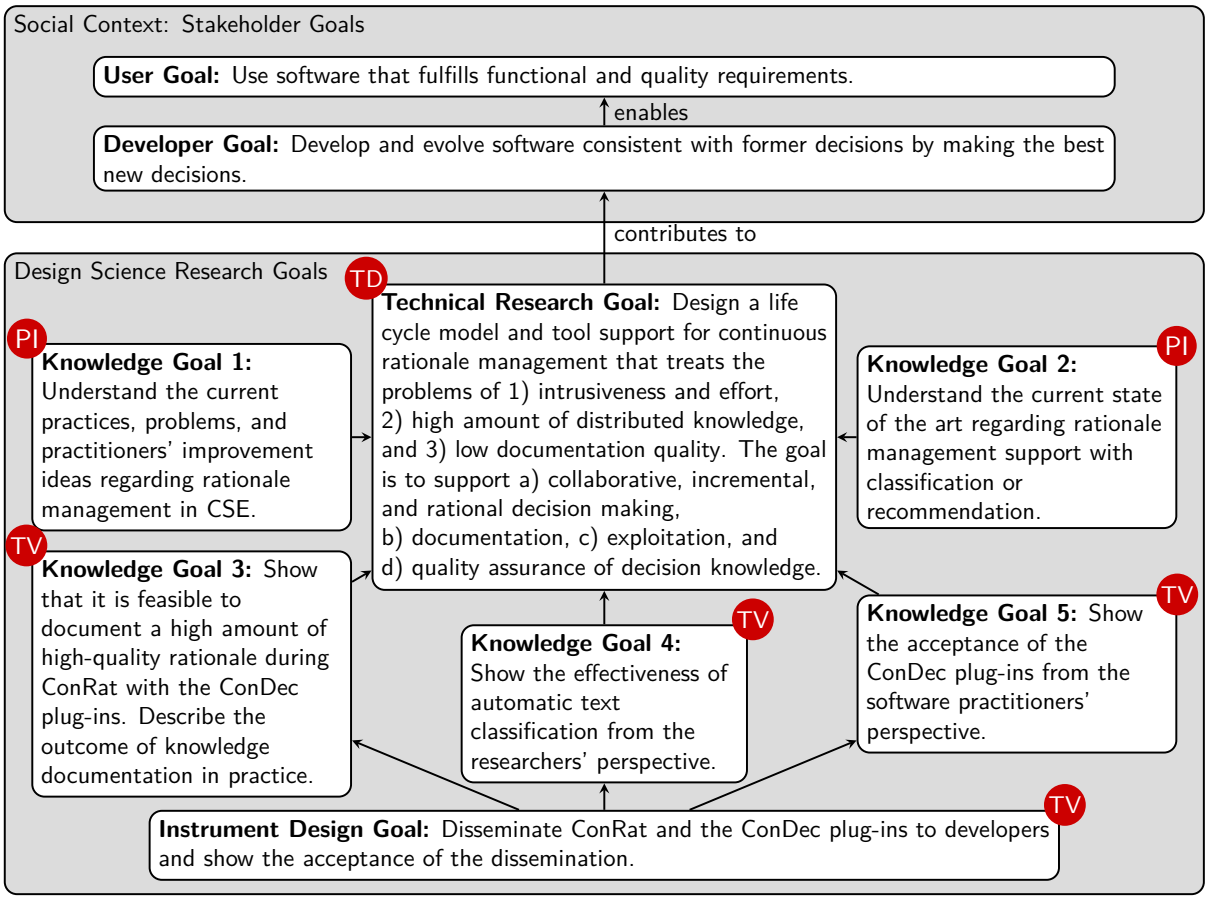

Figure 1.5.: Goal structure of the thesis. An arrow between two goals indicates that one goal's fulfillment contributes to the other's fulfillment. The Problem Investigation (PI), Treatment Design (TD), or Treatment Validation (TV) addresses the goals.

The first knowledge goal of this thesis aims to deepen the understanding of the rationale management practices and problems during CSE and to elicit improvement ideas:

> **Knowledge Goal 1 (Investigation):** Understand the current practices, problems, and practitioners' improvement ideas regarding rationale management in CSE.

Literature overviews of knowledge management support exist, e. g., by Ding et al. (2014), Alexeeva et al. (2016), Capilla et al. (2016), Weinreich and Groher (2016), and Hesse (2020). To narrow down the topic, the second knowledge goal of this thesis is to describe existing support that treats the rationale management problems in Section 1.2 with classification or recommendation. The first focus is on techniques that treat the intrusiveness and effort problem by automating parts of the documentation with text classification. The second focus is put on techniques that treat the problems with recommendations. *Recommendation systems* provide information to software engineers estimated to be valuable for their tasks in a given context. For example, recommendation systems can help developers to navigate through and reuse source code, discover tickets, find persons to contact, or assign a ticket (Robillard and Walker, 2014).

> **Knowledge Goal 2 (Investigation):** Understand the current state of the art regarding rationale management support with classification or recommendation.

The technical research goal of this thesis is to treat the rationale management problems in Section 1.2 and integrate rationale management activities into CSE and support them. The treatment should enable naturalistic decision making but encourage the developers to *collaboratively* and *incrementally* reflect on decisions made. This way, the decisions and their documentation evolve *from naturalistic to more rational*. The first and second knowledge goals' results are a basis for achieving the technical research goal. That means the treatment should integrate practitioners' improvement ideas and provide state-of-the-art support with classification and recommendation. The technical research goal is addressed by modeling a life cycle with continuous rationale management called ConRat and developing the ConDec plug-ins.

> **Technical Research Goal:** Design a life cycle model and tool support for continuous rationale management that treats the problems of 1) intrusiveness and effort, 2) high amount of distributed knowledge, and 3) low documentation quality. The goal is to support a) collaborative, incremental, and rational decision making, b) documentation, c) exploitation, and d) quality assurance of decision knowledge.

As described in the previous section, this thesis includes six case studies to validate whether the technical research goal was achieved. Developers, i. e., case study participants, need to know how to capture rationale and exploit the documentation during ConRat and using the ConDec views and features. The dissemination is, therefore, the instrument design goal:

> **Instrument Design Goal:** Disseminate ConRat and the ConDec plug-ins to developers and show the acceptance of the dissemination.

Based on the case studies, the following three knowledge goals address the treatment validation. The thesis validates the achievement of the technical research goal from different perspectives by investigating the three aspects of feasibility, effectiveness, and user acceptance. The third knowledge goal is to validate that it is feasible to treat the problems of a high amount and low documentation quality with ConRat and ConDec by analyzing the knowledge documentation of the six case studies. Next to the validation, the goal is to investigate rationale documentation in realistic projects to contribute to the body of knowledge.

> **Knowledge Goal 3 (Validation and Investigation):** Show that it is feasible to document a high amount of high-quality rationale during ConRat with the ConDec plug-ins. Describe the outcome of knowledge documentation in practice.

The fourth knowledge goal is to show the effectiveness of automatic text classification integrated into ConDec, thereby showing the treatment of intrusiveness and effort. The knowledge goal is addressed in a laboratory experiment from the researchers' point of view and using the rationale documentation created in the case studies.

> **Knowledge Goal 4 (Validation):** Show the effectiveness of automatic text classification from the researchers' perspective.

The fifth knowledge goal is to validate the acceptance of the ConDec plug-ins from the developers' point of view, i. e., the participants of the judgment study. This study is particularly important since it directly involves the main stakeholders of the thesis and validates the treatment of the three rationale management problems. The goal is to understand the developers' attitude toward the views and features of ConDec by asking for their perceived usefulness, ease of use, and intention to use in the future and monitoring their usage behavior.

> **Knowledge Goal 5 (Validation):** Show the acceptance of the ConDec plug-ins from the software practitioners' perspective.

1.5. Contributions

This thesis makes seven contributions to software engineering practitioners and researchers.

First, the main contribution is the validated approach for continuous rationale management consisting of the *ConRat life cycle model extension* and the *ConDec plug-ins* (Chapter 5, addressing the technical research goal). Adopting continuous rationale management is particularly interesting for practitioners since managing rationale has many positive effects, such as improved decision-making and change process, knowledge sharing, reuse, and accountability. ConRat shows how practitioners can integrate rationale-management activities into their current workflows (Chapter 6). Practitioners can easily extend their tools and systems with the views and features of the ConDec plug-ins (Chapter 7). ConRat and ConDec are also interesting for researchers as a basis for empirical studies on rationale management and further development.

Second, the thesis contributes an *interview study* with practitioners from the industry (Chapter 3, addressing knowledge goal 1). The interview study describes the current state of CSE as the context of the current rationale management. It contributes anecdotal evidence on the types of decisions worth capturing for practitioners, how they capture decision knowledge (documentation locations, techniques, tools, linked artifacts, evolution history of decisions, capturing practices and frequencies), what benefits they see in capturing decision knowledge, what hinders them from capturing decision knowledge, how they share decision knowledge and avoid knowledge vaporization, and how they deal with change. The interview study also contributes features for continuous rationale management beneficial for practitioners. The results of the interview study are interesting for practitioners to compare their current practices and to reflect on the necessity for adopting ConRat and ConDec. The interview study is also interesting for researchers, e. g., when performing future interview studies to compare their results.

Third, the thesis contributes a *systematic mapping study* of rationale management support with classification and recommendation (Chapter 4, addressing knowledge goal 2). The study contributes an overview of four approaches helpful for rationale management: Automatic text classification, automatic linking, decision guidance, and consistency support. The approaches were a basis for developing ConDec's recommendation systems. The overview is valuable for researchers to guide future systematic literature reviews and primary studies.

Fourth, this thesis contributes an *empirical study on the analysis of the knowledge documentation* of six case study projects (Chapter 9, addressing knowledge goal 3). This study validates the feasibility of ConRat and ConDec and contributes to understanding decision knowledge documentation in practice. The study reports and discusses various metrics on the knowledge documentation: a) types and numbers of decision knowledge elements at the end of the project and over time, b) documentation locations of decision knowledge, c) decision types and their correlation with other decision types and documentation locations, d) ratios between the number of decisions to requirements and code, e) intra-rationale completeness, f) states of issues and decisions, and g) links between code and tickets and decision coverage calculated on the links from requirements or code to decisions. The study is interesting as it shows how to operationalize the quality of knowledge documentation. Practitioners can reflect on the knowledge documentation quality of their projects using the metrics. The study is interesting for researchers as a basis for further work on developing guidelines and tool support for high-quality rationale documentation.

Fifth, this thesis contributes an *empirical study of the effectiveness of automatic text classification* (Chapter 10, addressing knowledge goal 4). The study reports precision, recall, and F-scores of experiments with various data sets and machine-learning algorithms. The results indicate the effectiveness, but researchers can further improve ConDec's automatic text classification in the future and use the study results as a benchmark.

Sixth, this thesis contributes an *empirical study on the user acceptance of ConDec* (Chapter 11, addressing knowledge goal 5). The study contributes feedback and improvement ideas for

the views and features of the ConDec plug-ins by software developers. The results show the acceptance and the improvement ideas are interesting for researchers as a basis for future work.

Seventh, this thesis contributes a *syllabus for the dissemination of ConRat and ConDec* (Chapter 12, addressing the instrument design goal). The syllabus consists of exercises where students perform rationale management activities using the ConDec views and features. It is interesting for practitioners as a starting point to adopt ConRat and ConDec.

1.6. Structure of the Thesis

This thesis is structured in five parts and 14 chapters. Table 1.2 shows an overview of the structure of the thesis. Part I introduces this thesis and describes the knowledge context. Part II contains the problem investigation that addresses the knowledge goal 1 and the knowledge goal 2. Part III describes the treatment design that addresses the technical research goal. Part IV describes the treatment validation that addresses the knowledge goal 3, the knowledge goal 4, and the knowledge goal 5. This part also describes the syllabus for the dissemination of ConRat and ConDec that addresses the instrument design goal. Part V provides a summary and an outlook of future work.

Table 1.2.: Structure of the thesis, including research goals and research questions (RQ).

		Preliminaries	Chapter
	Part I	Introduction	1
		Background	2
Design Cycle		Problem Investigation	
	Part II	**State of the Practice: Rationale Management during CSE** *Knowledge Goal 1:* Understand the current practices, problems, and practitioners' improvement ideas regarding rationale management in CSE. *RQ:* How do practitioners apply CSE during software evolution? *RQ:* How do practitioners manage decision knowledge during CSE? *RQ:* How can rationale management in CSE be improved according to practitioners?	3
		State of the Art: Classification and Recommendation for Rationale Management *Knowledge Goal 2:* Understand the current state of the art regarding rationale management support with classification or recommendation. *RQ:* What are the characteristics of (semi-)automatic classification and recommendation approaches to support rationale management?	4
		Treatment Design	
	Part III	*Technical Research Goal:* Design a life cycle model and tool support for continuous rationale management that treats the problems of 1) intrusiveness and effort, 2) high amount of distributed knowledge, and 3) low documentation quality. The goal is to support a) collaborative, incremental, and rational decision making, b) documentation, c) exploitation, and d) quality assurance of decision knowledge. *Sub-Goal:* Support low-intrusive decision making, documentation, and exploitation *Sub-Goal:* Support high amount of distributed knowledge *Sub-Goal:* Support high-quality decision knowledge documentation	
		Overview of Continuous Rationale Management and its Support with ConDec	5
		Life Cycle Modeling of Continuous Rationale Management	6
		Supporting Continuous Rationale Management with ConDec	7
		Treatment Validation	
	Part IV	**Overview of Evaluation Studies**	8
		Analysis of Knowledge Documentation *Knowledge Goal 3:* Show that it is feasible to document a high amount of high-quality rationale during ConRat with the ConDec plug-ins. Describe the outcome of knowledge documentation in practice. *RQ:* Is it feasible to document decision knowledge in practice with ConDec? *RQ:* Is it feasible to document a high amount of knowledge in practice with ConDec? *RQ:* Is it feasible to create high-quality knowledge documentation in practice with ConDec?	9
		Effectiveness of Automatic Text Classification *Knowledge Goal 4:* Show the effectiveness of automatic text classification from the researchers' perspective. *RQ:* How effective is the automatic text classification of ConDec at identifying rationale elements?	10
		User Acceptance of ConDec Plug-Ins *Knowledge Goal 5:* Show the acceptance of the ConDec plug-ins from the software practitioners' perspective. *RQ:* Do developers accept the ConDec support for decision making? *RQ:* Do developers accept the ConDec support for knowledge documentation? *RQ:* Do developers accept the ConDec support for knowledge exploitation? *RQ:* Do developers accept the ConDec support for quality assurance?	11
		Dissemination of ConRat and ConDec Plug-Ins *Instrument Design Goal:* Disseminate ConRat and the ConDec plug-ins to developers and show the acceptance of the dissemination. *RQ:* Do developers accept the dissemination?	12
		Conclusion	
	Part V	Summary and Future Work	13, 14

1.7. Previous Publications

Several results of this dissertation have already been published. Table 1.3 lists the publications in chronological order, including a reference to the corresponding chapters of the thesis.

Table 1.3.: Previous publications sorted by publication date.

	Publication	Chapter
Johanssen et al., 2017a	J. O. Johanssen, A. Kleebaum, B. Bruegge, and B. Paech (2017a). "Towards a Systematic Approach to Integrate Usage and Decision Knowledge in Continuous Software Engineering". In: *2nd Workshop on Continuous Software Engineering*. Hannover, Germany, pp. 7–11	2
Johanssen et al., 2017b	J. O. Johanssen, A. Kleebaum, B. Bruegge, and B. Paech (2017b). "Towards the Visualization of Usage and Decision Knowledge in Continuous Software Engineering". In: *5th IEEE Working Conference on Software Visualization (VISSOFT 2017)*. Shanghai, China, pp. 104–108. DOI: 10.1109/VISSOFT.2017.18	7
Johanssen et al., 2018	J. O. Johanssen, A. Kleebaum, B. Paech, and B. Bruegge (2018). "Practitioners' Eye on Continuous Software Engineering: An Interview Study". In: *International Conference on Software and System Process*. ICSSP '18. Gothenburg, Sweden: ACM, pp. 41–50. DOI: 10.1145/3202710.3203150	3
Kleebaum et al., 2018a	A. Kleebaum, J. O. Johanssen, B. Paech, R. Alkadhi, and B. Bruegge (2018a). "Decision knowledge triggers in continuous software engineering". In: *4th International Workshop on Rapid Continuous Software Engineering - RCoSE '18*. Gotheburg, Sweden: ACM Press, pp. 23–26. DOI: 10.1145/3194760.3194765	6
Kleebaum et al., 2018b	A. Kleebaum, J. O. Johanssen, B. Paech, and B. Bruegge (2018b). "Tool Support for Decision and Usage Knowledge in Continuous Software Engineering". In: *3rd Workshop on Continuous Software Engineering*, pp. 74–77. DOI: 10.11588/heidok.00024186	7
Johanssen et al., 2019b	J. O. Johanssen, A. Kleebaum, B. Paech, and B. Bruegge (2019b). "The Eye of Continuous Software Engineering". In: *Software Engineering and Software Management (SE)*. Bonn, Germany: Gesellschaft für Informatik e.V., pp. 67–68. DOI: 10.18420/se2019-17	3
Kleebaum et al., 2019a	A. Kleebaum, J. O. Johanssen, B. Paech, and B. Bruegge (2019a). "Teaching Rationale Management in Agile Project Courses". In: *16. Workshop Software Engineering im Unterricht der Hochschulen (SEUH)*. Bremerhaven, Germany, pp. 125–132. DOI: 10.11588/heidok.00026358	12
Kleebaum et al., 2019b	A. Kleebaum, J. O. Johanssen, B. Paech, and B. Bruegge (2019b). "How do Practitioners Manage Decision Knowledge during Continuous Software Engineering?" In: *31st International Conference on Software Engineering and Knowledge Engineering*. SEKE'19. Lisbon, Portugal: KSI Research Inc., pp. 735–740	3
Johanssen et al., 2019c	J. O. Johanssen, A. Kleebaum, B. Paech, and B. Bruegge (2019c). "Continuous software engineering and its support by usage and decision knowledge: An interview study with practitioners". In: *Journal of Software: Evolution and Process (JSEP)* 31.5, e2169. DOI: 10.1002/smr.2169	3

Continued on next page

	Publication	Chapter
Kleebaum et al., 2019c	A. Kleebaum, J. O. Johanssen, B. Paech, and B. Bruegge (2019c). "Sharing and Exploiting Requirement Decisions". In: *Fachgruppentreffen Requirements Engineering (FGRE)*. Heidelberg, Germany: Gesellschaft für Informatik, pp. 19–20. DOI: 10.11588/heidok.00028596	7
Kleebaum et al., 2019d	A. Kleebaum, M. Konersmann, M. Langhammer, B. Paech, M. Goedicke, and R. Reussner (2019d). "Continuous Design Decision Support". In: *Managed Software Evolution*. Ed. by R. Reussner, M. Goedicke, W. Hasselbring, B. Vogel-Heuser, J. Keim, and L. Märtin. Cham: Springer International Publishing. Chap. 6, pp. 107–139. DOI: 10.1007/978-3-030-13499-0_6	6
Kleebaum et al., 2020	A. Kleebaum, J. O. Johanssen, B. Paech, and B. Bruegge (2020). "Continuous Management of Requirement Decisions Using the ConDec Tools". In: *26th International Conference on Requirements Engineering (REFSQ20) Workshops, Doctoral Symposium, Live Studies Track, and Poster Track*. Pisa, Italy: CEUR-WS.org, p. 6. DOI: 10.11588/heidok.00028230	5, 7
Kleebaum et al., 2021a	A. Kleebaum, J. O. Johanssen, B. Paech, and B. Bruegge (2021a). "Continuous Rationale Management Using the ConDec Tools". In: *Software Engineering 2021 Satellite Events*. Ed. by S. Götz, L. Linsbauer, I. Schaefer, and A. Wortmann. Braunschweig/Virtual: CEUR-WS, pp. 1–2. DOI: 10.11588/heidok.00029976	7
Kleebaum et al., 2021b	A. Kleebaum, B. Paech, J. O. Johanssen, and B. Bruegge (2021b). "Continuous Rationale Identification in Issue Tracking and Version Control Systems". In: *REFSQ-2021 Workshops, OpenRE, Posters and Tools Track, and Doctoral Symposium*. Essen/Virtual: CEUR-WS.org, p. 9. DOI: 10.11588/heidok.00029966	7, 10
Kleebaum et al., 2021c	A. Kleebaum, B. Paech, J. O. Johanssen, and B. Bruegge (2021c). "Continuous Rationale Visualization". In: *Working Conference on Software Visualization (VISSOFT)*. Luxembourg: IEEE, pp. 33–43. DOI: 10.1109/VISSOFT52517.2021.00013	7, 11

Chapter 2

Background

"Formalizing knowledge is costly. One way to reduce cost is to formalize it incrementally—essentially transforming a semiformal representation to a formal one."

—J. Lee, 1997

This chapter introduces important terms used in this dissertation. Section 2.1 presents continuous software engineering as the software development process underlying Continuous Rationale Management (ConRat). Section 2.2 provides an overview of the topic of rationale management. Section 2.3 introduces relevant development tools and systems mentioned throughout this dissertation and extended by the ConDec plug-ins. Section 2.4 describes the CURES research project and the CURES prototype to improve rationale and usage knowledge management during CSE. The CURES prototype was a basis for the interview study described in the following chapter.

2.1. Continuous Software Engineering

The term *software engineering* was introduced in 1968 at a conference where issues related to software were addressed, and best practices for software development were established (Naur and Randell, 1968). The vision was to understand how software is developed and to define a software life cycle model (Mahoney, 1990). Lehman (1980) stated that the continual change to a software system, i. e., its evolution and never-ending maintenance, is necessary because of a changing environment and new and changing stakeholder requirements. *Continuous Software Engineering (CSE)* describes a family of agile software development processes. This thesis uses the terms *process* and *life cycle* interchangeably.

Section 2.1.1 presents the CSE process model called *Stairway to Heaven*. Section 2.1.2 presents the CSE process model called *BizDevOps*. Both models are a foundation for the interview study described in Chapter 3. Section 2.1.3 presents the *Rugby CSE process model*, which is a foundation for ConRat described in Part III of the thesis.

2.1.1. Stairway to Heaven

The transition from the waterfall-model-style software development to CSE is described in four steps, metaphorically named the *Stairway to Heaven*. In the first step, the process involves agile practices such as working with sprints, but feedback loops with the customers still take long (six months or more). The second step employs the frequent integration of work, daily builds, and fast commit of changes. The third step adopts continuous deployment to release software

Table 2.1.: Continuous * activities and their definitions by Fitzgerald and Stol (2017).

Activity	**Description** (see Fitzgerald and Stol (2017) for original references)
	Business strategy and planning
Continuous planning	Holistic endeavor involving multiple stakeholders from business and software functions whereby plans are dynamic, open-ended artifacts that evolve in response to changes in the business environment and thus involve a tighter integration between planning and execution.
Continuous budgeting	Budgeting is traditionally an annual event during which an organization's investments, revenue, and expense outlook are prepared for the coming year. The Beyond Budgeting model suggests that budgeting becomes a continuous activity to facilitate changes during the year.
	Development
Continuous integration	A typically automatically triggered process comprising inter-connected steps such as compiling code, running unit and acceptance tests, validating code coverage, checking coding-standard compliance, and building deployment packages. While some form of automation is typical, the frequency is also important because it should be regular enough to ensure quick feedback to developers. Finally, any continuous integration failure is also an important event that may have several ceremonies and highly visible artifacts that help to ensure that problems leading to integration failures are solved as quickly as possible by those responsible.
Continuous delivery	Continuous delivery is the practice of continuously deploying good software builds automatically to some environment but not necessarily to actual users.
Continuous deployment	Continuous deployment implies continuous delivery and ensures that the software is continuously ready for release and deployed to actual customers.
Continuous verification	Adoption of verification activities, including formal methods and inspections throughout the development process rather than relying on a testing phase towards the end of development.
Continuous testing	A process typically involving some automation of the testing process, or prioritization of test cases, that helps to reduce the time between the introduction of errors and their detection and eliminate root causes more effectively.
Continuous compliance	Software development seeks to satisfy regulatory compliance standards continuously, rather than operating a *big-bang* approach to ensuring compliance just before the release of the overall product.
Continuous security	Transforming security from being treated as just another non-functional requirement to a key concern throughout all phases of the development life cycle and even post-deployment, supported by a smart and lightweight approach to identifying security vulnerabilities.
Continuous evolution	Most software systems evolve during their lifetime. However, a system's architecture is based on a set of initial design decisions that were made during the system's creation. Some assumptions underpinning these decisions may no longer hold, and the architecture may not facilitate specific changes. Technical debt is incurred when an architecture is unsuitable for facilitating new requirements, but shortcuts are made.
	Operations
Continuous use	Recognizes that the initial adoption versus continuous use of software decisions are based on different parameters and that customer retention can be a more effective strategy than trying to attract new customers.
Continuous trust	Trust developed over time as a result of interactions based on the belief that a vendor will act cooperatively to fulfill customer expectations without exploiting their vulnerabilities.
Continuous run-time monitoring	As the historical boundary between design-time and run-time research in software engineering is blurring, in the context of continuously running cloud services, run-time behaviors of all kinds must be monitored to enable early detection of quality-of-service problems, such as performance degradation, and also the fulfillment of service level agreements.
	Improvement and Innovation
Continuous improvement	Based on lean principles of data-driven decision making and waste elimination, which lead to small incremental quality improvements that can have dramatic benefits and are hard for competitors to emulate.
Continuous innovation	A sustainable process responsive to evolving market conditions and based on appropriate metrics across the entire life cycle of planning, development, and run-time operations.
Cont. experimentation	A software development approach based on experiments with stakeholders consisting of repeated *build-measure-learn* cycles.

functionality to the customers frequently. The fourth step allows acting on customer feedback immediately. Software deployment is now the starting point for continuous tuning rather than delivering a final product (Olsson et al., 2012; Bosch, 2014).

2.1.2. DevOps and BizDevOps

DevOps was introduced to emphasize the collaboration between development and operations (Ebert et al., 2016; Leite et al., 2019). DevOps requires the automation of software development (e. g., quality assurance) and delivery to enable frequent software releases (Ebert et al., 2016). Fitzgerald and Stol (2017) introduced the term *BizDevOps* to add business to development and operations. They define 16 CSE practices listed in Table 2.1 grouped into four categories: The *business strategy and planning* category contains continuous planning and continuous budgeting activities and emphasizes the need for integrating the business strategy and the development. The *development* category contains software development activities of analysis, design, coding, and testing. The *operation* category includes software usage and run-time monitoring activities. The *improvement and innovation* category contains activities related to software process improvement and encouraging new ideas to create value for customers.

2.1.3. Rugby CSE Life Cycle Model

Krusche and Bruegge introduced the *Continuous Software Engineering Process Metamodel (CSEPM)* and the *Rugby process model* as an instance of the CSEPM (Krusche et al., 2014; Krusche, 2016; Krusche and Bruegge, 2017). The Rugby process combines concepts from *Scrum* (Schwaber and Beedle, 2002) and the *Unified Process* (Jacobson et al., 1998). Figure 2.1 shows a dynamic view of the Rugby process model.

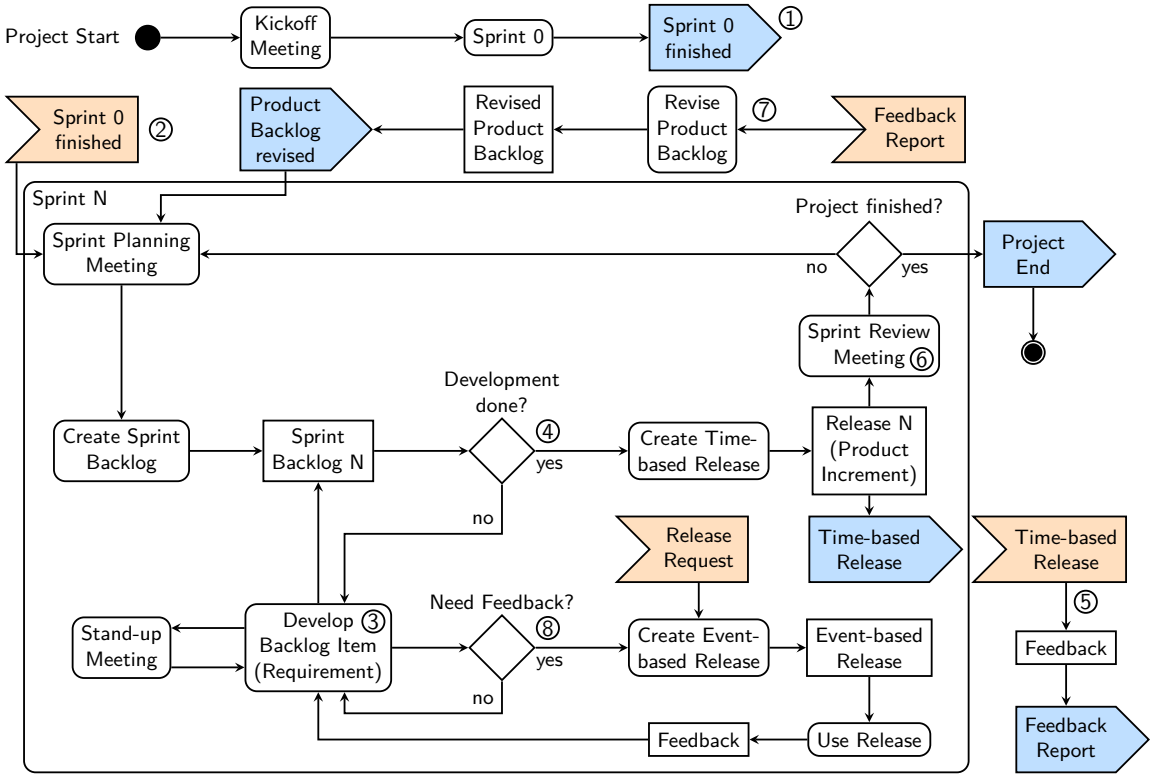

Figure 2.1.: Rugby CSE process model as a UML activity diagram by Krusche (2016) and Krusche and Bruegge (2017).

The model is represented using the Unified Modeling Language (UML) specified by the Object Management Group (2017). It is an *activity-centered life cycle model*, which also shows the entities produced and consumed by the activities and actions (Bruegge and Dutoit, 2010).

The Rugby process consists of parallel workflows that are activated and interrupted through change events. The parallel workflows are modeled as UML activity diagrams, and the change events are represented using UML signals. The incoming event activates a workflow and the outgoing event notifies other workflows about the finished work.

When performing Rugby, the developers start a project with a *kickoff meeting* and perform the *sprint 0* as an upfront project phase. The goals of sprint 0 are to 1) provide a common basis of communication for all the developers, 2) elicit initial project requirements in the *product backlog* and make high-level architectural decisions, and 3) set up the workflows for reviews, releases, and feedback. The sprint 0 ends with the outgoing event *sprint 0 finished* (Figure 2.1-①), which triggers the respective incoming event (Figure 2.1-②). Afterward, every sprint starts with a *sprint planning meeting*, in which the developers *create a sprint backlog*. The sprint backlog contains requirements elicited from the customer and will be implemented by the developers. Because the requirements can only be vaguely specified at the beginning of the sprint, developers detail the requirements specification during the sprint while they implement the backlog items (Figure 2.1-③). Like in Scrum, the team discusses the status, impediments, and promises regarding the backlog items in daily *stand-up meetings*. At the end of each sprint (Figure 2.1-④), the developers create a *time-based release* for the *product increment*. In a parallel workflow, the *product owner* uses the product increment and provides *feedback* (Figure 2.1-⑤). The *sprint review meeting* marks the end of the sprint (Figure 2.1-⑥). During the sprint review meeting, a *feedback report* can trigger a *revision of the product backlog* (Figure 2.1-⑦). The *revised product backlog* is then used for the next *sprint planning meeting*. Rugby–different from Scrum—allows to *create event-based releases* (Figure 2.1-⑧), which will enable users to provide feedback even during the sprint (Krusche, 2016; Krusche and Bruegge, 2017).

Figure 2.2 shows the six parallel workflows in Rugby and their synchronization through change events. The flow final node ⊗ models that a workflow is set to sleep. The workflows are a requirements elicitation workflow, a development workflow, a review workflow, a release workflow, a feedback workflow, and a usage workflow.

Workflows can be seen as threads of work practices, which means that they model the tasks of the roles in a project. The role involved in the *requirements elicitation workflow* (Figure 2.2-①) is the *product owner* who prioritizes *new backlog items*, i. e., requirements, and *specifies acceptance criteria* so that the requirement is ready for development. In the *development workflow* (Figure 2.2-②), the developers analyze, design, implement, and test in parallel to produce changes in the source code and documentation. If the developers need feedback during the work on the backlog item, they *request a release*. After finishing a backlog item, they *request a merge*. The *merge request* activates the *review workflow* (Figure 2.2-③). If the quality criteria are met, the reviewer *accepts the changes* resulting in the *backlog item finished event*. Otherwise, the reviewer *requests improvements*, which reactivates the *development workflow*. The *release request* activates the *release workflow* (Figure 2.2-④), which results in an *event-based release*. Releases can be delivered to test environments before the end users can use them. The *event-based release* activates the *usage workflow* (Figure 2.2-⑤). If the users want to provide feedback, they create a *feedback report*. The *feedback report* or any external *change request* activates the *feedback/change management workflow* (Figure 2.2-⑥) and provides developers with the task to *change an existing functionality* or a *new backlog item*, i. e., new requirement.

The CSEPM allows tailoring the CSE process model. The development team can add new workflows or customize the Rugby workflows shown in Figure 2.2. In Chapter 6, Rugby is tailored with ConRat, i. e., extended with explicit rationale management.

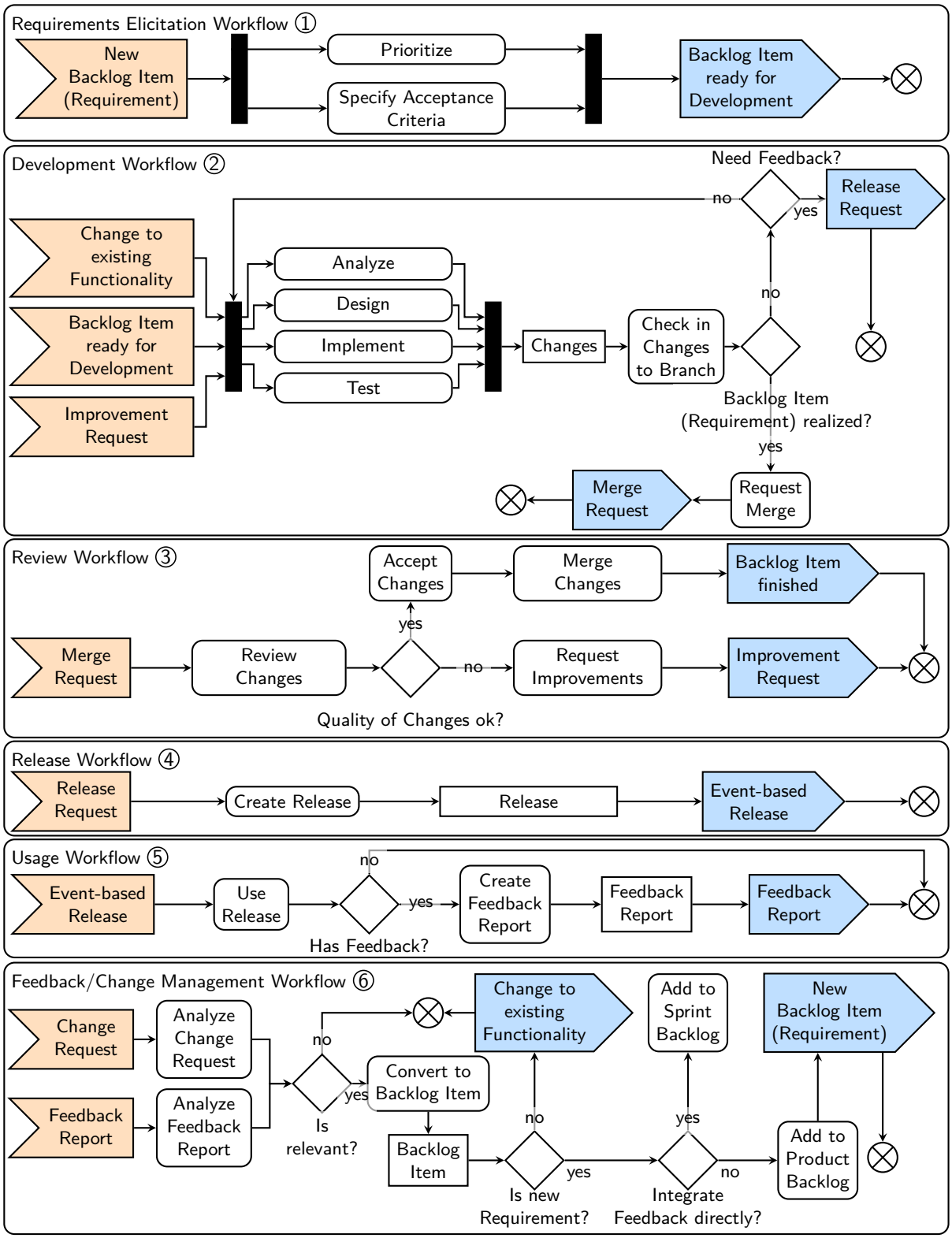

Figure 2.2.: Dynamic view of the synchronization of Rugby's parallel workflows through change events by Krusche (2016) and Krusche and Bruegge (2017) as UML activity diagrams.

2.2. Rationale Management

This section provides the foundations of rationale management. Section 2.2.1 describes its relationship to knowledge management and other knowledge types as a foundation for the knowledge model described in Chapter 6. Section 2.2.2 describes the naturalistic and rational decision-making strategies. ConRat and ConDec support both strategies. Section 2.2.3 describes the formalization and personalization knowledge-management strategies. ConRat and ConDec focus on the formalization strategy. Section 2.2.4 introduces the rationale model used in this thesis in a historical context.

A thorough introduction to rationale management is also provided by Dutoit et al. (2006), Burge et al. (2008), Bruegge and Dutoit (2010), Alkadhi (2018), and Hesse (2020).

2.2.1. Types of Knowledge and Knowledge Management

Decision knowledge, also referred to as *rationale* (Babar et al., 2009), is built up by requirements engineers, product owners, developers, and other stakeholders of other roles while they elicit, prioritize, document, validate, manage, implement, and verify requirements. For example, the stakeholders make decisions regarding the software development process, the existence and non-existence of software artifacts from all phases of software development, or the software quality (Kruchten, 2004). That means the stakeholders' decision knowledge, i. e., their rationale, justifies why a software system or a process is the way it is. Rationale is the motivation behind a decision (Bruegge and Dutoit, 2010).

Rationale management is a type of *knowledge management* because the stakeholders document and use decision knowledge (Dutoit et al., 2006; Burge et al., 2008).

Another type of knowledge management is *architectural knowledge management*, which addresses the management of *architectural knowledge* (Babar et al., 2009; Capilla et al., 2016). Architectural knowledge covers both the outcomes of a design and the major architectural decisions—also referred to as *architectural design decisions*—that led to it and their rationale (Bass et al., 2003; Kruchten et al., 2006). The architectural design decisions describe the software architecture (Jansen and Bosch, 2005). Historically, much software engineering research focused on architectural design decisions as these decisions are difficult to change in the future and thus very important. In this thesis, we use the term *rationale* to express *software engineering rationale*, i. e., we do not restrict the type (Dutoit et al., 2006; Burge et al., 2008).

Paech et al. (2014) distinguish three types of knowledge: system knowledge, project knowledge, and decision knowledge. To set architectural knowledge in context: architectural knowledge comprises system and decision knowledge regarding software architecture. *System knowledge* concerns the software itself, e. g., requirements, design, code, and test cases. *Project knowledge* involves the knowledge about the software development and process in a project, e. g., development tasks (work items), commits, and pull requests. Among these types of knowledge, decision knowledge is the most complex information the stakeholders generate, and thus, is the most difficult to maintain and update (Bruegge and Dutoit, 2010).

Knowledge is also distinguished into *design-time* and *run-time knowledge*. The stakeholders build up design-time knowledge when creating a software system while they build up run-time knowledge when operating it. *Usage knowledge* is a type of run-time knowledge that is particularly important during CSE to continuously validate the requirements against users' needs (Johanssen, 2019). Usage knowledge is the stakeholders' knowledge of how users apply software.

2.2.2. Implicit versus Explicit Knowledge and Decision Making Strategies

Implicit knowledge (also called *tacit* knowledge) is difficult to articulate and write down (Nonaka and Takeuchi, 1995; Kruchten et al., 2006). Similar to skills such as riding a bicycle, it is

acquired through personal experience (Hansen et al., 1999). When using implicit knowledge, the stakeholders make decisions in a *naturalistic way* (Zannier et al., 2007) because they are unaware of the decisions (Kruchten et al., 2006). Naturalistic decision making is more of an art than a craft because the reasoning process is rather ad-hoc (Capilla et al., 2016). Kahneman (2011) defines the concept of *system one thinking*, which involves shortcuts in thinking that make complicated problems tractable. Naturalistic decision making is related to system one thinking and can involve cognitive biases (Maule, 2010). Examples of *cognitive biases* are *anchoring* and *confirmation bias* as human issues that can lead to decisions that incur debt (Razavian et al., 2016; Soliman et al., 2021). Naturalistic decision making represents the dominant strategy in informal rationale documentation as observed by Hesse et al. (2016b).

Explicit knowledge is built up if stakeholders make a decision for a particular reason (Kruchten et al., 2006). *Rational decision making* means that stakeholders are aware of the decision, solution alternatives, and arguments (Zannier et al., 2007). Kahneman (2011) defines the concept of *system two thinking*, which involves rational decision making and explicit knowledge.

2.2.3. Knowledge Formalization versus Personalization

Formalized knowledge is documented and organized in a systematic way (Nonaka and Takeuchi, 1995; Kruchten et al., 2006). Hansen et al. (1999) distinguish two strategies for knowledge management: codification, i. e., *formalization*, and *personalization*. Hansen et al. (1999) also refer to the knowledge formalization as the "people-to-documents approach". When applying the personalization strategy, the stakeholders rely on experts' implicit knowledge and face-to-face communication for knowledge sharing. While the software engineering research community focuses on the formalization strategy, the practitioners often use a personalization strategy (Babar et al., 2007). Rationale management aims to make implicit decision knowledge explicit and formalizes it so stakeholders can share it without misunderstanding. The stakeholders should only formalize what is subsequently valuable for persons who exploit the knowledge because the effort of documenting rationale can outweigh its benefits. The benefits are felt later or by others (Kruchten et al., 2006). Rationale can also be *captured informally*, e. g., as natural language text, using drawings, and in video or audio recordings (J. Lee, 1997). The formal representation has the advantage that later access and exploitation are easier; for example, a tool can quickly provide an overview of the documented decisions. However, formal rationale documentation requires additional effort, and it is unnatural to break one's thoughts into discrete units (Jarczyk et al., 1992). A *wicked problem* is a decision problem that cannot be solved algorithmically but needs discussion and creativity (Rittel, 1972). Many decision problems in software engineering are wicked, as can be illustrated with the problem of this thesis *How to design tool support for continuous rationale management?*, which cannot be easily solved. Formalizing wicked problems and the related rationale is difficult. To benefit from the lightweight capture of an informal representation and the easier exploitation of a formal representation, an informal rationale representation can be formalized incrementally (J. Lee, 1997).

2.2.4. Rationale Representation

This section describes rationale models to document rationale in a formalized way. It presents three historical models and the model used in the thesis. A *rationale model* represents decision knowledge comparable to how a system model represents a system (Bruegge and Dutoit, 2010). An instance of a rationale model is a graph of *rationale elements*, i. e., vertices, and edges that represent the elements' *associations*. Rationale models differ in the types of rationale elements and associations they prescribe. In some cases, the types only differ by name but represent the same content, e. g., *position*, *alternative*, *proposal*, and *option*.

Kunz and Rittel (1970) introduced the *Issue-Based Information System* model. It consists of three types of rationale elements: The *issue* describes the problem that needs to be solved, one or more *positions* describe options to solve the issue, and *arguments*. Various associations exist between these rationale elements; for example, an argument can *support* or *object to* a position. MacLean et al. (1991) introduced the *Questions, Options, and Criteria* model. It consists of four types of rationale elements: The *question* describes the problem that needs to be solved, and *options* are possible solutions. These element types equal the *issue* and *position* types in the *Issue-Based Information System* model, respectively. The model also contains *arguments*. The fourth and new type of rationale element is the *criterion*. The *Questions, Options, and Criteria* model consists of similar associations as the *Issue-Based Information System* model plus additional associations involving the criterion. Associations express whether options are assessed negatively and positively against criteria. At the same time, J. Lee (1991) presented the *Decision Representation Language* that contains *goals*, *claims*, and *procedures* for answering questions as new types of elements.

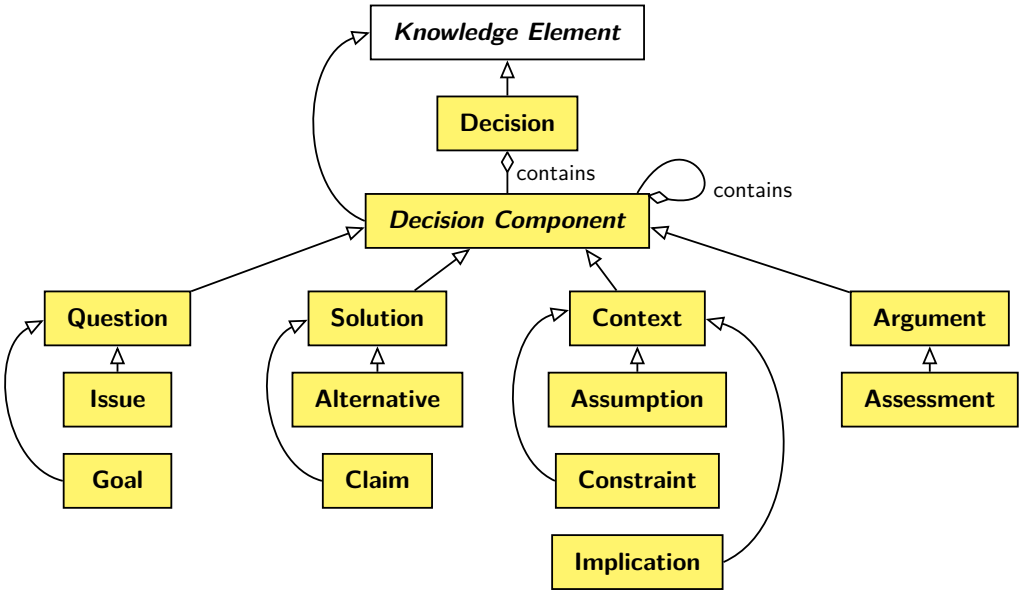

Figure 2.3.: *Decision Documentation Model* by Hesse and Paech (2013) and Hesse (2020) as a UML class diagram without attributes.

The *Decision Documentation Model* by Hesse and Paech (2013) and Hesse (2020) contains the rationale element types of the *Issue-Based Information System*, the *Questions, Options, and Criteria*, and the *Decision Representation Language* models. The Decision Documentation Model does not prescribe a fixed template for decision documentation and supports incremental documentation of decisions. Any part of the decision knowledge can be captured as soon as it is available. The Decision Documentation Model documents decision knowledge as *Decision Component*s, which can be nested and referred to other knowledge. Nesting enables retrospectively reconstructing rationale in natural language discussions. Other *Knowledge Element*s are software artifacts of project or system knowledge, such as, requirements, architectural design, code, and test cases. Figure 2.3 shows the decision components of the Decision Documentation Model. The *Decision Component* is an abstract class that can only be instantiated through its sub-classes. Decision components are the Question, i. e., decision problem, to be solved (Issues or Goals), Solution (Alternatives or Claims), Context information (Assumptions, Constraints, or Implications), and Arguments (Assessments). The Decision Documentation Model supports incremental documentation of decisions, in particular, naturalistic and rational decision making. Parts of the decision knowledge can be captured as soon as they are available. In addition,

stakeholders, such as developers, architects, and requirements engineers, can collaborate while documenting decisions. Each stakeholder contributes that part of the decision knowledge they know best. The requirements engineer can, for example, add constraints, which must be reflected for a particular solution. The Decision Documentation Model has been applied in an empirical study on Firefox issue reports, which showed that the model could reflect the decision knowledge captured in issue tracking systems (Hesse et al., 2016b). The dominance of naturalistic decision making in the results of this study confirms the need for incremental and collaborative decision documentation. In addition, the Decision Documentation Model has been applied in a case study on design session transcripts, which confirmed that the model reflects decision making in a team (Hesse and Paech, 2016). The instantiation of the model made complex decision knowledge structures in the design sessions explicit, with decisions containing more than 20 decision knowledge elements.

In addition to the different types of rationale elements, various *types of decisions* exist. Kruchten (2004) introduced a decision taxonomy: An *existence decision* indicates that some element or artifact will exist in the system. Existence decisions are divided into two subgroups: *Structural decisions* lead to the implementation of new components or subsystems. *Behavioral decisions* describe the interaction of elements, e. g., *The client shall communicate with the backend via REST API!* Existence decisions are the least important to capture as they are the most visible in a system. However, Kruchten argues that it is still important to document existence decisions, as they can relate to less obvious decisions or alternatives. A *ban* or *non-existence decision* states that a particular element does not appear in the system, e. g., *The system does not use a relational database for data storage!* These decisions must be captured because they cannot be reconstructed from the system. A *property decision* concerns the system's quality and can be guidelines, design rules, or constraints. It is either positively expressed (rules and guidelines) or negatively expressed (constraints). Property decisions are hard to trace to specific software artifacts because they are often cross-cutting concerns. An example is *The classifier must have a precision over 0.42!* An *executive decision* concerns the software development process, technologies, or applied tools. It does not directly relate to software artifacts or their underlying qualities. For example, a process decision is that *The change control board must approve changes!* Executive decisions often have a high impact since they constrain existence and property decisions. van der Ven and Bosch (2013) classify decisions according to their level: *High level decisions* affect the whole product. *Medium level decisions* concern specific components or frameworks and are relatively expensive to change. *Realization level decisions* concern the structure of the code or other specific aspects, such as API usage. Kruchten introduced association types between decisions with different semantic meanings: *enables*, *constrains*, *forbids*, *comprises*, *subsumes*, *overrides*, *conflicts with*, and *relates* (Kruchten, 2004; Kruchten et al., 2006; Kruchten, 2009). For example, a decision *Implement the algorithm in C++!* could *override* the decision *Implement the software in Java!*

Decisions and other rationale elements have *several attributes* (Kruchten, 2004; Kruchten, 2009; van Heesch et al., 2012; van der Ven and Bosch, 2013). The *epitome* of a rationale element describes its content. The *state* attribute expresses whether a solution option is an idea, decided, or rejected or whether an issue is solved or unsolved. The *author* attribute represents the person who made the decision. The *time-stamp* and *history* attributes capture when the decision was documented and the sequence of changes. The attributes *group* or *category* organize decisions according to concerns. The *scope* or *decision level* attributes express the granularity of a decision. Other attributes are *cost* and *risk*. Risk combines impact and likelihood factors.

2.3. Development Tools and Systems

Developers capture rationale in various documentation locations, for example, in the issue tracking system (Rogers et al., 2015; Hesse et al., 2016b; Bhat et al., 2017b), the version control system (van der Ven and Bosch, 2013), the wiki system, and using their integrated development environment. *Issue tracking systems* are widely applied to manage requirements, bug reports, and development tasks. They contain various information types such as functionality or quality requests and *as-is* descriptions (Merten et al., 2015). *Version control systems* are used to track software version history and for collaborative work on branches. A widely used version control system is *git* (Chacon and Straub, 2014). *Wiki systems* are collaborative writing tools that can be applied for various purposes, such as meeting management and requirements elicitation (Solis and Ali, 2010). *Integrated development environments* support software developers in writing code in the source code editor and provide valuable tools, such as debuggers and local test environments. A popular open-source integrated development environment is Eclipse.

2.4. Continuous Usage- and Rationale-based Evolution Decision Support (CURES)

Continuous Usage- and Rationale-based Evolution Decision Support (CURES) was a research project that was part of the Priority Programme 1593[1] of the German Research Foundation (DFG). To investigate the management of decision knowledge and usage knowledge during CSE, the CURES project was conducted between the years 2016 to 2019. The topic of this dissertation arose from the CURES project. CSE supports the developers in building up usage knowledge through the rapid release of new increments. For example, continuous deployment enables users to give feedback on the latest software version constantly.

As part of the CURES project, a prototype was developed to investigate the integration of decision and usage knowledge (Johanssen et al., 2017a; Johanssen et al., 2019c; Johanssen, 2019). The prototype models a CSE infrastructure with knowledge exploitation and documentation. The basic idea was to handle the evolution of documented knowledge like code evolution. Besides the individual benefits of decision and usage knowledge, the prototype expects synergies in their combination: in particular, to track the evolution of decisions based on user feedback.

Figure 2.4 depicts the stakeholders and components of the prototype. Developers and Users are the main stakeholders. Developers create feature branches in the CSE Infrastructure to add product increments in the form of code commits. Each feature branch can be released to users, allowing the delivery of different proposals for one feature simultaneously. Feature branches can be merged into the master branch containing the final software product. The Monitoring and Feedback component enables developers to examine usage information mapped to individual releases. They can apply A/B testing to decide between different software increments. The Knowledge Repository stores all information related to the development and monitoring process. In particular, it maintains decisions related to feature branches. The rationale can be accessed, visualized, and analyzed using a Dashboard component. For instance, the claimed solution to an issue could be compared to an alternative solution based on the impact of a feature branch in the context of previous decisions. This enables the developer to interact and reflect on collected decision and usage knowledge. For example, a feature F_1 is developed based on a proposal P_1. After providing the release R_1 to users and analyzing feedback FB_1, the developer makes decision D_1 and merges F_1 into the master branch. In the same manner, different proposals P_{2a} and P_{2b} for a feature F_2 are evaluated in A/B testing. Feedback FB_2 stops the work on P_{2a}, while FB_3 leads to decision D_2.

[1] SPP1593 website: http://www.dfg-spp1593.de

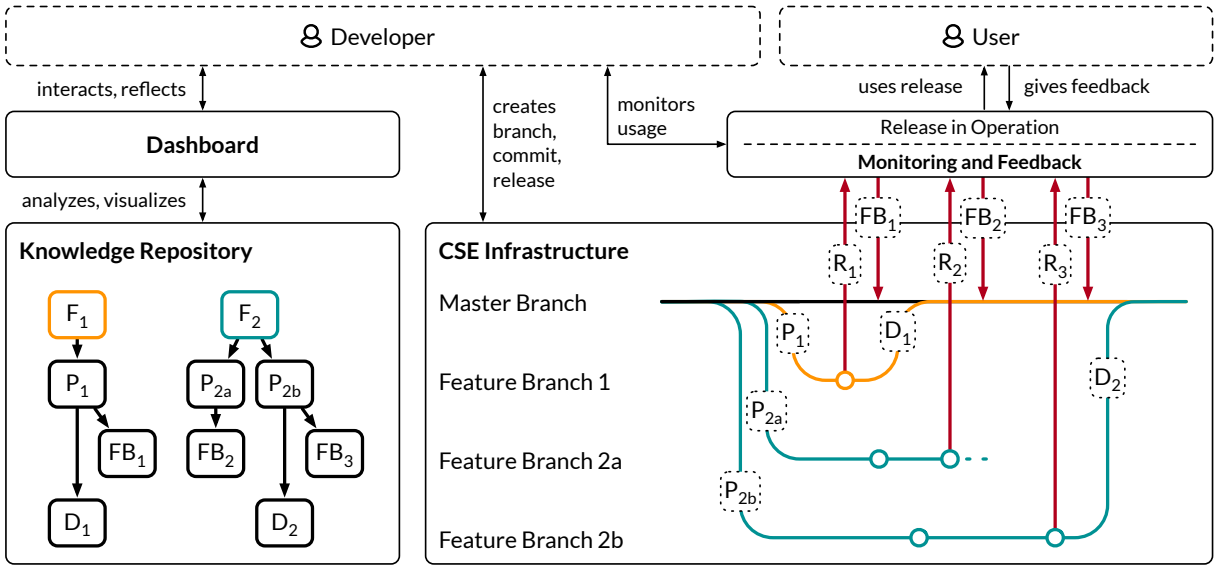

Figure 2.4.: Design of the CURES prototype (Johanssen et al., 2017a; Johanssen et al., 2019c; Johanssen, 2019).

Part II.

Problem Investigation

Chapter 3

State of the Practice: Rationale Management during Continuous Software Engineering

> *"Architectural decisions are like material floating in a pond. When not touched for a while, they sink and disappear from sight. These sunken decisions are the most difficult to change at a later stage. In particular, during evolution one may stumble upon these design decisions, try to undo them or work around them, and get into trouble when the change turns out to be very costly if not impossible. The future evolutionary capabilities of a system can be better assessed if these assumptions were explicit."*
>
> —Roeller et al., 2006

This chapter contributes to the knowledge goal 1 of the thesis: *Understand the current practices, problems, and practitioners' improvement ideas regarding rationale management in CSE.* It presents a semi-structured interview study with practitioners from 17 companies using CSE. First, the study describes the current state of CSE in general to describe the context of the current rationale management. Second, the study describes the current practices and problems of the rationale management in CSE. The *Agile Manifesto* suggests preferring working software over comprehensive documentation as well as individuals and interactions over processes and tools (Beck et al., 2001). This study contributes insights on which types of decisions practitioners think are important to capture, how they capture decision knowledge, what benefits they see in capturing decision knowledge, what hinders them from capturing decision knowledge, how they share decision knowledge, and how they deal with change. Third, the study describes the experts' opinions on improving the rationale management during CSE, i. e., on how to treat the rationale management problems.

Section 3.1 describes the study design. Section 3.2 provides and discusses the results of the interview study. Section 3.3 compares related empirical work on decision knowledge management in practice. Section 3.4 discusses threats to validity. Section 3.5 concludes this chapter.

Parts of the interview study were published: The results regarding RQ1 were published by Johanssen et al. (2018), Johanssen et al. (2019c), and Johanssen (2019). The results regarding RQ2 were published by Kleebaum et al. (2019b). Results regarding RQ3 were published by Johanssen et al. (2019c) and Johanssen (2019). This chapter describes the features and obstacles for continuous rationale management in more detail and omits the usage knowledge-specific results. Appendix B provides an excerpt of anonymized interview statements by the practitioners.

3.1. Study Design

Section 3.1.1 introduces three research questions. Section 3.1.2 describes the interview study and Section 3.1.3 provides descriptive data of the CSE practitioners, i. e., the interviewees. Section 3.1.4 discusses the research perspective.

3.1.1. Research Questions

The knowledge goal 1 of the thesis is refined into three research questions with sub-questions (Table 3.1). The sub-questions were used as the interview questions in the questionnaire, except that they addressed the practitioners. In the following, we describe the questions.

Table 3.1.: Research questions of the interview study.

	Research Question
RQ1	How do practitioners apply CSE during software evolution?
RQ1.1	How do practitioners define CSE?
RQ1.2	Which elements of CSE are perceived as most relevant by practitioners?
RQ1.3	What are practitioners' experiences with CSE?
RQ1.4	What are practitioners' future plans for CSE?
RQ2	How do practitioners manage decision knowledge during CSE?
RQ2.1	Which decisions are captured by practitioners during CSE, why, and how?
RQ2.1a	Which types of decisions do practitioners capture?
RQ2.1b	Where do practitioners capture the decisions, with which techniques and tools?
RQ2.1c	Do practitioners link decisions and rationale to other software artifacts and if so, how?
RQ2.1d	How do practitioners preserve the evolutionary history of decisions?
RQ2.1e	When and how often do practitioners capture decisions?
RQ2.1f	Why do practitioners capture decisions, i. e., what are the benefits and what do they do with the captured decisions?
RQ2.2	Which important decisions are not captured during CSE, why not, and would it be beneficial to capture these decisions?
RQ2.2a	Which important decisions do practitioners not capture during CSE?
RQ2.2b	Why do practitioners not capture these decisions?
RQ2.2c	What would be the benefits if practitioners captured these decisions?
RQ2.3	How do practitioners share decision knowledge during CSE?
RQ2.3a	What are the knowledge sources from which practitioners retrieve necessary information for decisions that are not captured?
RQ2.3b	How do practitioners share knowledge to avoid knowledge vaporization?
RQ2.4	How do practitioners deal with changing decisions during CSE?
RQ3	How can rationale management in CSE be improved according to practitioners?
RQ3.1	Which tool features for continuous rationale management do practitioners perceive as beneficial and why?
RQ3.2	What obstacles do the practitioners perceive regarding the CURES prototype toward continuous rationale management?

RQ1 How do practitioners apply CSE during software evolution?

The first research question aimed to investigate the current state of CSE practice, in particular, how the practitioners define and apply CSE. We aim to understand the context of the current state of rationale management. We identified a set of elements that are typical for CSE based on the CSE descriptions presented in Section 2.1, in particular, the *Stairway to Heaven* (Olsson et al., 2012; Bosch, 2014) and the continuous star activities (Fitzgerald and Stol, 2017). We grouped the CSE elements into CSE categories. To address the involvement of users, we introduced *user* as a CSE category that refers to both customers and end-users. *Software management* includes practices concerning the overall software process. The *development* category contains development activities, such as requirements engineering and design, excluding implementation and quality assurance. The *code* category includes implementation-related activities, such as *version control* and *branching strategies*. Activities such as *audits* and *pull requests* were bundled in the *quality* category. The *knowledge* category was introduced to deal with practices supporting (decision) knowledge management. Table 3.2 shows the CSE elements and CSE categories used in the interview study.

Table 3.2.: CSE categories and CSE elements derived from Olsson et al. (2012), Bosch (2014), and Fitzgerald and Stol (2017).

CSE Category	CSE Elements
User	Involved users and other stakeholders; learning from usage data and feedback; proactive customers
Software Management	Agile practices; short development sprints; continuous integration of work; continuous delivery; continuous deployment of releases
Development	Continuous planning activities; continuous requirements engineering; focus on features; modularized architecture and design; fast realization of changes
Code	Version control; branching strategies; fast commit of code; code reviews; code coverage
Quality	Automated tests; regular builds; pull requests; audits; run-time adaption
Knowledge	Sharing knowledge; continuous learning; capturing decisions and rationale

The first research question is refined into four questions:

RQ1.1 *How do practitioners define CSE?* We aim to learn about the practitioners' perception of CSE. Further, we want to know whether practitioners define a threshold that needs to be passed before a company can claim to practice CSE.

RQ1.2 *Which elements of CSE are perceived as most relevant by practitioners?* To understand the perception of CSE in more detail, we asked the practitioners about the three CSE elements most relevant to them of the elements listed in Table 3.2.

RQ1.3 *What are practitioners' experiences with CSE?* This research question aims to reveal positive, neutral, and negative experiences with the CSE elements. This is particularly interesting to other practitioners who plan to adopt them.

RQ1.4 *What are practitioners' future plans for CSE?* We asked for planned short- and long-term additions to understand trends of future CSE elements adoption.

RQ2 How do practitioners manage decision knowledge during CSE?

The second research question investigates the current state of rationale management during CSE. It is also refined into four questions:

RQ2.1 *Which decisions are captured by practitioners during CSE, why, and how?* This question investigates approaches to capture decisions and rationale during CSE within companies explicitly. In particular, we include interview questions regarding techniques for linking and preserving evolutionary history to understand how practitioners deal with the distributed knowledge documentation and decision changes during CSE, respectively. Interview questions are: Which types of decisions do practitioners capture? Where do practitioners capture the decisions, and with which techniques and tools? Do practitioners link decisions and rationale to other software artifacts, and if so, how? How do practitioners preserve the evolutionary history of decisions? When and how often do practitioners capture decisions? Why do practitioners capture decisions, i. e., what are the benefits, and what do they do with the captured decisions?

RQ2.2 *Which important decisions are not captured during CSE, why not, and would it be beneficial to capture these decisions?* This question aims to find types of implicit decisions and reasons why they are not captured. Interview questions are: Which important decisions do practitioners not capture during CSE? Why do practitioners not capture these decisions? What would be the benefits if practitioners captured these decisions?

RQ2.3 *How do practitioners share decision knowledge during CSE?* We want to investigate how practitioners share decision knowledge during CSE with these questions: What are the knowledge sources from which practitioners retrieve necessary information for decisions that are not captured? How do practitioners share knowledge to avoid knowledge vaporization?

RQ2.4 *How do practitioners deal with changing decisions during CSE?* We aim to investigate how practitioners identify parts of the system affected by new or changed decisions.

RQ3 How can rationale management in CSE be improved according to practitioners?

The third research question investigates improvement ideas for rationale management during CSE according to the practitioners. It aims to identify ideas for the treatment design, i. e., for continuous rationale management as introduced in Chapter 1. To initiate this part of the interview and to encourage discussions, we presented the CURES prototype to the practitioners described in Section 2.4. The third research question is refined into two questions:

RQ3.1 *Which tool features for continuous rationale management do practitioners perceive as beneficial and why?* This question aims to identify tool features to support rationale management during CSE. This includes the features that the CURES prototype already covers and new features suggested by the practitioners. Interview questions are: What are the benefits of the CURES prototype perceived by practitioners? How could the CURES prototype be extended to improve its benefits and feasibility or to overcome the obstacles to its implementation? What are potential additions to the CURES prototype according to practitioners?

RQ3.2 *What obstacles do the practitioners perceive regarding the CURES prototype toward continuous rationale management?* By asking the practitioners for major obstacles of the CURES prototype, we detailed the feasibility of the proposed approach. Responses help to understand practitioners' problems. We aim to strengthen and identify problems of rationale management.

3.1.2. Interview Study Procedure

We performed a *semi-structured interview study* consisting of a *design and planning*, *data collection*, and *data analysis* phase (Runeson et al., 2012). Two researchers were equally involved.

During the *design and planning* phase, we prepared a questionnaire. Its first part addresses the practitioners' background and working context. Furthermore, it contained the interview questions listed below the respective research questions in Section 3.1.1. The interviews also included research questions that were not addressed in this thesis. They are presented by Johanssen et al. (2019a) and Johanssen (2019). We contacted companies that, to our knowledge, apply CSE.

During the *data collection* phase, we conducted 20 interviews between April and September of 2017, either in person or via phone. The interviews took 70 minutes on average and were audio-recorded with the permission of the interviewees. We transcribed the audio recordings and sent the transcripts to the interviewees to correct misunderstandings. We guaranteed the anonymity of the practitioners by publishing only aggregated results.

In the *data analysis* phase, we analyzed the transcripts (Saldaña, 2009). We utilized a *qualitative data analysis* software to apply two stages. During the first stage, we allocated answers to an interview question. Hereafter, we performed a fine-grained coding stage. To answer the interview questions concerning the types of decisions captured and not captured, we used Kruchten's taxonomy (Kruchten, 2004; Kruchten, 2009). This taxonomy distinguishes between existence decisions, non-existence decisions (bans), property decisions, and executive decisions (Section 2.2.4). For the remaining interview questions, we identified emerging topics and coded the answers regarding these topics. We analyzed the results quantitatively. If two interviewees participated in an interview, we treated their answers as one subject.

3.1.3. Participants

During 20 interviews, we interviewed 24 practitioners from 17 companies. Four companies are *small or medium-sized enterprises* with a maximum number of 250 employees.[1] Of the remaining companies, eight have up to 2000 employees, two have around 50 000 employees, and three have 100 000 or more employees. While seven companies provide consultancy services, ten develop software products. We grouped the 24 practitioners into five categories: *CSE specialists* (5), e. g., a continuous deployment manager or a DevOps engineer, *developers* (6), *project managers* (6), *technical leaders* (6), and one *executive director*. On average, the practitioners have worked two years in the respective role, ten years in IT projects, and in 19 IT projects.

3.1.4. Research Perspectives

Researchers are influenced by different *philosophical stances* when performing empirical studies (Easterbrook et al., 2008; Runeson et al., 2012; Wohlin and Aurum, 2015). The stance of positivists and constructivists influences the interview study.

Positivists infer knowledge from basic observable facts. They are also called reductionists because they use simplification and abstraction, e. g., by neglecting the human context. When performing the interview study from the positivist stance, we aim to infer a generalizable as-is state of rationale management in practice by interviewing various practitioners. The aim to maximize the generalizability characterizes the study as a *sample study* (Stol and Fitzgerald, 2018). A modern form of positivism is called *dataism*. *Dataists* believe they can make perfect simulations and predictions of any aspect of the world if the data set is big enough (Harari, 2016). However, this is unlikely for human behavior such as decision making (Spiekermann, 2019).

Constructivists think that scientific knowledge cannot be separated from its human context and aim to understand human activities in a specific situation. In this study, we report subjective

[1] `http://ec.europa.eu/growth/smes/business-friendly-environment/sme-definition`

opinions of practitioners as *anecdotal evidence*, i. e., findings that might not be generalizable to provide rich qualitative data. Thus, the study has also characteristics of a *field study*.

3.2. Results and Discussion

The following sections present and discuss the results of the interview study with industry practitioners. Section 3.2.1 describes the as-is state of CSE, Section 3.2.2 presents the as-is state of rationale management during CSE. Section 3.2.3 describes the practitioners' ideas for continuous rationale management. Appendix B provides an excerpt of anonymized interview statements by the practitioners.

3.2.1. As-is State of CSE in Industry

This section describes the results for the research question *How do practitioners apply CSE during software evolution?* (RQ1). The subsections describe 1) practitioners' definition of CSE, 2) most relevant CSE elements, 3) experiences, and 4) future plans for CSE. In the last subsection, we discuss the results. Figure 3.1 and Figure 3.2 show the results of the quantitative analysis of the CSE elements and categories that we identified within the practitioners' answers.

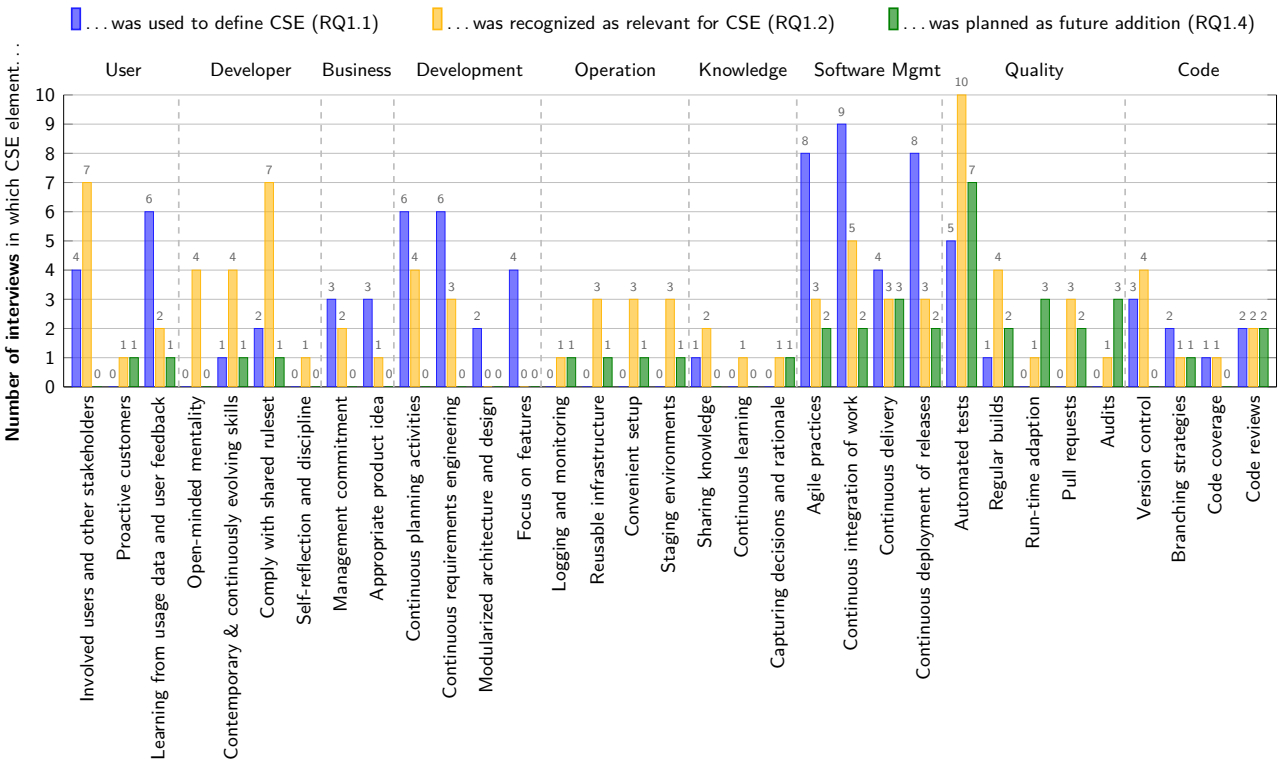

Figure 3.1.: Number of interviews in which the practitioners mentioned a CSE element. The left-hand bars (blue) indicate the number of interviews in which the respective CSE element was used for defining CSE (RQ1.1). The middle bars (yellow) indicate the number of interviews in which the respective CSE element was recognized as relevant for CSE (RQ1.2). The right-hand bars (green) indicate CSE elements intended as future additions (RQ1.4). The CSE categories group the CSE elements. Adapted from Johanssen et al. (2018), Johanssen et al. (2019c), and Johanssen (2019).

Practitioners' Definition of CSE

This section presents the results for the question *How do practitioners define CSE?* (RQ1.1). Since the term CSE was relatively new, we asked the practitioners to define CSE from their point of view. We coded their answers using the CSE elements. If a practitioner mentioned a CSE element not part of Table 3.2, we recorded it as a new CSE element, sometimes with a new CSE category. The quantitative results are shown in the left-hand bars (blue) in Figure 3.1. Table B.1 in Appendix B provides examples for CSE definitions by practitioners.

To define CSE from their point of view, the practitioners most frequently mention the CSE elements *continuous integration of work*, *agile practices*, *continuous deployment of releases*, *learning from usage data and user feedback*, *continuous planning activities*, and *continuous requirements engineering*.

We made the following observations: 1) Out of the 24 practitioners interviewed, 13 used the term CSE as part of their active vocabulary. About two-thirds of all interviewees defined their understanding of CSE. Notably, 75 % of the interviewees working in small or medium-sized enterprises both gave a definition and actively used the term CSE. For some practitioners, CSE is still ambiguous. 2) The practitioners emphasize the importance of short and ongoing iterations and that CSE makes changes instantly visible to users. As a result, user feedback can be elicited and used to match the software to the requirements. 3) According to the practitioners, CSE allows developers to focus on their tasks and creates a safe development environment. Tasks such as infrastructure management are automated and, as such, removed from the developers. However, developers get increased responsibility since they can deploy releases anytime. 4) The practitioners characterize CSE as the blending of different phases of software engineering, such as development, deployment, and operation. According to their perception, this makes long-living systems easier to maintain. 5) Six practitioners (developers and CSE specialists) make particular use of tool descriptions when defining CSE, i. e., they have a tool perspective. 6) The product determines whether CSE can be applied. For instance, the product cannot be developed using CSE if its deployment requires specific manual steps. For some products, continuous deployment is prohibited by safety regulations.

Practitioners' Relevant Elements of CSE

This section presents the results for the question *Which elements of CSE are perceived as most relevant by practitioners?* (RQ1.2). We asked the practitioners to list the three CSE elements that are most relevant. The middle bars (yellow) of Figure 3.1 show the quantitative results.

The practitioners perceive CSE elements from three categories as most relevant: *quality*, i. e., automated tests, *user*, i. e., involved users and other stakeholders, and *developer*, i. e., compliance with a shared ruleset. The practitioners mention more CSE elements: In particular, the developers consider elements from the *code* category, such as version control, as obligatory, pivotal, and indispensable to further steps in CSE. This strengthens the first stair in the *Stairway to Heaven* model by Olsson et al. (2012).

We made the following observations: 1) The practitioners perceive the users' commitment to actively participating in the development process as a relevant aspect of CSE. 2) The practitioners perceive an open-minded team mentality that complies with a shared set of rules as the basis of successful CSE teams. Management commitment is indispensable. 3) The practitioners perceive a high maturity level of automatization as essential for CSE. This is enabled by well-defined steps that form a non-linear process model.

Practitioners' Experience with CSE

This section presents the results for the question *What are practitioners' experiences with CSE?* (RQ1.3). We asked the practitioners about positive, neutral, and negative experiences with CSE elements. Figure 3.2 shows the results grouped by their respective categories. Note that not every practitioner provided an experience report. Table B.2 in Appendix B provides an excerpt of practitioners' experiences. When a practitioner responded to more than one CSE element, the answer is shown multiple times. We coded responses as either positive or negative in cases in which indicators by the practitioners were provided, for instance, if the practitioners used phrases such as *"we had problems with implementing <CSE element>"* or *"we could not work without <CSE element>"*.

The practitioners reported 19 positive, 56 neutral, and 17 negative experiences with CSE elements. Notably, more than 50 % of the positive experiences are stated by small or medium-sized enterprises, while forming roughly a quarter of the interviewee sample. Categories with many positive experiences, as in *code* and *software management*, are an indicator for CSE elements that can serve as an entry point to CSE since they may be easy to implement. Few positive mentions, as is the case with *knowledge*, *business*, and *user*, indicate immature CSE elements. Neutral responses indicate that the practitioners explore various CSE elements in the field. Many negative experiences, as with the *developer* category, are a sign of challenging CSE elements.

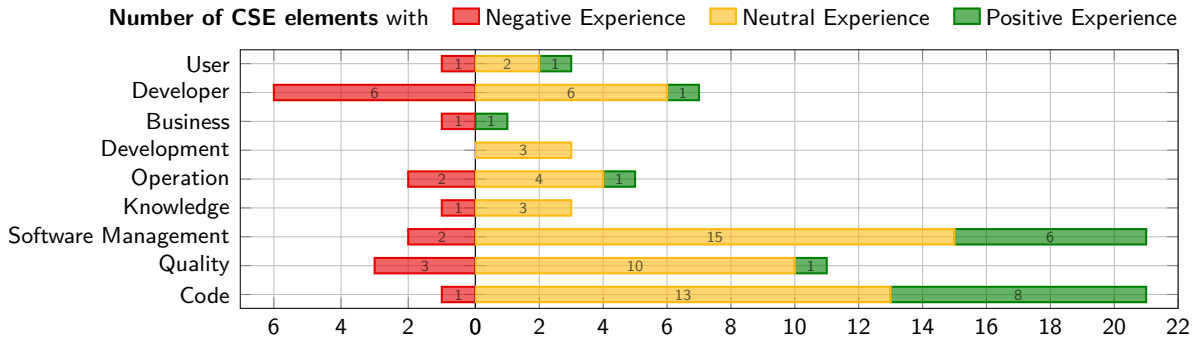

Figure 3.2.: Number of negative, neutral, and positive experiences that the practitioners reported per CSE category (RQ1.3). Adapted from Johanssen et al. (2018), Johanssen et al. (2019c), and Johanssen (2019).

We made the following observations: 1) The practitioners find it challenging to comply with shared rulesets and to evolve their skills continuously, i. e., to keep an open-minded mentality. 2) From the practitioners' experience, CSE does not solely build on the developers' skills but also on their ability to reflect on their work and sense of responsibility. 3) While the practitioners are willing to apply CSE, their company's current tools keep them from fully adopting CSE. Furthermore, requirements in regulated domains hinder the implementation of CSE. 4) The practitioners state that the successful implementation of CSE requires the ability to set up a new project without major cost or time penalties. 5) The practitioners attest that CSE elements related to software management, such as agile practices or continuous integration of work, are widely and successfully adopted in their projects. 6) The practitioners have not yet created processes that interact with users in a way similar to well-established practices such as continuous integration. This is mainly because users' responses to ongoing changes are difficult to record, trace, and assess. 7) The practitioners have had varying experiences with quality elements during CSE but still invest in improvements.

Practitioners' Future Plans for CSE

This section presents the results for the question *What are practitioners' future plans for CSE?* (RQ1.4). We asked practitioners which CSE elements they plan to add to discover future trends in CSE. The quantitative results are shown in the right-hand bars (green) in Figure 3.1. Table B.3 in Appendix B provides an excerpt of the plans.

Practitioners' plans are vague and mainly distributed across elements. Nineteen CSE elements received only one, two, or three mentions by the practitioners in the interviews. One CSE element stood out with seven mentions: *automated tests*. We found that most practitioners described plans that span multiple CSE categories.

We observed the following strategies in the practitioners' answers: 1) The practitioners aim to introduce more automatization, i. e., they aim for a fully automated loop. For example, seven practitioners mentioned automatization in the context of quality as one of their major plans for the short and long term. While *automated tests* are applied for some parts of the products, they should be made available for all. Another example is the plan to automatize the deployment of software components by applying container technology. 2) The practitioners aim to apply recently established CSE elements, such as continuous integration or delivery, to other project areas or similar products. 3) The practitioners improve their CSE process dependent on events that call for action. In general, they state that a process needs to be performed several times manually before they consider its automatization. They postpone decisions for further enhancements and additions of CSE elements to later.

Discussion: How do practitioners apply CSE during software evolution?

Based on the practitioners' answers for RQ1 (*How do practitioners apply CSE during software evolution?*), we added the *developer*, *business*, and *operation* categories, including new CSE elements, to the CSE elements and categories in Table 3.2. The *Eye of CSE* model consists of the final set of CSE elements and categories (Johanssen et al., 2018; Johanssen et al., 2019c; Johanssen, 2019). Figure 3.3 depicts this model. A well-established CSE process, as demanded by the CURES prototype (Section 2.4), incorporates all of the CSE elements.

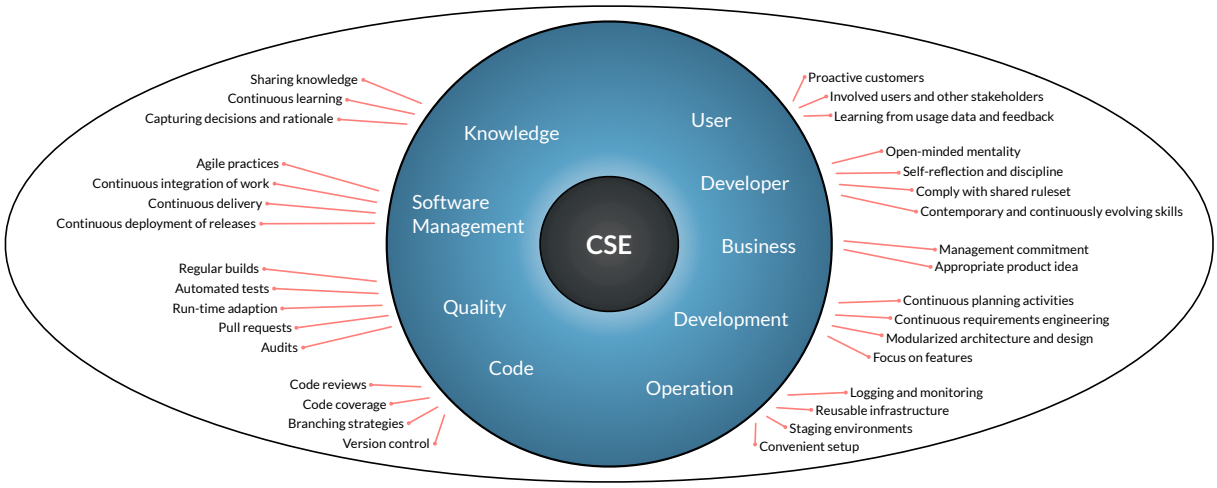

Figure 3.3.: Model *Eye of CSE* published by Johanssen et al. (2018; 2019c) and Johanssen (2019).

The contributions of the results regarding RQ1 for this thesis are threefold: First, next to the description of CSE in literature (Section 2.1), the results help the reader to understand the as-is state of CSE, i. e., the context of continuous rationale management. Second, the *knowledge* category received only a few answers regarding the practitioners' experiences (RQ1.3) and plans

(RQ1.4). The few answers indicate that practitioners do not perceive rationale management as a mature CSE workflow, motivating this thesis' design science project. Third, continuous rationale management should learn from advanced CSE elements well-known to the practitioners, such as agile practices, automated tests, and continuous integration. For example, this thesis presents how test coverage measuring, checking, and enforcing transfers to *decision coverage*. The thesis also introduces a *definition of done*, standard in agile practices, for knowledge documentation.

3.2.2. As-is State of Rationale Management during CSE in Industry

This section describes the anecdotal results for the research question *How do practitioners manage decision knowledge during CSE?* (RQ2). The subsections describe 1) decision types formalized and documentation approaches, 2) important decision types that remain implicit, reasons, and potential benefits if captured, 3) knowledge sharing approaches, and 4) change management approaches. In the last subsection, we discuss the results.

Decisions Captured during CSE

This section presents the results for the question *Which decisions are captured by practitioners during CSE, why, and how?* (RQ2.1). In three interviews, the practitioners stated not to capture decisions at all. In these cases, we started with the interview questions for RQ2.2.

Practitioners capture executive and existence decisions regarding the software architecture and feature implementation. They capture decisions in wiki and issue tracking systems in informal discussions and rely on techniques for establishing trace links and version control that come with these systems. Practitioners capture decisions as part of regular practices, such as code reviews and meetings, and during development. They mention improved decision making, accountability, knowledge sharing, and reuse support as benefits. However, the benefits and exploitation of the decision knowledge are unclear for some.

The following paragraphs detail the answers to the interview question of RQ2.1. Table B.4 – B.9 in Appendix B provide anonymized answers by practitioners.

Types of Captured Decisions Twelve practitioners capture *executive decisions* concerning the software development process, technologies, or applied tools. Such decisions impact the entire project or several projects. The executive decisions can be made by a steering committee. However, one practitioner highlights that CSE enables developers in making high-level decisions. As examples for executive decisions, practitioners mention to capture the decision to use a certain branching strategy or regarding setting up continuous delivery. One practitioner mentions to capture decisions on the definition of done of development tasks and on when a build can be deployed to the users. *Existence decisions* state that some elements will appear in the software (Kruchten, 2004). Thirteen practitioners capture existence decisions concerning requirements, architecture, implementation, test cases, and bug reports. Six practitioners capture decisions related to the elicitation, prioritization, and effort estimation of requirements for features. Eight practitioners capture architecture decisions and nine practitioners capture decisions regarding the implementation of features, e. g., on why a class was created. *Non-existence decisions or bans* state that some elements will not appear in the software (Kruchten, 2004). Five practitioners capture possible alternatives to solve a decision problem during their decision-making process. After evaluating the alternatives against the criteria, they pick one alternative as the decision. The alternatives they discard are documented non-existence decisions. One practitioner captures decisions regarding the prioritization of test cases and bug-fixing activities based on risk assessment. *Property decisions* concern the quality of the system and can be guidelines, design rules, or constraints (Kruchten, 2004). One practitioner mentions the issue on how to deal with data inconsistency after replacing the relational database with a NoSQL database.

Documentation Locations, Techniques, and Tools Practitioners use various documentation locations, techniques, and tools to capture decisions during CSE. Eight practitioners capture decisions in *external documents and tools* such as *Word* files, architecture design documents, or project reports. Only one practitioner uses a dedicated architecture management tool (*Enterprise Architect*). Thirteen practitioners capture decisions in a *wiki system*, such as Confluence. One practitioner uses a dedicated template page. Ten practitioners capture decisions in an *issue tracking and project management system*, such as *Jira* or *Redmine*, as part of the ticket description and its comments. One practitioner indicates decisions to be made using discovery tickets. Similarly, another practitioner marks tickets that contain an unsolved decision problem with a tag. One practitioner highlights that in their opinion *pull requests* are the best place to capture decisions to implement features. They create feature branches for a requirement and create a pull request directly afterward to discuss the feature implementation within the pull request. Another practitioner documents decisions as part of the *code* in comments and in *code reviews*. Code reviews can be done in pull requests, issue comments, or using dedicated code review systems, such as *gerrit*. Three practitioners capture decisions in *commit messages* and another three in informal communication systems, e. g., *chat tools* like *Slack*, and *emails*.

Linked Artifacts None of the practitioners uses a particular technique to establish links between captured decision knowledge and software artifacts. However, some practitioners use built-in techniques of the systems offering the documentation locations. For example, practitioners trace decisions captured in the issue tracking system to the respective tickets, such as user stories, and to artifacts linked to the tickets, e. g., software components and code. In addition, some practitioners tag separate documents or wiki pages: for instance, version numbers can enable traceability between decisions and software builds. Practitioners find it hard to keep the documentation of decisions and software artifacts in a consistent state. The practitioner using the architecture knowledge management tool criticizes that there are no links between the design models and the wiki system where they also capture decisions. They insert snapshots of the models into the wiki page, which they rate as highly unusable, especially when the models get changed. Another practitioner suggests capturing decisions as close to the code as possible.

Evolutionary History of Decisions Preservation of the evolutionary history is supported in those *documentation locations that offer version control*, e. g., in git and issue tracking systems. One practitioner describes a technique to *mark a rejected decision* and to preserve and link it with the new decision. However, the practitioner admits to having never used the technique.

Capturing Practices and Frequency The practitioners capture decisions during the *practices related to documentation locations*, e. g., in commit messages when committing changes, in the issue tracking system when working with tickets such as user stories, or in pull requests when working with feature branches. Six practitioners mainly capture decisions *on demand*, e. g., when planning bigger updates. One practitioner only captures decisions when discussion is needed, i. e., for controversial issues. Seven practitioners capture decisions as part of *regular practices* such as code reviews, meetings, and retrospectives. The practitioner reporting about the tag to mark an open decision states that the product owner regularly filters for such tagged tickets.

Benefits and Exploitation Five practitioners document decisions for improved *decision making* and better decisions due to clear criteria. Eight practitioners capture decisions and rationale for *accountability* reasons, e. g., as proof of why a particular feature has been developed and to avoid misunderstandings. One practitioner stresses that decisions help to recover former software versions. Three practitioners capture decisions for *knowledge sharing*. One practitioner highlights

that it is necessary to share where a new decision needs to be made, i. e., also to share issues. Two practitioners capture decisions to support *reuse* in the future and avoid duplicated work.

We asked the practitioners to rate the statement *The explicit capturing of decisions benefits our software development process* with one answer from a five-point Likert scale. In thirteen interviews, the practitioners rated this statement: one disagreed, three were neutral, and nine agreed (Figure 3.4). The practitioners, who disagreed and were neutral, emphasized that if the utilization of the captured knowledge was more clear, they would give a higher rating.

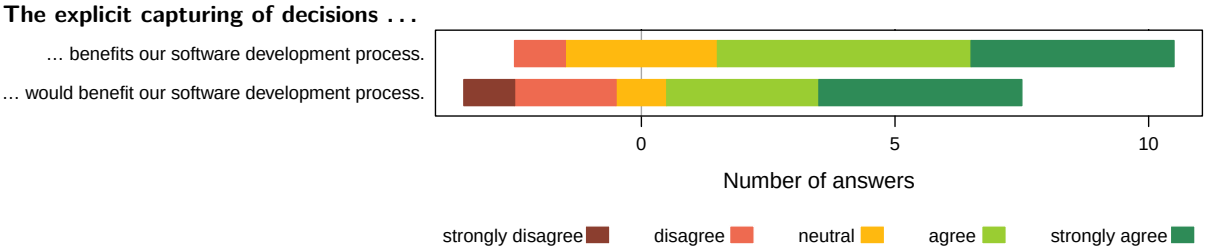

Figure 3.4.: The practitioners' attitude towards capturing decisions (above) and towards capturing decisions currently not captured (below).

Decisions not Captured during CSE

This section presents the results for the question *Which important decisions are not captured during CSE, why not, and would it be beneficial to capture these decisions?* (RQ2.2). Although some practitioners capture executive, existence, non-existence, and property decisions, others either a) do not capture the same type of decisions or b) provide other examples for decisions not captured. Decisions regarding the CSE process, prioritization, alternatives that are not selected (non-existence decisions), and the underlying rationale stay implicit. Practitioners do not capture decision knowledge because they fear intrusiveness and inconsistency, and they miss clear use cases for exploitation as well as techniques and tools. They see a potential benefit in supporting software evolution through captured non-existence decisions and decisions for code.

The following paragraphs detail the answers to the interview question of RQ2.2. Table B.11 – B.13 in Appendix B provide anonymized answers by practitioners.

Types of Decisions not Captured Seven practitioners provide examples for *executive decisions* regarding the CSE process not captured but important to capture. They state that the decisions on the continuous integration and deployment pipeline and the respective stages, e. g., the develop, test, and production stages, stay implicit. The practitioners capture the result of the prioritization of requirements based on cost estimation, test cases based on risk estimation, and bug-fixing activities. However, the rationale is not captured, especially if it comes to reprioritization. Two practitioners do not capture configuration decisions, e. g., which compiler or framework versions they use. Eleven practitioners do not capture certain *existence decisions*. Such decisions relate to features, software architecture, implementation, and tests. For example, one practitioner documents APIs between microservices using *Swagger* but does not capture decisions for the design of such interfaces and the underlying rationale. Three practitioners criticize not capturing the rationale behind decisions and why they discarded alternatives, i. e., *non-existence decisions or bans* stay implicit. Two practitioners have a common understanding of certain *property decisions*, e. g., about the coding style, but do not document property decisions. One practitioner provides the example of not having captured the decision on whether to use synchronous or asynchronous inter-service communication between microservices. Such kinds of decisions are made very quickly and then get reused by others but are neither discussed nor captured.

Reasons why Decisions are not Captured Two practitioners report that the decisions on how to deploy the software used to be captured in external documents but are no longer captured since the deployment is now automated. However, they still keep the former documents to externalize this knowledge. Four practitioners see a problem in *rapidly changing decisions* that lead to outdated decisions, i. e., to inconsistency between the captured decisions and their implementation. One practitioner associates the waterfall process with capturing decision knowledge. Five practitioners report that they *lack appropriate techniques or tools* to capture decisions and rationale. Three of them state that their *process is not mature enough* to involve decision management. Six practitioners do not capture decisions because they *lack techniques for easy retrieval and exploitation* of the captured decisions. Eight practitioners fear the *overhead and the intrusiveness* of capturing decisions and rationale. They could not spend the effort and do not have enough time. One practitioner mentions that the cost-benefit ratio would be too high if they captured more decisions than they currently do according to the 80/20 rule. However, the practitioners admit that the effort could be reduced by applying better capturing techniques.

Potential Benefits if Captured As for the captured decisions, practitioners see potential benefits in establishing accountability, improving decision making and knowledge sharing, as well as a support of reuse and maintenance activities. They also stress that capturing decisions and rationale would support continuous learning as part of the CSE process. Two practitioners see a potential benefit in retrieving decisions and rationale for code when evolving code. In their opinion, this could *ease the understanding of code*. Three practitioners state that it would be useful to know about alternatives for a decision and the rationale why they were not selected during software evolution. One practitioner mentions disaster recovery as an example of why knowledge sharing and capturing decisions were important.

We asked the practitioners to rate the statement *The explicit capturing of decisions would benefit our software development process* regarding the decisions they currently do not capture. Practitioners of eleven interviews rated this statement: three disagreed, one was neutral, and seven agreed (Figure 3.4). The practitioners, who disagreed or were neutral, feared that the extra effort would outweigh the benefits.

Sharing of Decision Knowledge during CSE

This section presents the results for the question *How do practitioners share decision knowledge during CSE?* (RQ2.3). Practitioners strongly rely on face-to-face communication, i. e., colleagues' knowledge, to recover implicit decisions. To share knowledge equally they apply techniques such as pair programming and inviting all team members as reviewers to pull requests. However, they also try to recover implicit decisions using reverse engineering.

The following paragraphs detail the answers to the interview question of RQ2.3. Table B.15 and B.16 in Appendix B provide anonymized answers by practitioners.

Alternative Knowledge Sources Six practitioners state that they try to do *reverse engineering* to recover knowledge from code and issue tracking systems. Ten practitioners mention that they *ask colleagues*, which has the disadvantage that both the inquiring person and the respondent need to interrupt their current activity. One practitioner reports that they have an *emergency mobile phone* that is carried by one knowledgeable project member for a period of time; afterwards, it is passed to the next project member. Two practitioners report that it can be hard to *scan through many emails and pull requests* to recover a decision. Thus, this decision was somehow documented but hard to retrieve. Another practitioner enforces that decisions are hard to retrieve in communication channels using the slogan "if it happens in [chat tool], it did not happen".

Avoidance of Knowledge Vaporization The practitioners try to avoid knowledge vaporization by sharing knowledge between project members. One practitioner states that in larger teams it is both necessary to share the knowledge within and across team boundaries. Knowledge management should address both the intra- and inter-team scope. Within teams, the practitioners try to share knowledge between all members as homogeneously as possible. They strongly rely on face-to-face communication. Further, one practitioner mentions that they always *invite all team members as reviewers for pull requests* and also do *pair programming* to distribute knowledge. One practitioner states that they encourage team members to always share their notes with others, e. g., by using a wiki system, instead of "writing diaries" in a notebook. Two practitioners mention having a dedicated process to onboard new project members. Generally, practitioners state that if a project member is about to leave the company, they would have a period during which this person tries to share and capture their knowledge.

Managing Changing Decisions during CSE

This section presents the results for the question *How do practitioners deal with changing decisions during CSE?* (RQ2.4). Overall, we received only a few responses from practitioners regarding the management of changing decisions during CSE. Table B.17 in Appendix B provides anonymized answers by practitioners. Practitioners apply cost estimation, risk estimation, and prioritization before integrating (changing) decisions. They depend on implicit knowledge and team communication to identify parts of the system affected by new or changed decisions. They rely on automated tests to detect side and ripple effects. None of the practitioners report a technique or tool to identify parts of the system affected by new or changed decisions. One practitioner reports about their *change management* process. For a change request, the project leader needs to decide whether the change will be integrated, and—if so—the developers *estimate the cost* for the change, *define a priority*, and break it down into tasks. Other practitioners emphasize the importance of *automated tests* to detect side and ripple effects as well as *risk management*. One practitioner of a consulting company criticizes that workflows often do not scale when the project and the respective team sizes increase. Change impact analysis would be critical for larger projects. However, it is not integrated since it was not necessary when the project was initially small.

Discussion: How do practitioners manage decision knowledge during CSE?

The results of RQ2 (*How do practitioners manage decision knowledge during CSE?*) provide anecdotal evidence on decisions captured and implicit decisions. Some practitioners mentioned capturing decisions that others do not capture and vice versa. The answers towards RQ2.1 and RQ2.2 provide examples of decisions practitioners consider important to be captured and for which purposes. Interestingly, many practitioners find executive decisions regarding the CSE process important to be captured. A reason might be that CSE involves continuous process improvement that comes with continuous decision making. The CSE process contains many defined workflows that developers need to decide on and need to have a common understanding (Figure 3.3).

The findings of the study confirm the rationale management problems described in Section 1.2. In 19 interviews, the practitioners mentioned that their decision-capturing method needs to be improved and that it is far from perfect. Only in one interview, a practitioner in the quality manager role states that they are very focused on capturing decisions. The degree of formalization of decision making and documentation in practice seems to be relatively low. The practitioners mainly capture decision knowledge informally in wiki and issue tracking systems. In the interviews, only five practitioners mention capturing alternatives for a decision, i. e., non-existence decisions. However, it is very important to capture non-existence decisions as they are not visible in the software artifacts and cannot be recovered using reverse engineering (Kruchten, 2004). Also, the

underlying rationale for solution decisions is not captured systematically. The practitioners use both a codification strategy, i. e., capturing the knowledge, and a personalization strategy (Babar et al., 2007; Hansen et al., 1999): They partly capture the decision knowledge (codification) but rely on experts' implicit and tacit knowledge and face-to-face communication as alternative knowledge sources (personalization). The practitioners argue not to capture decisions since the rapid change would make them outdated soon. Further, the practitioners state that using too many tools for capturing decisions can be frustrating. For reasons they list a) redundancy, i. e., they need to document knowledge in more than one tool, which means twice the effort and might result in inconsistent documentation, and b) a workflow interruption, i. e., they have to change their working context for documentation purposes, which means intrusiveness.

Although the practitioners confirm to document decision knowledge in typical documentation locations, e. g., the issue tracking system, the opportunities of CSE for improved decision knowledge management are not yet exhausted. The practitioners stress that utilizing the captured decision knowledge is not clear to them and that it is not appropriately exploited. They also highlight having difficulties finding and retrieving the decisions—especially if captured in informal communication channels such as *Slack*.

3.2.3. Practitioners' Assessment of Ideas for Continuous Rationale Management

This section describes the results for the research question *How can rationale management in CSE be improved according to practitioners?* (RQ3). We wanted to understand the practitioners' thoughts on the CURES prototype (Section 2.4). The subsections describe 1) features beneficial to practitioners and 2) obstacles they see. In the last subsection, we discuss the results.

Practitioners' Assessment of Features for Continuous Rationale Management

This section presents the results for the question *Which tool features for continuous rationale management do practitioners perceive as beneficial and why?* (RQ3.1). We asked the practitioners about important features or other additions of a CSE infrastructure extension to improve the management of decision and usage knowledge.

From the practitioners' responses, we identified eight functional features for tool-based rationale-management support during CSE: *decision-making support*, *decision knowledge documentation*, *change execution*, *knowledge presentation*, *filtering and searching*, *metrics calculation and reporting in dashboard*, *change impact analysis*, and *navigation*. Further, we identified the following non-functional features: *integration, interoperability, and traceability* as well as *automation*.

The following paragraphs describe the rationale management-related features beneficial to practitioners and explain why they are important. Table B.18 in Appendix B provides an excerpt of the anonymized answers by practitioners.

Decision-Making Support The practitioners request a *decision-making support*. First, the practitioners appreciate the possibility to *discuss decisions collaboratively* and to *comment on the decisions made* since "several stakeholders are usually involved in the decision-making process". One practitioner states that "it should be possible that you can also discuss the decisions again or ask: OK, there is a decision, but I have a completely different opinion about it". Second, two practitioners request a *voting possibility* for decision making, e. g., to record that "four people were in favor and three against" a solution option. Third, the practitioners acknowledge the idea to *support the developers in making decisions based on user feedback and usage data*, such as whether to continue improving functionality only used by a minority of users. One practitioner states that incorporating usage knowledge "would simplify the argumentation in many places because I can then discuss with other stakeholders at the beginning what is most important in terms of user feedback and customer satisfaction".

Decision Knowledge Documentation The practitioners request functionality for *decision knowledge documentation*. It relates to the decision-making support feature but focuses on what, where, and how to capture decision knowledge. First, the practitioners elaborate on what to capture: Besides the documentation of the solution decision, this feature should enable to *capture various types of decision knowledge elements*. One practitioner emphasizes that the documentation of alternatives for a decision should be supported. According to this practitioner, "alternatives do not have to be completely worked out, but you at least have to say: we could have done it that way. You do not have to write much code for that, but if you already have ideas about how to do something alternatively, then chances are that someone else will come up with that idea at some point. And then it would be good to say why it wasn't done that way". Second, the practitioners elaborated on where to capture the decision knowledge. They suggest capturing decision knowledge in *documentation locations typical for CSE, such as ticket comments, commit messages, and pull requests*. One practitioner states that "the most important thing is that it is close to the developer" and provides the example "if it's a Word document in a Sharepoint, that's so far away from the developers that when in doubt, nobody looks in there. If it's in the wiki, it's slightly closer to the developers. The supreme discipline would be integrating this information into a pull request". That means that documenting decisions within the development context, i. e., close to developers, leads to easy retrievability and is better than collecting the information in separate documents. Third, the practitioners discussed how to enable the decision knowledge documentation. According to practitioners, the decision knowledge documentation feature should support the *capturing of informal discussion* between the developers and other stakeholders. One practitioner is willing to apply a *dedicated language or syntax to capture decisions knowledge* when committing code—even if that means an additional workload at that stage. One practitioner emphasizes the importance of *storing meta-information for a documented decision, such as its discussants*: "Simply a decision is probably not yet sufficient as atomic information. It must also be recorded who was involved in the decision-making process."

Change Execution Next to the initial capturing of decision knowledge, the practitioners request a *change execution* feature to *change decision knowledge and related software artifacts*. One practitioner states that "the knowledge repository is suitable for recording why certain business processes are changed". Another practitioner requests from the tool support that "it automates everything that can be automated. I *can always edit it afterward*". For this practitioner, change execution is a must-be requirement taken for granted (Kano, 1984; Sauerwein et al., 1996).

Knowledge Presentation The practitioners request *knowledge presentation* functionality for the retrieval and (re)use, i. e., exploitation of the knowledge documentation. *Decision knowledge should be presented in context to other software artifacts*. One practitioner states that "the dependencies between the features should be shown, and one should be able to trace decision paths. It is crucial that you see the points that you want/must see at that moment. If everything is documented, it quickly becomes a lot. It is also vital to present this data in a concentrated way". Another practitioner states that it is "essential to see all possible alternatives".

Filtering and Searching Related to the knowledge presentation feature, the practitioners request functionality for *filtering and searching*. Two practitioners state that a powerful search would be the essential feature for them "because if you document something and then cannot find it, then it also brings nothing". One practitioner discusses a solution idea, i. e., the feature's design, as "I do not know how one would imagine that currently. Somehow it has to be searchable by text. That's what I would find most difficult but also most important".

Metrics Calculation and Reporting in Dashboard The practitioners acknowledge the idea of having the dashboard component as part of the CURES prototype, i. e., of having a feature for *metrics calculation and reporting in dashboard*. One practitioner states: "The most important feature for me would be the different metrics I get from monitoring usage and other technical metrics." The dashboard and the metrics could be used for *quality assessment and reporting*. For example, one practitioner suggests that "the dashboard can directly generate suggestions and warnings". Another practitioner suggests showing *run-time information* in the dashboard, e. g., "for a project team to know which feature is productive at all". The practitioner also emphasizes that developers should get "quick access to log data from production. This is mostly missing".

Change Impact Analysis Another feature beneficial to practitioners is *change impact analysis*. One practitioner describes it as "when a piece of information (like a decision) is recorded, you also know what is meant by it and that you see the impact, i. e., on which other artifacts this decision still has an impact".

Navigation The practitioners request a *navigation* feature for ubiquitous linking. One practitioner requests a "link to the code so that you also get from the code to the decisions". The practitioner discusses how to achieve the navigation between decisions and code: "You can go the way we are using now. If I want to know what has changed in the code, IntelliJ or Eclipse *annotates the class*, i. e., shows to the left of each line when it last changed by whom and in which commit. We also include the Jira ticket identifier in every commit message. This gives me a reasonably quick understanding of the context in which this class was touched." Another practitioner states: "I do not want to link it individually, but it should be obtained from the data I have across all my systems, whether in Jira or git, so that it can be linked automatically."

Integration, Interoperability, and Traceability The integration and interoperability with the tools developers usually use, and the automatic creation of trace links are important for practitioners. For instance, one practitioner states "you know how it is: the better the tool integration, the easier it is for you." Another practitioner states that "integration with the tools people usually use is very important. As soon as you have to document twice, nobody does it from experience." One practitioner requests that "there must be interfaces. Developers like to work close to the hardware. Reading and writing with markdown and YAML should be possible."

Automation Another essential feature to the practitioners is that tool support for continuous rationale management should be highly automated. One practitioner emphasizes that "a high degree of automation would be a prerequisite". As soon as one aspect that could be automated requires effort, such a system is difficult to establish; one practitioner justifies the need for automation because—if there is just one aspect that needs to be done manually—the data will turn inconsistent over time.

Practitioners' Assessment of Obstacles to Continuous Rationale Management

This section presents the results for the question *What obstacles do the practitioners perceive regarding the CURES prototype toward continuous rationale management?* (RQ3.2). We asked the practitioners for impediments to implementing the CURES prototype in their company. Table B.19 in Appendix B provides an excerpt of the practitioners' answers. From the practitioners' responses, we identified the following obstacles that tool support for continuous rationale management needs to treat: *overhead and intrusiveness of manual documentation*, *lack of techniques for easy retrieval and exploitation*, *inconsistency*, and *high amount of distributed documentation*. We found similar obstacles to the obstacles that already hinder the practitioners

from performing rationale management during CSE (RQ2.2). Besides, the practitioners mentioned that the *high amount of distributed documentation* is an obstacle to the implementation of the *knowledge presentation* and *filtering and searching* features. One practitioner states that "over time, the database will become huge because many features will be developed. I think it is not so easy to implement a meaningful search." Another practitioner states that "the knowledge repository already exists in parts in the form of ticketing systems, e. g., Redmine already does this. The problem, however, is that you always have to search in two places."

Discussion: How can rationale management in CSE be improved according to practitioners?

The capturing and exploitation of decision knowledge need to be better integrated into the daily practices of developers. The contribution of RQ3 (*How can rationale management in CSE be improved according to practitioners?*) is twofold: First, it contributes functional and non-functional features of support for continuous rationale management beneficial for the practitioners from the industry: Decision-making support (support for collaborative discussion, voting possibility, based on user feedback and usage data), decision knowledge documentation in documentation locations typical for CSE, such as ticket comments, commit messages, and pull requests, change execution, knowledge presentation, filtering and searching (support for recovering decision documentation from various sources), metrics calculation and reporting in a dashboard, change impact analysis, navigation through traceability, integration, interoperability, and automation. Second, it contributes obstacles to continuous rationale management that validate the three rationale management problems of intrusiveness and effort, high amount of distributed knowledge, and low documentation quality (Section 1.2). The features and obstacles constitute the basis of this thesis's treatment design consisting of the ConRat life cycle model extension and the ConDec plug-ins described in Part III.

3.3. Related Work

This section discusses related empirical studies investigating the as-is state of decision knowledge management in the industry and the improvement ideas suggested by practitioners. It discusses related work regarding the research questions RQ2 and RQ3, but also for non-CSE environments (meaning that the practitioners of the related studies might not have performed relevant CSE practices, such as automated testing and continuous integration). We included interview studies and written surveys with practitioners, i. e., primary sample studies, which—similar to our study—aim to maximize the generalizability to a population. Table 3.3 provides an overview of these studies and the rationale-management aspects that they investigate.

Section B.2 describes and compares the individual studies with ours. We make three conclusions from the comparison: 1) The results of the related studies enrich the results of our research. For example, we did not investigate influence factors during decision making as done by Tang et al. (2006) and Weinreich et al. (2015). 2) Our study confirms the results of the related studies, e. g., many documentation locations for decision knowledge, the problems, and (potential) benefits of rationale management. 3) Our study contributes new insights regarding rationale management during CSE. Some aspects of rationale management were not addressed in the related studies, in particular, *artifacts linked to the documented decision knowledge*, *techniques to preserve and use the evolutionary history of decisions*, *techniques for decision knowledge sharing*, and *techniques to identify change impacts*. Table 3.4 summarizes the findings from our study and the related studies. Our study contributes new findings regarding the aspects of rationale management investigated by other studies, highlighted in *italics* in Table 3.4. For example, we identified *pull requests* as a documentation location for decision knowledge.

Table 3.3.: Rationale management aspects investigated in the related studies with practitioners.

	This Thesis	**Tang et al., 2006**	**Weinreich et al., 2015**	**Capilla et al., 2016**	**Schubanz, 2021**
Data Collection Method	Interview	Written Survey	Interview	Interview	Written Survey
#Participants	24 from 17 companies	81	25 from 22 companies	6	102
Aspects regarding Decision Making					
Influence Factors (Forces, Drivers)	✗	✓	✓	✗	✗
Aspects regarding Decision Knowledge Documentation					
Types of Captured Decisions	✓(RQ2.1 a)	✓	✓	(✓)	✓
Documentation Locations, Techniques, Tools	✓(RQ2.1 b)	✓	✓	✓	✓
Linked Artifacts	✓(RQ2.1 c)	✗	✗	✗	✗
Evolutionary History of Decisions	✓(RQ2.1 d)	✗	(✓)	✗	✗
Capturing Practices and Frequency	✓(RQ2.1 e)	✗	✓	✗	✓
Benefits and Exploitation	✓(RQ2.1 f)	✓	✓	✓	✓
Aspects regarding Implicit Decision Knowledge					
Types of Decisions not Captured	✓(RQ2.2 a)	✓	✓	✓	(✓)
Reasons why Decisions are not Captured	✓(RQ2.2 b)	✓	✓	✓	✓
Potential Benefits if Captured	✓(RQ2.2 c)	✗	✗	✓	✗
Aspects regarding Decision Knowledge Sharing					
Alternative Knowledge Sources	✓(RQ2.3 a)	✗	✗	✗	✗
Avoidance of Knowledge Vaporization	✓(RQ2.3 b)	✗	✗	✗	✗
Aspects regarding Changing Decisions					
Techniques to Identify Change Impacts	✓(RQ2.4)	✗	✗	✗	✗
Aspects regarding Improvement of Decision Knowledge Management					
Features for Tool Support	✓(RQ3.1)	(✓)	✓	✓	✗
Guidelines and Social Aspects	✗	✓	✓	✓	✗

Table 3.4.: Findings of our and related studies with practitioners. Findings only mentioned in our study are highlighted in *italics*.

Aspect	**Overall Findings from our Study and Related Studies**
Aspects regarding Decision Knowledge Documentation	
Types of Captured Decisions	Existence decisions (e. g., regarding architecture, design, feature implementation and refinement, user experience), non-existence decisions (by evaluating multiple alternatives against criteria, selecting one as the decision, and preserving the discarded alternatives), executive decisions (e. g., regarding *CSE process such as for branching strategy, deployment, and definition of done*, tools, feature prioritization, team, to-do-items), property decisions (e. g., regarding quality such as *avoidance of data inconsistency when using a server cluster*), solution-oriented vs. driver-oriented, classified according to level, granularity, scope, and impact

Continued on next page

Aspect	Overall Findings from our Study and Related Studies
Documentation Locations and Tools	Wiki, issue tracking system, text document (Word file), *pull/merge request*, commit message, code, meeting minute (protocol), diagram, presentation, email, agile backlog, (UML) modeling tool such as e. g., Enterprise Architect, *chat message*, architecture design document, project report
Techniques	Informal capturing in natural language text vs. formal capture using rationale model (templates), *marking the status of a decision problem in tickets using a tag (solved/unsolved)*, *usage of discovery ticket type for discussing and solving decision problems*
Linked Artifacts	*Decisions captured in the issue tracking system can be traced to the respective tickets, such as user stories, and also to artifacts that are linked to these tickets, e. g., software components and code*
Evolutionary History of Decisions	*Usage of built-in version control of documentation locations such as git and issue tracking systems*, *marking rejected decisions*
Capturing Practices and Frequency	*During the practices that are related to certain documentation locations, e. g., in commit messages when committing changes, working with tickets such as user stories, or in pull requests when working with feature branches*, during code reviews, meetings, retrospectives, sprint planning, backlog refinement, and on demand
Benefits and Exploitation	Improving/systematizing/supporting decision making, accountability (arguing with customers during development, preparation for later audits), resolving production problems (disaster recovery), support maintenance and modification tasks, impact analysis, knowledge sharing (training of new employees, preventing knowledge vaporization, continuous learning, ease the understanding of code), reuse, compliance with legal regulations
Aspects regarding Implicit Decision Knowledge	
Types of Decisions not Captured	Some practitioners capture executive, existence, non-existence, and property decisions during CSE, while others either a) do not capture the same type of decisions or b) provide other concrete examples for decisions that they do not capture: *existence decisions (e. g., regarding design of microservices and the APIs between them)*, non-existence and ban decisions, executive decisions (e. g., regarding *(re-)prioritization of feature development, CSE development process, decisions which version to use of a framework*, tools), property decisions (e. g., regarding *whether to generally use synchronous or asynchronous inter-service communication between microservices*)
Reasons why Decisions are not Captured (Barriers, Obstacles, Problems)	Lack of time or budget (documentation of design decisions is often too time- and cost-intensive), unawareness of the need and usefulness of documenting design rationale, lack techniques for easy retrieval and exploitation, unclear cost/benefit ratio when documenting decisions, a lack of a formal review process, rapidly changing decisions (dynamic nature of technology and solutions makes it useless to document design rationale, outdated documentation, and redundancies that lead to inconsistencies) difficulty in finding a documented decision or determining whether a specific decision is documented or not, difficulty in deciding what to document and how best to document the knowledge, lack of motivation or incentive, lack of adequate techniques or tools, uncertainty of what to capture, overhead and intrusiveness (disrupting the design flow, effort in capturing), lack of stakeholder understanding, *process is not mature enough*, *high amount is hard to handle*
Potential Benefits if Captured	Same benefits and exploitation scenarios as for captured decisions, e. g., explanation of why the software was designed in a certain way (accountability), improved decision making and knowledge sharing, support of reuse and maintenance activities, alternatives for a decision and the rationale why they were not selected would be useful during software evolution, ease the understanding of code
Aspects regarding Decision Knowledge Sharing	
Alternative Knowledge Sources	*Software system (recovering of existence decisions using reverse engineering)*, *colleagues' knowledge (asking them during work, usage of an emergency mobile phone)*, *informal written communication (for example, reading emails and pull requests)*

Continued on next page

Aspect	Overall Findings from our Study and Related Studies
Avoidance of Knowledge Vaporization	*Sharing knowledge between project members (within and across team boundaries) through face-to-face communication*, *inviting all team members as reviewers for pull requests*, *pair programming*, *using shared platforms such as wiki systems*, *process to onboard new and offboard leaving project members*
Aspects regarding Changing Decisions	
Techniques to Identify Change Impacts	*Reliance on implicit knowledge and team communication to identify parts of the system affected by new or changed decisions*, *execution of automated tests to detect side and ripple effects after change execution*, *risk assessment*
Aspects regarding Improvement of Decision Knowledge Management	
Features for Tool Support	*Decision-making support (support for collaborative discussion, voting possibility, based on user feedback and usage data)*, *decision knowledge documentation (documentation locations typical for CSE, such as ticket comments, commit messages, and pull requests)*, *change execution*, *knowledge presentation*, filtering and searching (support for recovering decision documentation from various sources), *metrics calculation and reporting in dashboard*, *change impact analysis*, navigation through traceability, integration, interoperability, automation

3.4. Threats to Validity

This section discusses four validity criteria of primary empirical studies as defined by Easterbrook et al. (2008) and Runeson et al. (2012):

Construct validity focuses on whether the theoretical constructs are measured and interpreted correctly. The practitioners might have interpreted the interview questions differently from what we intended. To reveal misinterpretations, we allowed them to ask questions at any time and conducted two interviews with colleagues we discussed afterward. We used open-ended questions to elicit as much information as possible.

Internal validity concerns whether the results we draw really follow from the data, e. g., whether confounding factors influence the results. The practitioners might have provided answers that do not fully reflect their daily work since they knew the results would be published. We guaranteed the anonymity of interviewees and companies to address this. The interpretation of answers might be biased by the authors' *a priori* expectations, which we addressed by coding the transcriptions and discussing the codes.

External validity addresses the generalizability of the study results. We contacted companies we already knew, which might result in a selection bias. It is mitigated by the fact that two researchers from two universities had different industrial contacts that they contacted. Interviews are subjective since they rely on the practitioners' statements. We conducted 20 interviews with 24 practitioners to reduce subjectivity and acquire broader opinions. However, a major threat to the external validity is that most of the *evidence collected in this study is anecdotal*. For parts of the results, we reported the number of practitioners who gave a response. The sample size of 24 practitioners is too small to generalize. Still, similar answers by multiple practitioners can hint at standard practices, e. g., ten practitioners capture decisions in an issue tracking system. We also reported subjective opinions of individual practitioners that might not be generalizable to provide rich qualitative data, and the results should be interpreted from a constructivist stance.

Reliability validity concerns the study's dependency on specific researchers. After we conducted coding training and checked intercoder reliability, two researchers individually coded different transcripts. We addressed this threat by discussing questions during coding. In addition, the supervisors of this thesis supported the interview analysis.

3.5. Conclusion

This chapter reported findings from an interview study on how practitioners define and perform CSE and manage rationale during CSE. It reported ideas on how practitioners would improve the rationale management. The contributions to the remainder of the thesis are the following:

The interview study contributed the *Eye of CSE* model that represents a well-established CSE process. The model helps to understand the context of continuous rationale management and contributes to the description of CSE in literature (Section 2.1). Besides, the study identified advanced CSE elements well-known to the practitioners, such as agile practice, automated tests, and continuous integration. Continuous rationale management adopts concepts from these well-known CSE elements; for example, measuring, checking, and enforcing *decision coverage* similar to test coverage and a *definition of done for knowledge documentation*.

The interview study contributed insights into rationale management practices and problems, complementing related work. The results indicate that rationale management is not systematically integrated into CSE. The practitioners capture decision knowledge informally, for example, in natural language discussions in issue tracking systems. For them, capturing rationale has many positive effects, such as improved decision-making and change processes, accountability, knowledge sharing, and reuse. However, the practitioners lack systematic techniques and tools for rationale management. The reported challenges confirm and illustrate the rationale management problems in Section 1.2: 1) Documenting rationale is seen as an overhead and intrusive. 2) It is not clear how to access and exploit the decision knowledge documentation when needed during software evolution. 3) Rapid changing decisions lead to outdated documentation, i. e., inconsistency between the captured decisions and their implementation. Even if the decision knowledge is captured, e. g., in the issue tracking system, it is difficult to access in the context of requirements, code, and other software artifacts.

The interview study contributed functional and non-functional features for continuous rationale management that benefit practitioners and obstacles, confirming the rationale management problems. The treatment described in Part III of the thesis builds upon the features and treats the problems. It consists of the ConRat life cycle model extension and the ConDec plug-ins. A non-functional aspect frequently requested by the practitioners is that rationale management should be well integrated into CSE rather than treating it separately. An important decision for the treatment is the *integration* of ConRat and ConDec into the existing workflows and tools by the developers to minimize the intrusiveness. ConDec offers comprehensive support for collaborative *decision knowledge documentation* in various documentation locations close to artifacts, such as requirements, code, and feature branches. ConDec enables explicit, formalized documentation in the description and comments of tickets, commit messages, and code comments. It does not restrict the type of decisions. Developers can capture executive, existence, non-existence, and property decisions. The documentation builds on lightweight annotations that are automated with ConDec's automatic text classification. It supports *knowledge presentation* with interactive views, with functionalities for *filtering, searching, navigation*, and *change impact analysis*. ConDec also offers comprehensive support for *metrics calculation and reporting in a dashboard*. ConRat and ConDec support *decision making* by making decision knowledge explicit. In addition, ConDec supports decision making with the criteria matrix view, decision guidance recommendation system, rationale backlog, meeting agenda with decision knowledge, and nudging mechanisms to indicate where a decision needs to be made.

The study results are interesting for practitioners to compare their current practices and to reflect on the necessity for adopting ConRat and ConDec. The interview study is also interesting for researchers, e. g., when performing future interview studies to compare their results.

Chapter 4

State of the Art: Classification and Recommendation for Rationale Management

"We learn best from mistakes, but who said all these mistakes have to be our own ones?"

—Zimmermann, 2011

This chapter contributes to the knowledge goal 2 of the thesis: *Understand the current state of the art regarding rationale management support with classification or recommendation.* It presents a systematic mapping study contributing an overview of approaches to treat the rationale management problems described in Section 1.2. The study contributes to the problem investigation, and the overview of the approaches is a basis for the treatment design of ConRat and ConDec described in Part III of the thesis.

Section 4.1 describes the study design, including research questions and the publication search. Section 4.2 presents and discusses the results of the systematic mapping study. Section 4.3 discusses threats to validity. Section 4.4 concludes this chapter.

4.1. Study Design

Section 4.1.1 introduces the research questions. Section 4.1.2 describes the procedure of the literature study, including the publication search and the search results.

4.1.1. Research Questions

The knowledge goal 2 is refined into a research question with sub-questions (Table 4.1):

RQ1 What are the characteristics of (semi-)automatic classification and recommendation approaches to support rationale management?

The research question asks for the characteristics of rationale management support with classification and recommendation. A result of the interview study in Chapter 3 is the automation request; thus, the question asks for (semi-)automatic approaches. To create a standardized description of the approaches, the research question is refined into four questions:

RQ1.1 *Which (semi-)automatic classification and recommendation approaches for rationale management exist?* This question identifies the existing types of approaches and the respective publications. It also investigates the frequency of published approaches over time to see trends.

Table 4.1.: Research questions of the systematic mapping study.

	Research Question
RQ1	What are the characteristics of (semi-)automatic classification and recommendation approaches to support rationale management?
RQ1.1	Which (semi-)automatic classification and recommendation approaches for rationale management exist?
RQ1.2	How do the approaches support software practitioners?
RQ1.2a	Which rationale management problems are treated?
RQ1.2b	Which rationale management activities are supported?
RQ1.2c	Are the approaches supported with tools, and how do the tools integrate into the process?
RQ1.3	How do the approaches internally work?
RQ1.4	Which evaluation aspects are investigated?

RQ1.2 *How do the approaches support software practitioners?* We aim to understand how the approaches can support software practitioners in performing rationale management. This question investigates the user's perspective on the approaches. It investigates the problems treated by the approach described in Section 1.2 (*intrusiveness and effort*, *high amount of distributed documentation*, or *low documentation quality*), the supported activities (*decision making*, *documentation*, *exploitation*, or *quality assurance*) and the implementation in tools.

RQ1.3 *How do the approaches internally work?* We aim to understand the functioning of the approaches. Classification and recommendation approaches make predictions based on heuristics (also called features). These heuristics can be 1) based on human intuition about the problem, 2) based on data mining to identify patterns, or 3) *machine learning-based* (Robillard and Walker, 2014). This thesis refers to the 1) and 2) techniques as *rule-based*. We aim to investigate the heuristics used in classification and recommendation approaches for rationale management.

RQ1.4 *Which evaluation aspects are investigated?* Empirically evaluations can investigate various aspects, and literature uses different terms. The primary aspect an approach needs to fulfill is *feasibility*. On top, researchers evaluate more advanced aspects described in Section 1.3. With this question, we aim to understand which of the following aspects the researchers validate: 1) *user acceptance* by interviewing or surveying software practitioners, e. g., asking for *ease of use*, *usefulness*, or *satisfaction*, 2) *effectiveness* or *efficiency*, or 3) feasibility in another way.

4.1.2. Literature Study Procedure

Two methods for systematically studying literature exist: *systematic mapping study* and *systematic literature review*. Both methods are used in secondary studies to provide an overview of the primary studies in a research area. They help in preventing selection bias and incompleteness in the research. A systematic mapping study (also called *scoping study*) aims to provide a wider overview with broader research questions than a systematic literature review. A systematic mapping study is used to examine a research area. In contrast, the systematic literature review is a well-defined methodology to identify, analyze, and interpret all available evidence related to a research question. A systematic mapping study is used to identify evidence clusters to guide future systematic literature reviews and identify evidence deserts lacking primary studies (Kitchenham and Charters, 2007; Petersen et al., 2008).

Both methods require the researchers to document the search procedure so that readers can repeat the search and assess the rigor and completeness. Researchers can *search in online databases using a search string* or via a manual search, e. g., in conference proceedings. *Snowballing* is a manual search strategy that starts from specific publications. When performing *backward snowballing*, researchers scan the references of a publication. When performing *forward snowballing*, researchers identify new publications that cite a specific publication (Wohlin, 2016).

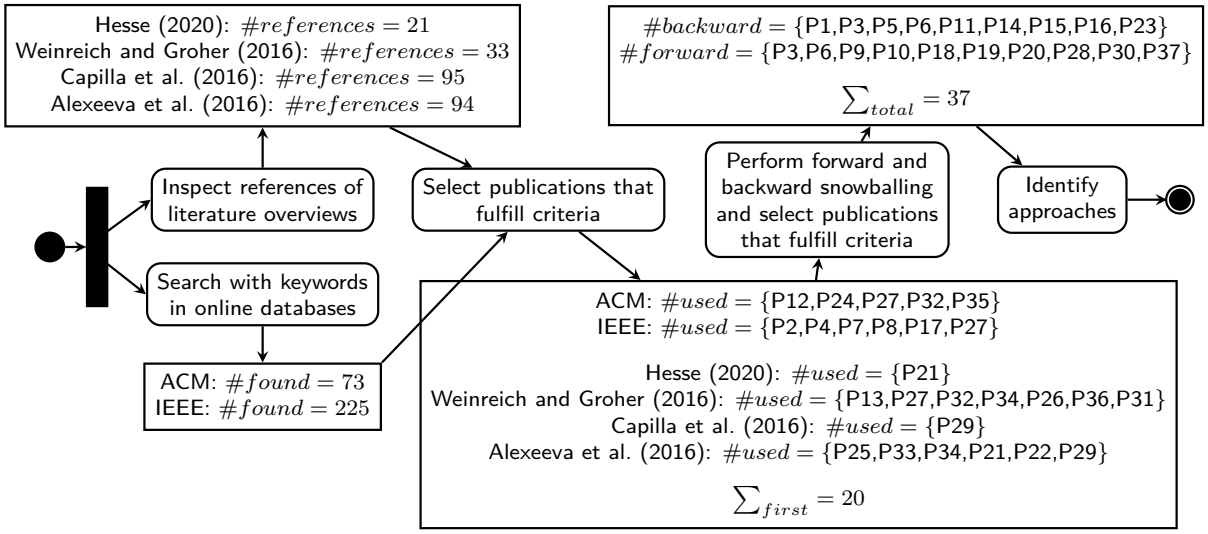

Figure 4.1.: Search procedure and results of the systematic mapping study on classification and recommendation approaches for rationale management (UML activity diagram).

The literature study in this thesis is a systematic mapping study because it aims to provide a broad overview of the existing approaches that support rationale management with classification or recommendation. The systematic mapping study follows the guidelines by Kitchenham and Charters (2007). Figure 4.1 visualizes the search procedure as a UML activity diagram including the search results as object nodes. We identified the relevant publications in two ways:

First, we performed backward snowballing on existing literature overviews regarding decision knowledge management. We *inspected the references of the literature overview* by Hesse (2020), who studied existing approaches and tools for decision knowledge documentation. We also inspected the references of the three most recent literature overviews collected as related work by Hesse (2020): Alexeeva et al. (2016), Capilla et al. (2016), and Weinreich and Groher (2016).

Second, we *performed a keyword search by querying online databases* using a search string. We searched two prominent scientific databases on software engineering research: ACM and IEEE Xplore. The search string consists of search terms reflecting the research question *Which (semi-)automatic classification and recommendation approaches for rationale management exist?* The final search string was as follows:

```
1 ("rationale" OR "decision") AND ("manage*" OR "document*" OR "knowledge")
2     AND ("classif*" OR "recommend*")
3     AND ("approach" OR "technique" OR "method" OR "support" OR "system")
4     AND ("software")
```

The first line identifies publications of rationale management and related terms such as decision knowledge, decision documentation, or rationale documentation. The second line narrows down the topic to classification or recommendation. The third line specifies that we are searching for approaches instead of purely empirical studies. The fourth line narrows down the topic to software. The search string was applied in the publication title and abstract to find only publications focused on the topic. Appendix C contains the resulting database queries.

From all publications found by the keyword search or snowballing, we *selected the publications that fulfill the inclusion and exclusion criteria* listed in Table 4.2. We read the title and abstract to decide if a publication fulfills the criteria. In ambiguous cases, we also read the publication content. The exclusion criteria E_1 to E_3 are commonly used in systematic mapping studies and literature reviews (Kitchenham and Charters, 2007). Capilla et al. (2016) divide tools for architectural knowledge management into three generations. We decided to exclude first- and second-generation publications published before 2010. The tools of the first and second generations focus on must-be requirements for knowledge management, such as capturing, representing, and sharing knowledge. In contrast, this study systematizes (semi-)automated classification and recommendation support that goes beyond. The criterion E_3 excludes publications that did not go through a peer-review process, such as dissertations. After applying E_1 to E_3, we checked the fulfillment of the inclusion criterion, which reflects the research question *Which (semi-)automatic classification and recommendation approaches for rationale management exist?* The criterion E_4 excludes approaches that cannot support developers during software development, such as medical decision support approaches. The criterion E_5 excludes approaches to improve management decisions in terms of cost, time, and recommendations of human resources. While management decisions are important, this study focuses on decisions related to system knowledge (e. g., requirements and code) to narrow down the topic. The criterion E_6 excludes approaches that solely work with non-decision knowledge artifacts, such as approaches on automatically identifying requirements in text. It excludes text mining approaches with results different from decisions and other typical rationale elements. For example, Pan et al. (2021) identify problems, information, technical discussions, and task progress in chat messages but no decisions. Zhu et al. (2015) guide developers in deciding where to include logging statements in source code. The approach is excluded because the input and output are code snippets without rationale. The criterion E_6 also excludes approaches to recommend experts, such as by Bhat et al. (2018). We are interested in (semi-)automatic approaches that can help to treat intrusiveness and effort. The criterion E_7 excludes approaches that developers perform manually, i. e., without (prototypical) implementation, such as the approach for analyzing design meetings by Pedraza-García et al. (2015). The criterion E_8 excludes approaches that are early ideas without empirical validation.

Table 4.2.: Exclusion criteria E_n and inclusion criterion I to identify publications on classification and recommendation approaches for rationale management.

Criterion	Description
E_1	publication is published before 2010
E_2	publication is not written in English
E_3	publication is not peer-reviewed
I	publication describes a (semi-)automatic approach to support rationale management with classification or recommendation
E_4	publication describes an approach for an application domain outside software development
E_5	publication describes an approach solely focusing on executive decisions
E_6	publication describes an approach solely for or resulting in non-decision knowledge artifacts
E_7	publication describes a manual approach
E_8	publication describes an approach without empirical validation

The backward snowballing on existing literature overviews resulted in 12 publications (*#used* in Figure 4.1). Three publications (P21, P34, P29) are referenced in different literature overviews. The keyword search resulted in 10 publications (March 1, 2023). One publication (P27) is part

of both databases. We identified 21 publications in the first search iteration ($\sum_{first}$). Two publications (P27, P32) are the results of the literature overviews and keyword search.

We then performed one iteration of *forward and backward snowballing on the results* to ensure the completeness of the search, again applying the inclusion and exclusion criteria. We used Google Scholar to perform forward snowballing. We only included new publications not found so far, i. e., removed duplicates. The forward and backward snowballing resulted in 17 new publications (*#backward* and *#forward* in Figure 4.1). Two publications (P3, P6) are results of both forward and backward snowballing. In total, we found 37 relevant publications ($\sum_{total}$).

In the last step, we *identified approaches* by considering the supported rationale management activities and the internal functioning of the approaches reported in the publications.

4.2. Results and Discussion

The following sections present and discuss the results of the systematic mapping study. Section 4.2.1 presents the identified approaches and the publications. Section 4.2.2 describes the users' perspective on the approaches. Section 4.2.3 presents the machine-learning techniques and rules applied to understand the functioning. Section 4.2.4 describes the approach evaluation.

4.2.1. Overview of Approaches and Publications

This section presents the results for the question *Which (semi-)automatic classification and recommendation approaches for rationale management exist?* (RQ1.1). We grouped the 37 publications into four approaches supporting rationale management: Automatic text classification, automatic tracing, decision guidance, and consistency support. Four publications (P4, P29–P31) present two approaches, and the other publications present one approach. The following subsections introduce the approaches and list the publications. Figure 4.2 shows the timeline of the number of approaches (above) and the extrapolated proportion (below) published per year. The four publications with two approaches are counted twice. The timeline shows a shift in research interests over time: In 2011 and 2012, automatic linking was researched, but

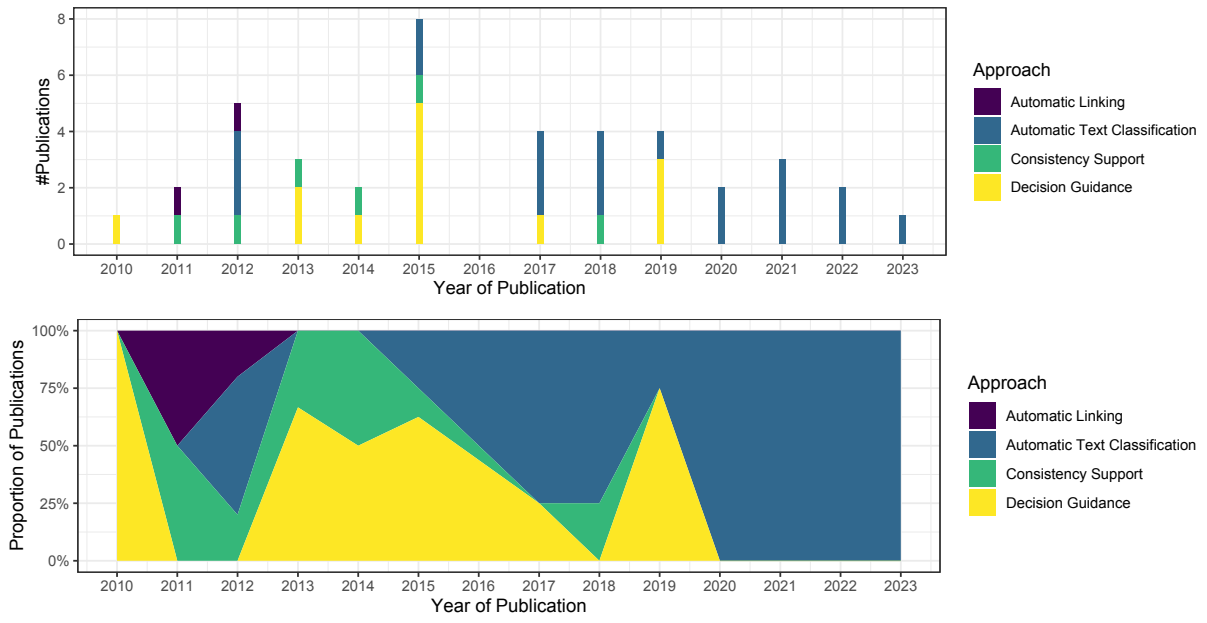

Figure 4.2.: Number (above) and proportion (below) of approaches published per year.

not afterward. From 2020 onward, we only found publications on automatic text classification. Automatic text classification is often machine learning-based (Section 4.2.3). In recent years, machine learning-based approaches have gained interest due to improved computing power, which might be the reason for this shift.

Automatic Text Classification

Approaches for *automatic text classification* aim to identify decision knowledge in informal discussions to generate *extractive summaries* of the decisions made. The approaches involve text mining from various sources such as chat messages, issue tracking systems, and mailing lists. Table 4.3 lists the 20 relevant primary publications on automatic text classification.

Table 4.3.: Primary publications on automatic text classification.

ID	Primary Publication on Automatic Text Classification
P1	R. Alkadhi, T. Laţa, E. Guzman, and B. Bruegge (2017b). "Rationale in Development Chat Messages: An Exploratory Study". In: *14th International Conference on Mining Software Repositories*. MSR '17. Buenos Aires, Argentina: IEEE Press, pp. 436–446. DOI: 10.1109/msr.2017.43
P2	R. Alkadhi, M. Nonnenmacher, E. Guzman, and B. Bruegge (2018). "How do developers discuss rationale?" In: *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Campobasso, Italy: IEEE, pp. 357–369. DOI: 10.1109/saner.2018.8330223
P3	M. Bhat, K. Shumaiev, A. Biesdorf, U. Hohenstein, and F. Matthes (2017b). "Automatic Extraction of Design Decisions from Issue Management Systems: A Machine Learning Based Approach". In: *11th European Conference on Software Architecture (ECSA'17)*. Ed. by A. Lopes and R. de Lemos. Cham, Switzerland: Springer, pp. 138–154. DOI: 10.1007/978-3-319-65831-5_10
P4	M. Bhat, C. Tinnes, K. Shumaiev, A. Biesdorf, U. Hohenstein, and F. Matthes (2019). "ADeX: A Tool for Automatic Curation of Design Decision Knowledge for Architectural Decision Recommendations". In: *International Conference on Software Architecture Companion (ICSA-C)*. Hamburg, Germany: IEEE, pp. 158–161. DOI: 10.1109/ICSA-C.2019.00035
P5	M. Dhaouadi, B. J. Oakes, and M. Famelis (2022). "End-to-End Rationale Reconstruction". In: *37th IEEE/ACM International Conference on Automated Software Engineering*. Rochester, MI, USA: ACM, pp. 1–5. DOI: 10.1145/3551349.3559547
P6	L. Fu, P. Liang, X. Li, and C. Yang (2021). "A Machine Learning Based Ensemble Method for Automatic Multiclass Classification of Decisions". In: *Evaluation and Assessment in Software Engineering*. Trondheim, Norway: ACM, pp. 40–49. DOI: 10.1145/3463274.3463325
P7	A. Josephs, F. Gilson, and M. Galster (2022). "Towards Automatic Classification of Design Decisions from Developer Conversations". In: *19th International Conference on Software Architecture Companion (ICSA-C)*. Honolulu, HI, USA: IEEE, pp. 10–14. DOI: 10.1109/ICSA-C54293.2022.00009
P8	Z. Kurtanović and W. Maalej (2017). "Mining User Rationale from Software Reviews". In: *25th IEEE International Requirements Engineering Conference (RE)*. ed. by A. Moeira and J. Araújo. Lisbon, Portugal: IEEE, pp. 53–62. DOI: 10.1109/RE.2017.86
P9	Z. Kurtanović and W. Maalej (2018). "On user rationale in software engineering". In: 23.3, pp. 357–379. DOI: 10.1007/s00766-018-0293-2
P10	M. Lester, M. Guerrero, and J. Burge (2020). "Using evolutionary algorithms to select text features for mining design rationale". In: *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 34.2, pp. 132–146. DOI: 10.1017/S0890060420000037
P11	Y. Liang, Y. Liu, C. K. Kwong, and W. B. Lee (2012). "Learning the "Whys": Discovering design rationale using text mining — An algorithm perspective". In: *Computer-Aided Design* 44.10, pp. 916–930. DOI: 10.1016/j.cad.2011.08.002
P12	X. Li, P. Liang, and Z. Li (2020). "Automatic Identification of Decisions from the Hibernate Developer Mailing List". In: *Evaluation and Assessment in Software Engineering*. December. Trondheim, Norway: ACM, pp. 51–60. DOI: 10.1145/3383219.3383225
P13	C. López, V. Codocedo, H. Astudillo, and L. M. Cysneiros (2012). "Bridging the gap between software architecture rationale formalisms and actual architecture documents: An ontology-driven approach". In: *Science of Computer Programming* 77.1, pp. 66–80. DOI: doi.org/10.1016/j.scico.2010.06.009

Continued on next page

ID	Primary Publication on Automatic Text Classification
P14	B. Rogers, J. Gung, Y. Qiao, and J. E. Burge (2012). "Exploring techniques for rationale extraction from existing documents". In: *2012 34th International Conference on Software Engineering (ICSE)*. Zurich, Switzerland: IEEE, pp. 1313–1316. DOI: 10.1109/ICSE.2012.6227091
P15	B. Rogers, Y. Qiao, J. Gung, T. Mathur, and J. E. Burge (2015). "Using Text Mining Techniques to Extract Rationale from Existing Documentation". In: *Design Computing and Cognition '14*. Springer, pp. 457–474. DOI: 10.1007/978-3-319-14956-1_26
P16	B. Rogers, C. Justice, T. Mathur, and J. E. Burge (2017). "Generalizability of Document Features for Identifying Rationale". In: *Design Computing and Cognition '16*. Cham: Springer International Publishing, pp. 633–651. DOI: 10.1007/978-3-319-44989-0_34
P17	A. Shahbazian, Y. Kyu Lee, D. Le, Y. Brun, and N. Medvidovic (2018). "Recovering Architectural Design Decisions". In: *International Conference on Software Architecture (ICSA)*. Seattle, WA: IEEE, pp. 95–104. DOI: 10.1109/ICSA.2018.00019
P18	P. N. Sharma, B. T. R. Savarimuthu, and N. Stanger (2021). "Extracting Rationale for Open Source Software Development Decisions — A Study of Python Email Archives". In: *43rd International Conference on Software Engineering (ICSE)*. Madrid, Spain: IEEE, pp. 1008–1019. DOI: 10.1109/ICSE43902.2021.00095
P19	P. N. Sharma, B. T. R. Savarimuthu, and N. Stanger (2023). "How are decisions made in open source software communities? — Uncovering rationale from python email repositories". In: *Journal of Software: Evolution and Process* November 2022, pp. 1–29. DOI: 10.1002/smr.2526
P20	L. Shi, Z. Jiang, Y. Yang, X. Chen, Y. Zhang, F. Mu, H. Jiang, and Q. Wang (2021). "ISPY: Automatic Issue-Solution Pair Extraction from Community Live Chats". In: *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021*, pp. 142–154. DOI: 10.1109/ASE51524.2021.9678894

Automatic Linking

Tracing between decisions and other artifacts, such as requirements, architecture, and implementation, is essential for rationale exploitation. Approaches for *automatic linking*, also called *automatic tracing* or *link recommendation*, automate the manual linking between decisions and other artifacts. Table 4.5 lists the two relevant primary publications on automatic linking.

Table 4.4.: Primary publications on automatic linking.

ID	Primary Publication
P21	G. Buchgeher and R. Weinreich (2011). "Automatic Tracing of Decisions to Architecture and Implementation". In: *Ninth Working IEEE/IFIP Conference on Software Architecture (WICSA)*. Boulder, CO, USA: IEEE, pp. 46–55. DOI: 10.1109/WICSA.2011.16
P22	C. Miesbauer and R. Weinreich (2012). "Capturing and Maintaining Architectural Knowledge Using Context Information". In: *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA)*. Helsinki, Finland: IEEE, pp. 206–210. DOI: 10.1109/WICSA-ECSA.212.30

Decision Guidance

Decision guidance approaches support software practitioners in decision making using a knowledge base. The knowledge base contains general decision knowledge, such as regarding design patterns, or reusable decision models generated from other projects. Decision guidance approaches distinguish between the problem and solution space. The *problem space* anticipates decisions to be made along with their solution options for a domain. For example, the problem space of the technical domain of cloud computing collects decision problems and solution options for migrating applications to the cloud. The *solution space* contains the decision problems, decisions, and related decision knowledge relevant to a specific project. The solution space is a tailored

problem space, i. e., trimmed down to the decision knowledge for the project (Zimmermann et al., 2015). Thirteen primary publications present a decision guidance approach (Table 4.5).

Table 4.5.: Primary publications on decision guidance.

ID	Primary Publication on Decision Guidance
P23	M. Bhat, K. Shumaiev, A. Biesdorf, M. Hassel, U. Hohenstein, and F. Matthes (2017a). "An ontology-based approach for software architecture recommendations". In: *Twenty-third Americas Conference on Information Systems (AMCIS)*. Boston, Massachusetts, USA, p. 10
P4	M. Bhat, C. Tinnes, K. Shumaiev, A. Biesdorf, U. Hohenstein, and F. Matthes (2019). "ADeX: A Tool for Automatic Curation of Design Decision Knowledge for Architectural Decision Recommendations". In: *International Conference on Software Architecture Companion (ICSA-C)*. Hamburg, Germany: IEEE, pp. 158–161. DOI: 10.1109/ICSA-C.2019.00035
P24	K. Brandner and R. Weinreich (2019). "A Recommender System for Software Architecture Decision Making". In: *13th European Conference on Software Architecture (ECSA)*. vol. 2. Paris, France: ACM, pp. 22–25. DOI: 10.1145/3344948.3344959
P25	Z. Durdik and R. H. Reussner (2013). "On the Appropriate Rationale for Using Design Patterns and Pattern Documentation". In: *9th International ACM SIGSOFT Conference on the Quality of Software Architectures (QoSA)*. Vancouver, BC, Canada: ACM, pp. 107–116. DOI: 10.1145/2465478.2465491
P26	P. Gaubatz, I. Lytra, and U. Zdun (2015). "Automatic Enforcement of Constraints in Real-time Collaborative Architectural Decision Making". In: *Journal of Systems and Software* 103, pp. 128–149. DOI: 10.1016/j.jss.2015.01.056
P27	S. Gerdes, M. Soliman, and M. Riebisch (2015). "Decision Buddy: Tool Support for Constraint-Based Design Decisions during System Evolution". In: *1st International Workshop on Future of Software Architecture Design Assistants (FoSADA)*. Montreal, QC, Canada: ACM, pp. 13–18. DOI: 10.1145/2751491.2751495
P28	S. Haselböck, R. Weinreich, and G. Buchgeher (2019). "Using Decision Models for Documenting Microservice Architectures: A Student Experiment and Focus Group Study". In: *International Conference on Service-Oriented System Engineering (SOSE)*. San Francisco, CA, USA: IEEE, pp. 37–3709. DOI: 10.1109/SOSE.2019.00016
P29	I. Lytra, H. Tran, and U. Zdun (2013). "Supporting Consistency between Architectural Design Decisions and Component Models through Reusable Architectural Knowledge Transformations". In: *Software Architecture: 7th European Conference, ECSA 2013, Montpellier, France, July 1-5, 2013, Proceedings*. Ed. by K. Drira. Vol. LNCS 7957. Lecture Notes in Computer Science. Montpellier, France: Springer, pp. 224–239. DOI: 10.1007/978-3-642-39031-9_20
P30	I. Lytra and U. Zdun (2014). "Inconsistency Management between Architectural Decisions and Designs Using Constraints and Model Fixes". In: *23rd Australian Software Engineering Conference*. Milsons Point, NSW, Australia: IEEE, pp. 230–239. DOI: 10.1109/ASWEC.2014.33
P31	I. Lytra, H. Tran, and U. Zdun (2015). "Harmonizing Architectural Decisions with Component View Models using Reusable Architectural Knowledge Transformations and Constraints". In: *Future Generation Computer Systems* 47, pp. 80–96. DOI: 10.1016/j.future.2014.11.010
P32	I. C. Lopes Silva, P. H. S. Brito, B. F. dos S. Neto, E. Costa, and A. A. Silva (2015). "A decision-making tool to support architectural designs based on quality attributes". In: *30th Annual ACM Symposium on Applied Computing (SAC)*. Salamanca, Spain: ACM, pp. 1457–1463. DOI: 10.1145/2695664.2695928
P33	W. Wang and J. E. Burge (2010). "Using Rationale to Support Pattern-Based Architectural Design". In: *ICSE Workshop on Sharing and Reusing Architectural Knowledge - SHARK '10*. Cape Town, South Africa: ACM, pp. 1–8. DOI: 10.1145/1833335.1833336
P34	O. Zimmermann, L. Wegmann, H. Koziolek, and T. Goldschmidt (2015). "Architectural Decision Guidance across Projects: Problem Space Modeling, Decision Backlog Management and Cloud Computing Knowledge". In: *12th Working IEEE/IFIP Conference on Software Architecture (WICSA '15)*. Ed. by L. Bass, P. Lago, and P. Kruchten. Montréal, Québec, Canada: IEEE, pp. 85–94. DOI: 10.1109/WICSA.2015.29

Consistency Support

Approaches for *consistency support* help software practitioners in performing consistent changes and maintaining consistent documentation. They either support the consistency among decisions

(P35) or between decisions and other artifacts (P29, P30, P31, P36, P37). The linking (integration) and visualization of decision knowledge with other artifacts help to estimate the impact of changes and to detect inconsistency (Cleland-Huang et al., 2013; Manteuffel et al., 2015). The approaches in this section go beyond the mere integration of decisions with other artifacts and knowledge visualization. Table 4.6 lists the six relevant primary publications on consistency support.

Table 4.6.: Primary publications on consistency support.

ID	Primary Publication on Consistency Support
P35	C. Carrillo and R. Capilla (2018). "Ripple Effect to Evaluate the Impact of Changes in Architectural Design Decisions". In: *12th European Conference on Software Architecture (ECSA'18)*. Madrid, Spain: ACM, pp. 1–8. DOI: 10.1145/3241403.3241446
P36	I. Lytra, H. Tran, and U. Zdun (2012). "Constraint-Based Consistency Checking between Design Decisions and Component Models for Supporting Software Architecture Evolution". In: *16th European Conference on Software Maintenance and Reengineering*. Szeged, Hungary: IEEE, pp. 287–296. DOI: 10.1109/CSMR.2012.36
P29	I. Lytra, H. Tran, and U. Zdun (2013). "Supporting Consistency between Architectural Design Decisions and Component Models through Reusable Architectural Knowledge Transformations". In: *Software Architecture: 7th European Conference, ECSA 2013, Montpellier, France, July 1-5, 2013, Proceedings*. Ed. by K. Drira. Vol. LNCS 7957. Lecture Notes in Computer Science. Montpellier, France: Springer, pp. 224–239. DOI: 10.1007/978-3-642-39031-9_20
P30	I. Lytra and U. Zdun (2014). "Inconsistency Management between Architectural Decisions and Designs Using Constraints and Model Fixes". In: *23rd Australian Software Engineering Conference*. Milsons Point, NSW, Australia: IEEE, pp. 230–239. DOI: 10.1109/ASWEC.2014.33
P31	I. Lytra, H. Tran, and U. Zdun (2015). "Harmonizing Architectural Decisions with Component View Models using Reusable Architectural Knowledge Transformations and Constraints". In: *Future Generation Computer Systems* 47, pp. 80–96. DOI: 10.1016/j.future.2014.11.010
P37	L. Zhang, Y. Sun, H. Song, F. Chauvel, and H. Mei (2011). "Detecting Architecture Erosion by Design Decision of Architectural Pattern". In: *23rd International Conference on Software Engineering and Knowledge Engineering*. Skokie, IL, USA: Knowledge Systems Institute Graduate School, pp. 758–763

4.2.2. Support for Software Practitioners

This section presents the results for the question *How do the approaches support software practitioners?* (RQ1.2). First, it describes the treatment of rational management problems and the support for activities. Second, it presents the tool support. Third, it discusses the results.

Rationale Management Problems Treated and Activities Supported

This section presents the results on *Which rationale management problems are treated?* and *Which rationale management activities are supported?* Table 4.7 shows the problems treated and the activities supported by the approaches. A checkmark ✓ indicates a focus, whereas a cross ✗ indicates no focus. A checkmark in brackets (✓) indicates that some publications have a focus, but not all. All the approaches treat the problem of *intrusiveness and effort* since they aim to reduce developers' manual work. Still, the intrusiveness depends on whether developers can use the tool support within their workflows, as discussed in the next section. All the approaches treat the problem of a *high amount of distributed knowledge* since they support rationale management activities that become demanding or exhausting when the amount of knowledge is increased. All the approaches treat the problem of *low documentation quality* to some extent by completing the documentation. In particular, consistency support helps maintain consistent documentation.

Decision making is supported by the decision guidance approach and—to some extent—by consistency support for deciding whether changes are integrated based on change impact analysis. *Rationale documentation* is supported by decision guidance through the creation of the solution

Table 4.7.: Rationale management problems treated and activities supported by the approaches.

Approach	Automatic Text Classification	Automatic Linking	Decision Guidance	Consistency Support
Rationale Management Problems				
Intrusiveness and effort	(✓)	(✓)	(✓)	(✓)
High amount of distributed knowledge	✓	✓	✓	✓
Low documentation quality	(✓)	(✓)	(✓)	✓
Rationale Management Activities				
Decision making	✗	✗	✓	(✓)
Rationale documentation	(✓)	(✓)	✓	✗
Rationale exploitation	✓	✓	(✓)	✗
Rationale quality assurance	(✓)	(✓)	(✓)	✓

space model. The automatic text classification and linking approaches could help to document new decision knowledge elements and links during development. However, automatic text classification and linking are mainly intended for retrospective *rationale exploitation* in the publications. Decision guidance supports exploiting external knowledge from other projects or knowledge bases. The approaches support *rationale quality assurance* by treating the problem of low documentation quality.

Tool Integration

This section presents the results on *Are the approaches supported with tools, and how do the tools integrate into the process?* Of the 37 publications, 12 (32.4 %) present plug-ins for commonly used development tools (Figure 4.3). The plug-ins are less intrusive than separate tools since they directly integrate into the development process. Most publications present Eclipse plug-ins (Table 4.8). Only two publications present extensions for other tools, namely for Slack (P7) and for the Enterprise Architect (P34).

Table 4.8.: Approach implementation into tools. Publications presenting the same tool are comma-separated. Publications presenting different tools are semicolon-separated.

	Automatic Text Classification	Automatic Linking	Decision Guidance	Consistency Support
Plug-Ins for Commonly Used Tools	**Slack**: P7	**Eclipse**: P21, P22	**Eclipse**: P25; P29, P30, P31 (ADvISE); P33 (SEURAT_Architect); **Enterprise Architect**: P34 (ADMentor)	**Eclipse**: P36, P29, P30, P31 (ADvISE); P37
Separate Tools	P1, P2 (A-REACT); P3, P4 (ADeX); P5; P6; P8, P9; P10; P11; P12; P13 (TReX); P14, P15, P16; P17 (RecovAr); P18, P19 (Rationale Miner); P20		P23, P4 (ADeX); P24; P26 (CoCoADvISE); P27 (Decision Buddy); P28; P32	P35

Of the 37 publications, 26 (68.4 %) publications present separate (research) tools (Table 4.8). Note that separate tools can have interfaces to commonly used development tools, for example,

ADeX in publication P4 by Bhat et al. (2019). ADeX offers the SyncPipe component to extract data from various systems, such as Jira, GitHub, Microsoft Project, and Enterprise Architect. The document-classifier component of ADeX offers automatic text classification.

Most existing approaches apply the automatic classification retrospectively on a ground truth that the researchers created. Only a few approaches integrate automatic text classification into tools and workflows for software development. As part of their future work, Rogers et al. (2015) and Lester et al. (2020), i. e., the authors of P15 and P10, plan to integrate text classification into existing knowledge management tools. Lester et al. (2020) work on text classification integration into the C_SEURAT tool for *Collaborative Software Engineering Using Rationale*. Rogers et al. (2015) aim to integrate automatic text classification into a wiki system.

Discussion: How do the approaches support software practitioners?

The approaches help software practitioners to perform rationale management activities and overcome problems. However, only a few approaches are implemented as plug-ins for current development tools. Most tools are separate, requiring software practitioners to use additional tools and making them intrusive. The results indicate a research gap in developing and investigating classification and recommendation support for rationale management that directly integrates into the tools used by software practitioners, such as issue tracking systems, version control systems, wikis, chat systems, and integrated development environments (as identified in Chapter 3).

4.2.3. Machine Learning Techniques and Rules Applied in the Approaches

This section presents the results for the question *How do the approaches internally work?* (RQ1.3). Figure 4.3 shows the number of publications per approach fulfilling a synthesis criterion. The publications P4, P29 – P31 are counted twice since they contribute to two approaches.

While 16 (80 %) of the publications on automatic text classification contribute a *machine learning-based* approach, all approaches for automatic linking, decision guidance, and consistency support are *rule-based* (Figure 4.3). Machine learning-based approaches apply machine-learning algorithms to preprocessed data with classification features. All identified approaches are supervised and, thus, need a labeled ground truth for the learning. The preprocessing and extraction of classification features result in vectors and can involve the following techniques: Sentence splitting, tokenization, stop-word removal, sentence length filtering, stemming, lemmatization, term-frequency inverse document frequency, Word2Vec, part-of-speech tagging, bag-of-words, skip-gram, n-grams, or data balancing. Algorithms applied are, for instance, Naïve Bayes, Logistic Regression, Support Vector Machine, Decision Tree, K-Nearest Neighbors, Random Forests, or Convolutional Neural Networks. The approaches in P7 and P20 apply the pre-trained machine-learning technique *Bidirectional Encoder Representations from Transformers (BERT)*.

Of the 41 approaches, 25 (61 %) are rule-based (Figure 4.3). Rule-based automatic text classification is presented in P13, P17, P18, P19. The automatic text classification approach in P13 uses ontologies and text mining techniques to recover rationale from plain-text documents. The RecovAr approach in P17 maps architectural changes to tickets in Jira to extract the decisions. The Rationale Miner in P18 and P19 uses language term patterns containing specific words, such as *idea* or *proposal*, proximity-based heuristics based on the dates and location of sentences in paragraphs, and other heuristics. The automatic linking approach in P21 creates interaction logs by observing developers during architecture design and implementation work and determines potential links to currently active decisions. The automatic linking approach in P22 calculates an indicator for the relation of two architectural elements using various metrics (called context information providers), such as the textual similarity between two elements, documentation time, author, and existing trace links. The decision guidance approach in P23 and P4 queries the external knowledge base DBPedia to retrieve alternatives for decisions and to

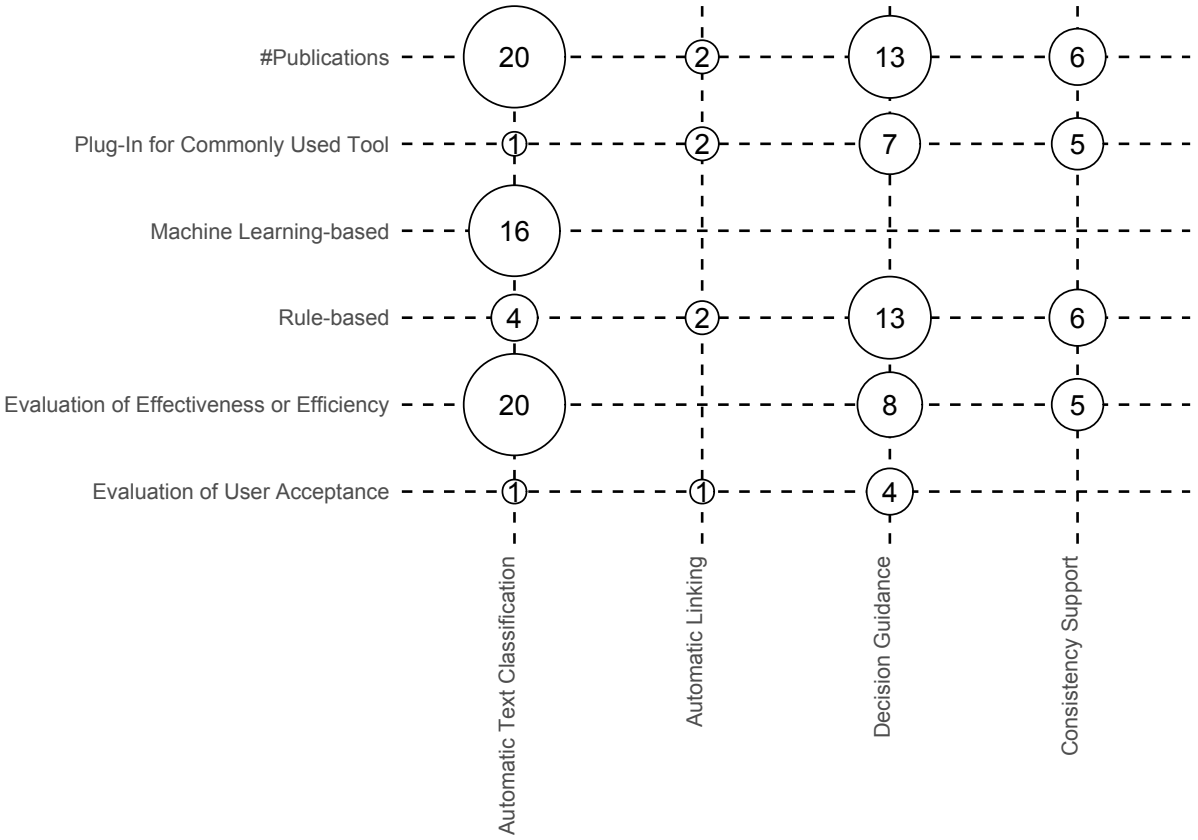

Figure 4.3.: Number of publications per approach (x-axis) and synthesis criterion (y-axis).

guide their implementation. The decision guidance approaches in P25, P26, and P29 – P31 apply questionnaires based on reusable decision models to guide design decisions. For example, the ADvISE tool assists architectural decision making by presenting a set of questions along with potential options for design issues. The approach in P29 – P31 transforms the decisions selected through the questionnaire into an architecture component model. The consistency support in P35 estimates the ripple effect, i. e., how a change in a design decision may affect other decisions by traversing and weighing dependencies between decisions by their type. The consistency support in P29 – P31, P36, and P37 maps decisions onto components and uses constraints for consistency checking between architectural design decisions and component models.

The publication P4 presents a machine learning-based approach for automatic text classification and a rule-based decision guidance approach.

4.2.4. Evaluation of Approaches

This section presents the results for the question *Which evaluation aspects are investigated?* (RQ1.4). The researchers of the publications validated various aspects, such as scalability or reusability. This study investigates the following aspects defined in the *Quality in Use* and *Technology Acceptance* models (Section 1.3): *Feasibility* defines whether stakeholders can apply the treatment. *Effectiveness* defines how well the approach can achieve a goal. *Efficiency* defines how efficiently the approach can achieve a goal (ISO/IEC 25010, 2011). *User acceptance* expresses the users' attitude toward the approach (Davis et al., 1989; Marangunić and Granić, 2015).

Of the 37 publications, five (13.5 %) are only validated through the description of illustrative examples, e. g., of the tool usage, to demonstrate the *feasibility* (Table 4.9).

Table 4.9.: Evaluation aspects investigated in the primary studies per approach.

	Automatic Text Classification	Automatic Linking	Decision Guidance	Consistency Support
Feasibility Only		P22	P27; P32; P33	P37
Effectiveness or Efficiency	P1, P2; P3, P4; P5; P6; P7; P8, P9; P10; P11; P12; P13; P14, P15, P16; P17; P18, P19; P20		P23, P4; P24; P26; P28; P29, P30, P31	P35; P36, P29, P30, P31
User Acceptance	P4	P21	P4; P25; P28; P34	

The remaining 32 (86.5 %) publications provide a more thorough evaluation. Five (13.5 %) of the 37 publications investigate the *user acceptance*, e. g., by implementing the approach in a case study with practitioners and interviewing or surveying their attitude (Figure 4.3, Table 4.9). Of the 37 publications, 29 (78.4 %) investigate the *effectiveness*, in terms of accuracy or completeness, or the *efficiency* in terms of time spent (Figure 4.3, Table 4.9). Various publications on automatic text classification measure precision, recall, and F-scores for the effectiveness evaluation, synthesized later in the thesis in Section 10.3. In other publications, the researchers compare the results (effectiveness) or time spent (efficiency) for a development task that practitioners using the approach achieved with those of a control group. Two publications P4 and P28 validate both user acceptance and effectiveness or efficiency.

The results reveal a research gap in validating the user acceptance of the approaches. In particular, only one of the 20 publications on automatic text classification (P4) reports on a user study, but the description of results is very brief.

4.3. Threats to Validity

This section discusses three validity criteria of secondary studies (Ampatzoglou et al., 2020):

Study selection validity concerns the publication search and selection with criteria. A threat is to miss out on relevant publications. For the initial search, we performed backward snowballing on existing literature overviews of rationale management approaches and queried two online databases. To find adequate publications in the online databases, we constructed the search string systematically considering the terms in the research question. However, the study defines recommendation approaches broadly, which might not be reflected in the search string. The study also includes approaches that guide and support software developers, e. g., through automatic linking and change impact analysis. While the two online databases ACM and IEEE Xplore are commonly used in secondary studies in software engineering, we could have queried more databases, such as Springer, Scopus, and Science Direct. We performed forward and backward snowballing on the initial results to mitigate the threat of missing relevant publications.

Data validity concerns data extraction and analysis. A threat is that the results of the systematic mapping study might be biased from inaccurate results reported in the primary studies. To mitigate the threat of invalid primary studies, we only included peer-reviewed publications and publications that present an empirical validation of their approach.

Research validity concerns the study design. The thesis author conducted the publication search, selection, data extraction, and analysis, and the supervisors reviewed the procedure. A threat is that the study might not be repeatable because other researchers might select other publications as relevant. To support repeatability and make the search and selection procedure transparent, Appendix A contains a protocol with the found and excluded publications.

4.4. Conclusion

This chapter presented a systematic mapping study to investigate the state of the art of (semi-)automatic rationale management support with classification and recommendation. It contributed an overview of four approaches for automatic text classification, automatic linking, decision guidance, and consistency support.

The study identified the following future research directions for primary studies: 1) More classification and recommendation approaches for rationale management should be integrated into easily usable tools for software practitioners, e. g., as plug-ins for standard development tools, such as issue tracking systems, version control systems, wikis, chat systems, and integrated development environments. 2) Different rule-based and machine learning-based techniques could be combined. 3) The approach evaluation should be done from different perspectives; user acceptance should be studied. We argue that the approaches can only be adopted in practice if researchers integrate them into tools for practitioners and if they validate user acceptance.

The remainder of the thesis builds on the study results and addresses the identified research directions. ConDec's recommendation systems implement the approaches: Automatic text classification (Section 7.6.8), decision guidance (Section 7.6.6), link recommendation (Section 7.6.7) implementing automatic linking, and change impact analysis (Section 7.6.5) implementing consistency support. The contribution of ConDec is the integration of the approaches into the issue tracking system so that software practitioners can easily use them in CSE. The thesis validates ConDec from different perspectives. Chapter 10 investigates the effectiveness of automatic text classification, and Chapter 11 validates the user acceptance of the approaches implemented into ConDec.

This study gave an overview of the existing approaches. A follow-up systematic literature review could further detail the approaches' internal functioning and evaluation in the future. Section 10.3 will synthesize automatic text classification in more detail.

Part III.

Treatment Design

Chapter 5

Overview of Continuous Rationale Management and its Support with ConDec

"Rationale models help us deal with change. Unfortunately, rationale is itself subject to change when we revise decisions."
—Bruegge and Dutoit, 2010

This part of the thesis describes the realization of the technical research goal: *Design a life cycle model and tool support for continuous rationale management that treats the problems of 1) intrusiveness and effort, 2) high amount of distributed knowledge, and 3) low documentation quality. The goal is to support a) collaborative, incremental, and rational decision making, b) documentation, c) exploitation, and d) quality assurance of decision knowledge.* This chapter presents an overview of the treatment design. The treatment consists of the Continuous Rationale Management (ConRat) life cycle model extension (detailed in Chapter 6) and the ConDec plug-ins that support ConRat (detailed in Chapter 7).

Section 5.1 describes a scenario where software developers perform ConRat and use the ConDec plug-ins. It illustrates the perspective of practitioners, who are the main stakeholders of the thesis. The scenario provides a first overview of the integration of ConRat and ConDec into existing workflows and tools. Section 5.2 provides an overview of the design of ConRat and ConDec. It refines the technical research goal into sub-goals and presents decision problems and decisions in the design of ConRat and ConDec.

Parts of the treatment design were published: A first version of the ConRat knowledge model and the integration of rationale management into short-cycled CSE practices was published in Kleebaum et al. (2018a; 2018b; 2019d). An overview of the ConDec plug-ins and their application for rationale management was published in Kleebaum et al. (2019c; 2020; 2021a). Ideas for visualizing decision and usage knowledge were published in Johanssen et al. (2017b). The ConDec views and their features were published in Kleebaum et al. (2021c). The automatic text classification was published in Kleebaum et al. (2021b).

5.1. Usage of ConDec to Support Continuous Rationale Management

This section describes a scenario to illustrate how software practitioners in the roles of developers and requirements engineers perform ConRat and use the ConDec plug-ins. In Figure 5.1, the requirements engineer creates the user story *As a user, I want to choose a password so that I can securely log in to the system* and captures the issue whether passwords should pass a security check as well as the decision to integrate a library for it. The requirements engineer creates

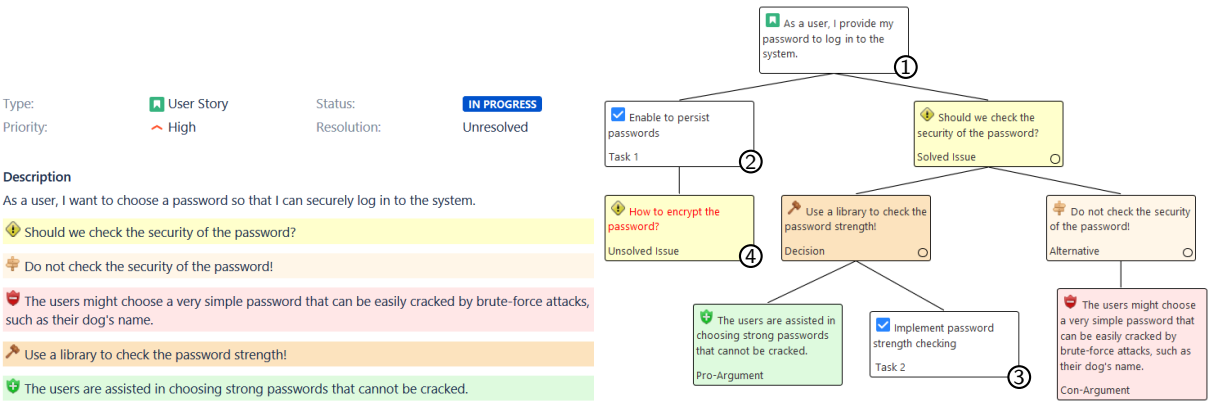

Figure 5.1.: **Left**: Decision knowledge captured in the description of a Jira user story. **Right**: The user story ①, development tasks ②, ③, and decision knowledge visualized as a tree. The summary of the unsolved issue ④ is highlighted with red font. The knowledge tree is interactive, e. g., via a context menu and drag & drop possibility.

the following development tasks that split the requirement: ☑ *Enable to persist passwords* and ☑ *Implement password strength checking*. The developer assigned to the first development task captures the issue ◆ *How to encrypt the password?* in a comment. The developers discuss the issue collaboratively during their work. ConDec provides various means to support the decision making: ConDec automatically integrates this issue into the knowledge graph, offering multiple visualizations. For instance, the developers see the knowledge tree, including the unsolved issue (Figure 5.1-④), when working on the user story, a related development task, or the linked code. ConDec also provides a rationale backlog that lists unsolved issues to support the discussion. During a meeting, the requirements engineer and developers spot unsolved issues from the meeting agenda (Figure 5.2, left). They orally decide to hash passwords with the SHA1 algorithm but do **not** document it. The developer implements the first development task on a feature

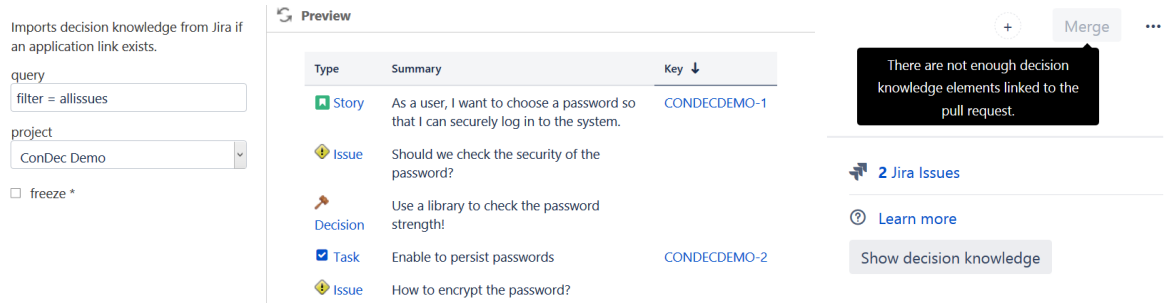

Figure 5.2.: **Left**: Macro to create a meeting agenda as part of the ConDec Confluence plug-in. **Right**: Merge check for fulfilling the definition of done (decision coverage and intra-rationale completeness) in the ConDec Bitbucket plug-in.

branch. The branch cannot be merged to the mainline until the issue is solved (Figure 5.2, right). The developer makes a commit on this branch with the decision ◆ *Encrypt the password with the SHA1 algorithm* being part of the commit message (Figure 5.3, left). The text classifier automatically identifies the decision in the commit message. Since the branch and the commits are linked to the development task, ConDec automatically relates the decision to the issue. The issue is marked as solved, and the red highlighting is removed (Figure 5.3, right). The developer can merge the branch back to the mainline, as the documentation is complete in that both the issue and the decision are documented, i. e., it fulfills the definition of done.

```
developer@conrat  /home/dev0/myproject (CONDECDEMO-42.pw.storage)
$ git commit -a
Encrypt the password with the SHA1 algorithm
```

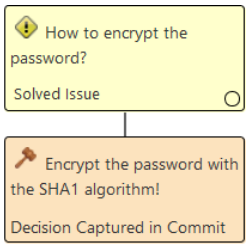

Figure 5.3.: **Left**: Commit message containing a design decision. The developer works on a branch related to a development task.
Right: Subtree of the knowledge tree in Figure 5.1 with the design decision as a new leaf node. The formerly open issue is now solved.

5.2. High-Level Decision Problems and Decisions

This section refines the technical research goal into three sub-goals treating the three rationale management problems intrusiveness and effort, high amount of distributed knowledge, and low documentation quality. It gives an overview of issues for the design of ConRat and ConDec, the key decisions, and solution alternatives if considered. The following chapters detail and explain the decisions. Figure 5.4 depicts the rationale management problems and the treatment through the ConRat life cycle model extension and the ConDec plug-ins. The ConRat life cycle model extension and the ConDec plug-ins are in a *task and support* relationship. ConRat defines the rationale management activities, i. e., tasks, that the developers perform. ConDec supports these tasks through tool features. The thesis uses the notations of *Task and Object-oriented Requirements Engineering* to specify tasks and the ConDec support (Paech and Kohler, 2004).

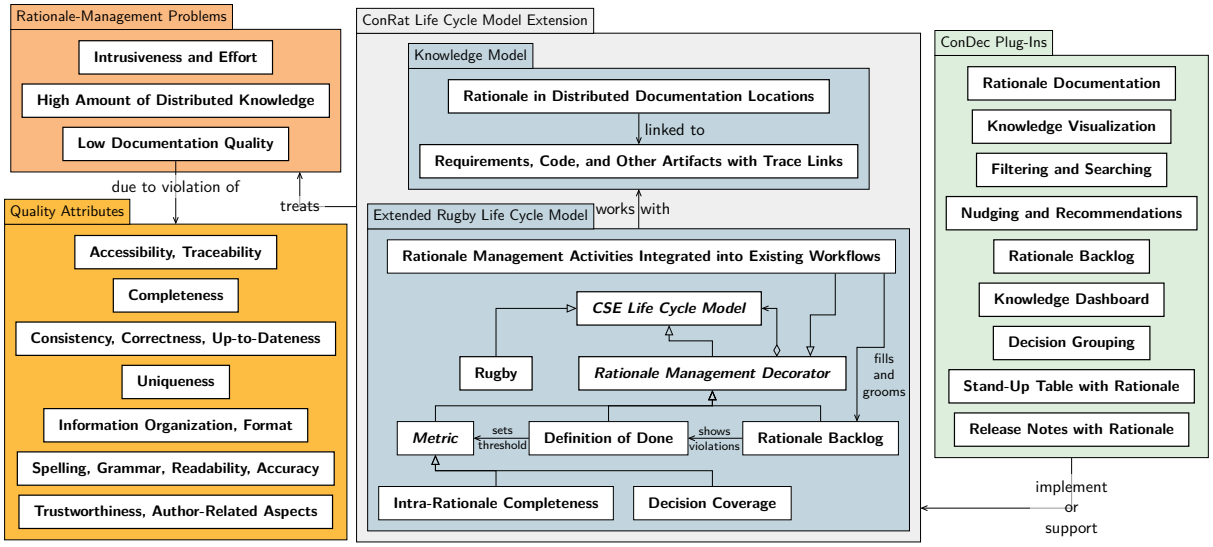

Figure 5.4.: Rationale management problems, quality attributes for software documentation (Zhi et al., 2015), and key decisions of the treatment through ConRat and the ConDec plug-ins (UML package diagram).

The ConRat life cycle model extension directly treats the rationale management problems. The ConDec plug-ins indirectly treat the problems by supporting ConRat or by implementing important concepts, such as metrics, a definition of done for knowledge documentation, and the rationale backlog. A high-level decision was to use the Rugby CSE life cycle model (Section 2.1.3) as the basis for the extension with explicit rationale management. Rugby is a CSE life cycle

model that models the workflows by the CSE roles. In Figure 5.4, the decision to extend Rugby is modeled by the usage of the decorator design pattern in the package Extended Rugby Life Cycle Model. The decorator design pattern consists of the abstract class *CSE Life Cycle Model* extended by Rugby. ConRat tailors Rugby by adding rationale management activities, which are modeled by the abstract class *Rationale Management Decorator*. The extending classes of *Rationale Management Decorator* are concrete decorators.

The following sections describe how ConRat and ConDec treat the three rationale management problems, outlined in Table 5.1, and detail further entities and relationships of Figure 5.4. Section 5.2.1 presents the treatment of the problem of intrusiveness and effort. Section 5.2.2 presents the treatment of the high amount of distributed knowledge. Section 5.2.3 presents the treatment of the low documentation quality.

Table 5.1.: Rationale management problems and their treatment through ConRat life cycle model extension and ConDec plug-ins.

Problem	Treatment through ConRat and ConDec
Intrusiveness and effort problem	**ConRat:** Rationale Management Activities Integrated into Existing Workflows **ConDec:** Integration into existing tools as ConDec Plug-Ins, support for easy Rationale Documentation in various distributed documentation locations, access to Knowledge Visualization from various tools and artifacts, Nudging and Recommendations (automatic text classification, summarization of source code changes), support for easy exploitation, e. g., in Stand-Up Table With Rationale and Release Notes With Rationale
High amount of distributed knowledge problem	**ConRat:** Knowledge Model with Rationale in Distributed Documentation Locations and Requirements, Code, and Other Artifacts with Trace Links **ConDec:** Instantiation of knowledge model as a knowledge graph, interactive Knowledge Visualization integrated into existing tools and accessible from various artifacts, Filtering and Searching (transitive linking), Decision Grouping
Low documentation quality problem	**ConRat:** Operationalization of knowledge quality through Decision Coverage and Intra-Rationale Completeness metrics, Definition of Done, Rationale Backlog **ConDec:** Implementation of metrics, definition of done, and Rationale Backlog, Nudging and Recommendations (quality checking, change impact analysis, decision guidance, link recommendation), Knowledge Visualization, Knowledge Dashboard

5.2.1. Treatment of Intrusiveness and Effort Problem

The first goal is to *support a) collaborative, incremental, and rational decision making, b) documentation, c) exploitation, and d) quality assurance of decision knowledge with low intrusiveness*. It addresses the *problem of intrusiveness and effort*. A rationale management activity is low intrusive if developers perform the activity in a lightweight way as part of their development practices, i. e., integrated into existing workflows. A rationale management activity is lightweight if developers only require a little effort. The thesis interchangeably uses the terms *little effort*, *lightweight*, and *easy*. Four issues are related to the treatment:

When should the developers perform the rationale management activities? A solution option would be to introduce new workflows for performing rationale management activities. For instance, the developers could document decision knowledge solely at the end of a sprint. However, this would result in a *big-bang* documentation that requires a lot of effort from developers at a particular time. A non-functional aspect requested by practitioners is the integration of rationale management into CSE rather than treating it separately (Chapter 3). For this reason, ConRat low-intrusively integrates the collaborative, incremental, and rational decision making,

documentation, exploitation, and quality assurance of decision knowledge into the short-cycled CSE workflows, for example, into working with requirements in the issue tracking system, implementing code, or working with branches in the version control system. During ConRat, the developers perform rationale management constantly in small increments over time instead of only once. ConDec integrates into existing tools as plug-ins rather than providing a separate tool. ConDec enables developers to document and exploit the knowledge during their workflows from various tools and CSE artifacts. For example, developers exploit the knowledge during meetings for rationale-based meeting management, when working on a requirement or other ticket, and when implementing requirements from code. The class Rationale Management Activities Integrated into Existing Workflows in Figure 5.4 models the decision. While ConRat integrates rationale management into CSE workflows, it also integrates lightweight, agile practices into rationale management, such as filling and grooming a rationale backlog and a definition of done for the knowledge documentation. The Rationale Backlog, Definition of Done for the knowledge documentation, and the abstract class *Metric* are concrete decorators of Rugby in Figure 5.4.

Where can developers document decision knowledge so that intrusiveness is low? A solution option to this issue would be creating a new software system for decision knowledge documentation, such as an external web-based platform. However, developers would need extra effort to install a new software system and change their development context for the documentation, which is too intrusive. The interview study with practitioners (Chapter 3) contributed documentation locations for decision knowledge. ConDec makes it possible to capture decisions and related decision knowledge in the documentation locations developers already use, e. g., in the issue tracking system, integrated development environment, and during committing in the version control system. Thus, developers are not required to change their development context to capture decision knowledge. In Figure 5.4, the Knowledge Model package bundles key decisions for the knowledge documentation. The class Rationale in Distributed Documentation Locations represents the decision to document decision knowledge in various places.

How can the developers easily formalize the decision knowledge? ConDec offers lightweight annotations to mark text as decision knowledge. ConDec supports the annotation through automatic text classification to reduce the documentation effort. Automatic text classification is based on the respective approach identified in the systematic mapping study (Chapter 4) and the practitioners' request for automation in the interview study (Chapter 3). In Figure 5.4, the classes Rationale Documentation and Nudging and Recommendations in the package ConDec Plug-Ins model the decisions.

How to motivate the developers to document decision knowledge despite additional effort? ConDec motivates the developers through exploitation support, recommendations, and nudging: The developers need to know when and how to exploit the decision knowledge documentation as incentives for the documentation. The ConDec plug-ins offer various exploitation features, such as an interactive knowledge visualization, a Stand-Up Table with Rationale for easy knowledge sharing during meetings, and a feature for semi-automatic creation of Release Notes with Rationale. ConDec provides recommendation systems that implement the four approaches identified in the systematic mapping study (Chapter 4): Automatic text classification, decision guidance, link recommendation implementing automatic linking, and change impact analysis implementing consistency support. ConDec also offers nudging mechanisms to motivate the developers. The respective classes of package ConDec Plug-Ins in Figure 5.4 model the features for exploitation, Nudging and Recommendations. A further motivation mechanism currently not implemented is gamification. Gamification can be seen as a nudging mechanism incorporating game elements, such as points and badges (García et al., 2017; Cursino et al., 2018). It has not yet been

implemented into ConDec because it often incorporates tracking individual developers' (players') history, which requires careful ethical considerations. It is conceptually more complicated than the current nudging mechanisms that solely work on knowledge documentation.

5.2.2. Treatment of High Amount of Distributed Knowledge Problem

The second goal is to *support a high amount of distributed knowledge*. It addresses the *problem of the high amount of distributed knowledge*. Three issues are related to the treatment:

What constitutes high amount of knowledge, i. e., how to quantify it? The thesis considers the amount of system knowledge (requirements and code) and decision knowledge. Defining threshold values for a high number of requirements and lines of code is challenging as there seem to be no definitions in the literature. The thesis considers the system knowledge documented in the validation projects in industrial settings, with a duration of about six months, a high amount. These projects resulted in between 27 to 68 documented requirements and code bases of 10 000 to 50 000 lines of code (Chapter 9). There do not appear to be definitions of a high amount of decision knowledge in the literature. The thesis considers the following threshold for a high number of decisions: at least one decision is documented for each requirement. In the validation projects, at least 27 to 68 decisions must be documented to consider the number as high. For each decision, an issue and at least one alternative, one pro-argument, and one con-argument must be documented. Thus, at least 135 to 340 rationale elements must be documented to consider the number as high in the validation projects.

How can the distributed knowledge be made accessible from various CSE artifacts? ConRat defines a knowledge model that builds on lightweight traceability between typical CSE artifacts and formalized decision knowledge documentation, represented by the Knowledge Model package in Figure 5.4. The lightweight traceability enables the developers to access and exploit the distributed knowledge. The knowledge model integrates decision knowledge from various distributed documentation locations, in particular, from the issue tracking system, commit messages, and code comments in the version control system, modeled by the class Rationale in Distributed Documentation Locations. The knowledge model enables the developers to access the relevant parts of the documentation from the artifacts they work on, modeled by the class Requirements, Code, and Other Artifacts with Trace Links. The ConDec plug-ins implement and visualize the knowledge model, modeled by the Knowledge Visualization class. Since the decision knowledge is documented in the issue tracking and version control systems, it has the same accessibility as tickets, commits, and code. Developers can access the documented decision knowledge when using Filtering and Searching functionalities provided by ConDec and the built-in search functionalities in the issue tracking system, version control system, and integrated development environment.

How can the high amount of knowledge be exploited without information overload? The practitioners of the interview study (Chapter 3) considered filtering and searching functionalities essential to reduce the amount of knowledge. ConDec offers Filtering and Searching functionalities that enable developers to customize, i. e., tailor, the knowledge visualizations. A contribution of ConDec is called *transitive linking*. Transitive linking is helpful for distributed knowledge documentation in combination with other filters. For instance, developers can examine decisions directly or indirectly linked to a particular requirement or code file, filtering out development tasks. Besides, ConDec offers the Decision Grouping feature that enables developers to filter for specific decision types, such as only process or design decisions.

5.2.3. Treatment of Low Documentation Quality Problem

The third goal is to *support the high quality of the documented decision knowledge*. It addresses the *problem of low documentation quality*. Nine issues are related to the treatment:

What constitutes high documentation quality? Zhi et al. (2015) collected quality attributes for software documentation in a systematic mapping study: accessibility, traceability, completeness, consistency, correctness, up-to-dateness, uniqueness, information organization, format, spelling and grammar, readability, accuracy, trustworthiness, and author-related aspects. High documentation quality means that all of these attributes are fulfilled. Figure 5.4 contains the Quality Attributes package with classes for related attributes. The quality attributes are defined as follows: Accessibility describes how easily practitioners can find the documented decision knowledge. Traceability means that practitioners can trace the decision knowledge to and from the related requirements, code, and other software artifacts. Traceability can be established through direct links or indirect links within a specific number of connections. Completeness describes how comprehensive the decision knowledge documentation is in supporting developers in their tasks. Consistency means that the decision knowledge has no conflict with other knowledge. Correctness means that the decision knowledge does not conflict with factual information. Up-to-Dateness means that the knowledge is kept updated during the evolution of software systems. Uniqueness addresses whether parts of the documentation are duplicated. Information organization concerns the organization of the documentation and Format addresses the writing style. The attributes Spelling, Grammar, Readability, and Accuracy refer to the correctness of spelling, grammar, and ease of reading the documentation. Readability describes how easily documents can be read. Accuracy represents the preciseness of documentation content. Trustworthiness means that software practitioners perceive the documents as reliable. Author-related aspects are traces of who created the documents and author collaboration.

What general concepts support high quality? ConRat integrates knowledge quality checking and enforcement into the CSE workflows similar to checking and enforcing unit test coverage. This decision is inspired by automated testing, which is an important CSE element to practitioners (Section 3.1.2). This decision also treats the intrusiveness and effort problem and is modeled by the class Rationale Management Activities Integrated into Existing Workflows in Figure 5.4. ConRat decorates the Rugby CSE life cycle model with the abstract class *Metric* as well as the classes Definition of Done and Rationale Backlog to support high quality. The rationale backlog lists the knowledge elements that violate the definition of done and is also a *decision-making support* (Chapter 3). The definition of done sets the thresholds that the metrics must exceed and the development teams can tailor it. The ConDec plug-ins implement the concepts and a Knowledge Dashboard that calculates and plots metrics, as requested by practitioners with the feature *metrics calculation and reporting in dashboard* (Chapter 3). During ConRat, reviewers check the quality of the rationale documentation, for instance, before accepting a merge request.

How do ConRat and ConDec support accessibility and traceability? As described in Section 5.2.2, ConRat establishes accessibility and traceability using the Knowledge Model consisting of Requirements, Code, and Other Artifacts with Trace Links and Rationale in Distributed Documentation Locations. ConDec provides tool support through Knowledge Visualization, Filtering and Searching including transitive linking, and Decision Grouping. Knowledge exploitation is impeded if trace links are incomplete or wrong. ConDec enables linking knowledge elements and marking wrong links, modeled with the Rationale Documentation class. Two mechanisms improve traceability in terms of complete links: 1) A criterion of the Definition of Done is that requirements and code files need to have a certain Decision Coverage (modeled as a concrete

Metric in Figure 5.4). ConDec offers mechanisms to check and enforce decision coverage, modeled by the Nudging and Recommendation and the Knowledge Dashboard classes. 2) ConDec offers the link recommendation feature that helps developers to access and link related issues and decisions. Link recommendation builds on the automatic linking approach identified in the systematic mapping study (Chapter 4) and is modeled as part of the Nudging and Recommendation.

How do ConRat and ConDec support completeness? ConRat and ConDec support two types of completeness: First, important decisions regarding eliciting and implementing requirements, code, and other artifacts must be made explicit. As with accessibility and traceability, the measurement and enforcement of the Decision Coverage metric as part of the Definition of Done and link recommendation support completeness between artifacts and decisions. Besides, ConDec offers the decision guidance feature, which implements the approach identified in the systematic mapping study (Chapter 4). It completes the documentation by making recommendations taken from external knowledge sources. Link recommendation and decision guidance are modeled as part of the Nudging and Recommendation class in Figure 5.4. Second, the decision knowledge must be completely documented, which is addressed by the Intra-Rationale Completeness metric as part of the Definition of Done.

How do ConRat and ConDec support consistency, correctness, and up-to-dateness? The Knowledge Visualization of ConDec visualizes instances of the Knowledge Model called knowledge graph. The visualization lets developers access the decision knowledge documentation from requirements, code, and other artifacts during their daily work. Developers continuously reflect on whether there are outdated, inconsistent knowledge elements or links and improve the documentation. On top of the mere presentation, ConDec offers change impact analysis, which implements consistency support identified in the systematic mapping study (Chapter 4). The decision guidance feature supports correctness by making recommendations from external knowledge sources. Change impact analysis and decision guidance are modeled as part of the Nudging and Recommendation class in Figure 5.4.

How do ConRat and ConDec support uniqueness? ConDec supports the identification of duplicates in the knowledge documentation as part of the link recommendation system, which is modeled as part of the Nudging and Recommendation class in Figure 5.4.

How do ConRat and ConDec support information organization and format? The Knowledge Model organizes the knowledge documentation for consistent Information Organization and Format. ConDec enables configuring the types of rationale elements and links used in the rationale model to support Information Organization and Format. ConRat demands that rationale elements are phrased in a specific way, e. g., issues as questions ending with a question mark and decisions ending with an exclamation mark. This criterion supports a coherent writing style and is part of the Definition of Done. The fulfillment is checked in reviews.

How do ConRat and ConDec support spelling, grammar, readability, and accuracy? ConRat demands the coherent phrasing of rationale elements to support readability. The coherent phrasing is part of the Definition of Done, and reviewers check the fulfillment.

How do ConRat and ConDec support trustworthiness and author-related aspects? During ConRat, reviewers check the trustworthiness, but ConDec does not offer particular features. Author-related aspects can be examined from the decision knowledge documentation, e. g., to identify developers who are experts or accountable for certain decisions. Accountability is a benefit of rationale management frequently mentioned by practitioners (Table 3.4).

Chapter 6

Life Cycle Modeling of Continuous Rationale Management

> *"We should not confuse the outcome of the design process with the process itself. The outcome of the design process is a 'rational reconstruction' of that process."*
>
> —Dingsøyr and van Vliet, 2009

This chapter presents a CSE life cycle model with Continuous Rationale Management (ConRat). ConRat extends the Rugby CSE life cycle model described in Section 2.1.3 by adding rationale management activities: collaborative, incremental, and rational decision making, documentation, exploitation, and quality assurance of decision knowledge. ConRat is based on a knowledge model describing the knowledge developers consume and produce during the activities.

The sections of the chapter explain the entities of the ConRat Life Cycle Model Extension in Figure 5.4. Section 6.1 presents the ConRat knowledge model for knowledge elements and their associations in CSE. The section explains different types of states for decision knowledge. It describes a demonstration project to exemplify ConRat. Section 6.2 presents the rationale-management extension of the Rugby CSE process model. The section provides metrics, the *definition of done* for the knowledge documentation, and the *rationale backlog*. It describes the roles involved in continuous rationale management, their tasks as workflows, and short-cycled CSE practices that include rationale management activities. Section 6.3 concludes the chapter.

6.1. Knowledge Model

The ConRat knowledge model consists of a graph of knowledge elements and associations between them and a dynamic model describing the states of the knowledge elements. Section 6.1.1 introduces the types of elements and associations. Section 6.1.2 describes the different types of states. Section 6.1.3 presents the instantiation of the knowledge model for an example project.

6.1.1. Knowledge Elements and Associations

Figure 6.1 shows the ConRat knowledge model including typical CSE artifacts. The knowledge elements and the associations can be of various types. High-level types of knowledge elements are *decision knowledge/rationale*, *system knowledge*, and *project knowledge* (Section 2.2). The *CSE artifacts* are also called *knowledge elements* because they model the knowledge by the stakeholders who created them.

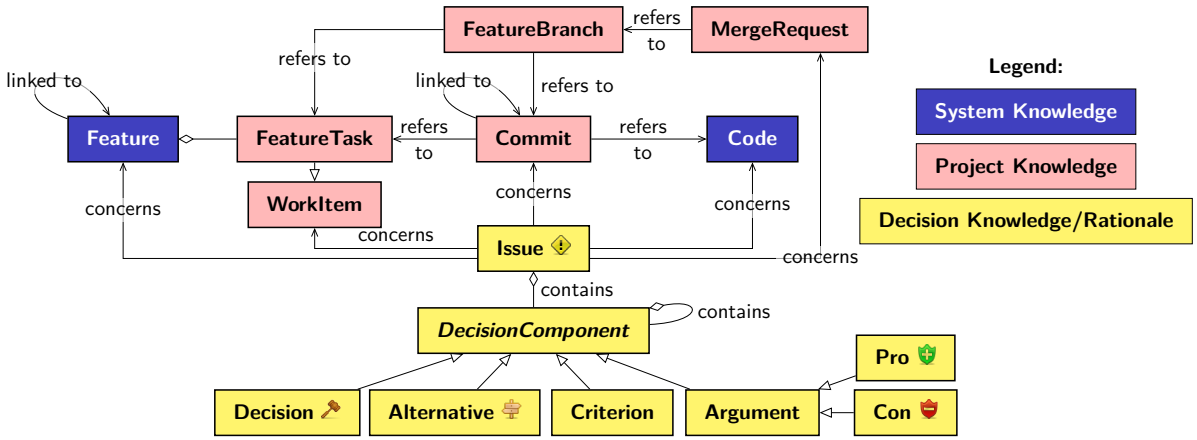

Figure 6.1.: ConRat knowledge model with CSE artifacts and associations: Features, tasks to implement a feature (feature tasks), work items, short-lived feature branches, commits, code, merge requests, and decision knowledge (UML class diagram).

The model focuses on Features and Code as essential system knowledge elements. Features represent both functional and quality requirements. They can be split into sub-features or contained in bigger features. Tasks that developers fulfill to implement a software feature are called Feature Tasks. A feature task is a development-related Work Item. Other work items unrelated to software features exist, such as setting up the CSE infrastructure. Short-lived Feature Branches encapsulate the development work (Krusche et al., 2014). A feature branch contains one or more Commits that refer to code files that were changed in the commit. A Merge Request handles whether a branch can be merged and supports that only high-quality code changes are integrated into the main code base. Work items, feature tasks, feature branches, merge requests, and commits are types of project knowledge.

The ConRat knowledge model is based on the *Decision Documentation Model* by Hesse (2020) to represent the decision knowledge in relation to project and system knowledge. The decision knowledge types are adopted from the *Issue-Based Information System* model: Issue, Alternative, Decision, Pro- and Con-argument. The Issue is the top element to emphasize that a decision should be made based on an issue, i. e., decision problem or question, and so that issues are linked to the system and project knowledge elements. As in the *Questions, Options, and Criteria* model, the ConRat knowledge model contains the Criterion type for decision making.

The ConRat model also supports other entities produced and consumed during the CSE workflows described in Section 6.2, in particular, user feedback reports, product backlogs, sprint backlogs, and releases. Additional documentation locations for decision knowledge exist, for instance, wiki pages, pull requests, chat messages, text files, and diagrams (Table 3.4). The artifacts in Figure 6.1 are the core set of CSE knowledge artifacts that enable lightweight tracing to make distributed documentation accessible. The ConRat model supports tracing between the system, project, and decision knowledge elements. Tracing can be accomplished either using a) textual annotations such as decision annotations (Hesse et al., 2015), b) feature task identifiers in the commit messages, c) distinctly documented trace links, e. g., within a table, and d) trace retrieval techniques (Cleland-Huang et al., 2013). Tracing is a prerequisite for developers to ensure the consistency of documented decision knowledge. Tracing enables developers to reflect decision, system, and project knowledge simultaneously. Developers can explore code and decision knowledge that evolved during the implementation of a feature. Likewise, developers can explore decision knowledge and features relevant to a specific code.

The implementation of the ConRat knowledge model is as follows: Work items, feature tasks, and features are stored in the issue tracking system, whereas code, commits, and merge requests

Table 6.1.: Synonymous names for knowledge elements in the ConRat knowledge model. For the decision knowledge elements, states and their synonyms are listed.

Name in Model	Synonymous Names
	System Knowledge
Feature	Functionality, software quality, (functional or quality) requirement, epic and user stories (that the epic is broken down into), use case and scenarios, user task and sub-tasks
	Project Knowledge
Feature task	Action item, work item, backlog item, development task, to-do, ticket, can also describe bug fixing task based on bug report or task based on change or improvement request
Merge request	Pull request
Feature branch	Topic branch, short-lived branch, feature task branch, i. e., branch related to a task to implement (parts of) a feature
	Decision Knowledge/Rationale
Issue	Decision problem, question; *States:* open, un(re)solved *or* closed, (re)solved
Decision	Solution, resolution; *States:* decided *or* challenged *or* rejected
Alternative	Solution option, option, proposal, position; *States:* idea *or* discarded

are stored in the version control system *git* (Chacon and Straub, 2014). Note that some types of knowledge elements in the knowledge model have synonymous names listed in Table 6.1 used throughout this thesis. For example, feature tasks are often called *tickets* (Saito et al., 2017), *work items* (Paech et al., 2014), *action items* (Bruegge and Dutoit, 2010), or *development tasks*. In the issue tracking system, developers capture decision knowledge elements linked to the respective features and feature tasks—either by textually capturing them in the description and comments of the features and feature tasks or by creating entire tickets for the decision knowledge elements. In the version control system, developers textually capture decision knowledge in commit messages and code. Developers mark it as such knowledge using *decision annotations* as suggested by Hesse et al. (2015). Trace links between tickets in the issue tracking system and code files in the version control system are created and maintained. The linking is commonly done commit-based using ticket identifiers in commit messages (Rath et al., 2018; Hübner and Paech, 2020). A good commit message expresses the rationale of the change and is linked to a ticket (Codoban et al., 2015; Al Safwan et al., 2022). Also, the tickets in the issue tracking system are linked. For example, a feature is linked to its feature tasks or—when using a hierarchical requirements model—epics are linked to user stories. In ConRat, developers use decision annotations, ticket identifiers in the commit messages, and distinctly documented trace links to establish tracing.

Instances of the knowledge model form a graph data structure called *knowledge graph*. The *nodes* and *edges* of the knowledge graph are the knowledge elements and associations, respectively. The knowledge graph tends to become big with many nodes and edges. Edges are also called *links* or *relationships*. Developers usually work with knowledge subgraphs, i. e., parts of the graph related to, for example, a specific feature. Knowledge subgraphs allow developers to consider previous decisions when working on a feature. Figure 6.1 omits attributes of the knowledge elements. However, knowledge elements and associations can have various attributes, such as a documentation location illustrated in Figure 6.2 or states described in Section 6.1.2. Figure 6.5 and Figure 6.15 later in this chapter show a knowledge subgraph at two different points in time.

The ConRat knowledge model is not the first to link features, work items, commits, and code, as shown in Figure 6.1. For example, Saito et al. (2017) and Rastkar and Murphy (2013) use a similar model. However, this is the first model to show how decision knowledge refers to the artifacts and to suggest decision annotations in commit messages.

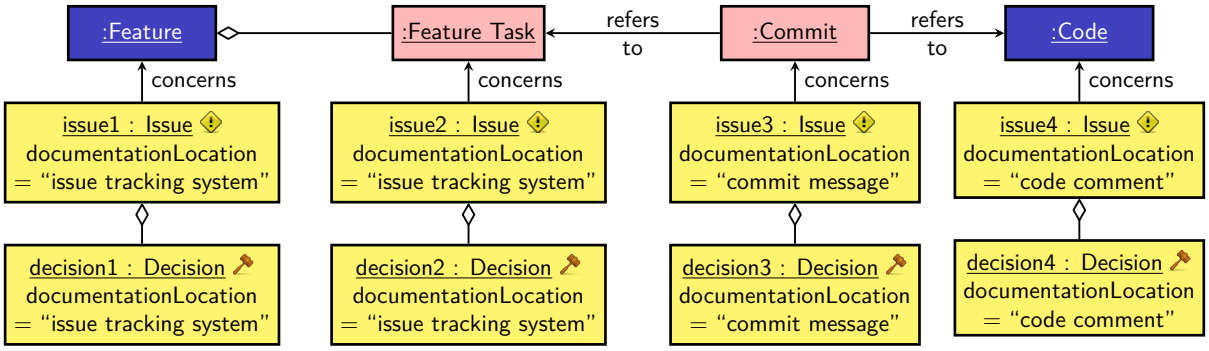

Figure 6.2.: Schematic illustration of a knowledge subgraph, i. e., an instance of the knowledge model (UML object diagram). Decisions 1 and 2 are documented in the feature and feature task description or comments, respectively. Decision 3 is captured in a commit message, and decision 4 is documented in code comments.

6.1.2. State of Rationale Elements

ConRat distinguishes the *decision-making state*, *documentation state*, and *implementation state*. The following subsections describe these states.

Decision-Making State

Figure 6.3 shows the decision-making states of issues, alternatives, and decisions and their transitions as used in ConRat. Issues (decision problems) can either be *unsolved/open* or *solved/closed* (Bruegge and Dutoit, 2010). For decisions and alternatives, ConRat uses a subset of the states suggested by Kruchten (2004), Kruchten et al. (2009), and van Heesch et al. (2012): Alternatives can either be an *idea* or *discarded*. Decisions can be *decided*, *challenged*, or *rejected*. Table 6.1 lists the possible states of issues, decisions, and alternatives along with their synonyms.

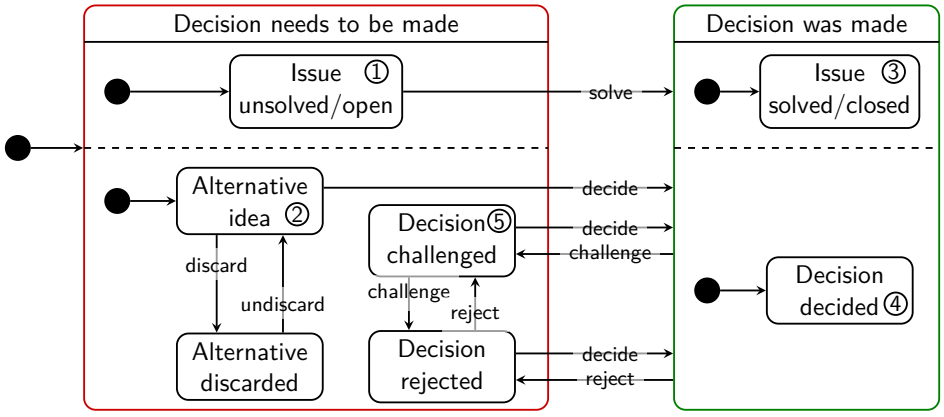

Figure 6.3.: Decision-making states of issues, alternatives, and decisions and their transitions (UML state machine diagram with parallel regions). Decision making and (collaborative) discussions are needed for the states on the left side. The decision was made for the states on the right side but could be changed.

The initial state of an issue is to be *unsolved* (Figure 6.3-①), while the initial state for a solution option (alternative) is to be an *idea* (Figure 6.3-②). An alternative that has never been a decision, i. e., was never incorporated in the system, can be *discarded*. The issue is *solved* (Figure 6.3-③) if the stakeholders decide on an alternative. The former alternative then becomes a decision with state *decided* (Figure 6.3-④). The UML state machine diagram does

not contain a final state because the stakeholders can change the decision. If a stakeholder raises a concern about a previously decided decision, the decision becomes *challenged* (Figure 6.3-⑤). The decision can be *rejected*, that is, in a state also referred to as *dropped* (Dragomir et al., 2014). While its implementation is removed from the system, the rejected decision is saved with the reason it was rejected. The respective issue becomes once again *unsolved* (Figure 6.3-①).

Documentation State

The developers' decision knowledge can be *tacit*, i. e., *implicit*, or *explicit* (Section 2.2). Tacit or implicit decision knowledge is based on experience and is difficult for developers to articulate. The documentation of decision knowledge can either be *informal*, i. e., unstructured in natural language texts, or *formalized* using the rationale model in Figure 6.1.

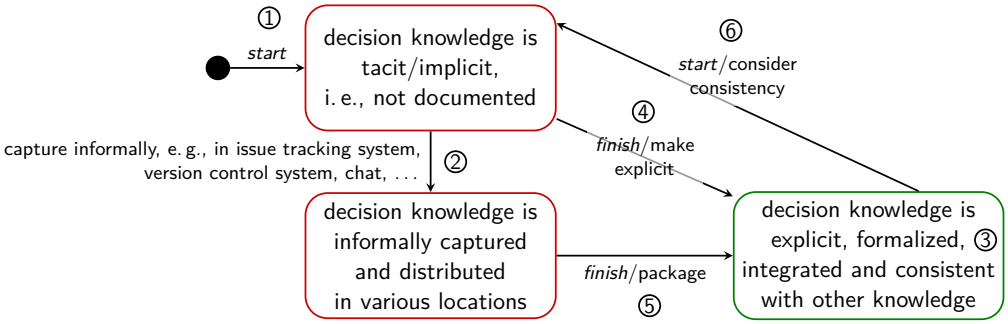

Figure 6.4.: Documentation states of decision knowledge (UML state machine diagram). The state on the lower right side is the preferred state.

Figure 6.3 shows the documentation states of decision knowledge. At the beginning of the work, decision knowledge is often *tacit* or *implicit* in the head of a few developers (Figure 6.4-①). Tacit knowledge is likely to vaporize and is hard to understand by others; thus, this first state is not preferred in ConRat. If decisions are not tacit, they are often discussed informally, captured partly and in a distributed manner (Figure 6.4-②), such as in ticket comments (Hesse et al., 2016b), commit messages, merge requests (Brunet et al., 2014), wikis, emails, chat messages (Alkadhi, 2018), or another documentation location (Table 3.4). Such knowledge is called *distributed decision knowledge*. Informally captured, distributed decision knowledge is hard to access later and might even be outdated, i. e., inconsistent with artifacts or other decision knowledge. Therefore, informally captured, distributed decision knowledge is also not preferred in ConRat. ConRat wants important decision knowledge to be explicit, formalized, as well as integrated and consistent with the software artifacts and other decision knowledge (Figure 6.4-③). ConRat encourages the developers to reach this preferred state when *finishing* their current work, e. g., when finishing the work on a feature task. Two ways exist to reach the preferred state: First, developers can *make the tacit decision knowledge explicit* and directly document it in a formalized way integrated with other knowledge (Figure 6.4-④). Second, developers can formalize the informally captured distributed decision knowledge and integrate it with other knowledge by linking it to the corresponding feature, feature task, code, or commits. The formalization and integration is called *packaging* (Figure 6.4-⑤). When the developers *start* new work, e. g., a new feature task, they use the documented knowledge to make new decisions consistent with the former ones, i. e., they *consider consistency* (Figure 6.4-⑥). The trace links in Figure 6.1 determine relevance: For instance, relevant decisions are those from other feature tasks of the same feature. Transitions between these states frequently occur during CSE, while some decision knowledge is in the tacit or informally captured state and other decision knowledge is in the formalized state. For this reason, the UML state machine diagram has no final state.

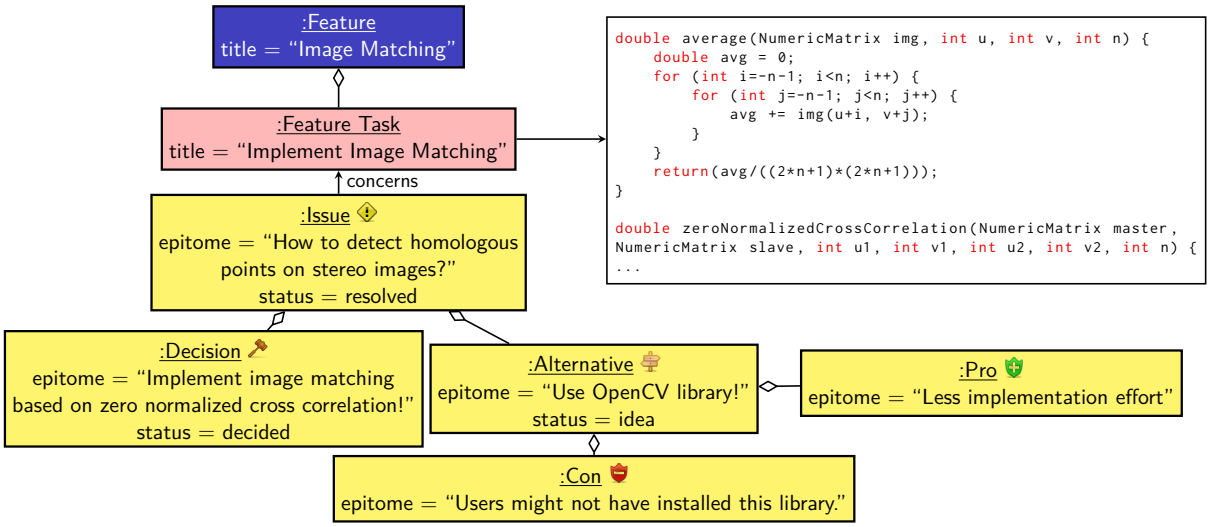

Figure 6.5.: Image-matching decision as an example instance of the knowledge model (UML object diagram). The status attribute refers to the decision-making state.

Implementation State

Decisions also have an *implementation state*, e. g., *envisioned* and *applied/incorporated* (Hesse, 2020). In ConRat, the implementation state of decisions is the same as for the linked work item. Thus, it is not tracked separately.

6.1.3. Demonstration Project

This chapter uses a radargrammetry project to demonstrate ConRat: Imagine the development of software that computes three-dimensional surface models of the earth from satellite stereo images[1]. The idea behind this software is the following: An *image matching* algorithm detects pixels belonging to the same object—homologous points—on the stereo images. The distance between the homologous points—the disparity—is then used to calculate the relative height of the object, e. g., of a mountain. Thus, image matching is one essential feature of the software. In this example, developers decide how to implement the image-matching feature.

A developer opens a feature task in the issue tracking system to implement image matching, indicated by the *start* transition in Figure 6.4-①. The developers discuss image-matching algorithms informally in chat messages and comments of the feature task and, thus, get into the state *decision knowledge is informally captured and distributed in various locations* (Figure 6.4-②). One developer proposes to implement a standard image-matching algorithm based on detecting the maximum zero normalized cross correlation. A second developer proposes to take advantage of algorithms provided by the open-source computer vision library (OpenCV), which they think comes with less implementation effort. However, this also introduces third-party code. Users will have a higher installation effort since they must ensure their operating system correctly provides the library. Finally, the developers decide to implement a custom image-matching algorithm and create a feature branch in the version control system. When merging this branch back to the mainline, they perform a *finish practice* and *package* the distributed knowledge (Figure 6.4-⑤). That means the developers document the decision knowledge in a formalized way in the issue tracking system and make sure that it is linked to the feature task and consistent with the implemented code (Figure 6.4-③). Figure 6.5 shows the instance of the knowledge model.

[1] Radargrammetry R package

6.2. Extended Rugby Life Cycle Model

Continuous rationale management is not a single task by a single role but integrates into existing tasks by the CSE roles. Rugby is a CSE life cycle model that describes the workflows, i. e., the tasks by the roles (Section 2.1). ConRat extends the Rugby life cycle model by tailoring the existing workflows and adding new workflow descriptions. ConRat adds explicit rationale management to Rugby: collaborative, incremental, and rational decision making, documentation, exploitation, and quality assurance of decision knowledge.

Section 6.2.1 describes metrics for the knowledge documentation, Section 6.2.2 the definition of done, and Section 6.2.3 the rationale backlog. These concepts are means to operationalize the quality of knowledge documentation. Section 6.2.4 describes the life cycle model and Section 6.2.5 parallel workflows. Section 6.2.6 presents CSE practices ideal for breaking down rationale management into small, manageable pieces.

6.2.1. Metrics for Rationale Documentation

A software life cycle model can be characterized by its level of maturity. The *capability maturity model* distinguishes five levels: 1) initial, 2) repeatable, 3) defined, 4) managed, and 5) optimized (Paulk et al., 1993). A software process has only reached the fourth level if it defines and measures metrics for its activities (workflows) and deliverables (entities produced and consumed during the workflows). The following subsections define two metrics regarding the rationale documentation: *intra-rationale completeness* and *rationale coverage*. Both metrics are concerned with the structure of the knowledge documentation and, thus, are syntactic metrics. They help to look for information that is missing (Burge and D. C. Brown, 2008a), which means they support the completeness of the knowledge documentation.

Intra-Rationale Completeness Metrics

The *intra-rationale completeness* assesses whether a single decision or a single decision problem is completely documented so that the decision-making process can be better understood. That means it assesses whether *all decision knowledge elements are documented that justify a decision.* ConRat judges intra-rationale completeness using the following criteria (formulated as questions): 1) Is the decision problem solved, i. e., is a decision documented? 2) Is the decision problem for the decision documented? 3) Is there at least one alternative documented for the decision problem, i. e., are at least two solution options (decision and alternative) documented? 4) Is there at least one pro-argument documented for the decision? 5) Is there at least one con-argument documented for the alternative? 6) Does the decision have more pro-arguments than its alternatives?

ConRat's intra-rationale completeness criteria were inspired by the syntactic inferences that the Software Engineering Using Rationale (SEURAT) tool supports (Burge and D. C. Brown, 2008a; Burge and D. C. Brown, 2008b; Malloy and Burge, 2016). SEURAT produces warnings if it detects incompleteness, e. g., if an alternative has more pro-arguments than the decision.

In the example in Figure 6.5, the criteria of the intra-rationale completeness are evaluated as follows, where ✗ indicates that the rationale documentation is incomplete according to the criterion: 1) The decision problem is solved. ✓ 2) The decision problem for the decision is documented. ✓ 3) At least one alternative is documented for the decision problem. ✓ 4) At least one pro-argument is documented for the decision. ✗ 5) At least one con-argument is documented for the alternative. ✓ 6) The decision has more pro-arguments than its alternatives. ✗

Rationale Coverage

Coverage metrics are commonly used for quality assessment. For instance, *test coverage* measures the percentage that code or requirements are tested, and *clone coverage* measures duplication of source code (Wagner et al., 2015; Martinez-Fernandez et al., 2018). We introduce the *rationale coverage* to measure the amount of documented rationale traceable from a software artifact, e. g., from a feature or a code file, within a certain link distance (number of hops). The rationale coverage supports the completeness, accessibility, and traceability of the knowledge documentation. In particular, the *decision coverage* is a type of rationale coverage that measures the number of traceable decisions. The *issue coverage* measures the number of traceable issues.

In the example in Figure 6.5, the decision coverage of the *image matching* feature is one, since the decision *Implement image matching based on zero normalized cross correlation!* is documented in the link distance of three from the feature. The issue coverage is also one.

6.2.2. Definition of Done for Knowledge Documentation

The *definition of done* defines the criteria developers need to fulfill so that a work item, feature, or sprint is finished (Kopczyńska et al., 2022). ConRat introduces a definition of done regarding the rationale documentation. Table 6.2 shows the definition-of-done checklist and its intended influence on the quality requirements for the rationale documentation. The checklist is designed to cover all quality requirements: The development team fulfills the definition of done if the requirements are met. As introduced in Section 5.2.3, the high quality is also supported by the ConDec plug-ins, e. g., through knowledge visualization, nudging, and recommendations.

Table 6.2.: Influence of the definition of done on quality requirements for the rationale documentation. Criteria 1 – 4 demand documentation properties, criterion 5 is a task to be performed, and criterion 6 defines exclusion from checking.

	Quality Requirements for Documentation according to Zhi et al. (2015)					
Criteria of Definition of Done	Accessibility, Traceability	Completeness	Consistency, Correctness, Up-to-Dateness	Uniqueness	Information Organization, Format	Spelling, Grammar, Readability, Accuracy
1) intra-rationale completeness	✗	✓	✗	✗	✗	✗
2) decision coverage	✓	✓	✗	✗	✗	✗
3) phrasing of rationale elements	✗	✗	✗	✗	✓	✓
4) assignment to decision types	✓	✗	✗	✗	✗	✗
5) review of rationale documentation	✓	✓	✓	✓	✓	✓
6) exclusion criteria: test code, small files	✗	✗	✗	✗	✗	✗

In ConRat, the criteria of the definition of done are: 1) The *intra-rationale completeness* is fulfilled, i. e., decisions are completely documented. 2) Features and code files are covered with a certain number of decisions, e. g., one decision in a trace link distance of three (*decision coverage*). 3) Rationale elements are *phrased in a specific way*, e. g., issues as questions ending with a question mark and alternatives ending with an exclamation mark. This criterion supports a coherent writing style, i. e., format, and easy-to-understand rationale documentation, i. e., readability. 4) The *attributes of the rationale elements fulfill specific rules*, e. g., the assignment

to decision types as introduced in Section 2.2.4. This enables accessibility to—for example—all architecture or process decisions. 5) The definition of done also requires that the rationale documentation is *reviewed by at least one other developer who assesses the semantic content.* The reviewer checks whether the knowledge documentation fulfills all quality requirements for documentation. 6) Finally, *rules can exclude artifacts from definition-of-done checking.* For example, test and small code files with only a few lines do not need to fulfill the decision coverage.

The development team can tailor the definition of done to their needs. For instance, they can decide which intra-rationale completeness criteria they require for their rationale documentation, the minimum number of decisions, and the maximal link distance for the decision coverage.

In the example in Figure 6.5, the checking of the criteria 1)–6) described above leads to the following result: 1) The *intra-rationale completeness* is violated due to the criteria *at least one pro-argument is documented for the decision* and *decision has more pro-arguments than its alternatives.* 2) The definition of done requires at least one decision within a link distance of three from features and code. The *decision coverage* is fulfilled. 3) The rationale elements are phrased correctly. 4) The criterion is fulfilled if a *decision type* is assigned, here, design decision. 5) A *reviewer must check the rationale documentation* before integrating the feature branch. Thus, the criterion that requires a review is fulfilled. 6) The code file in Figure 6.5 is included in definition-of-done checking because it is no test code file and has enough lines of code.

6.2.3. Rationale Backlog

The ConRat life cycle model is an *issue-based life cycle model* (Bruegge and Dutoit, 2010). In an issue-based life cycle model, all issues are stored in an *issue base* accessible to the project participants. The issue base contains both unsolved and solved issues. The *rationale backlog* only includes unsolved issues, similar to the notation of a product and sprint backlog. The rationale backlog can be a part of the product and sprint backlog. The number of unsolved issues in the backlog can be used to assess the status of the project or sprint: The fewer unsolved issues, the closer the project or sprint to being finished.

In ConRat, the rationale backlog contains *unsolved issues*, *challenged decisions* (Figure 6.3), and other knowledge elements that *do not fulfill the definition of done for the documentation* (e. g., because the decision coverage is too low). By listing all unsolved issues and challenged decisions, the rationale backlog serves as an agenda for what the development team must discuss and decide collaboratively. It also helps to improve documentation quality.

The rationale backlog was inspired by the literature: Yang et al. (2019) surveyed agile practices that can be integrated into architectural assumption management, which is related to rationale management: Backlog, iterative and incremental development, refactoring, continuous integration, effective communication, and just enough work. The idea of managing a *decision backlog* was suggested by Hofmeister et al. (2007), Dingsøyr and van Vliet (2009), Hoorn et al. (2011), and Zimmermann et al. (2015). The SEURAT tool by Burge and D. C. Brown (2008a) and Burge and D. C. Brown (2008b) shows errors and warnings about the rationale in a *rationale task list.*

In the example in Figure 6.5, the issue *How to detect homologous points on stereo images?* does not appear in the rationale backlog because it is solved. However, the rationale backlog contains other unsolved issues, e. g., *How to import radar images?*, *How to bring radar images to the correct location?*, *How to get the incidence angle for every image pixel?*, *How to calculate the relative height of an object using incidence angles?*, and *Which digital terrain model to subtract from the digital surface model?* The rationale backlog also contains the decision *Implement image matching based on zero normalized cross correlation!* because there is no pro-argument for it, which is a criterion to assess the intra-rationale completeness (if the definition of done is not tailored to exclude this criterion).

6.2.4. Overview of a Life Cycle Model Extended with ConRat

This section explains the integration of ConRat into the Rugby life cycle model. Life cycle models can be represented with the same techniques used for modeling software systems (Bruegge and Dutoit, 2010). Figure 6.6 shows a *functional model* that describes the tasks by the roles using a UML use case diagram. The outer system models Rugby and the inner system models the extension through ConRat. The ConRat activities are collaborative, incremental, and rational decision making, documentation, exploitation, and quality assurance of decision knowledge. Besides, Figure 6.6 includes the definition of done and rationale backlog. ConRat introduces the role of the *rationale manager*, responsible for rationale quality management and dissemination.

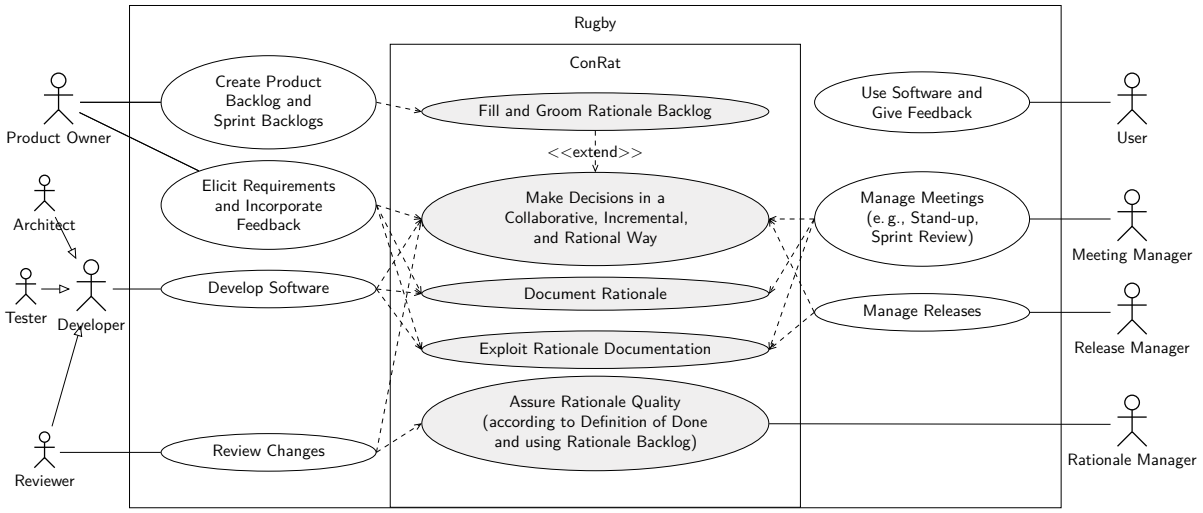

Figure 6.6.: Rugby workflows and ConRat activities (UML use case diagram). The workflows include the ConRat activities (stereotypes are omitted for readability).

Figure 6.7 shows an *object model* as a UML class diagram with classes and associations involved in the ConRat life cycle model extension. Figure 6.7 is based on the notation of the ISO/IEC/IEEE 24774 (2021) standard but adapted to the terms used in the description of Rugby by Krusche (2016) and in the *Unified Process* (Jacobson et al., 1998). A Life Cycle Model consists of Workflows that the CSE Roles perform in parallel. The workflows describe the tasks by the CSE roles. The workflows are performed during consecutive Sprints, which can also be called cycles as done in the Unified Process. Each workflow contains activities modeled by the Activity class. Activities are also referred to as sub-tasks. The activities can be refined into further (sub-)activities or into Actions. The refinement is modeled using the composite design pattern with the abstract superclass *Practice* in Figure 6.7. The actions produce or consume the Knowledge Elements of the ConRat knowledge model (Figure 6.1).

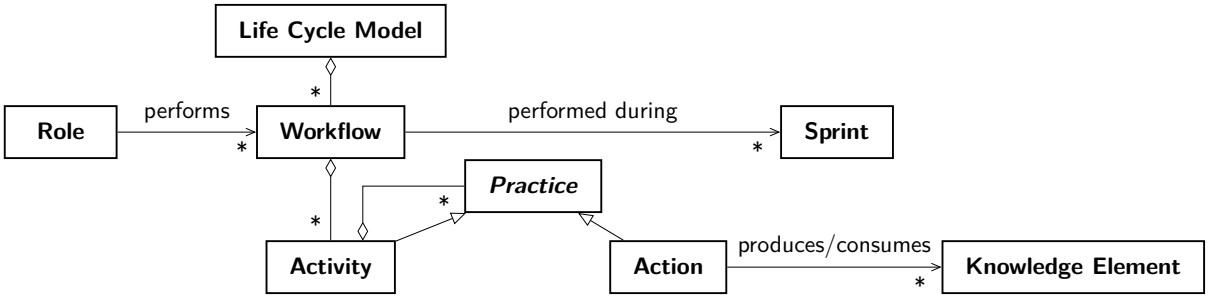

Figure 6.7.: Classes and associations involved in life cycle modeling (UML class diagram).

Figure 6.8 depicts an excerpt of a respective UML object diagram with eight parallel workflows and two hierarchical levels of rationale management activities. Roles, sprints, activities, and actions of Rugby, such as Design, Implement, Test, and Check in Changes to Branch, as well as knowledge elements and other involved entities, such as backlogs, are omitted for simplification. The gray-colored objects are added through ConRat.

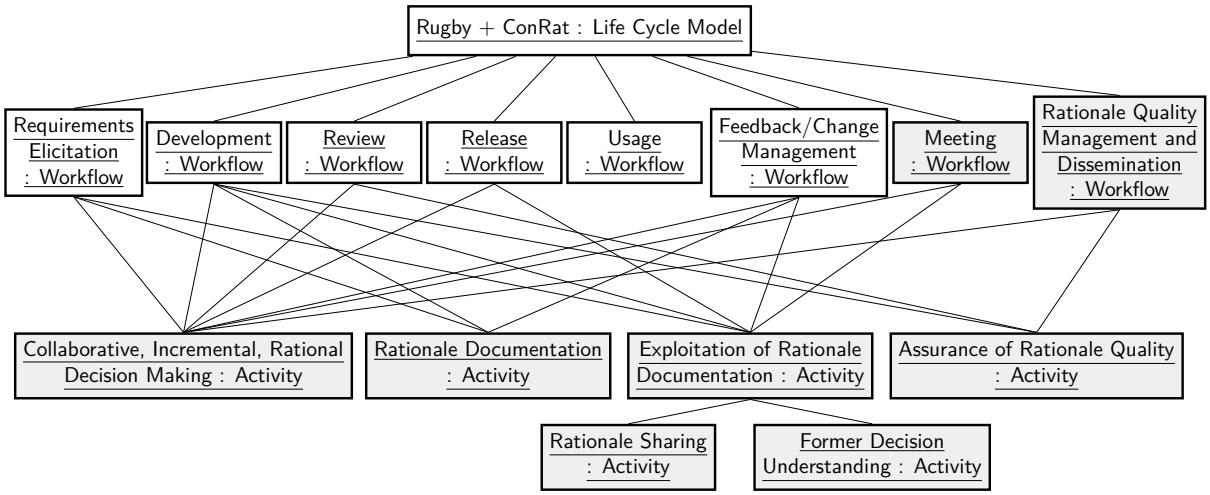

Figure 6.8.: Instances and associations involved in the extended Rugby life cycle model (UML object diagram). The diagram only shows activities belonging to ConRat.

Figure 6.9 shows a *dynamic model* of the life cycle including (activities) and (actions), the [entities] produced and consumed, as well as incoming and outgoing change events. The Rugby life cycle model and its parallel workflows provide the basis (Section 2.1.3). ConRat adds the rationale management activities, including the definition of done and the rationale backlog. The rationale management-related additions are indicated in the following with brown color.

During *sprint 0* (Figure 6.9-①), the development team fills the *rationale backlog* with unsolved issues. Besides, the development team makes high-level decisions. The team captures and discusses these decisions along with the respective issue, alternatives, pro-, and con-arguments, i. e., with explicit rationale. The development team also *fills the rationale backlog with unsolved issues when creating the sprint backlog* and continuously during the sprint (Figure 6.9-②). While they work on the software features (Figure 6.9-③), the team makes decisions by solving issues in the rationale backlog. They explicitly *capture rationale elements directly in their development tools*. Besides, they *exploit the knowledge documentation during development*, e. g., by reflecting on the decisions made in the past. The team discusses the unsolved issues in the rationale backlog and recently made decisions in the daily *stand-up meetings* (Figure 6.9-④). At the end of the sprint, when the *development is done*, the team must ensure that the *rationale documentation is of high quality* (Figure 6.9-⑤). That means the rationale backlog must only include unsolved issues but no other knowledge elements violating the definition of done. In ConRat, a product increment goes along with an increment of the (explicit and formalized) rationale documentation (Figure 6.9-⑥). The team also reflects on the rationale documentation of the sprint in the *sprint review meeting* (Figure 6.9-⑦). *Feedback reports* can trigger the development team to *revise the rationale backlog* (Figure 6.9-⑧). The revision means adding new issues and *updating the rationale documentation*, e. g., by rejecting a former decision and linking it to the new one.

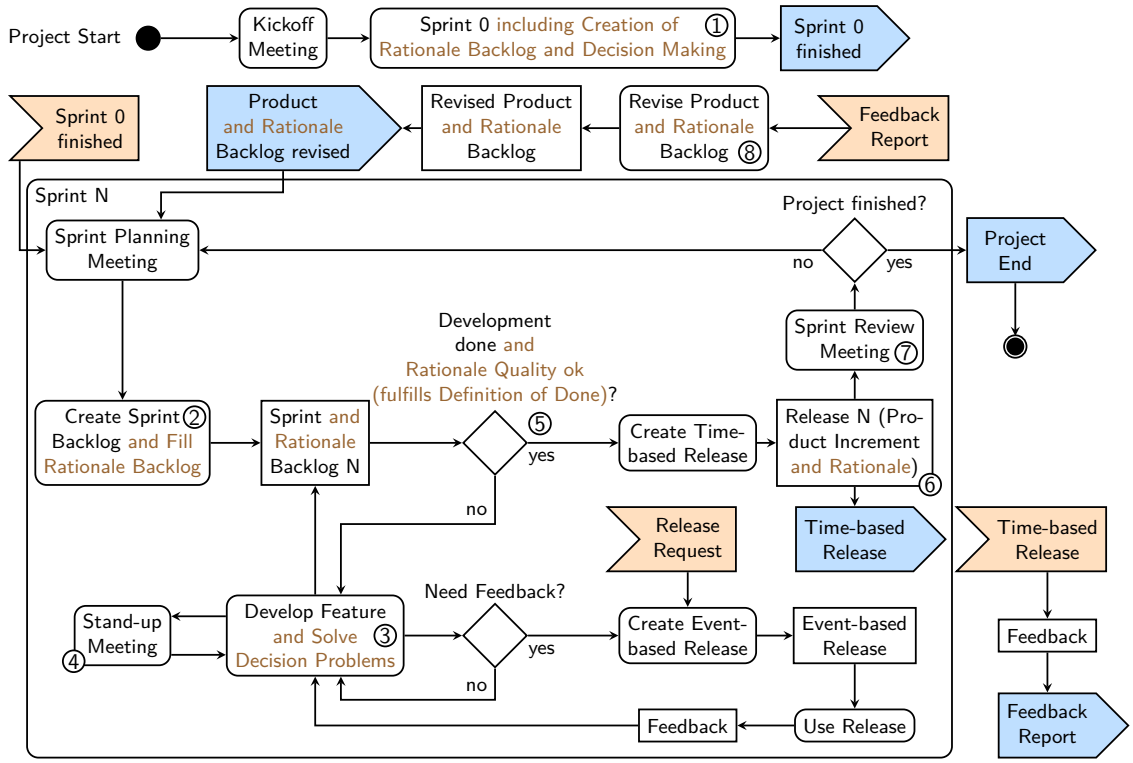

Figure 6.9.: Dynamic view of the Rugby life cycle model extended with rationale management (UML activity diagram).

6.2.5. Parallel Workflows: Roles and Their Tasks

This section describes seven parallel workflows tailored or added by ConRat. ConRat tailors the Rugby workflows except for the usage workflow (Figure 2.2) because users do not work with the rationale documentation when using the software. ConRat adds collaborative, incremental, and rational decision making, documentation, exploitation, and quality assurance of decision knowledge. ConRat also adds two new workflows for conducting meetings and for the *rationale manager* role. The roles are shown in the top right corner of the workflows. The section explains the workflows using the demonstration project introduced in Section 6.1.3.

In the *requirements elicitation workflow* (Figure 6.10), the *requirements engineer* or *product owner* adds and prioritizes a new backlog item, captures acceptance criteria, and rationale. In the project in Section 6.1.3, the requirements engineer creates the *image matching* feature task. The requirements engineer captures the following rationale based on informal discussions: *How to detect homologous points on stereo images?* with the alternatives *Implement image matching based on zero normalized cross correlation!* and *Use OpenCV library!*

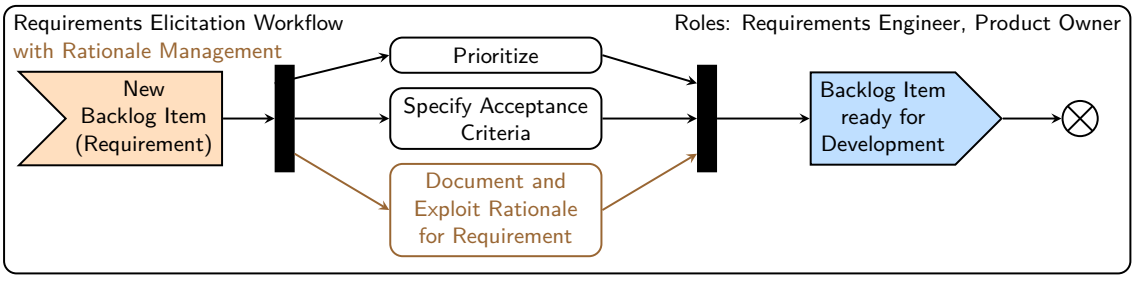

Figure 6.10.: Requirements elicitation workflow with explicit rationale management.

The end of the *requirements elicitation workflow* provides the event *requirement ready for development*, which starts the *development workflow* (Figure 6.11). The roles involved are the *architect*, *developer*, and *tester*. They analyze the *requirement* (or a development task to *change existing functionality* or an *improvement request*) along with the documented rationale. Then, they design, implement, and test the requirement. During their tasks, they solve decision problems (collected in the rationale backlog) by making decisions and documenting further rationale. In the example, they make and document the decision *Implement image matching based on zero normalized cross correlation!* They work on a feature branch and check in their changes, including the changed and new rationale documentation to this branch. The development is an incremental, iterative process. If the developers want to get user feedback, they request a release. They request a merge if the requirement is realized, i. e., it fulfills all acceptance criteria, and if the rationale is documented and *fulfills the definition of done for the rationale documentation*.

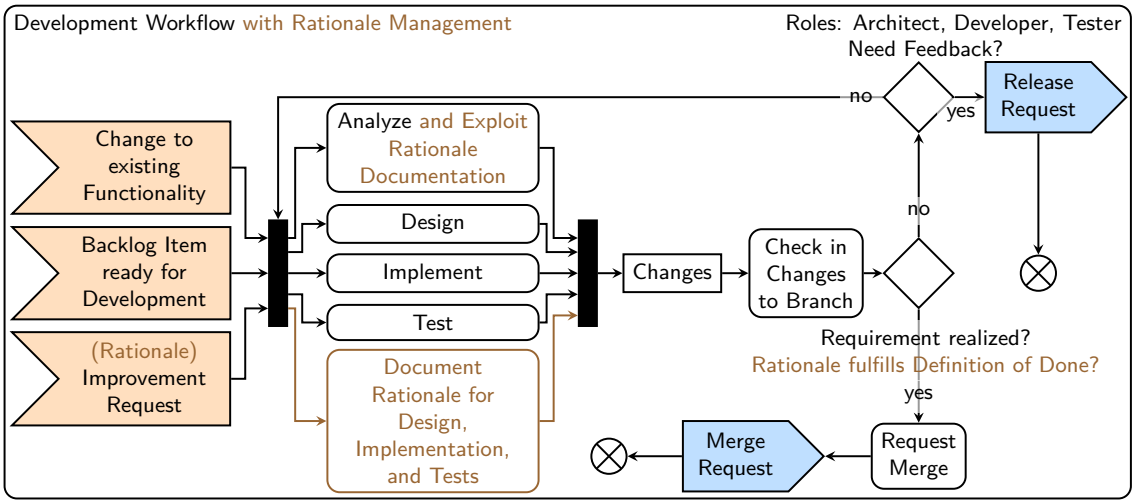

Figure 6.11.: Development workflow with explicit rationale management.

The *merge request*, or *pull request*, is the event that starts the *review workflow* (Figure 6.12). The responsible role is the *reviewer*, who is an experienced developer. In addition to reviewing the changes in the requirements specification, design, code, and tests, the reviewer inspects the rationale documentation. The requirement and related rationale documentation are finished if the reviewer accepts the changes. If the rationale documentation violates the definition of done, the reviewer requests improvements. In the example, the reviewer criticizes that the decision cannot be understood without arguments. That means the documentation is (intra-rationale) incomplete. Thus, the review workflow ends in a *rationale improvement request*.

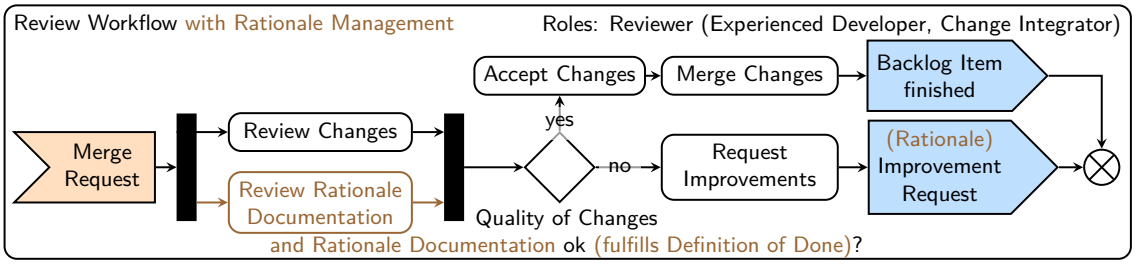

Figure 6.12.: Review workflow with explicit rationale management.

The rationale improvement request again starts the development workflow. The developers add pro- and con-arguments to the rationale documentation. For instance, they add the pro-argument *Works on gray-scale images* for the decision.

The *release request* is the event that starts the *release workflow* (Figure 6.13). A new, rationale management-related task of the release manager is to create release notes including relevant rationale to explain the changes made in the release. Not all users will be interested in the decisions made, however, the audience of release notes can differ with different interests (Klepper et al., 2016). In the example, the release manager creates a release together with the release notes that explain to the users that a new feature for image matching was implemented and that the developers made the decision to implement the zero normalized cross correlation algorithm.

Figure 6.13.: Release workflow with explicit rationale management.

The users use the software in the usage workflow (Figure 2.2) and provide user feedback. Feedback reports or change requests trigger the *feedback/change management workflow* (Figure 6.14). The responsible role, e. g., the product owner, analyzes the change request or feedback report and—if relevant—converts it to a backlog item (feature task) with the help of the rationale documentation. That means they exploit and update the rationale documentation.

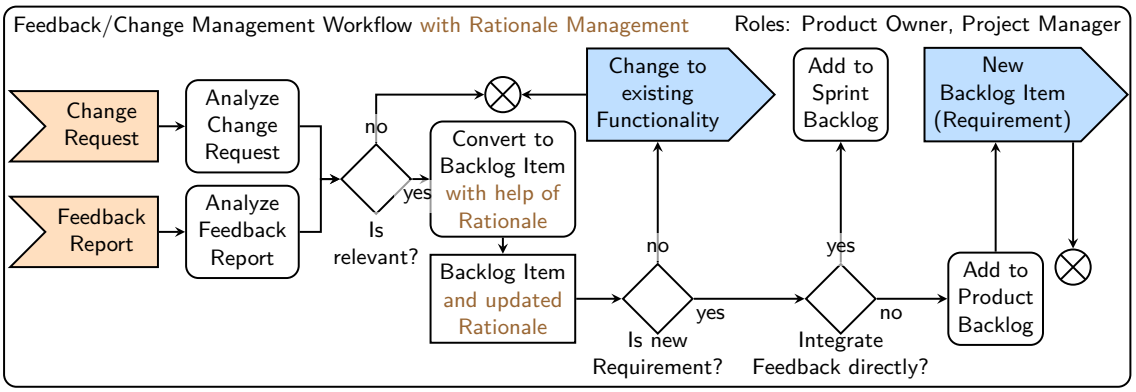

Figure 6.14.: Feedback/change management workflow with explicit rationale management.

In the example, the users provide explicit feedback stating that image-matching takes a long time to process new high-resolution satellite images. Figure 6.15 shows the updated knowledge documentation: Based on the user feedback report (Figure 6.15-①), a developer attaches the contra argument to the image-matching decision. After that, the developers create a new feature task to improve image matching (Figure 6.15-②), which again starts the development workflow. By reflecting on the former decision knowledge, the developers remember the alternative that they could use the OpenCV library, which offers image-matching methods. Therefore, they reject their former decision (Figure 6.15-③) and implement the new code.

To emphasize that decisions should be discussed and made collaboratively, ConRat adds the *meeting workflow* to Rugby (Figure 6.16). The meeting workflow is triggered whenever the project participants meet, for example, during stand-up meetings and sprint reviews. The *meeting manager* creates an agenda for the meeting with unsolved issues, i. e., questions to solve, and decisions that need to be shared and discussed. When the project participants conduct the meeting, they discuss the decision problems and decisions. In the example, the decision to *Implement image matching based on zero normalized cross correlation!* could also be made in a daily meeting. A *reporter* captures new rationale or improves the existing rationale documentation to be exploited in the requirements elicitation and development workflows.

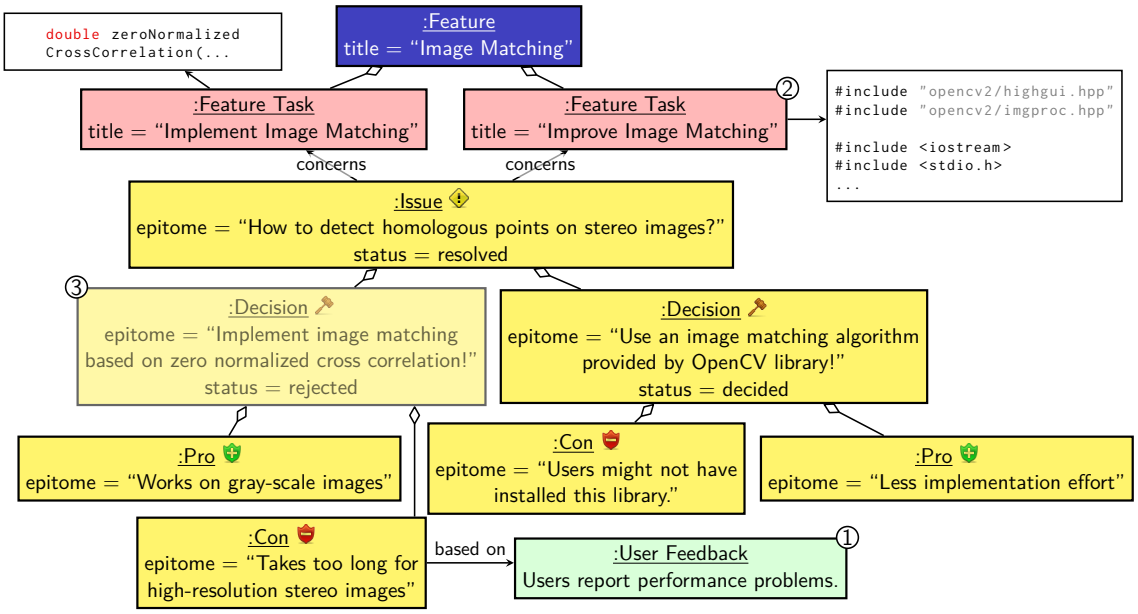

Figure 6.15.: Image-matching decision after employing usage knowledge (UML object diagram). The former decision is rejected and replaced with the former alternative.

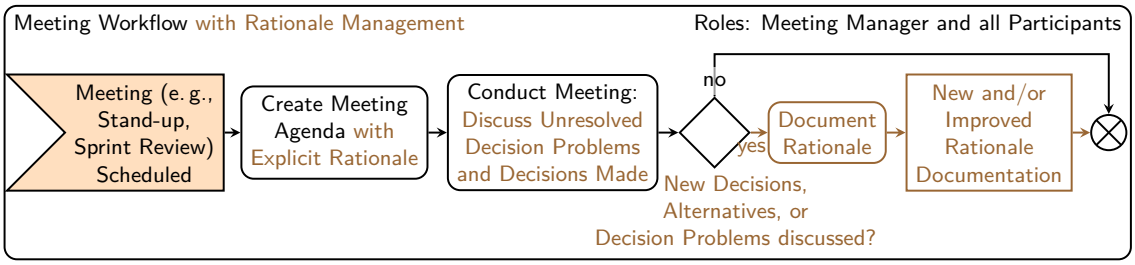

Figure 6.16.: Meeting workflow with explicit rationale management.

Quality management is the "process of establishing and directing a quality policy, quality objectives, quality planning, quality control, quality assurance, and quality improvement for an organization" (Spillner and Linz, 2021). The second workflow that ConRat adds to the parallel workflows of Rugby concerns quality management for the rationale documentation. It is called *rationale quality management and dissemination workflow* and performed by the *rationale manager* (Figure 6.17). The rationale manager is the role whose task is to set up rationale management, e. g., by defining the definition of done for the knowledge documentation. The rationale manager disseminates how to document and exploit decision knowledge (Chapter 12) and monitors the documentation, inspects it, and assures its high quality. For example, the rationale manager checks that the documented decision knowledge is complete, consistent, and linked to other knowledge. That means that this task of the rationale manager is similar to the reviewer's task in the review workflow. In contrast to the reviewer in the review workflow, the rationale manager inspects the rationale documentation from a global point of view for the entire project rather than regarding a single development task and merge request. The rationale manager can create a *rationale improvement request*. Besides, the rationale manager can schedule a meeting to discuss and disseminate how to document and exploit the rationale. For instance, the meeting can be a regular stand-up meeting or sprint review to improve the CSE process.

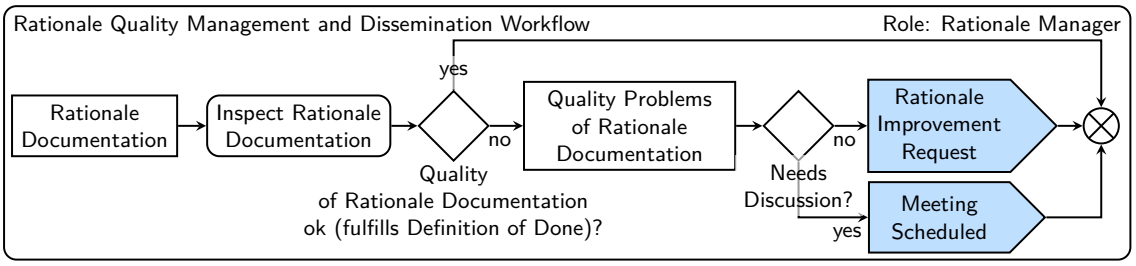

Figure 6.17.: Rationale quality management and dissemination workflow.

In summary, all of the roles contribute to the evolution of a software system by making decisions, e. g., on requirements, design, tests, or the process, and they reflect on decisions already made. Thus, they produce and consume decision knowledge. In particular, during ConRat, they document their decision knowledge and exploit the documented knowledge. For simplification, the thesis uses developers as examples of the different roles.

6.2.6. Starting and Finishing CSE Practices

CSE involves implementing and delivering many small increments. The practices advancing these increments indicate that developers *start* or *finish* work (Section 6.1.2, Figure 6.4). Table 6.3 lists starting and finishing practices that developers regularly perform.

Table 6.3.: CSE practices of different granularity. After the *starting practices*, developers exploit the knowledge documentation and *consider consistency* of new decisions. Before the *finishing practices*, developers *make tacit decisions explicit* and *package the distributed informal decision knowledge*, i. e., formalize and integrate it into the knowledge graph.

Starting Practice	**Finishing Practice**	**ConRat Workflow**
Start sprint (event: e. g., sprint 0 finished)	Finish sprint, sprint review, create time-based release	Overall life cycle model (Figure 6.9)
Schedule and start conducting a meeting	Post-process meeting by documenting rationale	Meeting workflow (Figure 6.16)
Specify new feature (event: new backlog item)	Finish specification of new feature (event: backlog item ready for development)	Requirements elicitation workflow (Figure 6.10)
Start feature task (events: backlog item ready for development, change to existing functionality, improvement request)	Finish feature task	Development workflow (Figure 6.11) and review workflow (Figure 6.12)
Create feature branch (action: check in changes to branch)	Accept merge request, merge branches	Development workflow (Figure 6.11) and review workflow (Figure 6.12)
Start working on code	Commit code (activity: check in changes to branch)	Development workflow (Figure 6.11)
Start creating release (event: release request)	Finish creation of release and release notes with explicit rationale	Release workflow (Figure 6.13)
Start handling change request or feedback report	Finish handling change request or feedback report with feature task and rationale update	Feedback/change management workflow (Figure 6.14)

The starting and finishing practices are part of the life cycle model and its parallel workflows. For instance, short-cycled CSE practices that indicate *start* are opening a feature task and creating a feature branch. Practices that indicate *finish* are to commit code, merge a feature branch, or close a feature task. When developers perform a *starting practice*, they use and build upon the existing decision knowledge. They make new decisions and must ensure that they are consistent with the former ones, i. e., they *consider consistency between old and new decisions.* Before performing a *finishing practice*, developers might have made important decisions. During ConRat, the developers document the decision knowledge explicitly and formally before the finishing practice. The developers must ensure the decision knowledge is formalized, integrated into the knowledge graph, and consistent with other documented knowledge. The developers' work needs to fulfill the definition of done to ensure the high quality of the decision knowledge documentation. The starting and finishing practices help to break down rationale management into small, manageable pieces to avoid *big-bang* documentation.

In the radargrammetry project, another open issue in the rationale backlog is the following: *How to bring radar images to the correct location?* Radar images need to be oriented before image matching can be performed. The developers specify a feature called *image orientation*, i. e., perform a starting practice. They solve the issue by deciding to *Use an image orientation algorithm provided by the OpenCV library!* The new decision is consistent with the former decision to apply the OpenCV library for image matching (Figure 6.15). The developers document the decision and related rationale before they finish the feature.

6.3. Conclusion

This chapter presented the ConRat life cycle model extension that integrates explicit rationale management into CSE. First, it presented a knowledge model that formalizes decision knowledge and links it to common CSE artifacts, in particular, features, work items, commits, merge requests, and code. It introduced three types of states, i. e., a decision-making state, a documentation state, and an implementation state. Subsequently, the chapter presented the rationale management extension of the Rugby CSE life cycle model. It introduced metrics, a definition of done for the rationale documentation, and a rationale backlog as essential concepts. Finally, this chapter described short-cycled starting and finishing practices intertwined with rationale management.

The chapter made four contributions: First, it presented the ConRat knowledge model as the first model combining decision knowledge with CSE artifacts. The model lays the basis to *support the high amount of distributed knowledge* in CSE. Decision knowledge can be documented and exploited from different artifacts using the model. Second, the chapter presented the *first life cycle model with explicit rationale management.* Including rationale management activities is a contribution to life cycle modeling. The chapter illustrated how life cycle models are tailored to include ConRat. ConRat could also extend other life cycle models analogously, such as Scrum. Development teams can tailor and extend the life cycle model presented in this chapter to their needs. Third, the chapter presented a *treatment for the problems of intrusiveness and effort and low documentation quality.* The treatment consists of integrating rationale management activities into existing workflows and concepts to operationalize the documentation quality, including metrics, the definition of done, and the rationale backlog. There is a trade-off between a lightweight life cycle and ensuring high quality. However, the continuous reflection of the rationale documentation and the integration of quality checking based on the definition of done into the various finishing practices help to keep the effort developers need to spend on quality-improvement tasks low. Fourth, the concepts and workflows described in this chapter are the *basis for automation and tool support.* The following chapter will describe the ConDec plug-ins that support the rationale management activities through views and features.

Chapter 7

Supporting Continuous Rationale Management with ConDec

"Any sufficiently advanced technology is indistinguishable from magic."

—Clarke, 1962

This chapter presents the Continuous Management of Decision Knowledge (ConDec) plug-ins that support continuous rationale management through views and features. While ConRat models the rationale management activities and concepts, ConDec is a model for tool support.

The sections of the chapter explain the entities of the ConDec Plug-Ins in Figure 5.4. Section 7.1 presents requirements in the form of *task and support* descriptions and *functional models*. Section 7.2 describes the system and object design of ConDec. The following sections detail the ConDec views and features, from must-be features for rationale documentation and exploitation to more advanced features. Section 7.3 describes explicit decision knowledge documentation features. Section 7.4 and Section 7.5 present the visualization of the knowledge graph data structure and features to customize the views to specific purposes, e. g., through transitive linking for targeted access. Section 7.6 describes how ConDec motivates the developers to perform rationale management using nudging mechanisms and recommendation systems continuously. Section 7.7 presents the implementation of the rationale backlog. Section 7.8 describes the knowledge dashboard. Section 7.9 provides a feature to group and filter decisions by their types. Section 7.10 presents the support for meeting agendas and Section 7.11 presents the support for release notes. Section 7.12 describes the knowledge export feature. Section 7.13 discusses related work and Section 7.14 concludes this chapter. This chapter provides examples from the case study projects performed for the treatment validation (Part IV).

The ConDec plug-ins are open source and available in Appendix A.

7.1. Requirements

Continuous rationale management involves activities for collaborative, incremental, and rational decision making, documentation, exploitation, and quality assurance of decision knowledge (Section 6.2.4). This section details the rationale management activities and specifies tool support with *task and support* descriptions and functional models. The rationale management activities are *sub-tasks* of the CSE workflows. The sub-tasks can have *variants* (a, b, c, ...) representing another way to do the task. *Problems* (p, p1, p2, ...) are specified that need to be eliminated (Lauesen, 2002; Lauesen and Kuhail, 2012). The problems refine the three

rationale management problems of the thesis (Section 1.2). A *functional model* describes the system's functionality from the user's point of view (Bruegge and Dutoit, 2010). The specification includes activities mentioned in the rationale management literature (Section 1.1), practices by practitioners (Table 3.4), and the tailored Rugby workflows (Section 6.2.5). The tool support includes features requested by practitioners (Section 3.2.3) and rationale management approaches with classification and recommendation (Chapter 4).

Section 7.1.1 specifies the task and support for documenting rationale. Section 7.1.2 specifies the task and support for exploiting the rationale documentation. Section 7.1.3 specifies the task and support for collaborative, incremental, and rational decision making. Section 7.1.4 specifies the task and support for the quality assurance of the rationale documentation. Section 7.1.5 specifies the task and support for setting up rationale management.

7.1.1. Rationale Documentation

Figure 7.1 shows the functional model related to rationale documentation, referencing views and features of ConDec. Developers are an example of different roles because the documentation support is the same for all the roles contributing to the development, e. g., architects or testers.

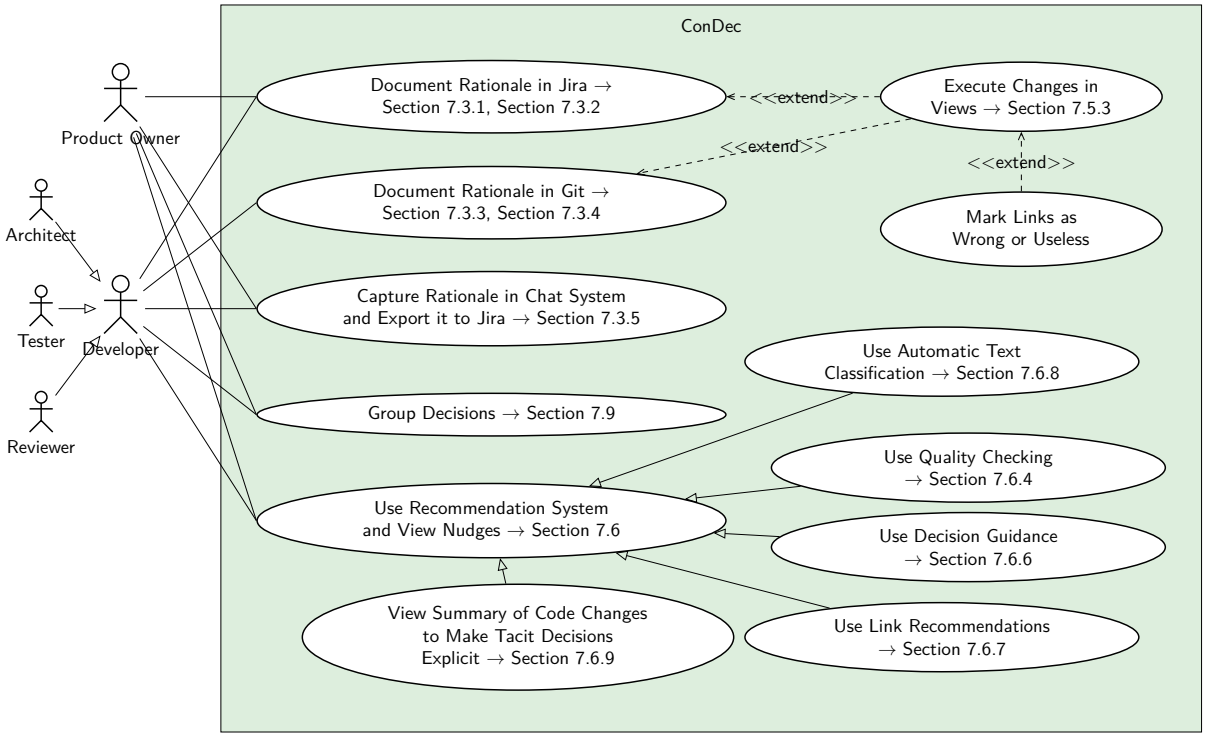

Figure 7.1.: Functional model of ConDec's support for documenting rationale (UML use case diagram).

Table 7.1 provides the task and support specification for rationale documentation. The system functions detail the support. Product owners (requirements engineers) elicit, document, change, and manage requirements for the software. Product owners and developers document decision knowledge in the issue tracking system Jira. They also discuss decision problems, capture decision knowledge, and collaboratively make decisions in chat messages. Developers implement the requirements in code files stored in the version control system git. Thereby, they document decision knowledge in commit messages and code comments. Product owners and developers group the decisions to enable targeted access to specific decisions. To support the documentation, they use recommendation systems and nudging mechanisms.

Table 7.1.: Task and support specification for documenting decision knowledge.

	Sub-task, Variants, and Problems	**ConDec Support**
	Document decision knowledge element with typical data like description (epitome), knowledge type, and status. Change or delete an existing element. Create or delete links between elements.	**Rationale documentation** (Section 7.3) and **change execution** (Section 7.5.3): Create decision knowledge element; Change knowledge element; Delete knowledge element; Link knowledge elements; Unlink knowledge elements
a	Document decision knowledge elements as entire Jira tickets.	**Rationale documentation as entire tickets** (Section 7.3.1)
b	Document decision knowledge elements in the text of Jira tickets.	**Rationale documentation in description or comments of tickets** (Section 7.3.2)
c	Capture decision knowledge elements in commit messages.	**Rationale documentation in commit messages** (Section 7.3.3): Post a git commit for a Jira ticket into the ticket comment and automatically add annotated decision knowledge from the commit message into the knowledge graph
d	Document decision knowledge elements in code comments.	**Rationale documentation in code comments** (Section 7.3.4): Automatically add code files from a git repository into the knowledge graph; Automatically add and link decision knowledge from comments in code files into the knowledge graph
e	Discuss decision problems and make decisions in chat.	**Rationale documentation in chat messages** (Section 7.3.5): Manually mark text as decision knowledge in Slack and export it to Jira
f	Group decisions to enable targeted access.	**Decision grouping** (Section 7.9)
p1	The decision knowledge documentation contains mistakes.	**Quality checking** (Section 7.6.4): Check the quality of knowledge documentation according to definition of done; Enforce definition of done fulfillment before finishing, e. g., of a work item or feature branch (**Merge check**) **Nudging** (Section 7.6): Show just-in-time prompts with recommendations when starting or finishing, show friction and ambient feedback nudges
p2	Manual documentation is not done (capture problem).	**Automatic text classification** (Section 7.6.8) **Decision guidance** (Section 7.6.6): Accept or discard solution recommendations, undo discard **Summarization of source code changes** (Section 7.6.9): Show summarized code changes to nudge the developers in retrospectively making tacit decisions explicit
p3	Links are not created between related knowledge elements.	**Link recommendation** (Section 7.6.7): Accept or discard link recommendations, undo discard
p4	The knowledge was already documented. Duplicated documentation can become inconsistent.	**Duplicate detection** (Section 7.6.7): Manually resolve duplicate or discard duplicate recommendation, undo discard
p5	The documentation is outdated.	**Change execution** (Section 7.5.3): Change an existing knowledge element instead of deleting it: Improve it by changing its description or state. A decision is set to the state *challenged* or *rejected* instead of deleting it.
p6	The code changes are tangled, i. e., are (partly) unrelated to what is written in the description of the development task, which hinders exploitation.	**Marking links as wrong or useless** (Section 7.5.3) Show probability of correctness of a link between a work item and a changed file (Section 7.6.9)

7.1.2. Exploitation of Rationale Documentation

Figure 7.2 shows the functional model related to knowledge exploitation, referencing views and features of ConDec. Developers are an example of different roles using exploitation support.

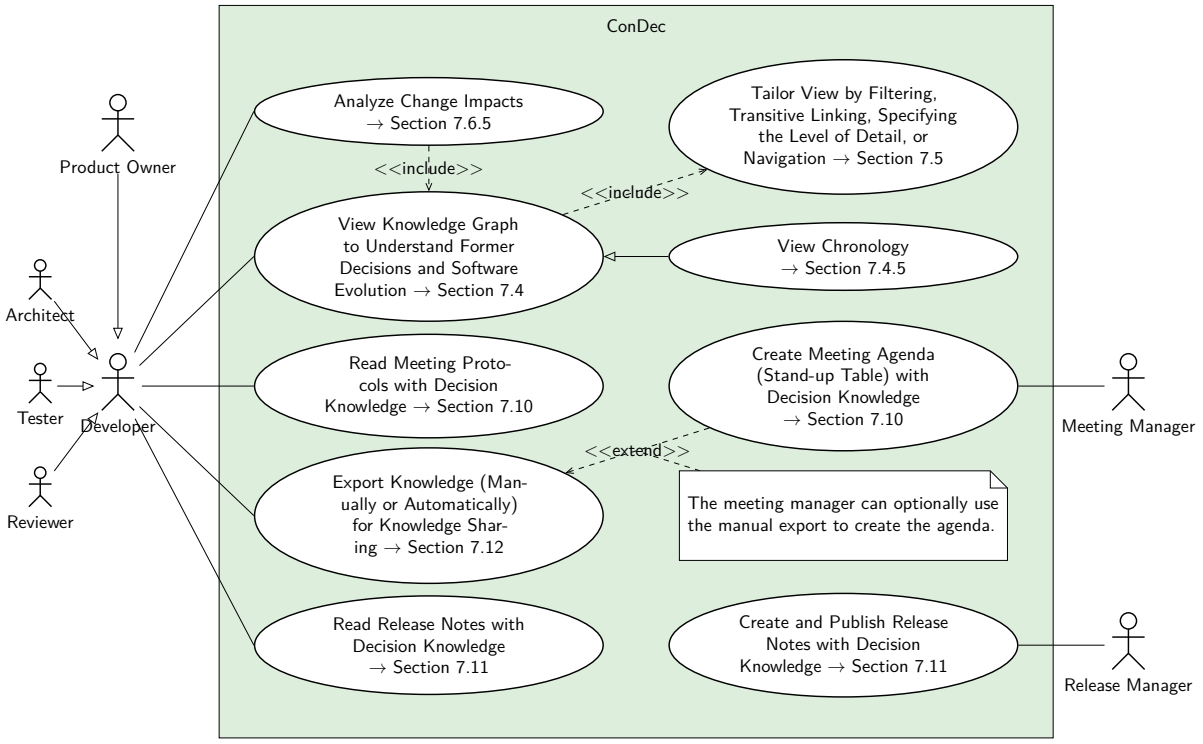

Figure 7.2.: Functional model of ConDec's support for exploiting rationale documentation (UML use case diagram).

Table 7.2 provides the task and support specification for exploiting the rationale documentation. It distinguishes three sub-tasks. Developers perform the first sub-task. The second and third sub-tasks are performed by the meeting manager and release manager, respectively. When developers start working on a project, they must understand the decisions made in the past to make new decisions consistently. They must know the major decisions that shaped the evolution of the software and the related requirements, rationale, and code. To understand former decisions and software evolution, developers use the views on the knowledge graph, particularly the chronology view. They read meeting protocols and release notes, including decision knowledge. Developers analyze the impact of the changes to estimate the effort required to accomplish the change, make decisions consistent with former decisions, and keep the documented knowledge consistent. Developers inform other stakeholders about decisions and the problems to be solved for knowledge sharing and to support collaborative decision making. They use ConDec's functionality for manual or automatic knowledge export. The release manager creates and publishes release notes at the end of a sprint or for an event-based release, including decision knowledge relevant to the release. The meeting manager creates an agenda, including a stand-up table with relevant decision knowledge to prepare for the meeting.

Table 7.2.: Task and support specification for the exploitation of the knowledge documentation.

	Sub-task, Variants, and Problems	ConDec Support
1	Exploit the knowledge documentation, for example, to understand former decisions and evolution history for a project.	**Knowledge graph views** (Section 7.4): Show evolution of knowledge over time as chronology (Section 7.4.5) **Features for view tailoring** (Section 7.5): Filter knowledge graph; Exploit transitive links; Specify the level of detail; Navigate between elements
1a	Analyze the impact that it has to change a knowledge element.	**Change impact analysis** (Section 7.6.5): Color nodes in the views on the knowledge graph according to the likelihood that the change impacts them; Explain impact value
1b	Read meeting protocols, including explicit decision knowledge.	**Stand-up table with decision knowledge** (Section 7.10)
1c	Read the release notes, including explicit decision knowledge.	**Release notes with decision knowledge** (Section 7.11): Show release notes; Filter release notes
1d	Share decision knowledge. Inform other stakeholders about recently made decisions and open issues to be solved.	**Knowledge export** (Section 7.12) **Rationale information channel in chat system**: Post open issues and recently made decisions into chat
2	Create a meeting agenda for a stand-up meeting, sprint review, or another meeting with recently made decisions and issues to be solved.	**Stand-up table with decision knowledge** (Section 7.10): Import decision knowledge into a meeting agenda as a stand-up table
3	Create and publish release notes at the end of a sprint	**Release notes with decision knowledge** (Section 7.11)

7.1.3. Decision Making

Documenting and exploiting the rationale described in the previous sections benefits decision making because the decision knowledge is made explicit, which helps its reflection and discussion. This section specifies additional support for collaborative, incremental, and rational decision making. Figure 7.3 shows the functional model related to decision making, referencing views and features of ConDec. Developers are an example of different roles using decision-making support. Table 7.3 provides the task and support specification for collaborative, incremental, and rational decision making. While eliciting and implementing requirements, the developers solve decision problems by considering solution options, weighing them against criteria, and choosing an option as the decision. The developers use non-functional requirements as decision-making criteria. *Non-functional requirements* describe user-level requirements not directly related to functionality (Bruegge and Dutoit, 2010). A *quality requirement* demands that a quality attribute is present in the software. Quality attributes include maintainability, security, performance efficiency, compatibility, usability, reliability, or portability (ISO/IEC 25010, 2011). A *constraint* limits the solution space beyond what is necessary to fulfill the functional and quality requirements (Pohl and Rupp, 2016). Constraints are also called *pseudo requirements* and can relate to implementation, interface, operations, packaging, and legal aspects (Bruegge and Dutoit, 2010). Non-functional requirements include quality requirements and constraints. To support the decision making, developers use the decision guidance recommendation system. The developers discuss open issues and challenged decisions listed in the rationale backlog. During meetings, the developers discuss open issues and recently made decisions using the stand-up table. The meeting manager guides the discussion and is also the reporter who logs the meeting minutes and captures the decision knowledge. ConDec provides nudging mechanisms to indicate open issues, challenged decisions, and recommendations to accept or discard.

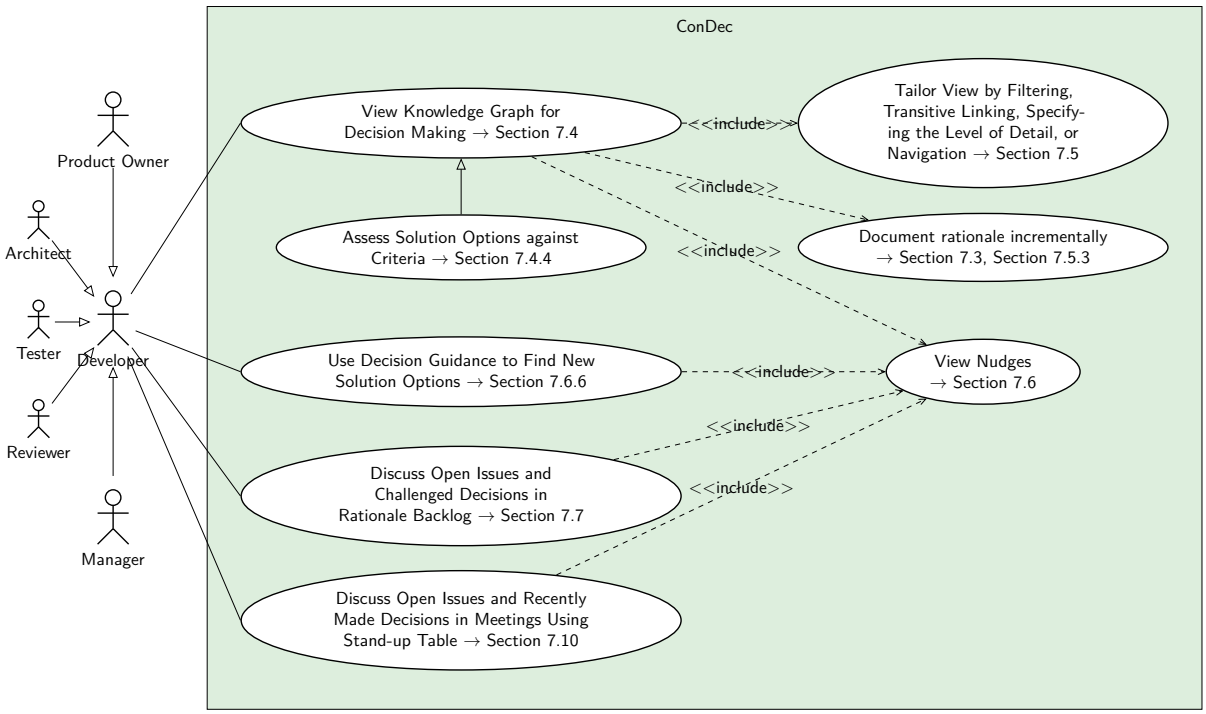

Figure 7.3.: Functional model of ConDec's support for decision making (UML use case diagram).

Table 7.3.: Task and support specification for collaborative, incremental, and rational decision making.

	Sub-task, Variants, and Problems	**ConDec Support**
	Solve a decision problem by considering possible solution options, weighing them against criteria, and choosing the optimal option as the decision (rational decision making). Capture the decision knowledge incrementally.	**Knowledge graph views** (Section 7.4) and **features for tailoring** (Section 7.5): Show criteria matrix (Section 7.4.4) **Rationale documentation** (Section 7.3) and **change execution** (Section 7.5.3): Incrementally document rationale **Nudging** (Section 7.6.2): Color unresolved decision problems
a	Already made decisions can be challenged. They need further discussion and might be rejected.	**Nudging** (Section 7.6.2): Color challenged decisions in the views on the knowledge graph
b	Decisions can be rejected, and alternatives can be discarded, which means that the developers no longer need to focus on them during decision making.	**Nudging** (Section 7.6.2): Gray out rejected decisions and discarded alternatives in the views on the knowledge graph
p1	Developers do not know which decision problems are unsolved and do not discuss solution options collaboratively.	**Rationale backlog** (Section 7.7): Show unsolved issues and challenged decisions for collaborative discussion **Stand-up table with decision knowledge** (Section 7.10): Show unsolved issues, challenged decisions, and recently made decisions in meeting agenda for collaborative discussion
p2	Developers do not know all the solution options and anchor on the first solution option that comes to mind. They do not make optimal decisions.	**Decision guidance** (Section 7.6.6): Recommend solution options for a decision problem (issue); Calculate a confidence score for recommended solution options and sort them according to this score; Accept recommended solution option; Discard/reject recommended solution option; Show all discarded recommendations; Undo the discarding of a recommended solution option **Nudging** (Section 7.6.2): Indicate recommendations

7.1.4. Quality Assurance

The documentation support in Section 7.1.1 introduced the recommendation systems and nudging mechanisms to support high-quality documentation during the development. This section introduces the support for the rationale manager and reviewer roles. Figure 7.4 shows the functional model related to quality assurance of rationale documentation, referencing views and features of ConDec. Table 7.4 provides the task and support specification for the quality assurance of the rationale documentation. The rationale manager and reviewers check and maintain the quality of the documented decision knowledge and other knowledge to be of high quality according to the definition of done. Reviewers conduct code reviews to check if the code

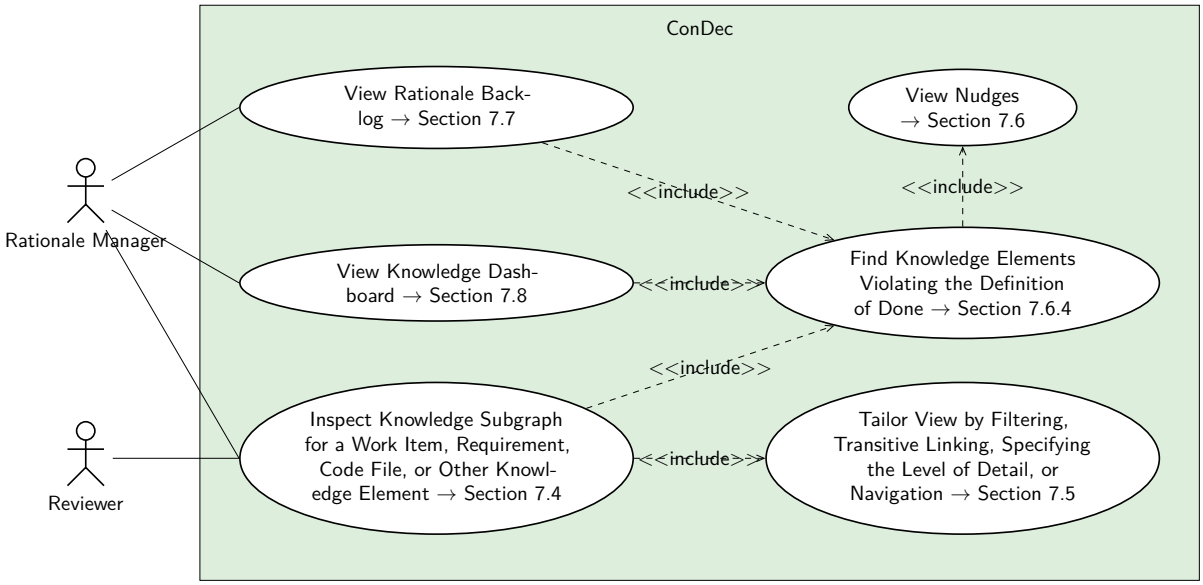

Figure 7.4.: Functional model of ConDec's support for the quality assurance of rationale documentation (UML use case diagram).

Table 7.4.: Task and support specification for the quality assurance of the documented knowledge.

	Sub-task, Variants, and Problems	**ConDec Support**
	Assure that developers correctly document rationale in the context of requirements, code, and other artifacts.	**Quality checking** (Section 7.6.4): Check the quality of knowledge documentation according to definition of done **Knowledge dashboard** (Section 7.8): Show general metrics, intra-rationale completeness, rationale coverage, metrics for the decision types, and feature branch-related metrics **Knowledge graph views** (Section 7.4)
a	Check a subset of the documented knowledge or documented for a specific knowledge element.	**Features for view tailoring** (Section 7.5): Filter knowledge graph; Exploit transitive links; Specify the level of detail; Navigate between elements; Filter dashboard (Section 7.8.6)
p	The decision knowledge documentation contains quality problems.	**Rationale backlog** (Section 7.7): Show requirements, code files, and rationale elements violating the definition of done **Nudging** (Section 7.6): Color knowledge elements violating the definition of done; Show just-in-time prompts with knowledge elements violating the definition of done **Integrated navigation** (Section 7.5.5, Section 7.8.6): Navigate from plots in the dashboard to knowledge element to improve the documentation; Create work item for improvement

and the knowledge documentation that another developer created for a specific development task or requirement are complete and consistent. The rationale manager inspects the rationale documentation from a global point of view for the entire project. The reviewer inspects the rationale documentation for a single development task, merge request, or requirement.

7.1.5. Setting Up Rationale Management

Table 7.5 provides the task and support specification, and Figure 7.5 shows the functional model for setting up rationale management, referencing configuration features of ConDec.

Table 7.5.: Task and support specification for setting up rationale management.

	Sub-task, Variants, and Problems	**ConDec Support**
	Define how developers should capture, manage, and exploit rationale.	Enable or disable the plug-in for a Jira project
a	Define which types of decision knowledge/rationale and which link/edge/relationship types should be documented by the developers.	Configure rationale model; Configure criteria in the criteria matrix/decision table (quality requirements or constraints)
b	Define where developers should document decision knowledge/rationale.	Documentation in Jira ticket description and comments is possible by default; Enable or disable whether decision knowledge elements can be documented as entire Jira tickets; Configure decision knowledge extraction from git (commit messages and code comments)
p1	Developers are reluctant to capture decision knowledge. The coverage of requirements and code with documented decision knowledge is low. Parts of the decision knowledge might be missing (e. g., only the issue is captured, not the decision).	**Configuration of quality checking** (Section 7.6.4, Figure 7.24): Configure definition of done for the knowledge documentation, including the configuration of nudging mechanisms; Enable or disable enforcement of rationale completeness in pull requests
p2	Developers implicitly capture decision knowledge in natural language text but do not explicitly mark it as such. The decision knowledge is hard to exploit when only captured implicitly.	**Configuration of automatic text classification** (Section 7.6.8): Enable or disable text classifier to identify decision knowledge elements in the text; Train classifier
p3	Developers might not know all solution options and anchor on the first solution option that comes to their mind. They might make sub-optimal decisions.	**Configuration of decision guidance** (Section 7.6.6): Configure solution option recommendation from external knowledge sources, including the configuration of nudging mechanisms
p4	Developers do not know the knowledge elements documented by others or in the past. Thus, they do not link new to existing elements or document duplicates.	**Configuration of link recommendation and duplicate detection** (Section 7.6.7): Configure linking support and duplicate detection, including the configuration of nudging mechanisms
p5	Developers are not aware of the impact of their changes and introduce inconsistency.	**Configuration of change impact analysis** (Section 7.6.5): Configure support for change impact analysis by setting default values for rules
p6	Developers do not know the decision knowledge documented by others and thus do not use it.	**Configuration of knowledge export** (Section 7.12): Configure automatic knowledge export to external system/stakeholders for knowledge sharing **Configuration of release notes with decision knowledge** (Section 7.11): Configure semi-automatic release notes creation

The rationale manager sets up rationale management for the project. For this purpose, the rationale manager defines the decision knowledge types, link types, documentation locations, the definition of done, the support through recommendation systems, and how the developers should share their knowledge, e. g., through automatic knowledge export and release notes.

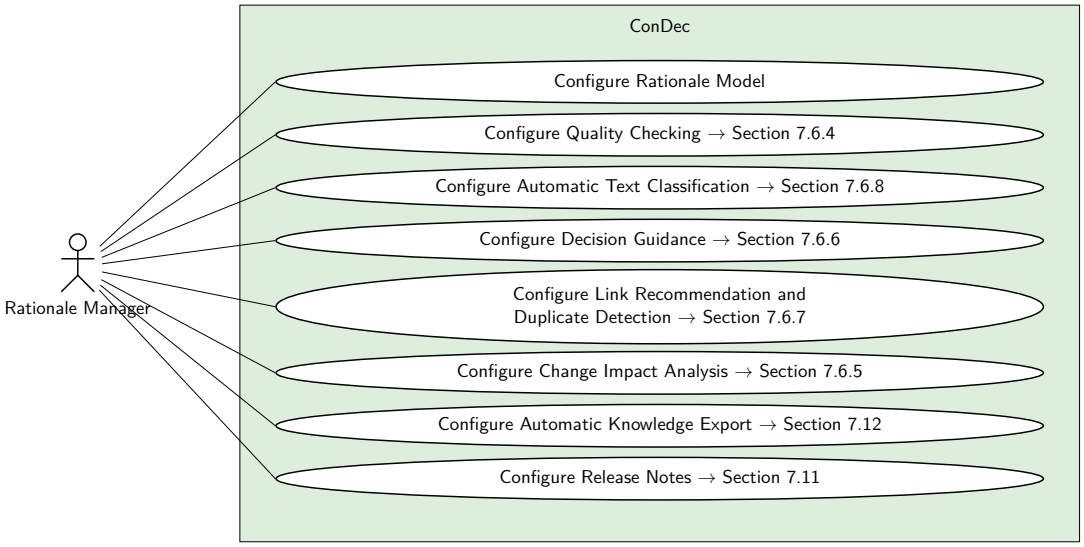

Figure 7.5.: Functional model of ConDec's support for setting up rationale management (UML use case diagram).

7.2. Design of ConDec

Software development activities include requirements elicitation, analysis, system design, object design, implementation, and testing (Bruegge and Dutoit, 2010). The ConRat knowledge model (Section 6.1), life cycle model (Section 6.2), and the functional models in the previous section are products of the requirements elicitation and analysis. This section provides an overview of the system and object design of ConDec. System design starts with defining *design goals*, which include performance, dependability, cost, maintenance, and end-user criteria (Bruegge and Dutoit, 2010). The technical design goal of the thesis is to support rationale management with low intrusiveness (Chapter 5), and it constrains ConDec's system design. From the system design perspective, design goals are to reduce the deployment cost, i. e., the cost of installing the system and training the users, and to achieve usability and high performance. ConDec is open source and must thus only integrate freely available, open-source components. The development and maintenance costs must be low since developers are involved for a limited time or voluntarily.

To address the design goals of low deployment cost and high usability (i. e., low intrusiveness), ConDec integrates into multiple standard development tools or systems rather than providing a standalone tool. Figure 7.6 shows a component diagram of the ConDec plug-ins. The ConDec plug-ins are subsystems of ConDec. They have interfaces to the underlying development tools or systems and each other. Every component represents one ConDec plug-in. ConDec consists of ConDec Jira as an extension for an issue tracking system, ConDec Bitbucket as an extension for a web-based git-client, ConDec Eclipse and ConDec VSCode as extensions for integrated development environments, ConDec Confluence as an extension for a wiki system, and ConDec Slack as an extension for a chat system. The *ConDec Integrated Development Environment* component is abstract and realized both through ConDec Eclipse and ConDec VSCode. Figure 7.6 includes the Git Version Control System component because developers can document decision knowledge

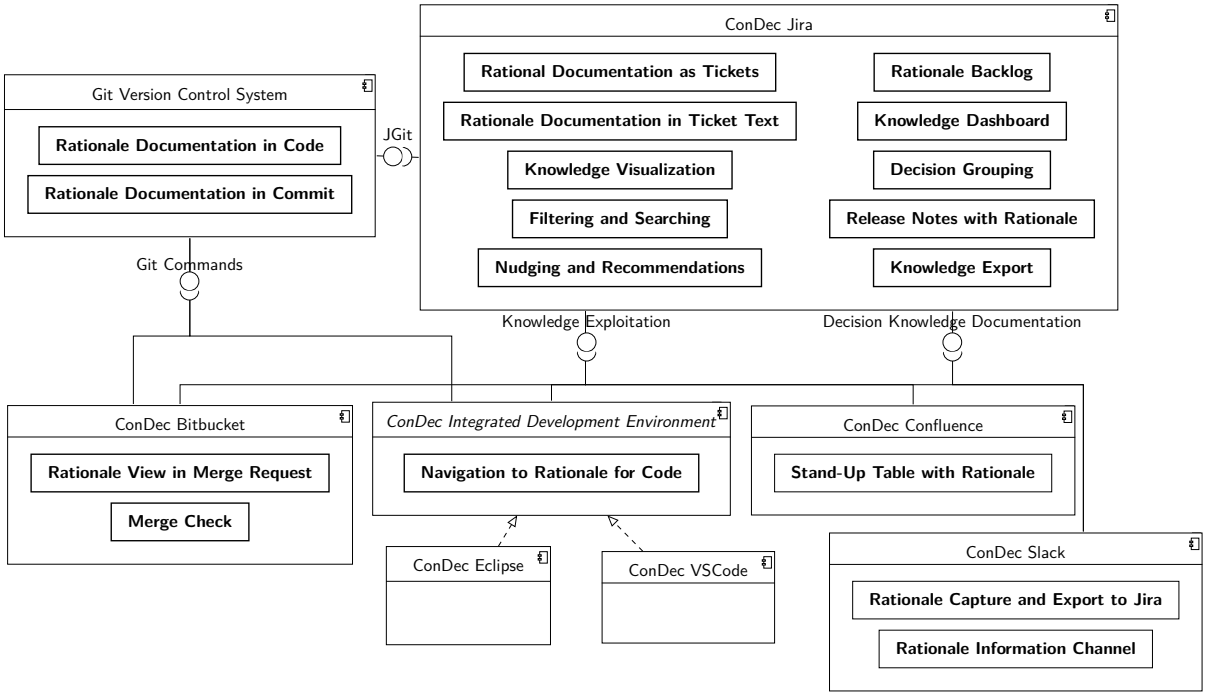

Figure 7.6.: ConDec plug-ins and the features they offer as classes (UML component diagram).

using every git client in commit messages and every editor in code comments. The ConDec Jira plug-in accesses git via the JGit library. The ConDec Bitbucket plug-in and the plug-ins for the integrated development environments also access git via Git Commands (Chacon and Straub, 2014). The classes in the components represent the features specified in Section 7.1.

The following paragraphs present issues and decisions related to system design, object design, and implementation. The remainder of this chapter also contains decisions for the design of particular features. The decisions in the thesis only provide an overview and often omit alternatives and justification for simplification. The decision knowledge documentation of ConDec is available in Appendix A.

Which development systems and tools should ConDec integrate into? Deciding which tools to integrate ConDec into is a *hardware/software mapping issue* (Bruegge and Dutoit, 2010). As already answered above, ConDec integrates into Jira, Bitbucket, Eclipse, VSCode, Confluence, and Slack, using git as the version control system. ConDec integrates into these tools because they are standard development tools practitioners use (Chapter 3). However, other development tools exist that ConDec should extend in the future, e. g., the IntelliJ development environment or GitHub and GitLab as web-based git-clients.

How to distribute the views and features among ConDec plug-ins? Section 7.1 derived features of ConDec from the rationale management activities and problems. During system design, it needs to be decided which of the ConDec plug-ins offers which feature. A trade-off exists between the design goals of high usability and low development and maintenance costs. From the ConDec users' view, accessing the same features directly within their development tools would be usable. For example, they could directly use ConDec's comprehensive knowledge visualization in the wiki system, integrated development environment, and issue tracking system. However, implementing the features redundantly in multiple tools is costly. ConDec avoids redundancy and integrates most features into the issue tracking system through ConDec Jira (Figure 7.6).

Some features are integrated into the other plug-ins because they would not make sense in the issue tracking system. For instance, the ConDec Confluence plug-in integrates the Stand-Up Table with Rationale for meetings into the wiki because meeting protocols are managed there in practice. All ConDec plug-ins let the user navigate to the Knowledge Visualization in Jira. The navigation functionality is part of many features. For example, the Stand-Up Table with Rationale lets the user navigate to the documentation location of the knowledge elements. The ConDec Jira plug-in offers services for Knowledge Exploitation and Decision Knowledge Documentation. The other ConDec plug-ins use these services.

Where to persist and how to synchronize knowledge from different locations? This issue is a *data management issue* (Bruegge and Dutoit, 2010). ConDec uses Jira and git as the central decision knowledge repositories so that the decision knowledge can easily be traced from and to requirements and code. Decision knowledge from other locations, such as chat messages, is exported to Jira and then maintained there—close to and traceable from the requirements and code. An alternative would be integrating decision knowledge captured in chat messages, wiki pages, pull requests, and other distributed locations directly into the knowledge graph without exporting it to Jira. However, this would require storing trace links between different systems in a separate database table. Setting up this new database table would contradict the system design goals of low deployment, development, and maintenance costs.

How to persist decision knowledge captured in Jira ticket text? This issue is also a data management issue. To enable the explicit capturing of decision knowledge in the description and comments of Jira tickets and their linking with other knowledge elements, the ConDec Jira plug-in introduces two new database tables for knowledge elements and links managed through object-relational mapping. The database table for knowledge elements stores their start and end positions in the Jira ticket text. The database tables to capture decision knowledge in Jira ticket text are lightweight because they manage data within an individual system rather than across systems. They do not require additional deployment costs.

How to represent the knowledge graph of requirements, code, rationale, and other artifacts? This issue is an *object design issue* (Bruegge and Dutoit, 2010). The knowledge graph is represented with the jGraphT library, as also done by Carrillo and Capilla (2018). Next to the data structure, the jGraphT library offers graph algorithms. For instance, ConDec uses the algorithm for finding the shortest path for transitive linking. An alternative to jGraphT is the guava library, which was discarded because it does not offer algorithms. The knowledge graph is a singleton object and is only updated by changes, e. g., when a new element is added. The alternative would be to recreate the entire knowledge graph for every user interaction, which is inefficient and leads to long response times. Recreation would contradict the design goal of high performance. For graph visualization, ConDec uses the jsTree, treant, vis.js network, vis.js timeline, and Apache ECharts JavaScript libraries as they are open source.

How can the knowledge be exchanged between the ConDec plug-ins? This issue is also an object design issue addressing the interfaces between the ConDec plug-ins. REpresentational State Transfer (REST) Application Programming Interfaces (APIs) are extensively used throughout all the plug-ins for communication. The ConDec Eclipse plug-in uses the Jira REST Java client library to access Jira's REST API. The ConDec Jira plug-in offers a REST API to document and access decision knowledge. For example, this REST API is used in the ConDec Slack plug-in to export decision knowledge elements from Slack to Jira.

How to create and maintain the links defined in the ConRat knowledge model? This issue is also an object design issue. The ConRat knowledge model is an application domain model (Figure 6.1). During system and object design, it needs to be refined and translated into the solution domain (Bruegge and Dutoit, 2010). In ConDec, associations can be established between all types of knowledge elements in the knowledge graph to support flexible linking. The developers can manually link knowledge elements, but ConDec also offers mechanisms for automatic link creation and maintenance: 1) ConDec automatically links tickets in Jira with code in git if the code was changed in a commit that contains the ticket identifier in its commit message. ConDec maintains the links based on recent changes when calling the git fetch command. 2) ConDec automatically links decision knowledge elements documented in Jira ticket text (description or comments), commit messages, and code comments to related elements. For example, ConDec automatically links a decision problem to the respective ticket it was documented in. It links the solution options to the decision problem and arguments to solution options according to their sequential order in the text. Association types between solution options can be the following as suggested by Kruchten (2004): *enables*, *constrains*, *forbids*, *comprises*, *subsumes*, *overrides*, *conflicts with*, and *relates* (Section 2.2.4). In addition, pro-arguments *support* solution options, whereas con-arguments *attack* solution options. The rationale manager can add custom element types and link types, which makes the knowledge model flexible.

Which programming languages to choose? This issue is an implementation issue. The programming languages for implementing ConDec are constrained by the respective plug-in frameworks underlying each ConDec plug-in. Due to the constraints, the ConDec plug-ins are mainly written in Java, JavaScript, and other languages for web development.

7.3. Rationale Documentation in Various Locations

ConDec supports developers in documenting decision knowledge in different documentation locations. This enables developers to document decision knowledge within their current development context. Developers document decision knowledge when they 1) capture it, i. e., write it down, 2) annotate it, and 3) link it to other knowledge elements in the knowledge graph. This section introduces four documentation locations supported by ConDec: entire tickets (Section 7.3.1), description and comments of tickets (Section 7.3.2), commit messages (Section 7.3.3), and code comments (Section 7.3.4). These documentation locations enable decisions to be traced to requirements and other tickets in the issue tracking system and code in the version control system and can be used interchangeably. As specified in the ConRat knowledge model (Section 6.1), trace links between tickets and code exist, and tickets are linked. Because of the traceability, it is only a minor difference whether a decision is documented within the issue tracking system or version control system. In either location, the decision can be accessed from the requirement. Section 7.3.5 presents that ConDec also supports capturing decision knowledge in chat messages and outlines other locations currently not implemented.

7.3.1. Entire Tickets

Like requirements, work items, or bug reports, developers can document rationale elements as Jira tickets with particular types for issues, alternatives, arguments, and decisions. Figure 7.7 shows the ticket types available in a Jira project. Arguments documented as entire tickets are unpositioned if they are not linked. An unpositioned argument becomes a pro- or a con-argument if it is linked to a solution option (alternative or decision) with the link type *supports* or *attacks*.

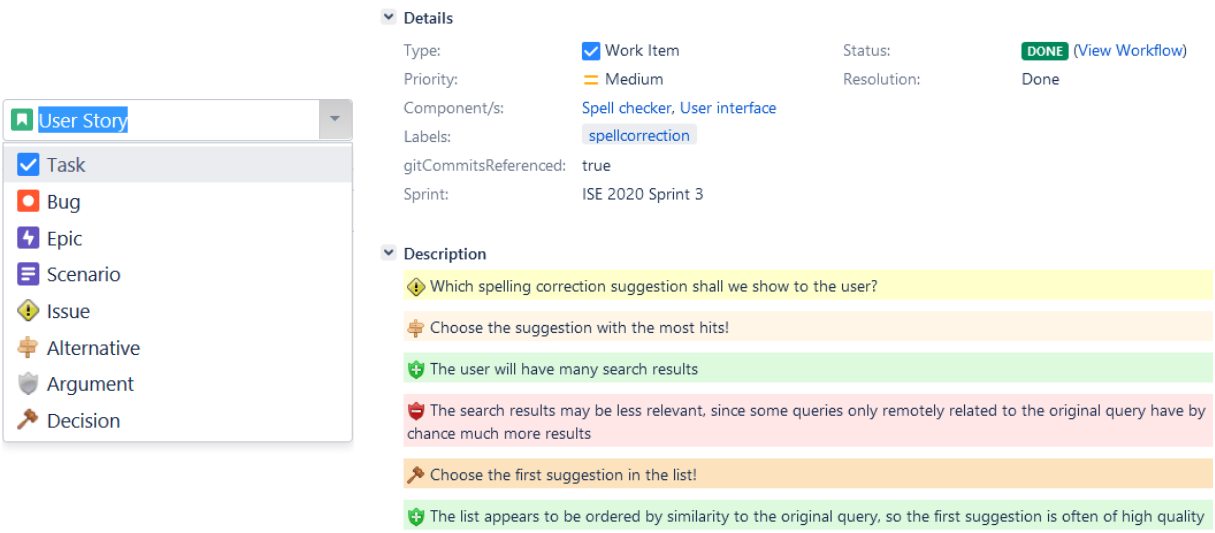

Figure 7.7.: **Left**: Jira ticket types available in a project. The four lower types are for rationale elements. **Right**: Decision knowledge captured in the description of a work item.

7.3.2. Description and Comments of Tickets

Developers can annotate text in Jira tickets, such as in requirements and work items, as decision knowledge elements to make the decision knowledge explicit. The developers can annotate text parts as decision knowledge elements in two ways: 1) They can use a markup syntax, e. g., `{decision} ... {decision}`. This markup syntax is similar to other built-in annotations offered by Jira. 2) They can annotate the sentences containing the decision knowledge elements with the decision knowledge icons similar to Alkadhi et al. (2017a) and Alkadhi (2018) in chat messages. Figure 7.7 shows a work item with decision knowledge captured in its description.

7.3.3. Commit Messages

In commit messages in git, developers can annotate text parts as decision knowledge elements using a markup syntax, e. g., `[decision] ... [/decision]`. Commit messages are different from the other three documentation locations as they are permanent. They could be changed using *reword rebasing*, but this should not be done for commits that exist in the remote repository

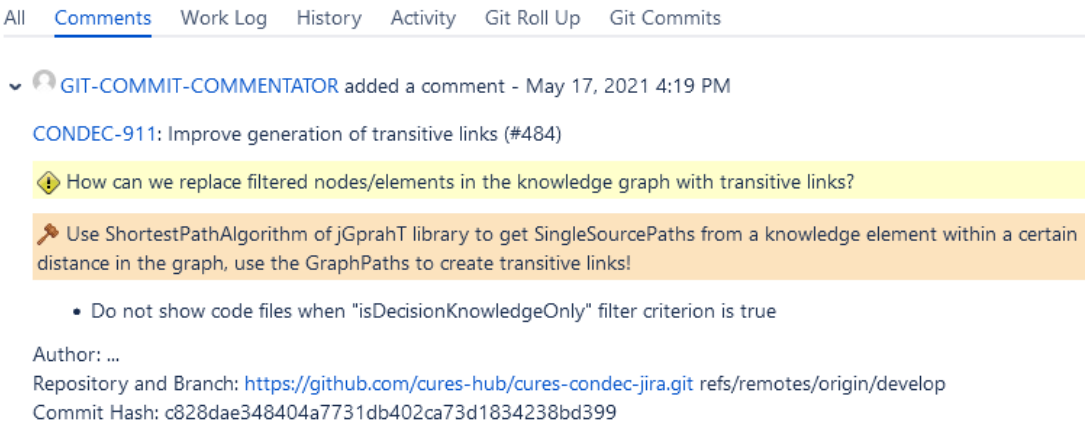

Figure 7.8.: Commit message with explicit decision knowledge that ConDec automatically transcribed into a Jira ticket comment.

and that people may have based work on (Chacon and Straub, 2014). To enable changing and annotating commits retrospectively, ConDec transcribes, i. e., posts commit messages into Jira ticket comments. For every commit message, ConDec creates a new comment for the Jira ticket mentioned in the commit message. Developers can annotate the text and improve the decision knowledge documentation in the Jira ticket comment. Figure 7.8 shows an example of a decision captured in a commit message of the ConDec Jira plug-in.

7.3.4. Code Comments

In code comments, developers annotate text parts as decision knowledge elements using a Javadoc-like syntax, e. g., `@decision`, in the integrated development environment (Hesse et al., 2015; Hesse, 2020). Figure 7.9 shows an issue discussed in the comment of a Java method and displays the resulting excerpt of the knowledge graph with the code file being the selected knowledge element. This issue is taken from the ConDec project. Developers can document decision knowledge in code through annotations in all integrated development environments.

```
/**
 * @issue Who should be the author of the new
     Jira ticket comment that a commit messages
     was posted into?
 * @decision The user "GIT-COMMIT-COMMENTATOR"
     creates the Jira ticket comment that a
     commit messages was posted into!
 * @pro It is clear that the comment originated
     from a commit message.
 * @alternative The user that opens the Jira
     ticket could be the creator of the Jira
     ticket comment that a commit messages was
     posted into.
 * @con It would be confusing to users if they
     see that they posted something that they
     did not write.
 * @alternative The git user could be the
     creator of the Jira ticket comment that a
     commit messages was posted into.
 * @con Git user names can be different to Jira
     user names and it is hard to match them.
 */
private ApplicationUser getUser() {
```

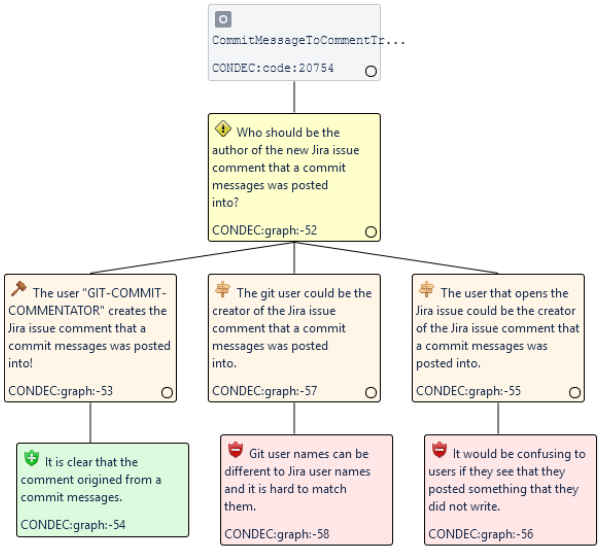

Figure 7.9.: **Left:** Decision knowledge captured in a code comment using annotations in the class CommitMessageToCommentTranscriber of the ConDec project.
Right: Node-link tree diagram with the class being the root knowledge element.

7.3.5. Chat Messages, Wiki Pages, and Pull Requests

The ConDec Slack plug-in enables developers to annotate chat messages with decision knowledge icons (Alkadhi et al., 2017a; Alkadhi, 2018) and to export the decision knowledge to Jira so that it can be accessed from tickets and code. After the developers annotate a message, a chatbot supports them in integrating the decision knowledge into the knowledge graph. The developers choose whether they want to add the decision knowledge from Slack as a comment to an existing Jira ticket (Section 7.3.2) or whether they want to create a new Jira ticket (Section 7.3.1). Similarly, the developers could apply decision annotations in other CSE artifacts (Table 3.4), e. g., in pull requests or wiki pages.

7.4. Views on the Knowledge Graph

An instance of the ConRat knowledge model is called a *knowledge graph* (Section 6.1). ConDec provides seven different views (V1 – V7) on the knowledge graph, which are common views for graphs in software visualization: Section 7.4.1 presents the node-link diagram. Section 7.4.2 introduces tree views. Section 7.4.3 presents list views. Section 7.4.4 shows matrix views. Section 7.4.5 presents the chronology view. Section 7.4.6 introduces metrics views. Section 7.4.7 presents knowledge element detail views. The views usually show only a part of a project's knowledge graph data structure, called *knowledge subgraph*.

7.4.1. Node-Link Diagram (V1)

ConDec visualizes the knowledge (sub-)graph as a node-link diagram. Figure 7.10 shows an example of a node-link diagram of a realistic, yet rather small knowledge subgraph. The node-link diagram shows an epic, six user stories, code, and decision knowledge elements. This visualization illustrates the problem of a high amount of distributed knowledge by showing that knowledge tends to become complex. In the knowledge subgraph, code files are linked to work items, and the work items are linked to user stories. In Figure 7.10, work items are omitted for simplification, and code files are transitively linked to user stories. ConDec explicitly presents the decision knowledge elements in addition to the requirements.

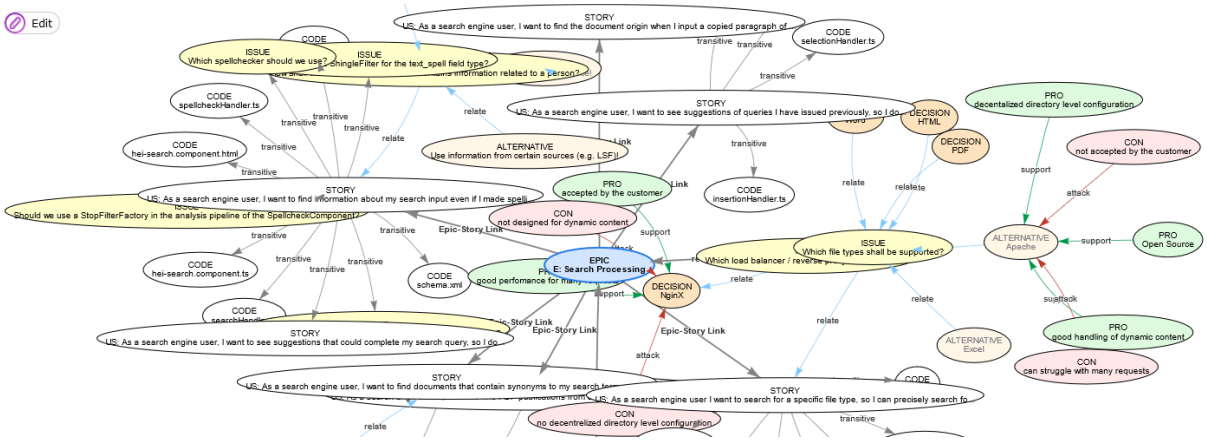

Figure 7.10.: Node-link diagram (V1) showing the context of the epic *Search Processing* in the center. The subgraph shows user stories, code, and decision knowledge traceable from the epic in a link distance of 3 in the knowledge graph. Transitive links between user stories and code replace filtered-out work items (development tasks).

7.4.2. Knowledge Tree View (V2)

ConDec visualizes knowledge subgraphs as a tree starting from a selected knowledge element as the root. ConDec provides two different visualizations: *indented outline* ($V2_{ind}$) and *node-link tree diagram* ($V2_{nld}$). Figure 7.11 shows an indented outline starting from a user story. Figure 7.12 shows a node-link tree diagram starting from a decision problem (issue).

Knowledge graphs can contain cycles. These cycles need to be removed when converting the graph into a tree by duplicating knowledge elements in the tree views. Figure 7.16 shows an example where a cycle is established because the solution option *Standard Parser* for the issue *Which Solr query should we use?* has two con-arguments linked to the same quality requirement.

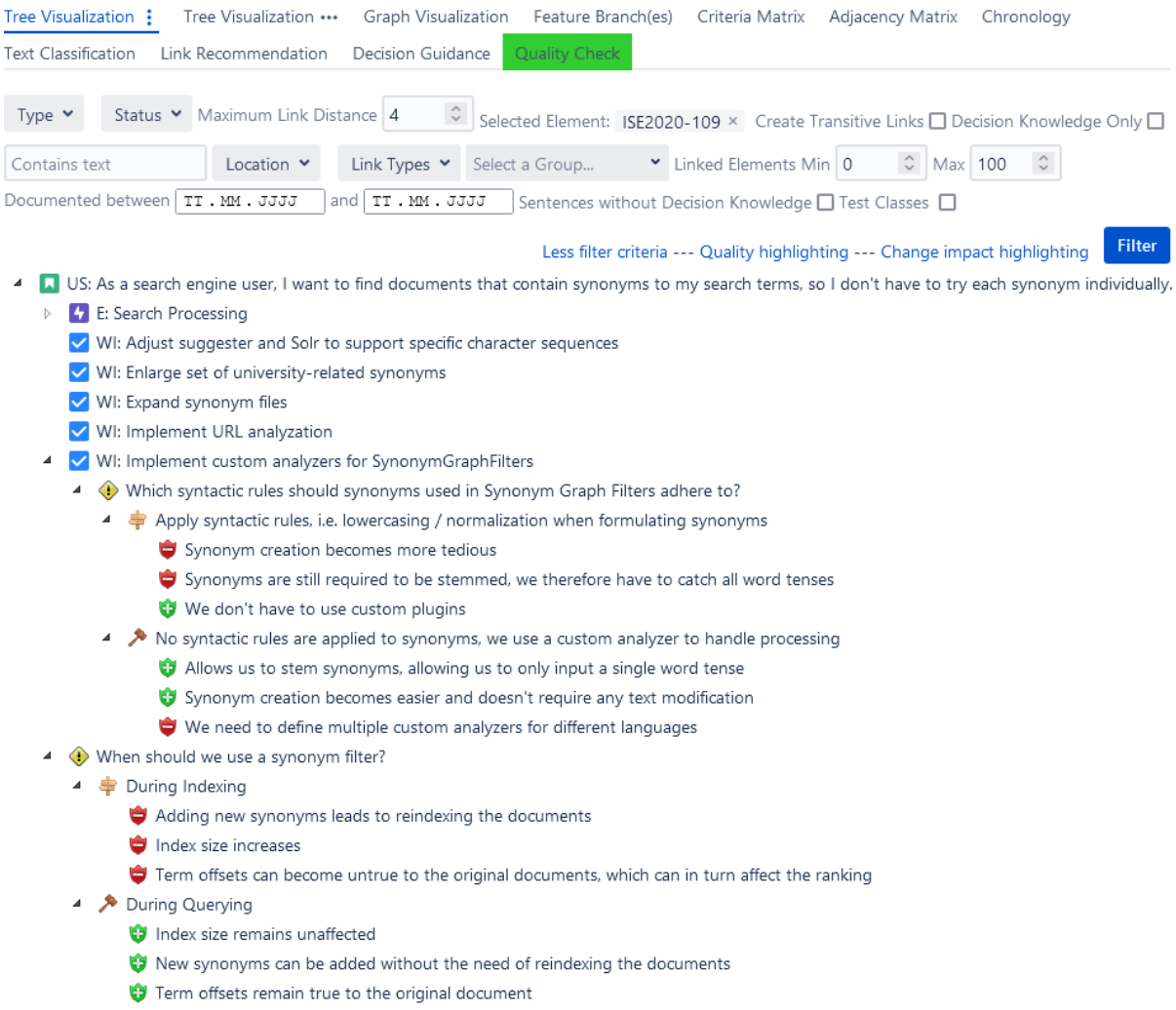

Figure 7.11.: Knowledge tree view ($V2_{ind}$) with a user story being the root element. The user story is linked to an epic (E), work items (WI), and decision knowledge. Decision knowledge is indirectly connected to the user story via the link to the work item.

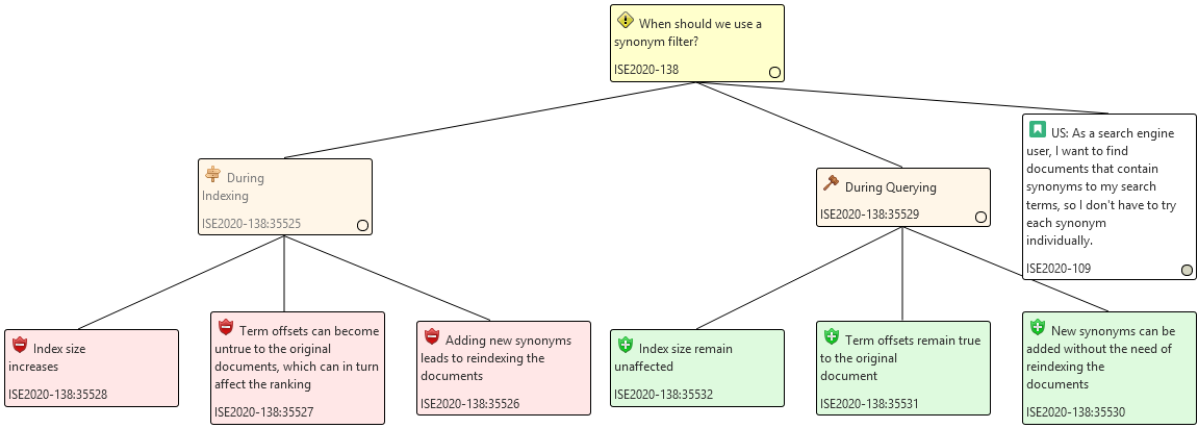

Figure 7.12.: Knowledge tree view ($V2_{nld}$) starting from a decision problem (issue). The decision problem is linked to a user story. The user story is also linked to further knowledge elements, but the subtree is collapsed.

7.4.3. List View (V3)

ConDec visualizes (parts of) the knowledge graph as a list of knowledge elements. The list view can be integrated as a stand-up table into meeting agendas in the wiki using the ConDec Confluence plug-in so that developers can discuss recently made decisions and open decision problems during meetings (Section 7.10). The ConDec Bitbucket plug-in lists the issues and decisions related to a feature branch within the pull request view. ConDec supports the creation of release notes, including explicit decision knowledge for knowledge sharing (Section 7.11). Besides, ConDec offers a dedicated view for the knowledge in git, which lists commits, code files, and decision knowledge documented in commit messages and code comments (Figure 7.13).

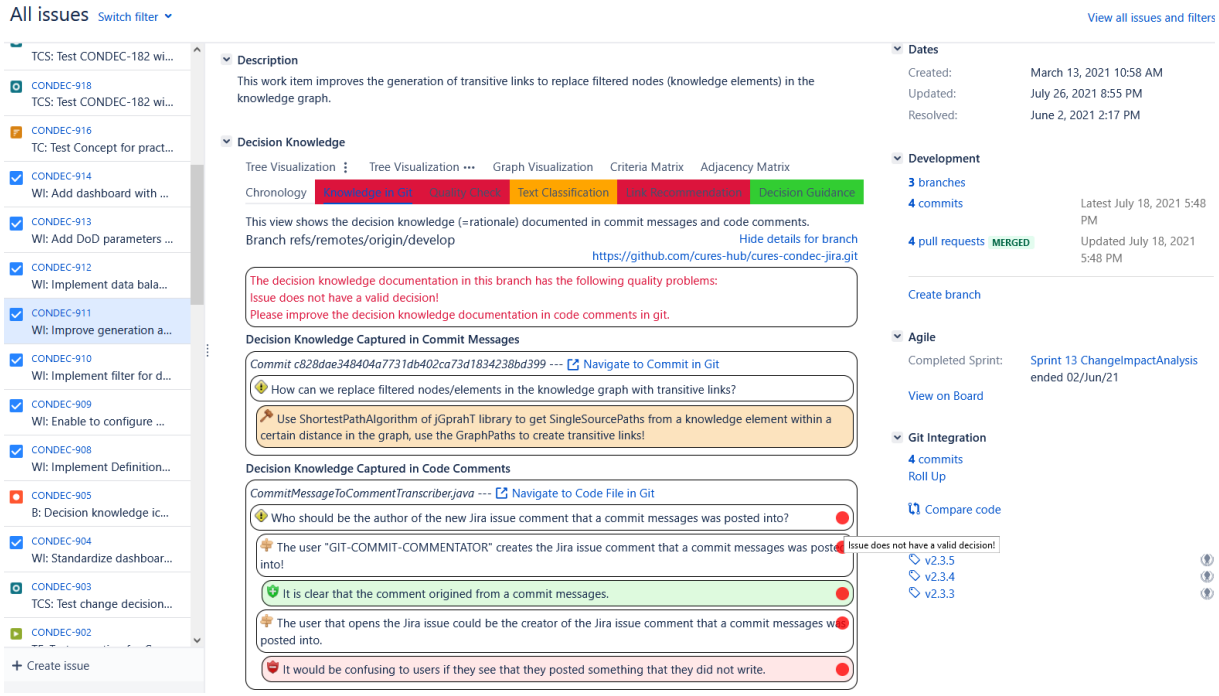

Figure 7.13.: View on knowledge in git for a specific Jira ticket highlighting quality problems to nudge the developers to improve the quality.

7.4.4. Adjacency and Criteria Matrix View (V4)

ConDec visualizes (parts of) the knowledge graph as an adjacency matrix. For decision problems, ConDec shows the solution options, arguments, and criteria as a criteria matrix. Figure 7.14 shows an *adjacency matrix* ($V4_{adj}$) for decision knowledge elements. A matrix cell is colored if a directed link exists from a knowledge element in the row to another element in the column. Link type colors are fixed for default link types such as *attacks* (red) and *supports* (green). To allow adaptable link types, such as *Epic-Story Link*, ConDec uses the hash value of the link type names as the color. However, these colors are sometimes hard to distinguish.

Figure 7.15 shows a *criteria matrix* ($V4_{cri}$) for a decision problem. This view visualizes criteria used during decision making as columns. Criteria can be quality requirements or constraints such as the implementation effort. Developers can then use the node-link diagram (V1) or tree view (V2) to see all decisions that support or attack a certain quality requirement (Figure 7.16).

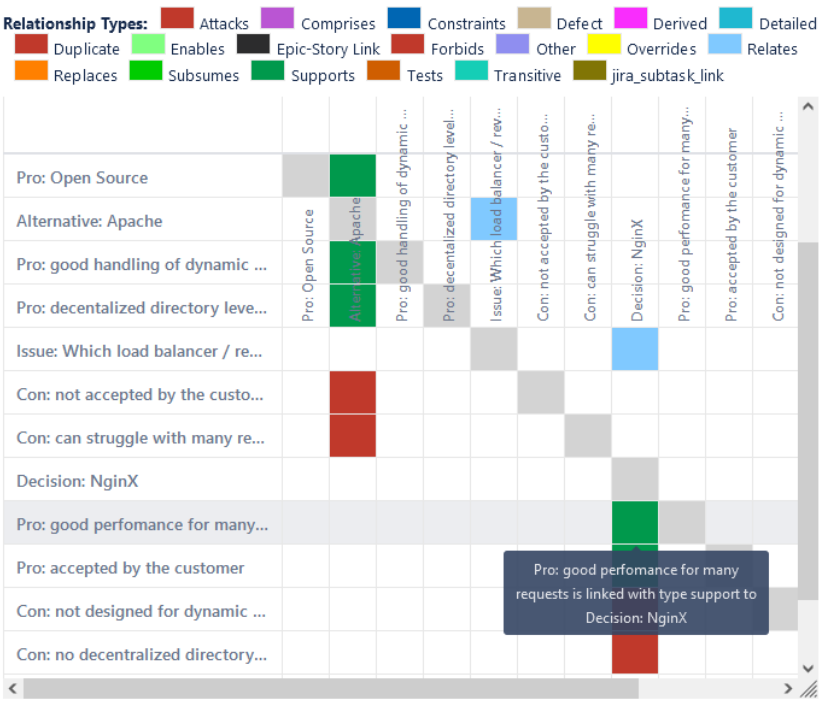

Figure 7.14.: Adjacency matrix view ($V4_{adj}$) for a decision problem. The matrix cells are colored if there is a directed link (direction matters). The color indicates the link/relationship/edge type in the knowledge graph.

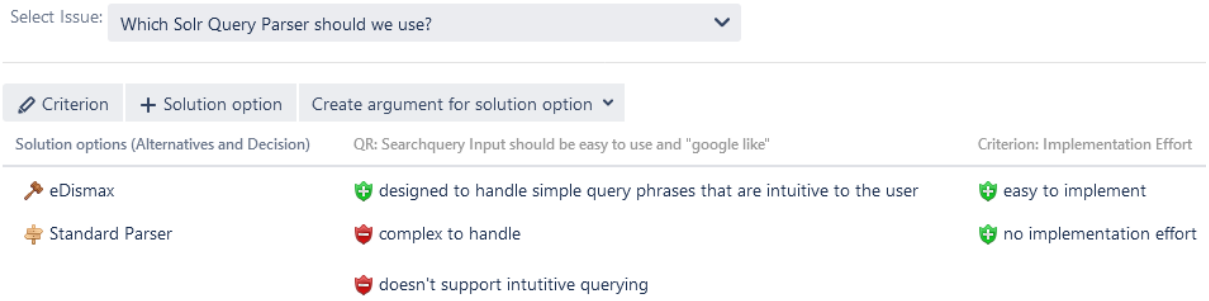

Figure 7.15.: Criteria matrix view ($V4_{cri}$) for a decision problem, including solution options (alternative and the decision), arguments, and criteria.

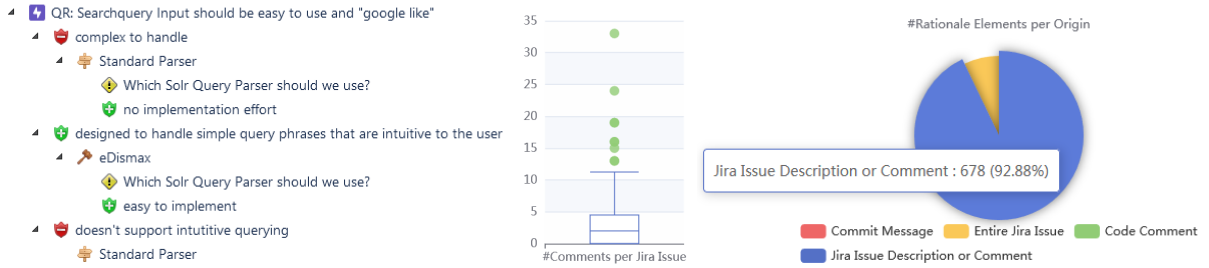

Figure 7.16.: **Left**: Knowledge tree view ($V2_{ind}$) for decision knowledge linked to a quality requirement, which was one criterion for decision making in Figure 7.15.
Right: Metrics view (V6) box plot to present the number of comments per Jira ticket and pie chart to give an overview where rationale elements are documented.

7.4.5. Chronology View (V5)

ConDec visualizes the knowledge elements of the knowledge graph in chronological order depending on their creation time or time of last change/update. Figure 7.17 shows decisions documented at the beginning of the ISE 20/21 case study project in chronological order. The chronology view can also plot other knowledge elements at their creation date or last update, e. g., requirements, code files, or other rationale elements. Filters specify if the elements are placed on a specific date (as shown in Figure 7.17) or range from their creation date to the last update of the last update.

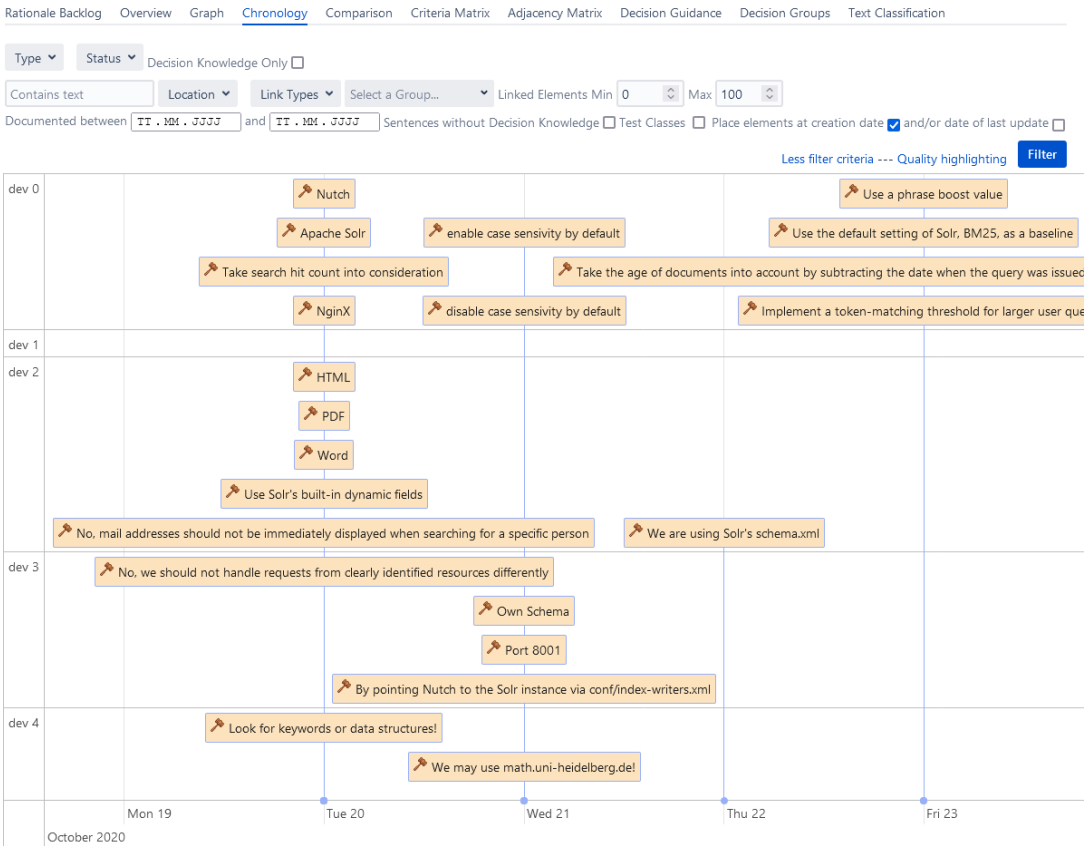

Figure 7.17.: Chronology view (V5) of decisions made at the beginning of the ISE 20/21 case study project. The x-axis shows the documentation date of the decisions. On the y-axis, the decisions are grouped according to the developers (anonymized) who documented them. Filtering criteria are shown at the top.

7.4.6. Metrics View (V6)

ConDec visualizes metrics calculated on the knowledge graph data structure in a dashboard, e. g., using pie charts and box plots. Figure 7.16 shows a box plot and a pie chart as an excerpt of the ConDec dashboard. The metrics help to assess the documentation quality or are merely informative to provide an overview of the amount of knowledge in a project.

7.4.7. Detail View of Knowledge Element (V7)

The detail view shows a specific knowledge element with its attributes. Attributes include the author, description, status, time of creation, and last update. For detail views, the developers use the views of the tools that ConDec extends, e. g., the Jira ticket view (Figure 7.7) or the code editor of the integrated development environment.

7.5. Features of the Knowledge Graph Views

ConDec allows adapting the views on the knowledge graph through filtering, interaction, and highlighting. This section describes the basic features (F1 – F5): Section 7.5.1 describes filtering functionality. Section 7.5.2 presents transitive linking as a special filtering introduced by ConDec. Section 7.5.3 describes possibilities for change execution. Section 7.5.4 presents possibilities to specify the level of detail. Section 7.5.5 presents the navigation functionality.

Table 7.6.: Features (F1 – F5) available in knowledge graph views (V1 – V7). The knowledge graph views include and highlight the results of recommendation systems (RS).

View/Feature	Filtering, F1	Transitive linking, F2	Change execution, F3	Specifying level of detail, F4	Navigation, F5	Quality highlighting, RS1	Change impact highlighting, RS2
Node-link diagram, V1	✓	✓	✓	✓	✓	✓	✓
Tree, V2	✓	✓	✓	✓	✓	✓	✓
List, V3	✓	✓	✓	✗	✓	✓	✓
Adjacency matrix, $V4_{adj}$	✓	✓	✓	✗	✓	✓	✓
Criteria matrix, $V4_{cri}$	✓	✗	✓	✗	✓	✓	✗
Chronology, V5	✓	✗	✓	✓	✓	✓	✗
Metrics, V6	✓	✓	✗	✓	✓	✓	✗
Detail, V7	✗	✗	✓	✗	✗	✓	✗

The features are must-be features in software visualization, except for transitive linking, a new feature useful for distributed knowledge documentation. Section 7.6 will introduce nudging mechanisms and recommendation systems. Developers see the results of the recommendation systems in the knowledge graph views. ConDec highlights quality problems and change impacts through colors. The goal was to support the same features in all views so that the developers do not have to change their working context for knowledge management. However, differences exist due to the nature of the view (Table 7.6). For example, developers cannot perform changes in metrics plots because metrics plots report measurement results.

7.5.1. Filtering (F1)

Developers filter the knowledge graph using various filter criteria. For instance, filter criteria are the knowledge type, status (e. g., resolved, unresolved, decided, rejected), documentation location (Jira ticket, Jira ticket text, commit message, code comment), number of hops/link distance, node degree (min and max number of links), textual content, time, and decision types. These basic filtering possibilities are the same for all views that show more than one element, but some views provide additional filter criteria, e. g., the chronology view (V5). Figure 7.11, Figure 7.17, and Figure 7.19 show filtering possibilities.

7.5.2. Transitive Linking (F2)

Developers exploit transitive links between knowledge elements. For example, they examine all decisions made in the context of an epic. The decisions can be documented in user stories, work items, commit messages, and code files traceable from the epic. Figure 7.18, Figure 7.10, and Figure 7.19 illustrate filtering of knowledge elements combined with transitive linking. ConDec only creates new transitive links for unconnected elements. The transitive links in the knowledge graph do not necessarily create a transitive closure (Figure 7.18). Algorithm 1 shows the pseudocode of the transitive linking. Transitive linking is supported in all views, except for V5 and V7 because they show no links.

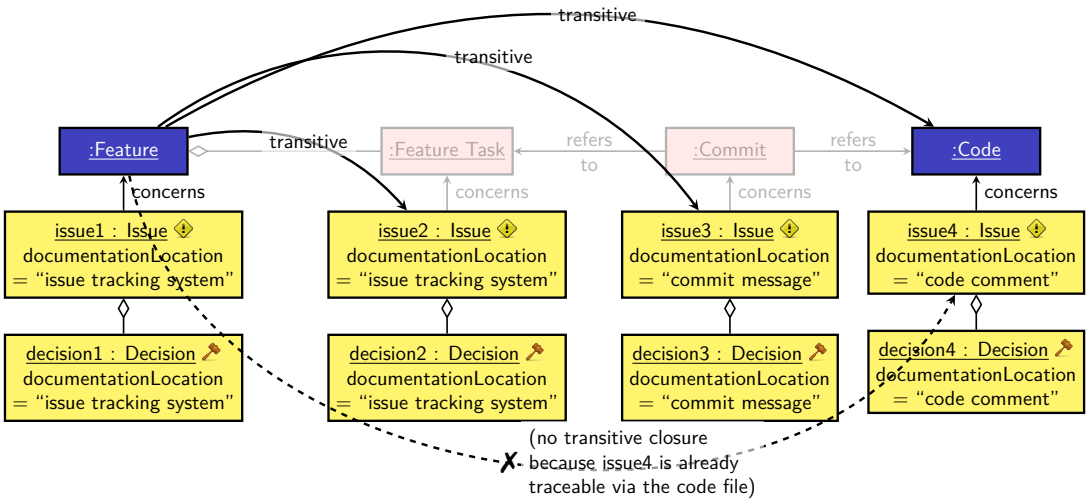

Figure 7.18.: Schematic illustration of transitive linking (UML object diagram). The faint knowledge elements and links are filtered out. The transitive links replace filtered-out elements. The dashed line indicates a link necessary to create a transitive closure, which ConDec does not create because all the elements are already traceable.

Algorithm 1: Replacing filtered-out knowledge elements on graph path with transitive links.

Input: graph, selectedElement, linkDistance, filterCriteria
Result: filtered graph in that filtered-out knowledge elements (nodes/vertices) are replaced with transitive links (edges/relationships)

```
1  singleSourcePaths ← findAllShortestPaths(graph, selectedElement, linkDistance)  // find all shortest paths starting from the selected element in the unfiltered knowledge graph within link distance using Dijkstra
2  filteredGraph ← createFilteredGraphThatMatchesFilterCriteria(graph, filterCriteria)  // filter knowledge graph according to filter criteria, e.g., element type, status, documentation location, node degree, decision type
3  for element : filteredGraph.vertexSet() do  // iterate over the remaining elements in the filtered graph
4      path ← singleSourcePaths.getPathTo(element)  // get path in the unfiltered knowledge graph
5      lastUnfilteredElementOnPath ← selectedElement  // remember visited element on path that is not filtered out
6      for elementOnPath : path.getVertexList() do  // iterate over elements on the path in the unfiltered graph
7          if !filteredGraph.vertexSet().contains(elementOnPath) then  // element on the former path is filtered out
8              continue  // keep on walking along the path until we find an element not filtered out
9          else  // the element on the former path is still existing in filtered graph
10             if !filteredGraph.containsEdge(lastUnfilteredElementOnPath, elementOnPath) then  // no link yet?
11                 transitiveLink ← new Link(lastUnfilteredElementOnPath, elementOnPath, LinkType.TRANSITIVE)
12                 filteredGraph.addEdge(transitiveLink)  // add new transitive link between elements on same path
13             lastUnfilteredElementOnPath = elementOnPath  // remember visited element on path not filtered out
```

7.5.3. Change Execution (F3)

Developers create, update, and delete knowledge elements and links within the views of the knowledge graph. To maintain high-quality knowledge documentation, developers must correctly document the knowledge elements and links. They can add new decisions for implementing a requirement using a *context menu* as shown in Figure 7.19. The context menu lets developers change the type of a rationale element, for example, to pick an alternative as the decision. ConDec automatically changes the annotations if the decision knowledge element is documented in the text. The following paragraphs detail the functionalities of the change execution feature.

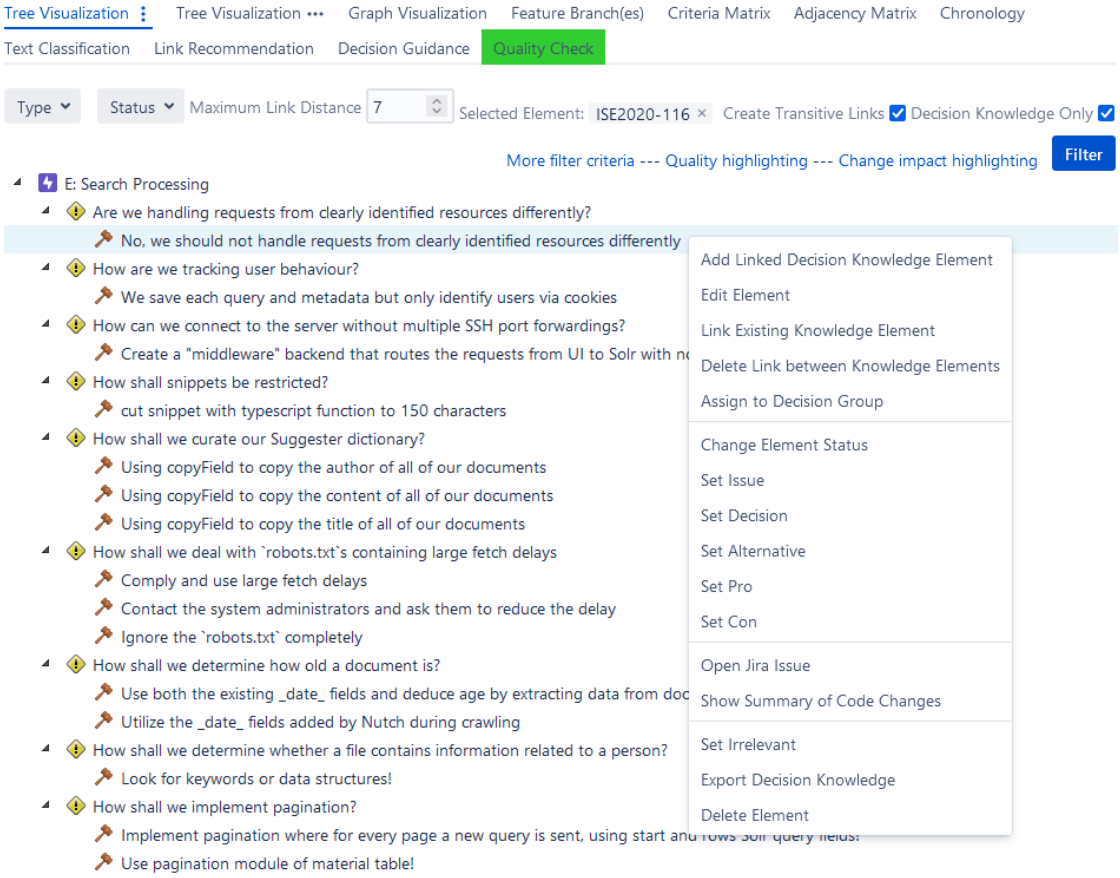

Figure 7.19.: Epic with decision problems (issues) and decisions transitively linked to the epic within the link distance 7 in the knowledge graph data structure. Filtering possibilities and the context menu are shown.

Linking *Developers link decision knowledge elements to other knowledge elements to be accessed within the knowledge graph.* Explicit decision knowledge elements are nodes in the knowledge graph that can only be accessed from other nodes (knowledge elements) if they are linked. For this purpose, ConDec supports manual linking between all knowledge elements. The developers can add and delete links, as well as update link types using *drag and drop* or by clicking a matrix cell (V4). Some visualization libraries such as vis.js offer additional manipulation possibilities.

Attribute Assignment to Elements *Developers add, update, and delete attributes for knowledge elements.* Similar to requirements, rationale elements have attributes for metadata. For example, attributes can be the state, e. g., idea, decided, rejected (Section 6.1.2), or a decision type (Section 7.9). ConDec stores attributes for text parts via object-relational mapping.

Marking Links as Wrong or Useless *Developers mark links that are wrong or useless during knowledge exploitation.* Trace links between Jira tickets and code can be wrong (Hübner and Paech, 2020). A reason for wrong links can be that a commit linked to a Jira ticket contains *tangled changes* (Herzig and Zeller, 2013). Tangled changes are unrelated changes. For instance, a commit linked to a feature task for improving a specific feature is tangled if it also contains a bug fix for a different feature. Tao and Kim (2015) empirically found that 29 % of commits in four open source projects were tangled, i. e., leading to wrong links. Wrong links hinder the exploitation of the knowledge graph in change impact analysis or in using transitive links. ConDec enables developers to mark links as wrong as follows: If developers delete a link that involves a code file, ConDec sets the type of the link to *wrong* instead of completely deleting it. Otherwise, the links would be recreated when ConDec creates the knowledge graph. In the same way, developers can mark useless transitive links as wrong. In Figure 7.19, the issue *How shall we implement pagination?* is transitively linked to the epic *Search Processing* but is unrelated and, thus, marked as wrong. Links marked as wrong are filtered-out, i. e., not shown in the views on the knowledge graph, and not traversed during change impact analysis and transitive linking.

7.5.4. Specifying the Level of Detail (F4)

Developers change the level of detail to either understand the big picture (e. g., how knowledge elements relate to each other) or to see details (e. g., a summary of a particular knowledge element). The node-link diagram (V1) and chronology view (V5) allow specifying the level of detail by *zooming* in and out. Besides, parts of the view can be *collapsed* in the node-link diagram (V1), tree (V2, Figure 7.12), and metrics (V6) views.

7.5.5. Navigation (F5)

Developers navigate to other parts of the knowledge graph and different knowledge graph views. For example, the navigation is enabled through the context menu (Figure 7.19), through hyperlinks on user interface elements, such as menu items and matrix headers (V4), or by clicking on data points in the metrics plots (V6). ConDec also provides extensions for integrated development environments (Eclipse and Visual Studio Code) that enable to navigate from code files to the knowledge graph views in Jira with the code file being the selected element.

7.6. Nudging Mechanisms and Recommendation Systems

This section describes three nudging mechanisms (N1 – N3) and six recommendation systems (RS1 – RS6) for rationale management. They are combined into one section because recommendation systems typically involve nudging mechanisms, and recommendations can be seen as a nudging mechanism themselves (Jesse and Jannach, 2021). ConDec's nudging mechanisms and recommendation systems are inspired by the approaches identified in the systematic mapping study (Chapter 4) and practitioners' request for automation (Section 3.2.3).

The integration of capturing possibilities and views on the documented knowledge in the developers' tools motivates them to perform rationale management because they are frequently presented with the knowledge documentation. Knowledge graph views enable tailored access to the part of knowledge related to the requirements or code that developers work on. Developers can spot quality problems, e. g., inconsistency and incompleteness, and improve the knowledge documentation without changing their working context in a low-intrusive way. Nevertheless, the documentation of decision knowledge is still a task that developers must manually perform. Furthermore, developers make decisions in a naturalistic way (Zannier et al., 2007; Hesse et al., 2016b) and can be subject to *cognitive biases* (Section 2.2.2). When influenced by cognitive

biases, developers might not pick the best solution to a decision problem but cling to the solution that comes first to their mind (Razavian et al., 2016; Razavian et al., 2023). In addition, the decision documentation is hard to understand for others if alternatives and arguments are missing. Thus, continuous rationale management can benefit from 1) a further reduction of manual documentation work and 2) support to overcome cognitive biases if potentially harmful, such as anchoring, and 3) exploiting cognitive biases, such as the status-quo bias.

The term *nudging* was introduced by Thaler and Sunstein (2008). Nudging means to alter people's behaviors in predictable ways by subtly changing the *choice architecture*. The concept of nudging originates from behavioral economics but is also used in human-computer interaction. Nudging mechanisms—or short *nudges*—aim to guide the "users towards desired choices and behaviors". Nudging mechanisms can help developers to adopt software engineering activities (C. Brown, 2019). Caraban et al. present a framework for technology-mediated nudging that contains 23 distinct nudging mechanisms grouped into six categories (Caraban et al., 2019; 2020).

Section 7.6.1 presents facilitate nudges. Section 7.6.2 describes ambient feedback and friction nudges. Section 7.6.3 presents just-in-time prompts. Section 7.6.4 describes quality checking. Section 7.6.5 presents change impact analysis. Section 7.6.6 describes decision guidance. Section 7.6.7 presents link recommendation and duplicate detection. Section 7.6.8 describes automatic text classification. Section 7.6.9 introduces the summarization of source code changes.

7.6.1. Facilitate Nudges (N1)

ConDec offers several nudging mechanisms to reduce the effort for tool usage. It implements *default options* and *opt-out policies* as *facilitate nudges*. They are motivated by the observation that humans generally take the path of least resistance, which leads to the *status-quo bias* (Caraban et al., 2019). An example of the *default options* nudge in ConDec is the following: A new issue has the state *unresolved* per default. ConDec automatically changes the state of the issue to *resolved* if a decision is linked to the issue. *Opt-out policies* can be found throughout the configuration features of ConDec. For instance, the decision guidance from DBPedia (RS3) is activated per default but could be deactivated by the rationale manager.

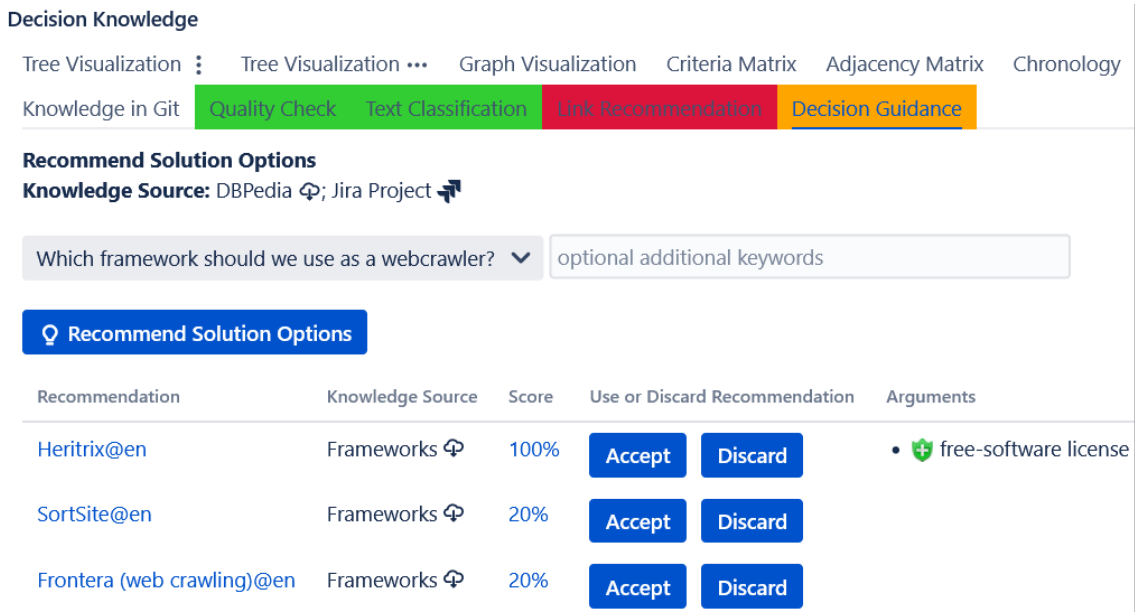

Figure 7.20.: Decision guidance view with three recommendations generated from DBPedia. The colored menu items indicate whether action is needed.

7.6.2. Ambient Feedback and Friction Nudges (N2)

ConDec colors graphical user interface elements, such as menu items, according to the number of recommendations that have not been accepted or discarded by the developers. The ambient feedback and friction nudging mechanism indicates if recommendations regarding rationale management exist and motivates the developers to consider the recommendations. Figure 7.20 shows colored menu items of four recommendation systems: Quality checking (RS1), decision guidance (RS3), link recommendation and duplicate recognition (RS4), and automatic text classification (RS5). For quality checking, no recommendations exist if the definition of done is fulfilled. For the other recommendation systems, all recommendations must be accepted or discarded by the developers. If there are no recommendations, menu items are colored green to give *ambient feedback*. If there are many recommendations, menu items are colored red, and if there are a few, menu items are colored orange. This is a way to *create friction* to nudge the developers to take action. Besides, ConDec highlights the knowledge elements violating the definition of done with red text in the knowledge graph views to indicate quality problems (RS1).

7.6.3. Just-in-Time Prompts (N3)

ConDec shows a just-in-time prompt to the developers when they change the state of a Jira ticket, e. g., when they start or finish a requirement. Figure 7.21 shows a just-in-time prompt with recommendations generated by the following recommendation systems: Quality checking (RS1), decision guidance (RS3), link recommendation and duplicate recognition (RS4), and automatic text classification (RS5). The rationale manager can activate or deactivate the events for that just-in-time prompts are shown. The events are activated per default for opt-out nudging (N1).

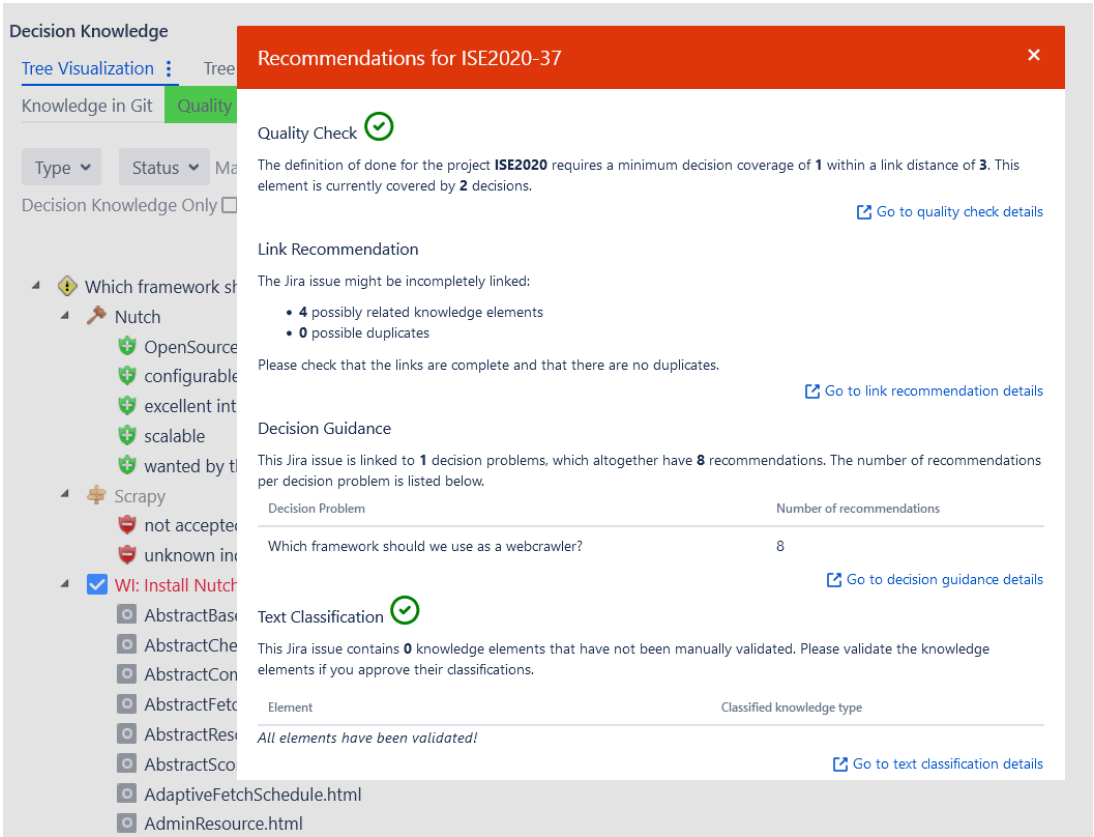

Figure 7.21.: Just-in-time prompt (N3) showing the status of the quality check (RS1), decision guidance (RS3), link recommendation (RS4), and text classification (RS5).

7.6.4. Quality Checking (RS1)

ConDec checks if the knowledge documentation fulfills the definition of done. Developers see which knowledge elements violate or fulfill the definition of done. If the definition of done is not fulfilled, they see which definition of done criteria are violated. ConDec implements the definition of done for knowledge documentation (Section 6.2.2). It supports and partly automates reviewing the rationale documentation demanded in ConRat. ConDec checks whether the definition of done is fulfilled for knowledge elements in the knowledge graph and visualizes the check results to trigger developers to improve the quality.

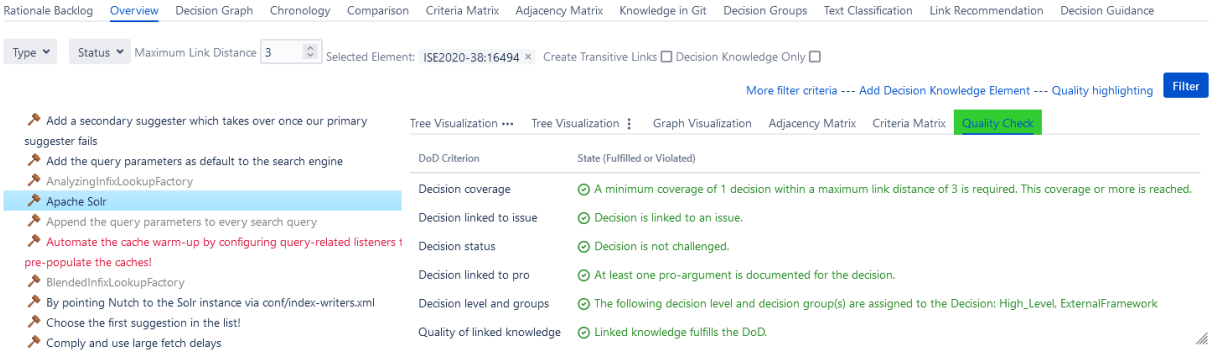

Figure 7.22.: Quality check view showing the quality check results for the selected decision. The decision and the linked knowledge elements fulfill the definition of done.

ConDec displays the result of the quality checking in six ways to confront the developers with the quality often and to nudge improvements: 1) ConDec displays the result of the quality checking in the quality check view accessible from every knowledge element in the knowledge graph (Figure 7.22). The menu item to access the quality check view is colored in green if the definition of done is fulfilled, orange if some but not all definition of done criteria are fulfilled, or red if no definition of done criterion is fulfilled. 2) ConDec highlights the knowledge elements that

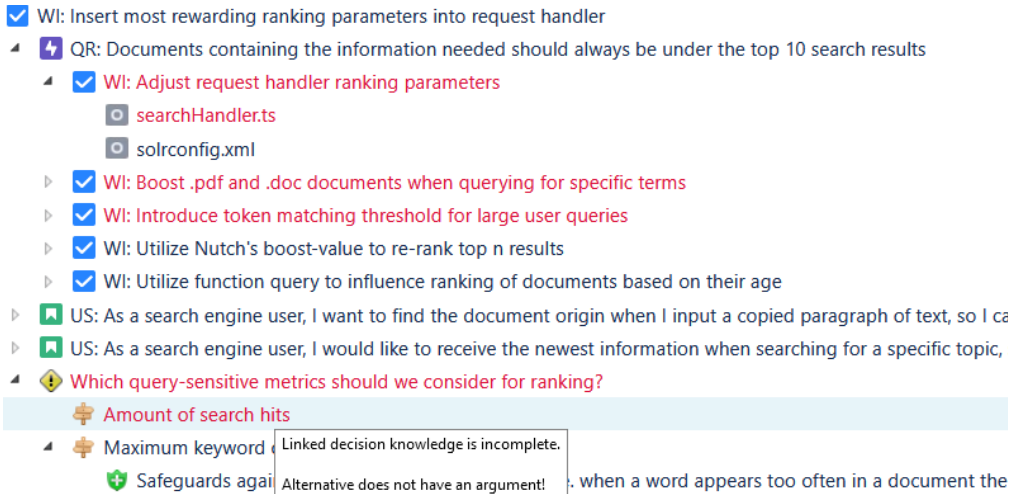

Figure 7.23.: Knowledge tree view ($V2_{ind}$) with knowledge elements violating the definition of done and quality highlighting. The decision problem (issue) is colored in red because it is unresolved, i. e., a decision needs to be made. The alternative is colored in red because no arguments are linked, i. e., a criterion for intra-rationale completeness is violated (as indicated by the tooltip). Three work items (WI) and one code file (searchHandler.ts) are colored red because their decision coverage is too low.

violate the definition of done with red text within the knowledge graph views (Figure 7.23) to indicate quality problems. Tooltips explain which definition of done criteria are violated. ConDec indicates quality problems through text coloring in the node-link diagram, tree, list, matrix, and chronology views (V1 – V5). 3) ConDec displays the result of the quality checking in a just-in-time prompt (N3, Section 7.6.3) that is shown during status changes, e. g., when finishing a requirement (Figure 7.21). 4) ConDec offers the rationale backlog, which is a dedicated view for knowledge elements that violate the definition of done (Section 7.7). 5) ConDec displays the result of the quality checking in metric plots in the knowledge dashboard (Section 7.8). 6) ConDec offers the merge-check in pull requests (Figure 5.2): The developers can only accept a pull request if the linked knowledge fulfills the definition of done.

ConDec checks the fulfillment of the following criteria automatically: 1) ConDec checks the criteria of the intra-rationale completeness for decision knowledge elements, e. g., if a pro-argument is linked to the decision. The rationale manager can configure the criteria of intra-rationale completeness. 2) ConDec checks the decision coverage for every knowledge element, i. e., if a minimum number of decisions is traceable within a certain link distance in the knowledge graph. The rationale manager can configure the required number of decisions and the maximum link distance. 3) ConDec checks the assignment to decision types (Section 7.9). 4) It checks specific aspects (number of lines of code, test code) for code files. 5) ConDec also propagates definition of done violations to neighbor elements in the knowledge graph to motivate the developers to improve the legacy knowledge documentation. If a neighbor knowledge element of the checked element violates the definition of done, the checked element will also violate the definition of done. Figure 7.24 shows the view to configure the criteria for the definition of done.

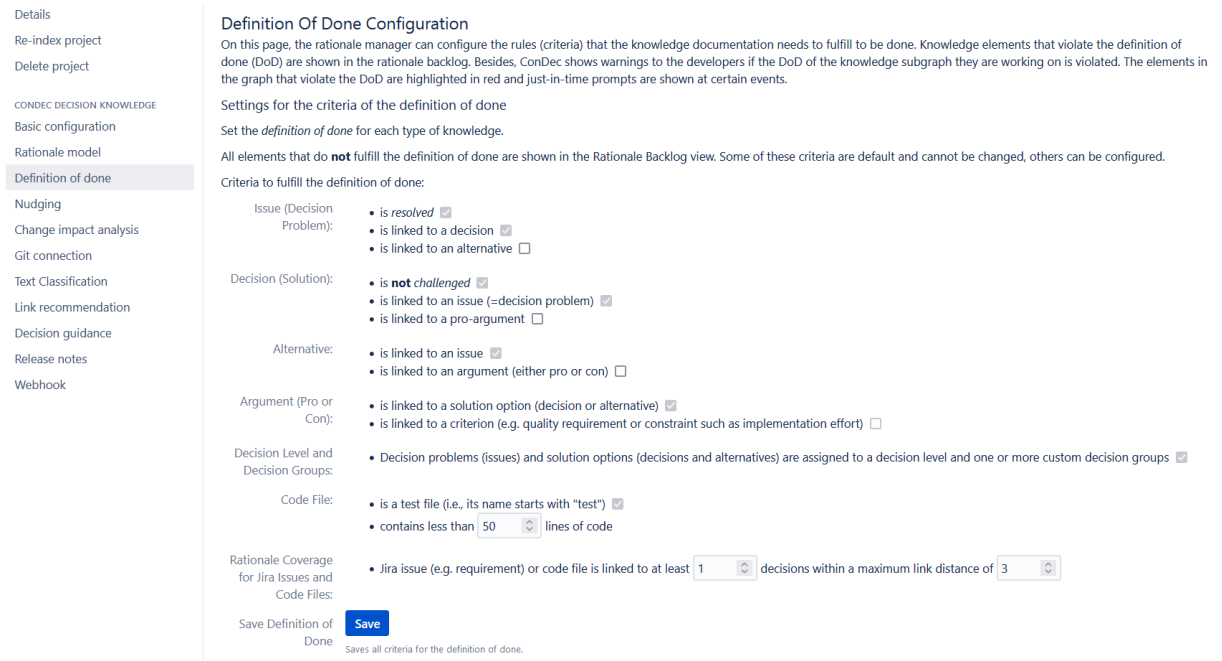

Figure 7.24.: Configuration view for the definition of done for the knowledge documentation.

7.6.5. Change Impact Analysis (RS2)

ConDec recommends knowledge elements impacted by a change. When changing a knowledge element, developers see which other elements need to be changed. The systematic mapping study (Chapter 4) contributed *consistency support* that goes beyond the integration of decisions with other artifacts and knowledge visualization. ConDec implements parts of this consistency support

in the recommendation system for *change impact analysis*. ConDec focuses on lightweight decision capturing in informal communication channels and does not currently support integrating decisions with architectural component models. For this reason, ConDec does not support consistency between decisions and architectural component models through constraints as suggested by Lytra et al. (2015). *Change impact analysis* is "the activity of identifying what to modify to accomplish a change, or of identifying the potential consequences of a change" (Arnold and Bohner, 1993). ConDec recommends knowledge elements affected by a change and thus helps identify potential consequences of a change. The node-link diagram, tree, list, and matrix views (V1 – V4) can be used for change impact analysis since these knowledge graph views show linked knowledge elements. That means that the knowledge graph views support the developers to make new decisions consistent with the requirements, code, and former decisions. The knowledge element(s) selected by the developer form the *starting impact set*, i. e., the input for change impact analysis. The other knowledge elements in the knowledge graph view form the *estimated impact set*. Besides, ConDec colors the knowledge elements in these views according to their *estimated impact value*, i. e., to the likelihood that they are affected by a change in the starting impact set (Figure 7.25). Darker elements have a high estimated impact value, whereas bright elements have a low impact value. Developers can see an explanation for the impact value on each knowledge element via a tooltip.

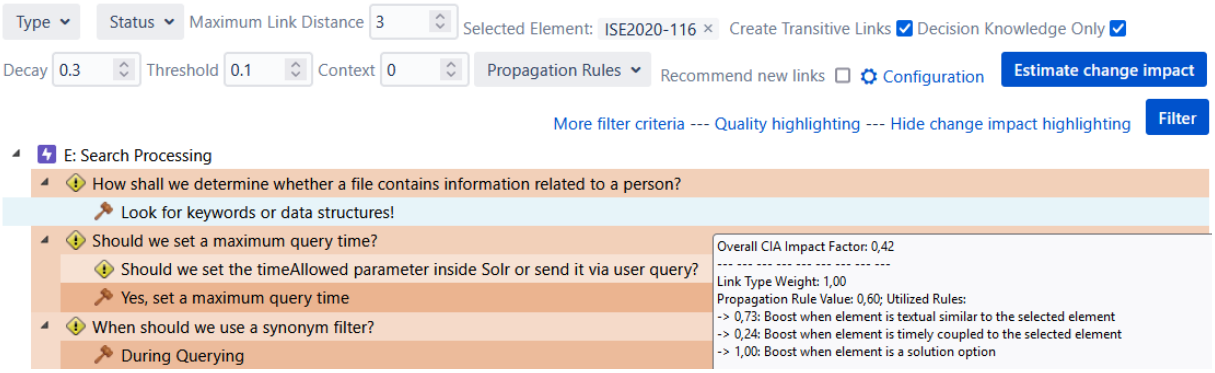

Figure 7.25.: Indented outline ($V2_{ind}$) with change impact highlighting. Issues and decisions are shown that a change in the root element (starting impact set) might impact. The color indicates the likelihood of change impacts: A change probably impacts darker elements more. The tooltip explains the estimated impact value.

The change impact analysis is traceability- and rule-based. The rules are inspired by the rules of *automatic linking* identified in the systematic mapping study (Chapter 4). For instance, a rule is that a higher impact value is assigned to the knowledge elements that are textually similar to those in the starting impact set. Another rule is that the relationship type between two knowledge elements influences the change propagation as inspired by Carrillo and Capilla (2018). For example, relationships with the type *enables* are stable, whereas *constrains* relationships are unstable. The following decisions of the ConDec project illustrate the different stability: *ConDec integrates the vis.js library!* enables *ConDec visualizes the knowledge graph as a node-link diagram!* If we changed the library decision from *vis.js* to *D3.js*, the enabled decision would be unaffected since ConDec could still visualize node-link diagrams. In contrast, *constrains* relationships are unstable, as illustrated by the following decisions: *ConDec is implemented with the JavaScript programming language!* constrains *ConDec integrates JavaScript visualization libraries, such as vis.js!* If we decided on a different programming language, the second decision would be affected since ConDec could no longer integrate JavaScript libraries. Thus, the estimated impact value is higher for *constrains* than for *enables* relationships.

7.6.6. Decision Guidance (RS3)

ConDec recommends solution options for decision problems from external knowledge sources. Anchoring means that developers give undue weight to experiences, information, or ideas and may restrict them from arriving at a more appropriate design solution (Razavian et al., 2023). The decision guidance recommendation system addresses the *anchoring bias* by presenting developers with solution options that they might not have considered. External knowledge sources can be DBPedia (Bhat et al., 2017a) or the decision knowledge documentation of other software development projects (Zimmermann et al., 2015). Figure 7.20 shows the decision guidance view with three recommendations generated from DBPedia for the decision problem *Which framework should we use for the web crawler?* The developers can accept or discard the recommendations. ConDec adds the solution option and arguments to the knowledge graph if they accept a recommendation. The score represents the predicted relevance of how likely the developers are to accept the recommendation and is used for ranking.

7.6.7. Link Recommendation and Duplicate Detection (RS4)

ConDec recommends new links between knowledge elements and tries to identify duplicated knowledge elements. For example, ConDec recommends potentially relevant decisions for a requirement from the same project. The developers can accept or discard the recommendations. If they accept a recommendation, the respective knowledge elements get linked. ConDec offers a view for link recommendation and duplicate detection (Figure 7.26). Besides, developers see the link recommendations directly in the knowledge graph views, linked with the *recommended* link type. The example in Figure 7.26 shows two issues *In which file format do we export the team data?* and *In which file format do we export the player data?* that have the duplicated decision *We export it as a .csv file!* If the developers decide to support a different export format than .csv files for both team and player data they would need to change both decisions. To avoid inconsistency, the duplicated decisions must be merged into one decision.

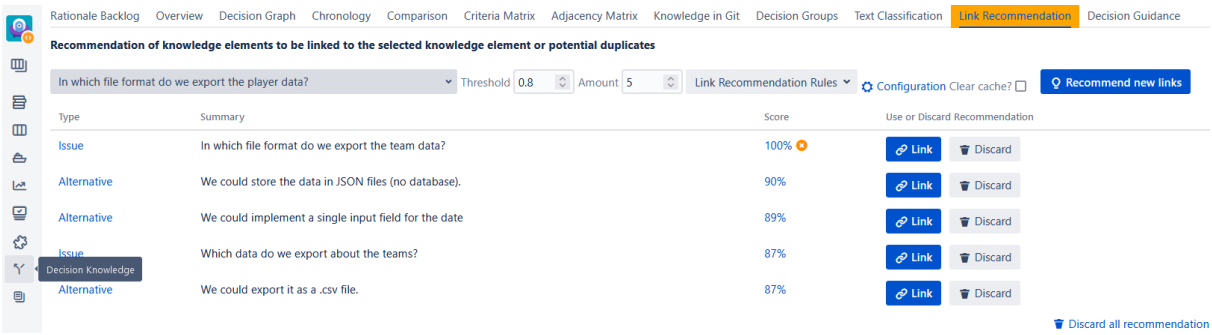

Figure 7.26.: Link recommendation view showing link recommendations and a potential duplicate marked with the x for a decision problem.

This feature identifies related knowledge elements using the context information of knowledge elements as done by Miesbauer and Weinreich (2012).

7.6.8. Automatic Text Classification (RS5)

ConDec predicts whether the textual parts in the Jira ticket description, comments, or commit messages are relevant decision knowledge elements, and—if yes—it annotates the parts accordingly. Automatic text classification generates *extractive summaries* of the text that developers wrote by identifying the decision knowledge in the text. It supports the developers in formalizing informally captured decision knowledge.

ConDec implements the respective approach identified in Chapter 4. Section 10.3 will compare the effectiveness of work on automatic text classification. ConDec's automatic text classification is mainly inspired by Automated Rationale ExtrAction from Communication arTifacts (A-REACT) by Alkadhi (2018), which has been shown to be effective for developers' conversations.

This section describes the functionalities for the automatic identification of rationale and their integration into the development process. Then, it introduces the ground truth used for training and evaluation, the preprocessing, the classifiers, and implementation details. Lastly, it illustrates the usage of ConDec's automatic text classification in a scenario.

Functionalities of ConDec's Automatic Text Classification to Identify Rationale

The following paragraphs describe the functionalities of ConDec's automatic text classification.

Automatic Annotation *The classifier predicts whether the textual parts in the Jira ticket description, comments, or commit messages are relevant decision knowledge elements, and—if yes—it annotates the parts accordingly.* The classifier supports developers in making decision knowledge elements explicit, i. e., it automates the annotation. ConDec automatically annotates the text of the Jira ticket description and comments when developers save textual changes. ConDec does not automatically classify code comments because they are unlikely to be used as locations for natural language discussions. Automatic annotation is part of daily development. Besides, it can be applied retrospectively to existing projects.

Easy Training *Developers train the classifier on training data from other projects and the decision knowledge documented for their current development project.* The text classifier is supervised, i. e., it needs training data to learn how to make predictions. ConDec provides a default training file to classify text of new or existing projects without explicit rationale management. The training is performed in Jira. Online training is possible, i. e., newly documented knowledge elements are directly used for training. Besides, developers can create training data files from their development projects and import and export them. The preprocessing is automatically performed.

Easy Evaluation *Developers evaluate the classifier on the data of their current development project or other projects.* The evaluation is directly possible in Jira. The training data can also be used for evaluation but split via k-fold cross validation.

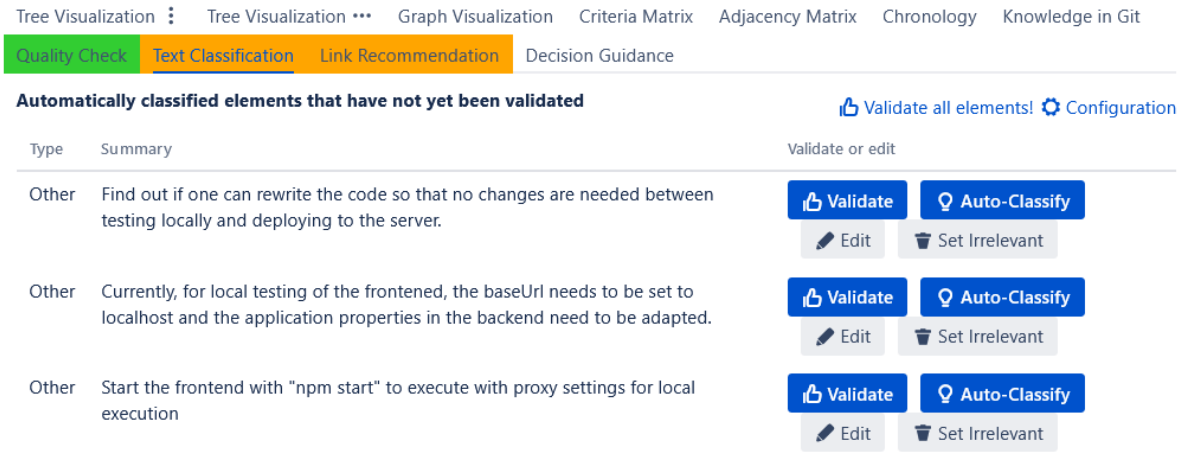

Figure 7.27.: Text classification view with three sentences not yet manually approved/validated.

Manual Approval *Developers manually approve the classification result, i. e., developers decide whether the annotations are correct.* Developers validate the classification result, edit it, or set a text part that ConDec wrongly classified as decision knowledge (false positive) as irrelevant. For this purpose, ConDec offers a view for text classification (Figure 7.27). The text classification view shows whether there are recommendations regarding the text classification to be approved by the developers. The manual approval of classified text parts is important to determine which text parts can be used as training and evaluation data, i. e., as the ground truth. Only the text parts correctly annotated or identified as irrelevant are used as the ground truth.

Ground Truth Data

The *ground truth*, also called *gold standard*, consists of text parts with correct annotations. It is required for training and evaluating the text classifier. The text parts are either relevant decision knowledge elements or irrelevant. Each text part is associated with a binary and a fine-grained label for its relevance and decision knowledge type, respectively. Five types are distinguished: issue, decision, alternative, pro-, and con-argument. As one preprocessing step for the classification is sentence splitting, the text classifier predicts new classifications (labels) for single sentences. However, the developer could manually annotate sub-sentences or multiple sentences as one decision knowledge element. Then, the respective training data entry, i. e., the text part, consists of a sub-sentence or multiple sentences. Table 7.7 shows a subset of the default training data in the ConDec Jira plug-in.

Table 7.7.: Binary and fine-grained labeled text parts in ConDec's default training data.

Binary Label	Fine-grained Label	Text Part (Sentence)
Irrelevant	-	Here's a small screenshot of the current state.
Relevant	Issue (Decision Problem)	What configuration possibilities should users have?
Relevant	Decision (Solution)	We decided to allow the following configuration ...
Relevant	Alternative	The user could configure the classifier algorithm.
Relevant	Pro	This will improve the performance.
Relevant	Con	I think this may confuse the user.

The text parts in the training data are taken from Jira ticket text (description and comments), commit messages, and code comments. Code comments only contribute relevant decision knowledge elements. One decision problem in ConDec is how to deal with text parts from different documentation locations: Should ConDec use the same classification model for Jira ticket text and commit messages? ConDec treats the documentation locations equally for simplification. In the future, it needs to be evaluated whether two different classifiers for Jira ticket text and commit messages would perform better than one unique classifier.

Preprocessing

ConDec uses the following preprocessing steps: 1) sentence splitting, 2) conversion of sentences to lowercase, 3) tokenizing, 4) vector representation of words via Global Vectors for Word Representation (GloVe), 5) n-gram generation (with n=3). These preprocessing steps are commonly applied in Natural Language Processing (NLP) applications for requirements engineering (Zhao et al., 2020) except for step 4: GloVe aims to represent the semantics of words (Pennington et al., 2014) and is similar to Word2Vec (Zhao et al., 2020). The conversion of words into numerical vectors is necessary for the machine learning classifier to work with them. An alternative technique for converting words to vectors is Term Frequency-Inverse Document Frequency (TF-IDF),

as used in A-REACT by Alkadhi (2018). ConDec uses GloVe instead of TF-IDF because GloVe enables online learning more easily than TF-IDF. Before online learning, the changed ground truth would require the recalculation of TF-IDF, while GloVe is independent of the words in the ground truth. ConDec does not apply stemming and stop-word removal because the words in GloVe are not stemmed and because stop words might be important parts of rationale (Rogers et al., 2015; Alkadhi, 2018; Li et al., 2020).

Classifiers and ConDec Implementation

ConDec uses two classifiers in a row, as done by Alkadhi (2018): one for binary predictions and one for fine-grained predictions. If the binary classifier predicts a sentence as relevant, it serves as the input for the fine-grained classifier. ConDec integrates the Statistical Machine Intelligence and Learning Engine (SMILE) library because SMILE is implemented in Java and provides online-learning capabilities. For the preprocessing, ConDec uses the SMILE NLP component. ConDec offers a settings page for automatic text classification to train and evaluate the binary and fine-grained classifiers. ConDec allows to choose the machine learning algorithm, e. g., Logistic Regression, Naïve Bayes, and Support Vector Machine (SVM). For the fine-grained Logistic Regression classifier, ConDec implements Multinomial Logistic Regression. For the fine-grained Naïve Bayes and SVM classifiers, ConDec implements the one-versus-one strategy to reduce the problem of multiclass classification to multiple binary classification problems.

Example for Explicit Rationale Automatically Identified by ConDec

The following decision problem was solved for the implementation of the text classifier: *Which library should we choose to implement the classifier?* This decision problem could be documented in each of the four documentation locations in the issue tracking system or version control system (Section 7.3). Figure 7.28 shows that it is documented in a Jira ticket comment of the requirement

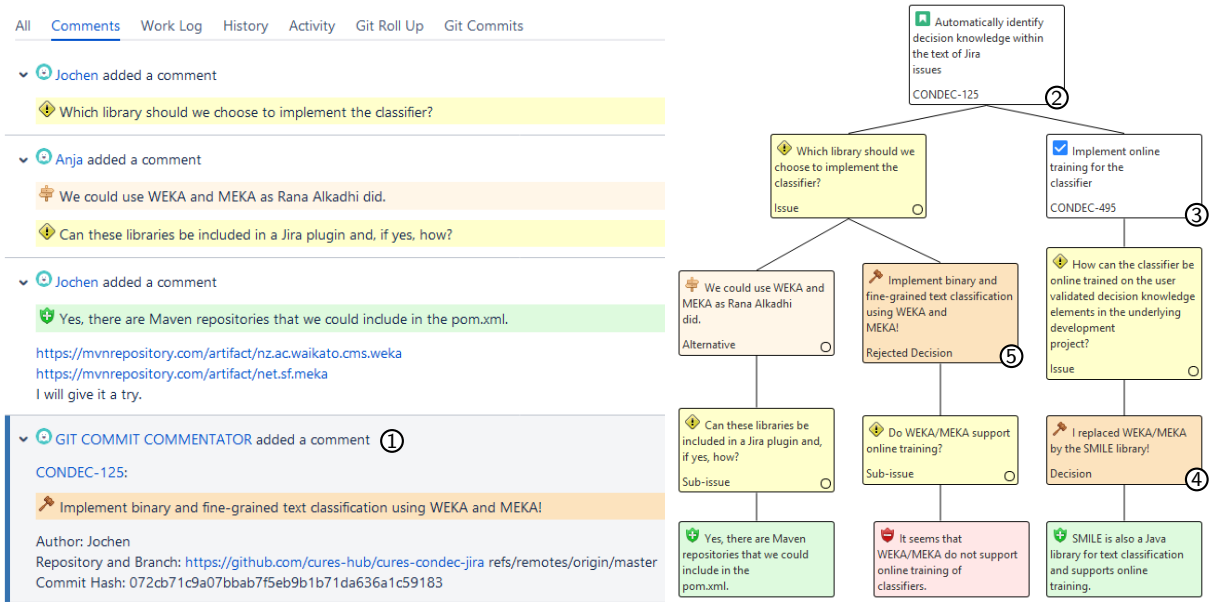

Figure 7.28.: **Left**: Decision knowledge captured in the comments of a requirement in Jira. ① is a commit message that ConDec automatically transcribed into the new comment. **Right**: Knowledge tree visualization ($V2_{nld}$) with the requirement ② being the root element. The subtree ③ on the right shows decision knowledge documented for a later development task. The decision ④ replaces the former decision ⑤.

Automatically identify decision knowledge within the text of Jira issues (Figure 7.28-②). The discussion on how to solve this decision problem is scattered in further comments and a commit message (Figure 7.28-①) for the requirement as well as in a linked development task (Figure 7.28-③). ConDec automatically classified, i. e., annotated the decision knowledge elements in the discussions, and the developers manually approved, i. e., validated them. The state of the first decision (Figure 7.28-⑤) is rejected as it was replaced by a new decision (Figure 7.28-④) documented during the implementation of the development task.

Figure 7.29 shows a different tree view than in Figure 7.28. The developers filtered this view to see decisions and their arguments concerning the requirement. ConDec supports feature F2 by enabling to replace filtered-out knowledge elements with transitive links (Section 7.5.2).

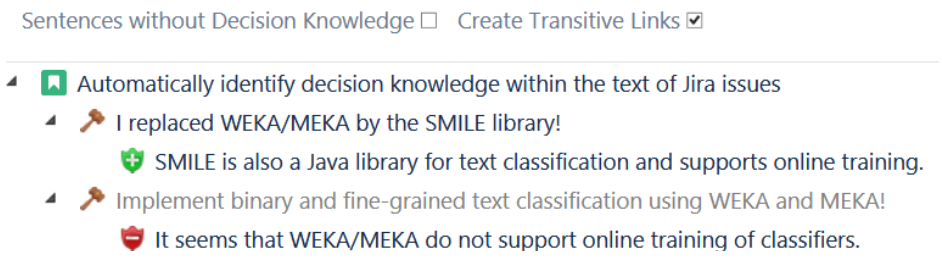

Figure 7.29.: Filtered knowledge tree visualization ($V2_{ind}$) for the requirement so that only the decisions and their arguments are shown. The rejected decision is colored in gray.

7.6.9. Summarization of Source Code Changes (RS6)

ConDec presents summarized source code changes to nudge developers in making tacit decisions explicit. Many decisions remain tacit. They are not captured anywhere but are already incorporated into the software. By presenting *abstractive summaries*, ConDec aims to trigger developers in making tacit decisions explicit, i. e., in reconstructing decision knowledge. When developers finish a ticket, ConDec posts a summary of source code changes into a new ticket comment. Besides, ConDec offers a view for the code summary related to a knowledge element developers can access throughout their work (Figure 7.30). ConDec extracts change sets for

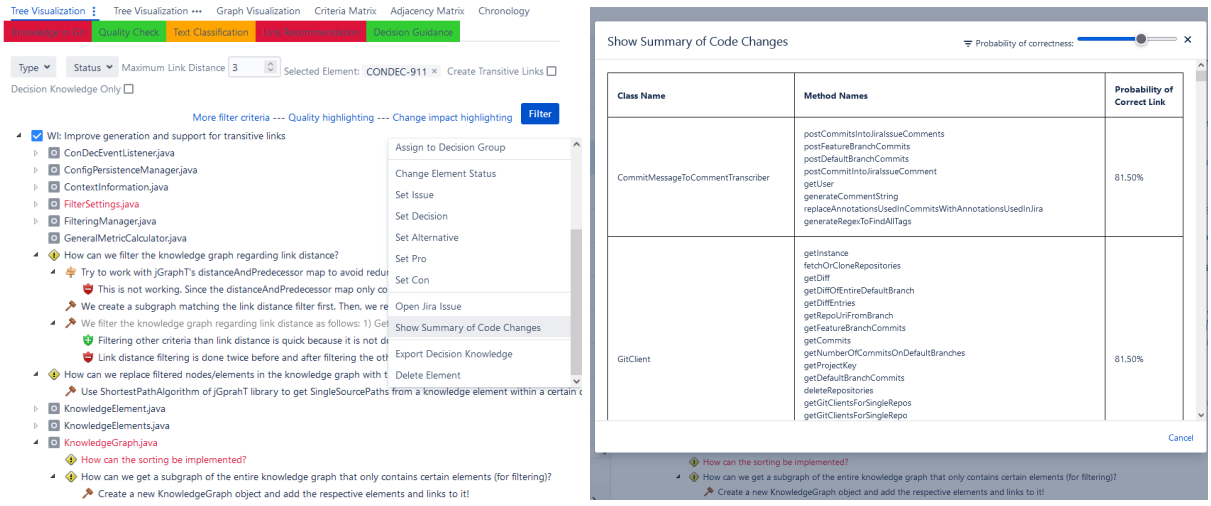

Figure 7.30.: **Left:** Context menu to access the code summary view showing correct linkage probability. **Right:** Code files linked to the Jira ticket with a probability that a file is correctly linked.

the commits linked to a feature task by comparing the code before and after the changes and lists the code files in the change set (Figure 7.30, right). These change sets are the basis for generating more advanced abstractive summaries, e. g., as suggested by Cortés-Coy et al. (2014).

To suggest meaningful abstractive summaries, it is essential that developers only commit small changes—instead of *code bombs* (Tao and Kim, 2015). CSE supports this since it involves techniques for work breakdown and encourages developers to commit changes often and merge branches (Krusche et al., 2014). In addition, code changes must be atomic (untangled) only to address the respective feature task. ConDec offers a mechanism to detect wrong links through tangled code changes, similar to Dias et al. (2015). ConDec support feature F3 by enabling the developers to mark links as wrong (Section 7.5.3).

7.7. Rationale Backlog

ConDec implements the concept of the *rationale backlog* described in Section 6.2.3. The rationale backlog contains the knowledge graph views with special filtering. It shows open decision problems for which a decision still needs to be made or documented, challenged decisions, and all knowledge elements that violate the definition of done. Figure 7.31 shows an example of the rationale backlog. The red text color indicates that action for improvement is needed as a nudging mechanism (N2). In the example in Figure 7.31, all knowledge elements violate the definition of done. However, there are cases in which the subgraphs on the right contain knowledge elements fulfilling the definition of done, which would have black text color. Next to

Figure 7.31.: Rationale backlog listing knowledge elements violating the definition of done on the left side. The right side shows the context of an issue in the node-link tree diagram ($V2_{nld}$).

rationale elements, ConDec also shows other knowledge elements, e. g., requirements, work items, and code files, that violate the definition of done in the rationale backlog. These other knowledge elements are part of the rationale backlog if they 1) fall below the minimum required decision coverage or 2) are linked to any other knowledge element that violates the definition of done. ConDec propagates definition of done violations to neighbor elements in the knowledge graph to increase the developers' awareness of quality problems. These are differences between the rationale backlog of ConDec to the decision backlogs suggested in the literature (Section 6.2.3).

7.8. Knowledge Dashboard

ConDec presents metrics calculated on the knowledge graph data structure in a knowledge dashboard. The rationale manager uses the metrics to evaluate the quality of documentation or to get an overview of the documented knowledge, and to assess the developers' engagement in documenting rationale. The dashboard consists of five dashboard items: Section 7.8.1 presents the rationale coverage. Section 7.8.2 presents the intra-rationale completeness. Section 7.8.3 provides other general metrics. Section 7.8.4 presents metrics on decision knowledge documented in the version control system. Section 7.8.5 provides the dashboard item for decision types. Section 7.8.6 describes the filtering and navigation functionality.

7.8.1. Dashboard Item for Rationale Coverage

The dashboard item for rationale coverage shows the coverage of requirements, code, and other software artifacts with issues and decisions as introduced in Section 6.2.1. For example, this dashboard item shows how many decisions are traceable from a requirement or a code file within a certain link distance in the knowledge graph (Figure 7.32).

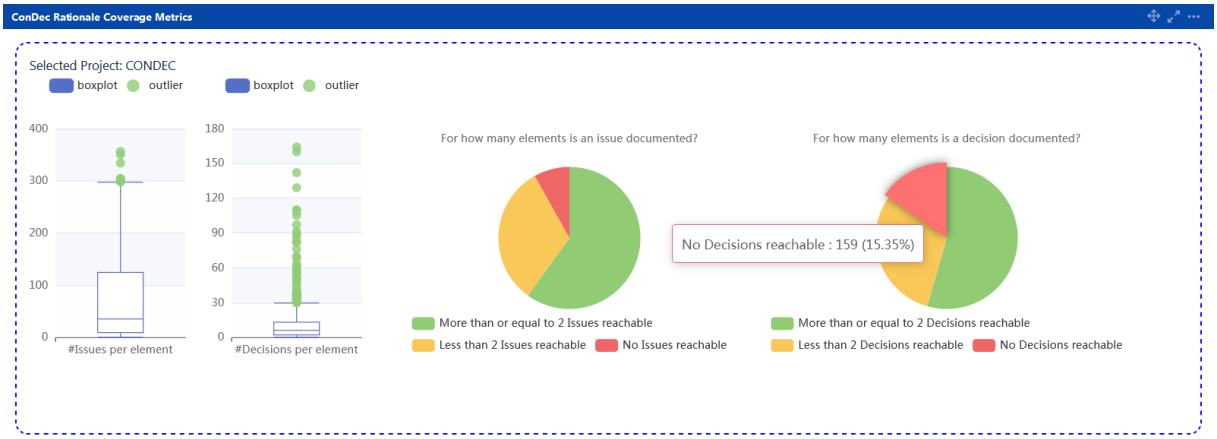

Figure 7.32.: Rationale coverage dashboard item showing the issue coverage and decision coverage using boxplots and pie charts.

7.8.2. Dashboard Item for Intra-Rationale Completeness

The dashboard item for intra-rationale completeness presents metrics regarding the intra-rationale completeness as introduced in Section 6.2.1 (Figure 7.33) to answer the following questions: a) How many issues are solved by a decision? How many issues are not solved by a decision? b) For how many decisions is the issue documented? For how many decisions is no issues documented? c) How many decisions have at least one pro-argument documented? How many decisions have no pro-arguments documented? d) How many decisions have at least one con-argument documented? How many decisions have no con-arguments documented? e) How many alternatives have at least one pro-argument documented? How many alternatives have no pro-arguments documented? f) How many alternatives have at least one con-argument documented? How many alternatives have no con-arguments documented? The metrics plots either help to find incomplete elements or are merely informative. Merely informative metrics are the number of decisions with and without con-arguments (d) and alternatives with and without pro-arguments (e).

Figure 7.33.: Intra-rationale completeness dashboard item showing the metrics using pie charts.

7.8.3. Dashboard Item for General Metrics

The dashboard item for general metrics gives an overview of the amount of knowledge and supports assessing the traceability between tickets and code and the fulfillment of the definition of done. The dashboard item presents the following metrics (Figure 7.34): a) Number of comments per Jira ticket. b) Number of commits per Jira ticket. c) Number of linked Jira tickets per code file (via commits with a ticket key in the commit message), number of unlinked code files that are not traceable from Jira tickets. The metric helps to find code files and tickets with missing trace links, which hinders knowledge exploitation. d) Number of Lines of Code (LOC) per code file and the total number of lines of code. e) Number of code files and requirements in the project. f) Number of rationale elements per documentation location. g) Number of comments with and without decision knowledge. h) Number of decision knowledge elements per decision knowledge

Figure 7.34.: General metrics dashboard item showing metrics using boxplots and pie charts.

type. i) Number of knowledge elements fulfilling and violating the definition of done. The metrics involving code files work on those files added to the knowledge graph. The rationale manager can configure which file types (e. g., java, js, ts, . . .) to include in the knowledge graph.

7.8.4. Dashboard Item for Metrics on Rationale in Code, Commits, and Branches

This dashboard item presents metrics on the knowledge documentation related to branches, code, and commits in the version control system (Figure 7.35). The dashboard item helps the rationale manager to monitor the rationale documentation on feature branches to ensure that only high-quality documentation is integrated into the mainline. It also gives an overview of the amount of knowledge documented in code and commits to assess the acceptance of these documentation locations. The following metrics are shown: a) Quality status of git branches: *incorrect* branches violate the definition of done, branches with the status *good* fulfill the definition of done, or branches having *no rationale* documented in code comments and commit messages. b) Quality problems in git branches to explain the definition of done violations, e. g., *Issue does not have a valid decision.* c) Number of issues in code comments and commit messages of a branch. d) Number of decisions in code comments and commit messages of a branch. e) Number of alternatives in code comments and commit messages of a branch. f) Number of pro-arguments in code comments and commit messages of a branch. g) Number of con-arguments in code comments and commit messages of a branch. h) Overview of Jira tickets related to git branches.

Figure 7.35.: Dashboard item showing metrics about the knowledge in git.

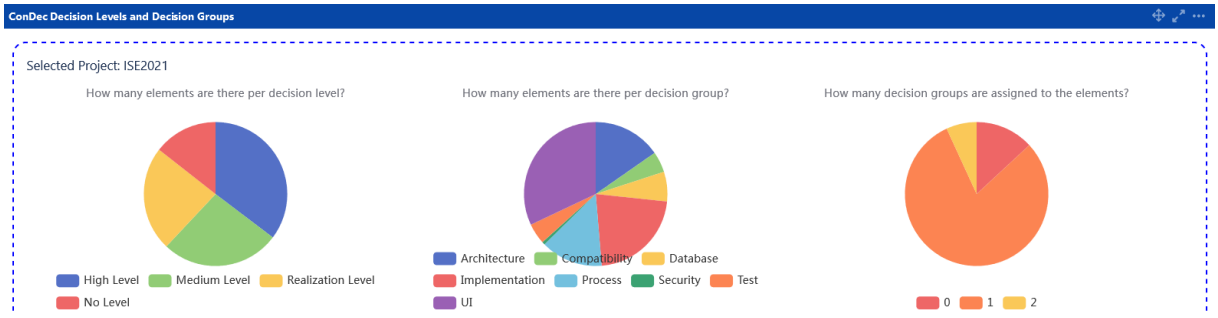

Figure 7.36.: Dashboard item showing metrics about the decision levels and decision groups.

7.8.5. Dashboard Item for Metrics about Decision Types

This dashboard item presents an overview of the decision types that the decisions documented within a project belong to (Figure 7.36) to answer the questions *Which types of decisions are documented in the project? How many decisions are documented per decision type?* The following metrics are shown: a) Number of knowledge elements per decision level. b) Number of knowledge elements per decision group. c) Number of decision groups assigned to the knowledge elements.

7.8.6. Filtering and Navigation from Knowledge Dashboard to Details

ConDec supports feature F1 by allowing filtering in the knowledge dashboard (Section 7.5.1). ConDec offers and persists filter settings for every dashboard item (Figure 7.37). They filter the knowledge graph underlying the metrics. For example, the status filter specifies whether issues are resolved or unresolved. Per default, resolved and unresolved issues are included. Similarly, the filter settings can be used to specify that the coverage of decisions should only include decisions with the status decided and no decisions with the status rejected or challenged.

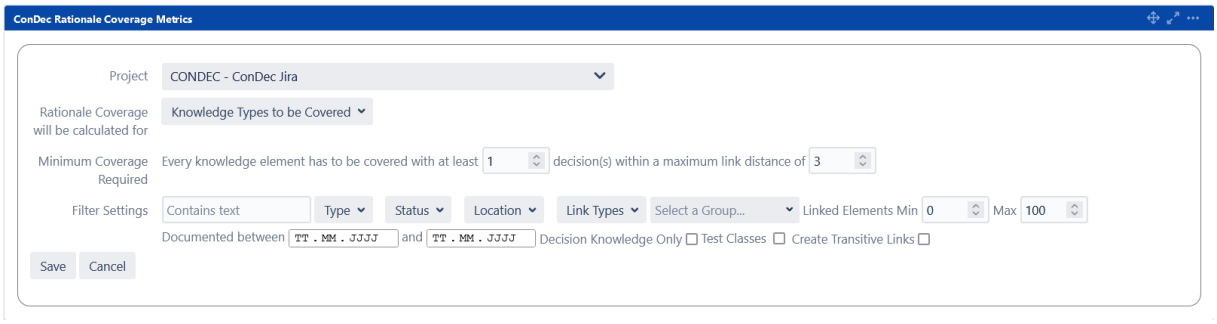

Figure 7.37.: Filter settings for rationale coverage dashboard item.

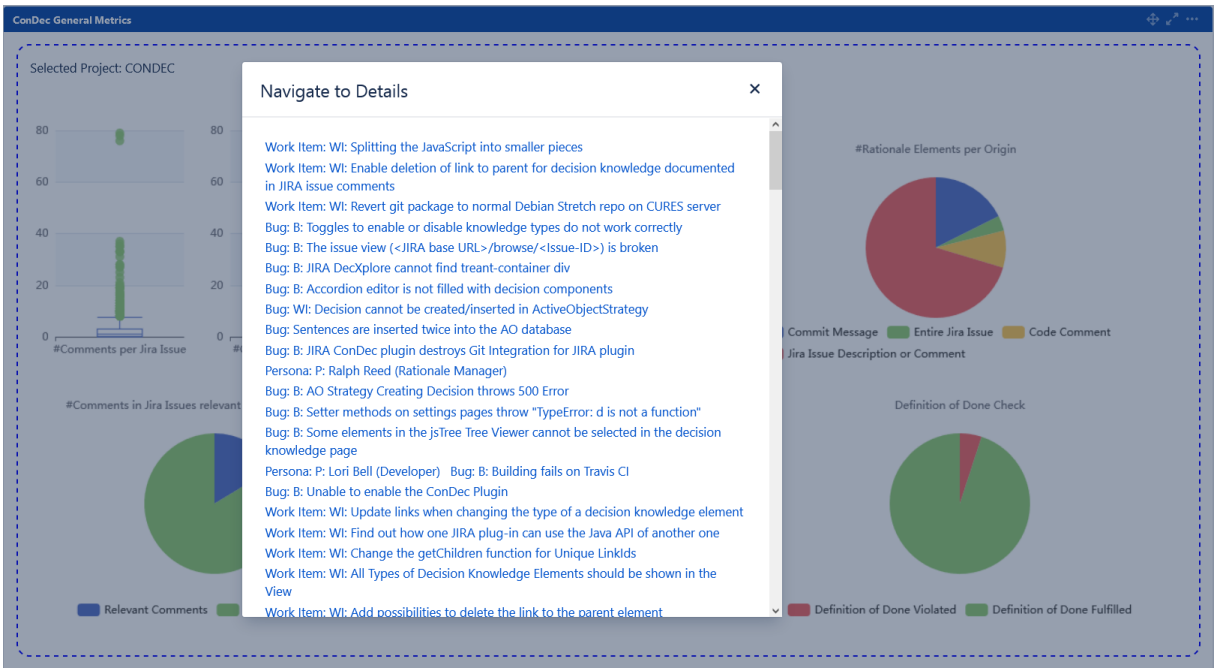

Figure 7.38.: Navigation dialog with elements violating the definition of done.

ConDec supports feature F5 by allowing developers to navigate to the detail views of the knowledge elements or git branches underlying the metrics (Section 7.5.5). There, developers can fix violations of the definition of done. The metric plots are interactive, and the developers can click on data points in the plots. When clicking on a plot, ConDec presents a dialog with hyperlinks to the detail views of the respective knowledge elements or git branches (Figure 7.38).

7.9. Decision Grouping

To enable targeted access to specific decisions, ConDec enables grouping decisions and related knowledge elements by one or more *decision types*, also called *decision groups*. Developers can group decisions according to predefined levels *high*, *medium*, and *realization* (Section 2.2.4). They can also define custom groups, such as *executive decisions*, *quality-driven decisions*, *functionality-driven decisions*, *user interface decisions*, and *testing decisions*. Chapter 9 of the thesis will introduce and explain a coding scheme for decision types and analyze the decision types used in practice. Note that the issue tracking system Jira offers a labeling mechanism, enabling grouping tickets. Differences between the decision grouping of ConDec and the Jira labeling mechanism are: 1) In ConDec, tickets and decision knowledge elements from various documentation locations, such as documented in comments and commit messages, can be gathered in decision groups. Developers can access e. g., a user-interface decision documented in code from a knowledge graph view in Jira through decision group filtering. 2) The group assignment is inherited within the decision knowledge. For example, if a developer assigns a decision to the decision group *process*, the linked decision problem and alternatives will inherit this group.

Section 7.9.1 describes how developers can assign and filter for decision types in ConDec and provides an overview of the decision types documented in a project. Section 7.9.2 details quality checking related to decision types.

7.9.1. Assignment, Filtering, and Overview

ConDec supports feature F3 by allowing developers to assign decision levels and custom groups using the context menu on a specific knowledge element in a knowledge graph view (Section 7.5.3). Figure 7.39 shows how a decision and the related rationale to apply a specific git workflow is classified as a *process* decision.

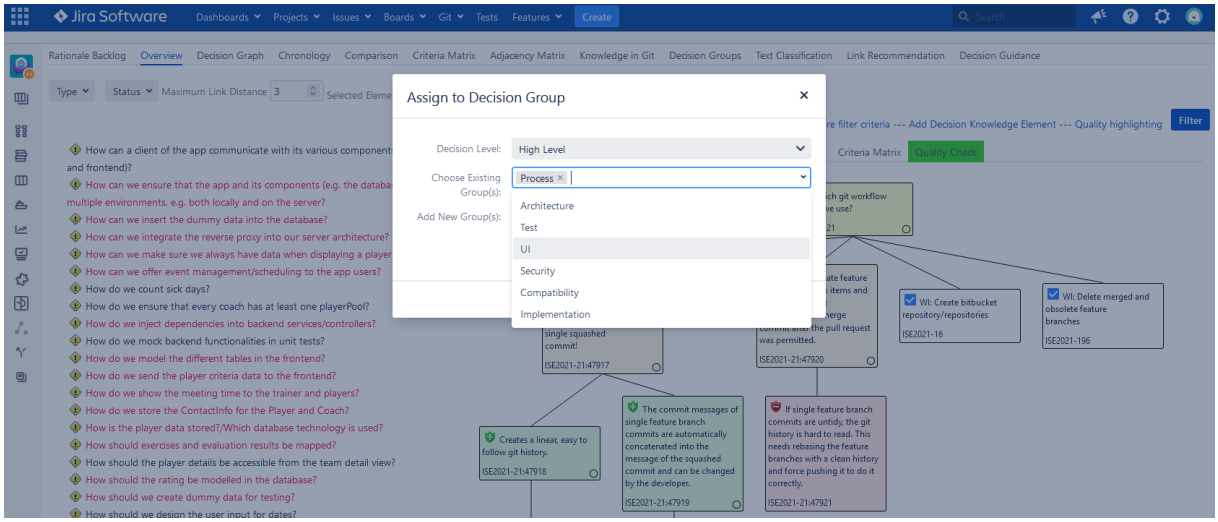

Figure 7.39.: ConDec dialog to assign a level and custom groups to a decision.

ConDec supports feature F1 by allowing developers to filter for decision levels and groups (Section 7.5.1). For example, developers can filter only to see decisions for the *user interface* or *process* decisions (Figure 7.40).

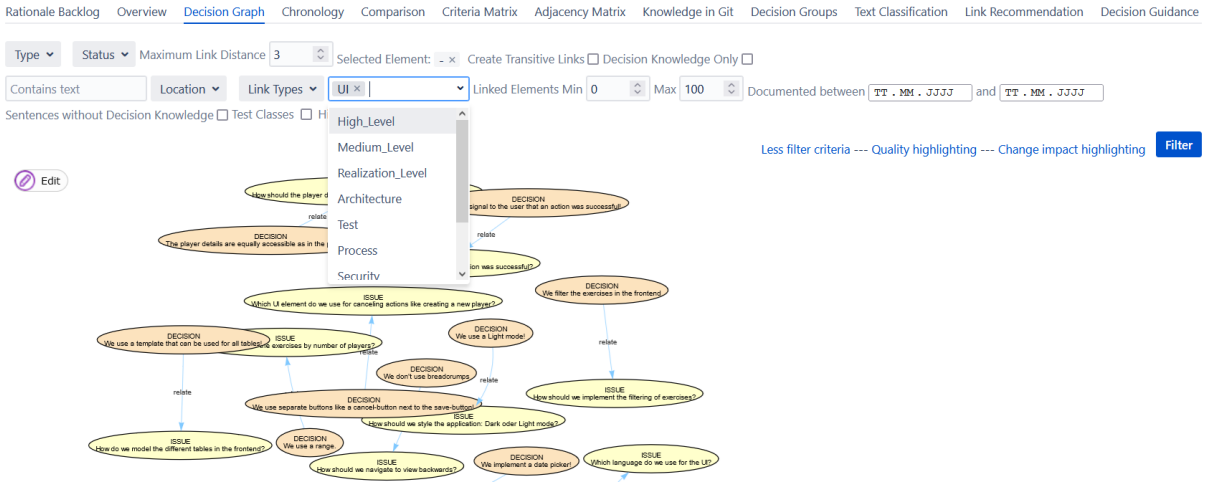

Figure 7.40.: Issues and decisions of *user interface (UI)* decision group in ISE 2021/22 project.

ConDec offers an overview of a project's decision levels and decision groups. In this overview, the developers can rename and delete decision groups via a context menu on group names (Figure 7.41). They can also navigate to the details of the decisions and change the decision group/level assignment via a context menu on the decisions. As described in Section 7.8.5, ConDec also offers a dashboard item for decision levels and groups.

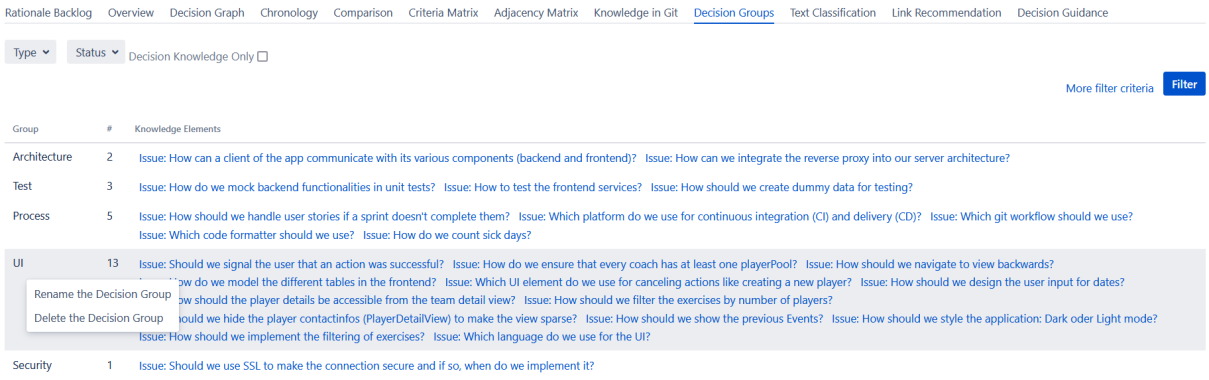

Figure 7.41.: Decision groups overview with context menu to rename or delete a group.

7.9.2. Decision Grouping as a Definition of Done Criterion

The rationale manager can enforce that developers assign decision problems and solution options to decision types by making the assignment a criterion of the definition of done. Suppose a decision problem or solution option is not assigned to a decision type. In that case, the quality check fails, and ConDec highlights the respective element in the knowledge graph views with red color as means of nudging (Section 7.6.4). The quality check view and the tooltip explain which criteria of the definition of done are fulfilled or violated (Figure 7.42).

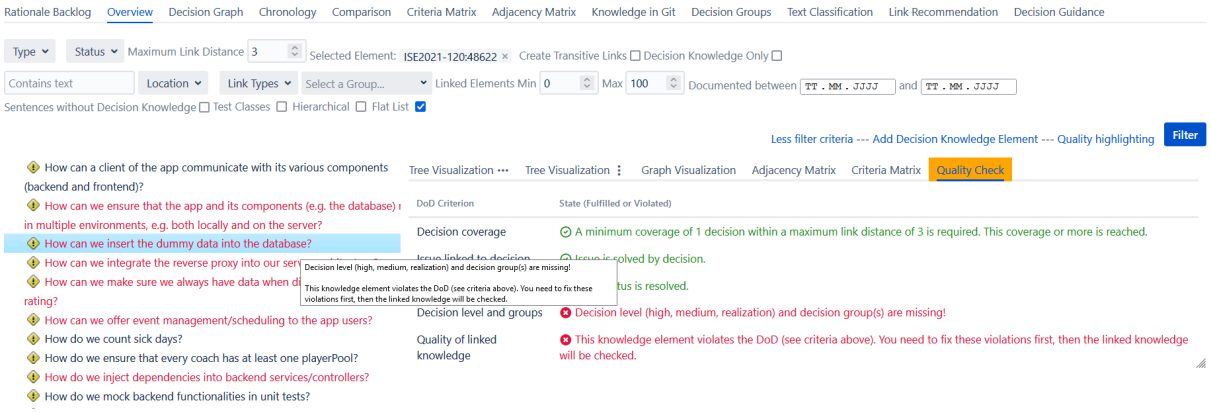

Figure 7.42.: Decision problem violating the definition of done since no decision level and no custom decision group is assigned to it. Developers can see an explanation for the definition of done violation in the quality check view and the tooltip.

7.10. Stand-up Table with Decision Knowledge

During stand-up meetings, sprint reviews, or other meetings, developers collaboratively discuss recently made decisions and decision problems that need to be solved. For this purpose, the ConDec Confluence plug-in enables integrating decision knowledge into meeting agendas. It enables the meeting manager to filter for decision knowledge to be shown in the meeting agenda. For example, it enables the meeting manager to filter for decision knowledge created and relevant for the last sprint. The developers get an overview of which issues need to be solved during the upcoming sprint and were solved during the last sprint (Figure 7.43).

Type	Summary
Issue	Which URLs shall be blacklisted and excluded from the search engine?
Decision	HeiBOX URLs shall be blacklisted!
Decision	University Library pages other than those containing general information shall be blacklisted!
Decision	GitLab servers shall be blacklisted!
Alternative	LSF URLs shall be blacklisted!
Alternative	Files marked as "disallow" in the "robot.txt" shall be blacklisted!
Alternative	URLs from public-law institutions such as Universitätsklinikum or Studierendenwerk shall be blacklisted!
Issue	How shall we show the subdomain to the user?
Alternative	Show the subdomain as-is!
Alternative	Replace the subdomain by the institute's name!

Figure 7.43.: Stand-up table (V3) as part of a meeting agenda. The open issue is highlighted to nudge the developers to make and document a decision collaboratively.

7.11. Release Notes with Decision Knowledge

The ConDec Jira plug-in supports the release manager in creating release notes including explicit rationale semi-automatically (Figure 7.44). The plug-in offers a view for release notes creation, management, and export. The release manager creates new release notes by choosing a time range—either manually or by selecting a sprint or release. The plug-in creates the release notes content including the decision knowledge elements linked to the Jira tickets. The release notes content is written in a markdown language so that it can be imported into other systems.

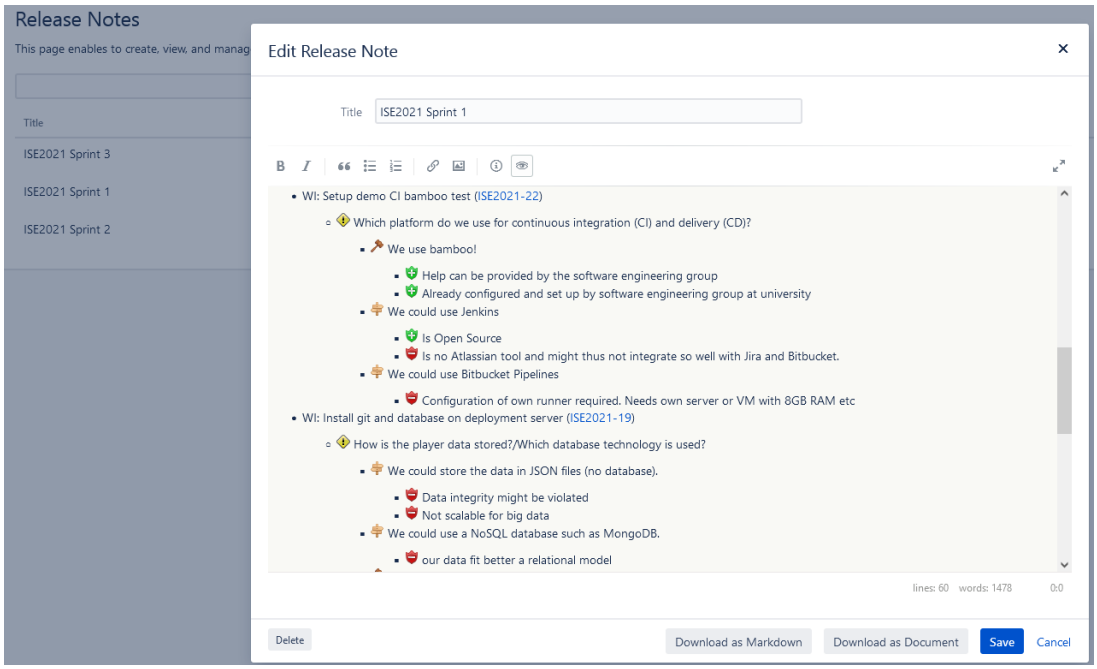

Figure 7.44.: Release notes editor with two work items finished during the sprint and the linked rationale documentation.

7.12. Knowledge Export

For knowledge sharing, ConDec enables exporting decision knowledge and related knowledge elements, such as requirements, code, and work items. Three export formats are supported: Word, JavaScript Object Notation (JSON), and Markdown. ConDec supports feature F1 by enabling to filter the knowledge to be exported using the same filter criteria as in the knowledge graph views, e. g., by knowledge type, status, and decision types (Section 7.5.1). The user chooses the export format in the export dialog (Figure 7.45). In addition to the manual export, ConDec enables the automatic knowledge export. For automatic export, ConDec provides a webhook sending new or changed decision knowledge to a receiver system via HTTP post requests.

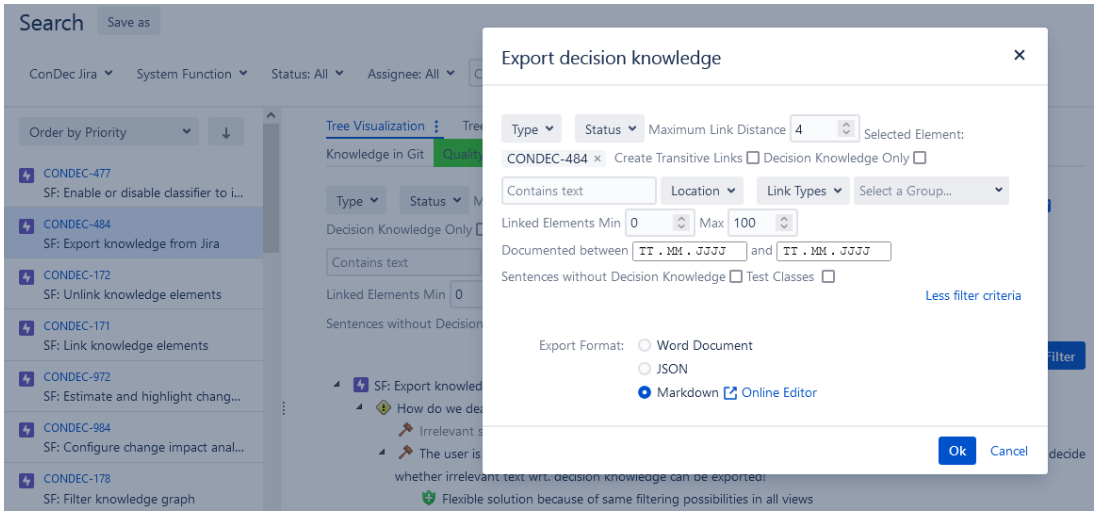

Figure 7.45.: Dialog to export the knowledge subgraph offering the same filter criteria as in the views on the knowledge graph.

7.13. Related Work

This section discusses related tools for the management of decision knowledge. It compares ConDec with tools found in the systematic mapping study in Chapter 4 and in the literature overviews by Hesse et al. (2016a), Capilla et al. (2016), Weinreich and Groher (2016), and Alexeeva et al. (2016). Many other tools focus on documenting architectural design decisions as they are difficult to change in the future. ConDec allows capturing and visualizing a wide range of decision types important to practitioners (Table 3.4), including design decisions, requirements elicitation and prioritization decisions, and decisions related to the development process.

ConDec is inspired by Unicase DecDoc (Hesse et al., 2016a; Hesse, 2020). Both tools, ConDec and DecDoc, support developers in documenting decision knowledge in a structured, collaborative, and incremental way based on the Decision Documentation Model. ConDec transfers concepts from the Eclipse-based research prototype Unicase DecDoc to other widely used tools, such as the issue tracking system Jira and the wiki Confluence, to fulfill the goal of low intrusiveness. DecDoc supports capturing distinct decision knowledge elements and implementation decisions as annotations in the code. ConDec offers features of DecDoc and more features, such as capturing decision knowledge in commit messages, recommendation systems, and nudging mechanisms. Thurimella et al. (2017) suggested customizing Jira to support rationale guidelines. The ConDec Jira plug-in is the first tool to explicitly capture decision knowledge in Jira.

The remainder of the section compares tool support treating the rationale management problems: Section 7.13.1 compares support treating the problem of intrusiveness and effort. Section 7.13.2 compares support treating the problem of the high amount of distributed knowledge. Section 7.13.3 compares support treating the problem of low documentation quality.

7.13.1. Tools for Low-Intrusive, Lightweight Rationale Management

This section discusses related tools *supporting rationale management activities with low intrusiveness*. To support low intrusiveness, tool support *integrates into standard development tools rather than providing a standalone tool*. For example, SEURAT (Burge and D. C. Brown, 2008b), Archie (Cleland-Huang et al., 2013), ADvISE (Lytra et al., 2015), and DecDoc (Hesse et al., 2016a) integrate into Eclipse. The Decision Architect (Manteuffel et al., 2015) and ADMentor (Zimmermann et al., 2015) integrate into Enterprise Architect. The ADeX tool by Bhat et al. (2019) retrieves decision knowledge from Jira, GitHub, Microsoft Project, and Enterprise Architect. ADeX uses the architecture knowledge management system AMELIE as the central repository and viewpoint for requirements, architectural elements, and decisions. ConDec uses Jira and git as repositories for decision knowledge, requirements, and code so that developers are not required to install a new tool and to enable *lightweight tracing* of the decision knowledge from and to requirements and code. Unlike other tools, ConDec integrates into multiple standard development tools and enables capturing decision knowledge in *various locations, such as ticket comments, commit messages, and code comments*. The developers either directly document decision knowledge in Jira or git or transcribe it there supported by ConDec, e. g., from chat messages. In addition, ConDec enables *access to the documentation from within the various tools* so that the developers do not have to change their development environment. *Decision annotations* represent a lightweight approach to mark text as decision knowledge. Hesse et al. (2015) use decision annotations to capture decision knowledge in code. Alkadhi et al. (2017a) capture decision knowledge in chat messages using annotations. ConDec also uses decision annotations in commit messages since empirical evidence exists that commit messages contain rationale (van der Ven and Bosch, 2013). ConDec combines annotations to important artifacts like code or commit messages with explicit decision models, as the former ease the capture and the latter eases the understanding of decisions.

Classification and recommendation approaches reduce the developers' effort to document decision knowledge. However, only a few approaches are implemented in development tools, which hinders their usage (Chapter 4). ConDec is the first tool integrating multiple *recommendation systems and nudging mechanisms* into Jira. While most existing approaches apply *automatic text classification* retrospectively on a ground truth that researchers created, ConDec integrates automatic text classification into the daily work in addition to using it retrospectively.

7.13.2. Tools Supporting a High Amount of Distributed Knowledge

This section discusses related tool support for *a high amount of distributed knowledge*. While no clear thresholds for what constitutes a high amount exist, documented knowledge tends to contain many knowledge elements and links. This section discusses related work on decision knowledge visualization as an essential feature for exploitation. Shahin et al. (2010) provide an overview of tools that support the visualization of architectural design decisions. Of the 58 approaches systematized by Alexeeva et al. (2016), five focus on the visualization of documented decisions. ConDec supports *targeted access to and comprehensive visualization of decision knowledge*. Table 7.8 compares the ConDec views with views of other tools.

The tool by L. Lee and Kruchten (2008) shows not only a list of elements (V3) but also a *list of relationships*. In ConDec, the adjacency matrix and the node-link diagram show relationships. While the change impact analysis view by L. Lee and Kruchten (2008) is separate, ConDec offers change impact highlighting as an option in the knowledge graph views.

The Architecture Design Decision Support System (ADDSS) and Decision Architect tools provide views for the *stakeholder involvement*. While ADDSS depicts different architecture views for various stakeholders (Capilla et al., 2010), the Decision Architect shows the actions of the stakeholders involved in the decision-making process (Manteuffel et al., 2015). ConDec enables filtering for decisions that specific stakeholders documented, and the chronology view (V5) groups the decisions documented by stakeholders.

The AMELIE Decision Explorer (ADeX) tool by Bhat et al. (2019) and Bhat (2020) offers three views currently not supported by ConDec: An *expert matrix view* depicting the expertise of architects and developers for architectural elements, an *expert recommender view* recommending experts for open issues, an *architectural elements view* visualizing the relevance of architectural elements in the project. ADeX shows a *quality attributes view* listing the number of decisions per quality attribute. ConDec's rationale coverage dashboard item (Section 7.8.1) is filterable to show the number of decisions per quality attribute.

SEURAT offers a *quality profile* showing the number of occurrences of a specific argument regarding software quality, which ConDec provides in the knowledge dashboard. SEURAT offers a requirements traceability matrix that shows solution options and code files linked to requirements. The ConDec users get the information when selecting a requirement in the node-link diagram, tree, list, or adjacency matrix view.

The ontology-driven visualization tool by de Boer et al. (2009) offers an *effect matrix* to display all solution options that affect a quality requirement. The ConDec users get the information when they select a quality requirement in either the node-link diagram, tree, list, or adjacency matrix view. The tool by de Boer et al. (2009) shows the overall negative, respectively positive effects of a solution option on all quality requirements, using scaled color bars. This helps the developers to make a decision and could be added to ConDec's criteria matrix (V4) in the future.

The tools DecDoc (Hesse, 2020), ADeX (Bhat et al., 2019) and ADDSS (Capilla et al., 2010) enable grouping decisions by their type (also called classification or category). On top, ConDec offers a comprehensive decision grouping feature that allows targeted access to and managing decision types (Section 7.9). It is the first tool to support *filtering (F1) in combination with transitive link visualization (F2)* to reduce the high amount of knowledge documentation.

Table 7.8.: Rationale management tools and their views.

Tool	Node-link, V1	Tree, V2	List, V3	Matrix, V4	Chronology, V5	Metrics, V6	Detail, V7	Other Views/Specific Purpose
ConDec	✓	✓	✓	✓	✓	✓	✓	rationale backlog, knowledge dashboard, change impact visualization, ...
Architectural Design Decision Ontology Tool (L. Lee and Kruchten, 2008)	✓	✗	✓	✗	✓	✗	✓	relationship list, change impact view
ADDSS (Capilla et al., 2006; Capilla et al., 2008; Capilla et al., 2010)	✓	?	✓	?	✓	✗	✓	stakeholder involvement
ADeX (Bhat et al., 2019; Bhat, 2020)	✗	✗	✓	✗	✗	✓	✓	quality attributes, expert matrix, expert recommender, architectural elements view
Compendium-based Architectural Design Decision Tool (Shahin et al., 2010; Shahin et al., 2011)	✓	✗	✗	✗	✗	✗	✓	
DecDoc (Hesse et al., 2016a; Hesse, 2020)	✓	✓	✗	✗	✗	✓	✓	
Decision Architect (Manteuffel et al., 2015) and ADMentor (Zimmermann et al., 2015)	✓	✓	✓	✓	✓	✗	✓	stakeholder involvement, decision backlog
Ontology-Driven Visualization Tool (de Boer et al., 2009)	✗	✓	✗	✓	✗	✗	?	effect matrix
SEURAT (Burge and D. C. Brown, 2008a; Burge and D. C. Brown, 2008b)	✗	✓	✗	✓	✗	✓	✓	quality profile, rationale task list, requirements traceability matrix

Visualizing rationale in context to other knowledge is closely related to traceability visualization. Filho and Zisman (2017) present the D3TraceView tool, which offers similar views as ConDec (matrix, list, and tree), but also a radial tree and sunburst view. Bacher et al. (2016) develop different tree visualizations for source code comprehension, in particular, a circular tree map and an icicle tree. Kugele and Antkowiak (2016) use the metaphor of an *impact city* to visualize results of change impact analysis. Unlike ConDec, these works do not focus on rationale. ConDec's views can also be used for general traceability visualization.

ConDec's views on the knowledge graph represent flexible building blocks tailorable for many purposes. ConDec adapts the views on the knowledge graph for specific purposes, such as for the rationale backlog, the knowledge dashboard, change impact analysis, a view on the knowledge in git, a stand-up table in meeting agendas, and release notes with rationale. It offers many *filter criteria to give targeted access* and cope with the high amount of documentation.

7.13.3. Tools Supporting High Documentation Quality

This section discusses related tools *supporting the high quality of the documented decision knowledge*. Of the 58 approaches systematized by Alexeeva et al. (2016), 21 aim to facilitate *traceability* between design artifacts by decision documentation. Twelve approaches aim to enable architecture *consistency* or compliance checks. For example, Lytra et al. (2015) present an approach using *architectural knowledge transformations* to support the consistency between design decisions and component models and code. In contrast, ConDec builds on *lightweight traceability and easily accessible knowledge visualization* for consistency.

The ADeX tool offers *decision guidance* with recommendations generated from DBPedia (Bhat et al., 2017a). The ADMentor by Zimmermann et al. (2015) is a plug-in for the Decision Architect,

which offers decision guidance and a decision backlog view (Table 7.8). ConDec is the first tool that integrates decision guidance into the issue-tracking system, allowing multiple knowledge sources and problem space models (knowledge documentation of other software development projects) to generate comprehensive recommendations.

Carrillo and Capilla (2018) present an algorithm for *change impact analysis* on decision graphs. ConDec extends this algorithm by adding further rules inspired by approaches for automatic linking (Buchgeher and Weinreich, 2011; Miesbauer and Weinreich, 2012). In addition, ConDec applies change impact analysis on knowledge graphs with decisions and other artifacts. ConDec also offers *link recommendation and duplicate detection* and enables performing change impact analysis on documented and recommended links.

SEURAT offers metrics to infer whether the rationale documentation is complete and produces warnings if it detects *incompleteness* in a rationale task list (Burge and D. C. Brown, 2008a; Burge and D. C. Brown, 2008b; Malloy and Burge, 2016). ConDec uses similar intra-rationale completeness metrics but newly introduced the *decision coverage* metric to support completeness and traceability between decisions and other artifacts. In addition to the rationale task list, ConDec's *rationale backlog* also shows requirements, code, and other artifacts that violate the definition of done, for example, since the decision coverage is too low or a linked issue is unresolved. Besides, new features of ConDec are the *definition of done checking* before finishing work (for example, as part of the *merge-check in pull requests*), the *knowledge dashboard*, and the *nudging mechanisms to motivate the developers to improve the documentation quality.*

7.14. Conclusion

This chapter presented the ConDec plug-ins that support continuous rationale management through views and features. First, it presented requirements by describing tasks of the roles involved in CSE, deriving tool support, and providing functional models. The requirements specification is complete in that it derives all existing ConDec features from rationale management activities and problems. Chapter 14 will discuss future work. Second, the chapter provided an overview of the design of ConDec, including decision problems and decisions. Third, it presented the views and features of ConDec and highlighted new aspects by discussing related work.

The contribution of ConDec for supporting CSE is the following. To *minimize intrusiveness and additional effort*, ConDec integrates into multiple standard development tools rather than providing a standalone tool. ConDec enables capturing decision knowledge using lightweight annotations in various documentation locations typical for CSE, in particular, ticket description and comments, commit messages, and code comments. To enable the exploitation of the *high amount of distributed knowledge*, ConDec provides comprehensive knowledge visualizations that developers can access from various tools and artifacts. ConDec offers various views on the knowledge graph (V1 – V7) and features to tailor the views for specific purposes (F1 – F5). These views are the building blocks for advanced features, such as the rationale backlog, knowledge dashboard, decision grouping feature, a stand-up table for meetings, and release notes with decision knowledge. ConDec implements concepts to *operationalize rationale quality.* It is the first tool that supports high-quality rationale documentation with a *definition of done* and the *decision coverage* metric that counts decisions traceable from requirements and code. ConDec offers a *rationale backlog* showing knowledge elements violating the definition of done. It offers a *knowledge dashboard* presenting the decision coverage and other metrics. ConDec reduces the developers' manual work and motivates them using *recommendation systems* (RS1 – RS6) and *nudging mechanisms* (N1 – N3). First, the *quality checking* recommendation system indicates violations of the definition of done with friction nudges and in just-in-time prompts. Second, *change impact analysis* highlights decisions and other artifacts affected by a change in the views on the knowledge graph to support consistent changes. Third, *decision guidance* recommends

solution options to decision problems from other software development projects and external knowledge sources. Fourth, the *link recommendation system* detects missing links and duplicates within the knowledge graph. Fifth, ConDec supports developers in explicitly capturing decision knowledge through *automatic text classification*. So far, automatic text classification has only been applied retrospectively. ConDec is the first rationale management tool that integrates automatic text classification into the ongoing development during CSE. Sixth, the *summarization of source code changes* helps to make tacit decisions explicit.

Part IV.

Treatment Validation

Chapter 8

Overview of Evaluation Studies

"The scientist builds in order to study. The engineer studies in order to build."

—Brooks, 1996

This part of the thesis 1) validates ConRat and the ConDec plug-ins and 2) answers empirical knowledge questions beyond the validation. This chapter provides an overview of the empirical studies. Section 8.1 describes six evaluation projects. Section 8.2 introduces the ConDec plug-ins and features applied in the projects. Section 8.3 gives an overview of the evaluation methods.

The treatment validation addresses this thesis's third, fourth, and fifth knowledge goals and the validation of the instrument design goal introduced in Section 1.4. Table 8.1 presents an overview of the empirical studies, their goals, the validation aspects, and whether the study contributes empirical knowledge beyond the validation, i. e., answers empirical knowledge questions, which do not depend on stakeholder goals.

Table 8.1.: Overview of the empirical studies in the following chapters, the goals they address, the validation aspects, and if they contribute empirical knowledge beyond the validation.

Chapter	Addressed Goal	Validation Aspects	Beyond Validation?
Analysis of Knowledge Documentation (Chapter 9)	Knowledge goal 3: Show that it is feasible to document a high amount of high-quality rationale during ConRat with the ConDec plug-ins. Describe the outcome of knowledge documentation in practice.	feasibility	✓
Effectiveness of Automatic Text Classification (Chapter 10)	Knowledge goal 4: Show the effectiveness of automatic text classification from the researchers' perspective.	effectiveness	✗
User Acceptance of ConDec Plug-Ins (Chapter 11)	Knowledge goal 5: Show the acceptance of the ConDec plug-ins from the software practitioners' perspective.	user acceptance	✗
Dissemination of ConRat and ConDec Plug-Ins (Chapter 12)	Instrument design goal: Disseminate ConRat and the ConDec plug-ins to developers and show the acceptance of the dissemination.	user acceptance	✗

Chapter 9 addresses the knowledge goal 3 of this thesis: *Show that it is feasible to document a high amount of high-quality rationale during ConRat with the ConDec plug-ins. Describe the outcome of knowledge documentation in practice.* It validates feasibility and contributes empirical knowledge beyond the validation. Chapter 10 addresses the knowledge goal 4 of this

thesis: *Show the effectiveness of automatic text classification from the researchers' perspective.* Chapter 11 addresses the knowledge goal 5 of this thesis: *Show the acceptance of the ConDec plug-ins from the software practitioners' perspective.* Chapter 12 addresses the instrument design goal: *Disseminate ConRat and the ConDec plug-ins to developers and show the acceptance of the dissemination.* While the dissemination was necessary for introducing ConRat and its support through ConDec in the case studies, it is the last chapter of this part of the thesis because the other chapters contribute more empirical knowledge.

8.1. Evaluation Projects

Table 8.2 provides an overview of the case study projects. We applied ConRat supported through the ConDec views and features in five projects with industrial settings and while developing the ConDec plug-ins. The following sections describe the projects. Section 8.1.1 describes two iPraktikum projects, Section 8.1.2 describes three Information Systems Engineering (ISE) projects, and Section 8.1.3 describes the ConDec project.

Table 8.2.: Characteristics of the evaluation projects: developed product, customer, number of developers (#Dev), and the time when the project took place.

	iPraktikum		**ISE 19/20**	**ISE 20/21**	**ISE 21/22**	**ConDec**
Product	Workplace Control App	Car Charging App	IoT Platform	Web Search Engine	Soccer App	ConDec Plug-Ins
Customer	Carnegie Mellon University	Company A from industry	Company B from industry	Heidelberg University	Voluntary soccer trainer	None (dissertation project)
#Dev	9	9	7	5	6	26 (over time)
Time	10/2018 – 02/2019		10/2019 – 03/2020	10/2020 – 03/2021	10/2021 – 03/2022	since 2016

8.1.1. iPraktikum

The iPraktikum is a multi-project course at the Technical University of Munich in which up to 100 students work in eight to ten teams on real problems provided by an industry customer (Bruegge et al., 2015). In particular, in the first half of the semester, the practical character of the course is supported by theoretical yet interactive lectures. During these lectures, the students learn the basic concepts of agile development, release and merge management, modeling, and usability engineering. For the iPraktikum in the winter semester of 2018/2019, the lecture on rationale management described in Chapter 12 was added. After the rationale management lecture, which all students participating in iPraktikum attended, two teams performed ConRat and applied ConDec views and features in their projects in the winter semester of 2018/2019. We introduced the role of the rationale manager, who also had the tasks of the meeting manager. The rationale manager is responsible for checking and improving the rationale quality, i. e., they make sure that important elements are documented and consistent. Further, the rationale manager imports issues and decisions important for the last sprint into the meeting agenda in Confluence. They update and add rationale elements after the meeting in Jira. One student per team takes the role of the rationale manager. The role is passed on after a week to a different student, i. e., it is an interchanging role. The following subsections describe the two projects.

Workplace Control App

The first project dealt with smart device management for an intelligent workplace at Carnegie Mellon University. The customer was a professor from Carnegie Mellon University. Nine students participated in the project. The goal was to create a mobile app that enables workplace users to control smart devices, e. g., lamps, in this workplace. The app users can aim at a device with their smartphone camera and control the device with a tap on their screen. This way, the users can manage a large number of devices intuitively. Besides, the mobile app tracks energy consumption to raise energy awareness and motivate users to reduce it. Energy reports can be analyzed within the app to raise awareness of energy consumption and identify opportunities to improve one's ecological footprint. High-level requirements of the *workplace-control app* are 1) track energy consumption, 2) utilize location awareness, 3) add new devices to an intelligent environment, and 4) discover devices using augmented reality. The project also contributed to the dissertation project by Henze (2020) called *Dynamically Scalable Fog Architectures*. The technology of *Fog Computing* was applied. Fog Computing enables to offload of computational-intense tasks from a mobile device with little computational power to near components with higher computational power. Components such as mobile devices can join and leave a so-called *Fog Architecture* at runtime. For example, Fog Computing can be beneficial for dynamic vehicles, smart grids, distributed sensor networks, and intelligent environments (Henze, 2020).

Car Charging App

The goal of the second project was to develop a mobile app that supports owners of electric cars to share access to charging stations. For example, the app suggests new charging stations in the user's area. High-level requirements of the *car charging app* are 1) charging station management, 2) intelligent home interaction, 3) gamification, and 4) intelligent scheduling. The customer was a company from the industry. Nine students participated in the project.

8.1.2. Information Systems Engineering Projects

The ISE project is an agile course at Heidelberg University, performed every winter semester. Like the iPraktikum, the students taking part in the ISE project work on real problems provided by a customer. We applied ConDec in three consecutive ISE projects during the winter semesters 2019/2020, 2020/2021, and 2021/2022. Each project started in October and finished in March of the following year. Each project had two block courses at the beginning and end of the project, lasting fourteen days, respectively. During these block courses, the students worked full-time on the project. Like in the iPraktikum, we introduced the students to rationale management and the ConDec plug-ins using the lecture on rationale management at the beginning of the project (Chapter 12). During the semester, the students worked part-time on the project. The project followed the Scrum process extended with ConRat, with sprints lasting from two weeks during the block courses to four weeks during the semester. During every sprint, one student took the role of the Scrum Master and was responsible for the sprint review held at the end of every sprint. At the same time, this student also took the role of the rationale manager. All other students took the role of developers. Process requirements regarding rationale management were (Section 6.2): 1) The rationale manager's task was to present explicit decision knowledge, i. e., decisions made and open decision problems during the sprint review. For that purpose, the developers should document decision knowledge during the sprint using ConDec. 2) The rationale backlog should only include unresolved decision problems at the end of the sprint but no other knowledge elements that violate the definition of done so that the documentation quality is high. 3) The students should include a stand-up table with decision knowledge into agendas and protocols of their meetings. They should create release notes with decision knowledge at the

end of every sprint. The students used the issue tracking system Jira to document requirements, work items (development tasks), and bug reports. They committed code to work items or to bug reports mentioning the respective ticket identifier in the commit messages. The following subsections describe the three projects.

ISE 2019/2020: IoT Platform

The first ISE project aimed to develop an Internet of Things (IoT) platform to manage IoT devices and their delivered data. The users of the IoT platform can get an overview of the devices and see the details of each device. They can create tags and master data to group the devices. The platform also includes an analytical service to detect device data anomalies. Seven students were involved in the project. The project's customer was a company from the industry.

ISE 2020/2021: Web Search Engine

The goal of the second ISE project was to develop a web search engine for the websites of Heidelberg University. Five students were involved in the project. The customer of the project was a professor from Heidelberg University.

ISE 2021/2022: Soccer App

The goal of the third ISE project was to develop a mobile app that supports the trainers of soccer teams in organizing training sessions and other events, choosing training exercises, and evaluating teams and players. In particular, the app should support voluntary trainers who did not receive a thorough education in training teams and who train children or amateur players rather than professional teams. Six students were involved in the project. The customer was a voluntary soccer trainer.

8.1.3. ConDec Project

While developing the ConDec plug-ins, we documented decision knowledge in Jira and git. That means that we applied the ConDec plug-ins during their development. In total, 26 developers were involved in developing ConDec between 2016 and 2022. The developers were students performing practicals, their bachelor's or master's thesis, or a part-time job. Thus, the time these developers contributed to the ConDec plug-ins was limited to a few months. The thesis author also worked as a developer in all ConDec projects. In particular, the following number of developers were involved in the respective ConDec plug-in development projects: 22 developers for ConDec Jira, three developers for ConDec Confluence, four developers for ConDec Eclipse, two developers for ConDec VSCode, two developers for ConDec Bitbucket, and three developers for ConDec Slack. Since the ConDec plug-ins are part of this dissertation project, various stakeholders exist, in particular, the thesis supervisors. However, there is no dedicated customer for ConDec. The ConDec plug-ins are available in Appendix A.

8.2. ConDec Plug-Ins and Features Applied in Evaluation Projects

Table 8.3 shows which ConDec plug-ins were applied in the validation projects. The ConDec Jira plug-in was applied in all projects but with a varying number and maturity of features (Table 8.4). The ConDec Confluence plug-in that supports rationale-based meeting management was applied in all projects starting from the ISE 19/20 project. During the iPraktikum 18/19, the students also conducted rationale-based meetings but created the stand-up table that lists relevant decision knowledge using a built-in import macro for Jira tickets. The ConDec Bitbucket plug-in that adds a view for decision knowledge in pull requests and the merge check was applied in the ISE 19/20 project. The other projects did use different git servers, namely Gitolite, GitHub, or GitLab, or—in the ISE 21/22 project—did not work with pull requests. The ConDec Slack plug-in enables capturing decision knowledge in chat messages and offers an information channel in that recent decision knowledge is posted. This plug-in was applied in the ISE 19/20 project. The ConDec plug-ins for integrated development environments that enable viewing decision knowledge for code were partly used in the ISE 21/22 and ConDec projects.

Table 8.3.: ConDec plug-ins applied in the validation projects

ConDec Plug-In	Underlying System	iPraktikum	ISE 19/20	ISE 20/21	ISE 21/22	ConDec Project
ConDec Jira	Issue tracking system	✓	✓	✓	✓	✓
ConDec Confluence	Wiki system	✗	✓	✓	✓	✓
ConDec Bitbucket	Version control system	✗	✓	✗	(✓)	✗
ConDec Slack	Chat system	✗	✓	✗	✗	✗
ConDec Eclipse and VSCode	Integrated development environment	✗	✗	✗	✓	✓

Table 8.4 shows which ConDec views and features were investigated in the validation projects. Some views and features support multiple rationale-management activities because they support different user sub-tasks specified in Section 7.1. The checkmarks in brackets (✓) denote that a view or feature was immature, i. e., only recently introduced or still developing, during the time of the project. The configuration features of ConDec were applied to set up ConRat, but we did not assess the study participants' attitudes toward these features. The same holds for the facilitate nudges, such as the opt-out policies for feature configuration. We validated the feasibility of the knowledge export feature by exporting the knowledge documentation of all validation projects retrospectively for the documentation analysis in Chapter 9.

Table 8.4.: Overview of the application of ConDec views and features in the validation projects and their support of the four course-grained rationale-management activities: decision making (DM), documentation (D), exploitation (E), and quality assurance (Q).

ConDec View or Feature	Description	iPraktikum	ISE 19/20	ISE 20/21	ISE 21/22	ConDec Project	Activity
Rationale Documentation in Various Locations in ...							
Entire Jira tickets	Section 7.3.1	✓	✓	✓	✓	✓	**D**, DM
Description and comments of tickets	Section 7.3.2	✗	✓	✓	✓	✓	**D**, DM
Commit messages	Section 7.3.3	✗	(✓)	(✓)	✓	✓	**D**, DM
Code comments	Section 7.3.4	✗	(✓)	(✓)	✓	✓	**D**, DM
Chat messages	Section 7.3.5	✗	✓	✗	✗	✗	**D**, DM
Views on the Knowledge Graph							
Node-link diagram, V1	Section 7.4.1	✗	✓	✓	✓	✓	E
Tree, V2	Section 7.4.2	✓	✓	✓	✓	✓	E
View for rationale in pull requests, V3	Section 7.4.3	✗	✓	✗	✗	✗	**E**, Q
View for rationale from git in Jira, V3	Section 7.4.3	✗	✗	✗	✓	✓	**E**, Q
Adjacency matrix, $V4_{adj}$	Section 7.4.4	✗	✗	✓	✓	✓	E
Criteria matrix, $V4_{cri}$	Section 7.4.4	✗	✗	✓	✓	✓	**E**, DM
Chronology, V5	Section 7.4.5	✗	✓	✓	✓	✓	E
Features of the Knowledge Graph Views							
Filtering, F1	Section 7.5.1	(✓)	(✓)	(✓)	✓	✓	E
Transitive linking, F2	Section 7.5.2	✗	✗	(✓)	✓	✓	E
Changing elements and links, F3	Section 7.5.3	(✓)	(✓)	✓	✓	✓	D
Linking arguments to criteria in criteria matrix, F3	Section 7.4.4, Section 7.5.3	✗	(✓)	(✓)	✓	✓	**D**, DM
Marking links as wrong or useless, F3	Section 7.5.3	✗	✗	✗	✓	✓	Q
Integrated navigation, F5	Section 7.5.5	✗	(✓)	(✓)	✓	✓	E
Nudging Mechanisms and Recommendation Systems							
Facilitate nudges, N1	Section 7.6.1	*internally validated*					D
Ambient feedback and friction nudges, N2	Section 7.6.2	✗	✗	✗	✓	✓	**Q**, DM
Just-in-time prompts, N3	Section 7.6.3	✗	✗	✗	✓	✓	Q
Quality checking, RS1	Section 7.6.4	✗	✗	✗	✓	✓	Q
Change impact analysis, RS2	Section 7.6.5	✗	✗	✗	✓	✓	E
Decision guidance, RS3	Section 7.6.6	✗	✗	✗	✓	✓	**D**, DM
Link recommendation and duplicate detection, RS4	Section 7.6.7	✗	✗	✗	✓	✓	D
Automatic text classification, RS5	Section 7.6.8	✗	✓	✓	✓	✓	D
Summarization of source code changes, RS6	Section 7.6.9	✗	✗	(✓)	(✓)	(✓)	**D**, Q
Other Views and Features							
Rationale backlog	Section 7.7	✗	✗	✓	✓	✓	**Q**, DM
Knowledge dashboard, V6	Section 7.8	✗	(✓)	✓	✓	✓	Q
Decision grouping	Section 7.9	✗	(✓)	(✓)	✓	✓	D
Stand-up table with rationale, V3	Section 7.10	(✓)	✓	✓	✓	✓	**E**, DM
Release notes with rationale, V3	Section 7.11	✗	✓	✓	✓	✓	E
Knowledge export	Section 7.12	*validated through retrospective export*					E
Configuration	Table 7.5	*internally validated*					–

8.3. Evaluation Methods

Table 8.5 provides an overview of the evaluation methods applied during the six case studies. Since the same evaluation methods in both iPraktikum projects were used, Table 8.5 contains one column for the iPraktikum. The evaluation covers four types of studies: 1) field experiment, 2) sample study, 3) laboratory experiment, and 4) judgment study. The six case studies were *field experiments* in that the study participants applied ConRat and ConDec. We observed the study participants, collected and discussed feedback, e. g., in meetings, email, and chat, and made field notes. During the last two ISE projects, we logged the usage of the different ConDec views on the knowledge graph to compare their acceptance. In addition, we collected qualitative feedback systematically with surveys and semi-structured interviews. The thesis author conducted semi-structured interviews in two ISE projects. Before the interviews, the study participants were asked to fill out a survey questionnaire. The results of surveys and interviews are part of the *judgment study*. We mined and analyzed the knowledge documentation of the six case studies in a *sample study*. Besides, we used the knowledge documentation to assess the effectiveness of ConDec's automatic text classification in a *laboratory experiment*. The following chapters will detail the study design.

Table 8.5.: Overview of evaluation methods used in the case study projects and the respective study type classification according to Stol and Fitzgerald (2018).

Evaluation Method	**iPraktikum**	**ISE 19/20**	**ISE 20/21**	**ISE 21/22**	**ConDec Project**	**Study Type**
Observation, informal discussions, and field notes	✓	✓	✓	✓	✓	Field experiment
Usage logging	✗	✗	✓	✓	✗	
Survey	✓	(✓)	✗	(✓)	✗	Judgment study
Semi-structured interview	✗	✓	✗	✓	✗	
Mining and analysis of knowledge documentation	✓	✓	✓	✓	✓	Sample study
Usage of knowledge documentation for text classification evaluation	✓	✓	✓	✓	✓	Laboratory experiment

Chapter 9

Analysis of Knowledge Documentation

> *"We claim that architectural design decisions and architecturally significant requirements are really the same; they're only being observed from different directions. [...] In a way, architecturally significant requirements and architectural design decisions seem to accumulate in some kind of a 'magic well'. Observers peering into the well see what they wish for."*
>
> —de Boer and van Vliet, 2009

This chapter contributes to the knowledge goal 3 of this thesis: *Show that it is feasible to document a high amount of high-quality rationale during ConRat with the ConDec plug-ins. Describe the outcome of knowledge documentation in practice.* It validates the feasibility of the treatment through the rationale documentation that the software developers created in the six validation projects (Section 8.1). The chapter also adds to the empirically-gained knowledge regarding decision making and documentation in software development projects. The contribution is a comprehensive description of rationale documentation in relation to requirements and code produced by software developers in an industrial setting. In total, the developers of the six projects documented 957 issues, 1146 decisions, and 2165 arguments with the help of ConDec.

Section 9.1 describes the study design. Section 9.2 presents and discusses the results of the knowledge documentation analysis. Section 9.3 presents related empirical work on analyzing decision knowledge documented during software development projects. Section 9.4 discusses threats to validity. Section 9.5 concludes this chapter. Appendix A provides the data and analysis scripts. Appendix D describes the knowledge documentation of the six validation projects and provides additional material.

9.1. Study Design

Section 9.1.1 introduces three research questions. Section 9.1.2 describes the data acquisition. Section 9.1.3 provides metrics for the analysis of code in the version control system git and trace links to the issue tracking system Jira to investigate the connectivity of the knowledge documented in both systems. Section 9.1.4 describes the coding of decisions with types.

9.1.1. Research Questions

The knowledge goal 3 is refined into three research questions with sub-questions (Table 9.1). This section presents the questions, including the indicators for feasibility and the metrics to describe the outcome of knowledge documentation in practice.

Table 9.1.: Research questions and metrics of the empirical study on the rationale documentation.

	Research Question	Metrics
RQ1	Is it feasible to document decision knowledge in practice with ConDec?	
RQ1.1	Which and how many decision knowledge elements were documented and when were they documented?	Types and numbers of decision knowledge elements at the end of the project and over time
RQ1.2	Where were the decision knowledge elements documented, i. e. in which documentation locations, and when?	Numbers of decision knowledge elements per documentation location at the end of the project and over time
RQ1.3	How many rationale elements were updated after their first documentation?	Comparison of dates of creation and last update of decision knowledge elements
RQ1.4	What types of decisions are documented and when? Were specific types of decisions documented at specific points in time?	Numbers of decisions per decision type at the end of the project and over time
RQ1.5	Are decision types often or never assigned to the same decisions?	Correlation of decision types with other decision types
RQ1.6	Are there decision types that often or never are documented in specific documentation locations?	Correlation of decision types with documentation locations
RQ2	Is it feasible to document a high amount of knowledge in practice with ConDec?	
RQ2.1	Which and how many requirements and other tickets are documented?	Types and numbers of requirements and other tickets in the issue tracking system Jira
RQ2.2	Which and how many code elements were created?	Types and numbers of code elements in the knowledge graph, lines of code
RQ2.3	What are the ratios between decision knowledge and system knowledge elements?	Ratios between number of decisions to requirements and code
RQ3	Is it feasible to create high-quality knowledge documentation in practice with ConDec?	
RQ3.1	How well is the knowledge documented in git linked to the knowledge documented in Jira?	Proportion of commits with valid ticket identifier in their commit message, code elements linked to at least one ticket
RQ3.2	Is the decision knowledge completely documented?	Numbers of 1) issues with at least one alternative, 2) decisions with an issue, 3) decisions with at least one pro-argument, 4) alternatives with at least one con-argument
RQ3.3	What are the states of the issues and decisions?	Proportion of unsolved issues, proportion of rejected decisions
RQ3.4	How many decisions are traceable from requirements and code, i. e., what is the decision coverage?	Decision coverage calculated as the number of traceable decisions from requirements and code in a maximal link distance of 3

RQ1 Is it feasible to document decision knowledge in practice with ConDec?

The first research question is to show that ConDec supports decision knowledge documentation. The feasibility is indicated if developers documented decision knowledge during their projects with ConDec. The first research question consists of six sub-questions detailing the indicator:

RQ1.1 *Which and how many decision knowledge elements were documented and when were they documented?* The question asks for the number of elements per rationale type at the end

of the projects and over time. We aim to show that it is feasible to document rationale 1) of different types and 2) continuously, i. e., in small increments over time instead of only once.

RQ1.2 *Where were the decision knowledge elements documented, i. e. in which documentation locations, and when?* The question investigates the amount of rationale per documentation location at the end of the projects and over time. We aim to show that it is feasible to document rationale in the four documentation locations in the issue tracking and version control systems.

RQ1.3 *How many rationale elements were updated after their first documentation?* With this question, we aim to show that it is feasible to change the rationale elements with ConDec.

RQ1.4 *What types of decisions are documented and when? Were specific types of decisions documented at specific points in time?* The question asks for the decision types (Section 7.9). We aim to show that it is feasible to document various types of decisions with ConDec, i. e., not only architectural design decisions. In the state-of-practice study (Chapter 3), the practitioners frequently mentioned executive decisions as important. It is interesting to see whether executive decisions were frequently documented in the validation projects. We aim to investigate whether certain types of decisions were documented at specific points in time, e. g., executive decisions at the beginning of the project to set up the development process.

RQ1.5 *Are decision types often or never assigned to the same decisions?* The question asks for correlations among decision types. A decision can be assigned to more than one type. The correlation amongst decision types is interesting to reveal relations between the decision types in the data, e. g., whether testing decisions are often executive decisions, and to reveal disjunctive decision types.

RQ1.6 *Are there decision types that often or never are documented in specific documentation locations?* The question asks for correlations between decision types and documentation locations. Developers can interchangeably use the documentation locations in the issue tracking and version control system since ConDec integrates all decision knowledge in the knowledge graph. They can access the decision knowledge from requirements and code. Nevertheless, the correlation between decision types and documentation locations is interesting to reveal if certain types of decisions are often or never documented in certain places. For instance, executive decisions might not be documented in code comments because they do not concern a specific implementation.

RQ2 Is it feasible to document a high amount of knowledge in practice with ConDec?

The second research question is to show that ConDec supports a high amount of knowledge documentation (Section 5.2.2). Indicators for the high amount are the exceedance 1) of an absolute number of requirements and code elements and 2) of a relative number of rationale elements compared to requirements and code. Three sub-questions detail the indicators:

RQ2.1 – 2.2 *Which and how many requirements and other tickets are documented?* and *Which and how many code elements were created?* The questions aim to descriptively analyze the types and numbers of system knowledge and project knowledge elements, i. e., of the non-decision knowledge elements. This is interesting because requirements, other tickets, and code elements are part of ConDec's knowledge graph. There are no clear thresholds that the absolute number of requirements, lines of code, and rationale elements need to exceed to be considered high. We consider the numbers measured in the projects with a duration of about six months as high.

RQ2.3 *What are the ratios between decision knowledge and system knowledge elements?* With this question, we aim to show that it is feasible to document a high amount of rationale related to the requirements and code with ConDec. We assume that at least one decision, including the issue, alternative, and arguments, should be documented per requirement and code file as a threshold for a high number of decisions.

RQ3 Is it feasible to create high-quality knowledge documentation in practice with ConDec?

The third research question is to show that ConDec supports high-quality knowledge documentation (Section 5.2.3). Indicators of high quality are the 1) traceability between code in git and tickets in Jira, 2) completeness of decision knowledge, 3) low number of unsolved issues, and 4) accessability of decisions from requirements and code. Four sub-questions detail the indicators:

RQ3.1 *How well is the knowledge documented in git linked to the knowledge documented in Jira?* The question investigates the quality of trace links between code and tickets via commits to validate the knowledge model in Section 6.1. High-quality trace links between code and tickets are important for exploitation, e. g., for knowledge visualization and change impact analysis.

RQ3.2 *Is the decision knowledge completely documented?* The question quantifies the number of rationale elements that fulfill the criteria of the intra-rationale completeness (Section 6.2.1). This helps to assess how well others can understand the rationale documentation.

RQ3.3 *What are the states of the issues and decisions?* The question asks for the decision-making states (Section 6.1.2). A low number of unsolved issues indicates high quality since it complies with the agile principle of just enough work (Yang et al., 2019). In the state-of-practice study (Chapter 3), the practitioners mentioned that decisions could frequently be changed during CSE. Quantifying the amount of rejected decisions investigates this hypothesis.

RQ3.4 *How many decisions are traceable from requirements and code, i. e., what is the decision coverage?* With this question, we aim to determine how many decisions are documented in the context of requirements and code. The higher the decision coverage, the better the accessibility of the decision knowledge from tickets and code, but there can be wrong links.

9.1.2. Data Acquisition

Every validation project has a Jira project that we mined to obtain the data for the analysis. Besides, we mined the respective git repositories—if available—to acquire the code elements and the decision knowledge from commit messages and code comments. We used the views and features of the ConDec Jira plug-in for the mining, in particular, the knowledge dashboard (Section 7.8) as well as the filter functionality of the issue tracking system Jira. We exported the decision knowledge documentation of the validation projects via the export feature of the ConDec Jira plug-in. We created an R script for the data analysis (R Core Team, 2022). The decision knowledge data and the R script are available in the Appendix A.

9.1.3. Analysis of Code and Trace Links to Tickets

The metric $\#\text{Commits}_L$ counts the number of commits with a valid ticket identifier in their commit message so that the code changes bundled in the commit are linked to the respective ticket. The metric $\#\text{Code}_{graph}$ counts the number of code elements integrated into the knowledge graph. ConDec only integrates code files of selected file types into the knowledge graph to exclude files without decision knowledge elements in comments, such as binary files (e. g., jar).

The project participants selected the types of code files to be integrated into the knowledge graph. For example, they excluded particular files with little abundance, such as shell scripts. The metric $\#\text{LOC}_{graph}$ counts the lines of code with comments without blank lines (Schroeder, 1999). The metric $\#\text{Code}_{comL}$ counts the code elements integrated into the knowledge graph linked to at least one ticket via one or more commits, i. e., traceable from tickets.

9.1.4. Coding of Decisions with Decision Types

ConDec represents decision types (i. e., decision groups) using decision levels and custom decision types (Section 7.9). In four of the six validation projects, the developers assigned decision levels and custom types to the decisions. We provided examples of decision types when disseminating ConDec, and the development teams could also choose further types. We prescribed using the decision levels *high*, *medium*, and *realization*. We noticed that the grouping into levels is sometimes ambiguous and subjective, such as for the executive decision on a git branching strategy. Thus, this section only reports the results regarding the other decision types.

After the end of the projects, we created a unique coding scheme using the decision types by Kruchten (2004; 2009), the types collected in the interview study in Table 3.4, and the types introduced by the developers. We aimed for disjunctive decision types to make the coding unambiguous. However, some decision types overlap. Thus, decisions can be assigned to more than one type. We did not introduce a code for *non-existence* or *ban decisions* because they are partly documented as alternatives for decisions and rejected decisions. The author of this thesis applied the coding scheme to the decisions of the six projects. The following paragraphs define the codes, i. e., the decision types, and provide example decisions taken from the projects. For every decision, we list other decision types assigned or explain the assignment in brackets. Appendix A provides the 1146 coded decisions and the related issues, alternatives, and arguments.

Executive Decisions *Executive decisions* concern the software development process, technologies, or tools. They include prioritization and deployment decisions, which are important to practitioners (Chapter 3). Technology decisions concerning the entire system are executive. Technology decisions regarding the backend and data storage, frontend, API, or external frameworks are assigned to the respective group described below. Examples are: *Create feature branches for work items and squash all commits if the pull request passed the review!* (process) *Use nyc and mocha to display the test coverage in the frontend!* (tool, testing) *We create separate git repositories for frontend and backend!* (process, technology) *Use GitHub actions for continuous integration!* (process and tool) *We no longer support Jira version 7!* (technology) *Use Java Development Kit version 11 to compile the plug-in!* (technology)

Quality-Driven Decisions *Quality-driven decisions* concern eliciting and implementing quality requirements related to the quality attributes maintainability, security, performance efficiency, compatibility, usability, reliability, or portability (ISO/IEC 25010, 2011). Synonyms are *property decisions* (Kruchten, 2004) and *driver-oriented decisions* (Weinreich et al., 2015). They include decisions for design guidelines, such as naming conventions. Examples are: *We use under_score names in the API!* (maintainability, API) *We separate HTML and JavaScript code!* (maintainability) *We include third-party JavaScript libraries by manually copying them!* (maintainability, external frameworks) *Do not always make "git pull" calls to improve the performance of the GitClient! Create isPullNeeded() method!* (efficiency, backend)

Functionality-Driven Decisions *Functionality-driven decisions* concern the elicitation and implementation of requirements for functional features and should be discussed and made together with the customer of the software product. Examples from the validation projects

are: *Players and teams are created in a player pool administrated by one or more coaches! Exercises are recommended based on the difficulty that varies by age group! Mail addresses are not immediately displayed when searching for a person! A configurable definition-of-done criterion is that a decision needs to have at least one pro-argument linked! The plug-in is disabled per default (opt-in policies) and can be manually activated by the rationale manager!*

Frontend Decisions All the products developed in the validation projects consist of frontend and backend components. We introduced the respective decision groups because the developers used these groups during the projects. *Frontend decisions* cover decisions for a software application's user interface and the client side. Examples are: *We use Angular as a frontend framework! Unresolved issues and challenged decisions are colored crimson red! Discarded alternatives and rejected decisions are colored in gray! We support a browser app!* (executive) *Integrate the reverse proxy into the frontend! We use a "Welcome" dialog to create an initial player pool by starting the app for the first time! We use a paginator to show just a few events on one page! We don't use breadcrumbs! We implement a date picker!*

Backend and Data Storage Decisions *Backend and data storage* decisions concern the server side of a software application and the database. They partly overlap with technology (executive) decisions but bundle other decisions regarding the backend and storage components. Examples are: *We use Java and Spring Boot as a backend technology!* (executive) *We store the knowledge graph as a singleton object in random-access memory and work with knowledge subgraphs!* (quality-driven for better performance) *We use the PostgreSQL database!*

Application Programming Interface (API) Decisions Decisions regarding the API of microservices were mentioned as important in the interview study in Chapter 3. We include this decision type in the coding scheme because we collected various examples in the projects: *We use get, post, and delete methods in the REST API! Use JSONFilter annotation above getter methods to determine the properties of serialized objects! The path of the REST API for knowledge management is: base-URL/rest/condec/latest/knowledge/service!*

External Library and Framework Decisions This decision type concerns third-party frameworks or libraries. It emerged as easy to classify and was used in the projects. It partly overlaps with executive, backend, and frontend decisions. Examples are: *We use the jgit library to access git repositories, commits, and code!* (backend) *We use jQuery's contextMenu!* (frontend)

Testing Decisions *Testing decisions* concern the quality assurance of the software. They are a special executive decision if they concern the testing process or tools. They are non-executive if they concern mock objects that simulate the software or the selection of test data. We introduced this decision group because the developers frequently used it in the validation projects. Examples are: *The plug-in should have coverage with unit tests of at least 85 %!* (executive) *We create at least 30 test cases each! We create at least one test case per user story!* (executive) *We use Jasmine to test the frontend service!* (executive, not frontend decision because it is not an existence decision concerning the software structure or behavior) *We add the MockPluginSettings.java class to mock the default plug-in configuration!* (not executive)

9.2. Results and Discussion

The following sections present and discuss the results of the knowledge documentation analysis. Section 9.2.1 describes the results to show that it is feasible to document decision knowledge in

practice with ConDec. Section 9.2.2 describes the results to show that it is feasible to document a high amount of knowledge. Section 9.2.3 describes the results to show that it is feasible to create high-quality knowledge documentation. To answer the research questions, this section draws conclusions from 1) the iPraktikum and ISE projects since they had a similar duration and number of developers and 2) the ConDec project (Section 8.1). Table 9.2 provides decision-knowledge examples. The entire decision knowledge is available in Appendix A. Table 9.3 presents knowledge-documentation metrics of the projects.

9.2.1. Feasibility of Documenting Decision Knowledge with ConDec

This section presents the results for the research question *Is it feasible to document decision knowledge in practice with ConDec?* (RQ1). The subsections describe 1) the numbers and types of decision knowledge elements, 2) the documentation locations, 3) whether the decision knowledge was changed after the first documentation, and 4) the decision types as well as their correlation 5) with each other and 6) with the documentation locations. For these aspects, we draw conclusions from the metrics in Table 9.3. In the last subsection, we answer the RQ1.

Decision Knowledge Types and Numbers

This section presents the results for the question *Which and how many decision knowledge elements were documented and when were they documented?* (RQ1.1). The number of decision knowledge elements varies between 99 and 774 for the iPraktikum and ISE projects (Table 9.3 on page 164). The ConDec developers documented 2732 decision knowledge elements. The developers of the three ISE projects documented a similar number of decision knowledge elements (590 – 774). As discussed in the following subsection, the ISE projects resulted in more elements than the iPraktikum projects might be due to the different documentation locations used.

The arguments are either documented as pro-, con-, or unpositioned arguments. An unpositioned argument becomes a pro- or a con-argument if linked to a solution option with the link type *supports* or *attacks*, respectively. The number of pro-arguments is higher than the number of con-arguments. The ratios between the number of rationale elements of different types vary between the projects (Table 9.3). The ratios between issues and decisions range from 0.6 (workplace-control app) to 1.9 (car charging app). A number higher than one means that more issues than decisions are documented. The ratios between the number of decisions and alternatives range from 0.4 (car charging app) to 3.5 in the (workplace-control app). A number higher than one means that more alternatives than decisions are documented. The ratios between the number of arguments and solution options (decisions and alternatives) range from 0.4 in the workplace-control project and 2.23 in the ISE 21/22 project. A number higher than one means more arguments than solution options are documented. Numbers below one indicate incomplete documentation, which Section 9.2.3 will further analyze.

Figure 9.1 shows the number of rationale elements documented per date in the validation projects. While there are times with little to no rationale documentation, the rationale elements were documented during the entire project. Notably, many rationale elements were documented at the end of the ISE 19/20 project. The developers documented many decisions during the final sprint to consolidate the rationale documentation. In contrast, in the ISE 20/21 project, many rationale elements were documented at the beginning of the project. In general, there seems to be increased rationale documentation before sprint reviews since the developers consolidated the rationale documentation of the sprint.

Table 9.2.: Decision knowledge documentation examples of the validation projects. The entire decision knowledge documentation is available in Appendix A. The status of the issues and decisions is shown in brackets if it deviates from *resolved* or *decided*.

Project	Decision Knowledge Types and Summaries	Decision Types	Quality Problems
iPraktikum Workplace	How should we discover devices? Discover devices using the User Datagram Protocol!	API	no pro-argument(s), no alternative(s)
Control	How do we handle user stories/story points? Client and server tasks in the same user story	executive	no pro-argument(s), no alternative(s)
iPraktikum Car Charging	Which transition should be used between UserClass – HouseClass – SmartDeviceClass? Aggregation House/Smart devices can exist without the car owner. In our case, it would make no sense if they exist without the car owner. Composite between UserClass – HouseClass – SmartDeviceClass	backend and data storage	relation of arguments to decision not clear
	How to coordinate preferences of different users within the recommendation algorithm? *(unsolved)*	functionality-driven	no alternatives, really unsolved?
ISE 2019/20	Which database should we use? MongoDB JSON objects would be in use in the back- and frontend Queries and joins are more difficult We will use PostgreSQL! Already in use and required by the customer. Stiffer development practice - harder to iterate	backend and data storage	relation of arguments to opposite solution option not clear
	How do we display the test coverage for the frontend? Use nyc and mocha to display the test coverage in the frontend! *(rejected)* Additional libraries have to be installed. Use the Jest library with an in-built coverage tool to display the test coverage! everything in one library and good vue and typescript support	testing, executive	–
ISE 2020/21	When should we use a synonym filter? During indexing Adding new synonyms leads to reindexing the documents During querying New synonyms can be added without the need of reindexing the documents	quality-driven (efficiency), backend and data storage	–
	Which technology shall the backend use? Use NodeJS! We have more experience with NodeJS than with Flask. Flask We have some experience with Flask.	backend and data storage, ext. lib or framework	naturalistic arguments, might not be the best solution
ISE 2021/22	How can a client of the app communicate with its various components (backend and frontend)? We could use a forward proxy to enable a client of the app to access the backend and frontend. We use the reverse proxy server nginx to enable a client of the app to access the frontend!	API	no pro-argument(s) for decision, no con-argument(s) for alternative
	How should we filter the exercises by the number of players? We could use a single number. We use a range. The coach doesn't always know the exact number of players that will participate in training	frontend	relation of argument to alternative not clear
ConDec	Which library to use to determine textual similarity between two texts? Use Apache Commons text library to determine textual similarity between two texts! It is easier to use an external library than to implement the mechanism to determine textual similarity ourselves.	external framework	no alternative(s)
	How can we create coverage reports? Use the jacoco-maven-plugin to create coverage reports! Use the cobertura-maven-plugin to create coverage reports! The cobertura-maven-plugin is incompatible with java 11.	testing, executive	no pro-argument(s) for decision

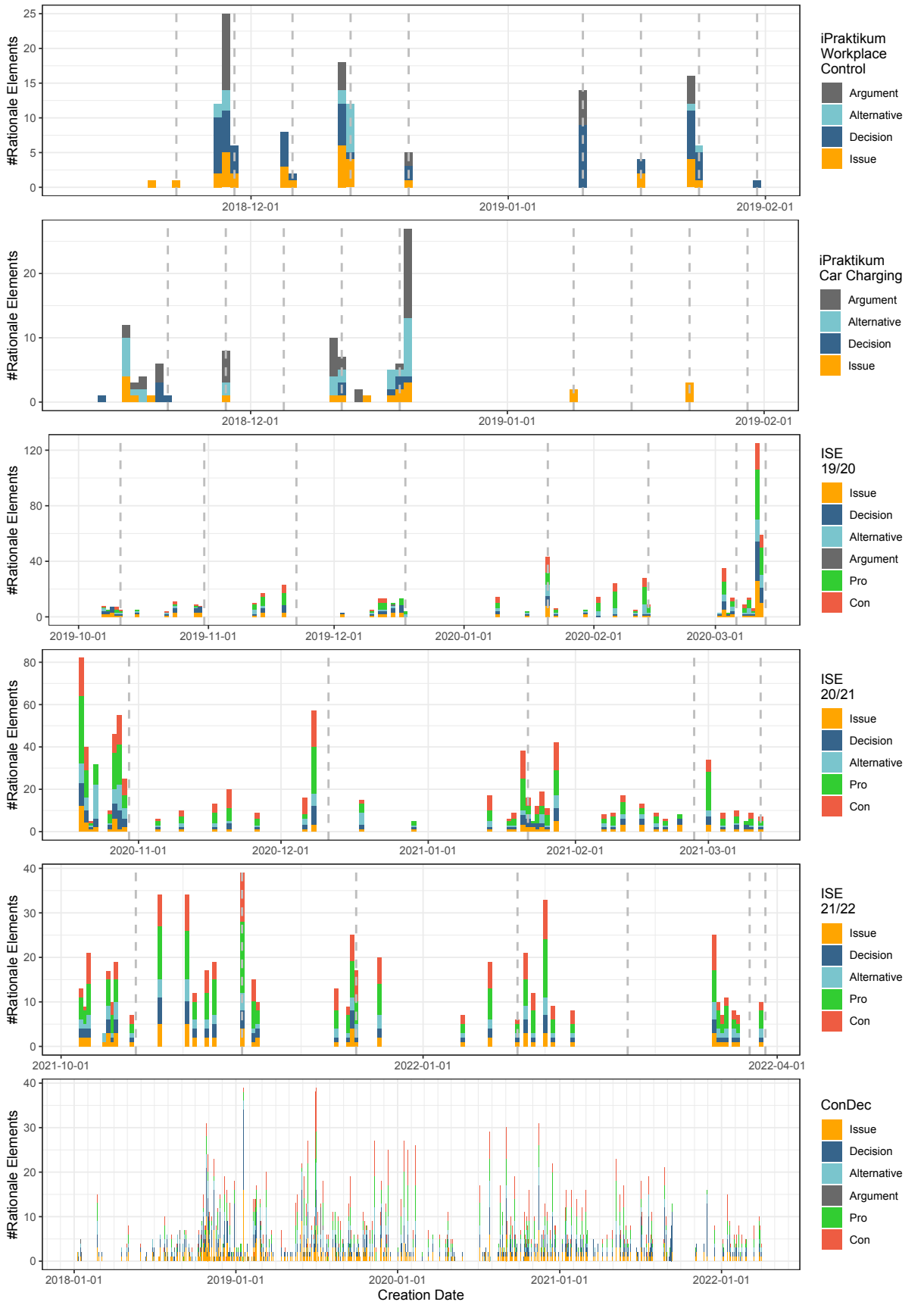

Figure 9.1.: Number of rationale elements documented per date in the validation projects. The gray dashed vertical lines indicate sprint reviews.

Table 9.3.: Knowledge documentation of the six validation projects: 1) requirements and other tickets in Jira, 2) commits and code in git and their trace links to Jira, and 3) decision knowledge in Jira and git. The numbers in brackets either denote the numbers for test code or the respective percentage.

		iPraktikum 18/19		ISE 19/20	ISE 20/21	ISE 21/22	ConDec
	Product	Workplace Control App	Car Charging App	IoT Platform	Web Search Engine	Soccer App	ConDec Plug-Ins
	Requirements and other Tickets in Jira						
	Requirements Specification	11 epics, 35 user stories	31 scenarios	7 epics, 53 user stories, 8 quality attributes	3 epics, 13 user stories, 6 quality requirements, one user role, 4 personas	4 epics, 33 user stories, 7 quality requirements, 2 user roles, 4 personas	5 user roles, 5 user tasks, 14 sub-tasks, 98 system functions, 48 workspaces, 14 quality requirements
	Other Tickets	172 development tasks, 309 development sub-tasks, 50 bug reports	62 development tasks, 593 development sub-tasks, 57 bug reports	27 development tasks, 129 development sub-tasks, 35 bug reports	131 work items, 34 bug reports	179 work items, 39 bug reports	494 work items, 151 bug reports, 50 system test cases, 5 test execution tickets
	#Tickets	557	743	259	191	268	999
	Commits and Code in git						
	#Repositories			5	1	2	6
	#Commits			998	185	983	2033
	#Commits$_L$			192 (19 %)	145 (78 %)	474 (48 %)	1658 (82 %)
	Code Types in Knowledge Graph			ts, vue, js, xml, scss, conf, html	ts, html, xml	java, ts, js, xml, css, conf, html, yaml	java, js, xml, vm, soy, ts
	#Code$_{graph}$ (Test)			101 (24)	658 (12)	292 (26)	944 (463)
	#LOC$_{graph}$ (Test)			10553 (1730)	141266 (1724)	19268 (3355)	293151 (29337)
	#Code$_{comL}$			56 (56 %)	658 (100 %)	284 (97 %)	940 (99.5 %)
	Decision Knowledge Documentation in Jira and git						
Rationale Types	**#Issues**	33	21	111	81	73	638
	#Decisions	56	11	116	116	76	771
	#Alternatives	16	29	94	118	84	336
	#Arguments	26	38	184, 113, 1	262, 197	203, 154	516, 453, 18
Ratios	**#/#**	$\frac{33}{56} = 0.6$	$\frac{21}{11} = 1.9$	$\frac{111}{116} = 0.96$	$\frac{81}{116} = 0.7$	$\frac{73}{76} = 0.96$	$\frac{638}{771} = 0.8$
	#/#	$\frac{56}{16} = 3.5$	$\frac{11}{29} = 0.4$	$\frac{116}{94} = 1.2$	$\frac{116}{118} = 0.98$	$\frac{76}{84} = 0.9$	$\frac{771}{336} = 2.3$
	#(+++)/#(+)	$\frac{26}{72} = 0.4$	$\frac{38}{40} = 0.95$	$\frac{298}{210} = 1.4$	$\frac{459}{234} = 1.96$	$\frac{357}{160} = 2.23$	$\frac{987}{1107} = 0.9$
Intra-Rationale Completeness	**# with**	10 (30.3 %)	12 (57.1 %)	66 (59.5 %)	64 (79 %)	69 (94.5 %)	282 (44.2 %)
	# with	42 (75 %)	10 (90.9 %)	116 (100 %)	116 (100 %)	76 (100 %)	771 (100 %)
	# with	9 (16.1 %)	1 (9.1 %)	85 (73.3 %)	109 (94 %)	74 (97.4 %)	310 (40.2 %)
	# with	3 (18.8 %)	13 (44.8 %)	67 (71.3 %)	90 (76.3 %)	80 (95.2 %)	224 (66.7 %)
	#Rationale Elements	131	99	619	774	590	2732
Documentation Origin	**#Elements as Tickets**	131 (100 %)	99 (100 %)	12 (1.9 %)	50 (6.5 %)	51 (8.6 %)	87 (3.2 %)
	#Elements in Ticket Text			606 (97.9 %)	724 (93.5 %)	534 (90.5 %)	1812 (66.3 %)
	#Elements in Commit Messages			1 (0.2 %)	0 (0 %)	0 (0 %)	443 (16.2 %)
	#Elements in Code Comments			0 (0 %)	0 (0 %)	5 (0.8 %)	387 (14.2 %)

Continued on next page

		iPraktikum 18/19		ISE 19/20	ISE 20/21	ISE 21/22	ConDec
	Product	Workplace Control App	Car Charging App	IoT Platform	Web Search Engine	Soccer App	ConDec Plug-Ins
Status	**#Open Issues**	5 (15.2 %)	11 (52.4 %)	4 (3.6 %)	0 (0 %)	1 (1.4 %)	14 (2.2 %)
Status	**#Rejected Decisions**			8 (6.9 %)	9 (7.8 %)	3 (3.9 %)	65 (8.4 %)
	Decision Knowledge in Relation to Requirements and Code						
	#🔨/#Requirements	$\frac{56}{46} = 1.2$	$\frac{11}{31} = 0.4$	$\frac{116}{68} = 1.71$	$\frac{116}{27} = 4.3$	$\frac{76}{50} = 1.5$	$\frac{771}{184} = 4.2$
	#🔨/#Code$_{graph}$			$\frac{116}{101} = 1.15$	$\frac{116}{658} = 0.2$	$\frac{76}{292} = 0.3$	$\frac{771}{944} = 0.8$
	Decision Coverage of Requirements (User Stories, Scenarios, or System Functions) in Link Distance <= 3						
	No 🔨 Traceable	34 (97.1 %)	31 (100 %)	29 (54.7 %)	0 (0 %)	0 (0 %)	3 (3.1 %)
	1 🔨 Traceable	1 (2.9 %)	0 (0 %)	8 (15.1 %)	0 (0 %)	0 (0 %)	3 (3.1 %)
	> 1 🔨 Traceable	0 (0 %)	0 (0 %)	16 (30.2 %)	13 (100 %)	33 (100 %)	92 (93.9 %)
	Max. #🔨 Traceable	1 (2.9 %) → 1 🔨	31 (100 %) → 0 🔨	2 (3.8 %) → 6 🔨	1 (7.7 %) → 27 🔨	1 (3 %) → 8 🔨	2 (2 %) → 44 🔨
	Decision Coverage of Code in Link Distance <= 3						
	No 🔨 Traceable			82 (81.2 %)	27 (4.1 %)	89 (30.5 %)	12 (1.3 %)
	1 🔨 Traceable			2 (2 %)	11 (1.7 %)	42 (14.4 %)	34 (3.6 %)
	> 1 🔨 Traceable			17 (16.8 %)	620 (94.2 %)	161 (55.1 %)	898 (95.1 %)
	Max. #🔨 Traceable			17 (16.8 %) → 2 🔨	1 (0.2 %) → 39 🔨	2 (0.7 %) → 13 🔨	1 (0.1 %) → 225 🔨

Documentation Locations

This section presents the results for the question *Where were the decision knowledge elements documented, i. e. in which documentation locations, and when?* (RQ1.2). In the iPraktikum, ConDec only enabled to document decision knowledge as entire tickets. Thus, it was the only location used. In the ISE projects (90.5 – 97.9 %) and ConDec project (66.3 %), the prevailing documentation location was ticket text, i. e., the description and comments (Table 9.3). The results indicate that documenting decision knowledge in Jira ticket text is more accessible than creating entire tickets for every decision knowledge element. In addition to documenting decision knowledge in the text of requirements and work items, the developers in the ISE projects often created entire tickets for issues. Then they captured the related solution options and arguments in their text. The ConDec developers also documented decision knowledge in commit messages (16.2 %) and code comments (14.2 %). While the developers' preferences differ, the results show that it is feasible to document decision knowledge in different locations with ConDec.

Figure 9.2 shows the proportion of used documentation locations over time for the projects with more than one documentation location. In the ISE 19/20 project, the developers mainly used ticket descriptions and comments to document decision knowledge. In the ISE 20/21 and 21/22 projects, the developers mainly documented decision knowledge in ticket descriptions and comments, but they also continuously used entire tickets to report issues. The ConDec developers continuously used all four locations to document decision knowledge.

Change of Decision Knowledge Elements Over Time

This section presents the results for the question *How many rationale elements were updated after their first documentation?* (RQ1.3). Figure 9.3 compares the date of first documentation (creation date) of the rationale elements in the projects against their last update as scatter plots (above) and boxplots and histograms (below). Some rationale elements were never updated in all projects (positioned on the diagonal lines in the scatter plots in Figure 9.3). The histograms show that most rationale elements were never or shortly updated after their first documentation (the first bar is the highest). The further away the elements are from the diagonal line in the

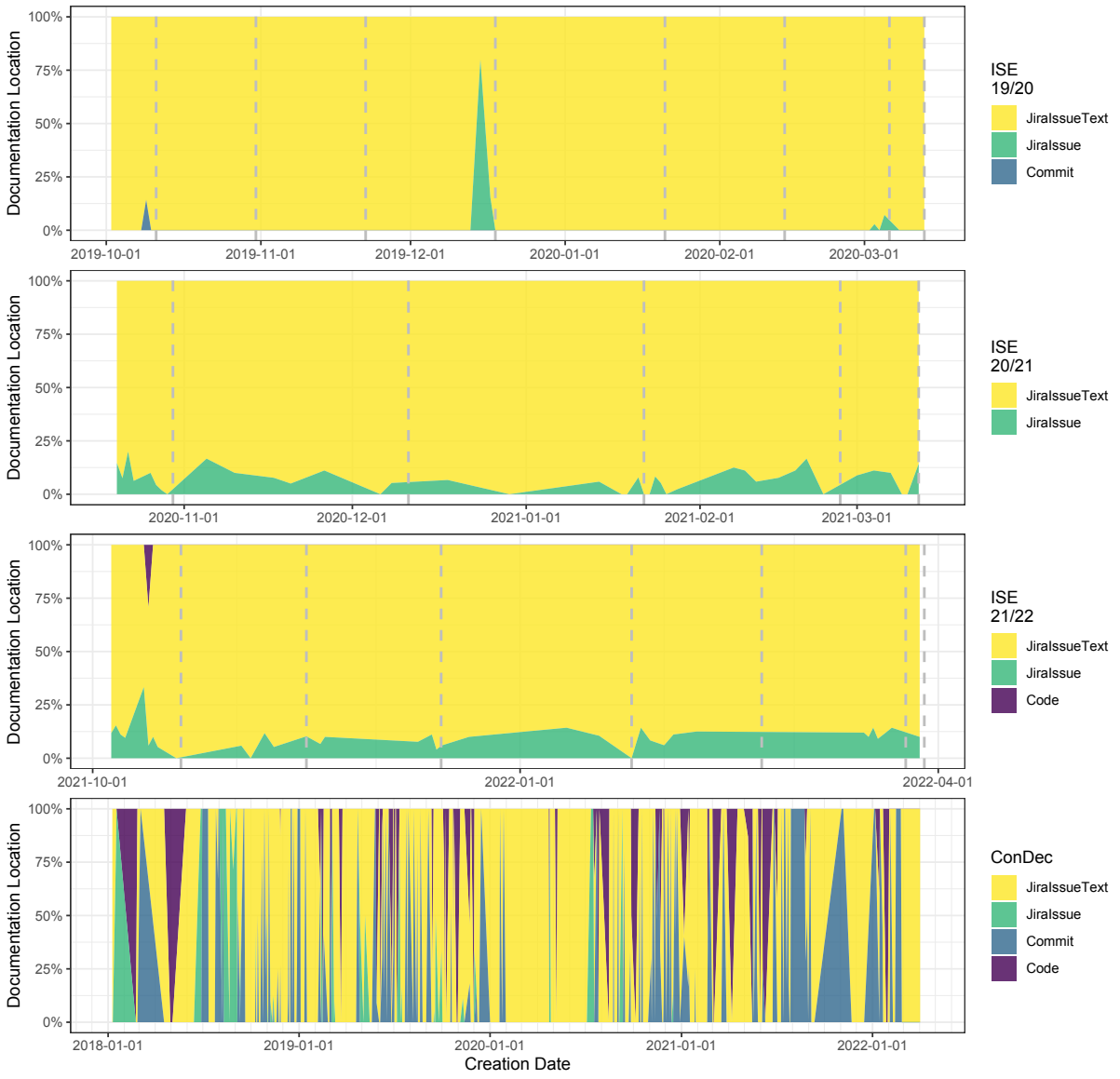

Figure 9.2.: Proportion of documentation locations used over time in the validation projects. The gray dashed vertical lines indicate the sprint reviews.

scatter plots, the longer the duration between their creation and the last update. While some rationale elements were never changed, others were changed after different periods. The results show that the developers constantly updated rationale elements throughout the projects, maybe because ConDec presented the elements to the developers as part of the knowledge subgraph for a development task. Interestingly, elements positioned along horizontal lines in the scatter plots were changed on the same date, indicating that a more extensive set of rationale documentation was improved at this date. Notably, all rationale elements documented in the description or comment of a ticket, in a commit message (which is transcribed into a ticket comment), or in a code comment have the same creation date and last update due to a lack of tool support to keep a fine-grained version history. Thus, rationale elements of different types documented in these locations are on top of each other in the scatter plots in Figure 9.3. The current lack of tool support is why we cannot answer how long it takes from the first capturing of an issue to its solution through a decision. Figure 9.3 omits elements that were updated after the project was finished, e. g., to fix spelling mistakes.

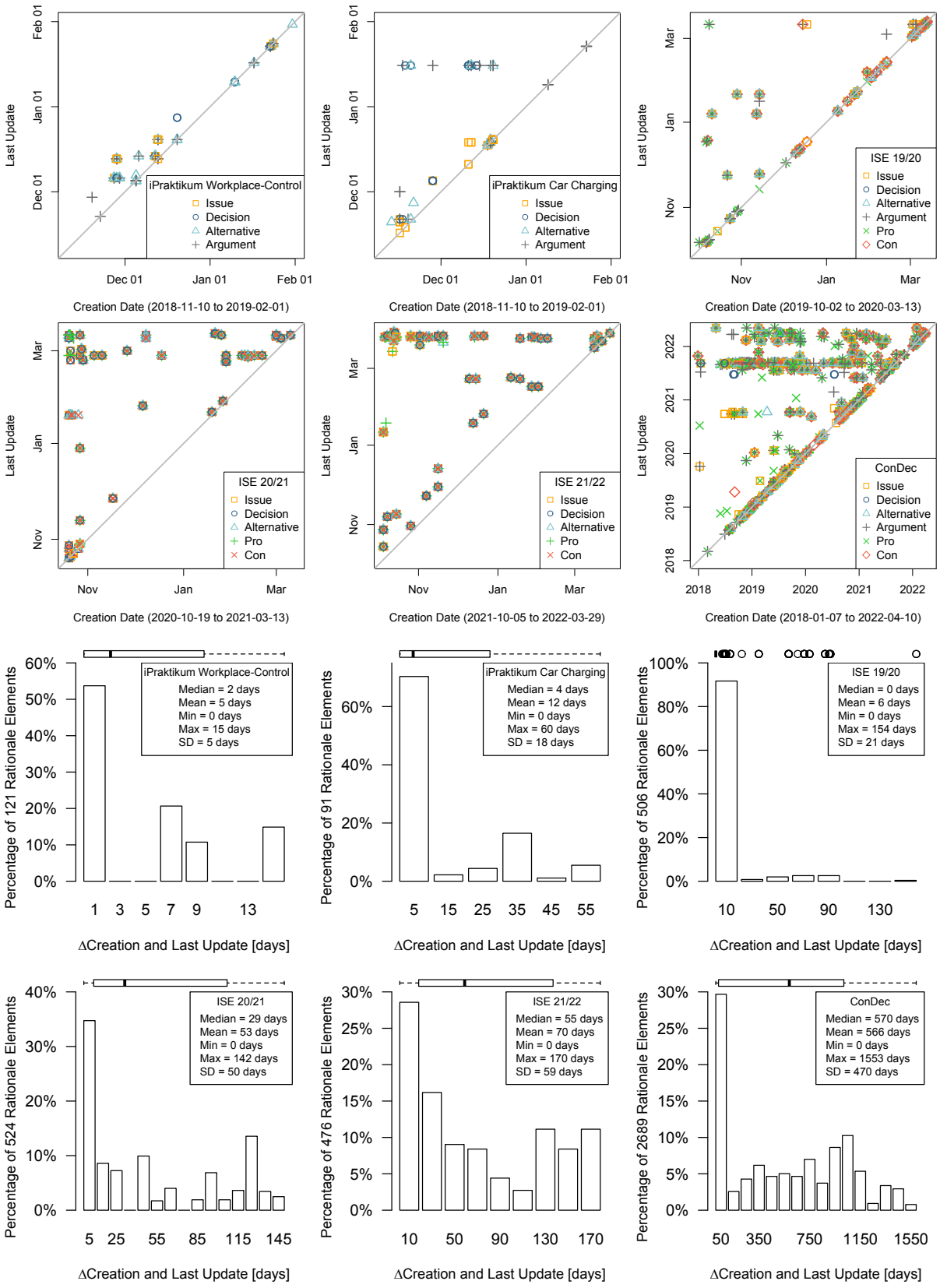

Figure 9.3.: **Above:** Date of first documentation (creation date) plotted against the last update of the rationale elements in the validation projects during the project duration. **Below:** Boxplots and histograms of the time difference between the first documentation and the last update of the rationale elements as well as the median, mean, minimum, maximum, and standard deviation (SD).

Decision Types

This section presents the results for the questions *What types of decisions are documented and when? Were specific types of decisions documented at specific points in time?* (RQ1.4). Figure 9.4 visualizes the number of decisions per decision type (left) and the number of types per decision (right) in the iPraktikum and ISE (above) and the ConDec projects (below). The largest group of decisions are frontend (33 %) as well as backend and data storage decisions (29 % in the iPraktikum and ISE projects or 32 % in ConDec). 18 % or 26 % of the decisions are functionality-driven, 14 % or 8 % of are quality-driven, 6 % or 5 % concern the API, 3 % or 6 % concern an external library or framework, 19 % or 8 % are executive decisions, and 6 % or 5 % relate to software testing. While the majority of the decisions (74 % or 75 %) is assigned to one decision type, about a quarter of the decisions are assigned to two types and very few decisions to three types. Further data are required to investigate to which extent the distribution of decision types is the same for other projects or differs depending on the software domain or team size (executive). However, the classified decisions (in Appendix A) can be helpful in other software development projects as a basis for decision guidance. The lowest percentage of executive decisions was documented in the ConDec project. Especially in the iPraktikum, more than one-third of the documented decisions are executive. The high number matches the finding that practitioners mentioned executive decisions as important to be documented (Chapter 3).

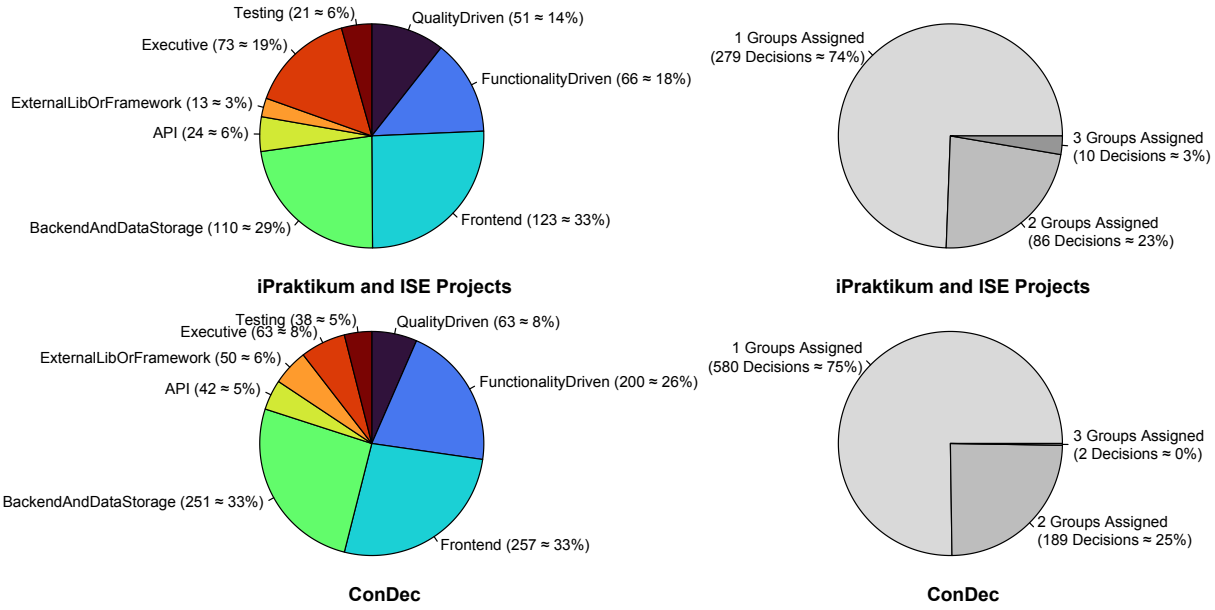

Figure 9.4.: **Left:** Number of decisions per decision type in the validation projects. A decision can be assigned to more than one type. Thus, the sum of percentages exceeds 100 %. **Right:** Number of assignments per decision. **Above:** Average of all 375 decisions in the iPraktikum and ISE projects. **Below:** 771 in the ConDec project.

Figure 9.5 shows the proportion of decision types documented over time. In the ISE 20/21 project, testing decisions were only documented shortly before and after the third sprint review. In the ISE 21/22 project, executive decisions were documented subsequently to the first, second, third, and fourth sprint reviews. Generally, there seems not to be a tendency to document a specific type of decision during a particular period. This underpins the idea of CSE that the developers frequently iterate over all phases of software development instead of performing waterfall-like projects. Figure 9.5 shows that it is feasible to document different types of decisions with ConDec continuously.

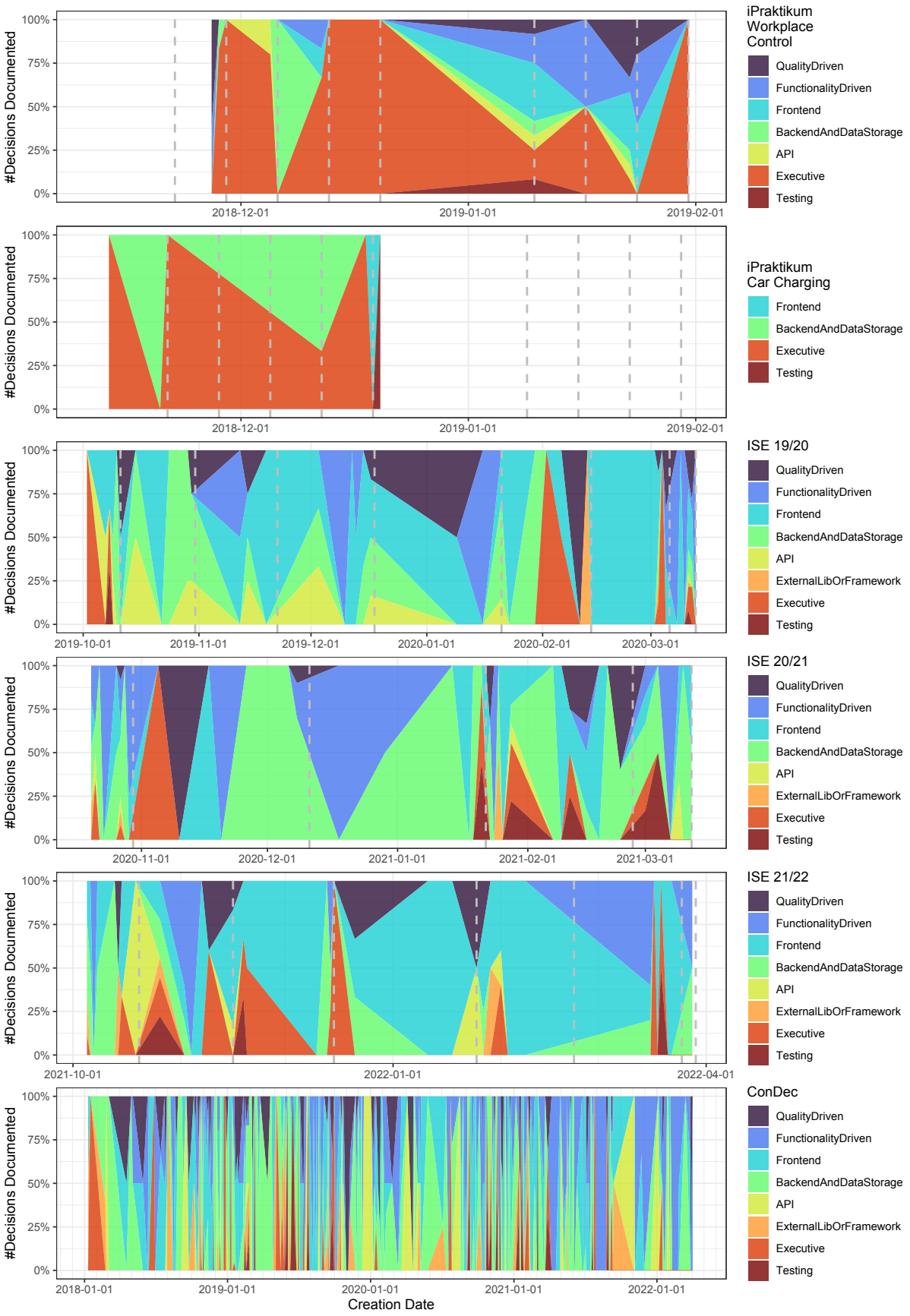

Figure 9.5.: Proportion of decision types documented over time in the validation projects. The gray dashed vertical lines indicate sprint reviews.

Correlation of Decision Types With Other Decision Types

This section presents the results for the question *Are decision types often or never assigned to the same decisions?* (RQ1.5). Figure 9.6 shows a Venn diagram (left) and the correlation matrix (right) for the decision types in all validation projects. The Venn diagram and the correlation matrix were calculated with all 1146 decisions because the distribution of decision types is similar for the validation projects (Figure 9.4). In the Venn diagram, the size of the circles indicates the number of decisions per decision type, and the intersections indicate typical combinations. For example, the intersection between the circles for testing and executive decisions indicates that most of the testing decisions (44) are specific executive decisions, while 15 are non-executive. The correlation matrix expresses the likelihood that two decision types are assigned to the same decision. Only executive and testing decisions have a positive correlation value (0.43). Crossed-out correlation coefficients are insignificant (significance level is 0.05). All other decision types are negatively correlated. While some decisions have two types, such as functionality-driven and frontend, the majority of the frontend decisions are not classified as functionality-driven. Especially, frontend decisions and decisions regarding the backend and data storage are disjunctive (-0.42).

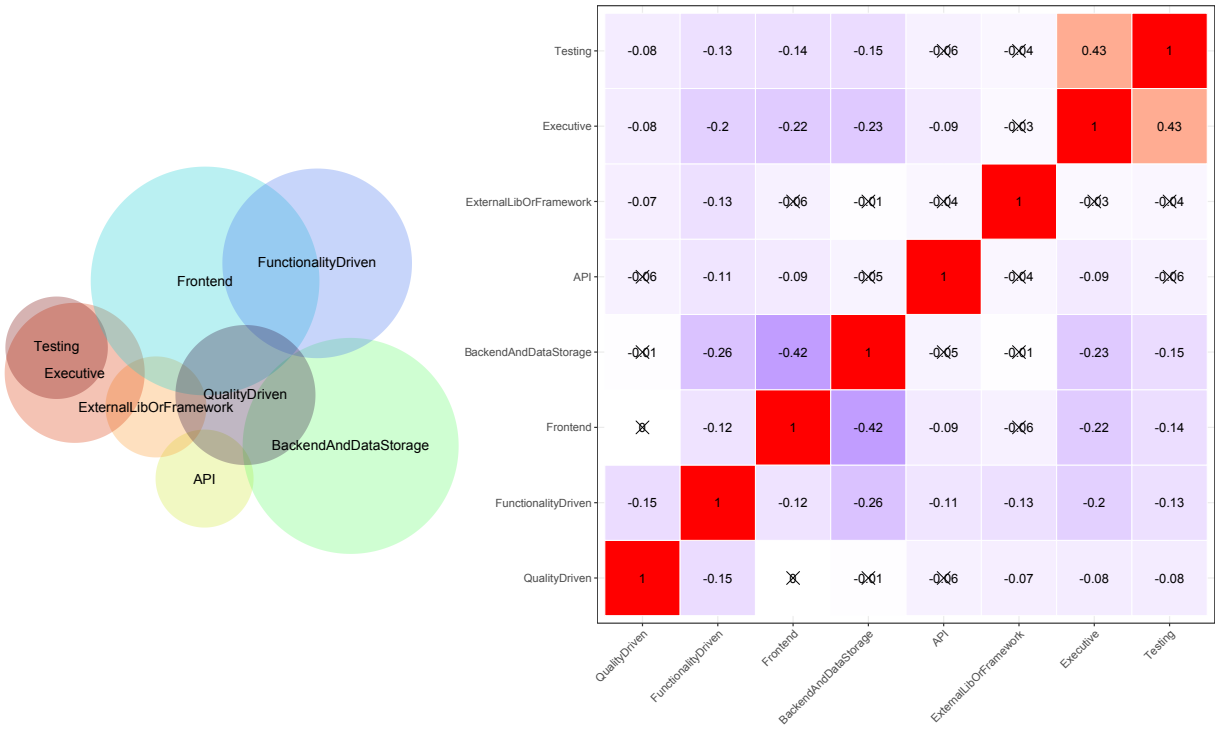

Figure 9.6.: **Left:** Venn diagram of decision types and intersections in all projects.
Right: Correlation between the decision types in all projects.

Correlation of Decision Types with Documentation Locations

This section presents the results for the question *Are there decision types that often or never are documented in specific documentation locations?* (RQ1.6). Figure 9.7 shows the correlation matrix for the decision types and documentation locations in the ConDec project. The correlation matrix was calculated on the decisions of the ConDec project because it is the only project in which the developers documented decision knowledge in the four documentation locations currently supported (entire Jira tickets, description and comments of existing Jira tickets such as

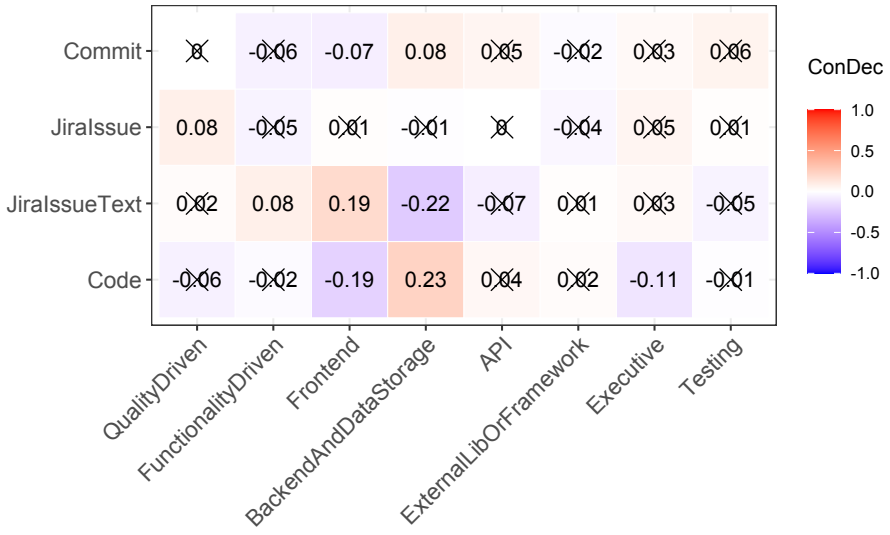

Figure 9.7.: Correlation between the decision types and documentation locations for the decisions documented in the ConDec project.

work items or requirements, commit messages, and code comments). The documentation in code comments is positively correlated with the documentation of decisions regarding backend and data storage (correlation coefficients 0.23) and negatively correlated with the documentation of frontend (-0.19) and executive (-0.11) decisions. Frontend decisions are rarely documented in code comments. The reason might be that the feature to extract rationale from code comments only initially supported the Java programming language. Java is mainly used in the backend of the ConDec plug-ins (except for the Eclipse plug-in). In the future and other projects, more decisions might be documented in frontend code, such as JavaScript files. No executive decisions are documented in the code, but non-executive testing decisions, e. g., *We have one mock git repository for testing!* Crossed-out correlation coefficients are insignificant (significance level is 0.05). A few other positive and negative correlations exist, e. g., to document quality-driven decisions such as *Use basic authentication!* in entire Jira tickets. However, the correlations are relatively low, and, in general, the documentation locations of ConDec are interchangeable, meaning developers can choose their preferred location.

Discussion: Is it feasible to document decision knowledge in practice with ConDec?

With the answers for the sub-questions of RQ1, we showed that it is feasible to document decision knowledge with various rationale types, in four documentation locations in the issue tracking system Jira and version control system git, with various decision types, and continuously, i. e. over time in small increments instead of only once in practice with ConDec. Besides, we showed that it is feasible to continuously change decision knowledge in practice with ConDec.

9.2.2. Feasibility of Documenting a High Amount of Knowledge with ConDec

This section presents the results for the research question *Is it feasible to document a high amount of knowledge in practice with ConDec?* (RQ2). The subsections describe 1) the requirements specification and other tickets in Jira, 2) the code in git, 3) the ratios between the number of rationale elements and requirements as well as code. In the last subsection, we answer the RQ2.

Requirements Specification and Other Tickets in the Issue Tracking System Jira

This section presents the results for the question *Which and how many requirements and other tickets are documented?* (RQ2.1). The requirements elicitation, documentation, checking, and management was an ongoing, iterative process, i. e., the requirements documentation was not fixed from the beginning. Still, for simplicity, this section only reports the results at the end of the project (Table 9.3 on page 164). The requirements of the validation projects are specified either as epics and user stories, scenarios, or in the notations of *Task and Object-oriented Requirements Engineering* (Paech and Kohler, 2004). The different types of requirements are represented by the Feature entity in the ConRat knowledge model (Section 6.1). All projects use a hierarchical requirements model that breaks down high-level requirements (e. g., epics, user tasks) into fine-grained requirements (e. g., user stories, system functions) or work items. Four of the six projects explicitly specify quality requirements either in the form of distinct tickets or as user stories. Amongst the iPraktikum and ISE projects with comparable duration and number of developers, the specification for the IoT platform contains the highest number of requirements (68). In contrast, the specification for the web search engine contains the lowest number (27). The specification for the ConDec plug-ins contains 184 requirements in total. It must be noted that these numbers might not represent the size and complexity of the software systems because the granularity of requirements might differ between the projects, even for the same requirement types, such as user stories. In all projects, work items resemble the ticket type with the highest number. The project documentation contains other tickets, in particular, bug reports and test-related tickets.

Code in the Version Control System Git

This section presents the results for the question *Which and how many code elements were created?* (RQ2.2). Frequently used languages and code file types are Java, JavaScript (js), TypeScript (ts), Hypertext Markup Language (HTML), Extensible Markup Language (XML), Cascading Style Sheets (CSS), and configuration (conf) in the available git repositories (Table 9.3 on page 164). The number of code elements $\#\text{Code}_{graph}$, i. e., code files integrated into the knowledge graph, varies between 101 and 658, and the $\#\text{LOC}_{graph}$ between 10553 and 141266 for the ISE projects. The ConDec project contains 944 code elements and 293151 $\#\text{LOC}_{graph}$. The particularly high numbers are due to the inclusion of library code in git.

Ratios between Number of Decisions and Requirements as well as Code

This section presents the results for the question *What are the ratios between decision knowledge and system knowledge elements?* (RQ2.3). The ratios between the number of decisions and requirements range from 0.4 to 4.3 for the iPraktikum and ISE projects (Table 9.3 on page 165). For the ConDec project, the ratio is 4.2, which means that more than four decisions are documented for a requirement on average. The ratios between the number of decisions and code elements range from 0.2 to 1.15 for the ISE projects, and the ratio is 0.8 for the ConDec project.

Discussion: Is it feasible to document a high amount of knowledge in practice with ConDec?

We showed that it is feasible to document a high amount of system knowledge, i. e. requirements and code, with ConDec through its application in the validation projects. We consider the amount of system knowledge documented in the validation projects with a duration of about six months as high. These projects resulted in between 27 to 68 documented requirements and 101 to 658 code files. The ISE 19/20 project (IoT platform) shows the feasibility to document a high amount of rationale related to requirements and code with ConDec using the following two indicators (Table 9.3 on page 165): First, the ratios between the number of decisions and

requirements or code element both exceed 1.0, which means that more than one decision was documented for each requirement and code element. Second, for every of the 116 decisions more than four other rationale elements are documented on average (619 rationale elements are documented in total). Notably, a ratio of 1.0 could mean that a decision was documented for every requirement or code element but could also mean that all the decisions relate to only one requirement or code element. To further analyze the relation, Section 9.2.3 quantifies the number of traceable decisions from requirements and code using the decision coverage metric.

9.2.3. Feasibility of Documenting High Quality Knowledge with ConDec

This section presents the results for the research question *Is it feasible to create high-quality knowledge documentation in practice with ConDec?* (RQ3). The subsections describe 1) the completeness of links between code in git to tickets in Jira via commits, 2) the intra-rationale completeness, 3) the states of issues and decisions, and 4) the decision coverage of requirements and code. In the last subsection, we answer the RQ3.

Trace Links between Commits and Code in Git to Tickets in Jira

This section presents the results for the question *How well is the knowledge documented in git linked to the knowledge documented in Jira?* (RQ3.1). The number of commits with a valid ticket identifier in their commit message ($\#\text{Commits}_L$) strongly varies between 19 % and 78 % for the ISE projects. It is 82 % in the ConDec project (Table 9.3 on page 164). This confirms the finding by Saito et al. (2017) and Rath et al. (2018) that only parts of the commits are linked to tickets. The number of code files traceable from at least one ticket ($\#\text{Code}_{comL}$) is higher (56 % to 100 % in the ISE projects and 99.5 % in the ConDec project). We conclude that the majority of code elements are traceable from at least one ticket, which enables the knowledge exploitation of decision knowledge documented in the issue tracking system from code and vice versa. However, missing links and wrong links might hinder the exploitation. Currently, ConDec offers basic mechanisms to detect wrong links (Section 7.6.9), and developers can manually mark links as wrong (Section 7.5.3). Techniques to automatically improve and maintain trace links as systematized by Hübner and Paech (2020) should be added to ConDec in the future.

Intra-Rationale Completeness

This section presents the results for the question *Is the decision knowledge completely documented?* (RQ3.2). It assesses the criteria of intra-rationale completeness: 1) issues with at least one alternative, 2) decisions with an issue, 3) decisions with at least one pro-argument, and 4) alternatives with at least one con-argument documented. The section exemplifies violations.

The number of issues with at least one alternative documented (next to the solution decision) varies between 30.3 % and 94 % in the iPraktikum and ISE projects (Table 9.3 on page 164). In the ConDec project, 44.2 % of issues is relatively low. In all of the projects, it was not mandatory to document alternatives next to the solution decision. Still, we encouraged the documentation of alternatives while disseminating rationale management (Chapter 12). An exception is finding out what to do rather than deciding between solution options. For example, the issue *How can we clean up after the deletion of a Jira project?* has only a decision *Delete all knowledge elements and links from Jira ticket description and comments after project deletion in database! Remove all singleton objects for the project!* but no alternatives. In other cases, the documentation of alternatives would have been useful: An example for the workplace-control app is the *Which tool should we use to write/publish the API during the development?* with the decision *Postman!* We cannot know if the developers discussed alternatives such as *Swagger* and why they discarded the alternatives.

The amount of decisions with an issue documented and linked varies between 75 % and 100 % in the iPraktikum and ISE projects. It is 100 % in the ConDec project (Table 9.3 on page 164). The fulfillment of the criterion was mandatory during the ISE and ConDec projects and, thus, is very high. In the iPraktikum projects, decisions without an issue are documented, making it hard to understand the problem they solve. Examples for the workplace-control app are *Shared devices energy consumption is distributed among relevant occupants!* and *Use core location and geofence to track time user spent at the workplace!*

The number of decisions with at least one pro-argument documented varies between 9.1 % and 97.4 % in the iPraktikum and ISE projects (Table 9.3 on page 164). In the ConDec project, 40.2 % of decisions are relatively low. In all of the projects, it was not mandatory to document pro-arguments for decisions. Still, we encouraged the documentation of pro-arguments while disseminating rationale management (Chapter 12) since it enables an understanding of why a decision was selected. We did not enforce the arguments documentation in favor of lightweight decision documentation. In all projects, there are decisions without documented pro-arguments. An example for the IoT platform is the issue *When should we load the list of devices from the database in the frontend?* The decision is *We get the list of devices from the backend while loading the page in the frontend!* There is no documented pro-argument (and no alternative) for the decision, making it hard to understand why the issue was discussed.

The amount of alternatives with at least one con-argument varies between 18.8 % and 95.2 % in the iPraktikum and ISE projects and is 66.7 % in the ConDec project (Table 9.3 on page 164). Again, it was not mandatory to document con-arguments for alternatives. In all projects, there are alternatives without documented con-arguments, which makes it difficult to understand why they were discarded. An example for the IoT platform is the issue *How to implement charts, i. e., with which library?* with the alternative *We could use Chartist.js to implement charts.* For this alternative, only pro-arguments are documented, e. g., *open source.* This pro-argument is also linked to the decision *We use chart.js to implement charts!*, but the decision has a further pro-argument *We've already had some experience with it, therefore quick and easy to use.* This additional argument for the decision indicates why the alternative was discarded. However, it would be more explicit to add an *attacks*-association from the pro-argument for the decision to the alternative.

The ConDec quality checking feature enables the enforcement of all criteria of intra-rationale completeness, which can be valuable in strictly regulated domains (e. g., medical or aerospace). Burge and D. C. Brown (2008a) distinguish syntactic and semantic inferences over rationale documentation. Syntactic inferences are concerned with the structure of the rationale. Semantic inferences look into the content. While the analysis of the intra-rationale completeness was syntactical, a semantic analysis of the rationale elements should be done in the future. In some cases, decisions cannot be understood outside the development team. An example is the issue *Should we merge our issues with the sign-in database developers?* with the decision *yes.* Such decision knowledge documentation is probably not helpful after the specific discussion.

States of Issues and Decisions

This section presents the results for the question *What are the states of the issues and decisions?* (RQ3.3). The proportion of unsolved issues varies between 0 % and 52.4 % in the iPraktikum and ISE projects and is 2.2 % in the ConDec project (Table 9.3 on page 165). Proportionally, the iPraktikum projects have the highest number of unsolved issues (15.2 % and 52.4 %) not linked to a solution decision. Examples for the workplace-control app are *Should the world maps and anchors be saved on the server?* and *What should be in the energy consumption report?* Examples for the car charging app are *How to display the calendar for the charging station owner?* and *What should be the next view after the sign-in view?* This might be due to

missing tool support for quality checking at this time, missing links between issues and decisions, and also because the rationale documentation was not demanded in sprint reviews as in the ISE projects. The ISE projects have only a few unsolved issues (0 – 4) because the developers aimed to solve the issues before the end of the project and they consolidated the (decision knowledge) documentation. The ConDec development is an ongoing open-source project to be continued in the future and the open issues represent opportunities for improvement.

Rejected decisions were incorporated into the system and then changed (Section 6.1.2). Between 3.9 % and 7.8 % of the decisions in the ISE projects and 8.4 % of the decisions in the ConDec project are rejected (Table 9.3 on page 165). These decisions are valuable since they 1) can prevent going into dead ends in the same or other projects and 2) the rejected code is preserved and linked via the feature tasks in the git repository, which—next to the documented rationale—can support understanding and undoing the rejection.

Decision Coverage of Requirements and Code

This section presents the results for the question *How many decisions are traceable from requirements and code, i. e., what is the decision coverage?* (RQ3.4). The decision coverage expresses the direct and indirect traceability from system knowledge elements, i. e., requirements in tickets and code, to documented decisions. In general, the higher the decision coverage, the better the accessibility of the decisions from tickets and code (but there can also be wrong links, as described below). The decision coverage is calculated with the maximal link distance (number of hops) of three from requirements and code elements to include transitively (indirectly) linked decisions. For example, developers can access a decision from a requirement if this decision is documented in a comment of a work item linked to the requirement. Requirements considered are scenarios, user stories, and system functions because of their similar granularity instead of user tasks and epics. Table 9.3 on page 165 provides numbers and Figure 9.8 on page 176 visualizes boxplots and histograms for the decision coverage.

The car charging app project is not included in Figure 9.8 because no requirement (scenario) is transitively linked to decisions. Decisions might be documented for the requirements but not linked. For instance, a scenario for the car charging app is called *Manage own bookings*, and an issue is *How to determine free time slots for booking?* Both seem to be related, but they are not linked and, thus, not traceable. In general, the requirements of the iPraktikum and the ISE 19/20 projects have a low decision coverage. A low decision coverage indicates requirements and code files with missing decision making (unsolved issues), missing decision documentation, or missing links. During these projects, ConDec did not offer a quality checking and nudging mechanism (Section 7.6.4) that frequently confronts the developers with a low decision coverage.

In the ISE 20/21, ISE 21/22, and ConDec projects, the majority of the requirements (93.9 % – 100 %) is covered with more than one decision, as requested by the definition of done. In these projects, the requirements and code files are covered with 2 – 17 decisions on average (mean value in Figure 9.8), which are reasonable numbers for exploitation. The decision coverage varies for the requirements and code files (standard deviation 2 – 27 in Figure 9.8). The decision coverage of code files varies stronger than the decision coverage of requirements, maybe due to the heterogeneity of code written in different languages.

The maxima of traceable decisions in link distance three can be very high, e. g., 225 decisions can be reached from the *atlassian-plugin.xml* file of the ConDec Jira plug-in. It would not benefit developers to see 225 or 44 decisions when working on code or requirements, respectively. The high numbers can be due to the following reasons: 1) Trace links between code files and work items can be wrong because of tangled commits. Configuration files such as the *atlassian-plugin.xml* and *pom.xml* in ConDec or the *schema.xml* in the web search engine development were often changed. They are linked to tickets and transitively related to decisions that do not explain

the changes in these files. Thus, a very high decision coverage can be used as an indicator to detect wrong links in the knowledge graph. 2) A very high decision coverage can indicate decision "hot spots" in that it might be worth splitting a requirement or code file. For instance, the system function *Manually classify text in the description or comments of a Jira ticket as decision knowledge* of the ConDec project is covered with 44 decisions since its implementation was complex. 3) The calculation of the decision coverage might traverse useless links. Currently, link distance three is used in all directions. However, it might be useless to see the decisions directly linked to a user story from other user stories of an epic.

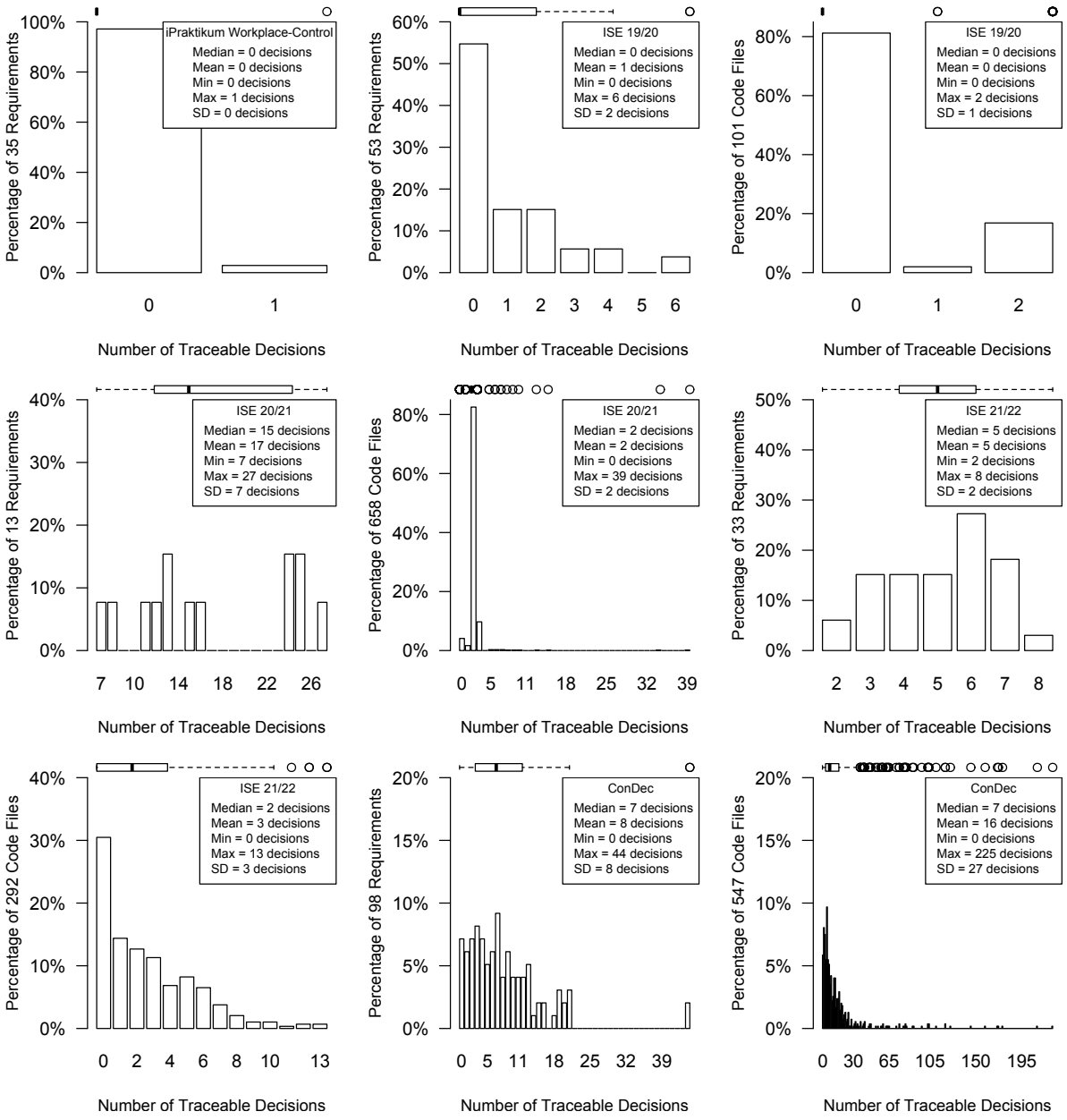

Figure 9.8.: Decision coverage of requirements and code in a maximal distance of three links. For each project, the number of traceable decisions from requirements and code (if available) is shown as boxplots and histograms, as well as the median, mean, minimum, maximum, and standard deviation (SD).

Discussion: Is it feasible to create high-quality knowledge documentation with ConDec?

We showed that it is feasible to create high-quality knowledge documentation using the following indicators: 1) Most code elements in the projects are traceable from at least one ticket in the issue tracking system, which supports knowledge exploitation. 2) Most decision knowledge elements of the ISE projects are completely documented according to four criteria of intra-rationale completeness, in particular, if checked and enforced by ConDec, in addition to promoting completeness through guidelines. 3) The ISE projects contain only a few unsolved issues. 4) The decision coverage of requirements and code files fulfills the definition of done checked and enforced by ConDec.

In the future, the semantic content of the rationale documentation should be analyzed. ConDec supports rational and naturalistic decision documentation (Section 2.2.2), but naturalistic decisions might not be the best solution. An example is the issue *Which technology shall the backend use?* with the decision *Use NodeJS!* and the pro-argument *We have more experience with NodeJS than with Flask.* In few cases, the documentation deviates from the ConRat knowledge model (Section 6.1): The decision *Use a single stop word file to keep maintenance low!* includes the pro-argument *to keep maintenance low.* The development team can configure the rationale model and decide on the format. ConDec supports creating consistent, correct, up-to-date, and unique documentation by 1) making the decision knowledge traceable from requirements, other tickets, and code, 2) offering easily accessible knowledge visualizations and frequently presenting the documented knowledge, and 3) recommendation systems. Next to tools, dissemination and guidelines are important to support high quality.

9.3. Related Work

This section discusses related work on analyzing rationale documentation created in software development projects. While there are approaches to identify and mine rationale retrospectively, this section discusses studies in which developers documented rationale during the project. Table 9.4 compares the study in this chapter with the related studies. The iPraktikum and ISE projects are similar in duration and team size, while the ConDec project differs.

Alkadhi et al. (2017a) developed the *Rationale ExtrAction from Communication arTifacts (REACT)* approach to capture rationale in chat messages and applied it in two studies. In their first study, they applied REACT in a short-term design task with eleven teams. In their second study, they applied REACT in an iPraktikum 2017/18 project over two months with one team of ten developers. The setting of the second study by Alkadhi et al. (2017a) is similar to the iPraktikum and ISE projects used for treatment validation in this dissertation. The developers captured about five decisions in chat messages, while in the iPraktikum and ISE projects of this thesis, the developers captured 11 – 116 decisions. Alkadhi et al. (2017a) analyzed different documentation aspects than us, namely the correctness of the rationale annotations and the collaborativeness of developers. The studies are also published in Alkadhi (2018).

Schubanz and Lewerentz (2020) performed eight case studies between 2015 and 2019 with software engineering students. In total, about 400 students formed 82 development teams, resulting in an average team size of about five. These projects are similar to the iPraktikum and ISE projects. Each team developed a software product during the lecturing period of one semester, i. e., about 15 weeks. The decisions are documented in *Markdown Architecture Decision Record (MADR)* templates (Kopp et al., 2018; Kopp and Armbruster, 2019). On average, nine decisions were documented by each team, while in the iPraktikum and ISE projects, the developers captured 11 – 116 decisions. Schubanz and Lewerentz (2020) analyzed the decision types documented by the students. They used a partly different coding scheme for the decisions. They split the executive decision type into the types *development platform*, *prioritize features*,

Table 9.4.: Characteristics of empirical studies that analyze rationale documentation created by developers, i. e., not created retrospectively by researchers.

	This Thesis		**Alkadhi et al., 2017a, Alkadhi, 2018**		**Schubanz and Lewerentz, 2020**
Project/ Setting	2 iPraktikum and 3 ISE projects	ConDec project	short-term design task	iPraktikum project	8 case studies in 3 universities
#Teams	5	1	11	1	82
#Developers	5 – 9 per team	26 (over time)	≈ 10 per team	10	≈ 5 per team
Duration	≈ 6 months	6 years	20 min	2 months	≈ 15 weeks
Tool/Approach	ConDec		REACT		MADR
Documentation Locations	issue tracking system, code and commit messages in version control system		chat messages		markdown files in version control system
#Rationale Elements	99–774 per team 1536 in total	2732	≈ 31 per team, 342 in total	32	702 + **?** in total
#Decisions	11 – 116 per team, 375 in total	771	≈ 8 per team, 79 in total (23 % of 342)	≈ 5 (15 % of 32)	≈ 9 per team, 702 decisions in total
Analyzed Aspects of Decision Knowledge Documentation					
Intra-Rationale Completeness	✓	✓	✗	✗	✓
Decision Coverage	✓	✓	✗	✗	✗
Decision Types	✓	✓	✗	✗	✓
Status	✓	✓	✗	✗	✗
Evolution	✓	✓	✗	✗	✓
Correctness of Annotations	✗	✗	✓	✓	✗
Collaborativeness	✗	✗	✓	✓	✗

development tools, and *development process*. We combined these decision types into the *executive* decision type. They also had a type for decisions on *external libraries and frameworks*, for *quality-driven* decisions (*software quality*) and *functionality-driven* decisions (*definition or refinement of features/requirements*). They did not have decision types for the *frontend*, *backend*, and *API* but combined these types into the *software architecture* type. However, we introduced these decision types because they were important to the practitioners in the interview study (Chapter 3) and the developers used the types during the validation projects. Schubanz and Lewerentz (2020) only assigned one decision type per decision, while ConDec allows multiple assignments. The most frequent decisions concerned the *definition or refinement of features/requirements* (19 %). In our study, 23 % of all 1146 decisions were classified as *functionality-driven*, which is a similar result. Schubanz and Lewerentz (2020) also investigated aspects of intra-rationale completeness. In their documentation, 94 % of the decisions have an alternative documented, while this number varies between 30.3 % and 94 % in the validation projects of this thesis. With ConDec, the rationale manager can tailor the definition of done to enforce the documentation of alternatives. Schubanz and Lewerentz (2020) also visualized the evolution of the decision types over time as done in Figure 9.5. Similar to the results of this thesis, they found no simple answer to which types of decisions are documented when in the project.

A contribution of this thesis is the analysis of the decision coverage and the status of issues and decisions in a decision knowledge documentation created by developers (Table 9.4).

9.4. Threats to Validity

This section discusses four validity criteria of primary empirical studies as defined by Easterbrook et al. (2008) and Runeson et al. (2012):

Construct validity focuses on whether the theoretical constructs are measured and interpreted correctly. A threat is that the decision coverage and intra-rationale completeness metrics we measured have limited suitability to answer the research questions of whether the decision knowledge is completely documented for requirements, code, and within. They are merely syntactical and do not consider the semantic meaning and usefulness of the documented knowledge. In addition to the metrics, this chapter contains examples of decisions of the validation projects to minimize the threat of cutting off too much detail with *Ockham's razor*.

Internal validity concerns whether the results we draw really follow from the data, e. g., whether confounding factors influence the results. The students might have documented decision knowledge only because they received credits and grades when working on the projects. To mitigate this threat of a conflict of interest, we explicitly pointed out to the students that the rationale management will not affect the grading.

External validity addresses the generalizability of the study results. The generalizability of the results based on six projects with University students as developers is limited. However, the instantiation of ConRat and applying the ConDec plug-ins worked well in different domains. The students created thorough rationale documentation in a limited amount of time while they had to learn various software development workflows, tools, and technologies. Thus, we expect applying ConRat and ConDec in other industry projects is also feasible.

Reliability validity concerns the study's dependency on specific researchers. A threat to the reliability validity is that the coding of decision types was only performed by the author of this thesis. However, the author is very familiar with the decisions in the validation projects since she was either part of the development team (in ConDec), supervising or monitoring the projects. In addition, the supervisors of this thesis supervised the analysis. Appendix A provides the coded data and analysis scripts for transparency and repeatability.

9.5. Conclusion

This chapter presented an empirical study to analyze decision knowledge documentation in relation to other software artifacts of six validation projects. The contribution is twofold: First, the study showed that it is feasible to document a high amount of high-quality decision knowledge during ConRat with the ConDec plug-ins. Second, the study contributes empirical knowledge about rationale documentation in CSE projects: a) The developers documented 11 – 116 decisions and related rationale in projects lasting about six months with ConDec, continuously throughout the project and mainly in the description and comments of tickets or as entire tickets. The developers of the ConDec project documented 771 decisions and related rationale, also in commit messages and code comments. b) The developers constantly updated rationale elements throughout the projects, maybe because ConDec presented them. c) The developers documented different types of decisions throughout the projects. The *decision types* with most decisions assigned are *frontend*, *backend and data storage*, and *functionality-driven* decisions, followed by *executive* and *quality-driven* decisions. The decision types can overlap: *Testing* decisions are often *executive* but not always. While the documentation locations of ConDec are interchangeable, developers can tend to (not) document specific types of decisions in particular locations, e. g., executive decisions not in code comments. d) Most decision knowledge elements are completely documented according to four criteria of the *intra-rationale completeness* if checked and enforced by ConDec. e) The projects with a duration of about six months resulted in between 27 to 68 documented requirements as tickets and 101 to 658 code files in the knowledge graph. Most of the code files

are traceable from at least one ticket, which supports knowledge exploitation. The *decision coverage* of the requirements and code files varies. A low decision coverage indicates requirements and code files with missing decision making or missing documentation (of decisions or links). A very high decision coverage indicates wrong links in the knowledge graph or decision "hot spots" in that it might be worth splitting a requirement or code file. f) The study analyzed the states of issues and decisions: The final knowledge documentation of the ISE projects contains only a few unsolved issues (0 – 4). Less than ten percent of the decisions are rejected.

ConDec operationalizes the quality of knowledge documentation. Practitioners can reflect on the knowledge documentation quality of their projects using these metrics. Like test-coverage measurement, the measurement of decision coverage and intra-rationale completeness is now a standard part of CSE. The study provides the basis for further work on developing guidelines for creating high-quality decision knowledge documentation and tools to support high quality.

Chapter 10

Effectiveness of Automatic Text Classification

"The proper place to study elephants is the jungle, not the zoo. The proper place to study bacteria is the laboratory, not the jungle."
—Stol and Fitzgerald, 2018

This chapter contributes to the knowledge goal 4 of the thesis: *Show the effectiveness of automatic text classification from the researchers' perspective.* It presents an empirical study to validate the effectiveness of ConDec's automatic text classification described in Section 7.6.8.

Section 10.1 describes the study design. Section 10.2 presents and discusses the validation results. Section 10.3 discusses related work on automatic text classification for rationale management and compares the results of the effectiveness validation. Section 10.4 discusses threats to validity. Section 10.5 concludes this chapter. Appendix A contains the ground truth data and describes how to reproduce the results with ConDec.

A first study concerning the validation of ConDec's automatic text classification with the decision knowledge of the ConDec project as the ground truth was published in Kleebaum et al. (2021b). This chapter extends the previous study by using more data from different projects for training and cross validation and comparing machine learning algorithms.

10.1. Study Design

Section 10.1.1 introduces the research questions. To train and validate the text classifier, researchers need a *ground truth*, also called *gold standard*. Section 10.1.2 describes the ground truth. Section 10.1.3 introduces evaluation metrics. Section 10.1.4 describes the procedure.

10.1.1. Research Questions

The knowledge goal 4 is refined into a research question with four sub-questions. Table 10.1 gives an overview of the questions and metrics. This section describes the questions.

RQ1 How effective is the automatic text classification of ConDec at identifying rationale elements?

The research question aims to determine the effectiveness of ConDec's automatic text classification in detecting and classifying rationale into issues, alternatives, decisions, pros, and cons. It compares different training and validation settings and machine-learning algorithms.

Table 10.1.: Research questions and metrics of the empirical study on the effectiveness of automatic text classification.

	Research Question	Metrics
RQ1	How effective is the automatic text classification of ConDec at identifying rationale elements?	
RQ1.1	How effective is the automatic text classification of ConDec at identifying rationale elements within the **same project that it was trained in** using cross validation?	Precision, recall, and F-scores for data of single project
RQ1.2	How effective is the automatic text classification of ConDec at identifying rationale elements when being **trained on the data of a different project than being validated**?	Precision, recall, and F-scores for cross-project validation
RQ1.3	How effective is the automatic text classification of ConDec at identifying rationale elements when being trained and cross validated on the **combined data of different projects**?	Precision, recall, and F-scores for combined data of different projects
RQ1.4	Which **supervised machine learning algorithm** is most effective for the different training and validation settings?	Number of times algorithms achieve best F-scores

RQ1.1 *How effective is the automatic text classification of ConDec at identifying rationale elements within the same project that it was trained in using cross validation?* We use the data of various projects as the gold standard for training and cross validation. We aim to show that the text classifier is effective when trained within a project (either retrospectively or online during the development) and then used to detect and classify new rationale elements in this project.

RQ1.2 *How effective is the automatic text classification of ConDec at identifying rationale elements when being trained on the data of a different project than being validated?* We perform cross-project validation on the data of various projects: We use the data of a single project as the gold standard for the training and then validate the trained classifier on the data of a different project. ConDec allows training a text classifier in one project and using the pre-trained classifier in another. We aim to test the generalizability of the classification models across projects.

RQ1.3 *How effective is the automatic text classification of ConDec at identifying rationale elements when being trained and cross validated on the combined data of different projects?* We combine the data of different projects into a single gold standard for training and cross validation. We aim to evaluate whether the text classifier becomes more effective when trained on a more extensive data set of different domains.

RQ1.4 *Which supervised machine learning algorithm is most effective for the different training and validation settings?* This question compares the effectiveness of supervised machine learning algorithms for detecting and classifying rationale into issues, alternatives, arguments, and decisions. It compares the algorithms for the different training and validation settings: data from a single project, cross-project validation, and combined data from multiple projects.

10.1.2. Ground Truth Data

We evaluated the effectiveness of the automatic text classification on decision knowledge documentation created with ConDec during CSE (Chapter 9) and on retrospectively annotated data by Alkadhi (2018). First, we used the data from the ConDec project and the three ISE projects as ground truth, i. e., the gold standard. We did not use the documentation of the iPraktikum projects since, at this time, ConDec did not yet offer the feature to document decision

knowledge in ticket descriptions and comments and, thus, is not representative. Second, we used the issue tracking data by Alkadhi (2018), who retrospectively annotated ticket comments of the three open-source projects Apache Lucene, Ubuntu, and Mozilla Thunderbird. In contrast to the data created with ConDec, some text parts annotated by Alkadhi (2018) have more than one rationale type. We removed these multi-labeled text parts because ConDec's fine-grained text classifier predicts one rationale type. Figure 10.1 illustrates how retrospectively annotated decision knowledge in a ticket of the Apache Lucene project would appear in ConDec.

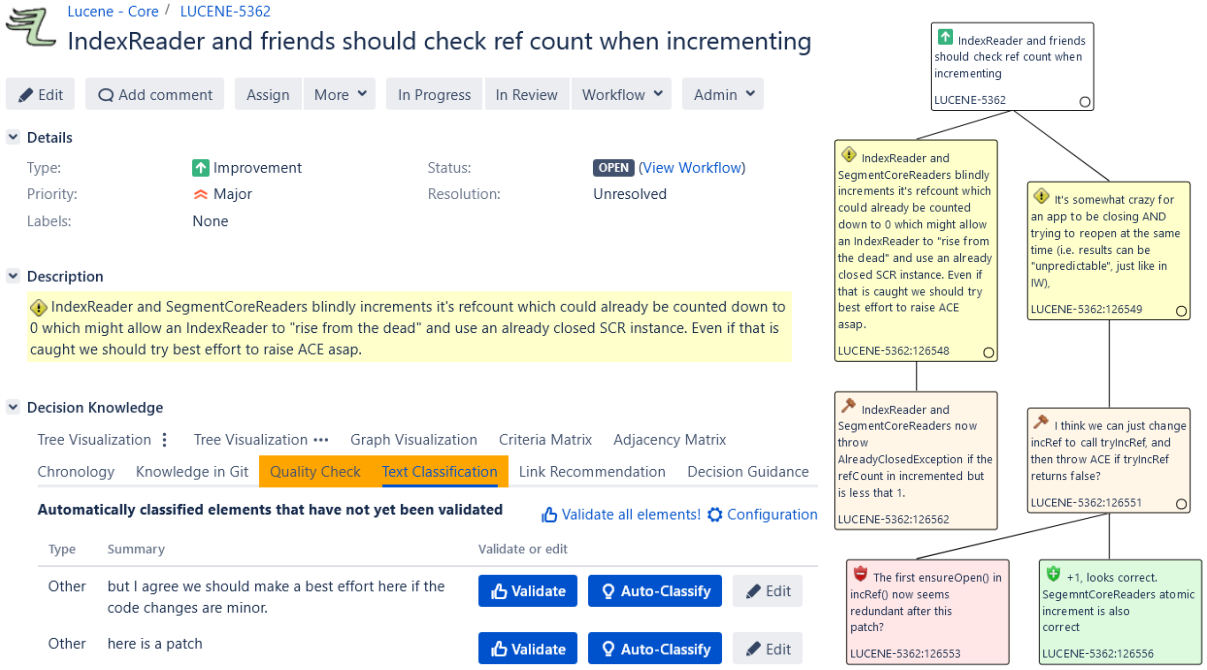

Figure 10.1.: Example of retrospectively annotated decision knowledge in ConDec. **Left:** Jira ticket of the Apache Lucene project with an issue in its description. The text classification view shows two irrelevant text parts (type *other*) that need to be manually approved, i. e., validated. **Right:** Knowledge tree (node-link tree diagram, $V2_{nld}$) of retrospectively annotated rationale elements in the ticket comments.

The ground truth consists of decision knowledge elements, called *relevant* text parts, and *irrelevant* text parts. Table 10.2 shows the number of entries in the ground truth per project, including the combined ground truths of the ISE data and the data by Alkadhi (2018).

10.1.3. Evaluation Metrics

The text classifier consists of a binary and a fine-grained classifier that are applied sequentially (Section 7.6.8). To assess their quality, the classification results of the binary and a fine-grained classifier are compared with the annotated text from the ground truth. The metrics *precision*, *recall*, and *F-scores* are used for the comparison.

Precision is the number of correct classifications, i. e., *true positives*, within all classifications made by the text classifier (Equation 10.1). The latter is the sum of true positives and incorrect classifications, i. e., *false positives*. For the binary classifier, a precision of 100 % is achieved if all identified relevant text parts are really relevant, i. e., are decision knowledge elements. For the fine-grained classifier, a precision of 100 % is achieved if all identified decisions are really decisions and analogously for the other types.

Table 10.2.: Number of text parts in the ground truth, i. e., gold standard, per project used to train and cross-validate the binary and fine-grained text classifiers. The minority class that determines the undersampling size of the other classes is bold-faced.

Data Source	#Irrelevant	#Issues	#Alternatives	#Decisions	#Pros	#Cons	#Relevant ∑
ConDec	10904	638	**336**	771	516	453	**2714**
ISE 19/20	1448	111	**94**	116	184	113	**618**
ISE 20/21	1165	**81**	118	116	262	197	**774**
ISE 21/22	818	**73**	84	76	203	154	**590**
ISE all ∑	3431	**265**	296	308	649	464	**1982**
Lucene	**661**	242	560	191	319	**158**	1470
Thunderbird	**1323**	747	484	**80**	224	228	1763
Ubuntu	**1227**	687	410	**117**	189	152	1555
R. Alkadhi ∑	**3211**	1676	1454	**388**	732	538	4788

$$Precision = \frac{\#true\ positives}{\#true\ positives + \#false\ positives} \tag{10.1}$$

Recall or *sensitivity* is the amount of true positive classifications made by the text classifier within all existing correct classifications (Equation 10.2). The latter is the sum of true positives and *false negatives*, i. e., not found rationale elements. For the binary classifier, a recall of 100 % is achieved if all relevant text parts are identified. For the fine-grained classifier, a recall of 100 % is achieved if all existing decisions within a text are identified and analogously the other types.

$$Recall = \frac{\#true\ positives}{\#true\ positives + \#false\ negatives} \tag{10.2}$$

F_β-scores combine the results for precision and recall in a single metric to judge the correctness of a text classifier. As shown in the Equation 10.3 for F_β, β can be used to weigh precision in favor of recall and vice versa.

$$F_\beta = (1 + \beta^2) \cdot \frac{Precision \cdot Recall}{(\beta^2 \cdot Precision) + Recall} \tag{10.3}$$

Berry (2017) discusses the importance of precision and recall for what he calls *hairy tasks*. Such tasks involve natural language documents and are relatively easy for humans on a small scale but become unmanageable on a large scale. Identifying and annotating decision knowledge elements in natural language text is *hairy*. The importance of precision and recall for the two application scenarios of the automatic text classification of ConDec differ:

First, the automatic text classification can be *applied retrospectively*, for example, to establish ConRat in a *brownfield* project with existing implicit rationale documentation in the issue tracking system and commit messages. Automatic text classification of a high amount of text needs high precision because ConDec integrates the identified decision knowledge directly into the knowledge graph rather than after manual approval. False positives in the knowledge graph confuse the developers during knowledge exploitation. The recall must also be high because false negatives lead to incomplete documentation, which impedes exploitation. Researchers commonly use the F_1-score to judge whether the retrospective application is well supported (Section 10.3). The F_1-score weighs precision and recall equally.

Second, automatic text classification is *applied during active development*. In this scenario, ConDec presents the automatic classification results to the developers constantly. We favor recall over precision in this ongoing application since the developers need to approve the identified decision knowledge manually and can directly improve false positives. We argue that false positives are a means to nudge the correct documentation; therefore, the number of false negatives must be reduced. We use the F_2-score, which emphasizes recall, to judge whether automatic text classification during active development is well supported.

To our knowledge, there are no clear thresholds for precision, recall, and F-score values in the existing literature that automatic rationale identification should at least achieve to be effective. Kurtanović and Maalej (2018) filtered all results having precision, recall, or the F1-score below 0.6. Usually, researchers compare their best-achieved precision, recall, and F-scores with related work. This study makes the following assumptions: ConDec's automatic text classification should be better than guessing. For binary classification, values > 0.5 are better than guessing because of a 50/50 chance to decide whether a text contains relevant rationale (since the ground truth data is balanced). For fine-grained classification, values > 0.2 are better than guessing because of a 1/5 chance to decide the rationale type. Furthermore, ConDec's automatic text classification should be at least as effective as related approaches, discussed in Section 10.3.

10.1.4. Evaluation Procedure

We trained and validated the automatic text classification in ConDec. Figure 10.2 shows a screenshot of ConDec's configuration and evaluation view for the automatic text classification. ConDec automatically preprocesses the data in the ground truth as explained in Section 7.6.8, except for sentence splitting because the data entries were already split. We created the binary and fine-grained classifiers with the three supervised machine-learning algorithms Logistic Regression (LR), Naïve Bayes (NB), and Support Vector Machine (SVM). We decided to use these algorithms because they were used in the related work and are easy to implement using the *Statistical Machine Intelligence and Learning Engine (SMILE)* library.

The gold standard in Table 10.2 is *imbalanced*, i. e., the classes vary in the number of text parts. A classifier trained on an imbalanced dataset can be biased toward the majority class, i. e., the class with the higher number of instances, and neglect the minority class (Chawla et al., 2004). To avoid such bias, ConDec balances the data before performing cross validation. ConDec uses *undersampling* to balance 1) relevant and irrelevant text parts and 2) the decision knowledge elements by their type. Undersampling means that only a subset of the majority class is used for the training and validation. ConDec applies undersampling by selecting the first data entries in the ground truth instead of random selection, which makes the validation results reproducible. For example, of the 3431 irrelevant text parts in the ISE ground truth (Table 10.2), ConDec uses 1982 for the binary classifier because this is the number of relevant text parts. Of the 687 issues in the Ubuntu ground truth, ConDec's fine-grained classification only uses 117 issues to balance the data. In this ground truth, the 117 decisions represent the minority class (Table 10.2). An alternative balancing technique is *oversampling*. The *Synthetic Minority Over-Sampling Technique (SMOTE)* enables synthetically creating entries of the minority class for oversampling (Fernandez et al., 2018). We decided to use undersampling instead of oversampling because it reduces the size of the training data and, thus, reduces the computational cost for training and validation. Since training and validation of automatic text classification are functionalities of the ConDec Jira plug-in, resources are limited to those of the underlying system. ConDec restricts the maximum training data size of support vector machines to 6000 3-grams to avoid out-of-memory exceptions of the underlying Jira system.

When using the same data set for training and validation, we applied *10-fold cross validation* to train and evaluate the binary and fine-grained classifiers. In 10-fold cross validation, the

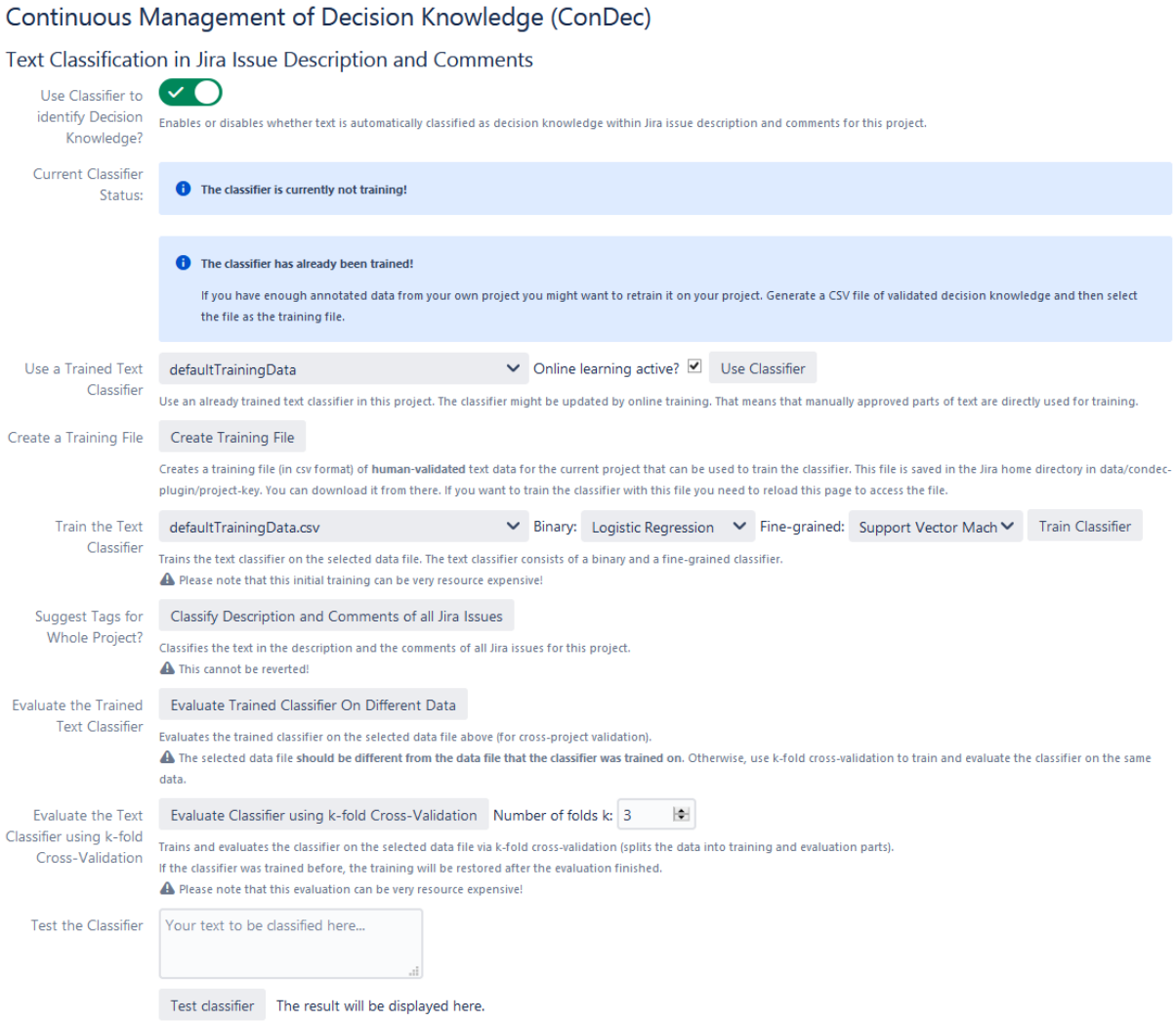

Figure 10.2.: ConDec's configuration view for automatic text classification. The view enables to train and validate the binary and fine-grained classifiers.

ground truth is split into ten parts of equal sizes, called folds. Nine folds are used for training the classifier, and the remaining fold is used for validation. The procedure is repeated ten times, rotating the training and validation folds (James et al., 2021). The final evaluation is calculated by averaging the results of the ten iterations. We also applied *cross-project validation*, which means that we trained the binary and fine-grained classifiers on the data of one project and validated them on the data of a different project.

10.2. Results and Discussion

The following sections present and discuss the results for the research question *How effective is the automatic text classification of ConDec at identifying rationale elements?* (RQ1). Section 10.2.1 provides the results for individual projects, Section 10.2.2 for the cross-project validation, Section 10.2.3 for the combined data from different projects, and Section 10.2.4 for the comparison of machine-learning algorithms. This section partly discusses the results, and Section 10.3 will further discuss the results compared with related work. This chapter provides the F-scores as the aggregated results in Table 10.3 – Table 10.5. Appendix E contains the precision and recall

values. The best results are bold-faced. Results better than guessing are colored green, whereas results worse than guessing are colored red; threshold values are white.

Table 10.3.: Evaluation results of the binary and fine-grained classifiers on the data of single projects using 10-fold cross validation. The machine-learning algorithms are Logistic Regression (LR), Naïve Bayes (NB), and Support Vector Machine (SVM).

Metric	Project	Alg.	Binary (Relevant/Irrelevant)	Issue	Alternative	Decision	Pro	Con
F1	ConDec	LR	0.66	0.39	0.1	0.39	0.1	0.32
	ConDec	NB	0.7	0.52	0.24	0.36	0.21	0.44
	ConDec	SVM	0.63	0.55	0.33	0.4	**0.42**	0.33
	ISE 19/20	LR	0.72	0.57	0.21	0.41	0.23	0.47
	ISE 19/20	NB	0.7	0.57	0.36	**0.49**	0.18	0.42
	ISE 19/20	SVM	0.63	0.62	0.1	0.41	0.36	0.48
	ISE 20/21	LR	0.65	0.48	0.02	0.24	0.26	**0.54**
	ISE 20/21	NB	**0.81**	**0.69**	0.46	0.07	0.39	0.35
	ISE 20/21	SVM	0.7	0.54	0.08	0.1	0.34	0.26
	ISE 21/22	LR	0.75	0.56	0.37	0.36	0.11	0.45
	ISE 21/22	NB	0.71	0.54	0.33	0.4	0.3	0.07
	ISE 21/22	SVM	**0.81**	0.65	**0.47**	0.42	0.22	0.34
	Lucene	LR	0.68	0.35	0.22	0.39	0.14	0.23
	Lucene	NB	0.65	0.29	0.34	**0.49**	0.4	0.38
	Lucene	SVM	0.71	0.31	0.3	0.29	0.23	0.16
	Thunderbird	LR	0.69	0.42	0.36	0.4	0.23	0.27
	Thunderbird	NB	0.65	0.24	0.15	0.45	0.32	0.1
	Thunderbird	SVM	0.68	0.46	0.43	0.48	0.25	0.34
	Ubuntu	LR	0.67	0.04	0.18	0.39	0.16	0.36
	Ubuntu	NB	0.66	0.03	0.07	0.26	0.34	0.37
	Ubuntu	SVM	0.65	0.13	0.33	0.4	0.12	0.37
F2	ConDec	LR	0.63	0.32	0.07	**0.6**	0.07	0.26
	ConDec	NB	0.83	0.46	0.21	0.36	0.17	0.58
	ConDec	SVM	0.58	0.54	0.31	0.45	0.41	0.31
	ISE 19/20	LR	0.72	0.53	0.19	0.52	0.19	0.48
	ISE 19/20	NB	0.68	0.53	0.44	0.55	0.14	0.38
	ISE 19/20	SVM	0.56	0.61	0.08	0.52	0.33	0.44
	ISE 20/21	LR	0.59	0.66	0.01	0.18	0.23	0.56
	ISE 20/21	NB	**0.91**	**0.74**	**0.6**	0.05	0.36	0.31
	ISE 20/21	SVM	0.66	0.65	0.06	0.07	**0.43**	0.23
	ISE 21/22	LR	0.71	0.49	0.33	0.34	0.08	**0.62**
	ISE 21/22	NB	0.66	0.52	0.37	0.41	0.33	0.05
	ISE 21/22	SVM	0.78	0.63	0.43	0.35	0.19	0.44
	Lucene	LR	0.67	0.54	0.17	0.29	0.1	0.19
	Lucene	NB	0.67	0.24	0.33	0.55	0.4	0.4
	Lucene	SVM	0.85	0.47	0.26	0.2	0.17	0.13
	Thunderbird	LR	0.8	0.44	0.38	0.4	0.21	0.27
	Thunderbird	NB	0.73	0.21	0.12	0.47	**0.43**	0.07
	Thunderbird	SVM	0.79	0.44	0.44	0.39	0.21	0.43
	Ubuntu	LR	0.82	0.02	0.14	0.59	0.12	0.34
	Ubuntu	NB	0.77	0.02	0.05	0.23	0.42	0.48
	Ubuntu	SVM	0.64	0.09	0.29	0.49	0.09	0.44

Table 10.4.: Evaluation results of the binary and fine-grained classifiers for cross-project validation. The machine-learning algorithms are Logistic Regression (LR), Naïve Bayes (NB), and Support Vector Machine (SVM).

Metric	Training Project	Validation Project	Alg.	Binary	Issue	Alternative	Decision	Pro	Con
F1	ISE 19/20	ISE 20/21	LR	0.24	0.37	0.05	0.15	0.49	0.13
	ISE 19/20	ISE 20/21	NB	0.64	0.44	**0.35**	0.24	0.14	0.42
	ISE 19/20	ISE 20/21	SVM	0.69	0.45	0.09	0.27	0.52	0.2
	ISE 19/20	ISE 21/22	LR	0.37	0.39	NA	0.21	**0.54**	0.14
	ISE 19/20	ISE 21/22	NB	**0.72**	0.48	0.26	0.32	0.09	0.35
	ISE 19/20	ISE 21/22	SVM	0.61	**0.61**	0.16	0.22	**0.54**	0.16
	ISE 20/21	ISE 19/20	LR	0.37	0.11	NA	0.11	0.46	0.32
	ISE 20/21	ISE 19/20	NB	0.48	**0.61**	0.32	0.38	0.26	**0.43**
	ISE 20/21	ISE 19/20	SVM	0.52	0.33	0.02	0.09	0.47	0.34
	ISE 20/21	ISE 21/22	LR	0.42	0.23	NA	0.02	0.5	0.32
	ISE 20/21	ISE 21/22	NB	0.64	0.46	0.27	0.26	0.13	0.36
	ISE 20/21	ISE 21/22	SVM	0.68	0.41	0.02	0.08	0.53	0.29
	ISE 21/22	ISE 19/20	LR	0.59	0.15	0.02	0.07	0.44	0.26
	ISE 21/22	ISE 19/20	NB	0.58	0.51	0.31	**0.46**	0.09	0.41
	ISE 21/22	ISE 19/20	SVM	0.48	0.33	0.05	0.11	0.48	0.3
	ISE 21/22	ISE 20/21	LR	0.61	0.24	NA	0.06	0.5	0.27
	ISE 21/22	ISE 20/21	NB	0.61	0.57	0.31	0.23	0.11	0.37
	ISE 21/22	ISE 20/21	SVM	0.7	0.5	0.05	0.05	0.51	0.3
F2	ISE 19/20	ISE 20/21	LR	0.17	0.31	0.04	0.12	0.65	0.1
	ISE 19/20	ISE 20/21	NB	0.63	0.45	**0.47**	0.28	0.1	0.38
	ISE 19/20	ISE 20/21	SVM	**0.83**	0.35	0.06	0.24	0.68	0.14
	ISE 19/20	ISE 21/22	LR	0.28	0.35	NA	0.18	**0.71**	0.1
	ISE 19/20	ISE 21/22	NB	0.73	0.56	0.35	0.38	0.06	0.31
	ISE 19/20	ISE 21/22	SVM	0.79	0.52	0.12	0.17	**0.71**	0.12
	ISE 20/21	ISE 19/20	LR	0.3	0.07	NA	0.07	0.65	0.29
	ISE 20/21	ISE 19/20	NB	0.69	**0.63**	0.42	0.33	0.2	**0.47**
	ISE 20/21	ISE 19/20	SVM	0.7	0.24	0.01	0.06	0.66	0.29
	ISE 20/21	ISE 21/22	LR	0.33	0.17	NA	0.01	0.66	0.29
	ISE 20/21	ISE 21/22	NB	0.82	0.57	0.37	0.23	0.1	0.36
	ISE 20/21	ISE 21/22	SVM	0.81	0.3	0.01	0.05	**0.71**	0.24
	ISE 21/22	ISE 19/20	LR	0.59	0.1	0.01	0.05	0.61	0.25
	ISE 21/22	ISE 19/20	NB	0.59	0.44	0.43	**0.53**	0.06	0.39
	ISE 21/22	ISE 19/20	SVM	0.68	0.24	0.04	0.07	0.65	0.28
	ISE 21/22	ISE 20/21	LR	0.56	0.17	NA	0.04	0.66	0.23
	ISE 21/22	ISE 20/21	NB	0.58	0.56	0.46	0.26	0.07	0.3
	ISE 21/22	ISE 20/21	SVM	**0.83**	0.38	0.04	0.04	0.68	0.25

10.2.1. Effectiveness For Ground Truth From Single Project

This section presents the results for the question *How effective is the automatic text classification of ConDec at identifying rationale elements within the* ***same project that it was trained in*** *using cross validation?* (RQ1.1). Table 10.3 shows the results of evaluating the binary and fine-grained classifiers on the data of individual projects using 10-fold cross validation and for different machine-learning algorithms. The best F1-scores are 0.81 for binary classification, 0.69 for issues,

0.46 for alternatives, 0.49 for decisions, 0.42 for pro-arguments, and 0.54 for con-arguments. The best F2-scores are 0.91 for binary classification, 0.74 for issues, 0.6 for alternatives, 0.6 for decisions, 0.43 for pro-arguments, and 0.62 for con-arguments. The best F2-scores exceed the best F1-scores, which means that automatic text classification is more effective when weighing recall over precision. The binary and fine-grained classifiers are often better than guessing but, in some cases, are worse (red colored in Table 10.3), which are unacceptable results for automatic text classification and will be discussed in Section 10.3.

Identifying issues is more effective on the data created with ConDec during the development than on the retrospectively annotated data of the Lucene, Thunderbird, and Ubuntu projects (Table 10.3). The reason is probably that issues in the data created with ConDec are often formulated as questions. In contrast, the issues in the retrospectively annotated data are more heterogeneous, as illustrated in Figure 10.1.

10.2.2. Effectiveness For Cross-Project Validation

This section presents the results for the question *How effective is the automatic text classification of ConDec at identifying rationale elements when being* ***trained on the data of a different project than being validated****?* (RQ1.2). Table 10.4 shows the results of evaluating the binary and fine-grained classifiers for cross-project validation in the three ISE projects and different machine-learning algorithms. Alkadhi (2018) performed cross-project validation with the retrospectively created data sets of the Lucene, Thunderbird, and Ubuntu projects, and we did not repeat the experiments in this study. Section 10.3 will compare and discuss the results.

The best F-scores for the cross-project validation are smaller than for the 10-fold cross validation within a single project. Not available (NA) values in Table 10.4 indicate that the precision and recall values are zero, which are unacceptable results for automatic text classification. The results indicate that it is better to train the text classifiers on the data of the current project than on the data of a different project because the phrasing of rationale elements is specific to the project. This confirms the findings by Alkadhi (2018), who suggests using communication artifacts from the same project to train the classifiers rather than generic rationale classifiers.

Interestingly, the classification of pro-arguments is more effective when training and applying the fine-grained classifier on different projects than when using 10-fold cross validation on the data of a single project. A reason might be that similar pro-arguments, such as *required by the customer*, are documented in different projects but only once within a single project. As for the cross validation within a single project, the best F2-scores exceed the best F1-scores.

10.2.3. Effectiveness For Combined Ground Truth From Different Projects

This section presents the results for the question *How effective is the automatic text classification of ConDec at identifying rationale elements when being trained and cross validated on the* ***combined data of different projects****?* (RQ1.3). Table 10.5 shows the results of evaluating the binary and fine-grained classifiers on the combined data sets using 10-fold cross validation and for different machine learning algorithms. We created two combined data sets with the data of the three ISE projects and the data by Alkadhi (2018) because of the different settings: The developers of the ISE projects performed continuous rationale management, while Alkadhi (2018) retrospectively identified the decision knowledge.

The best F-scores for the combined data validation are smaller than for the validation within single projects. Section 10.3 will discuss the reason by comparing related classification approaches.

Table 10.5.: Evaluation results of the binary and fine-grained classifiers on the combined data of the ISE projects using 10-fold cross validation. The machine-learning algorithms are Logistic Regression (LR), Naïve Bayes (NB), and Support Vector Machine (SVM).

Metric	Data Source	Alg.	Binary (Relevant/Irrelevant)	Issue	Alternative	Decision	Pro	Con
F1	all ISE	LR	0.69	**0.64**	0.02	**0.43**	0.33	**0.46**
	all ISE	NB	**0.74**	0.61	**0.38**	0.38	**0.27**	0.41
	all ISE	SVM	0.52	**0.64**	0.19	0.27	0.41	0.42
	R. Alkadhi	LR	0.68	0.36	0.27	0.31	0.15	0.33
	R. Alkadhi	NB	0.67	0.15	0.2	0.23	0.34	0.37
	R. Alkadhi	SVM	0.68	0.33	0.32	0.18	0.2	0.25
F2	all ISE	LR	0.65	0.6	0.01	**0.56**	0.3	**0.48**
	all ISE	NB	**0.86**	0.6	**0.46**	0.39	0.21	0.38
	all ISE	SVM	0.43	**0.71**	0.16	0.22	**0.52**	0.38
	R. Alkadhi	LR	0.75	0.53	0.22	0.24	0.11	0.28
	R. Alkadhi	NB	0.78	0.12	0.15	0.18	0.43	0.46
	R. Alkadhi	SVM	0.83	0.49	0.28	0.12	0.14	0.21

10.2.4. Effectiveness Of Different Supervised Machine Learning Algorithms

This section presents the results for the question *Which **supervised machine learning algorithm** is most effective for the different training and validation settings?* (RQ1.4). Table 10.6 counts the number of times a supervised machine learning algorithm was the most effective in terms of F1- and F2-score in Table 10.3, Table 10.4, and Table 10.5. Undecided cases in that two algorithms achieved the same maximum F-score are omitted.

Table 10.6.: Number of times a machine-learning algorithm achieved the best F-score.

	Validation	Logistic Regression		Naïve Bayes		Support Vector Machine	
		F1-Score	F2-Score	F1-Score	F2-Score	F1-Score	F2-Score
Binary	Single Project	3x	3x	2x	2x	2x	2x
	Cross Project	1x	0x	1x	1x	4x	5x
	Combined Data	0x	0x	1x	1x	0x	1x
Issue	Single Project	1x	1x	1x	1x	5x	4x
	Cross Project	0x	0x	4x	6x	2x	0x
	Combined Data	1x	1x	0x	0x	0x	1x
Alternative	Single Project	0x	0x	3x	3x	4x	4x
	Cross Project	0x	0x	6x	6x	0x	0x
	Combined Data	0x	0x	1x	1x	1x	1x
Decision	Single Project	1x	3x	2x	4x	4x	0x
	Cross Project	0x	0x	5x	6x	1x	0x
	Combined Data	2x	2x	0x	0x	0x	0x
Pro	Single Project	0x	0x	5x	4x	2x	3x
	Cross Project	0x	0x	0x	0x	5x	5x
	Combined Data	0x	0x	1x	1x	1x	1x
Con	Single Project	2x	3x	2x	3x	2x	1x
	Cross Project	0x	0x	6x	6x	0x	0x
	Combined Data	1x	1x	1x	1x	0x	0x

The Naïve Bayes and support vector machine classifiers often outperformed the logistic regression classifiers regarding best F-scores. For the project cross validation, the logistic regression did only once achieve the best F1-score in binary classification. This indicates that the Naïve Bayes and support vector machine are more effective for detecting and classifying new data that differs from their training data than the logistic regression. The support vector machine outperformed the other classifiers in detecting decision knowledge (binary classification) and pro-arguments classification. The Naïve Bayes classifiers outperformed the other classifiers in classifying issues, alternatives, decisions, and con-arguments.

10.3. Related Work

This section discusses related work on automatic text classification for rationale documentation. It includes the publications on automatic text classification found in the systematic mapping study in Chapter 4 that measure precision, recall, and F-score for effectiveness validation. Table 10.7 compares ConDec's automatic text classification and the results of the effectiveness evaluation with seven related publications.

The approaches *classify and are validated on different data* (Table 10.7): Four related approaches classify tickets, e. g., chrome bug reports. Two approaches classify chat messages from Gitter, Slack, or Internet relay chat. One approach classifies design discussion transcripts and another approach classifies messages from a mailing list. ConDec's automatic text classification is the only one that also classifies commit messages. Kurtanović and Maalej (2018) classify user rationale, whereas the other approaches (including ConDec) classify developer rationale.

All the approaches require a *ground truth for training and validating the supervised machine-learning classifiers.* ConDec is the only tool that provides labeling support in Jira. With ConDec, developers or researchers can directly annotate text as decision knowledge elements in the description and comments of tickets (Section 7.3.2). Other researchers labeled text parts with the *General Architecture for Text Engineering (GATE)* tool to create the ground truth (Table 10.7).

All except one approach by Kurtanović and Maalej (2018) perform a *binary classification* to detect text parts with rationale, decisions, or issues before the more detailed (fine-grained) classification (Table 10.7). The binary classification resulted in best F1-scores between 0.68 to 0.95 (for ConDec 0.81). The approaches aim to identify different rationale types or decision types for *fine-grained classification.* ConDec and five other approaches identify different rationale types, e. g., the issue (decision problem), alternatives, the decision, as well as pro- and con-arguments. The approaches by Bhat et al. (2017b) and Fu et al. (2021) classify decisions into decision types. The fine-grained classification is less effective than the binary classification since it resulted in best F1-scores between 0.36 to 0.83 (for ConDec 0.42 to 0.69).

Three approaches do not apply *data balancing* of the ground truth, i. e., they worked with an imbalanced data set (Rogers et al., 2015; Kurtanović and Maalej, 2018; Lester et al., 2020). The other approaches, including ConDec, apply undersampling for data balancing of the ground truth. Alkadhi (2018) applies a combination of undersampling and oversampling using SMOTE. The approaches use and experiment with various *classification features and preprocessing steps.* Basic preprocessing techniques are tokenization and lowercase conversion. ConDec is the only approach that uses *Global Vectors for Word Representation (GloVe)* for word-to-vector conversion and it uses 3-grams as classification features, whereas other approaches use a variety of features. ConDec uses GloVe because GloVe considers the semantics of words, which is helpful for classification (Pennington et al., 2014). Others use *Term Frequency-Inverse Document Frequency (TF-IDF)*, (continuous) bag-of-words, skip-gram, and Word2Vec. Further preprocessing techniques applied are part-of-speech tagging, word combinations (n-grams), sentence length, stemming and lemmatization, and stop-word removal. The approaches apply and experiment with various *classification algorithms*, e. g., Naïve Bayes, SVM, Logistic Regression,

Decision Tree, and Random Forest. One approach uses a convolutional neural network (CNN) and the pre-trained *Bidirectional Encoder Representations from Transformers (BERT)* model. The algorithms' effectiveness varies by the data and classification tasks, as observed in Section 10.2.4.

Table 10.7.: Characteristics of empirical studies that evaluate automatic text classification for rationale documentation.

Publication	What is Classified?	Binary Results	Fine-Grained Results	Preprocessing, Features; Classifiers; External Dependencies and Tools
This Thesis: ConDec's Automatic Text Classification	Text parts from descriptions and comments of tickets and commit messages transcribed into ticket comments (Table 10.2)	F1 up to 0.81 (rationale/ non-rationale)	F1 up to 0.69 (issues), 0.47 (alternatives), 0.49 (decisions), 0.42 (pro-arguments), 0.54 (con-arguments)	Sentence splitting, lowercase, tokenization, GloVe, 3-grams, undersampling for data balancing, no stop-word removal and no stemming; Naïve Bayes, SVM, Logistic Regression; SMILE
Alkadhi (2018): A-REACT	Ticket comments from Apache Lucene, Ubuntu, and Mozilla Thunderbird; chat messages from Slack and internet relay chat	F1 up to 0.95 (rationale/ non-rationale for tickets)	F1 up to 0.76 (issues), 0.77 (alternatives), 0.7 (decisions), 0.62 (pro-arguments), 0.5 (con-arguments)	Lowercase, tokenization, stemming, n-grams, TF-IDF, undersampling and SMOTE for data balancing; Naïve Bayes, SVM, Logistic Regression, Decision Tree, Random Forest; GATE, WEKA, MEKA
Bhat et al. (2017b): ADeX	Summaries and descriptions of tickets of Apache Spark and Apache Hadoop Common: 1571 tickets, 480 labeled decisions	F1 up to 0.91 (decision/ non-decision)	F1 up to 0.83 for classification of decisions into decision types *structural*, *behavioral*, and *ban* (Section 2.2.4)	Lowercase, tokenization, stop-word removal, stemming, n-grams, TF-IDF, random sampling for data balancing; Naïve Bayes, SVM, Logistic Regression, Decision Tree; MLlib, LibSVM
Kurtanović and Maalej (2018)	1020 user reviews sampled from 52 software applications, classify user rationale	no binary classification into rationale and non-rationale	F1 up to 0.77 (issues), 0.82 (alternatives), 0.83 (decisions), 0.74 (justifications), 0.77 (criteria)	Stop-word removal, lemmatization, bag-of-words, n-grams, part-of-speech, ...; Naïve Bayes, SVM, Logistic Regression, Decision Tree, Gaussian Process Classifier, Random Forest, Multilayer Perceptron; NLTK, Standford Parser, scikit-learn
Lester et al. (2020)	200 chrome bug reports and transcribed design discussions	F1 up to 0.77 for chrome (rationale/ non-rationale)	F1 up to 0.41 (decisions), 0.48 (alternatives), 0.39 (arguments) for chrome	Sentence parsing and shuffling, n-grams, part-of-speech, ...; use evolutionary algorithms to optimize feature selection; Naïve Bayes; GATE, WEKA, NLTK
Li et al. (2020) and Fu et al. (2021)	Hibernate developer mailing list, 650 decision sentences and 650 non-decision sentences	Li et al., 2020: F1 up to 0.76 (for decision/ non-decision)	Fu et al., 2021: F1 up to 0.73 for classification of decisions into *design*, *requirement*, *management*, *construction*, *testing*	Stop-word removal or inclusion, stemming and lemmatization, filtering by sentence length; feature extracting using (continuous) bag-of-words, TF-IDF, Word2Vec, skip-gram; Naïve Bayes, SVM, Logistic Regression, Decision Tree, Random Forest; NLTK, scikit-learn, gensim
Rogers et al. (2015)	200 chrome bug reports (17410 sentences without rationale, 2131 sentences with rationale, e. g., 424 decisions, 352 alternatives, 494 arguments)	F1 up to 0.68 (rationale/ non-rationale)	F1 up to 0.8 (decisions), 0.36 (alternatives), 0.36 (arguments)	Tokenization, part-of-speech, sentence splitting, verb group chunking, stemming, n-grams, contextual information, sentence length; Naïve Bayes, SVM, Random Forest, BayesNet; GATE, WEKA
Shi et al. (2021): Issue-Solution Pairs from communitY live chats	Chat messages from eight projects with communication in Gitter, 750 dialogs including 171 issue-solution pairs	F1 up to 0.76 (issue/ non-issue)	F1 up to 0.63 for identification of solution for issue	Spell checking, low-frequency token replacement, acronym and emoji replacement, recovery of broken utterance; convolutional neural network with BERT layer, random sampling for data balancing; TextCNN, NLTK

The approaches depend on various *libraries, frameworks, and tools*. ConDec is the only tool integrating the SMILE library since SMILE provides online-learning capabilities and is implemented in Java. Three approaches apply the *Waikato Environment for Knowledge Analysis (WEKA)*. Alkadhi (2018) uses the *Multi-Label Extension to WEKA (MEKA)* for fine-grained classification. Two approaches use the Python library *scikit-learn*. Other libraries, frameworks, and tools applied are: *Apache Spark's scalable machine learning library (MLlib)*, *LibSVM*, *Natural Language Toolkit (NLTK)*, *gensim*, the *Standford Parser*, and *TextCNN*.

Alkadhi (2018) also automatically detected and classified rationale in their retrospectively created ground truths of Apache Lucene, Ubuntu, and Mozilla Thunderbird. Alkadhi (2018) achieved F1-scores up to 0.95 for binary classification, 0.76 for issues, 0.77 for alternatives, 0.62 for pro-arguments, 0.5 for con-arguments, and 0.7 for decisions. ConDec's text classification is currently less effective. As described in Section 10.1.2, ConDec only used a subset of the data by Alkadhi (2018) due to the exclusion of multi-labeled rationale elements and the mere undersampling for data balancing, whereas Alkadhi (2018) combines undersampling with SMOTE. The different results can also be because ConDec uses GloVe instead of term frequency-inverse document frequency for converting words to numerical vectors and 3-grams as classification features. Alkadhi (2018) uses n-grams of lengths 1 to 3. Alkadhi (2018) also found out that automatic classification is more effective if the classifiers are trained on the data of the underlying project than when using classifiers trained on the data of a different project.

The best F1-scores measured with ConDec in our study are among the best F1-scores of the related work in Table 10.7, which means that ConDec can be similarly effective. However, our study's precision, recall, and F1-score results vary strongly, and ConDec's automatic text classification is sometimes worse than guessing. One reason might be the mere usage of 3-grams as classification features. For example, the 3-gram *an external library* might belong to an issue *How can we integrate an external library?*, a decision *We decided to use an external library*, an alternative *We could use an external library*, or a pro-argument *It is easier to use an external library*. The training data might classify the 3-gram into a different rationale type than the validation data, leading to wrong classifications. In general, alternatives and decisions can be hard to distinguish, even for humans (Alkadhi, 2018), so the two classes could be replaced by one class for solution options. In the future, ConDec's automatic text classification could combine multiple classification features, e. g., part-of-speech tagging. The classification could also benefit from a rule-based approach as suggested by Sharma et al. (2021). For example, many issues are formulated as questions in the data created with ConDec. A rule that classifies questions as issues could be valuable. However, this rule might not work out in the retrospectively annotated data because issues are not necessarily questions. The technical problem is that we had to restrict the training data size of support vector machine classifiers to avoid out-of-memory exceptions since the training and validation are done in Jira.

This study evaluated whether the effectiveness of the classification can be improved using more extensive data from different domains (Section 10.2.3). The best F1-scores for the combined data were smaller than when classifiers were trained and validated on the data of a single project. The reason might be that the mere usage of 3-grams as classification features was insufficient, as discussed above. None of the related studies evaluated the effect of the data size, so we cannot compare the results. It needs to be further investigated to which extent more extensive data sets can improve the effectiveness of automatic classification.

The contribution of ConDec is that it *integrates automatic text classification directly into the issue tracking system*. The seven related approaches in Table 10.7 use separate tools (or tool stacks) for retrospective rationale identification and extraction. ConDec is also the only tool that makes online learning possible: When the developers manually approve or improve the automatic classification, they contribute to the ground truth.

10.4. Threats to Validity

This section discusses four validity criteria of primary empirical studies as defined by Easterbrook et al. (2008) and Runeson et al. (2012):

Construct validity focuses on whether the theoretical constructs are measured and interpreted correctly. The effectiveness of ConDec's automatic text classification depends on the effectiveness of the binary and fine-grained classifiers since ConDec applies the binary and fine-grained classifiers sequentially. False positives of the binary classifier would be incorrect inputs, and false negatives would be missing inputs for the fine-grained classifier. In this study, we evaluated the fine-grained classifier by solely inputting relevant decision knowledge elements, i. e., we evaluated the fine-grained classifier independently of the binary classifier. The independent evaluation is also performed in the related work, enabling comparing the fine-grained results.

Internal validity concerns whether the results we draw really follow from the data, e. g., whether confounding factors influence the results. The developers or researchers who retrospectively classified implicit rationale manually approved every entry in the ground truth. However, also the manual classification is subjective and might be wrong. To mitigate the risk of wrong classification, the rationale manager constantly reviewed the knowledge documentation of the ISE and ConDec projects. ConDec supported the quality assurance with the quality checking recommendation system (Section 7.6.4) and by presenting the developers with the knowledge to nudge its improvement. For the issue tracking data of Apache Lucene, Ubuntu, and Mozilla Thunderbird, Alkadhi (2018) created coding guides. Two researchers independently coded, i. e., manually classified, the data and resolved disagreements through discussions.

External validity addresses the generalizability of the study results. We evaluated the automatic text classification on different data and used various training and validation configuration to understand to which extent the results can be generalized. We used the data of seven different projects, four in which the developers documented decision knowledge with ConDec during development and three by Alkadhi (2018) with retrospectively annotated decision knowledge. We also performed project cross validation. The precision, recall, and F-score values vary and may differ in other software projects.

Reliability validity concerns the study's dependency on specific researchers. A threat is that the experiments with different training and validation data and various machine learning classifiers were only conducted by the author of the thesis. To mitigate the threat, the digital Appendix A describes how to reproduce the results with the ConDec Jira plug-in and the ground truth data.

10.5. Conclusion

This chapter presented an empirical study to evaluate the effectiveness of ConDec's automatic text classification by running experiments with different training and validation data sets and supervised machine-learning algorithms. We performed cross-project validation and evaluated the effectiveness of the classification on projects with and without explicit decision knowledge documentation. We used the measures of precision, recall, and F-scores as indicators for effectiveness.

The evaluation revealed that ConDec can detect rationale with an F1-score up to 0.81 and an F2-score up to 0.91 and classifies the detected rationale into different rationale elements with an F1-score up to 0.69 and an F2-score up to 0.74. These results for best F1-scores are similar to the results of related work. Still, the effectiveness of ConDec's automatic text classification varies for different training data, classifiers, and classes to be classified, which complicates its use.

The best F2-scores exceed the best F1-scores, which means that automatic text classification is more effective when weighing recall over precision, i. e., if false positives are experienced as less inconvenient than false negatives. ConDec's automatic text classification supports two tasks:

First, it supports the task by the rationale manager to retrospectively formalize the implicit rationale in the development artifacts of a project in that continuous rationale management has not yet been performed. The retrospective application of automatic text classification can be of great importance in practice for brownfield development. Second, it supports the developers during their daily work in formalizing rationale by automatically detecting and classifying rationale elements and asking them to approve the classification manually. We argue that automatic text classification with higher F2-scores than F1-scores, i. e., higher recall than precision, is more effective during active development than when applied retrospectively because the developers can directly correct false positives. False positives can be seen as a mechanism to nudge the developers into improving the rationale documentation.

For future improvements, various ways exist to extend the automatic text classification and work on its evaluation. Further experiments should be performed with different preprocessing steps, classification features, oversampling using SMOTE, other classifiers, e. g., random forest, and the different training data. The language representation model BERT could replace the GloVe model, which ConDec currently uses. BERT received promising results in various natural language processing tasks (Deshpande et al., 2021). The automatic text classification could also benefit from integrating a generative pre-trained transformer model by OpenAI[1]. This study did not answer the question of why different algorithms have different effectiveness on different data. Future researchers should put effort into making the classification results explainable.

While automatic text classification can be technically improved, it is unlikely that its effectiveness can become perfect because decision knowledge is very complex. A strength of the documentation created with ConDec is that decision knowledge is more structured than the retrospectively classified documentation, which supports the exploitation. For example, issues are often formulated as questions in the documentation created with ConDec. The developers are responsible for clear formulation and structuring. Thus, the automatic text classification is most valuable when being part of ConRat, i. e., when reviewed and improved by the developers. During ConRat, the automatic classification nudges the developers to document decision knowledge in a lightweight way so that they can put more effort into decision making instead of documentation.

[1] `https://openai.com`

Chapter 11

User Acceptance of ConDec Plug-Ins

"A toolmaker succeeds as, and only as, the users of the tool succeeded with its aid."

—Brooks, 1996

This chapter contributes to the knowledge goal 5 of the thesis: *Show the acceptance of the ConDec plug-ins from the software practitioners' perspective.* It presents an empirical study on user acceptance with practitioners from industry and University students working in project courses in an industrial setting. While the study participants used ConDec in CSE, they were in the role of users. Section 11.1 describes the study design. Section 11.2 presents and discusses the results of the study on user acceptance validation. Section 11.3 discusses threats to validity. Section 11.4 concludes this chapter. An initial study of usage analysis based on logging was published in Kleebaum et al. (2021c).

11.1. Study Design

Section 11.1.1 introduces the research questions. Section 11.1.2 describes the study participants. and Section 11.1.3 presents the metrics, i. e., indicators, to measure acceptance and the data acquisition and analysis methods.

11.1.1. Research Questions

We refine the knowledge goal 5 into four research questions shown in Table 11.1. The questions address the course-grained rationale management activities demanded by the technical research goal of the thesis (Section 1.4): a) collaborative, incremental, and rational decision making, b) documentation, c) exploitation, and d) quality assurance of decision knowledge.

RQ1 Do developers accept the ConDec support for decision making?

With this research question, we aim to show that ConDec is beneficial for decision making from the point of view of software practitioners. For simplification, we omitted the specific aspects.

RQ2 Do developers accept the ConDec support for knowledge documentation?

In Chapter 9, we showed that it is feasible to document a high amount of decision knowledge related to other software artifacts with ConDec. With this research question, we want to show that software practitioners accept the ConDec support for knowledge documentation.

Table 11.1.: Research questions and metrics of the empirical study on the user acceptance.

	Research Question	Metrics
RQ1	Do developers accept the ConDec support for decision making?	Usefulness, intention to use
RQ2	Do developers accept the ConDec support for knowledge documentation?	Ease of use, usefulness, intention to use, usage frequencies
RQ3	Do developers accept the ConDec support for knowledge exploitation?	Ease of use, usefulness, intention to use, usage frequencies
RQ4	Do developers accept the ConDec support for quality assurance?	Ease of use, usefulness, intention to use, usage frequencies

RQ3 Do developers accept the ConDec support for knowledge exploitation?

With this research question, we aim to show that ConDec is beneficial for the exploitation of a high amount of distributed knowledge from the point of view of software practitioners.

RQ4 Do developers accept the ConDec support for quality assurance?

With the research question, we aim to show that ConDec is beneficial for creating and maintaining high documentation quality from the point of view of software practitioners.

11.1.2. Participants

This section provides descriptive data on the study participants, to understand their background. Table 11.2 shows their experiences in software development and rationale management.

The study mainly reports findings of the ISE 21/22 project since the ConDec features were most mature in this project. The study reports findings of the previous projects if features were not applied in ISE 21/22 and if they had consequences for the further development of ConDec. Six student software developers and two practitioners from the industry (IT consultants) were involved in the ISE 21/22 project (Table 8.2 on page 148). The consultants helped the developers to improve the development process and disseminated app technologies that the developers needed to decide on. We interviewed both the developers (ISE 21/22 D) and the consultants (ISE 21/22 C) at the end of the project. The student developers rated their experience in

Table 11.2.: Study participants' answers on statements regarding their experience and the weighted mean μ_w. "ISE 21/22 D" represents the six developers and "ISE 21/22 C" the two IT consultants from industry in the ISE 21/22 project.

Statement	Study	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	μ_w
Before the project/workshop, I was experienced with rationale management.	Workshop	0	1	1	2	0	0.2
	ISE 21/22 D	0	3	0	3	0	0
	ISE 21/22 C	0	0	1	1	0	0.5
Before the project, I was experienced in developing software for a customer.	ISE 21/22 D	0	0	3	2	1	0.7
Before the project, I was experienced in developing software in a team.	ISE 21/22 D	0	0	0	4	2	1.3
I am experienced with CSE.	Workshop	0	1	2	0	1	0.2
	ISE 21/22 C	0	0	0	1	1	1.5

developing software for a customer and in a team (Table 11.2). The IT consultants provided the time they worked in software engineering. On average, the consultants had an experience in IT projects of ten years (SD = 6.4 years). One consultant had documented rationale in two projects and exploited the rationale documented by others in one project. The other consultant had documented and exploited the rationale in more than 100 projects as a site manager.

The study also reports the findings of a one-day rationale-management workshop in the research division of an industrial company in September 2021. In total, the company has more than 100 000 employees. During the workshop, we presented ConDec to practitioners and asked them to document and exploit knowledge in a demo project. Four practitioners filled in a survey questionnaire after the workshop. They described their roles as researchers who develop prototypes but also as project leads, architects, and software developers. On average, the practitioners have an experience in IT projects of seven years (SD = 6.8 years). They documented rationale themselves in two projects (SD = 1.7) and exploited the rationale documented by others in four projects (SD = 5.5).

The descriptive data show that the study participants were less experienced in rationale management than in software development in general, confirming the interview study's findings in Chapter 3. Since all participants were experienced in software development, they have enough prerequisites to help evaluate user acceptance of ConDec.

11.1.3. Indicators for Acceptance and Research Methods

The study uses four indicators to measure the acceptance of ConDec views and features. It uses the variables of the *Technology Acceptance Model*, i. e., the participants' perceived 1) *ease of use*, 2) *usefulness*, and 3) *intention to use the solution in the future* (Davis et al., 1989; Marangunić and Granić, 2015). The participants accept a view or feature if they approve of these variables. Besides, the study assesses the 4) *usage frequencies* by asking the participants how often they used a view or feature and through usage logging. A high usage frequency is an indicator that the participants accept a feature. The following subsections describe the methods for collecting and analyzing user feedback from the study participants and usage logging.

Collecting and Analyzing User Feedback

ConDec was applied in six validation projects, i. e., the iPraktikum, ISE 19/20, ISE 20/21, ISE 21/22, and the ConDec project (Section 8.1). We observed the rationale documentation during these projects and collected developer feedback through informal discussions. In addition, we used written surveys and semi-structured interviews to collect feedback based on questionnaires (available in Appendix F). The questionnaires contain person-related questions to describe the experience of the study participants, questions to ask for their attitude toward the ConDec views and features, and questions concerning the quality of the rationale documentation in the project. The questionnaires group the ConDec views and features by the rationale-management activities they primarily support. Often, we formulated the questions in the questionnaires as statements that the participants should rate on Likert scales, e. g., *ConDec is useful for decision making.* We derived the statements in the questionnaires by considering the perceived usefulness, ease of use, and intention to use of the Technology Acceptance Model. The participants should rate these variables on a five-point Likert scale between strongly disagree to strongly agree. Next to rating statements, the study participants needed to provide detailed feedback on the features for rationale management they applied. For example, the study participants should provide details on what they think is useful or useless, easy or difficult, and why they think so. For the ConDec features, we asked the study participants to rate their usage frequency on a four-point Likert scale, i. e., never, rarely, sometimes, often. We used an even-numbered instead of an odd-numbered Likert scale because there is no neutral option for usage frequencies. We detailed

the meanings of the frequencies as follows: rarely means 0 – 3 times, sometimes means 4 – 10 times, and often means > 10 times. For the detailed ConDec features, we omitted to ask for the intention to use a feature in the future to reduce the questionnaire size, and because the ease of use and usefulness are determinants of the intention to use (Marangunić and Granić, 2015). To enable an overall rating, we calculated the weighted mean μ_w of the participants' rating on Likert scales for the usage frequencies (Equation 11.1) and the variables of the Technology Acceptance Model (Equation 11.2),

$$\frac{-2 \cdot \#\text{never} - 1 \cdot \#\text{rarely} + 1 \cdot \#\text{sometimes} + 2 \cdot \#\text{often}}{\#\text{participants}} \tag{11.1}$$

$$\frac{-2 \cdot \#\text{strongly disagree} - 1 \cdot \#\text{disagree} + 0 \cdot \#\text{neutral} + 1 \cdot \#\text{agree} + 2 \cdot \#\text{strongly agree}}{\#\text{participants}} \tag{11.2}$$

where #... represents the number of participants who selected a specific rating. For example, a weighted mean $\mu_w = 2$ means that all participants strongly agreed to a statement. A weighted mean $\mu_w = -1$ means that the participants disagreed with a statement on average.

Measurement of Usage Frequencies through Usage Logging

We performed usage analytics of ConDec's knowledge graph views to quantify the view usage and get an impression of which views the developers prefer. The ConDec implementation uses a REST API to create the knowledge graph views. We logged the usage of this REST API. The students knew that we monitor the ConDec usage for evaluation purposes, but we assured them that this does not influence the grading. We collected the data in two projects: ISE 20/21 and ISE 21/22. For the ISE 20/21 project, we started the data collection after the second sprint. For the ISE 21/22 project, we logged the usage during the entire project. We analyzed the usage of the REST API using an R script (R Core Team, 2022), available in Appendix A. We anonymized the collected data but ensured we only analyzed the REST API usage of the project participants. We analyzed the usage over time and the usage per ConDec view.

11.2. Results and Discussion

The following sections present and discuss the results of the acceptance validation. Section 11.2.1 describes the acceptance of the decision-making support. Section 11.2.2 describes the acceptance of the knowledge-documentation support. Section 11.2.3 describes the acceptance of the knowledge-exploitation support. Section 11.2.4 describes the acceptance of the quality-assurance support. This chapter reports the weighted means of the study participants' statement ratings on the Likert scales. Appendix F provides the detailed ratings.

11.2.1. Acceptance of Benefits for Decision Making

This section presents the results for the question *Do developers accept the ConDec support for decision making?* (RQ1). We asked the study participants to judge the extent to that ConDec supports decision making. Table 11.3 shows the results for the statements *ConDec is useful for decision making* and *I would use ConDec to support decision making in the future.* On average, the participants agreed on the usefulness for decision making (rating 1 – 1.5) and would use ConDec to support decision making in the future (rating 0.2 – 1).

Table 11.3.: Study participants' assessment of whether ConDec fulfills the technical research goal: answers on their perceived ease of use, usefulness, and intention to use as the weighted means μ_w of the Likert answers.

Sub-Goals of Technical Research Goal	Study	Ease of Use	Usefulness	Intention to Use
Decision-making support (*ConDec is useful for decision making* and *I would use ConDec to support decision making in the future*)	ISE 21/22 D		1.5	1
	ISE 21/22 C		1	0.5
	Workshop		1	0.2
Documentation support	ISE 21/22 C	1	1	0
	Workshop	0	1.5	0.2
Exploitation support and support for a high amount of distributed knowledge	ISE 21/22 C	1	1	1
	Workshop	0.5	0.5	0.2
High quality support	ISE 21/22 C	0	1	0
	Workshop	0	0	0

The participants gave the following positive feedback: One participant emphasized that ConDec enables collaborative decision making because all team members can discuss and reflect on decisions. ConDec could particularly support decision making in bigger teams in that not all developers can share decisions otherwise. One participant highlighted that ConDec is easy to learn and intuitive when familiar with the underlying tools.

The participants gave the following negative feedback or improvement ideas: One practitioner from the industry stated, "ConDec is not for decision making in my view; it is for decision documentation. Decision making involves discussions, investigations, weighing pros/cons, and talking to stakeholders. Just writing down options, pros, and cons is not sufficient to make a decision. ConDec is rather supporting that rationale does not get lost." The same practitioner also stated, "I am not sure the value of using such a tool is indeed much higher than simply noting down decisions in a Word document." In contrast to the practitioner, we argue that ConDec supports decision making because it makes decision knowledge explicit, which helps its reflection and discussion. The practitioners from the industry hesitate to use ConDec in the future because they partly use a different tool stack. For instance, they use Azure DevOps or Microsoft Teams instead of the issue tracking system Jira. One practitioner uses Mural as a digital whiteboard for decision making. Another practitioner requests the integration with arc42 templates. Since ConDec is a research prototype developed during a dissertation project, we could not build extensions for all the software development tools used in practice. In the future, it would be valuable to build ConDec extensions for more tools such as Azure DevOps, Teams, and Mural and to integrate templates used in the industry, such as arc42.

11.2.2. Acceptance of Knowledge Documentation Features

This section presents the results for the question *Do developers accept the ConDec support for knowledge documentation?* (RQ2). The subsections describe 1) the overall acceptance from the point of view of practitioners from the industry (workshop participants and IT consultants), 2) the acceptance of specific documentation locations and the respective documentation features, 3) the acceptance of other specific documentation features, and 4) the documentation features the study participants prefer to use in the future. In the last subsection, we discuss the results.

Acceptance of Knowledge Documentation Support by Practitioners from Industry

Table 11.3 shows the results for the statements *It is easy to document decision knowledge with ConDec*, *ConDec is useful for decision knowledge documentation* and *I would use ConDec to document decision knowledge in the future*. These statements were rated by the practitioners from the industry, i. e., the ISE 21/22 consultants and workshop participants, who only briefly used ConDec. For the student developers, who used ConDec over several months, we asked more detailed questions (cf. the following subsections). On average, the practitioners from the industry were neutral or agreed on the ease of use (rating 0 – 1) and agreed on the usefulness (rating 1 – 1.5). They were neutral or slightly agreed that they would use ConDec for decision knowledge documentation in the future (rating 0 – 0.2). The practitioners from the industry gave the following feedback: They like the structured, formalized way of decision knowledge documentation, which makes it easy to "track made decisions and their reasoning". However, one practitioner stated: "In my view, a word document, PowerPoint slide, text file, or wiki page is easier to create. These do not force the users into a specific structure. That, of course, does not allow to benefit from trace links or dependencies between decisions but is often the only option in a tight development schedule that does not allocate much time for documentation tasks. Maintaining trace links is brittle in my experience. Often a simple text search on the code repository or the documentation is sufficient to find certain artifacts and could thus substitute for trace links." For future improvements, the practitioner requests the integration with architecture models and import functionality for decision points so you do not have to start each project from scratch. The practitioner states: "Maybe pattern catalogs could be encoded in such tools and then get imported as a starting point. In my view, the point is not in having different forms, visualizations, or tracing mechanisms. These are all fine, but the core is the decision making and knowing and selecting the best choices. That is a much bigger problem for the developers than documenting their rationale." The import functionality could add to ConDec's decision guidance feature, which recommends solution options but no decision points, i. e., issues. Again, the practitioners from the industry request that ConDec plug-ins be available for the tools they use, such as Mural, Azure DevOps, and Microsoft Teams.

Acceptance of Documentation Locations and Respective Features

We asked the study participants which of the four documentation locations in the issue tracking system and version control system supported by ConDec they prefer and why. They could choose more than one location. We also asked them if they would like to document decision knowledge in another documentation location. Table 11.4 shows the answers. We included the answers of the developers of the ISE 19/20 project whom we asked for their preferred documentation location in the issue tracking system Jira. In general, the participants preferred

Table 11.4.: Study participants' answers on their preferred documentation locations for decision knowledge.

Study	Entire Jira Tickets	Jira Ticket Text	Commit Messages	Code Comments	Other
ISE 19/20	2	6			Wiki
Workshop	1	2	0	2	Free Text, PowerPoint, Wiki
ISE 21/22 D	6	3	0	0	-
ISE 21/22 C	1	1	0	0	Markdown, Mural, Wiki

different documentation locations. Some participants stated that it is a matter of habit which documentation location to use. One practitioner would like to have a "single source of truth"

instead the decentralized documentation in ConDec. However, ConDec enables the centralized access of the decision knowledge since it integrates the decision knowledge from various locations into one model (Section 6.1). When stating to prefer entire Jira tickets, the participants described the workflow to create a ticket for a decision problem and then capture other rationale elements, i. e., alternatives, arguments, and the solution decision, in its description. The advantage of the entire tickets for decision problems is that all Jira features, such as Jira filters, can be applied to the tickets instead of only the ConDec features. When stating to prefer Jira ticket text, i. e., the description and comments of existing Jira tickets such as user stories, they justified their selection as follows: It is easy to annotate the text with the decision knowledge annotations, and decisions related to requirements or work items are automatically linked in the knowledge graph. None of the participants picked commit messages as their preferred location for decision capturing. They argue that commit messages should only contain absolutely necessary information. They think it is too complicated to annotate and complete the decision knowledge in a commit message after the transcription into a comment of the related Jira ticket. Two practitioners from the industry picked code comments as their preferred documentation location as "the truth is in the code". Others argue that it reduces the readability of code if there are too many comments. One practitioner would prefer markdown files instead of code comments as done by Kopp et al. (2018) and Kopp and Armbruster (2019) through *Markdown Architecture Decision Record* templates. As further documentation locations, the participants mention wiki pages, free-text documents (possibly integrated with architecture documentation), the digital whiteboard Mural, and PowerPoint.

Further, we asked the developers who used ConDec for several months to rate the usage frequencies, the ease of use, and the usefulness of the features. Table 11.5 lists the weighted means by the study participants of two validation projects. In the ISE 19/20 project, we did not ask for features that were not existing yet or mature. Commit messages and code comments were rarely used as documentation locations during the ISE 21/22 development (ratings -1.8 and -1.7). We showed in Chapter 9 that it is feasible to document decision knowledge in commit messages and code comments by analyzing the documentation of the ConDec project. In the ISE 19/20 project, the developers applied the ConDec Slack plug-in to annotate decision knowledge in chat messages and to export the decision knowledge elements to Jira. The ISE 19/20 developers slightly disagreed on the support's ease of use and usefulness because they thought it easier to document the decision knowledge in Jira directly. Also, they used a different chat messenger in the team than Slack, for which no ConDec plug-in is available. They slightly agreed that capturing decision knowledge in wiki pages or pull requests would also be useful, which is not yet supported by ConDec.

Acceptance of Other Documentation Features

Table 11.5 shows the ratings regarding other documentation features of ConDec. We also asked the IT consultants for their perceived usefulness. In the following, we discuss the acceptance of the documentation features.

We added the *decision grouping* feature based on requests by the ISE 19/20 developers. The participants explained their positive rating of the decision grouping feature as follows: The feature is particularly useful for projects with many decisions to get an overview. It enables filtering for the important decisions for a developer, e. g., only for backend decisions if a developer is not responsible for developing the frontend.

The participants rarely used the *automatic text classification*, meaning the feature was mainly disabled (rating -0.7 and -1.3). There are two different opinions regarding the text classifier. The first opinion is that the practitioners find the text classifier easy to use and useful, but only if it has good accuracy. Otherwise, improving wrong classifications can become tedious. The second

Table 11.5.: Weighted means μ_w of the study participants' Likert ratings on their perceived usage frequency, ease of use, and usefulness of ConDec's documentation features.

ConDec Feature	Feature Description	Project	Usage Frequency	Ease of Use	Usefulness
Documentation of Decision Knowledge in ...					
Entire Jira tickets	Section 7.3.1	ISE 19/20	-0.9	0	0.3
		ISE 21/22 D	2	1.5	1.7
Description and comments of Jira tickets	Section 7.3.2	ISE 19/20	1.6	0.4	0.3
		ISE 21/22 D	0.5	1.2	1.7
Commit messages using annotations during committing or afterward in Jira comment	Section 7.3.3	ISE 21/22 D	-1.8		0
Code comments using annotations	Section 7.3.4	ISE 21/22 D	-1.7		0.4
Chat messages and exporting it to Jira	Section 7.3.5	ISE 19/20	-1.1	-0.1	-0.1
Wiki pages (not implemented)	Section 7.3.5	ISE 19/20			0.6
Pull requests (not implemented)	Section 7.3.5	ISE 19/20			0.3
Other Documentation Features					
Decision grouping	Section 7.9	ISE 21/22 D	0.5	1.3	1
		ISE 21/22 C			1.5
Automatic text classification to identify decision knowledge in Jira ticket text, RS5	Section 7.6.8	ISE 19/20	-0.7	0	0
		ISE 21/22 D	-1.3	0.5	-0.7
		ISE 21/22 C			1
Link recommendation and duplicate detection, RS4	Section 7.6.7	ISE 21/22 D	-0.2	1.5	0
		ISE 21/22 C			1
Recommendation of solution options from knowledge sources (decision guidance), RS3	Section 7.6.6	ISE 21/22 D	-1.5	1	0.2
		ISE 21/22 C			2
Changing elements and links through interaction with knowledge graph views, F3	Section 7.5.3	ISE 19/20	0.9	0.6	0.9
		ISE 21/22 D	0.5	0.8	1
		ISE 21/22 C			1.5
Linking arguments to criteria in criteria matrix, F3	Section 7.4.4, Section 7.5.3	ISE 21/22 D	1.8	0.7	0.6
		ISE 21/22 C			2

opinion is that the text annotation in Jira ticket descriptions and comments is already very easy, and the participants thought automating the annotation was unnecessary.

The participants rarely used the *link recommendation and duplicate detection* feature (rating -0.2). They stated that they find the feature useful if the recommendations are true positives and there are no false negatives. Besides, they stated that the feature would be particularly useful in long-living projects with changing developers, which was not the case in the ISE 21/22. One practitioner disagreed about the usefulness because too many links would lead to information overload when exploiting the knowledge graph.

The participants very rarely used the *decision guidance* feature (rating -1.5). Like the link recommendation and duplicate detection feature, they stated that the decision guidance is only useful if the recommendations are high-quality. For this purpose, one needs similar projects or another kind of fruitful knowledge source. One participant emphasized that it is still very important to think about your own decisions instead of just accepting decisions from others.

In general, the participants had a positive attitude toward the *interaction possibilities of the knowledge graph views* (V1 – V6). They like the context menu and drag & drop functionality. They state that adding new decision knowledge elements via the context menu is easier than

writing it in text because ConDec automatically adds the annotations. However, they stated that the view interactions need practice and are not self-explanatory.

The ISE 21/22 participants had a positive attitude toward the *linking of arguments to criteria*. For instance, criteria are quality requirements, e. g., efficiency and security, or constraints and context factors, e. g., implementation effort. They justify the positive rating as follows: It helps during decision making to weigh the arguments, e. g., to see that usability is more important in the project than security. It helps structure pros and cons, see all decisions affecting a specific criterion, and see whether quality requirements are fulfilled.

Intended Future Usage of Documentation Features

We asked the ISE 21/22 developers which of the documentation features they would use in the future. The answer indicates which features the developers prefer. All developers intend to use the documentation of decision knowledge in Jira as tickets and ticket text. One developer answered that they intended to document decision knowledge in code comments in the future. Further, they intend to use decision grouping, linking criteria to arguments, link recommendation and duplicate detection, and decision guidance features.

Discussion: Do developers accept the ConDec support for knowledge documentation?

Overall, the study participants accept the knowledge documentation features since they mostly agreed on the ease of use and usefulness. The acceptance differs between the features as discussed above. The results regarding the preferred documentation locations illustrate that study participants have different preferences about where to document decision knowledge. It is a strength of ConDec to integrate the documentation in tools and documentation locations that the practitioners already use instead of providing a separate tool. With ConDec, the practitioners can document decision knowledge in their preferred documentation locations, which makes the documentation low-intrusive and helps to overcome the capture problem. In the future, the documentation should be supported in even more tools and documentation locations used in the industry, which would likely increase the acceptance by the practitioners. The practitioners accept the recommendation systems concerning the documentation, i. e., the automatic text classification (RS5), link recommendation and duplicate detection (RS4), and the decision guidance (RS3), if the accuracy of their recommendations is high. In the future, these recommendation systems should be further improved.

11.2.3. Acceptance of Knowledge Exploitation Features

This section presents the results for the question *Do developers accept the ConDec support for knowledge exploitation?* (RQ3). The subsections describe 1) the acceptance from the point of view of practitioners from the industry (workshop participants and IT consultants), 2) the acceptance of the knowledge graph views, 3) the acceptance of other exploitation features, and 4) the exploitation features the study participants prefer to use in the future. In the last subsection, we discuss the results.

Acceptance of Knowledge Exploitation Support by Practitioners from Industry

Table 11.3 on page 201 shows the results for the statements *It is easy to exploit decision knowledge with ConDec*, *ConDec is useful for decision knowledge exploitation* and *I would use ConDec to exploit decision knowledge in the future*. On average, the practitioners from the industry were neutral or agreed on the ease of use (rating 0.5 – 1) and agreed on the usefulness (rating 0.5 – 1). They agreed they would use ConDec for decision knowledge exploitation in the future (rating

0.2 – 1). The practitioners from the industry gave the following feedback: Some like the knowledge view because they are "visual persons". One practitioner likes that the decision knowledge is accessible from tools such as Jira because you "do not have to read Javadoc". However, one practitioner states: "I am not sure if special visualizations like trees or graphs are actually superior to a text file and a search function."

Acceptance of Knowledge Graph Views

To understand the acceptance of the knowledge graph views, we combined the collection of qualitative feedback with view access logging. Figure 11.1 shows the view usage over time for the ISE 20/21 (starting from the second sprint review) and the ISE 21/22 projects. The developers accessed the views with varying frequencies and accessed different views during the projects. In the ISE 20/21 project, the view usage seemed to be higher before the sprint reviews (Figure 11.1, first plot). A reason might be that the developers had to tidy up the documentation to present it to the customer in the sprint review. For that purpose, they accessed the ConDec views. In the ISE 21/22 project, the proportion of the accessed views changed (Figure 11.1, fourth plot). In the first to third sprint, the developers mainly accessed the knowledge tree views (V2). In the fourth and fifth sprints, the developers mainly accessed the criteria matrix ($V4_{cri}$).

Table 11.6.: Total number of accesses per ConDec view measured by logging REST API calls in the validation projects ISE 20/21 and ISE 21/22 as well as the perceived usage frequencies stated by the ISE 21/22 developers. The last column lists the number of ISE 21/22 developers who selected a view as their preferred view.

ConDec View	API Name	ISE 20/21	ISE 21/22		
		#Accesses	#Accesses	Perceived Usage	Is View Preferred?
Node-link diagram, V1	view/node-link-diagram	317 (3.8 %)	483 (6.6 %)	-0.7	0 (0 %)
Tree, $V2_{ind}$	view/indented-outline	4207 (50.9 %)	1645 (22.6 %)	0.7	5 (83.3 %)
Tree, $V2_{nld}$	view/treant	2193 (26.5 %)	2390 (32.8 %)	0.7	5 (83.3 %)
Adjacency matrix, $V4_{adj}$	view/adjacency-matrix	383 (4.6 %)	485 (6.6 %)	-0.8	0 (0 %)
Criteria matrix, $V4_{cri}$	view/criteria-matrix	0 (0 %)	2034 (27.9 %)	1.8	6 (100 %)
Chronology, V5	view/chronology	28 (0.3 %)	22 (0.3 %)	-1.8	0 (0 %)
Metrics, V6	dashboard	1088 (13.2 %)	235 (3.2 %)	0.8	5 (83.3 %)

Table 11.6 shows the total amount of accesses per view during the project duration of both ISE projects and the perceived usage frequencies stated by the ISE 21/22 developers. We also asked ISE 21/22 developers for their preferred views and to explain their selections.

The *knowledge tree views* (V2) were most often accessed. They account for 6400 (77.4 %) of the view accesses in ISE 20/21 and 4035 (55.3 %) in ISE 21/22. When asking the ISE 20/22 developers for their attitude towards the ConDec views, they stated that the indented outline was most useful because it provided the best overview. Regarding the node-link tree diagram, they criticized that they often had to scroll a lot, which they found not useable. On average, the ISE 21/22 developers said they sometimes used both knowledge tree views with an equal perceived usage frequency (rating 0.7). Five of the six developers in the ISE 21/22 project selected the knowledge tree views as preferred.

The developers accessed the *node-link diagram* (V1) and *adjacency matrix* ($V4_{adj}$) similarly often (between 3.8 % to 6.6 % of the view accesses). The ISE 21/22 developers said they rarely used these views with similar ratings (-0.7 and -0.8). They stated that both views are useful for seeing and managing links, i. e., their types and directions. Link types and directions are not shown in knowledge tree views. They emphasized that they liked the colorful presentation of link

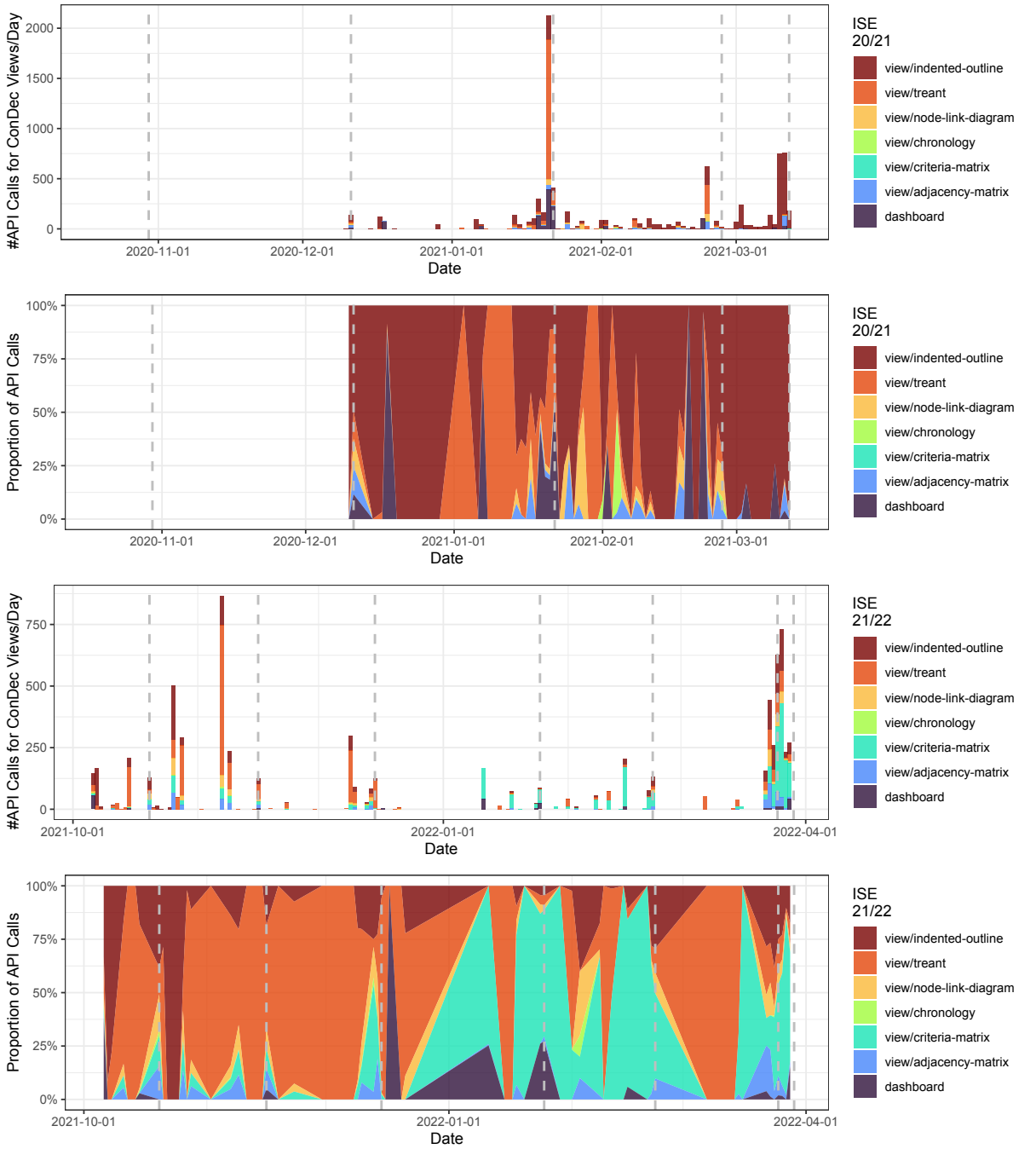

Figure 11.1.: View usage measured by the number of REST API calls per day in the validation projects ISE 20/21 and ISE 21/22. The first and third plots show the number of API calls, whereas the second and fourth plots show the proportion.

types in the adjacency matrix view. However, they found the node-link diagram and adjacency matrix unsuitable for getting a good overview of the documented knowledge.

The *metrics plots* (V6) are part of the knowledge dashboard, which plots various metrics at once. The ISE 21/22 developers said they sometimes used the dashboard with metrics plots (rating 0.8), equally often as the knowledge tree views. However, the developers accessed the knowledge dashboard less often (3.2 % to 13.2 % of the view accesses in both projects). Five of

the six developers in the ISE 21/22 project selected the knowledge dashboard as a preferred view. We discuss the developers' attitude toward the knowledge dashboard as part of RQ4.

The *chronology view* (V5) was hardly ever accessed (0.3 % of the view accesses). This was confirmed by the ISE 21/22 developers, who stated that they almost never used the chronology view (rating -1.8). When asking about their attitude, they did not use the chronology view because they had already included a list of decision knowledge elements (V3) in their meeting agenda and used it as a stand-up table. They found that the chronology view could replace the list in meeting agendas but that using both the list and the chronology view is unnecessary.

In the ISE 20/21, the developers did not use the *criteria matrix* ($V4_{cri}$). They stated that, in general, they find it useful to consider criteria such as quality requirements during decision making, but they did not find it necessary in their project. However, they linked work items to quality requirements and used other knowledge graph views to inspect the knowledge documentation in the context of the quality requirements. In contrast, in the ISE 21/22 project, the criteria matrix was often accessed and accounts for 27.9 % of the view accesses. When asking the ISE 21/22 developers, they also stated that they often used the criteria matrix (rating 1.8) and selected the criteria matrix as their preferred view. One developer stated that "the criteria matrix is most useful because it offers the best overview of pros and cons. You can see contradictions and make decisions better." The reason for the completely different usage of the criteria matrix in both projects might be twofold: First, we focused on this view when disseminating ConDec since it was not used in the prior ISE project. Second, the developers in both projects developed different habits and preferences.

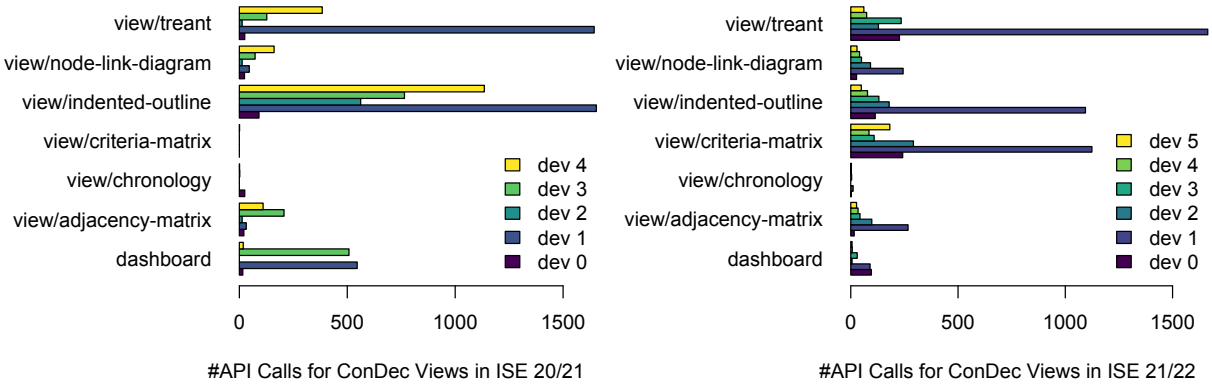

Figure 11.2.: View usage measured by the number of REST API calls per view and developer during the ISE 20/21 (starting from second sprint review) and ISE 21/22 projects.

Figure 11.2 shows that the view usage differed between the ConDec views and also between the developers dev 0 – dev 4 in ISE 20/21 and dev 0 – dev 5 in ISE 21/22.

We asked the developers to rate the overall usefulness of *presenting, i. e., visualizing knowledge in Jira* (Table 11.7). They agreed to strongly agreed on the usefulness, but—as discussed above—gave feedback that some views are more useful than others.

Acceptance of Other Exploitation Features

Table 11.7 shows the ratings regarding other exploitation features of ConDec. In the following, we discuss the acceptance of these features.

The participants rarely used the *stand-up table with decision knowledge in Confluence*. On average, they agreed on its ease of use, and they were neutral to agreed on its usefulness. They like that the stand-up table provides a good overview of the open issues and recently made decisions. However, they state that the information is redundant to what is shown in the rationale

Table 11.7.: Weighted means μ_w of the study participants' Likert ratings on their perceived usage frequency, ease of use, and usefulness of ConDec's exploitation features.

ConDec View or Feature	Description	Project	Usage Frequency	Ease of Use	Usefulness
Presenting/visualizing knowledge in Jira	Section 7.4	ISE 19/20			0.9
		ISE 21/22 D			1.7
Stand-up table with decision knowledge in Confluence	Section 7.10	ISE 21/22 D	-0.7	1.2	0.5
		ISE 21/22 C			0.5
Semi-automatic release notes creation including decision knowledge	Section 7.11	ISE 21/22 D	-0.7	1.2	1.2
		ISE 21/22 C			-1
Change impact analysis, RS2	Section 7.6.5	ISE 21/22 D	-1.5	0.7	0.3
		ISE 21/22 C			0.5
Navigation from code to knowledge graph view in Jira, F5	Section 7.5.5	ISE 21/22 D	-2	1	-0.2
Filtering the knowledge graph views, F1	Section 7.5.1	ISE 21/22 D	0	0.8	1.7
		ISE 21/22 C			1
Exploiting transitive links, F2	Section 7.5.2	ISE 21/22 D	0	0.8	1
		ISE 21/22 C			0.5
View for rationale in pull requests, V3	Section 7.4.3	ISE 19/20	-0.3	0.1	0.3
View for rationale from git in Jira, V3	Section 7.4.3	ISE 21/22 D	-2		

backlog and decision knowledge overview in Jira. Instead of the stand-up table in Confluence, they used the backlog in Jira because then they could directly make changes, which the ConDec Confluence plug-in currently not supports. The IT consultants were neutral to agreed about the usefulness of the stand-up table with decision knowledge in Confluence. They use boards in their daily work, sometimes analog, not digital.

The participants rarely used the *semi-automatic release notes creation including decision knowledge*, as each developer only applied it at the end of the sprint when they took the role of the Scrum master and rationale manager. The ISE 21/22 developers agreed that the semi-automatic release notes creation is easy to use and useful. They liked presenting the release notes with decision knowledge to the customer and explaining what they did in the sprint and why they did it. In the ISE projects, we requested the students to present their knowledge documentation during the sprint reviews. For this purpose, the release notes with decision knowledge are useful. The IT consultants disagreed on the usefulness of the release notes with decision knowledge because the customer might not be interested in decisions. The developers refined this statement as the customer might only be interested in decisions for the problem space, i. e., for requirements, and not for the solution space. One IT consultant stated that it depends on the domain. For example, a software development project to develop a source code library might benefit more from release notes with decision knowledge than an app development project.

The participants very rarely used the *change impact analysis*. They were neutral to agreed on the ease of use and its usefulness. They liked to see the affected knowledge elements of a new decision or a decision to be changed "since you cannot have all dependencies in mind when making changes". However, two developers stated that they still had the dependencies in mind as implicit knowledge in their project and, thus, did not need the feature. The feature will become more useful for them in larger projects with changing developers.

After we disseminated ConDec, the participants never used the *navigation feature from code to the knowledge graph views in Jira* on their own. We developed a ConDec Eclipse and a VSCode plug-in that offer this feature. However, the developers in the ISE 21/22 mainly used the IntelliJ

integrated development environment, for which currently no ConDec plug-in exists. While they agreed that the feature is easy to use, they slightly disagreed on its usefulness. They stated that knowing about the Jira tickets that changed a specific code file and the related decision knowledge was not necessary. Again, this might be more useful in larger projects with changing developers. One developer stated to have used git functionalities such as git blame or git annotate to find out about the history of a code file. ConDec also uses these git functionalities to establish the links between code files and Jira tickets in the knowledge graph.

The participants agreed on the ease of use of *filtering the knowledge graph views* and *exploiting transitive links*. They are must-be features to cope with a high amount of knowledge elements. They state that they needed some time to get familiar with the faceted ConDec filters, which are separate from the Jira filter functionality. Currently, the filters are reset when reloading a view. It would ease the usage if the ConDec filters could be cached.

The ISE 19/20 developers rarely used the *view for decision knowledge in pull requests*. The ISE 21/22 developers did not use pull requests but directly merged their feature branches. Thus, they did not use this feature. On average, the ISE 19/20 developers were neutral on its ease of use and slightly agreed on its usefulness. They like to see the decision knowledge related to the pull request because it helps them understand the code changes when reviewing the pull request. Besides, presenting decision knowledge enables the review of the decision knowledge itself. The decision knowledge is currently shown in a dialog, which is triggered manually, and should better be directly shown within the pull request view.

The ISE 21/22 developers did not use the *view for decision knowledge from git in Jira* since they mainly documented decision knowledge in Jira and not in git.

Intended Future Usage of Exploitation Features

We asked the ISE 21/22 developers which of the exploitation features they would use in the future. The answer indicates which features the developers prefer. All developers intend to use their preferred knowledge views and the filtering possibilities in the future. Three developers state that they intend to use the exploitation of transitive links. Although they rarely used the features in the projects, three developers intend to use the stand-up table with decision knowledge, and three intend to use the semi-automatic release notes creation.

Discussion: Do developers accept the ConDec support for knowledge exploitation?

Overall, the study participants accepted the knowledge exploitation features since they mostly agreed on their ease of use and usefulness. The acceptance differs between the features as discussed above. While the developers who used ConDec over several months agreed on the usefulness of the knowledge visualization based on trace links, one practitioner from the industry discussed that simple text files and a search function might "also do the job". We argue that the knowledge visualization based on trace links is superior to text files for projects with many requirements and code files. For example, one can easily see all decisions related to a requirement by filtering and exploiting transitive links and can perform change impact analysis. ConDec enables the export of the knowledge graph data structure in a text file if needed. The trace links in ConDec are lightweight as they are established either a) automatically by ConDec, e. g., between an issue and a requirement if the issue is documented in a comment of the requirement, or b) by the developers during their usual practices as done between code and tickets through providing ticket identifiers in commit messages. In the future, knowledge exploitation should be supported in all tools used in industry, which would likely increase their acceptance.

11.2.4. Acceptance of Quality Assurance Features

This section presents the results for the question *Do developers accept the ConDec support for quality assurance?* (RQ4). The subsections describe 1) the acceptance from the point of view of practitioners from the industry, 2) the acceptance of specific quality assurance features, and 3) the quality assurance features the study participants prefer to use in the future. In the last subsection, we discuss the results.

Acceptance of Quality Assurance Support by Practitioners from Industry

Table 11.3 on page 201 shows the results for the statements *It is easy to create and maintain high documentation quality with ConDec*, *ConDec is useful to create and maintain high documentation quality* and *I would use ConDec to create and maintain high documentation quality in the future*. On average, the practitioners from the industry were neutral on the ease of use (rating 0) and were neutral or agreed on the usefulness (rating 0 – 1). They were neutral about whether they would use ConDec for creating and maintaining high documentation quality in the future (rating 0). One practitioner states: "The quality comes from the user entering high-quality content. I think you can still very much misuse the tool and enter inconsistent and incomplete information. The tool can only syntactically check if certain forms are filled, but not, for example, if an important decision option is missing."

Usage Frequency, Ease of Use, and Usefulness of Quality Assurance Features

Table 11.8 shows the ratings regarding features for creating and maintaining high documentation quality of ConDec. In the following, we discuss the acceptance of these features.

Table 11.8.: Weighted means μ_w of the study participants' Likert ratings on their perceived usage frequency, ease of use, and usefulness of ConDec's quality assurance features.

ConDec View or Feature	Description	Project	Usage Frequency	Ease of Use	Usefulness
Defining and checking of a definition of done for the knowledge documentation, RS1	Section 7.6.4	ISE 21/22 D			0.2
		ISE 21/22 C			1.5
Knowledge dashboard with metrics, V6	Section 7.8	ISE 21/22 D	0.8	1.3	1.5
		ISE 21/22 C			1
Rationale backlog showing knowledge elements that violate the definition of done	Section 6.2.3, Section 7.7	ISE 21/22 D	1	0.8	1.7
		ISE 21/22 C			1
Result presentation of definition of done checking in the quality check view	Section 7.6.4	ISE 21/22 D	0.7	1.5	1
Ambient feedback nudging: coloring menu items and knowledge elements, N2	Section 7.6.2	ISE 21/22 D	0	1.5	1.5
Just-in-time prompt nudging, N3	Section 7.6.3	ISE 21/22 D	0	0.8	0.8
Marking links as wrong or useless	Section 7.5.3	ISE 21/22 D	-0.7	0.3	0.2
		ISE 21/22 C			0.5
Merge check of decision knowledge in pull requests	Section 7.6.4	ISE 19/20	-0.3	0.6	-0.3

The participants slightly agree to strongly agree that *defining and checking of a definition of done for the knowledge documentation* is useful. The participants agree that it is useful to make the criteria explicit that need to be fulfilled before finishing a requirement or task. Otherwise, the documentation might be neglected, and there will be "uncontrolled development". However, they would like to have the possibility to disable the definition of done checking for particular

Jira tickets, such as those related to bug fixes, which quickly need to be finished and do not involve decision making, in their opinion. One developer who strongly disagrees states that a decision should be postponed to the last possible moment in agile software development. In contrast, the definition of done would enforce quick decision making to fulfill it.

The participants sometimes used the *knowledge dashboard with metrics*. They agreed on its ease of use and usefulness. They found it useful in the role of the rationale manager to supervise the rationale documentation. One developer fears that calculating and optimizing too many *key performance indicators*, i. e., metrics in the dashboard would destroy the agile nature of the software development process. For example, they stated the metric that shows how many knowledge elements fulfill and violate the definition of done is generally useful. Still, they could not satisfy it since many code files did not reach the minimum decision coverage.

The participants sometimes used the *rationale backlog showing knowledge elements that violate the definition of done*. They agreed on its ease of use. Some developers like that the rationale backlog is a new view in Jira introduced by the ConDec Jira plug-in for separating concerns. Others would find the rationale backlog easier to use if it was directly integrated into the view of Jira's product and sprint backlogs. The participants agreed to strongly agreed on the usefulness of the rationale backlog. They like that they can easily find knowledge elements that need improvement. They used the rationale backlog during meetings to discuss open issues as an alternative to the stand-up table in Confluence.

The participants sometimes used the *definition of done checking result presentation in the quality check view*. The developers agreed to strongly agreed on its ease of use and agreed on its usefulness. While the ambient feedback nudging mechanisms indicate quality problems, the quality check view explains which criteria of the definition of done are violated.

The participants agreed to strongly agreed on the ease of use and usefulness of the *ambient feedback nudging mechanism*. They state that the coloring helps to understand the quality check results without navigating to the quality check view. One participant with a red-green deficiency points out that other means should indicate quality problems next to colors, such as structures or font style.

The participants agreed on the ease of use and usefulness of the *just-in-time prompt nudging mechanism*. It is easy to use because the prompt is automatically shown when changing a Jira ticket state, e. g., when starting or finishing a requirement. It is useful to see hints for improvement of the decision knowledge documentation. However, the participants admitted that—in some cases—they ignored and closed the prompt because of time pressure and because the prompt only gives hints and does not enforce the improvements.

The participants rarely *marked links in the knowledge graph as wrong or useless*. They slightly agreed that the feature is easy to use and useful. One developer calls it an advanced feature that they would have used more often if the project had gone longer. The feature helps to exclude decisions that are transitively, i. e., indirectly, linked to a requirement but are semantically unrelated. One developer thinks that the concept of marking a link as wrong or useless is hard to grasp, stating "either you have a link, or you have no link".

The ISE 19/20 developers rarely used the *merge check of decision knowledge in pull requests*. While they agree that the merge check is easy to use, they slightly disagree that it is useful. Some pull requests related to bug fixing could not fulfill the required decision coverage. They created a Jira ticket for the bug-fixing task but did not make decisions worth documenting. Note that in the ISE 19/20 project, ConDec did not yet have the functionality to include indirect links in the calculation of the decision coverage. The decisions had to be directly linked to the bug report. In the current version of ConDec, the bug report fulfills the decision coverage if it is linked to a requirement with documented decisions. The ISE 19/20 developers discussed that enforcing the decision coverage can lead to the documentation of trivial decisions. However, one developer states that they would not document decisions if such checks and enforcement did not exist.

Table 11.9.: Study participants' assessment of the rationale documentation quality in the project and weighted means μ_w.

ConDec Statement	Project	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	μ_w
The rationale documented during the ISE project has high quality.	ISE 21/22 C	0	0	0	2	0	1
Keeping the documented rationale complete is easy.	ISE 21/22 D	0	1	1	4	0	0.5
Keeping the documented rationale consistent with other rationale and artifacts (e. g., requirements, code) is easy.	ISE 21/22 D	0	3	1	2	0	-0.2

We asked the IT consultants from the industry to rate whether the rationale documentation created during the ISE 21/22 project is of high quality, which they agreed on (Table 11.9). One IT consultant stated, "it reflects what they did, from the point of view of the project supervisor". However, they discussed that the developers sometimes documented the decisions retrospectively, which they should ideally do during decision making.

We asked the developers how easy it was to maintain the high quality of the documented decision knowledge. We focused on completeness and consistency as two important quality attributes of software documentation (Section 1.2). Table 11.9 shows the Likert selections. They found keeping the documentation complete (rating 0.5) easier than consistent (rating -0.2). Regarding completeness, one developer stated, "it is easier with ConDec than without because you get slapped on the fingers". Another developer stated, "with ConDec, it is easier to see where you have forgotten something". Regarding consistency, one developer stated, "Keeping documentation up to date is never easy, I think". A second developer stated "we had decision problems that became outdated, but it is difficult for a system to point them out. You have to deal with it manually". A third developer stated, "with ConDec, it is easier because you stumble over the documentation again and again; it is a strength of ConDec that it is built into Jira".

Intended Future Usage of Quality Assurance Features

We asked the ISE 21/22 developers which of the features for creating and maintaining high documentation quality they would use in the future. The answer indicates which features the developers prefer. Four developers intend to use the definition of done for the knowledge documentation and the related checking features, such as the coloring of user-interface elements (nudging mechanisms), the quality check view, and the rationale backlog. Three developers intend to use the knowledge dashboard. Two developers intend to use the marking of wrong or useless links if necessary during knowledge exploitation.

Discussion: Do developers accept the ConDec support for quality assurance?

Overall, the study participants accepted the quality assurance features since they mostly agreed on their ease of use and usefulness. The acceptance differs between the features as discussed above. Currently, ConDec can either indicate violations of the definition of done through warnings or enforce the improvement of violations by treating them as errors. On the one side, the indication through warnings might lead to low documentation quality since the developers delay the quality improvement. On the other side, enforcement is intrusive in the development process. It might lead to pseudo decisions, i. e., decisions being not useful for future development, solely being documented to optimize a key performance indicator. In the future, the rationale manager should get more possibilities to configure when ConDec should a) ignore violations of the definition of

done, as discussed for bug reports for that a development team might decide that no decisions need to be linked, b) indicate violations of the definition of done through warnings, or c) enforce their improvement.

11.3. Threats to Validity

This section discusses four validity criteria of primary empirical studies as defined by Easterbrook et al. (2008) and Runeson et al. (2012):

Construct validity focuses on whether the theoretical constructs are measured and interpreted correctly. The study participants might have interpreted the questions or statements in the questionnaires differently from what we intended. To mitigate misinterpretations, we allowed them to ask questions at any time. A threat is that we interviewed practitioners from the industry who did not have the chance to use ConDec over a longer time. They might not be able to assess the acceptance reliably. Nevertheless, we report their attitude because we think that their experience in industry projects and with other tools outweighs the lack of usage. Similarly, the student developers rarely used some views and features. However, their feedback is interesting for collecting improvement ideas.

Internal validity concerns whether the results we draw really follow from the data, e. g., whether confounding factors influence the results. Positive ratings could be because students received credits and grades when working on the projects. We mitigated this threat by assuring the students that their interview answers do not affect the grading and that we are interested in honest feedback. Besides, the students should always explain their rating to reflect on it. The students knew that we monitor the ConDec usage for evaluation purposes, but we assured them that this does not influence the grading. The number of API calls is one indicator for view usage but needs to be assessed with caution: The knowledge graph views V1 – V7 are the building blocks supporting a continuous rationale visualization. The knowledge tree view $V2_{ind}$ lists the knowledge elements with violating definition of done in the rationale backlog. Thus, whenever the developers accessed the rationale backlog, the REST API of $V2_{ind}$ was called. There is no default view when accessing the knowledge graph from Jira issues such as requirements or work items. However, the ConDec Jira plug-in remembers the view that a developer selected and makes it the default until the browser session is finished. We mitigated the threat of drawing wrong conclusions from the measured view accesses by asking the developers for their perceived usage frequencies and attitudes regarding the views.

External validity addresses the generalizability of the study results. While we mainly reported the feedback of the last ISE project 21/22, we studied the acceptance of ConDec during other validation projects starting from the iPraktikum 18/19. We constantly improved ConDec based on the feedback of the previous studies. We expect that industry practitioners would generally accept the use of ConRat and the ConDec plug-ins. However, the analysis of the view accesses of the criteria matrix showed that the acceptance of specific features could differ a lot between development teams.

Reliability validity concerns the study's dependency on specific researchers. In the iPraktikum projects, two researchers from the Technical University of Munich who managed the projects collected feedback regarding ConDec. Afterward, a threat to the reliability validity might be that the collection and analysis of feedback through surveys, interviews, and informal discussions were only done by the author of this thesis. We asked the study participants to review their feedback and Likert ratings to mitigate the threat.

11.4. Conclusion

This chapter presented an empirical study on the acceptance of the ConDec plug-ins from the point of view of software practitioners from industry and University students working in project courses in an industrial setting. The study participants generally accepted the ConDec views and features and the decision-making benefit. However, the acceptance differs between the study participants and the ConDec views and features. Many factors influence the acceptance, for example, whether the practitioners usually use the tools extended by the current ConDec plug-ins and the dissemination technique. The study reported feedback on what the practitioners liked about the features and improvement ideas. The study participants liked the following ConDec views and features the most: Documentation of decision knowledge in Jira (as tickets or in ticket text), decision grouping, interaction and filtering possibilities in the knowledge views, the knowledge tree views and criteria matrix, the stand-up table with decision knowledge for meetings, the quality checking functionality including nudges, the rationale backlog, and the knowledge dashboard. While one practitioner from the industry particularly questioned ConDec's benefit for decision making, the application of ConDec in the case study projects indicated that it helps the discussion and reflection on decisions. ConDec particularly supports decision making with the criteria matrix, the stand-up table, the rationale backlog, and the decision guidance recommendation system (Section 7.1.3).

The study is interesting for practitioners since it showed that the ConDec plug-ins could support ConRat in industrial projects as long as the underlying tools fit, e. g., Jira is used as the issue tracking system. The study makes two contributions to researchers: First, it provided ideas about future work on rationale management. Based on the study participants' feedback, the ConDec views and features should be improved. In particular, further ConDec plug-ins must be developed for other tools used in the industry, e. g., GitLab, Azure DevOps, and IntelliJ. Second, the methods presented in the study are interesting for researchers as a template to validate user acceptance in their projects. The study contributed a questionnaire in Appendix F and a method to analyze the Likert results with weighted means. Researchers could adapt the questionnaire for other features. The study contributed methods and results of REST APIs usage logging as an indicator for acceptance. Since REST APIs are standard interfaces in many applications, researchers could apply the methods in other domains.

Chapter 12

Dissemination of ConRat and ConDec Plug-Ins

> *"Although the benefits are well-known and undisputed, sharing architectural knowledge is not something architects automatically do."*
>
> —Hoorn et al., 2011

This chapter contributes to the instrument design goal of the thesis: *Disseminate ConRat and the ConDec plug-ins to developers and show the acceptance of the dissemination.* To overcome the rationale management problems in practice, developers need to know how to capture rationale and be aware of the benefits of rationale management. This chapter presents a syllabus on rationale management to disseminate ConRat and the ConDec plug-ins with *hands-on* exercises. The syllabus includes teaching students a rationale model, introducing them to workflows, ConDec views and features for rationale management, and motivating them to apply rationale management in the future. We gave lectures based on the syllabus in the iPraktikum and ISE projects (Section 8.1) as well as in a rationale management workshop in the research division of an industrial company in September 2021.

Section 12.1 describes the syllabus, including 15 exercises. Section 12.2 describes the students' feedback and the exercise results and discusses lessons learned from the first instantiation of the lecture in the iPraktikum 18/19. Section 12.3 discusses related work on disseminating rationale management in student courses. Section 12.4 concludes this chapter.

An initial version of this chapter was published in Kleebaum et al. (2019a). This thesis added new exercises based on the experiences and for disseminating newer ConDec views and features.

12.1. Syllabus on Rationale Management

This section describes the syllabus for the dissemination of rationale management activities supported through ConDec. As a prerequisite, the students are expected to have the following knowledge: They must be familiar with the concepts and usage of the issue tracking system Jira, the wiki system Confluence (or similar tools), the version control system git, and an integrated development environment. They should know how to manage requirements and work items as tickets and how to create and merge feature branches linked to the tickets. The instantiation of the syllabus as a lecture lasts 120 minutes and can be tailored to teach only the basic rationale management activities (Table 12.1). Students are grouped into teams and require a web-connected device with access to the internet.[1] During the lecture, four systems are required: Jira,

[1] An alternative would be to run the servers locally and to give students access to the intranet (not possible for Slack).

Table 12.1.: Schedule for the rationale management lecture. Bold parts are mandatory to teach basic rationale management activities. The other exercises are more advanced.

Part of Lecture	**Introduction**	**Exercise 1:** View Issues for Requirement	**Exercise 2:** Discuss Decision Types	**Exercise 3:** Link Issue to Requirement	**Exercise 4:** Filter Knowledge	**Exercise 5:** Capture Rationale in Jira	Exercise 6: Link Criteria	Exercise 7: View Knowledge	Exercise 8: Capture Rationale in Git	Exercise 9: Capture Rationale in Chat	Exercise 10: Analyze Change Impacts	Exercise 11: Review Rationale Quality (1)	Exercise 12: Conduct Meeting (1)	**Exercise 13:** Conduct Meeting (2)	Exercise 14: Review Rationale Quality (2)	Exercise 15: Create Release Notes	Ending/Buffer	$\sum$
Duration [min]	**20**	**4**	**5**	**5**	**5**	**10**	6	7	10	8	5	5	5	**5**	5	5	10	120

Confluence, git, and the instant messaging service Slack[2]. Further, the ConDec Jira, Confluence, and Slack plug-ins[3] must be installed in the systems. Rationale elements that the students can explore must be added to the respective Jira projects (as exemplified in Figure 12.1 and Figure 12.2). A git repository and a Confluence space must be created for every team. Slack is used as a communication tool between instructors and students. Polls can be created with the Polly Slack app[4] through which students participate during the lecture.

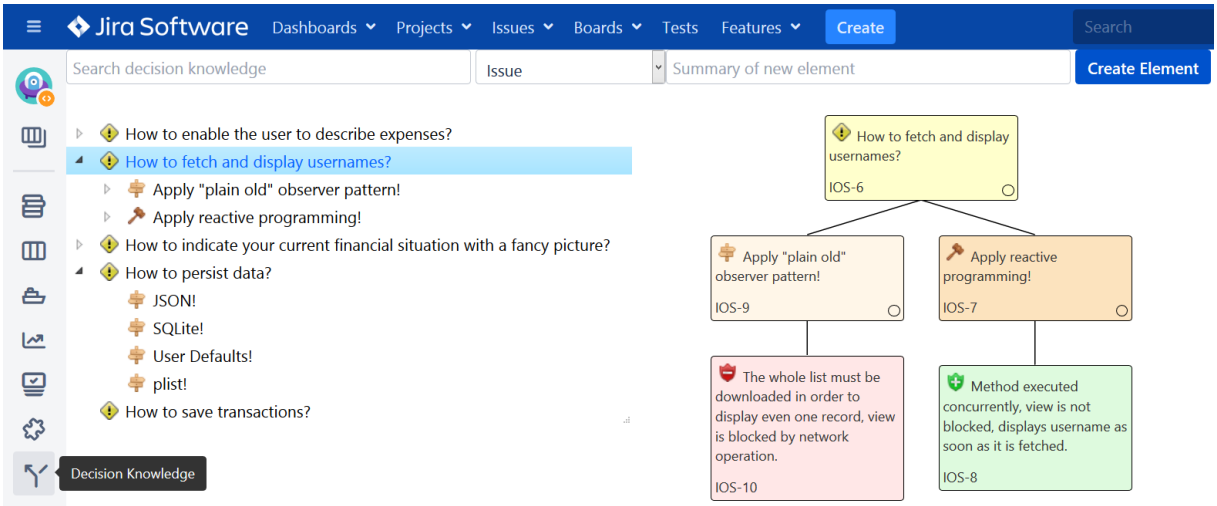

Figure 12.1.: The decision knowledge view for performing rationale management in Jira.

The introduction of the lecture covers background information about rationale management, such as its definition, expected benefits, and rationale models (Section 2.2). It introduces the students to the ConRat knowledge model (Section 6.1). ConRat demands to use specific phrases when talking about and capturing rationale elements (Table 12.2). The students should phrase issues as questions ending with a question mark and end decisions with an exclamation mark. For dissemination, the knowledge model distinguishes three types of associations. The *relates to* association is the default type. Only for arguments, different types are used: A pro-argument *supports* a solution option, whereas a con-argument *attacks* an option.

[2] `https://slack.com`
[3] `https://github.com/cures-hub`, see also Appendix A
[4] `https://polly.ai`

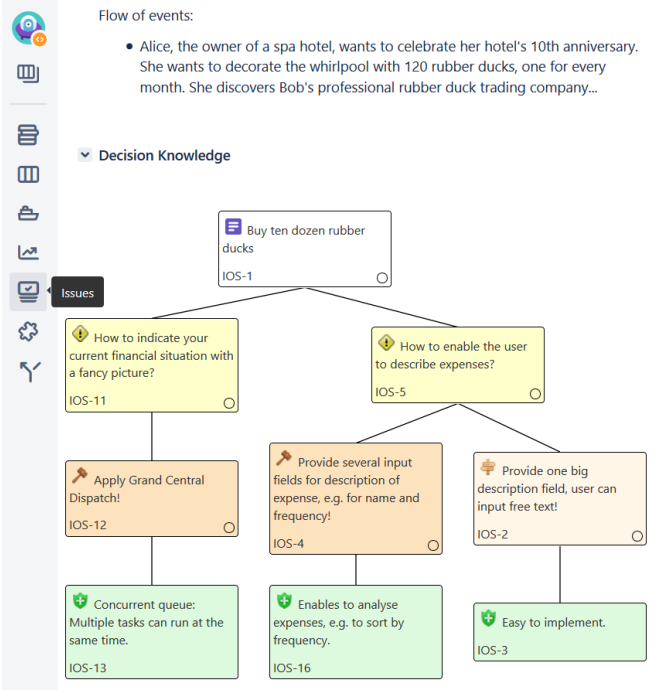

Figure 12.2.: The Jira ticket view includes the interactive rationale tree. Note that the tooltip *Issues* refers to *Jira issues* as a synonym for *Jira tickets*. A *Jira issue* is an abstract container in contrast to the rationale element issue, i. e., decision problem.

The remainder of the section describes the exercises grouped by ConDec features. It starts from must-be features for rationale documentation and exploitation to more advanced features.

Capturing, Visualizing, and Filtering Rationale in Jira

The instructor introduces the students to the Jira ConDec plug-in including the Jira ticket types for rationale elements. Two views for rationale management are shown to the students: the decision knowledge view (Figure 12.1) and the Jira ticket view (Figure 12.2). The decision knowledge view is a separate view that holds all knowledge elements and their links of the given project. In this view, a user can select rationale elements and see the views on the knowledge graph, such as the knowledge tree views (V2). The rationale attached to Jira tickets can be explored in the Jira ticket view. For example, Figure 12.2 shows the knowledge tree for a scenario. Then, the instructor demonstrates how to create and link the rationale elements, as shown on the right side of Figure 12.1. Afterward, the students gather in teams and perform the first exercise.

Table 12.2.: Rationale types, their representing icon, and indicating phrases for informal capture adapted from Doyle and Straus (1993).

Icon	Name	Indicating Phrases
	Issue	I have a question . . . ; How should . . . ; . . . , any suggestions?; We need to solve how . . .
	Alternative	I { suggest | propose } . . . ; One { option | proposal } is . . . ; What { about | do you think } . . .
	Pro	The { advantages | pros } are . . . ; I { like | prefer } it because . . . ; I agree with user . . .
	Con	The { disadvantages | cons } are . . . ; I don't like it because . . . ; I disagree with user . . .
	Decision	Let's do . . . ; We decided . . . ; The best option is . . .

Exercise 1 *Gather your team, open your Jira project, and find the scenario already documented in the project. Answer the question: How many issues are linked to the scenario?*

The students can answer the question via a poll. In the example, the correct answer is that two issues are linked to the scenario (Figure 12.2). The next exercise is to discuss the content of the solution proposals, i. e., the alternatives and decisions.

Exercise 2 *Answer the question: Which of the proposed alternatives and decisions focus on requirements, and which are on implementation? Add the alternatives and decisions to the respective decision groups.*

In Figure 12.2, the decision and alternative on the left are requirements-related, whereas the decision on the right is implementation-specific. With this exercise, the students learn that rationale can be captured for all software engineering workflows, including requirements elicitation, implementation, testing, and when processing user feedback. They get to know ConDec's decision grouping feature. Now, the students will learn to document rationale elements.

Exercise 3 *Link the existing issue "How to save transactions?" to the scenario.*

This issue is already part of the Jira project (Figure 12.1) but is not linked to the scenario yet. The students can link the issue via a context menu on the scenario node (root node in Figure 12.2). They can search for the issue themselves or use the link recommendation system. The students should now realize that graphs of rationale can become large and complex. Therefore, filtering is essential. The next exercise addresses filtering and exploiting transitive links.

Exercise 4 *Filter the element types so that only the scenario and decisions are shown in the knowledge tree.*

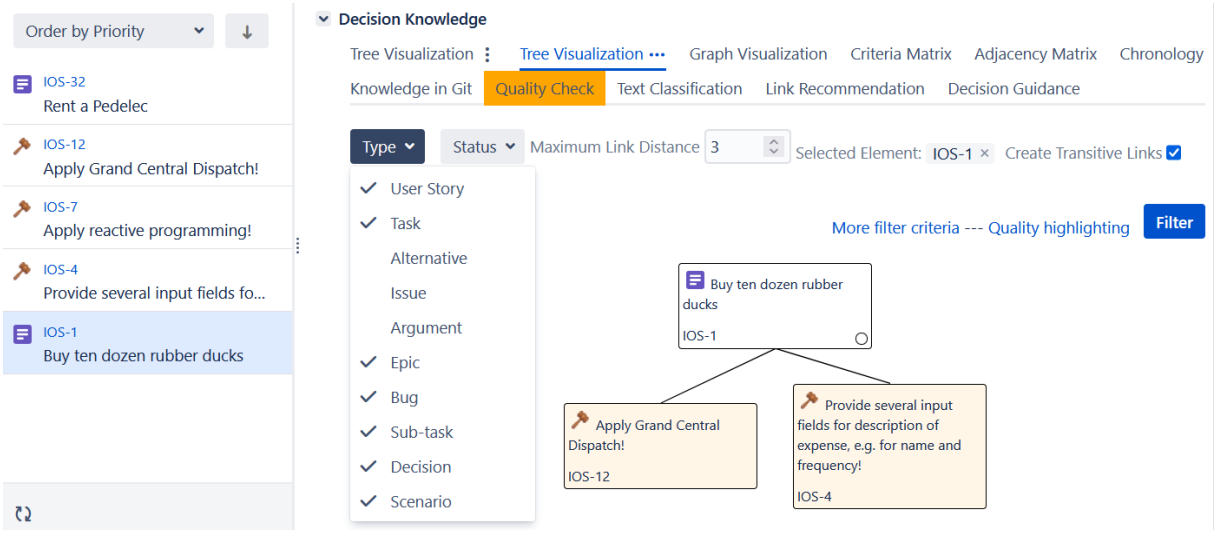

Figure 12.3.: Filtered knowledge tree (node-link tree diagram, $V2_{nld}$) in that issues are filtered out and replaced by transitive links to the decisions.

Figure 12.3 shows the solution of this exercise. Rationale can be captured in many places. Jira tickets are just one documentation location to store rationale. To demonstrate other documentation locations, the instructor presents capturing rationale in ticket comments (Figure 12.4).

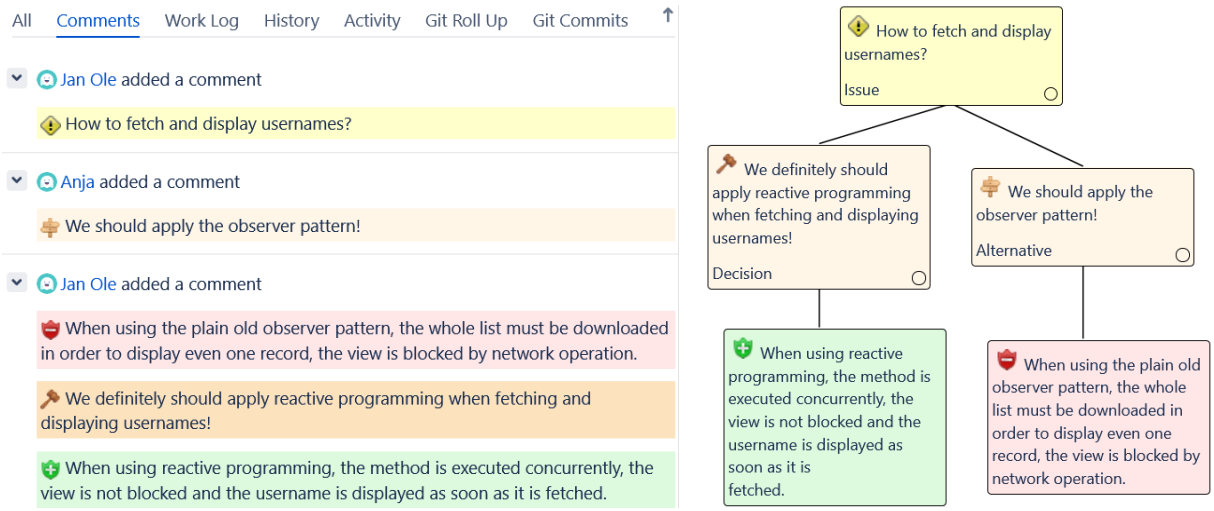

Figure 12.4.: **Left:** Explicit rationale in the comments of the scenario. **Right:** Knowledge tree (node-link tree diagram, $V2_{nld}$) automatically created from rationale in comments.

Now, students discuss rationale in their team:

Exercise 5 *Create a new issue "How to persist data?" as a Jira ticket. Add the following alternatives as comments: "JSON!", "SQLite!". Discuss and capture the pros and cons of each persistence alternative in your team. Add more alternatives and make a decision.*

To solve this exercise, the students can add rationale elements collaboratively from different devices. They can also use the decision guidance feature to see recommended solution options. In the following exercise, the students structure their arguments using criteria to check whether they evaluated the solution options systematically.

Exercise 6 *Navigate to the criteria matrix for the issue "How to persist data?" Link your pros and cons for the solution options to the criteria "Implementation effort", "Maintainability", and "Performance". Think about other criteria, add them, and link them to your arguments. Add new arguments or links between solution options and existing arguments to fill empty cells of the criteria matrix.*

Figure 12.5 shows a criteria matrix ($V4_{cri}$) with alternatives but no decision. Until now, the students got to know the knowledge tree view (node-link tree diagram, $V2_{nld}$) and the criteria matrix ($V4_{cri}$). In the next exercise, they explore other knowledge views.

Figure 12.5.: Criteria matrix ($V4_{cri}$) for the issue "How to persist data?". The pro-arguments are reused as con-arguments for the opposite alternatives using the *attacks* relationship.

Exercise 7 *Until now, we explored rationale in two views called node-link tree diagram (knowledge tree view) and criteria matrix. Get familiar with the following other knowledge visualizations you can access from the scenario: Indented outline (another knowledge tree view), node-link diagram, adjacency matrix, and chronology view. Discuss the strengths and weaknesses of each view.*

The strengths and weaknesses of each view are discussed in Section 11.2.3.

Capturing Rationale in Code, Commits, and Chat Messages

The next two exercises introduce capturing rationale outside of Jira.

Exercise 8 *Create a feature branch for the scenario in your git repository with the name "IOS-1.implement.scenario" (one team member). Check out the feature branch and add dummy code files to the repository, e. g., DataPersistenceManager and PasswordPersistenceManager (every team member). Add decision knowledge in code comments to the code files, e. g., the issue "How to persist passwords?" with solution options and arguments. Commit the changes (new code files) with the commit message "Use SHA-1 to encrypt passwords!". Merge the branch back to the mainline when every team member made their commits. Navigate to the Jira view for the scenario. Check that the decision knowledge from code is added to the knowledge tree and shown in the view for "knowledge in git". Check that the commit message was transcribed into a comment of the scenario in Jira. Annotate the decision in the transcribed commit message and add the issue "How to encrypt passwords?" in Jira.*

The students learn that the decision knowledge from code comments and commit messages is added to the knowledge graph and visualized in Jira. The last step of annotating the decision can be omitted if the automatic text classification is enabled. Now, the students discuss rationale in chat messages and use the decision knowledge icons in Table 12.2 to annotate them:

Exercise 9 *Informally discuss the issue "Which programming languages should we use?" in chat messages. Add solution options, pros, and cons. Annotate the rationale elements with the decision knowledge icons and export them to Jira. Navigate to Jira and check that you can see the rationale elements there.*

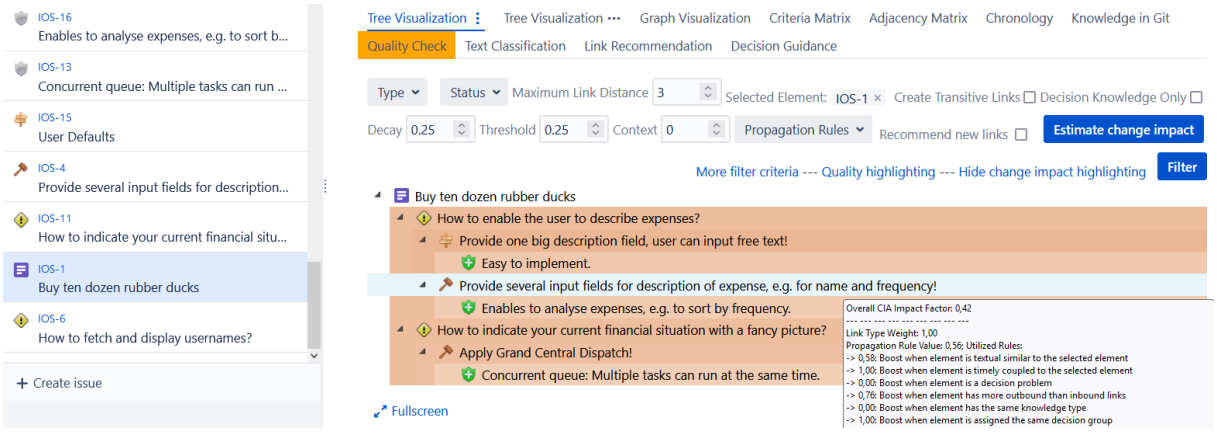

Figure 12.6.: Knowledge tree view (indented outline, $V2_{ind}$) for the scenario with change impact highlighting.

Change Impact Analysis

In this exercise, the students learn how to exploit the documented knowledge during changes. They get to know the change impact analysis feature.

Exercise 10 *Apply change impact analysis on the scenario in the project. Which impact factor is assigned to the decision "Provide several input fields for description of expense, e. g., for name and frequency!" that is indirectly linked to the scenario? How is the impact factor calculated?*

The impact factor is 0.42, as shown in the tooltip in Figure 12.6. It is calculated based on propagation rules, also shown in the tooltip.

Rationale Backlog and Quality Checking as a Developer

The following exercise introduces the rationale backlog and quality checking features.

Exercise 11 *Add the issue "How can the frontend communicate with the backend?" to Jira without solution options. Navigate to the rationale backlog and open the quality check view for the issue. What do you see?*

The answer is that the students see the open issue in the rationale backlog. The issue is colored in red to nudge the developers to resolve it. The quality check view explains that 1) the issue is open, 2) a decision needs to be linked, and 3) at least one alternative needs to be linked if this is configured in the definition of done.

Rationale-based Meeting Management

Capturing rationale pays off during sprint meetings. The meeting agenda has an information-sharing section that lists issues discussed and decisions made during the last sprint. If meeting agendas are managed in Confluence, the rationale elements can be easily imported. The import can be done in two ways: 1) via a built-in macro for Jira tickets only or 2) via the ConDec Confluence plug-in for decision knowledge from various documentation locations.

Exercise 12 *Gather your team, open your Confluence space and create a new page called <Rationale Lecture> (one per team). Create a subpage called <Your Name> (every team member). Use the Jira issues macro to display decisions from your Jira project. Answer the question: How many decisions do you see?*

The Jira issues macro is used in the exercise, and the query is expressed using the *Jira Query Language*. The query is:

```
project = <Project Key> AND
issuetype = Decision AND
created > -7d.
```

In the example, two decisions are shown. However, no decisions documented in the Jira ticket comments or other documentation locations are shown. With the next exercise, the students import rationale elements from the distributed documentation locations using the ConDec Confluence plug-in and its decision knowledge import macro.

Exercise 13 *Add a stand-up table showing all rationale elements created or updated within the last two weeks to your sub-page called <Your Name> (every team member). Use the decision knowledge import macro to import rationale elements from distributed documentation locations. Answer the question: How many open issues do you see?*

In the example, at least one open issue that was added in Exercise 11 is shown.

Inspecting Rationale Quality as a Rationale Manager

Now, the students learn about the tasks of the rationale manager role. They get familiar with the knowledge dashboard, which shows metrics calculated on the knowledge documentation.

Exercise 14 *Navigate to the knowledge dashboard. Inspect the intra-rationale completeness metrics. For how many decisions is at least one pro-argument documented? Are there decisions without pro-arguments? Inspect the decision coverage metrics. Are there scenarios that are not covered by a decision?*

In the example, all decisions have a pro-argument documented. The students might have added decisions as part of Exercise 5, Exercise 6, Exercise 8, or Exercise 9 without pro-arguments. In the example, there is only one scenario. This scenario is covered with at least two decisions (Figure 12.3), meaning these decisions are indirectly, i. e., transitively, linked to the scenario in a maximal link distance of 3. The instructor tells the student that the rationale manager would assign rationale improvement tasks to the developers.

Creating Release Notes at End of Sprint

The students should assume that they have completed a sprint. To publish what they did, the students create release notes with the issues they solved and their decisions.

Exercise 15 *Set the status of the scenarios in Jira to done. Navigate to the release notes view in Jira and create new release notes with the default settings. How many decisions are included in the release notes?*

The answer is that all decisions documented in the project are included in the release notes.

12.2. Results of the First Instantiation

This section presents the results and discusses the lessons learned from the first instantiation of the lecture in the iPraktikum at the Technical University of Munich on November 8, 2018. The lecture included six exercises (Exercise 1, Exercise 2, Exercise 3, Exercise 4, Exercise 5, and Exercise 12). The lecture only included this subset because the ConDec views and features applied during the other exercises did not yet exist or were not mature.

The instantiation of the lecture included a study to assess the attitude of the students toward the dissemination of ConRat and ConDec. The research question of the study is *Do developers accept the dissemination?* We issued several polls to receive feedback from the students in a survey. The survey questions ask for the student's acceptance of tool features. Thus, the answers to the survey questions are only indirect indicators to answer the research question. The study reported here is a pre-study to the user acceptance study described in Chapter 11. The remainder of the section provides the exercise results and the survey questions and answers.

Table 12.3.: Students' attitude toward the presented methods and tools for rationale management and weighted means $\mu_w = (-2 \cdot \#strongly\ disagree - 1 \cdot \#disagree + 0 \cdot \#neutral + 1 \cdot \#agree + 2 \cdot \#strongly\ agree)/\#participants$.

ConDec Statement	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	μ_w
Linking an existing ticket to a scenario is easy.	4	2	11	16	26	1
Discussing rationale using ConDec is simple.	0	3	14	35	4	0.7
I would capture rationale in Jira ticket comments.	0	18	24	17	2	0
I would apply presenting rationale in Confluence pages.	0	7	13	23	12	0.7

The first poll asked for the team a student belongs to. With this poll, we wanted the students to get warmed up for voting and to get the number of participating students. In total, 88 students answered the poll, i. e., about 88 students participated in the lecture. This number is a reference point for the following quantitative evaluation results. Exercise 1 was answered by 64 students, i. e., 73 % of the participating students, of whom 63 gave the correct answer.

Regarding Exercise 2, we initially asked the students to describe the difference between the alternatives and decisions. Since this question seemed hard to answer, we restated the question to answer which of the solution options focuses on requirements, and which of them on implementation. This exercise was verbally performed. ConDec's decision grouping feature (Section 7.9) did not yet exist at the first instantiation of the lecture. We added the task of applying the decision grouping feature after the second ISE project.

After performing Exercise 3, we asked students to assess how easy it was to link an existing issue to a scenario via a poll. They were asked to rate the statement *Linking an existing issue to a scenario is easy* with one answer from a five-point Likert scale. Fifty-nine students participated in this poll, of whom six disagreed, 11 were neutral, and 42 agreed with the statement (Table 12.3). After the lecture, we checked whether the teams correctly performed Exercise 3. In every team project, the issue was correctly linked to the scenario. We collected no results for or attitudes toward Exercise 4. For Exercise 5, we created a poll in which the students could rate the statement *Discussing rationale using ConDec is simple.* Fifty-six students participated in this poll, of whom three disagreed, 14 were neutral, and 39 agreed with the statement (Table 12.3). The students documented all rationale elements as entire Jira tickets since this was the only documentation location supported. We asked the students to provide written feedback on discussing rationale (Table 12.4), and we analyzed the knowledge graphs. A total of 126 rationale elements were documented, i. e., a mean value of 12.6 rationale elements per team with a Standard Deviation (SD) of 9.4 elements. That means that each student contributed a mean value of 1.4 rationale elements. A mean value of 3.3 alternatives was documented per team (SD = 1.2). 61 % of the alternatives correctly ended with an exclamation mark. Seven of the ten issues correctly ended with a question mark. Studying the rationale trees that the students created, we noticed that we did not ask the students to make a decision. Thus, only four decisions were documented. As a result, we restated the exercise. The types of elements were correctly chosen, e. g., arguments were not accidentally classified as alternatives. The students seem to understand the elements of the rationale model (Table 12.2) since they correctly classified the element types.

At the time of the first instantiation of the lecture, we have been developing the feature of documenting rationale in Jira ticket comments. After demonstrating that rationale could also be captured in the comments of the scenario, we asked the students to rate the statement *I would capture rationale in Jira ticket comments.* Sixty-one students participated in this poll; 18 disagreed, 24 were neutral, and 19 agreed with the statement (Table 12.3). The reason for the

Table 12.4.: Summarized feedback provided by students.

Student Feedback	Rationale
Deletion of elements is not possible.	Permission scheme in the Jira project forbids deletion for non-admin users.
If there are many alternatives and arguments, it is hard to get an overview. The user needs to scroll very far to the right to see all the content.	A better graph visualization and better, easy-to-apply filter possibilities are needed.
Rationale elements should not always be a single ticket. This seems to end up in a ticket overflow. Pro- and con-arguments should be comments.	Capturing rationale elements in separate Jira tickets has the advantage that they can easily be imported into Confluence. We also develop the ConDec Confluence plug-in to import rationale from comments.

overall neutral attitude toward this feature might be that we presented the prototype and that the students could not use it themselves.

After the students performed Exercise 12, we asked them to rate the statement *I would apply presenting rationale in Confluence pages.* Fifty-five students participated in this poll, of whom seven disagreed, 13 were neutral, and 35 agreed with the statement (Table 12.3). A mean value of 58 students participated in the polls analyzed in Table 12.3, representing 66 % of all students.

The results collected during the lecture represent the students' first impression about the rationale management activities and the support through ConDec. The voting results on the statements in Table 12.3 indicate that most students liked applying rationale management. The written feedback summarized in Table 12.4 encouraged us to improve the visualization and filtering components of ConDec. The students seemed to like voting on the polls and performing the exercises since this made the lecture more interactive, as also observed by Krusche et al. (2017). We concluded that students accepted the dissemination from this instantiation and the application during the iPraktikum afterward. Subsequently, we improved ConDec by adding new views and features, and we added new exercises to the syllabus.

12.3. Related Work

This section discusses related work on disseminating rationale management in student projects focusing on teaching aspects. It includes related work investigating the students' acceptance after a project course instead of only a lecture, as reported in this chapter. Thus, this section also includes related work for the user acceptance study in Chapter 11, which reported results of applying ConDec over a semester. This section omits publications about evaluations of rationale management tools in student projects without the teaching focus, e. g., by Falessi et al. (2006) and Alkadhi et al. (2017a). Table 12.5 compares the dissemination approach in this thesis with related publications. In addition to these publications, the book by Bruegge and Dutoit (2010) offers a thorough introduction to rationale management, including exercises for students.

Dutoit et al. (2005) discuss experiences with the integrated, rationale-based modeling environment Sysiphus in various software engineering courses. By applying rationale management, they aim to enhance the communication between instructor and students, to support students in reflecting on their work, and to enable the instructor to monitor students' progress better. We also experienced these benefits when students applied rationale management during the semester as part of the agile project courses (iPraktikum and ISE projects). Like the syllabus presented in this chapter, Dutoit et al. (2005) describe exercises to teach rationale management. For example, the instructor presents an incomplete rationale for an architectural decision, only listing alternatives and asking students to complete it, similar to the Exercise 5 and Exercise 6. The syllabus for disseminating ConRat and ConDec includes other process-oriented exercises, such as

Table 12.5.: Related work on disseminating rationale management in student projects.

	This Thesis	Dutoit et al. (2005)	Malloy and Burge (2016)	Schubanz and Lewerentz (2020)	Capilla et al. (2020b)
Tool/Approach	ConDec	Sysiphus	SEURAT_Edu	MADR	ADMentor
Rationale Management Guidelines and Process Extensions	✓ ConRat	✗	✗	✓	✗
Description of Exercises	✓	✓	✓	✗	✗
Survey on Students' Attitude	✓	✗	✓	✓	✓
Experiment to Study the Effectiveness of the Approach	✗	✗	✗	✗	✓

rationale-based meeting management and release note creation at the end of a sprint. Like the Sysiphus modeling environment, ConDec integrates the rationale and system knowledge elements, such as requirements. In addition, ConDec supports linking rationale with development tasks since they represent where developers need to solve issues related to design and implementation.

Malloy and Burge (2016) developed the software engineering using rationale tool SEURAT_Edu as a web-based system that replaces the former SEURAT Eclipse[5] extension. Like ConDec, SEURAT_Edu aims to support students in making the best decision for an issue under consideration by explicitly reasoning about design alternatives. SEURAT_Edu enables the teacher to supply students with an incomplete rationale they are asked to complete in exercises. Like the quality checking features of ConDec, SEURAT_Edu performs automatic error checks, e. g., whether there are issues not solved by a decision. In addition, SEURAT_Edu enables the teacher to create a set of "solution" rationale. It displays the status to which students reached a solution to encourage them during their tasks. During their evaluation, Malloy and Burge surveyed the students' attitudes. They found that the students using their tool felt they considered more alternatives and put more thought into decision making. Like their tool, the ConDec Jira plug-in is web-based, allowing students to collaborate efficiently. SEURAT_Edu integrates with learning management systems such as Moodle[6], which has the advantage that the learning management system takes care of authentication and assignment creation. For ConDec, this is handled by the underlying systems, such as Jira, Confluence, and Bitbucket.

Schubanz and Lewerentz (2020) performed eight case studies between 2015 and 2019 with software engineering students. They disseminated rationale management in workshops with the students. During the workshops, they discussed issues relevant to the students, elicited alternatives, made decisions, and documented them. Further, they provided the students with guidelines. Similar to the ConRat life cycle model extension (Chapter 6), Schubanz and Lewerentz (2020) extended Scrum with two process elements for rationale management: 1) At the end of each sprint planning, the Scrum Master identifies and documents the three most relevant decisions for the current sprint together with the team. Their students documented decisions in markdown files using the Markdown Architecture Decision Record (MADR) template (Kopp et al., 2018; Kopp and Armbruster, 2019). They uploaded the MADR files to their project git repositories. 2) At the end of the sprint, the Scrum Master reviews the documented decisions with the team. In case of changes, inconsistencies, or recently emerged important decisions, the Scrum Master has to revise the documentation. Schubanz and Lewerentz (2020) surveyed the students' attitudes toward performing rationale management after completing the case studies.

[5] https://eclipse.org
[6] https://moodle.de

Overall, the attitudes were positive. Like us, they encourage the capture of decisions and their rationale to be part of software engineering education.

Capilla et al. (2020b) describe a setup to teach students software architecture decision making and documentation using the ADMentor tool. They surveyed the students' attitudes using a questionnaire. In addition, they investigated the effects of the rationale management approach on the quality of software architectures modeled by students. They conducted an experiment and used previous course instances and the respective architecture models as the control group. They found that groups using ADMentor arrived at "better" architectures as they identified the design problems more clearly. Since the students developed different software systems in each validation project, we cannot compare their solutions with previous projects to investigate the effectiveness of applying rationale management. However, the students' attitudes we surveyed via polls and later in the projects described in Chapter 11 were overall positive.

12.4. Conclusion

This chapter presented a syllabus for disseminating rationale management supported through ConDec. The syllabus contains 15 exercises in which the students apply ConDec. The chapter reported the results of the first instantiation of the syllabus as a lecture. The results of this pre-study provided the first indication of the feasibility and user acceptance of ConDec. Chapter 11 described further user acceptance studies performed afterward.

The syllabus disseminates most ConDec views and features but not all because it would be too much information for one lecture. Omitted features are the presentation and merge check of decision knowledge in pull requests, just-in-time prompts, wrong link marking, and the features of the ConDec integrated development environment plug-ins. In the validation projects, we introduced these features in follow-up meetings with the students.

A contribution of the syllabus is to teach the integration of rationale management into existing development workflows, such as rationale-based meeting management and release note creation. While others also perform exercises teaching documentation and the concept of intra-completeness, this syllabus contains various new exercises using innovative ConDec features. The syllabus includes exercises on manual activities with a hint of how ConDec's recommendation systems support and automate them. For example, ConDec's automatic text classification and link recommendation system support the manual activities of documenting and linking rationale. The students must understand the manual activities before they are automated, and this syllabus teaches manual activities and innovative support. Practitioners can use the syllabus and exercises as a starting point to incorporate ConRat and ConDec in their software development process.

Part V.

Conclusion

Chapter 13

Summary

"When discussing research design, researchers often cite a number of dichotomies: field versus laboratory research, desk versus field research, in vitro versus in vivo, quantitative versus qualitative research, fixed versus flexible research, positivist versus interpretivist research, inductive versus deductive research, exploratory versus confirmatory research, rigor versus relevance, internal versus external validity... The dichotomy suggests two mutually exclusive "extremes", as if it were a tradeoff, while in actual fact reality is more complicated."

—Stol and Fitzgerald, 2018

This chapter summarizes the goals of the thesis and its contributions. The contributions are valuable for both software engineering practitioners and researchers. Continuous Software Engineering (CSE) is a software life cycle model that is characterized by frequent changes. The overall goal of this dissertation was to integrate rationale management into CSE while treating the three rationale management problems of intrusiveness and effort, high amount of distributed knowledge, and low documentation quality. We refined the overall goal into one technical research goal, five knowledge goals, and one instrument design goal.

Knowledge goal 1 was to investigate the current state of the practice of rationale management during CSE. We conducted a semi-structured interview study with practitioners from the industry to investigate how practitioners define and perform CSE, to understand the rationale-management practices and problems during CSE, and to elicit improvement ideas (Chapter 3). The results of the interview study indicated that rationale management is not systematically integrated into CSE. The practitioners capture decision knowledge informally, for example, in natural language discussions in issue tracking systems. For them, capturing rationale has many positive effects, such as improved decision-making and change processes, accountability, knowledge sharing, and reuse. However, the practitioners lack systematic techniques and tools for rationale management. The reported challenges confirm the rationale management problems: 1) Documenting rationale is seen as an overhead and intrusive. 2) It is not clear how to access and exploit the decision knowledge documentation when needed during software evolution. 3) Rapid changing decisions lead to outdated documentation, i. e., inconsistency between the captured decisions and their implementation. Even if the decision knowledge is captured, e. g., in the issue tracking system, it is difficult to access in the context of requirements, code, and other software artifacts. The interview study contributed functional and non-functional features for continuous rationale management that benefit practitioners. The results are interesting for practitioners to compare their current practices and to reflect on the necessity for adopting ConRat and ConDec. The

interview study is also interesting for researchers when performing future interview studies to adopt the methods and to compare the findings.

Knowledge goal 2 was to investigate the current state of rationale management support with classification and recommendation (Chapter 4). The study contributed an overview of four approaches helpful for rationale management: Automatic text classification, automatic linking, decision guidance, and consistency support. The study outlined the functioning and evaluation of the approaches and revealed that only a few approaches are implemented in development tools, hindering practitioners' usage. The results are valuable for researchers as a basis for future systematic literature reviews and primary studies.

The technical research goal was to develop the treatment for the three rationale management problems based on the outcome of the interview study and systematic mapping study (Chapter 5). The treatment consists of the ConRat life cycle model extension (Chapter 6) and its support through the views and features of the ConDec plug-ins (Chapter 7). ConRat treats the problem of *intrusiveness and effort* by integrating rationale management activities for collaborative, incremental, and rational decision making, documentation, exploitation, and quality assurance of decision knowledge into existing CSE workflows. ConDec minimizes intrusiveness by integrating into standard development tools rather than providing a standalone tool. ConDec enables capturing decision knowledge using lightweight annotations in various documentation locations typical for CSE, in particular, Jira ticket description and comments, commit messages, and code comments. To enable the exploitation of the *high amount of distributed knowledge*, ConRat defines a *knowledge model* and ConDec instantiates the model as a *knowledge graph*. ConDec provides comprehensive knowledge visualizations that developers can access from various tools and artifacts. ConRat operationalizes rationale quality and ConDec implements the concepts, to treat the problem of *low documentation quality*. ConDec is the first tool that supports high-quality rationale documentation with a *definition of done* and the *decision coverage* metric that counts decisions traceable from requirements and code. ConDec offers a *rationale backlog* showing knowledge elements violating the definition of done. It offers a *knowledge dashboard* presenting the decision coverage and other metrics. ConDec reduces the developers' manual work and motivates them using *recommendation systems* and *nudging mechanisms*. ConDec offers six recommendation systems, inspired by the approaches identified in the systematic mapping study. First, the *quality checking recommendation system* indicates violations of the definition of done with friction nudges and in just-in-time prompts. Second, *change impact analysis* highlights decisions and other artifacts affected by a change in the views on the knowledge graph to support consistent changes. Third, *decision guidance* recommends solution options to decision problems from other software development projects and external knowledge sources. Fourth, the *link recommendation system* detects missing links and duplicates within the knowledge graph. Fifth, ConDec supports developers in explicitly capturing decision knowledge through *automatic text classification*. So far, automatic text classification has only been applied retrospectively. ConDec is the first tool that integrates automatic text classification into the ongoing development during CSE. Sixth, the *summarization of source code changes* helps to make tacit decisions explicit. Adopting ConRat and ConDec is particularly interesting for practitioners since managing rationale has many positive effects, such as improved decision-making and change process, knowledge sharing, reuse, and accountability. ConRat shows how practitioners can integrate rationale management activities into their current workflows. They can easily extend their tools and systems with the views and features of the ConDec plug-ins. ConRat and ConDec are interesting for researchers as a basis for further development and empirical studies on rationale management.

Knowledge goal 3 was to analyze the decision knowledge documentation of six case study projects to validate the feasibility of ConRat and ConDec and to investigate the outcome of decision knowledge documentation in practice (Chapter 9). In total, the software developers of the six projects documented more than 1000 decisions using ConDec in four different documentation

locations, which demonstrated ConDec's feasibility for decision documentation. The developers documented different types of decisions. The *decision types* with most decisions assigned are *frontend*, *backend and data storage*, and *functionality-driven* decisions, followed by *executive* and *quality-driven* decisions. Most decision knowledge elements are completely documented according to four criteria of the *intra-rationale completeness* if checked and enforced by ConDec. The *decision coverage* of the requirements and code files varies. A low decision coverage indicates requirements and code files with missing decision making or missing documentation of decisions or links. A very high decision coverage indicates wrong links in the knowledge graph or decision *hot spots* in that it might be worth splitting a requirement or code file. The study is interesting as it showed how to *operationalize the quality of knowledge documentation*. Practitioners can reflect on the knowledge documentation quality of their projects using the metrics. The study is interesting for researchers as a basis for further work on developing guidelines for creating high-quality decision knowledge documentation and tool support.

Knowledge goal 4 was to validate the effectiveness of automatic text classification (Chapter 10). The automatic text classification integrated into ConDec can detect rationale with an F1-score up to 0.81 and classifies the detected rationale into different rationale elements with an F1-score up to 0.69. The results indicated that automatic text classification could support developers in documenting rationale and retrospective reconstruction. Still, researchers can further improve ConDec's automatic text classification in the future and use the study results as a benchmark. While the classification results can be wrong, automatic text classification is a nudging mechanism, i. e., an incentive for creating and improving the rationale documentation.

Knowledge goal 5 was to validate software developers' acceptance of ConDec (Chapter 11). The study contributed feedback and improvement ideas for the views and features of the ConDec plug-ins. The study participants liked the following ConDec views and features the most: Documentation of decision knowledge in Jira as tickets or in ticket text, decision grouping, interaction and filtering possibilities in the knowledge views, the knowledge tree views and criteria matrix, the stand-up table for meetings, the quality checking functionality including nudges, the rationale backlog, and the knowledge dashboard. While one practitioner from the industry particularly questioned ConDec's benefit for decision making, the application of ConDec in the case study projects indicated that it helps the discussion and reflection on decisions. The results indicated that software developers accept ConDec and that ConDec can support ConRat in industrial projects as long as the underlying tools fit, e. g., Jira is used as the issue tracking system. The improvement ideas are interesting for researchers as a basis for further development and they can adapt the study methods to validate user acceptance in other projects.

The instrument design goal was to design a syllabus for disseminating ConRat and ConDec (Chapter 12). The syllabus consists of exercises where students perform rationale management activities using the ConDec views and features. It is interesting for practitioners as a starting point to adopt ConRat and ConDec.

In summary, the dissertation contributed a validated approach for continuous rationale management consisting of the ConRat life cycle model extension and the ConDec plug-ins. The thesis also contributed empirical knowledge of rationale management practices, problems, and solutions before ConRat and ConDec and the outcome of rationale documentation created with ConDec.

Chapter 14

Future Work

Digital ethics is the new organic movement.
—Spiekermann, 2019

This chapter discusses future work. Based on the ConRat life cycle model extension, the ConDec plug-ins, and the empirical evaluations, several directions for future work exist.

The knowledge management strategy of *codification* aims for explicit decision knowledge documentation, while the *personalization* strategy focuses on the interaction among knowledge workers (Section 2.2.3). Babar et al. (2007) found that codification in knowledge management research is often done intentionally, while personalization is done unintentionally. They suggest combining codification and personalization into a hybrid approach with an intentional focus on both strategies. ConRat and ConDec support both knowledge management strategies. ConDec supports the codification through the features for knowledge documentation. ConRat supports personalization by integrating rationale management activities into the CSE process, e. g., through rationale-based meeting management, which helps communicate knowledge instead of only storing it. In the future, personalization could be further enhanced and detailed.

Robillard et al. (2017) described the idea of an *on-demand developer documentation* that generates documentation when developers need it. The long-term vision of this thesis is an *on-demand decision documentation* which supplies developers with the rationale documentation they need when they evolve a software system. Information could be inferred using the retrospective reconstruction of decision knowledge, as done by researchers studying the effectiveness of automatic text classification (Alkadhi, 2018). However, decision reconstruction is likely to be incomplete, e. g., due to missing alternatives (Bruegge and Dutoit, 2010). Robillard et al. (2017) in particular discuss rationale as part of the on-demand developer documentation. They state that rationale cannot automatically be inferred and that an "incentive to motivate the more systematic capture of rationale" is needed. ConDec addresses both problems by supporting developers to capture decision knowledge and presenting it for easy use. ConDec supports the creation of rationale documentation during development instead of post-mortem.

Razavian et al. (2023) describe the vision of *On-demand Architectural Knowledge Systems (ODAKS)*, which addresses knowledge management issues holistically. For example, ODAKS use assistive conversation to provide architects with relevant knowledge. They use probing to understand the architects' decision problems, recommend knowledge, and present reflective hints to mitigate human decision-making issues, such as cognitive bias. While ConDec's nudging mechanisms and recommendation systems (Section 7.6) partly implement the ideas, ODAKS contains many ideas for future work (Razavian et al., 2023).

ConDec supports written informal discussions, but issues and decisions are also discussed verbally. ConDec could support *capturing informal discussions via voice* (Capilla et al., 2020a).

ConRat requires the manual inspection of the semantic content of the rationale as part of reviews. ConDec supports its checking through knowledge visualization and change impact analysis. In the future, ConDec could be extended with *semantic inferences* that inspect the content of the rationale. For example, the SEURAT tool alerts if an earlier discarded alternative is selected (Burge and D. C. Brown, 2008a). Consistency support, as suggested by Lytra et al. (2015), could be added to ConDec. ConDec establishes *lightweight traceability*, whereas the consistency support uses *transformations (formal mappings) between architecture and code*. These transformations are more powerful than traceability links since they create decision models that are interrelated with architecture models and the corresponding code. Hence, changes in the models can be propagated to the code. However, transformations are more intrusive than traceability links because they require extra notations. Thus, the trade-off between lightweight capturing and powerful consistency checks needs to be considered.

Since the usefulness of the *rationale exploitation depends on the trace link quality*, mechanisms for trace link improvement and maintenance should be added (Hübner and Paech, 2020). Wrong link detection should become a part of the link recommendation system in the future. Change impact estimation should combine more approaches (Lehnert, 2011; Kretsou et al., 2021), its visualization could be improved using *impact cities* (Kugele and Antkowiak, 2016), and automatic change execution could be supported to resolve inconsistency and duplicates automatically.

Six ConDec plug-ins exist for the issue tracking system Jira, the wiki system Confluence, the integrated development environments Eclipse and VSCode, the chat system Slack, and the web-based git-client Bitbucket. The user acceptance study in Chapter 11 showed that practitioners use partly other tools. In the future, *further ConDec plug-ins must be developed for other tools used in practice*, e. g., GitLab, Azure DevOps, and IntelliJ.

ConRat and ConDec provide a solid foundation for conducting *further empirical work for their validation and answering knowledge questions beyond validation.* ConRat and ConDec should be applied in industry projects to investigate decision knowledge documentation in practice, as started in Chapter 9. For instance, it is interesting to explore further which decisions practitioners consider important to be captured to develop guidelines on what to capture. The thesis investigated the validation aspects feasibility, effectiveness, and user acceptance (Section 1.3). We studied the effectiveness of automatic text classification in terms of F-scores. In addition, future studies should investigate the effectiveness from the developers' perspective and the efficiency of applying continuous rationale management supported with ConDec. Like in Bratthall et al. (2000), the hypothesis is that developers are faster in doing changes and arrive at better design solutions with ConDec than without in a software development project unknown to them. However, this hypothesis needs to be tested in an experiment.

Literature exists that critically reflects the current CSE processes. Spiekermann (2019) criticizes that—while CSE helps optimize software's technical functioning—it misses techniques to reflect whether the software adds ethical values that contribute to a better society. Software engineering generally misses techniques for incorporating digital ethics, but CSE aggravates the disregard for ethical values. During CSE, development tends to start early before clarifying whether there are negative impacts. For example, Spiekermann (2019) describes a food delivery software that leads to bad eating habits and isolation of its users. Transparent decisions are one aspect of explainability, an important non-functional requirement (Chazette et al., 2021). ConRat and ConDec are intended to support responsible decisions that support real human lives (Jonas, 1985). Future research should focus on *evaluating decisions regarding their ethical impacts.*

Part VI.

Appendix

Appendix A

Digital Appendix for Tools and Data

"I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail."

—Maslow, 1962

This thesis has a digital appendix[1] in the zenodo platform to enable other researchers to reproduce the results of this thesis and to guarantee the availability of the ConDec plug-ins and the data in the future. The digital appendix contains the following artifacts for the chapters:

- Chapter 4: Protocol of systematic mapping study and R script for analysis.
- Chapter 7: Source code and binary releases of the ConDec plug-ins.
- Chapter 9: Anonymized decision knowledge documentation of the case study projects and R script for analysis.
- Chapter 10: Ground truth data used to evaluate the automatic text classification.
- Chapter 11: R script for analysis of usage frequencies from log files.

[1] zenodo.org/communities/conrat

Appendix B

Supplementary Material of Interview Study on State of the Practice

"Some companies automate knowledge management; others rely on their people to share knowledge through more traditional means. Emphasizing the wrong approach—or trying to pursue both at the same time—can quickly undermine your business."

—Hansen et al., 1999

This appendix provides supplementary material for the interview study in Chapter 3. Section B.1 provides statements by the practitioners. Section B.2 describes the related work.

B.1. Interview Statements by Practitioners from Industry

We removed mentions of companies, persons, products, and self-developed tools to anonymize the statements. For the German interview transcripts, we translated the statements into English using the DeepL translator[1].

Section B.1.1 lists statements regarding the research question *How do practitioners apply CSE during software evolution?* (RQ1). Section B.1.2 lists statements regarding the research question *How do practitioners manage decision knowledge during CSE?* (RQ2). Section B.1.3 lists statements regarding the research question *How can rationale management in CSE be improved according to practitioners?* (RQ3).

B.1.1. Statements regarding As-is State of CSE in Industry

Table B.1.: Anonymized examples of practitioners' CSE definitions and the identified CSE elements (RQ1.1).

CSE Element(s)	Practitioner's Statement regarding the CSE Definition
Learning from usage data and feedback	For me, a feedback loop would be central, which runs through the whole thing. Not only from the finished product back to the software but also through the entire process until the software is built again. So I would see a feedback loop as the main criterion.

Continued on next page

[1] https://www.deepl.com

CSE Element(s)	Practitioner's Statement regarding the CSE Definition
Involved users and other stakeholders, learning from usage data and feedback, continuous planning activities, continuous requirements engineering	As a company, you do CSE when you are also continuously in contact with the customer or the users of a system to get new requirements or changed requirements for the software. You can understand that the user suddenly interacts differently with the system and define the appropriate tasks and develop the functionality possibly differently than it was originally intended (because the behavior to use it also changes). So CSE means that I consider the wishes of the customer in my strategy for further development and that not only randomly but also in a structured way. You determine the next development steps also by what you get back from the user as feedback.
Involved users and other stakeholders, comply with shared ruleset, management commitment, focus on features	CSE to me means that you can implement a feature pretty quickly to a ticket, that a review happens quickly, and that you get that feature live extremely quickly so that the customer or the end-user sees that. For me, that's CSE when the employees of the company can get their features live without being told that a manager still has to look over it or that there's only a release date every six months, for example. So in CSE, the teams should be able to do that relatively independently.
Comply with shared ruleset, self-reflection and discipline, continuous integration of work, version control	I'm not focused on the academic world, but we have cooperation with other companies that are also discussing similar things. It includes always committing the code and some have a strict definition that when you do continuous integration, you should always commit your code, every single day. You should never have any development only residing on a developer computer. You should always merge it back to the main code base before you leave for the day. In those terms, we are very far away from it yet. There are several maturity levels when it comes to the fulfillment of continuous integration.
Involved users and other stakeholders, learning from usage data, continuous integration of work, branching strategies	I believe CSE is consistently implemented when (a) there is a very sparse branching strategy and every commit leads to a finished product that is then seen by enough people. And (b) this feedback then flows directly back into new builds that are created as a result.
Continuous planning activities, continuous requirements engineering, focus on features, agile practices, continuous deployment of releases, automated tests, version control	For me, CSE starts right at the beginning with the requirements, i. e. with the continuous elicitation or adaptation of requirements. This means that continuous change management is part of it as well as continuous grooming of a feature/story backlog, i. e. that the list of things to be done is continuously maintained and that this information is continuously passed on to the teams. This implies that re-prioritization can be done continuously and that there is a kind of to-do pipeline whose contents are jointly prioritized, picked out, and processed by e. g. the Product Manager and the development teams (like on an assembly line). As soon as a feature or story is implemented, it is immediately tested automatically, i. e. automated tests must exist for it. Then this feature is also automatically brought into production. You can call this a pipeline, I would call it an assembly line because it works like an assembly line from requirements analysis/requirements engineering to deployment to production. And in that, of course, there are all these technical aspects like code management, issue management, test management, and things like that. That's the technology, but you should support a process and that's what's lacking most of the time because the process in most projects is not aimed at mapping a pipeline like that. If all these steps are met, I would say a company is doing CSE. However, if a company first writes a requirements catalog (and is busy with it for a year), prioritizes the requirements, and then the development team goes agile with Scrum or SAFe to implement these requirements in two- to three-week sprints where they pick the requirements to have something finished in two to three-week intervals, that is not continuous. That has nothing to do with agility and that has nothing to do with CSE. Continuous would be when, as soon as a feature is ready, you can show this feature to the stakeholders.
Continuous integration of work, continuous delivery	CSE is a very broad topic and deals with many different issues. For me, the focus is on continuous integration (i. e. real development) and delivering the software quickly and regularly.

Continued on next page

CSE Element(s)	Practitioner's Statement regarding the CSE Definition
Focus on features, continuous delivery, continuous deployment of releases, automated tests, version control	You are now asking me to draw a line on the floor and say "this is where it starts and that is where it ends". I don't like to do that, I think it's gradual. Some do it more, some do it less. It's not continuous in that sense, because software ultimately consists of bytes, and that's not analog, it's digital. In this respect, the software that is in production does not change continuously in the mathematical sense. What is meant by this is that the rollout into production happens more or less constantly. In this project, it was now the case that Monday to Thursday was usually rolled out in the morning. So there were daily releases. So those are still releases and if you're nitpicking, you can say that's not continuous yet. But there was the possibility that even a single developer, if they find a bug, can push their microservice into production in between, past the daily rhythm. That is wanted, that is desired. You will agree on this with your IT, you don't just do it on your own, but it's okay. The difference between continuous is that the threshold and the time between development and production has been reduced considerably. How do you manage that? We used to do quarterly releases, which means I code today and in late summer it goes into production. You used to do that because you had a big test department that tested the whole thing front to back and back to front. Today, we afford to just leave that out because we say we have automated testing. Unfortunately, not everything is still automated. Even back then, there was not only a release every quarter of a year, but sometimes a bug was found in production and then this bug was fixed (usually only three to four lines of code change) and pushed directly into production (sometimes even bypassing the test team if it had to go very quickly). Of course, this was all secured, there was version management control over it. That means we also learned back then that if you just make the change small enough, the risk is small. Now we don't do the huge release anymore, but push one small feature after the other and deliver in much smaller crumbs to the customer.
Involved users and other stakeholders, learning from usage data, continuous planning activities, agile practices, continuous integration of work, continuous delivery, automated tests	My understanding is that the continuous loop is the important thing. So you're iterating on-going, incorporating customer feedback. So ongoing, continuous improvement of the product is necessary. All processes, including those from the engineering perspective, are part of CSE: continuous builds, ongoing sprint planning, and everything that can be improved in the process to enable the rapid iterations of the product with respect to customer needs. I think if a company has the processes and tools to enable continuous delivery, testing, and feedback loops, uses them, and has integrated them, then they implement it. Then there's a level of reflection that can be achieved. But basically, if a programmer develops a feature, pushes commit, the whole thing automatically ends up with the person who's supposed to test it, and then you can deliver it to the customer with another button, and the feedback is integrated back in, through the product management side, then that's exactly what I understand it to be.
Continuous planning activities, continuous requirements engineering, modularized architecture and design, sharing knowledge, agile practices, continuous integration of work, continuous deployment of releases, automated tests	For me, CSE means that software development is no longer thought of in phases, but rather in short development cycles, after which executable, quality-assured software versions are always available and these are also passed on to the customer, i. e. made productive. One no longer differentiates when a project begins and ends, but the product life cycle is continuous, i. e., the project is always alive and change requests and new requirements can always be added as well as improvement potentials discovered. The project is continuously developed. This is what CSE means to me. In other words, all disciplines of software engineering must position themselves to enable continuous application life cycle management. In my opinion, a company must position itself differently in three areas to enable CSE: methodologically, concerning the technical infrastructure, and organizationally. 1) That means developing or introducing methods that enable continuous software development. I am thinking of the process models. For example, you have to introduce agile process models, you have to do requirements management differently, you have to define architectures differently, and you have to do testing differently. 2) This all requires tools and infrastructure, so you also have to build the software development infrastructure accordingly so that these fast cycles can be maintained. A lot of automation needs to be put in. 3) And in the third point (organizational), this means that the team formation is topic-specific, i. e., there are multidisciplinary teams that are responsible for a topic and work together continuously.

Table B.2.: Anonymized examples of practitioners' negative, neutral, and positive experiences per CSE category (RQ1.3).

Type	Practitioner's Description of Experience
	CSE Category: User
Positive	In our apps, there is a feedback channel, although classically as an email, but you can give feedback that something doesn't work or the content doesn't fit. We then receive a collection of this in our email inbox. And then we try to work through it. That's also such a feedback channel, where we see that there are problems with this or that feature, for example, that it's buggy on all Android devices. Then we try to get a solution for that. Since everything is manageable from an application point of view, we have kept it rather pragmatic.
Neutral	We do not have specific tools yet to involve users, but we try to communicate often in regular time intervals to get feedback.
Neutral	I think the user involvement is the part that is least technically supported in the process. The user part is the part that is still very fuzzy, you have to make sure that there is not just a sham continuity; if you have a support team that enters issues from time to time, I don't think you can already talk about CSE. Feedback has to become an active part of the process.
Negative	We currently have the problem that we receive user feedback very late and mainly in the form of *incidents* and *change requests*. We can only manually trace these incidents and change requests to the related development activities. That's where I would like to have an automated solution.
	CSE Category: Developer
Positive	The mindset of being agile, and also working with it, committing to it, using it, and evaluating it, was difficult to introduce in parts. Some of the developers are new, still very young, from university. It was no problem for them.
Neutral	There is the role of the *Review Manager*. Every week, someone else is the *Review Manager*. This is a normal developer who has the task of seeing if everyone has given it the thumbs up and makes the decision that it now goes into production. So that's always a different one every week.
Neutral	A current trend is that customers demand for *full-stack* developers.
Neutral	Unless the mindset of the employees is there, it won't work. In addition, the speed must be right for everyone involved. It may be that for a single employee, the feedback loop is closed too quickly, so they can't sustain attention. The attention must be able to be sustained by the developer without disengaging from the process. That's what an employee must be able to do.
Negative	What hasn't worked well, and presents great challenges, is to map the skills we already have in our company into this high speed of changing customer requirements. If someone has been programming ABAP or C for years, and now all of a sudden has to develop JavaScript. That permanent change is difficult for some people. And it requires different skills, because half a year (or two) ago, completely different programming languages or frameworks were in vogue than that is the case today.
Negative	But for some, it was completely new, it was something new and unfamiliar to work like that. It is so vaguely defined because no one says you have to do this or that. That was a challenge.
Negative	The biggest problem is that developers who are not yet familiar with it have to be trained on it. You have to know that you commit frequently, for example, that you should also make sure that the pipeline goes through, that it's deployed and that you can look at how the deployment works, i. e. how you use the tools. I've had the experience that some developers are not very familiar with that and think that that's just there and running or that they don't have to worry about it because they think that everything is automated. But of course, they do have to deal with it, e. g. whether the tests have run, that the code quality fits.
	CSE Category: Business
Positive	That only works with top management commitment, because it interferes with the culture, with the structures. And budgets are suddenly no longer my own, you have to think n-to-n and no longer in silos, that's hard. What works very well is that we have top management with us who want this, and have confidence in our ability.

Continued on next page

Type	Practitioner's Description of Experience
Negative	Many industrial projects would like to implement many of the CSE elements. But they are not allowed to or do not get the freedom and budget to implement it. It is often the case that project managers or middle management feel dispensable and use political means to prevent too much agility/too much CSE from coming in because they then suddenly no longer find themselves in such an environment.
	CSE Category: Development
Neutral	The thing to think about, or the thing to be able to do, is to break down a big functionality into small functionalities. There is mainly a recurring discussion about how features can be delivered and what is the smallest added value that can be realized within one or two days. In this process, we discuss among ourselves, i. e. me in the role of an architect and me as platform owner with the developers. I notice that the dual role is very difficult for me from an architectural point of view, where you try to anticipate as much as possible for future changes, with regard to a clear architecture, but on the other hand, the focus is also on delivering functionality quickly.
	CSE Category: Operation
Positive	We use Docker to realize the continuous delivery pipeline. For provisioning, we use tools such as Puppet, Ansible, or Chef to set up a server. That works quite well.
Neutral	In the afternoon, preproduction is closed. If new changes are made in development, they are no longer automatically transferred to preproduction. Then the automated tests run, on the environment that no longer changes. We have dozens of microservices there, so there's always a bit of movement. Then everyone has time to get back to us if anything doesn't work. And if everything works, the next day, the status from the afternoon the day before is put into production. Then the preproduction is opened again and everything that has accumulated up to then runs into it.
Neutral	The operating system is our base, we will not change it. We have to stay above it. The condition was that everyone who uses our software uses this operating system, with exactly the packages that we specified. And that, of course, is a very high requirement for the target systems.
Negative	Hierarchical structures and regulations hinder CSE. For example, the setup of infrastructure, such as requesting new servers, can take several months.
	CSE Category: Knowledge (also see Section 3.2.2)
Neutral	Each team or sub-team had to do knowledge sharing for itself, especially to absorb when new colleagues came and went.
Neutral	Continuous learning is supported through training to support the actual daily work and to get better there.
Neutral	To capture decisions, we have a wiki (Confluence) where we have captured major decisions, also so that later colleagues can read in there.
Negative	Knowledge management covers topics where there are problems in the projects. How do we make decisions in the first place? A saying of mine is "decisions should be made, not just happen". How do we make decisions, how do we document decisions? Is what is decided today still understandable a month from now? Can someone new to the team even read up on it anywhere? Things do get outdated. Who throws away the old stuff? No one dares to do that. There is a lot of documentation (including many decisions), but that is far too little used. These are exciting topics.
	CSE Category: Software Management
Positive	Continuous Integration is a very elementary component that we cannot do without. Neither from a developer, from product management, nor from any other stakeholder perspective. We automatically build all our products. Internal, i.e. custom software, ensures that this is also distributed internally.
Neutral	We use Scrum and have two-week sprints and all the regular Scrum meetings, so Planning 1, Planning 2, Review, Retro, and Grooming. Within these two weeks we have our tickets, which are on a Scrum board in the Jira system and have five steps, from *open*, *in progress*, *to review*, *to release*, and *done*.

Continued on next page

Type	Practitioner's Description of Experience
Neutral	For continuous planning, our tool is Jira. There is a well-filled backlog. There are Jira items such as epics at the entire level (for the 150 people) and then it is broken down to epics or stories for the own team. Then individual stories are created there and added to sprints. Then individual tasks are created day by day. The first thing you do for a sprint is a redefinition of what the tasks are under the stories, e. g. for the feature "Button on the user interface and something should end up in the database". This is then written in the story. That is then estimated, so estimation poker is what we do at that point as well. Then the next thing to do is to break that down.
Neutral	Interestingly, things that we used to think were quite important, like reproducibility of builds, no longer matter. You simply don't need it anymore. That is, if you want to reproduce in three weeks what we have in production today, you can't do that in most projects anymore, because software that we push into production consists only to a small part of our own code and to a huge part of some artifacts (e. g. Maven artifacts). Nobody bothers to keep track of the whole dependency tree, which version of what went into the build. You take the latest at the moment and that runs into production and if there is a bug, then it gets fixed, and the current version runs into production. When you had real releases, there was a requirement that you could reproduce the build from three months ago, otherwise you couldn't put a hotfix into production. Today, you make new releases so often that the need to reproduce an old build is no longer there.
Negative	I often see the problem that the process is not really agile (at best it is semi-agile). This hinders the implementation of more advanced CSE elements, such as continuous delivery.
Negative	Right now this is so far in the future for us so we have not really considered continuous deployment. It is still far away from being actually on the level that we can deliver a releasable quality level of code with that frequency. And also I would say the customers are not interested in getting this right now. It needs both: We need to prove that quality is not an issue and customers should not be concerned about doing the upgrades but we also need to provide more efficient ways for the customers to upgrade. Today, a software upgrade for our customers costs an awful amount of money since they might need to do system shutdowns and things like that. It is something they don't want to do. But of course, the general idea for us would be very appealing if we were able to continuously get our latest software out on the customer side without any risks for the end-users. Because it is a big cost for us also related to the fact that we need to maintain all our old software releases. In general, we have some 10 to 15 years of support-responsibility on a version, which we have released. It can become costly to correct things in very old software.
	CSE Category: Quality
Positive	In terms of automated tests, we have quite a good range of tools, i. e. also a choice of tools that can be used for different application scenarios.
Neutral	For automated tests, we use the standard tools that are available in the Java environment and also in front-end development with Angular 2. That means JUnit, JMeter, and Protractor, the counterpart to Angular 2: also a JUnit framework together with static code analysis like FindBug and JSLint and ESLint.
Neutral	For automatic tests of user interfaces we use Selenium. When we test backends, we often use JMeter or Gatling.
Neutral	We rarely do pull requests, but that's on the rise right now.
Neutral	We have hardly any audits, but we will soon have one.
Negative	Jenkins is not a good build server from my point of view but is widely used. The problem with Jenkins is that it was originally a hobby project by a Sun developer. It is unstable and does not scale. For example, it is very difficult for large companies with several 100 developers to run a large build farm. It doesn't scale reasonably.
Negative	We would like to do more on automated front-end tests that compare images. But we haven't yet found the right tool for this that works well.

Continued on next page

Type	Practitioner's Description of Experience
	CSE Category: Code
Positive	Version control, branching, code coverage, ... that are all very important. And it's all fully integrated with us as well. No one struggles with it, has problems with it, or doesn't like it. That's not the case at all. Everybody loves Git and uses it. So very positive experiences with it. We've gone through quite a bit as well, moving from SVN to Mercurial and then to Git. The changeover cost the developers a lot to use branching strategies. But as I said, this is now standard and we could never do without it.
Positive	An integration of the version control system, the build management system, and the issue tracking system is very useful and forms the *golden triangle*.
Neutral	We don't have code coverage explicitly, every developer can do that in their development environment, but it is not a statistic that is published or can be collected somewhere.
Neutral	Anyone can create and merge branches as they wish. That runs on a certain basis of trust. For example, we're not allowed to *rebase* because you can break other developers' work, but it's only excluded on paper. If you go for it, you can also break Git, so that's trust-based.
Neutral	Currently, there is the Jira ticket with the status *to review* and someone judges that the code fits and *gives green light*. Then you create a pull request and *merge* immediately. So you don't have to use the command line to merge the branch. Pull requests are currently only there to make merging easier. Once or twice we've had another team make changes to our code, and then there were regular pull requests where the other teams had committed something to a branch and then submitted it to us for review via a pull request, and that worked pretty well.
Negative	In interaction with external service providers, it is difficult to exchange files. That's where we have a problem: exchanging files and merging branches - these are all problems that require a lot of time and resources.

Table B.3.: Anonymized examples of practitioners' future plans per CSE elements (RQ1.4).

CSE Element(s)	Practitioner's Description of Future Plan
Proactive customers	If I understand this correctly, the customer provides proactive feedback. That is something we would still like to have in our process.
Logging and monitoring	In terms of monitoring metrics, etc., we are also in a weak position because we have put a lot of focus on features. In the process, that fell off for quite a while. Now we have the problem that things have become very slow and we don't know what the reason is. When something crashes, you restart a Docker container (we are in the cloud today, after all) and then everything runs again. For that, we use Kubernetes. For logging, we use an ELK stack (Elasticsearch, Logstash, Graylog, and Kibana). I am only moderately happy with the stack. For performance metrics (speeds, latency, how many requests fail, etc.) we are in a relatively poor position. At least we have some in Kibana. That means we don't have to pick it out of the log. You can click together statistics with Kibana, but that is also not quite usable. Monitoring is a topic that I don't enjoy that much but is important, I've learned that the hard way now.
Convenient setup	We may have containers, and we automatically install our software in them, and it then runs there automatically. We can certainly improve in this area as well.
Staging environments, continuous delivery, regular builds	There are concrete plans, yes. Certainly, we want to extend the builds ... we already have a relatively complex build pipeline that we used and that we make available to the projects, with the respective complete environments. On the other hand, we also plan to extend the builds to more platforms. Let's put it this way, while we are already relatively advanced in the Java world, we are not yet as well-positioned as we would like to be in the Node/JavaScript world, which is also becoming more and more important, and we have to generate one build per project manually. That is certainly another point where we say we would like to offer ready-made pipelines.

Continued on next page

CSE Element(s)	Practitioner's Description of Future Plan
Capturing decisions and rationale	In some places, we could document more in what way we do what and why. That you know why something was done and can reflect that again.
Continuous integration of work	I think the way it works for us, it's always step by step. So little things, until recently we had to trigger ourselves for almost everything in Jenkins to start a build. Now it grabs the tags, and if there's an appropriate tag, then it does the build itself. But these are always such small points where something is done, but it's not like: now we take care of our Continuous Integration and do this perfectly. It's always that in the day-to-day development, as far as it's kind of noticeable that something is clumsy, and you could do it better, then it's usually improved. That's how it was in this case. Because there were just too many things there: I did a lot of pull requests, and you always had to do a lot of things until it was finished. Then they say, you've done so much by hand, let's automate it. And then that improves it. I guess it's the same with other points. As soon as you realize that something needs to be improved, and someone knows how to improve it, then it will probably be implemented.
Continuous delivery	We will probably do more in the context of Continuous Delivery. We are on the way to the cloud. And you have the ability on AWS to deploy code pipelines in an even more automated way. Right now, there are still manual steps involved when we deploy it locally to our server. But that's a lot fewer manual steps than if you still have to manually create the jar file or the library and then copy it to the right place and launch it. It's not like that, but it's still 5 to 6 clicks that you have to do. You can still optimize a few clicks away from that.
Continuous deployment of releases	In the backend, we want to have more automated deployment, more tests in certain environments that give us so much security that we can also say we press the button because if the tests go through without a problem, you can also deploy it without a problem. We have that with new products, but of course, we still have old products where we haven't transferred that knowledge, that experience, yet.
Automated tests	Regarding integration tests, I am someone who likes to push this further and further, e. g. not only testing with mocks, but also with the database. The bigger you test with something else, i. e. the bigger the integration scope, the more expensive it becomes.
Automated tests	Some things don't yet work in this quality. What specifically comes to mind is testing, i. e. automated testing; that only existed for parts, and it should work for everything.
Code coverage, automated tests	I would still like to see greater test coverage in depth. Actually, we haven't done it cleverly enough. In the beginning, a monolith was built and then split into microservices only after a few months. This means that a microservice may still need information from another microservice when it processes a request that comes from the browser. Then that microservice and two or three others start sending requests. That is, there is a constant dependency on other microservices at runtime. This is a problem if somewhere, someone changes something. That's what you try to find with automated tests, I would like it better if we would notice such problems even earlier. If you change microservices and then call the other from one, then it doesn't work anymore, because you get a different answer back than expected, for example. That's where we come to your actual topic, knowledge management, we're a little bit bad at that. Hopefully, you get the consequences of the change, but sometimes you don't. There have also been cases where the testers didn't notice that.
Automated tests	Yes, of course, we are constantly looking at our processes and seeing where we can still improve them. I would say that we are relatively well-positioned in terms of continuous integration. Where we maybe don't have a direct focus yet is on code quality. We do have tools like Sonar where we test the code, but ultimately, I'll say, nothing happens yet if there are a lot of errors or a lot of observation there. That runs with it for now, but there's no consequence to it for now. For example, that one would say, we have to fulfill certain metrics or not.

B.1.2. Statements regarding As-is State of Rationale Management during CSE in Industry

Table B.4.: Anonymized examples of decisions that practitioners capture (RQ2.1).

Type	Practitioner's Statement regarding the Types of Decisions Captured
Existence decisions (requirements, architecture)	We capture two types of decision knowledge and have one way of capturing decision knowledge: in a wiki. There, we separate between *technical architecture decisions* that are customer-specific, but which we make to be able to develop further, and—when we do individual customer projects—the *understanding receipts* as to why we have agreed on certain things with the customer. That goes from the interface description that we negotiated with the customer to how we configured the client for the individual customer to get things the way they want them. It's recording the chronological order of when any technical activities happen. Those are the two topics where we record decisions. Sometimes this is rather bad than good. But it justifies why we/the customer decided the way we did.
Existence decisions, meta-decisions	We document decisions about features we implement in the Jira tickets. That works quite well. What is needed? You record queries there, for example. It is also important to record meta-decisions, as these affect the project much more than decisions made during development on a feature branch.
Existence decisions (features, architecture), non-existence decisions (alternative feature implementation), property decisions (code convention)	We have discussions in pull requests about coding conventions (e. g., whether variables are named correctly), which means we do our code review in them and the need for a separate code review tool is eliminated. We discuss code, we discuss how to implement the feature, whether there are bugs, test coverage, and test issues. Partly, we also discuss content-related things, i. e. whether we want to implement the feature this way or whether we would prefer to go a different way. The idea is that as soon as you work on something, you create a feature branch and you create a pull request. The build server builds that in the background and you can see whatever it is, it's still building. That's a good basis to discuss and that way you can also add nice bites without affecting people who are working on other branches.
Existence decisions (architecture, implementation, technical)	Implementation decisions and technical decisions. For example, if there is a feature where something is calculated, this technicality often has a certain fuzziness (you can go right or left). Then the business experts make the decision, for example, to do the calculation in a certain way. This is documented and must be documented because this is professionalism and forms the basis of the software. It has to be documented exactly why the subject matter experts chose exactly that way. If we then make decisions during implementation, such as architectural decisions, we also document that.
Executive decisions	We have decisions on the definition of done, for example. When does the team say that a piece of code is ready for production? That is quite well documented.
Executive decisions	Of course, important decisions relevant to the budget are recorded, and there are also decisions in the Steering Committee, especially in the case of large sums. You can't write everything down. At some point, you shake hands, look each other in the eye and say "that's how we do it". In my opinion, that's 90 % of the decisions. The number is probably not correct, but one thing is certain: most decisions are made and adhered to uncodified. And the Scrum team further evolves this knowledge. Important, cross-cutting constraints are, of course, recorded.
Executive decisions, existence decisions (architecture)	However, some decisions are made outside the project team, i. e., that come from above. For example, that the infrastructure is too expensive and has to be handed over, or that external employees have to leave the project for cost reasons. It also happens that management has already made technical decisions. The team had problems with that. So, for example, it happened that the management said that Java EE and a certain database would be used. It took the team a long time to get the management to revise these decisions, using guerrilla tactics from below.

Continued on next page

Type	Practitioner's Statement regarding the Types of Decisions Captured
Executive decisions	In what we do for AWS, the Chief Technology Officer says, "We want continuous delivery in the cloud." Everything after that is up to the teams (what we do, what frameworks are used, that's all in team authority).
Existence decisions (requirements, implementation, tests), executive decisions (decisions from management level), non-existence decision (several implementation options)	In daily discussions, we have always made decisions about different activities and topics. In our projects, we have always distinguished between information types such as requirements, developer to-dos, tester to-dos, and error messages. During the creation and processing of these information types, questions always arose where decisions had to be made. With the requirement topics, typical decisions are first of all, which requirements one has at all or how important these requirements are, and which requirement one can do without at the beginning. So the prioritization activity contains many decisions. Further complexity considerations of requirements contain decisions, thus as how complex we evaluate which requirement. This complexity assessment is related to the effort estimation of the developers and decisions come into play there as well. For a requirement, there can be several implementation options that differ in the effort estimation, quality, maintainability, or scaling. Of course, a distinction must be made here as to which implementation option is chosen, which quality characteristics are more important, and how much can be invested in each implementation. Depending on the requirements and the implementation options of the developers, decisions have to be made. Decisions have to be made in testing activities, especially when doing risk assessments. We didn't always test everything due to time constraints, but took a risk-based approach and tested the most important things. We had to make decisions about which test cases were the most important, which test cases to execute after which sprint cycle, which means that planning the test activities also involved decisions. Who tests, what are the test resources, how much time do we take for the test if there is a roll-out and the developers are possibly not yet ready, and how do we possibly redesign the planned test phase. That means there are also many decisions that are made in particular from the management level. After the test executions, we received error messages again and again, and we had to decide how important these error messages are, how we deal with these error messages, how critical are they, do we want to fix them immediately, and whether we can postpone the bug fixing to the future sprint.
Existence decisions (architecture), property decisions (non-functional properties), executive decisions	There are a lot of architecture decisions in there, of course. But some decisions affect the way it's used, so—sounds funny when I say it like that, but—decisions made by the user experience team. They offer that to the outside world: What do we document? When do we make changes? These are also decisions that affect not only architectural changes, but rather ... partly functional, or also non-functional properties that then affect the experience of the entire environment.
Property decisions, executive decisions	Code style guides, which describe recommendations or constraints on how to record changes, when to create repositories, ...
Property decisions regarding high availability and consistency	We have demand for high availability. For example, if a data center fails in China, the hope is that a European or American data center will compensate. This may be a little slower, but it will continue. If you want to implement this in software, you no longer have many things that you would otherwise have, e. g. the abstraction of the transaction into the database does not work then. This is the famous theorem on *consistency*, *availability* and *partition tolerance*. Of those three things, you can only ever get two. Software development has always held consistency high. And we then turned *partition tolerance* down. That means there is only one database server or if there are several database servers they have to be able to talk to each other. Now we suddenly took another database, the non-relational database Cassandra. For us as developers, this means a completely different approach to our software development. Consistency is suddenly no longer a given, how do we deal with that?

Table B.5.: Anonymized examples of where practitioners capture the decisions (in which documentation locations), with which techniques and tools (RQ2.1).

Location/Tool	**Practitioner's Statement regarding the Locations, Techniques, and Tools**
External document (word), wiki, issue tracking system	Decisions that were made once were documented in Word documents and archived in folders. Later, during the implementation of such decisions, e. g. when setting up the infrastructure for continuous integration, selecting the tools, and deciding how to work with these tools, such as the git workflow, how should the developers deal with the git management system during development (how should they define their branches, how should they merge the branches), we documented such fundamental decisions in a central wiki. For the developer decisions, we recorded them in the tickets, so the developers documented their implementation proposals in the tickets and the business department had to go through those proposals and set the ticket statuses accordingly, thereby documenting that they had read those implementation options and decided on one and why they decided on it. In this context, we worked with the tool "Redmine". But in other contexts, we also use Confluence and Jira, which are very comparable. There, we stored such decisions of all project participants in an accessible way, so that one could always read up on them.
Wiki, issue tracking system	To record decisions, we have a wiki (Confluence) where we have recorded major decisions, also so that later colleagues can read in there. We used Jira and documented the relevant steps and decisions for each story so that we could see in the future why we did what we did and what the reasons were. Otherwise, people have the knowledge and there is still the truth in the code.
Wiki, issue tracking system	The technical decisions are often discussed in Jira tickets. The stories are documented there and then the decisions are also documented there. With other customers, I have also seen that Confluence is used for documentation and that this is tracked in Jira. In the end, it doesn't matter, the main thing is that it's documented somehow and you can find it again.
Wiki	In some cases, decisions are documented in Confluence, where the basis for the decision is listed, i. e. what was the basis, what knowledge was available and what criteria were considered, what decisions were made and why these decisions were made. What are we doing for what reason? That came into the project over time. In between, there was some sloppiness, especially at times when things were hectic and decisions had to be made quickly. But then you also noticed that quickly because you could no longer understand things.
External document, issue tracking system, email	They are recorded in meeting minutes if they are big decisions. I think you have to differentiate, and I can't speak for everybody, but there is a tool, clearly the ticketing tool for the features, but other decisions, whether you're changing build plans or the pipeline somehow, that was only recorded in meeting minutes, email, or via personal contact, or in PowerPoints.
External document	When we go strategically in this or that direction, we discuss pros and cons in Google Docs. Everyone contributes. And then we record a decision below. Bottom line: we do XY, so decisions like that. Smaller things, the color of a button, or we do a feature a little bit differently after all, we make this decision rather ad-hoc and don't justify it in writing. We then record more the decision, less the reason for it.

Continued on next page

Location/Tool	Practitioner's Statement regarding the Locations, Techniques, and Tools
Issue tracking system, pull requests	We discuss a lot when we do our sprint planning, e. g. to better estimate a story and to know what happens behind it. Then we record that we first have to make a decision that we will do something "this way or that way" and think about what the impact is behind it. To not forget things, we create a story. We already include decisions there. Otherwise, there are discussions in pull requests. I'm still trying to push that a little bit. When we work with feature branches, we also work with pull requests. The idea is that even if we are not yet finished with the feature, we create the pull request relatively quickly so that the documentation can happen in it. From a developer's point of view, you are relatively close to the source code and you can discuss whether something is done well or badly. This is currently the place where most of our decisions are recorded or where you can see how something was done and for what reasons. Then, of course, we have the documentation in the wiki. I didn't use that in this project, but I did in other projects where I was a software architect. There was a document in there that documented the architecture decisions. Architecture decisions are very important because they are hard to change. For small things, I think pull requests are just the place to do it. It's close enough to the code, developers read it, and it's easy to discuss.
Issue tracking system with tag/label	This is done via Jira. In it, there is a tag that is used for certain tickets. The tag is called *decision template*. The tickets with this tag are collected and the responsible persons (usually the product owners) have to make the decisions. It is possible to filter for that tag and the product owner will see that there is still a decision to be made before anything can be done at that point.
Commit in git	Whereas I have to say, so far you try to record in every commit message that you make, for what reasons something was changed. So it's not that you don't have some form of tracing. What you don't have with the commit messages in this sense is that it's somehow structured—that's just a bit of text. For simple things, I've found it okay so far to do that via commits. But of course, if you can also form real chains here, so to speak, that you say that decision follows the other, then I could imagine, especially in more complex projects, that that could be helpful.
External document, issue tracking system	We have a steering committee meeting every three weeks where we have about 20 PowerPoint slides that are then archived. When it comes to programming, we move again in the direction of Jira, or another similar tool.
Chat tool	We also use chat tools for remote programming. Right now, Slack is popular for that. It's very dangerous. If you're not in the project for a day, there's a discussion in Slack and a decision is made there. Then there are 20 channels where there is an incredible amount of discussion, and the question is whether you read it all or not. Personally, I try to keep the slogan: "If it happens in Slack it didn't happen. If you make a decision, you have to send an email." Slack is disadvantageous when it comes to documenting things for the long term. It's all kind of there, and depending on what you paid, you can still get to the old content for a certain number of weeks, but the unimportant is next to the important content. Everything is jumbled. Sometimes there is a separation into channels, but that doesn't work well and you can't find anything. These chat tools are a blessing for communication but a curse for documentation from my point of view.
ReadMe file	In some cases, we have a ReadMe file in appropriate places, or some other kind of - let's say, if we deviate from specifications - document, then we document that in-place with a file that is there, in the repository, where the software is.

Table B.6.: Anonymized examples of how practitioners link decisions and related decision knowledge to other software artifacts (RQ2.1).

Technique	**Practitioner's Statement regarding Linked Artifacts**
issue tracking system functionality	In the ticket, it is stated which component this ticket concerns and accordingly a decision can be related to a component.
No links between model diagrams and decisions	As such, there are no real links. You can add images, of course, but the images are not linked to the actual diagram models. We use the Enterprise Architect to capture component diagrams, sequence diagrams, and domain models. But we don't have any linkage between the decisions in Confluence and the Enterprise Architect. This is done by copying the image from the Enterprise Architect and adding it to the wiki so that at least there you get the entry point where I can then look in the model repository.
Capturing decisions as close to the code for consistency	The things that are at the end in the product and the things the developers think are important to capture are continuously being updated and kept in the internal product documentation. There are important things that we definitely want to update in the documentation. Otherwise, the general thing is in the code that we comment. We have the information in the code. That's kind of my experience through the years that it makes sense to document the object model and all architectural thinking and so on but if you get into too much detail in your internal technical documentation you always end up being late and not being able to keep this updated with what is actually in the code. My personal thinking is that we don't have to bother too much about it and just accept the fact that we never will be able to keep such a document updated. Instead, we should put the effort into documenting the necessary things as close to the code as possible.

Table B.7.: Anonymized examples of how practitioners preserve the evolutionary history of decisions (RQ2.1).

Technique	**Practitioner's Statement regarding the Evolutionary History of Decisions**
Special technique to mark an obsolete/a rejected decision	We decided to do something that has not happened before, however: if we were to revise the technical decisions, we would mark the old decision that it is obsolete and create a new decision. We would link the old and new decisions together, but not overwrite them. By only starting with the project, we have already made and documented a few decisions, but have not yet done any rescheduling of existing decisions.
History function of issue tracking system	If this is done within Jira, the history is stored. For example, when a ticket was opened or closed.
History function of issue tracking system	Also decision changes (what changed when) could be tracked in Redmine's edit or history mode.
No dedicated technique to track history	During meetings in that decisions are made (e. g. developer workshops), the decisions are recorded in a protocol. But I don't know how that works with the *decision template* in Jira.

Table B.8.: Anonymized examples of when and how often the practitioners capture decisions (RQ2.1).

Practice/Frequency	**Practitioner's Statement regarding the Capturing Practices and Frequencies**
On demand	That is a sore point. We only document the most important decisions where we notice in the discussion that we have different opinions. We do that to record the pro- and con arguments that the stakeholders express, document the decisions, and communicate clearly and specifically how to proceed. So we only document decisions on controversial issues.
On demand	It's difficult to say exactly how and when to make decisions. On the one hand, there is long-term corporate planning, where you already ask yourself many questions. Some decisions are suddenly made from one moment to the next. So you don't have a fixed point. We have the project going, and then it's mostly on demand; depending on what's being touched at the moment, we look at and collect things and divide that up in terms of time. This also means we have to see where we get the money for it; that's also a question.
Weekly	I would say weekly. It depends on how many new tickets come in. We don't just have one Confluence page; we have many tickets. For the management issues, there are daily changes in Confluence. For our team, however, it is only relevant weekly or even less frequently.
Sprint planning	We discuss a lot when we do our sprint planning, e. g., to estimate a story better and to know what happens behind it.
Daily – weekly	At peak times, this is sometimes done several times a day or even several times an hour.
On demand via issue tracking system tag/label/keyword	1) A person (e. g., a developer) notices that s/he needs a decision and creates a ticket with the tag/keyword *decision template* or adds the keyword to an already existing ticket. 2) In Confluence, a page with a built-in Jira widget displays all current tickets with the keyword *decision template*. However, there is no other automatic notification. The person has to take action themselves and notify decision makers (e. g., via the mention function in Jira comments or by mail). 3) A second person (e. g., product owner) takes one of the *decision template* tickets and will make a decision if necessary. 4) This decision is then communicated somehow. In what way is not exactly specified, but the Jira comment function has been used here for the most part so far. 5) The keyword/tag/label *decision template* is removed and the keyword *decision* is added. This way, searching for all decisions made later will still be possible. 6) On the same Confluence page, another Jira widget shows all *decision* tickets and accordingly provides an overview of all decisions made.

Table B.9.: Anonymized examples of why practitioners capture decisions (benefits) and how they exploit the decision knowledge documentation (RQ2.1).

Benefit	**Practitioner's Statement regarding Benefits and Exploitation**
Decision-making support	We have started to work out a proposal in our team on how to deal with this. It contains concrete problems or restrictions and suggestions on how we can live with them. When it was finished, three colleagues who were particularly interested in the topic first looked at it. Then we formally said: "The following is up for decision, this is the current proposal. Do you all agree? If not, we need to discuss further." Because I had involved the people who were most interested in it before, that went through and so we made a formal decision. That was an example where we went very cleanly.
Decision-making support	As a basis for discussion in any case, yes. Especially when you discuss relatively rough architectures, then this is very good.

Continued on next page

Benefit	Practitioner's Statement regarding Benefits and Exploitation
Knowledge sharing, accountability	I don't know how important it is to preserve history, but having a rationale for the current state is important. You often ask yourself in the code "what's going on here and why something was done this way" and it would be good to say "ah, ok, here the customer said they absolutely need the feature and then it was done this way".
Knowledge sharing, reuse (don't have to do the work twice), accountability	I would definitely say it's moving us forward. It's not very formalized. It's more of an ad-hoc approach, we realize that we've discussed something bigger, then we try to record that as well. So that we don't have to do the work twice. These are often very detailed decisions that, if they are not written down somewhere, have to be derived from somewhere, which then takes some time. You can simply look back once again: "Why didn't we do that back then? Oh, that's right, that's why. . . " And also simply be able to read. "Such and such has already been discussed". "Such and such major features are planned for these reasons". Simply also to make it easier to collaborate.
Accountability, reuse	To see what happened back then. Or to check whether it made sense at the time. Yes, also to, when you need to make a new decision, look at the old ones again, that happens. But also rather very manually.
Accountability	For now, they are just documented. We are doing this because we think that at some point we will ask ourselves exactly why we decided to do something. Then we want to look into it. When it comes to the agreements with the customer to adjust interfaces or to consider whether to adjust them, we then also want to know what was agreed and what changes we have to make to the interfaces.

Table B.10.: Anonymized examples of practitioners' rating of the sentence *The explicit capturing of decisions benefits our software development process* (RQ2.1).

Likert Choice	Practitioner's Statement regarding their Current Decision Capturing
Agree	I would say it's not a hundred percent true because it's not currently used by every person and it's not completely transparent what happens with the decisions.
Disagree	I think we don't utilize them today on the level it would be good if we did. I think we could benefit more from them. We address the individual decisions that we make during reviews and within the project but we do not utilize that for general learnings and process improvements in a good way. I would say we are right now on a -1 but if we would utilize the information better we could be +1 or +2 on that.
Neutral	I would place this project right in the middle, i. e. neutral. The reason is that a lot is documented (including many decisions), but that this is used far too little.
Agree	The decisions that are recorded are clear to the person recording them anyway. That person then does it the same way. If he hadn't recorded it, he would still do it the same way. It would not change much. Afterward, no one looks at these Jira issues anymore. When something is done, it is documented somehow. So you could look again, but what's the point? I do think that we benefit from the things that we document in Confluence. That's an important resource. But I benefit more from the auxiliary material than the decisions.

Table B.11.: Anonymized examples of decisions that practitioners do not capture (RQ2.2).

Type	**Practitioner's Statement regarding the Types of Decisions not Captured**
Executive decisions	With newer decisions, e. g. regarding the build pipeline and deployment, you don't document it, you just do it because it has to be done very quickly because the developers are waiting. Afterward, you don't get to the documentation because something else happens that you have to react to quickly. That also "catches up with you again". Last week, I had to dig out an email that was a year and a half old, where we had discussed how to technically set something on servers and made a decision. That wasn't documented at all, and that wasn't considered when a change was made on the server, which is why something ended up not running. So I had to dig out those emails where it said how we decided that at the time. It would have been easier if I had known that it was on a particular Confluence page, for example.
Executive decisions	On the pipeline; here we capture too little knowledge, almost no knowledge. But from my point of view, we should.
Existence decisions	Primarily, these are former decisions. It can be architectural things, design decisions, or smaller things that make the product slowly turn in a specific direction. When you sum up any of those, the effect could be potentially significant.
Existence decisions, non-existence decisions/bans	Smaller things, the color of a button, or we do a feature a little bit differently after all, we make this decision rather ad-hoc and don't justify it in writing. We then record more the decision, less the reason for it.
Existence decisions, non-existence decisions/bans	In concrete terms, it's often smaller user-interface and functional things. In a review of a feature, you simply sit down with a colleague, do a code review or feature review, and then define changes and create issues for it. But there is no rationale behind it. Not: Why did we do it this way? But simply: We do it this way, maybe a little differently, the button a little nicer, and then it goes on. These are rather the things that happen ad-hoc. That is written down in the issue tracking system.
Existence decisions	Swagger is software to create and partially document our interfaces. But meta-information, which service is responsible for which things, that's kind of known, but not documented. A very simple architectural question like "When do I create a new microservice and when do I extend an existing one?" is also answered by intuition, and there are no clear criteria for this. Some things work, others don't.
Existence decisions	For example, a team decided to do a new microservice in a completely different way, using Spring Boot. (Spring Boot is a standard framework). The decision to do this with Spring-Boot was made incrementally, and not properly documented (except, of course, by the fact that the thing exists).
Property decisions	An important decision is also whether microservices work synchronously or asynchronously. Asynchronous is a bit more performant, asynchronous is much easier to program. That's just what happened, these are decisions that someone makes the moment they code something for the first time. Then everybody else imitates it and it's not really discussed. Something like that in that form unfortunately happens several times. That's because each microservice can do it a little bit differently. That's a freedom that microservices have brought. You actually want that, too. But this freedom also invites a bit of abuse. Everyone does what they feel like doing at the moment. For my taste, there is a bit too little consistency in this project and especially too few formal decisions and documented decisions in this project. "Why are we doing it this way? We do it this way here, we do it differently there. What's the plan?" We're relatively weak on that kind of thing. At the moment it's not about the day-to-day business, but more about the strategic issues, the longer-term issues like architecture issues, we're missing something. You could do that with Confluence, you would need more meetings. We are a distributed team, when we get together time is already tight because of all the Scrum formalities. There is not much time left for additional discussions.
Existence decisions (configuration)	For example, we would have to record our decisions regarding the compiler version numbers used as well.

Continued on next page

Type	Practitioner's Statement regarding the Types of Decisions not Captured
Existence decisions (tests)	Sometimes we have had situations, e. g. for troubleshooting or execution of the most important test cases, where test cases that were documented were so abstract that the testers did not necessarily know how exactly to perform the test steps. There you made decisions over the short communication path (just table to table), we didn't document such decisions. You just did that and it then came down to whether that test case passed or failed. If it failed, we then included that decision in the defect description after all: To this test case number x we came to the following defect, most of the time we also added a screenshot so that you could later reproduce/understand why a defect occurred. We have had the awareness in the team if someone has to do rework or repeat activities, what decisions and information they will need. We always documented these.
Existence decisions	When you develop software, you often decide how to do it. Maybe which framework to use. This is not always to be found in the documentation.
Existence decisions	For example, we have decided on how we send messages to other teams from our application. We've written those down for our team and they're clear to our team and we've also defined them in some Jira ticket in a schema, but we haven't captured them in a Confluence interface definition page, so the other team that's trying to consume the messages is still standing there saying, "We don't know about that now. We need something here." And we say, "How, you guys have everything." But then when they ask where we do realize we don't know. That happens.
Existence decisions	In terms of the microservice architecture: Up to the point where we adopt the microservices, we define very precisely where and what has to happen. But the internal nature of the microservices' data structures is not as well documented as I would like it to be to understand why certain data structures were chosen for the business functionality that the service is supposed to provide.
Executive decisions (prioritization)	This has something to do with prioritization. When one implements a change, it means that one has preferred this change to other changes that were also pending, due to a certain priority. In doing that, I've found that after a couple of weeks, the question comes up as to why you did that and not something else first. That's something that we didn't document in the decision-making process of why we were doing something. At that point, we determine that's the most important thing, but why the other thing wasn't more important and what assumptions were made there, that's not available.
Non-existence decisions/bans	Especially, we do not document why a particular option was not implemented.

Table B.12.: Anonymized examples of reasons for not capturing decisions reported by practitioners (RQ2.2).

Reason	Practitioner's Statement regarding the Reasons for Not Capturing Decisions
Overhead, exploitation (added value) unclear	Of course, it is time-consuming to document decisions that have been made. There is also the question of who reads through everything afterward. If you have to read through everything again before you can develop it yourself, there is a lot of overhead. On the other hand, it is often the case that people think, "It's logical, everyone else would probably have made the same decision." In other words, you don't even think about the fact that you're making a decision, and that's why you don't think about documenting it.
Overhead, exploitation (added value) unclear	It is once due to the time. But it is also due to the fact that developers prefer to develop microservices experimentally, i. e. to try out different data structures and use them for their use case, and that it no longer offers any added value for them to document this afterwards once the data structure that is best suited for this has been found.

Continued on next page

Reason	Practitioner's Statement regarding the Reasons for Not Capturing Decisions
Overhead	Usually, people say that it's a problem of time and money. Which is sometimes a bit of an excuse. But it is always an effort. And people always shy away from that and prefer to do something else first.
Overhead	I'm not a big proponent of codified knowledge management because we all don't have time to look at it. And secondly, I've never seen it work.
Immature process	Many projects do not even get into the situation of documenting decisions or being able to document and operate knowledge management, because the development processes do not correspond to what would be necessary, as many approach it with a semi-agile approach and thus often have problems.
Exploitation (added value) unclear	It's often smaller user-interface and functional things. In a review of a feature, you simply sit down with a colleague, do a code review or feature review, and then define changes and create issues for it. But there is no rationale behind it. Not: Why did we do it this way? But simply: We do it this way, maybe a little differently, the button a little nicer, and then it goes on. These are rather the things that really happen ad-hoc. That is written down in the issue tracking system.
Exploitation (added value) unclear	Because everything runs in a very agile manner. There is a new task, then someone takes care of it, implements it, then it is briefly checked, and then it is checked off and in. But it is not recorded for a longer period of time. Probably because there is no need, or because the need occurs so rarely that I can then simply think to myself in case of doubt: "Why did I do it that way back then?".
Overhead (asking colleagues as an alternative knowledge source is easier)	There are often retrospectives. We have a lot that we don't write down directly, but have in the process itself. You can just always talk to people a lot, and ask about it. This probably has nothing to do with it at first, but: If the knowledge is in people's heads, then it's very easy for us to ask them. I don't have to wait two weeks for an appointment, I can just go and ask. Of course, that is also the reason why this is not documented, because asking is simply easy. Then there is this situation that people you want to ask are no longer there. And then you regret it.
Overhead, exploitation (added value) unclear	The added value is sometimes not clear and it behaves like the 80/20 principle: to document the remaining 20 %, you would have to put in 80 % more. The benefit-cost ratio would then not be reasonable.
Overhead, lack of appropriate techniques or tools	You have different toolings for different things. That causes annoyance. For one it's IBM for requirements, for others it's a different system. Then you're doing work twice. That annoys the developer.
Rapid changing decisions that lead to outdated decisions, i. e. to inconsistency	Probably because parts of it change quickly, and then you have the problem that they are no longer up-to-date. And then you have another document that contains something that might have been true for an outdated version. That almost goes down to the code level. The problem is that we have a relatively large number of small, individual modules, some of which use data that comes from some test system, because it's much faster than mocking everything up, and the access to this test system alone changes-not so often, but it has changed before-so that some modules no longer work with the test system, because the test system is already newer than the module. This means that even at the code level, this problem arises that the systems sometimes develop so quickly that not even the test code in the individual modules keeps up. So module functionality of course, but the test code in the module does not work. And if it's just little things like that the test user is called differently, and that's not in all 20 individual modules already updated. Since it doesn't work there, I can very well imagine that if there's a text file lying around somewhere that says something, that it's 100 % wrong.

Table B.13.: Anonymized examples for potential benefits if practitioners captured the decisions that they currently do not capture (RQ2.2).

Potential Benefit	**Practitioner's Statement regarding Potential Benefits if Captured**
Accountability, knowledge sharing, decision-making support, reuse	Advantages would be that a person afterward, who works in the same work environment, does not wonder why the system is set up that way. If this person has a different opinion, then discussions arise as to why it could not be done differently. But if you have a reasoned decision, discussions can still arise, but then that person doesn't wonder. And one can profit from it in similar cases. That means, for example, if you have done something on one server, then you don't have to go through the whole process again on another server, weighing up the pros and cons, but can have results for the second server more quickly based on the decisions and arguments already made.
Accountability, knowledge sharing	Either in that case the team would find the decisions regarding the interfaces directly, which would reduce the overhead because they wouldn't have to ask us in the first place. If they did have to ask us, we could explain it to them better and wouldn't have to pick something out ourselves first and then also translate it from Jira to Confluence, but say, "Here it is, read it through and then if you have any questions, comment below it, then it's documented again."
Knowledge sharing, reuse	Some things happen now and then that are not documented. You notice that pretty quickly, too, that you stumble across it. Especially when you get used to the fact that all knowledge is documented in some way, and then you think that some issue is open and then it's not open because three people thought about it but didn't write it down.
Decision-making support, accountability, knowledge sharing	Continuous learning is very much part of that for me: Understanding exactly what they want to build and recording the decisions that are made exactly through the trade-offs and compromises that you find: what was decided, why was it decided, so that you can understand after the fact why it happened the way it did.
Knowledge sharing (regarding implementation decisions)	If it goes down to code level, that would be partially great, because then you would know why anything is done that way. I had code but didn't know exactly what it was for. I asked, and someone was able to tell me exactly why it was the way it was. But the bottom line is that it wasn't clear to me right away, and asking is then of course an additional expense.
Knowledge sharing (regarding ban decisions)	It is certainly not wrong to read up on why the decision was made at the time. Sometimes you look up in the history what you once developed and why you left it. It would be interesting to ask: why did you end it?
Knowledge sharing (to support disaster recovery)	I think the most important point in favor of capturing the knowledge is in the direction of disaster recovery. I mean, there are maybe 2-3 people who know how to reproduce the previous state, but with such a size, of course, it is already critical.

Table B.14.: Anonymized examples of practitioners' rating of the sentence *The explicit capturing of decisions would benefit our software development process* (RQ2.2).

Likert Choice	**Practitioner's Statement regarding more Decision Capturing**
Strongly agree	I think it really hurts that decisions happen but are not made properly. The moment you document them, you also force them to be made. If the other person doesn't agree, there are discussions. It would also be good to know what is already decided and what is not. That would be beneficial.
Strongly agree	Because there are so many decisions that are made and there are certainly some that are also dependent on each other or that are made twice. I think that makes sense, in any case, to save time afterward and to keep the overview.

Continued on next page

Likert Choice	Practitioner's Statement regarding more Decision Capturing
Disagree	We already have a high level of decision documentation and to a certain extent the documentation is not even used.
Strongly agree	Advantage for whom? You have to break down the development process: Who has which role? Who wants to do what with the knowledge? If you look at the overall success of the project, I totally agree, it's very interesting for that. Whether it's interesting to an individual developer has a different meaning than it does to a project coordinator or leader, or someone doing the integration. Tooling to understand decisions is definitely an advantage.
Disagree	I'm on thin ice here, because I don't program myself. But I don't believe in it myself, because: yes, in a waterfall method, in which nothing changes, in which the market is stable, and in which processes are stable—sure, that fits! But the development frameworks change relatively quickly, the market changes quickly, why should I look up how the decision was 6 months or 2 years ago, when the market is different again anyway? That doesn't do any good.
Agree	The problem: Should I take into account that this is extra work, which in many cases is, of course, negative or costs time? Even though it's probably better in the long run, in the short run it always has this initial problem of "okay, I need to use these systems to keep the decisions, I need to maintain them." Then the standard problem is that if I don't maintain the systems properly, then I have something in the system that doesn't match the code.
Neutral	However, the more agile the team becomes, the less I consider documentation. The increased feedback loop makes it more difficult, and the time for documentation is simply no longer available. Perhaps some of the efficiency is also lost if one were to invest more time in documentation, because tomorrow the decision-making situation can already look different again.

Table B.15.: Anonymized examples for knowledge sources from which practitioners retrieve necessary information for decisions that are not captured (RQ2.3).

Source	Practitioner's Statement regarding Alternative Knowledge Sources
Reverse engineering (of code)	As the saying goes, "The truth is in the code." That means we then look in there. Of course, that's not always so helpful. As I said, we are also trying to get better at this point. We have now also set up Jira and Confluence so that such decisions can perhaps be documented there. But that is still a process that is just getting started.
Reverse engineering, asking colleagues	My standard approach: 10 minutes or, let's say, a short time I look myself, then I ask any people. Some things I find out relatively quickly, then I do not need to annoy people with pointless questions, but I have then also noticed that it is sometimes 100 times smarter to ask after 10 minutes briefly and then get the answer, even if I could find out after 1 hour myself, what's behind it.
Reverse engineering (of code)	I hope to find out through the code. And that's why the review is so critical.
Asking colleagues	Then the first thing is to curse because it is difficult. We may ask the person who did it what is difficult when that person is no longer there ...

Continued on next page

Source	Practitioner's Statement regarding Alternative Knowledge Sources
Asking colleagues using emergency mobile phone	This goes as far as being able to be called at 4 a.m. There is an emergency mobile phone for this purpose, and it is also relatively well documented what the responsibility of the person who has the emergency mobile phone is, what tools they have, and what standard interventions are (shutting down containers and starting them up again). There are always two who are responsible—one main responsible and one as a backup. The main responsible can call in the other if necessary, if they have problems. It is well documented who is on duty and when.
Look through emails and pull requests, asking colleagues	One has certain decisions that are made in emails. In this project, we do not have a dedicated document for such decisions. Sometimes, having something like that would be quite useful because now you have to look through emails and pull requests to see when you have done something. In a small team, this works quite well; we all sit across from each other at the same table. It's still easy to keep this common picture in mind. In a larger team, that would also be more difficult.

Table B.16.: Anonymized examples of how practitioners share knowledge to avoid knowledge vaporization (RQ2.3).

Technique	Practitioner's Statement regarding Avoidance of Knowledge Vaporization
Sharing knowledge between project members (depends on team size)	It depends on the constellation of the team. In large areas where the tasks are documented in great detail, it's a different question than in a small team where everyone does everything. If someone then leaves, that is, of course, a problem. We have departments with 3-4 developers, and there are departments with 50 developers. They are structured completely differently.
Sharing knowledge between project members	We try to distribute knowledge as much as possible on many shoulders for our area and to document a lot—not only decisions—but also the project knowledge as a whole. We do this in Confluence, where we document exactly what we build and how we build, what we expect from the other teams and the project sub-teams, what they have to deliver so that we can build, what they have to configure, and so on. We document this process so that, ultimately, if I were to "drive in front of the freeway bridge tomorrow", the team would still be able to move forward.
Sharing knowledge between project members (through pull requests and knowledge transfer sessions)	A mechanism to distribute the knowledge goes back to pull requests. We have included the check that a feature branch can only be merged if the build server says "The build and all tests were successful." That's the technical check. And we also included that you can only merge the feature branch/pull request if at least one reviewer has approved the pull request and done the code review. We are a small team, and we have determined that we always invite the whole team for every pull request we make. It's up to each person to decide whether they can contribute to the discussion. If no one responds, the person who made the pull request can contact people again and ask for a review/briefing. A lot of knowledge sharing happens on pull requests because that's where it's documented. We also do pair programming from time to time. At the end of the project, we explicitly do a few *knowledge transfer sessions* on various topics. That's the usual game when you leave a project.
Sharing knowledge between project members (wiki instead of diary)	I told the new employee that I don't like to see research diaries. It's a notebook in which you write something down in these meetings where everyone sits and writes something down. You can ask yourself what the point is. That should be written in the wiki.

Continued on next page

Technique	Practitioner's Statement regarding Avoidance of Knowledge Vaporization
Sharing knowledge between project members (offboarding process)	In general, we try to spread the knowledge. For such central decisions that are not documented and someone abruptly leaves the company overnight, then it's almost too late, then you have to try to explore afterward why something was decided that way. We just had the case that someone left the company. But the person had worked for another month before that, and they spent one to three weeks of that month informing other people about the application that they and their team had developed and simply writing down the knowledge.

Table B.17.: Anonymized examples of how practitioners identify parts of the system that are affected by new or changed decisions during CSE (RQ2.4).

Technique	Practitioner's Statement regarding Managing Changing Decisions
Change management process	There is a process for change management. We have set up a Jira system in which for every project, regardless of whether it is a new project or already in progress, you enter the change request, and then the person who has to implement it, i. e. first of all the project manager, basically has to assess whether it is covered by the contract, whether it should be done or not. If it is, then you have to go to the corresponding developer, maybe several if it's something bigger, and get cost estimates of how big the impact on the entire software system is. That's the process at a very rough level. You decide: yes or no? If yes, you have to evaluate it, how is the implementation? The best thing is to create tasks from the change request. And then process them accordingly.
No tool for change impact analysis since scaling processes is difficult	We don't do these in a dedicated way. That usually results from the project knowledge. It also depends on the project. I also know many projects that are large and so complex in terms of architecture. These would need a change impact analysis, but they don't do that either and are always quite surprised that when you "pull the thread on the right, that the left corner fidgets a bit." Many projects do not scale with their processes. These are large projects that are still carried out as if they were five- or ten-man projects.
No tool for change impact analysis but risk management	We have not done a change impact analysis systematically or methodically. Of course, we intuitively asked ourselves what this meant for us. One situation was when there was a change in a company's name. Changing the name was a decision with a serious impact. In the user interface, changing the name was fairly easy, but in all the development environments, all the code and repository names, and the continuous integration infrastructure, getting the name out was very complicated. They didn't do a change impact analysis when they made this decision. That means we didn't use a systematic impact analysis, a lot of intuitive and ad hoc decisions were made. What we already did systematically was risk management. In risk management, we always wrote down the risks at the beginning of new phases or the beginning of the development of new features or decisions. Through a brainstorming session, we identified risks, categorized them by the probability of occurrence and potential impact, and for the important risks that affected us, we came up with measures to minimize the risks. If we can't eliminate the risks, we've also thought about how to fix the problems after the fact. There, we worked with Excel spreadsheets. We always had these risks in mind, and, in particular, it helped us to put in place good risk avoidance measures. We have introduced a lot of techniques and tools, especially related to operations. These have helped us to prevent certain error situations, but also to detect them as early as possible.
No tool for change impact analysis	No, unfortunately not, I would say. That is very hard and that's a challenge.
No tool for change impact analysis	Our boss has a very good overview of the entire system. It's very easy to assess where it might interact with something else, and then if necessary, there are still questions, such as what exactly does that mean in the app, in the backend, etc. But I would say that our boss has the entire overview to be able to assess, where what happens. Yes, it happens through reflection, it's not so much a defined process.

Continued on next page

Technique	Practitioner's Statement regarding Managing Changing Decisions
No tool for change impact analysis	There is no good system model here in which you can do any sensitivity analysis or influence analysis. There isn't. That was also one of the points that went wrong from time to time, rather on a smaller scale. No, there is no tool here, and then that happened more of good luck and common sense.
No tool for change impact analysis but automated tests	Often in planning, when you sit down with everyone, the most important dependencies already come out, but in the end, you have to be honest, with the first time compiling and starting, you see what you forgot. That is, when running or testing, at the latest then you see it. There are only two things for me: Automatic tests and a good type system. Of course, that's difficult with our recent JavaScript developments, because you're completely type-less there. Since the code base is still quite small, you still know the dependencies, you have them in your head. But if the system continues to grow—as we've already noticed several times—we won't be able to avoid testing.
No tool for change impact analysis	Because we have very clear technical boundaries through our microservice architecture, we can identify relatively quickly during an analysis which initial components are affected first by this change and derive from this through the architecture of the system, which goes beyond the service boundary (i. e., which domain-oriented objects do I have, where are the domain-oriented objects, in which services), I can derive very quickly which components, i. e., which services must be adapted. The change has to be encapsulated in a microservice, and this helps to prevent the change from spreading throughout the entire system. That allows our architecture and interface design quite well to not have that propagation. But that's human work, that's not a tool. Our system is not yet so big that you can't grasp it manually, but that will probably only be a matter of time. In three years, I could imagine that we might think about it, but right now, a manual change analysis takes about three-quarters of an hour, and since it doesn't happen that often, it's not a high priority to automate it.

B.1.3. Statements regarding Ideas for Continuous Rationale Management

Table B.18.: Anonymized examples of beneficial rationale management features and additions reported by practitioners (RQ3.1).

Features	Practitioner's Statement regarding Important Features and Additions
Filtering and searching	The most important thing for me would be a pretty powerful search. That you can find what you're looking for relatively quickly. I don't know how one would imagine that currently. Somehow it has to be searchable by text. That's what I would find most difficult, but also most important.
Knowledge presentation (exploitation, reflection), documentation of alternatives	The second thing would be to see exactly which considerations were made, but also to see what the alternatives were and why variant B and not variant A was chosen. We notice this again and again, and every developer knows it, that you see code and think "what kind of chaos is this, why was it done this way?" In half of the cases, the objection is correct and in the other half the answer is "because we could not do the way you suggest for such and such reasons". I think that's very important, to see what all the possible alternatives were. They don't have to be completely worked out, but you at least have to say "we could have done it that way". Two sentences are enough. You don't have to write a lot of code for that, but if you already have ideas about how you could do something alternatively, then chances are that someone else will come up with that idea at some point. And then it would be good to say why it wasn't done that way.
Navigation, traceability	The third feature would be the link to the code so that you also get from the code to the decisions. You can go the way we are using now. If I want to know what has changed in the code, then I can say in IntelliJ or Eclipse "annotate me the class" and then to the left of each line is when it last changed and by whom and in which commit. We also include the Jira ticket identifier in every commit message. This gives me a reasonably quick understanding of the context in which this class was touched.
Navigation (traceability), searching and filtering, knowledge visualization/presentation	The dependencies between the features should be shown and one should be able to trace decision paths. A search is very important in any case. Over time, the database will become huge, because many features will be developed. I think it is not so easy to implement a meaningful search. Depending on how often documentation is done. The search is in principle most important, because if you document something and then cannot find it, then it also brings nothing. It is important that you see the points that you want/must see at that moment. If everything is documented, it quickly becomes a lot. It is also very important to present this data in a concentrated way.
Knowledge management, exploitation (accountability), traceability	I find the topic very exciting. We are also active in this area in a certain way. However, more on the business processes, when a company wants to know how the customer reacts and what needs to be changed to make the customer happy. The knowledge repository is suitable for recording why certain business processes are changed.

Continued on next page

Features	Practitioner's Statement regarding Important Features and Additions
Metrics calculation and presentation in dashboard, decision-making support based on usage knowledge	The most important feature for me would be the different metrics that I get from monitoring usage and other technical metrics. That I can offer a function there for each feature that will fast-track the metrics together for me in the way that I think I should decide. So that after a certain amount of time, I implement or "keep alive" the feature that provides the better metric. That would be something that would simplify the argumentation in many places because I can then discuss with other stakeholders at the beginning what is most important in terms of user feedback and customer satisfaction. And I can then conclude with what we have agreed, which means that we have made that decision. That I could drill down. What would also be interesting afterward is to confirm the assumptions that you make in an experiment, but then also look afterward to see if they still hold. That would be a constant check on the numbers. A kind of interactivity in the knowledge repository. Several stakeholders are usually involved in the decision-making process. A third feature would be to be able to record discussions there, for example, if you can store formulas for the metrics, it must also be clear why you have chosen this formula. This is also a decision that subsequently leads to whether A or B is implemented. If the factors are weighted appropriately, the decision can lead to both A and B. There, too, I have to record the decisions.
Interoperability (technical interfaces), documentation of decisions and related discussions, decision-making support through voting	Branching and merging is important. It must also be possible to address the knowledge repository independently of the dashboard, i. e. there must be interfaces. Developers like to work close to the hardware. Reading and writing with markdown and YAML should be possible. The difference between discussion and decision should be possible, so it should also be possible to record discussions. In addition to feedback FB, developer and stakeholder discussion should also be able to be entered into the knowledge repository. Voting should be made possible in the tool. It should be a decision voting engine, so to speak, with which one can interact.
Accountability, decision-making support (voting)	Simply a D1 <a decision> is probably not yet sufficient as atomic information. It must also be recorded who was involved in the decision-making process and how, i. e. that four people were in favor and three against, and how the processes for this took place.
Traceability, integrated documentation locations, decision-making (discuss and reject decisions), change impact analysis	I think it's generally important to have documented decisions. I think the key to success is that it's close to the developers. For example, if it's a Word document in a Sharepoint, that's so far away from the developers that when in doubt, nobody looks in there. If it's in the wiki, it's a little bit closer to the developers. The supreme discipline would be to integrate this information into a pull request, so to speak. The most important thing is that it is close to the developer and that when a piece of information (like a decision) is recorded, you also know what is meant by it and that you see the impact, i. e., on which other artifacts this decision still has an impact. But this is all hypothetical because I have never seen such a system. It's supposed to be close to the developer and maybe the developer should also have the possibility to give feedback on the decisions. So it should be possible that you can also discuss the decisions again or ask: "OK, there is a decision, but I have a completely different opinion about it."
Integration, decision-making support based on usage knowledge	Well, you know how it is: the better the tool integration, the easier it is for you. If you have a tool that you can say, "hey look, on this feature branch there's this, and on each one there's this", that means the work is already done, two variants of it have already been developed, and I just have to click on them: that one to these 5 users, and that one to these 5 users, and tomorrow you'll look at the answers, then I'm sure you'll do that in certain cases because it's already interesting as well. Just if it all has to be done manually . . . It just has to be well integrated.

Continued on next page

Features	Practitioner's Statement regarding Important Features and Additions
Integration and traceability	Integration with the tools people normally use is very important. As soon as you have to document twice, nobody does it from experience. That is, in Google Docs we enter the version numbers, you can just about rely on that, but that you then deposit individual Jira issue numbers to a discussed decision, you then usually do not, because it is clear anyway, what it is about. Whether it can then be tracked again in five years to a specific Jira issue, we will then see. But having tooling in this direction is, I think, very helpful, such as linking integrated development environments and software parts. It just has to work with the existing tools. Otherwise, you don't do the abstract documenting. What I also find interesting would be to link it even more or to subdivide it even more. A feature can be: rebuild the view, but a feature can also be, we make a completely new huge feature, a new use case. So that you can still link it. That you say, we are still building this milestone, and all these features are attached to it, and these decisions then flow into the milestone. So in short: you have one more subdivision, maybe one level. And maybe you can also link the features, and not just have individual features.
Automation, navigation support, change execution, nudging, metrics and dashboard	Generally speaking: It automates everything that can be automated. I can always edit it afterward. And, that I can jump back and forth between my knowledge repository and my CSE infrastructure very quickly, so that at any point in my branch, if I see something, I'm directly in the knowledge repository at that point, and I can add the corresponding knowledge that I've gained. Maybe also that the dashboard can directly generate any suggestions, warnings, ... something like that maybe already itself. So that the dashboard not only visualizes knowledge.
Automation, traceability	I think that this should all be automated. Otherwise, I can't get it in. In other words, if that eats up part of my development or management capacity, then of course you have a problem establishing such a system. Therefore, a high degree of automation would be a prerequisite. I don't want to link it individually, but it should be obtained from the data that I have across all my systems, whether in Jira or git so that it can be linked automatically. Because I already enter the data somewhere.
Decision making based on run-time infos (logs, active features)	It would be important for a project team to know which feature is productive at all. That's why you need a link between the knowledge repository and the CSE infrastructure. That information exchange also takes place there, so that you can then see which version is deployed and which features are contained in it. That would be an important criterion for me. What would also be important (but this is more technical) is that the developer gets quick access to log data from production. This is mostly missing. When users report bugs, it's usually the case that the developer can't get to the data from production at all. There are tools for collecting these logs and making them available to developers via dashboards. However, this is always difficult because it involves personal data (which is critical), even if this data is not contained in it or if, for example, it runs like the US stock exchange and, according to SOX criteria, development must be very strictly separated from production. In the end, the developer gets nothing from the production.

Table B.19.: Anonymized examples of obstacles of continuous rationale management reported by practitioners (RQ3.2).

Obstacle	Practitioner's Statement regarding the Obstacles
Intrusiveness and effort problem	The disadvantage would be that from a developer's point of view you would have to put a lot of work into documenting it. So if I open a feature branch, and name it correctly, and then that already works automatically, then that would be super cool, then that would also be useful to many. But if I first have to make a new entry in such a system, then have to enter: these belong together in such and such a way, the feature branch belongs to this knowledge repository entry or proposal, and there I would like to link this analytics data, please, and have to set everything up manually, then it would all be such a big obstacle that people probably wouldn't use it.

Continued on next page

Obstacle	Practitioner's Statement regarding the Obstacles
Intrusiveness and effort problem, distributed documentation	The knowledge repository already exists in parts in the form of ticketing systems, e. g. Redmine already does this. The problem, however, is that you always have to search in two places. One is lazy and does not do exactly that; developers also want to do a lot via the console.
High amount	Over time, the database will become huge because many features will be developed. I think it is not so easy to implement a meaningful search.

B.2. Description of Related Work

Tang et al. (2006) surveyed 81 participants to investigate the value of design rationale to practitioners and how they use and document design rationale. They also investigated which types of rationale practitioners do (RQ2.1 a) and do not capture (RQ2.2 a). They refer to non-existence decisions as *discarded decisions*. About 44 % of their respondents answered to document discarded decisions very often, whereas about 36 % of their respondents do not document discarded decisions. That confirms our finding that some practitioners stated capturing, for example, non-existence decisions, while others do not capture them. Amongst others, they list the following documentation locations, techniques, and tools mentioned by their study participants (RQ2.1 b): office tools, UML tools, document architecture decisions using formal method and notation, internally developed tools, and requirements traceability matrix. They report the following benefits of documenting design rationale (RQ2.1 f): support maintenance and modification tasks as well as impact analysis. Similar to our research questions RQ2.2 b and RQ3.2 as well as the problems listed in Section 1.2, Tang et al. (2006) identified barriers to the use and documentation of design rationale. Lack of time or budget was found to be the most common cause of not documenting design rationale. For example, other barriers were a lack of appropriate standards and tool support, the unawareness of the need and usefulness of documenting design rationale, a lack of a formal review process, and the dynamic nature of technology and solutions make it useless to document design rationale. Tang et al. (2006) discuss their results regarding various aspects and provide ideas for general improvement and a few tool features (RQ3). They identified a need for *guidelines under which the use and documentation of design rationale will provide more benefits than the costs involved*. They mention a traceability feature to support the systematic design rationale retrieval to be useful.

Miesbauer and Weinreich (2013) and Weinreich et al. (2015) conducted two interview studies on kinds, influence factors, and documentation of design decisions in practice. Similar to our study (RQ2.1 a), Miesbauer and Weinreich (2013) classified these decisions according to Kruchten's taxonomy (Kruchten, 2004; Kruchten, 2009). They also found that most of the decisions were existence decisions and that non-existence and property decisions were rarely mentioned. Weinreich et al. (2015) created a new classification scheme based on the practitioners' examples. They found that software architects classify decisions as solution-oriented and driver-oriented. For example, solution-oriented decisions concern the software structure and technology selection. Driver-oriented decisions focus on the qualities that must be achieved, i. e., quality requirements. Further, they found that software architects often classify decisions according to granularity, scope, and impact. According to Weinreich et al. (2015), decisions regarding high-level design (architectural decisions) are usually documented. Decisions with a local scope are either documented in code, in the issue tracking system or not documented at all. Their study participants named the following documentation locations (RQ2.1 b): source code, meeting minutes, project diaries, tickets, text documents, emails, presentations, diagrams, and wikis. Regarding the evolutionary history (RQ2.1 d), they found that the versioning of decisions was considered to be very important by the practitioners but do not detail how versioning is done.

They asked when design decisions are made and documented (RQ2.1 e). They mention meetings and that architecture decisions are "typically a group effort", while developers make decisions with a local scope on their own. They identified the following benefits of design decision documentation (RQ2.1 f): training of new employees, arguing with customers during development, resolving production problems, impact analysis during product maintenance, and preventing knowledge vaporization. Weinreich et al. (2015) asked for types of decisions that the practitioners consciously do not capture. In contrast, we asked what important decisions the practitioners do not capture (RQ2.2 a), which is a different question. However, Weinreich et al. (2015) provide examples for decision types not captured that are important to the majority of their study participants: Tool and property decisions as well as decisions related to the development process are often not documented. Non-existence decisions are rarely documented. As reasons for not capturing decisions (RQ2.2 b) they found: the documentation of design decisions is often too time- and cost-intensive, outdated documentation and redundancies that lead to inconsistencies, an unclear cost/benefit ratio when documenting decisions, the difficulty in finding a documented decision or determining whether a specific decision is documented or not, and the difficulty in deciding what and how best to document the knowledge. Similar to our research question RQ3, they make the following recommendations for improved architectural knowledge management: Reducing biases is important to support an objective and reasoned decision process, organize decisions with requirements and qualities (e. g., in the issue tracking system), use patterns for ease of capturing (e. g., referencing pattern catalogs), identify main drivers for decision makers, adjust decision documentation techniques to team size, support the recovery of decision documentation from various sources e. g., from the version control system such as van der Ven and Bosch (2013).

Capilla et al. (2016) performed an interview study to understand the state of practice regarding architectural knowledge management. They interviewed six practitioners from the industry working in the role of software architects. They provide a few examples of architectural knowledge captured, e. g., API documentation, but not for decision types (RQ2.1 a). Word files, wiki systems, and PowerPoint were mentioned as documentation locations for general architectural knowledge, including requirements, architecture views, and design decision models (RQ2.1 b). They discuss the following benefits of architectural knowledge management (RQ2.1 f): avoid knowledge vaporization, understand the ripple effect of decisions, identify and track the root causes of changes and better estimate the impact analysis, share decisions among the relevant stakeholders, understand the evolution of the system, understand the underpinning reasons for decisions and build on experience, and identify critical decisions and run-time concerns. They asked for architectural knowledge not captured by the practitioners but worthwhile to be captured. The practitioners answered that design rationale would be relevant to be captured (RQ2.2 a) to explain why the software was designed in a certain way (RQ2.2 c). They identified the following barriers, i. e., reasons why architectural knowledge management is not performed (RQ2.2 b): Lack of motivation or incentive, lack of adequate tools, uncertainty of what to capture, effort in capturing architectural knowledge, disrupting the design flow, and lack of stakeholder understanding. Similar to our research question RQ3, they identified remedies: Encourage and convince the stakeholders, systematize the knowledge-capturing process, lean approaches for capturing architectural knowledge, embed the design rationale with current modeling approaches and tools.

Schubanz (2021) surveyed the as-is state of rationale management in agile software development, in particular Scrum. One hundred-two practitioners participated in the survey. They tested various hypotheses. For example, their responses indicate that a developer's experience significantly impacts the documentation frequency and the interest in the available documentation. They asked for types of decisions that practitioners document (RQ2.1 a) and that they think are worth documenting (similar to RQ2.2 a. Still, it is unclear whether the practitioners currently do not capture the decision types). They have predefined decision types that the practitioners rated

(in descending order by their rating to be worth documenting): decisions regarding development tools, feature refinement, architecture/design, quality, feature prioritization, user experience, deployment, process, technology, team, and to-do items. Examples of free-text answers were: changes in solution decisions and decisions where it is not immediately obvious that it is the optimal choice. They asked for media types and means to capture decisions (RQ2.1 b). They provided a list of Scrum practices, such as retrospective and sprint planning, and asked for the practitioners' decision documentation frequencies per activity (RQ2.1 e). They asked for criteria motivating the practitioners to document decisions and situations in that a decision documentation is useful (RQ2.1 f): for example, *preparation for later audits* and *compliance with legal regulations*. Further, they asked for reasons not to document decisions (RQ2.2 b). They received various answers, of which we arbitrarily picked the following: *Decisions can change faster than documentation, so small decisions are more likely to be documented directly in the code rather than using wikis. Sometimes decisions are made intuitively, and there is no analysis of the alternatives. Decisions made unconsciously are not documented.*

Appendix C

Supplementary Material of Systematic Mapping Study

"Quality deliveries with short cycle time need a high degree of automation."

—Ebert et al., 2016

This appendix provides supplementary material for the systematic mapping study in Chapter 4.

Listing C.1 shows the query for the ACM digital library[1]. We searched the ACM Full-Text Collection and excluded publications before 2010.

Listing C.1: ACM digital library key-word query

```
(Title:("rationale") OR Title:("decision") OR Abstract:("rationale") OR Abstract:("decision"))
AND (Title:("knowledge") OR Abstract:("knowledge") OR Title:(document*) OR
    Abstract:(document*) OR Title:(manage*) OR Abstract:(manage*))
AND (Title:(recommend*) OR Title:(classif*) OR Abstract:(recommend*) OR Abstract:(classif*))
AND (Title:("technique") OR Title:("approach") OR Title:("method") OR Title:("support") OR
    Abstract:("technique") OR Abstract:("approach") OR Abstract:("method") OR
    Abstract:("support") OR Title:("system") OR Abstract:("system"))
AND (Title:("software") OR Abstract:("software"))
```

Listing C.2 shows the query for the IEEE Xplore digital library. We entered the query in the command search interface[2] and excluded publications before 2010 and books.

Listing C.2: IEEE Xplore digital library key-word query

```
("Document Title":"decision" OR "Document Title":"rationale"
    OR "Abstract":"decision" OR "Abstract":"rationale")
AND ("Document Title":"knowledge" OR "Abstract":"knowledge" OR "Document Title":"document*"
    OR "Abstract":"document*" OR "Document Title":"manage*" OR "Abstract":"manage*")
AND ("Document Title":"recommend*" OR "Document Title":"classif*"
    OR "Abstract":"recommend*" OR "Abstract":"classif*")
AND ("Document Title":"approach" OR "Document Title":"technique" OR "Document Title":"method"
    OR "Document Title":"support" OR "Abstract":"approach" OR "Abstract":"technique" OR
    "Abstract":"method" OR "Abstract":"support" OR "Document Title":"system" OR
    "Abstract":"system")
AND ("Document Title":"software" OR "Abstract":"software")
```

[1] https://dl.acm.org/search/advanced

[2] https://ieeexplore.ieee.org/search/advanced/command

Appendix D

Supplementary Material of Knowledge Documentation Analysis

"Capturing the right set of links can help increase decisions' sustainability."

—Zdun et al., 2013

This appendix provides supplementary material for the knowledge documentation analysis in Chapter 9. Section D.1 describes the documentation of the six validation projects' decision knowledge, system knowledge, and project knowledge. Section D.2 contains additional plots of the knowledge documentation data.

D.1. Description of Knowledge Documentation of Validation Projects

We describe the knowledge documentation of the case study projects to enable their comparison and conclusions in the remainder of this chapter. We qualitatively and quantitatively describe 1) the requirements specification and other tickets in the issue tracking system, 2) code and commits in the version control system as well as their links to the issue tracking system of the four projects for that we have access to the version control system data, 3) the decision knowledge documentation including the rationale types, documentation locations, states of issues and decisions, intra-rationale completeness, and decision types, and 4) the relation of the decision knowledge to requirements and code.

Workplace-Control App

For the workplace-control app, the developers documented 11 epics. Seven epics relate to app features or components: *Setting up new Smart Devices*, *Smart Device Control*, *Energy Consumption Awareness*, *Augmented-Reality View of Smart Devices*, *Fog Node Architecture*, *Smart Environment Automation Server*, and *User Management*. A particularity is that the four remaining epics describe project-specific organizational aspects: *Client Acceptance Test*, *Design Review*, *Shoot a Trailer for the Design Review*, and *Documentation*. 35 user stories are documented, e. g., titled *Show aggregated energy consumption*, *Comparison of energy consumption to other occupants*, *Control smart lamps*, *Show devices in the current point of view*, and *Show additional information about selected device*. The project documentation of the workplace-control app contains 172 development tasks, 309 development sub-tasks, and 50 bug reports.

The developers of the workplace-control app documented 33 issues, 56 decisions, 16 alternatives, and 26 arguments. All of the 131 decision knowledge elements are documented as entire tickets. Five issues are unsolved. During the iPraktikum projects, ConDec did not offer the functionality to mark rejected decisions. Thus, the respective number is unknown. 30.3 % of the 33 issues have at least one alternative documented (next to the decision). For 75 % of the decisions, an issue is linked. For 16.1 % of the decisions, at least one pro-argument is linked. At least one con-argument is linked for 18.8 % of the alternatives.

The decisions for the workplace-control app are assigned to the following types: 45 % are executive decisions (e. g., *We do not assign tasks, but people grab them!* and *We will use our current setup with fog node on the virtual machine and the virtual-private-network tunnel to the lamps!*), 25 % are quality-driven (e. g., *Delete device via long press!* for usability), 21 % are functionality-driven (e. g., *Set consumption in contrast with the hours spent at the smart office!* and *Show consumption in the last 30 days in energy report!*), 20 % concern the frontend (e. g., *Add brightness slider with icons on side of labels!*), 7 % concern the backend and data storage (e. g., *Use InfluxDB for saving analytics data!*), 5 % concern the API (e. g., *Determine current occupants via fog node connection!*), and 2 % relate to software testing (*Allow creation of multiple World Maps [for testing] during development stage!*). Since we assigned the types retrospectively, no other decision types are assigned to the decisions for the workplace-control app.

97.1 % of the user stories are not linked to decisions. Only one user story titled *split up world maps and anchors again* is linked to one decision via the respective issue: *What action should we take when it is too dark for loading the world maps? An option to turn on the lights should be displayed!*

Car Charging App

For the *car charging app*, the developers documented 31 scenarios, e. g., titled *Configure smart device notifications*, *Enable and disable notifications*, *Join neighborhood as charging station owner*, *Add charging station to own account*, *Search for charging stations nearby*, *Accept/Deny friend request/invite*, and *Autonomous car drives to charging station.* Besides, the car charging app project documentation contains 162 development tasks, 593 development sub-tasks, and 57 bug reports.

The developers of the car charging app documented 21 issues, 11 decisions, 29 alternatives, and 38 arguments. All of the 99 decision knowledge elements are documented as entire tickets. Eleven issues are unsolved. 57.1 % of the 21 issues have at least one alternative documented. For 90.9 % of the decisions, an issue is linked. For 9.1 % of the decisions, at least one pro-argument is linked. At least one con-argument is linked for 44.8 % of the alternatives.

The decisions for the car charging app are assigned to the following types: 36 % are executive decisions (e. g., *Use docker!*), 18 % concern the frontend (e. g., *Replace the search tab bar element with discover view!*), 45 % concern the backend and data storage (e. g., *Composite between UserClass – HouseClass – SmartDeviceClass!*), and 9 % relate to software testing (*Mock Tesla!*). Since we assigned the types retrospectively, no other decision types are given to the decisions for the car charging app app.

All of the scenarios are not linked to decisions. The maximal decision coverage of scenarios is zero. However, one development task and two development sub-tasks are linked to decisions. For example, the task *Set up virtual machine* is linked to *Should the database run in docker?* and *Use docker!* with the pro-argument *Reproducible on other systems.*

IoT Platform

For the IoT platform, the developers documented seven epics: *Basic Device Management*, *Device Overview*, *Device Detail View*, *Master-data Management*, *Tag Management*, *Mock Device Data Provider*, and *Anomaly Detection*. The requirements specification for the IoT platform contains 53 user stories, e. g., titled *Register a device*, *Modify a device*, *Tag a device with arbitrary tags*, *See an overview of all devices*, *See all measured data from a device*, and *See a historical list of anomalies for a device*. Each of the user stories contains a description using the Connextra template, i. e., in the form *I as a <role> want <function> so that <business value/some reason>* (Cohn, 2004). For example, the user story titled *See a detailed view of an anomaly* is described as *As an IoT manager, I want to see anomalies highlighted in the device data so that I can figure out if something is wrong with my devices or their data.* A particularity is that 26 user stories specify non-functional requirements, e. g., *As an IoT manager, I want to store at least 50 000 devices to manage a vast IoT landscape.* Every non-functional user story is linked to a quality attribute: either to *Security*, *Maintainability*, *Portability*, *Usability*, *Compatibility*, *Reliability*, *Performance Efficiency*, or *Functional Suitability*. Besides, the IoT platform project documentation contains 27 development tasks, 129 development sub-tasks, and 35 bug reports.

The IoT platform project has five git repositories[1], for the frontend, backend, deployment, a crawler for air-quality data, and for setting up the development infrastructure. In total, the master branches of the repositories contain 998 commits. 19 % of the commits have a valid ticket identifier in their commit message. The knowledge graph contains 101 code files (10553 LOC) of the seven types ts, vue, js, xml, scss, conf, and html. Particular code types in the IoT platform project are vue for the Vue JavaScript framework and scss for Sass CSS, a preprocessing language for CSS. 23.8 % of the code files (24 files, 1730 LOC) are for testing. 56 % of the code files are linked to at least one ticket in the issue tracking system.

The developers of the IoT platform documented 111 issues, 116 decisions, 94 alternatives, 184 pro-, 113 con-arguments, and one unpositioned argument. In total, they documented 619 decision knowledge elements. 1.9 % of the decision knowledge elements are documented as entire tickets, 97.9 % in the description and comments of existing tickets such as user stories or development tasks, and 0.2 % in commit messages. Four issues are unsolved, and 8 of the decisions are marked as rejected. 59.5 % of the 111 issues have at least one alternative documented. For all of the 116 decisions, an issue is linked (there are issues with more than one decision). For 73.3 % of the decisions, at least one pro-argument is linked. For 71.3 % of the alternatives, at least one con-argument is linked.

The decisions for the IoT platform are assigned to the following types: 12 % are executive decisions, 17 % are quality-driven, 9 % are functionality-driven, 44 % concern the frontend, 28 % concern the backend and data storage, 10 % concern the API, 3 % concern an external library or framework, and 3 % relate to software testing. The developers assigned the following additional decision types: The quality attributes *usability*, *security*, and *availability*, the software features *information sharing*, *navigation*, and other types *user roles, architecture, implementation, responsive layout, prioritization, naming*, and *error handling*.

54.7 % of the user stories are not linked to decisions. Eight user stories are linked to one decision. Sixteen user stories are linked to more than one decision. The maximal decision coverage of six decisions is reached by two user stories titled *As an IoT manager, I want to register new devices, so that I can maintain my infrastructure* and *As an IoT manager, I want to tag a device with arbitrary tags, so that I can find this device with these tags.* 81.2 % of the code files are not linked to decisions. 2 % of the code files are linked to one decision. 16.8 % of the code files are linked to multiple decisions. Seventeen code files reach the maximal decision coverage of 2 decisions.

[1] `https://github.com/HEIOT`

Web Search Engine

The requirements specification of the web search engine contains the following three epics: *Search Input*, *Search Processing*, and *Search Result Presentation*. The epics are refined into 13 user stories, which are titled according to the Connextra template. For example, *Search Processing* is refined into *As a search engine user, I want to find documents that contain synonyms to my search terms, so I don't have to try each synonym individually* and *As a search engine user, I want to find information about my search input even if I made spelling mistakes, so I don't have to consider correct spelling*. Besides, the requirements specification contains one user role (*Search Engine User*), four personas, and six quality requirements (e. g., *A search should not take longer than 1 second* and *Documents containing the information needed should always be under the top 10 search results*). Besides, the web search engine project documentation contains 131 work items and 34 bug reports.

The web search engine has one git repository with 185 commits on the master branch. 78 % of the commits have a valid ticket identifier in their commit message. The knowledge graph contains 658 code elements (141266 LOC) of the three types ts, html, and xml. A particularity is that the code of the Apache Solr and Apache Nutch libraries are part of the repository, which results in high numbers of code elements and LOC. 1.8 % of the code files (12 files, 1724 LOC) are for testing. All the code files are linked to at least one ticket in the issue tracking system.

The developers of the web search engine documented 81 issues, 116 decisions, 118 alternatives, 262 pro-, and 197 con-arguments. In total, they documented 774 decision knowledge elements in two different documentation locations: 6.5 % of the decision knowledge elements are documented as entire tickets, which are mainly issues, such as *Which framework should we use for the search engine?* and *Which framework should we use as a webcrawler?* They captured solution options and arguments in the description and comments of these issues. Altogether, 93.5 % of the decision knowledge elements are documented in the description or comments of existing tickets, such as the issue tickets, user stories, work items, and bug reports. All of the 81 issues are solved, and nine decisions are marked as rejected. 79 % of the 81 issues have at least one alternative documented. For all of the 116 decisions, an issue is linked (there are issues with more than one decision). For 94 % of the decisions, at least one pro-argument is linked. For 76.3 % of the alternatives, at least one con-argument is linked.

The decisions for the web search engine are assigned to the following types: 12 % are executive decisions, 9 % are quality-driven, 26 % are functionality-driven, 22 % concern the frontend, 44 % concern the backend and data storage, 4 % concern the API, 4 % concern an external library or framework, and 9 % relate to software testing. The developers assigned the following additional decision types: *crawler, indexer, suggester, query processing*, and *legality*.

All user stories are linked to more than one decision. The maximal decision coverage of 27 decisions is reached by one user story titled *As a search engine user, I want to see my search results as a paginated list to navigate through the pages*. 4.1 % of the code files are not linked to decisions. 1.7 % of the code files are linked to one decision. 94.2 % of the code files are linked to more than one decision. The maximal decision coverage of 39 decisions is reached by one code file named *schema.xml*.

Soccer App

The requirements specification of the soccer app contains the following four epics: *Team Administration and Rating*, *Player Administration and Rating*, *Exercise Management*, and *Event Management*. The epics are refined into 33 user stories, e. g., *Create team*, *View exercise recommendations for entire team*, *Rate team*, and *View Exercise Details*. Each of the user stories contains a description using the Connextra template. For example, the user story titled *View exercise recommendations for entire team* is described as *As a coach, I want to see exercise*

recommendations for the entire team to integrate them into practice. Besides, the requirements specification contains two user roles (*Coach* and *Player*), four personas, and seven quality requirements (e. g., *The application should not take too long to load in any view, it should not take longer than 0.5s*). Besides, the soccer app project documentation contains 179 work items and 39 bug reports.

The soccer app project has two git repositories, one for the frontend and the backend. In total, the master branches of the repositories contain 983 commits. 48 % of the commits have a valid ticket identifier in their commit message. The knowledge graph contains 292 code files (19268 LOC) of the eight types java, ts, js, xml, css, conf, html, and yaml. A particular code type in the knowledge graph of the soccer app project is yaml for configuration. 9 % of the code files (26 files, 3355 LOC) are for testing. 97 % of the code files are linked to at least one ticket in the issue tracking system.

The developers of the soccer app documented 73 issues, 76 decisions, 84 alternatives, 203 pro-, and 154 con-arguments. In total, they documented 590 decision knowledge elements. 8.6 % of the decision knowledge elements are documented as entire tickets, mainly issues. 90.5 % of the decision knowledge elements are documented in the description and comments of existing tickets, and 0.8 % in code comments. One issue is unsolved, and three of the decisions are marked as rejected. 94.5 % of the 73 issues have at least one alternative documented. For all of the 76 decisions, an issue is linked (there are issues with more than one decision). For 97.4 % of the decisions, at least one pro-argument is linked. For 95.2 % of the alternatives, at least one con-argument is linked.

The decisions for the soccer app are assigned to the following types: 21 % are executive decisions, 9 % are quality-driven, 17 % are functionality-driven, 45 % concern the frontend, 24 % concern the backend and data storage, 5 % concern the API, 7 % concern an external library or framework, and 7 % relate to software testing. The developers assigned the following additional decision types: *architecture, compatibility, implementation, prioritization,* and *security.*

All user stories are linked to more than one decision. The maximal decision coverage of 8 decisions is reached by one user story titled *As a coach, I want to browse different exercises to get inspiration for practice.* 30.5 % of the code files are not linked to decisions. 14.4 % of the code files are linked to one decision. 55.1 % of the code files are linked to more than one decision. The maximal decision coverage of 13 decisions is reached by two code files named *Player.java* and *PlayerService.java.*

ConDec Plug-Ins

Table D.1 presents an overview of the knowledge documentation of the ConDec plug-ins. We use the notations of *Task and Object-oriented Requirements Engineering* to specify the requirements for the ConDec plug-ins (Paech and Kohler, 2004). The requirements specification of ConDec contains five major user roles: *Product Owner*, *Rationale Manager*, *Developer*, *Meeting Manager*, and *Release Manager* (Section 7.1). For each user role, we specified one user task, e. g., the user task *Rationale Management* for the *Rationale Manager* role. We refined the user tasks into 14 sub-tasks. The IT support for the sub-tasks is specified using 98 system functions. The user-interface structure consists of 48 workspaces. In total, 14 quality requirements are documented for all ConDec plug-ins. Diagrams for the domain data model and the user-interface-structure diagram are documented in the wiki. Besides, the ConDec project documentation contains 493 work items, 151 bug reports, 50 system test cases, and five test execution tickets. All ConDec plug-ins share the task specification. The ConDec Jira plug-in implements the highest number of system functions of the ConDec plug-ins. The ConDec Jira plug-in offers 74 system functions, and the Jira project contains 843tickets. Table D.1 does not list user tasks and sub-tasks because they are shared among the ConDec plug-ins.

Table D.1.: Knowledge documentation of the ConDec plug-in development projects (March 1, 2023): 1) requirements and other tickets in Jira, 2) commits and code in git and their trace links to Jira, and 3) decision knowledge in Jira and git.

		ConDec Jira	ConDec Confluence	ConDec Bitbucket	ConDec Eclipse	ConDec VSCode	ConDec Slack
Requirements and other Tickets in Jira							
	#System Functions	74	3	3	12	2	4
	#Workspaces	32	3	3	4	3	3
	#Tickets	843	24	19	54	13	45
Commits and Code in git							
	#Commits	1823	77	53	48	13	19
	#Commits$_L$	1539 (84 %)	39 (51 %)	26 (49 %)	41 (85 %)	2 (15 %)	11 (58 %)
	Code Types in Knowledge Graph	java, js, xml, vm	java, js, xml, vm	java, js, xml, soy	java, js, xml	ts	js
	#Code$_{graph}$ (Test)	792 (397)	26 (15)	29 (17)	82 (28)	4 (3)	11 (2)
	#LOC$_{graph}$ (Test)	278049 (24371)	2070 (942)	2310 (1314)	8421 (2582)	199 (114)	2102 (14)
	#Code$_{comL}$	791 (99.9 %)	25 (96 %)	29 (100 %)	82 (100 %)	2 (50 %)	11 (100 %)
Decision Knowledge Documentation in Jira and git							
Rationale Types	**#Issues**	529	17	13	62	6	11
	#Decisions	652	22	14	65	9	9
	#Alternatives	283	4	6	33	1	9
	#Arguments	453, 377, 11	7, 10	6, 6	38, 51	3, 2	9, 7, 7
Ratios	**#/#**	$\frac{529}{652} = 0.8$	$\frac{17}{22} = 0.8$	$\frac{13}{14} = 0.9$	$\frac{62}{65} = 1$	$\frac{6}{9} = 0.67$	$\frac{11}{9} = 1.2$
	#/#	$\frac{652}{283} = 2.3$	$\frac{22}{4} = 5.5$	$\frac{14}{6} = 2.3$	$\frac{65}{33} = 1.97$	$\frac{9}{1} = 9$	$\frac{9}{9} = 1$
	#(+)/#(+)	$\frac{841}{935} = 0.9$	$\frac{17}{26} = 0.65$	$\frac{12}{20} = 0.6$	$\frac{89}{98} = 0.9$	$\frac{5}{10} = 0.5$	$\frac{23}{18} = 1.3$
	#/#System Function	$\frac{652}{74} = 8.8$	$\frac{22}{3} = 7.3$	$\frac{14}{3} = 4.67$	$\frac{65}{12} = 5.4$	$\frac{9}{2} = 4.5$	$\frac{9}{4} = 2.2$
Intra-Rationale Completeness	**# with**	237 (44.8 %)	4 (23.5 %)	4 (30.8 %)	30 (48.4 %)	1 (16.7 %)	6 (54.5 %)
	# with	652 (100 %)	22 (100 %)	14 (100 %)	65 (100 %)	9 (100 %)	9 (100 %)
	# with	274 (42 %)	5 (22.7 %)	6 (42.9 %)	18 (27.7 %)	3 (33.3 %)	4 (44.4 %)
	# with	182 (64.3 %)	4 (100 %)	4 (66.7 %)	26 (78.8 %)	1 (100 %)	5 (55.6 %)
	#Rationale Elements	2305	60	45	249	21	52
Documentation Origin	**#Elements as Tickets**	68 (3 %)	0 (0 %)	1 (2.2 %)	0 (0 %)	0 (0 %)	18 (34.6 %)
	#Elements in Ticket Text	1507 (65.4 %)	43 (71.7 %)	29 (64.4 %)	183 (73.5 %)	16 (76.2 %)	34 (65.4 %)
	#Elements in Commit Messages	378 (16.4 %)	9 (15 %)	6 (13.3 %)	45 (18.1 %)	5 (23.8 %)	0 (0 %)
	#Elements in Code Comments	349 (15.1 %)	8 (13.3 %)	9 (20 %)	21 (8.4 %)	0 (0 %)	0 (0 %)
Status	**#Open Issues**	8 (1.5 %)	1 (5.9 %)	2 (15.4 %)	3 (4.8 %)	0 (0 %)	0 (0 %)
	#Rejected Decisions	60 (9.2 %)	1 (4.5 %)	2 (14.3 %)	1 (1.5 %)	1 (11.1 %)	0 (0 %)
Decision Coverage of System Functions in Link Distance <= 3							
	No Traceable	0 (0 %)	0 (0 %)	0 (0 %)	0 (0 %)	0 (0 %)	3 (75 %)
	1 Traceable	1 (1.4 %)	0 (0 %)	1 (33.3 %)	0 (0 %)	0 (0 %)	1 (25 %)
	> 1 Traceable	73 (98.6 %)	3 (100 %)	2 (66.7 %)	12 (100 %)	2 (100 %)	0 (0 %)
	Max. # Traceable	2 (2.7 %) → 44	1 (33.3 %) → 10	1 (33.3 %) → 4	1 (8.3 %) → 13	2 (100 %) → 3	1 (25 %) → 1
Decision Coverage of Code Elements in Link Distance <= 3							
	No Traceable	1 (0.1 %)	1 (3.8 %)	0 (0 %)	0 (0 %)	2 (50 %)	8 (72.7 %)
	1 Traceable	21 (2.7 %)	3 (11.5 %)	7 (24.1 %)	0 (0 %)	0 (0 %)	3 (27.3 %)
	> 1 Traceable	770 (97.2 %)	22 (84.6 %)	22 (75.9 %)	82 (100 %)	2 (50 %)	0 (0 %)
	Max. # Traceable	1 (0.1 %) → 225	1 (3.8 %) → 15	1 (3.4 %) → 10	1 (1.2 %) → 25	2 (50 %) → 4	1 (9.1 %) → 1

Each of the ConDec plug-ins (for Jira, Confluence, Bitbucket, Eclipse, VSCode, and Slack) has one git repository[2]. In total, the master branches of the repositories contain 2033 commits. 82 % of the commits have a valid ticket identifier in their commit message. The knowledge graphs of the ConDec plug-ins contain 944 code files (293151 LOC) of the six types java, js, xml, vm, soy, and ts. In the ConDec project, particular code types are Velocity (vm) and soy templates for the user interface. 49 % of the code files (463 files, 29337 LOC) are for testing. 99.5 % of the code files are linked to at least one ticket in the issue tracking system. Table D.1 on page 276 shows descriptive data about the code and commits as well as their links to the issue tracking system for every ConDec plug-in. The ConDec Jira plug-in has the highest number of commits (1823) and code elements (792) of the ConDec plug-ins. The particular high number of LOC (278049) is due to the inclusion of library code[3] in its current version.

The decision knowledge documentation of the ConDec plug-ins contain 638 issues, 771 decisions, 336 alternatives, 516 pro-, 453 con-arguments, and 18 arguments. 3.2 % of the decision knowledge elements are documented as entire issue tracking system tickets, 66.4 % in the description and comments of existing tickets, 16.2 % in commit messages, and 14.2 % in code comments. 14 issues are unsolved and 65 of the decisions are rejected. 44.2 % of the 638 issues have at least one alternative documented. For all of the 771 decisions, an issue is linked (there are issues with more than one decision). For 40.3 % of the decisions, at least one pro-argument is linked. At least one con-argument is linked for 66.1 % of the alternatives. The ConDec Jira plug-in has the highest number of documented rationale elements (2305) of the ConDec plug-ins.

The decisions for the ConDec plug-ins are assigned to the following types: 8 % are executive decisions, 8 % are quality-driven, 26 % are functionality-driven, 33 % concern the frontend, 33 % concern the backend and data storage, 5 % concern the API, 7 % concern an external library or framework, and 5 % relate to software testing. The developers assigned the following additional decision types: *architecture, chatbot, chronology view, change impact analysis, compatibility, configuration, context menu, criteria matrix, dashboard, decision grouping, decision guidance, documentation location, event listener, explainability, export, filtering, git, import, information channel, internationalization, knowledge graph, knowledge linking, knowledge management, knowledge visualization, link recommendation, maintainability, merge check, naming, navigation, nudging, performance, publishing, quality checking, rationale backlog, rationale model, release notes, security, summarization, text classification, tree visualization, usability, webhook,* and *wrong links*. These types represent a collection of features and other important aspects in ConDec.

3.1 % of the specified system functions are not linked to decisions. 3.1 % of the system functions are linked to one decision. 93.9 % of the system functions are linked to more than one decision. The maximal decision coverage of 44 decisions is reached by two system functions titled *Manually classify text in the description or comments of a Jira issue as decision knowledge* and *Automatically add decision knowledge from Jira issue description and comments in the knowledge graph*. 1.3 % of the code files are not linked to decisions. 3.6 % of the code files are linked to one decision. 95.1 % of the code files are linked to more than one decision. The maximal decision coverage of 225 decisions is reached by one configuration code file *atlassian-plugin.xml* of the ConDec Jira plug-in.

D.2. Additional Plots of Knowledge Documentation Analysis

Figure D.1 visualizes the proportion of rationale types documented over time in the validation projects. Gaps in the plot are due to a lack of decision knowledge data.

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

[3] https://github.com/cures-hub/cures-condec-jira/tree/v2.3.6/src/main/resources/js/lib

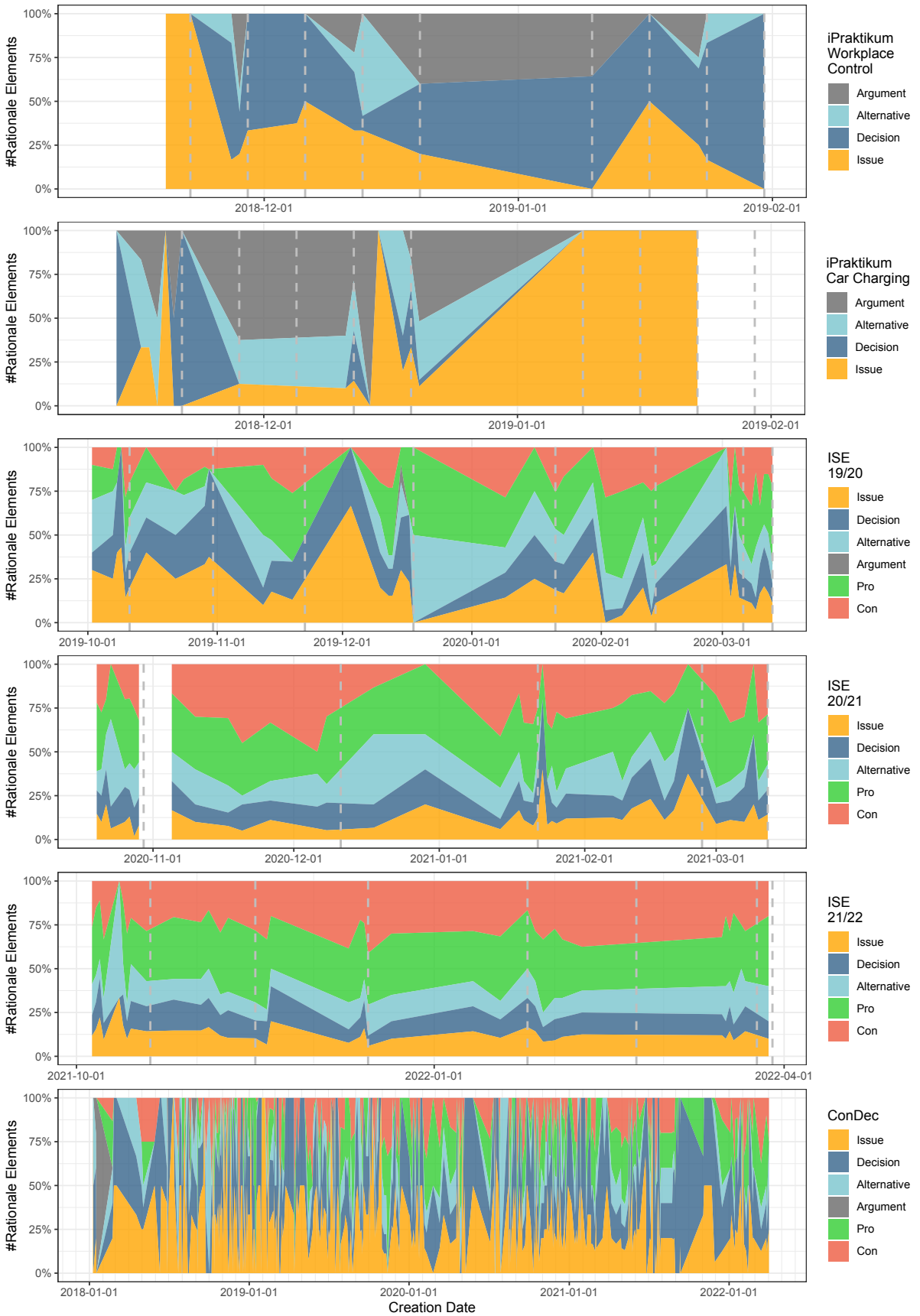

Figure D.1.: Proportion of rationale types documented over time in the validation projects.

Figure D.2 shows the used documentation locations over time. We did not plot the iPraktikum 18/19 projects because the only documentation location used was entire Jira tickets.

Figure D.2.: Number of rationale elements per documentation origin in the validation projects.

Table D.2 lists the number and percentage of decisions per type per validation project.

Figure D.3 visualizes the number of decision groups assigned to the decisions. The groups are a subset of our coding scheme executive, quality-driven, functionality-driven, frontend, backend and data storage, API, external library or framework, testing.

Figure D.4 shows the custom decision groups used in the projects and the respective number of decisions per group.

Figure D.5 shows the proportion of decision types documented over time in the validation projects.

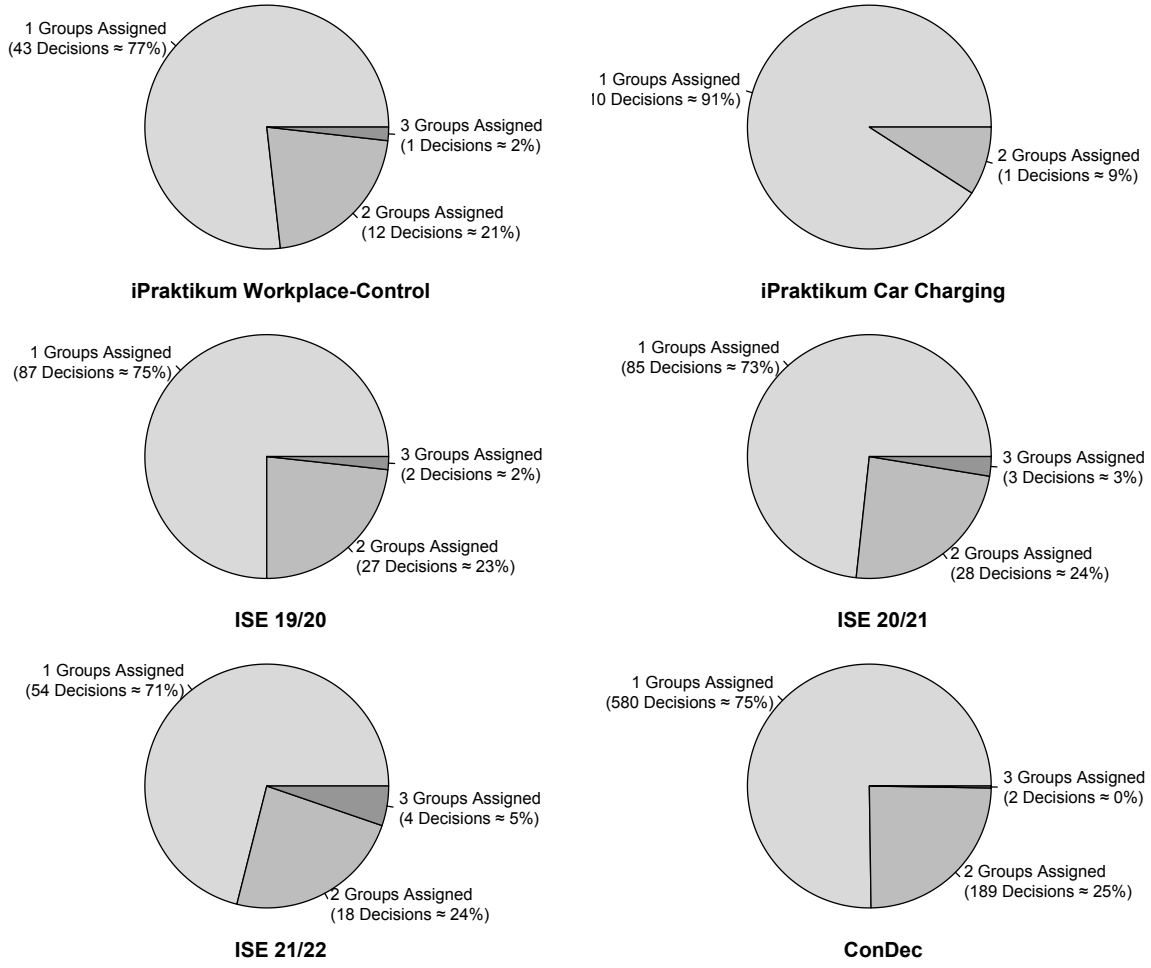

Figure D.3.: Number of groups assigned to the decisions in the validation projects.

Table D.2.: Number of decisions per decision type in the six validation projects. A decision can be assigned to more than one type, thus, the sum of percentages exceeds 100 %.

	iPraktikum 18/19		**ISE 19/20**	**ISE 20/21**	**ISE 21/22**	**ConDec**
Product	Workplace Control App	Car Charging App	IoT Platform	Web Search Engine	Soccer App	ConDec Plug-Ins
Executive	25 (45 %)	4 (36 %)	14 (12 %)	14 (12 %)	16 (21 %)	63 (8 %)
Quality-Driven	14 (25 %)	0 (0 %)	20 (17 %)	10 (9 %)	7 (9 %)	63 (8 %)
Functionality-Driven	12 (21 %)	0 (0 %)	11 (9 %)	30 (26 %)	13 (17 %)	200 (26 %)
Frontend	11 (20 %)	2 (18 %)	51 (44 %)	25 (22 %)	34 (45 %)	257 (33 %)
Backend And Data Storage	4 (7 %)	5 (45 %)	32 (28 %)	51 (44 %)	18 (24 %)	251 (33 %)
API	3 (5 %)	0 (0 %)	12 (10 %)	5 (4 %)	4 (5 %)	42 (5 %)
External Library and Frameworks	0 (0 %)	0 (0 %)	3 (3 %)	5 (4 %)	5 (7 %)	50 (6 %)
Testing	1 (2 %)	1 (9 %)	4 (3 %)	10 (9 %)	5 (7 %)	38 (5 %)

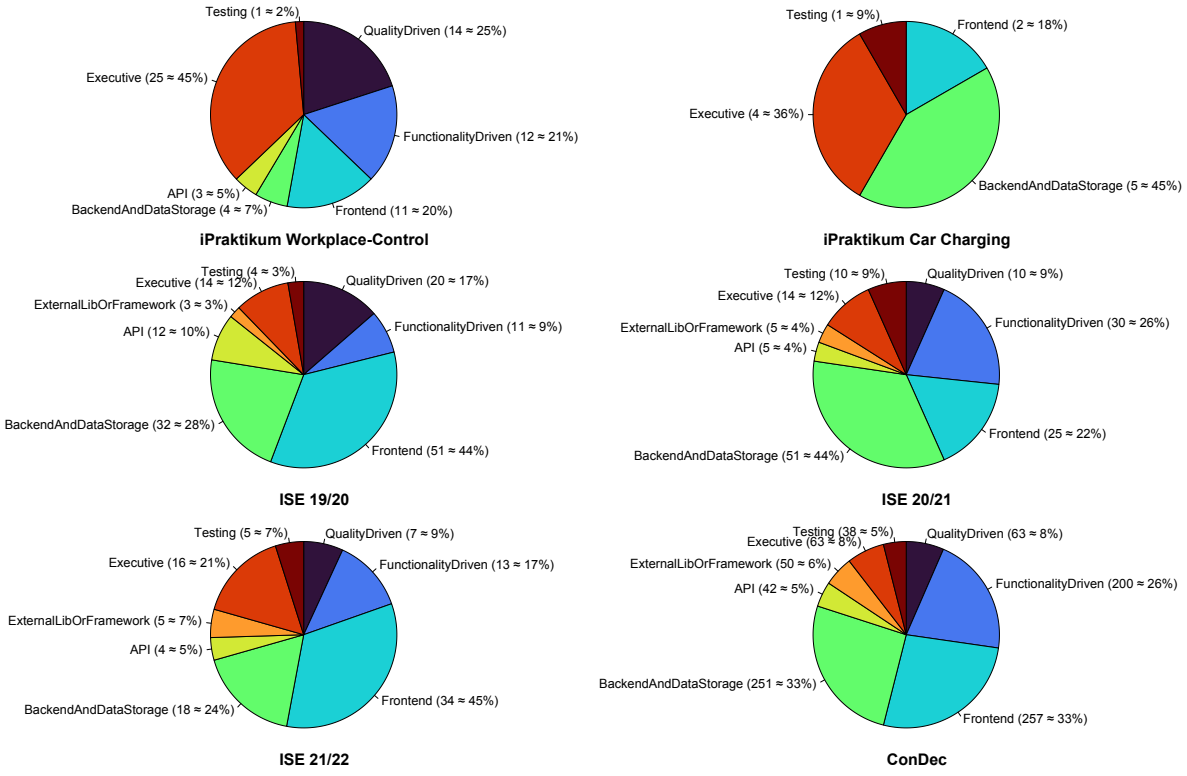

Figure D.4.: Number of decisions per decision type in the six validation projects. A decision can be assigned to more than one type. Thus, the sum of percentages exceeds 100 %.

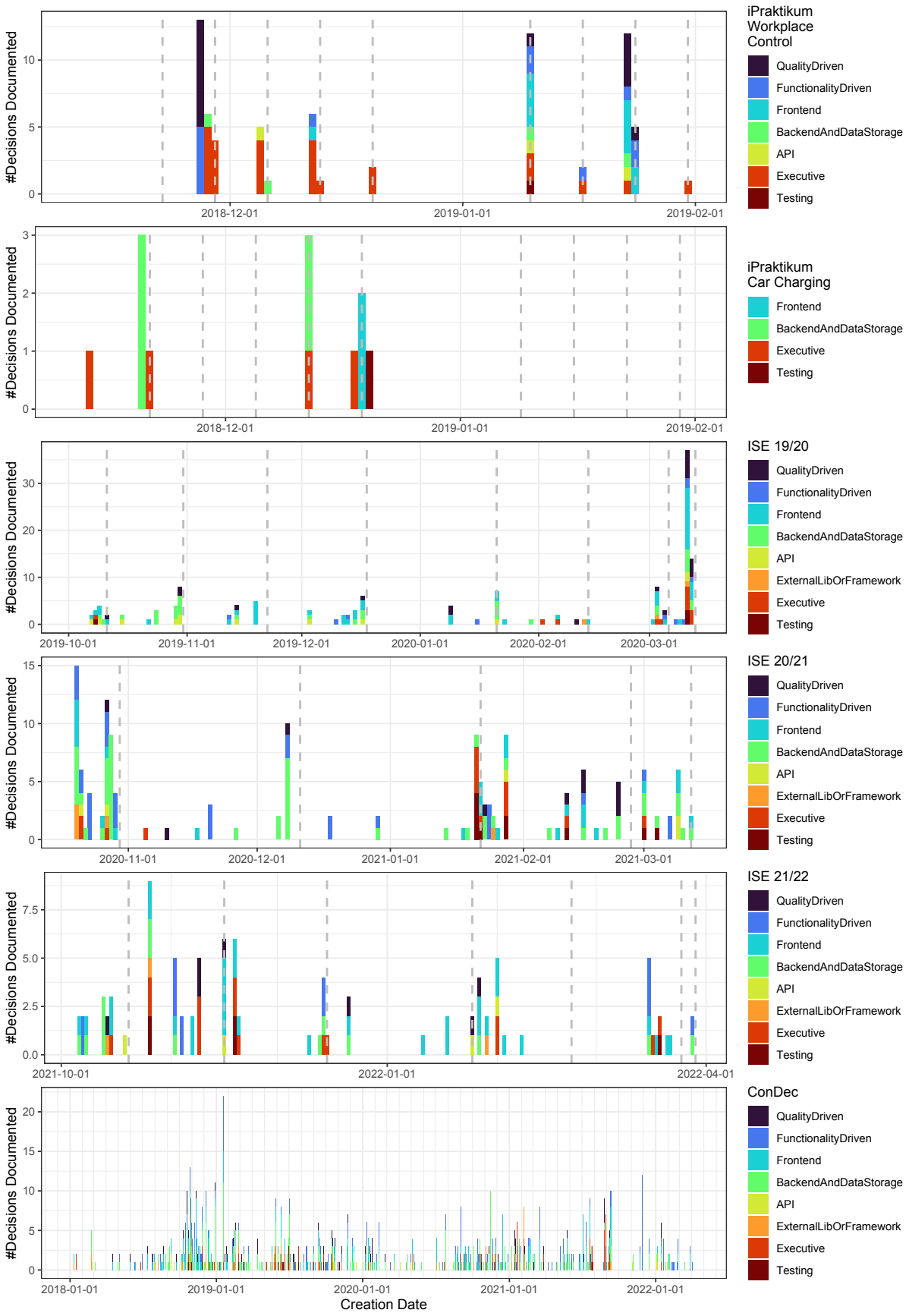

Figure D.5.: Decision types documented over time in the validation projects.

We also coded the decisions by their levels as defined by van der Ven and Bosch (2013). Figure D.6 shows the number and percentage of decisions assigned to the three decision levels. Every decision is assigned to precisely one decision level.

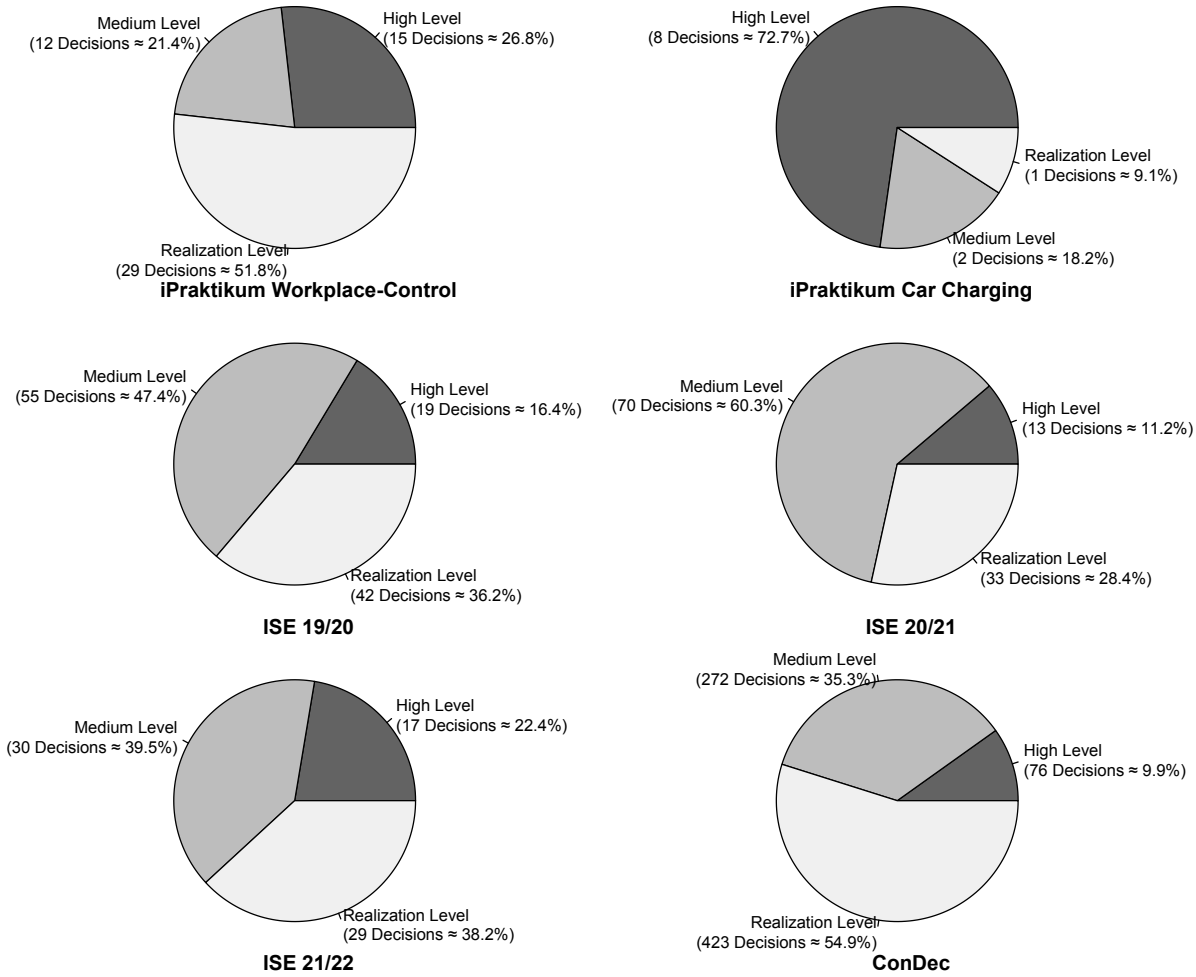

Figure D.6.: Decision level assignments in the validation projects.

Figure D.7 shows the proportion of decision levels documented over time in the validation projects.

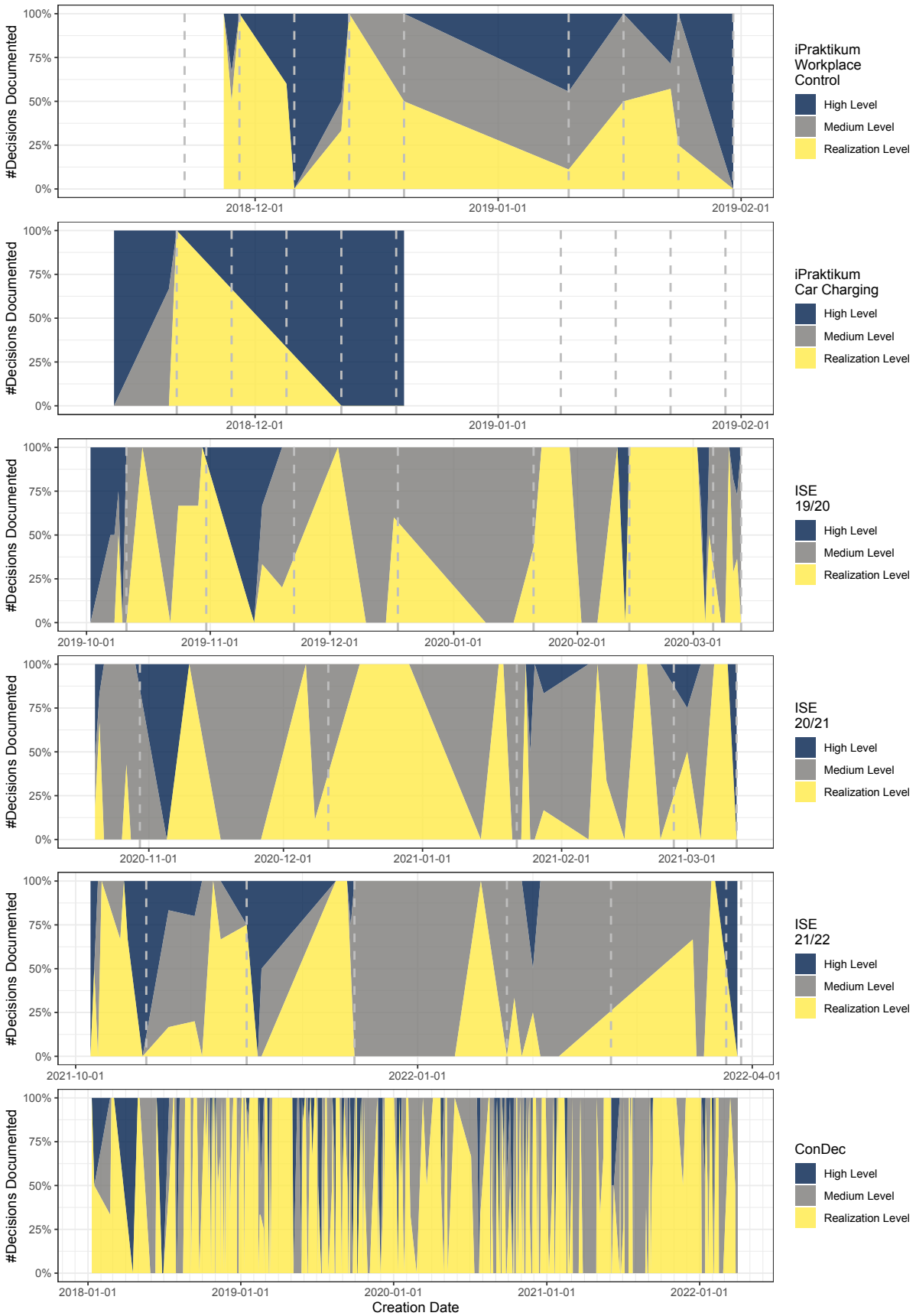

Figure D.7.: Proportion of decision levels documented over time in the validation projects.

Appendix E

Supplementary Material of Text Classifier Validation

"To reduce design documentation effort, we experimented with lean, minimalistic documentation."

—Zdun et al., 2013

This appendix provides supplementary material for the effectiveness evaluation of the automatic text classification in Chapter 10. Table E.1 shows the precision and recall values of the evaluation of the binary and fine-grained classifiers on the data of individual projects using 10-fold cross-validation and for different machine learning algorithms. Table E.2 shows the precision and recall values of the evaluation of the binary and fine-grained classifiers for cross project validation in the three ISE projects and different machine learning algorithms. Table E.3 shows the precision and recall values of the evaluation of the binary and fine-grained classifiers on the combined data of the three ISE projects and the issue tracking data by Alkadhi (2018) using 10-fold cross-validation and for different machine learning algorithms.

Table E.1.: Evaluation results of the binary and fine-grained classifiers on the data of single projects using 10-fold cross validation. The machine-learning algorithms are Logistic Regression (LR), Naïve Bayes (NB), and Support Vector Machine (SVM).

Metric	Project	Alg.	Binary (Relevant/Irrelevant)	Issue	Alternative	Decision	Pro	Con
Precision	ConDec	LR	0.7	0.65	0.31	0.25	0.42	0.5
	ConDec	NB	0.56	0.69	0.33	0.37	0.37	0.31
	ConDec	SVM	0.74	0.56	0.36	0.33	0.43	0.39
	ISE 19/20	LR	0.73	0.67	0.29	0.3	0.36	0.46
	ISE 19/20	NB	0.75	0.64	0.28	0.42	0.38	0.53
	ISE 19/20	SVM	0.79	0.65	0.17	0.3	0.41	0.57
	ISE 20/21	LR	0.77	0.33	0.2	0.46	0.32	0.5
	ISE 20/21	NB	0.68	0.62	0.33	0.43	0.46	0.43
	ISE 20/21	SVM	0.77	0.42	0.18	0.26	0.26	0.32
	ISE 21/22	LR	0.83	0.7	0.47	0.42	0.26	0.31
	ISE 21/22	NB	0.82	0.57	0.28	0.38	0.26	0.25
	ISE 21/22	SVM	0.86	0.69	0.58	0.67	0.28	0.24
	Lucene	LR	0.69	0.22	0.39	0.92	0.65	0.36
	Lucene	NB	0.63	0.45	0.35	0.42	0.4	0.36
	Lucene	SVM	0.56	0.2	0.39	0.96	0.67	0.24
	Thunderbird	LR	0.57	0.39	0.33	0.4	0.3	0.27
	Thunderbird	NB	0.55	0.32	0.24	0.42	0.22	0.24
	Thunderbird	SVM	0.55	0.52	0.41	0.84	0.36	0.25
	Ubuntu	LR	0.52	0.2	0.4	0.25	0.45	0.39
	Ubuntu	NB	0.53	0.12	0.5	0.34	0.26	0.27
	Ubuntu	SVM	0.66	0.31	0.41	0.3	0.24	0.29
Recall	ConDec	LR	0.62	0.28	0.06	0.93	0.06	0.23
	ConDec	NB	0.95	0.42	0.19	0.36	0.15	0.74
	ConDec	SVM	0.55	0.54	0.3	0.5	0.41	0.29
	ISE 19/20	LR	0.72	0.5	0.17	0.63	0.17	0.48
	ISE 19/20	NB	0.66	0.51	0.51	0.6	0.12	0.35
	ISE 19/20	SVM	0.52	0.6	0.07	0.64	0.32	0.42
	ISE 20/21	LR	0.56	0.89	0.01	0.16	0.22	0.58
	ISE 20/21	NB	0.99	0.78	0.75	0.04	0.34	0.29
	ISE 20/21	SVM	0.64	0.75	0.05	0.06	0.51	0.22
	ISE 21/22	LR	0.68	0.46	0.31	0.32	0.07	0.83
	ISE 21/22	NB	0.63	0.51	0.4	0.42	0.36	0.04
	ISE 21/22	SVM	0.76	0.62	0.4	0.31	0.18	0.56
	Lucene	LR	0.67	0.84	0.15	0.25	0.08	0.17
	Lucene	NB	0.68	0.21	0.33	0.6	0.4	0.41
	Lucene	SVM	0.97	0.72	0.24	0.17	0.14	0.12
	Thunderbird	LR	0.89	0.45	0.39	0.4	0.19	0.27
	Thunderbird	NB	0.8	0.19	0.11	0.48	0.56	0.06
	Thunderbird	SVM	0.88	0.42	0.45	0.34	0.19	0.53
	Ubuntu	LR	0.95	0.02	0.12	0.9	0.1	0.33
	Ubuntu	NB	0.87	0.02	0.04	0.21	0.5	0.6
	Ubuntu	SVM	0.64	0.08	0.27	0.58	0.08	0.5

Table E.2.: Evaluation results of the binary and fine-grained classifiers for cross-project validation. The machine-learning algorithms are Logistic Regression (LR), Naïve Bayes (NB), and Support Vector Machine (SVM).

Metric	Training Project	Validation Project	Alg.	Binary	Issue	Alternative	Decision	Pro	Con
Precision	ISE 19/20	ISE 20/21	LR	0.79	0.56	0.21	0.31	0.34	0.4
	ISE 19/20	ISE 20/21	NB	0.66	0.42	0.24	0.19	0.53	0.5
	ISE 19/20	ISE 20/21	SVM	0.53	0.92	0.3	0.34	0.37	0.53
	ISE 19/20	ISE 21/22	LR	0.8	0.48	0	0.32	0.39	0.5
	ISE 19/20	ISE 21/22	NB	0.71	0.38	0.18	0.26	0.31	0.45
	ISE 19/20	ISE 21/22	SVM	0.45	0.83	0.44	0.39	0.39	0.44
	ISE 20/21	ISE 19/20	LR	0.57	1	0	0.78	0.31	0.39
	ISE 20/21	ISE 19/20	NB	0.32	0.57	0.23	0.52	0.51	0.37
	ISE 20/21	ISE 19/20	SVM	0.36	0.96	0.12	0.6	0.32	0.48
	ISE 20/21	ISE 21/22	LR	0.78	0.67	0	1	0.36	0.37
	ISE 20/21	ISE 21/22	NB	0.47	0.35	0.19	0.35	0.38	0.38
	ISE 20/21	ISE 21/22	SVM	0.54	0.95	0.14	0.75	0.38	0.44
	ISE 21/22	ISE 19/20	LR	0.6	1	0.14	0.45	0.3	0.29
	ISE 21/22	ISE 19/20	NB	0.58	0.67	0.21	0.38	0.75	0.45
	ISE 21/22	ISE 19/20	SVM	0.32	0.96	0.25	0.58	0.33	0.35
	ISE 21/22	ISE 20/21	LR	0.69	0.92	0	0.33	0.35	0.38
	ISE 21/22	ISE 20/21	NB	0.67	0.58	0.2	0.19	0.78	0.57
	ISE 21/22	ISE 20/21	SVM	0.56	1	0.27	0.31	0.36	0.46
Recall	ISE 19/20	ISE 20/21	LR	0.14	0.28	0.03	0.1	0.85	0.08
	ISE 19/20	ISE 20/21	NB	0.62	0.46	0.62	0.31	0.08	0.36
	ISE 19/20	ISE 20/21	SVM	0.97	0.3	0.05	0.22	0.87	0.12
	ISE 19/20	ISE 21/22	LR	0.24	0.33	0	0.16	0.9	0.08
	ISE 19/20	ISE 21/22	NB	0.74	0.64	0.45	0.43	0.05	0.29
	ISE 19/20	ISE 21/22	SVM	0.97	0.48	0.1	0.15	0.89	0.1
	ISE 20/21	ISE 19/20	LR	0.27	0.06	0	0.06	0.9	0.27
	ISE 20/21	ISE 19/20	NB	0.98	0.65	0.52	0.3	0.17	0.5
	ISE 20/21	ISE 19/20	SVM	0.92	0.2	0.01	0.05	0.91	0.26
	ISE 20/21	ISE 21/22	LR	0.29	0.14	0	0.01	0.83	0.28
	ISE 20/21	ISE 21/22	NB	1	0.67	0.49	0.21	0.08	0.35
	ISE 20/21	ISE 21/22	SVM	0.92	0.26	0.01	0.04	0.9	0.22
	ISE 21/22	ISE 19/20	LR	0.59	0.08	0.01	0.04	0.83	0.24
	ISE 21/22	ISE 19/20	NB	0.59	0.41	0.59	0.59	0.05	0.38
	ISE 21/22	ISE 19/20	SVM	0.95	0.2	0.03	0.06	0.86	0.27
	ISE 21/22	ISE 20/21	LR	0.54	0.14	0	0.03	0.85	0.21
	ISE 21/22	ISE 20/21	NB	0.56	0.56	0.67	0.28	0.06	0.27
	ISE 21/22	ISE 20/21	SVM	0.94	0.33	0.03	0.03	0.88	0.22

Table E.3.: Evaluation results of the binary and fine-grained classifiers on the combined data of the ISE projects using 10-fold cross validation. The machine-learning algorithms are Logistic Regression (LR), Naïve Bayes (NB), and Support Vector Machine (SVM).

Metric	Data Source	Alg.	Binary (Relevant/Irrelevant)	Issue	Alternative	Decision	Pro	Con
Precision	all ISE	LR	0.77	0.72	0.16	0.31	0.4	0.42
	all ISE	NB	0.6	0.64	0.3	0.36	0.45	0.48
	all ISE	SVM	0.77	0.56	0.32	0.42	0.31	0.5
	R. Alkadhi	LR	0.59	0.23	0.48	0.59	0.5	0.44
	R. Alkadhi	NB	0.54	0.3	0.39	0.54	0.25	0.28
	R. Alkadhi	SVM	0.52	0.21	0.42	0.96	0.53	0.37
Recall	all ISE	LR	0.62	0.58	0.01	0.7	0.28	0.5
	all ISE	NB	0.96	0.59	0.53	0.4	0.19	0.36
	all ISE	SVM	0.39	0.76	0.14	0.2	0.62	0.36
	R. Alkadhi	LR	0.8	0.8	0.19	0.21	0.09	0.26
	R. Alkadhi	NB	0.88	0.1	0.13	0.15	0.53	0.54
	R. Alkadhi	SVM	0.97	0.74	0.26	0.1	0.12	0.19

Appendix F

Supplementary Material of User Acceptance Study

"Many ideas happen to us. We have intuition, we have feeling, we have emotion, all of that happens, we don't decide to do it. We don't control it."

—Kahneman, 2011

This appendix provides supplementary material for the user acceptance validation in Chapter 11. Section F.1 provides the questionnaire for surveying user acceptance. Section F.2 provides detailed Likert ratings used to calculate the weighted means of ratings in Chapter 11.

F.1. Questionnaire for Collecting the User Feedback

This section presents an aggregated questionnaire to survey the user acceptance of ConDec. The questions and statements were specific to the study participants as follows: Questions and statements marked with *D* are specific for (student) developers. Questions and statements marked with *E* are specific for IT experts, i. e., practitioners from the industry. Questions and statements not marked are for study participants who used ConDec over the period of the project.

The questionnaire consists of the following five parts: 1) General Questions I, 2) Questions regarding Fulfillment of ConDec's Design Goals, 3) General Questions II, 4) Usage Frequency of ConDec Features, and 5) Person-related Questions.

Please answer the following questions and rate the following statements regarding continuous rationale management and the ConDec tool support.

To read about the mentioned features below, please see the ConDec feature documentation in GitHub: `https://github.com/cures-hub/cures-condec-jira/tree/develop/doc/features`

1) General Questions I

#	Question/Statement	Answer
1.1	What ConDec features for rationale management were (the most) useful during the project? Why were they useful?	
1.2	What ConDec features for rationale management were not useful during the project? Why were they not useful?	
1.3	In retrospect, documenting our most important decisions did help the communication with the team or with the other stakeholders.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why was it useful, why not? What would have made it more useful?

2) Questions regarding Fulfillment of ConDec's Design Goals

ConDec Design Goal 1: Support collaborative, incremental, and rational decision making

#	Question/Statement	Answer
DG1.1	ConDec is useful for (collaborative, incremental, and rational) decision making.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision. Why is ConDec useful for decision making? Why is ConDec not useful for decision making? What would make it more useful?
DG1.2	I would use ConDec to support (collaborative, incremental, and rational) decision making in the future.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision. Why would you use ConDec to support decision making? Why would you not use ConDec for decision making? How can ConDec be improved so that you would use it in the future?
DG1.3	Do you have other feedback regarding "ConDec Design Goal 1: Support collaborative, incremental, and rational decision making"? Do you have feature requests regarding decision making support.	

ConDec Design Goal 2: Enable easy decision knowledge documentation

#	Question/Statement	Answer
DG2.1 (E)	Which documentation location of rationale do you prefer?	☐ Jira issue description + comments ☐ separate Jira issues with distinct types ☐ commit messages ☐ code comments ☐ I would prefer the following other documentation location: Please justify your decision, e. g., by providing examples. Why do you prefer one of these documentation locations over the other?
DG2.1	Documenting rationale in Jira issue descriptions and comments is easy.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make documenting rationale easier?
DG2.2	Documenting rationale in separate Jira issues is easy.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make documenting rationale easier?
DG2.3	Capturing rationale in Jira is useful (e. g. for decision making and documentation).	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make documenting rationale in Jira more useful?

#	Question/Statement	Answer
DG2.4	Which documentation location of rationale in Jira do you prefer?	☐ Jira issue description + comments ☐ separate Jira issues with distinct types ☐ commit messages ☐ code comments ☐ I would prefer the following other documentation location: Please justify your decision, e. g., by providing examples. Why do you prefer one of these documentation locations over the other? Or in case you selected both: Why do you think both documentation locations are equally useful?
DG2.5	Linking of criteria (quality requirements, such as efficiency, security, and constraints/context factors, such as implementation effort) to arguments in criteria matrix is easy.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make linking criteria easier?
DG2.6	Linking of criteria to arguments in criteria matrix is useful.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? hat would make linking criteria more useful?
DG2.7	Grouping decisions by decision level and decision groups is easy.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make grouping decisions easier? Which decision groups were easy to decide? Why? Which decision groups were difficult to decide? Why?
DG2.8	Grouping decisions by decision level and decision groups is useful.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make grouping decisions more useful? How useful was the grouping of decisions for the current development? How useful is the grouping of decisions for future development?
DG2.9	It is easy to use the automatic text classifier that tries to identify decision knowledge.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make the automatic text classifier easier to use?
DG2.10	The automatic text classifier is useful for decision knowledge documentation.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make the automatic text classifier more useful?
DG2.11	The link recommendation and and duplicate detection feature is easy to use.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make the link recommendation and and duplicate detection feature easier to use?
DG2.12	The link recommendation and duplicate detection feature is useful.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make the link recommendation and duplicate detection feature more useful?
DG2.13	The decision guidance feature (=recommendation of solution options from external knowledge sources) is easy to use.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make the decision guidance feature easier to use?

#	Question/Statement	Answer
DG2.14	The decision guidance feature is useful.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make the decision guidance feature more useful?
DG2.15	Which of the features for decision knowledge documentation do you intend to use in the future?	
DG2.16	Do you have other feedback regarding "ConDec Design Goal 2: Enable easy decision knowledge documentation"? Do you have feature requests regarding easy documentation.	
DG2.17 (E)	ConDec is useful for decision knowledge documentation.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is ConDec useful for decision knowledge documentation? Why is ConDec not useful for decision knowledge documentation? What would make it more useful?
DG2.18 (E)	I would use ConDec to document decision knowledge in the future.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision. Why would you use ConDec to document decision knowledge? Why would you not use ConDec to document decision knowledge? How can ConDec be improved so that you would use it in the future?

ConDec Design Goal 3: Enable easy usage/exploitation of decision knowledge

#	Question/Statement	Answer
DG3.1	Which presentation do you prefer and why? (Which presentation is most useful for you?)	☐ Node-link diagram (graph view, vis.js) ☐ Tree view: indented outline (jstree) ☐ Tree view: node-link tree diagram (treant.js) ☐ Chronology view ☐ Adjacency matrix ☐ Criteria matrix ☐ I would prefer the following other presentation: Please justify your ranking. Please explain what you like and what you do not like about each presentation.
DG3.2	Presenting/visualizing knowledge in Jira is useful.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is the presentation/visualization useful? Why is it not useful? What would make the presentation/visualization more useful?
DG3.3	The ConDec visualizations of the knowledge graph are easy to interact with (e. g. for knowledge management via drag&drop and the context menu).	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make the presentation/visualization easier to interact with?
DG3.4	It is useful that the ConDec visualizations of the knowledge graph are interactive (e. g. for knowledge management via drag&drop and the context menu).	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is the interaction useful? Why is it not useful? What would make the interaction more useful?
DG3.5	Importing rationale into Confluence is easy.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make importing rationale into Confluence easier?

#	Question/Statement	Answer
DG3.6	Presenting rationale in meeting agendas in Confluence is useful (e. g. during meetings and for knowledge sharing).	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is the presentation useful? Why is it not useful? What would make the presentation of rationale in Confluence more useful?
DG3.7	The semi-automatic creation of release notes in Jira is easy to use.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make the semi-automatic creation of release notes easier?
DG3.8	The semi-automatic creation of release notes in Jira is useful (e. g. for knowledge sharing).	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make the semi-automatic creation of release notes more useful?
DG3.9	Change impact analysis through change impact highlighting is easy to perform.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make change impact highlighting easier?
DG3.10	Change impact analysis through change impact highlighting is useful.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is navigating from code to the knowledge graph views useful? Why is it not useful? What would make change impact highlighting more useful?
DG3.11	Navigating from code to the knowledge graph views in Jira (using VSCode or Eclipse ConDec plug-in) is easy.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make the navigation from code to the knowledge graph views easier?
DG3.12	Navigating from code to the knowledge graph views in Jira (using VSCode or Eclipse ConDec plug-in) is useful.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is navigating from code to the knowledge graph views useful? Why is it not useful? What would make navigating from code to the knowledge graph views more useful?
DG3.13	Which of the features for knowledge exploitation do you intend to use in the future?	
DG3.14	Do you have other feedback regarding "ConDec Design Goal 3: Enable easy usage/exploitation of decision knowledge"? Do you have feature requests regarding easy usage/exploitation.	
DG3.15 (E)	ConDec is useful for the exploitation of decision knowledge.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is ConDec useful for decision knowledge exploitation? Why is ConDec not useful for decision knowledge exploitation? What would make it more useful?
DG3.16 (E)	I would use ConDec to document decision knowledge in the future.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision. Why would you use ConDec to exploit decision knowledge? Why would you not use ConDec to exploit decision knowledge? How can ConDec be improved so that you would use it in the future?

ConDec Design Goal 4: Support creating and maintaining high documentation quality

#	Question/Statement	Answer
DG4.1	The dashboard to visualize metrics on the decision knowledge documentation is easy to use.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make the dashboard easier to use?
DG4.2	The dashboard to visualize metrics on the decision knowledge documentation is useful to monitor and improve the documentation quality.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make the dashboard more useful?
DG4.3	Defining and checking a definition of done for the knowledge documentation is useful.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make the definition of done more useful?
DG4.4	The Rationale Backlog showing knowledge elements that violate the definition of done, e. g., incomplete knowledge documentation, unresolved decision problems (=issues) and challenged decisions is easy to use.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make the Rationale Backlog easier to use?
DG4.5	The Rationale Backlog showing knowledge elements that violate the definition of done is useful.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make the Rationale Backlog more useful?
DG4.6	Result presentation of definition of done checking in the quality check view is easy to understand.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make the Quality Check view easier to use?
DG4.7	Result presentation of definition of done checking in the quality check view is useful.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make the Quality Check view more useful?
DG4.8	Coloring user-interface elements (menu items and knowledge elements in the graph views) is easy to use/understand (ambient feedback nudging mechanism).	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make the ambient feedback nudging mechanism easier to use/understand?
DG4.9	Coloring user-interface elements (menu items and knowledge elements in the graph views) is useful.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make the ambient feedback nudging mechanism more useful?
DG4.10	The just-in-time prompt that is shown when changing the state of a Jira issue is easy to use (nudging mechanism).	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make the just-in-time prompt nugding mechanism easier to use/understand?

#	Question/Statement	Answer
DG4.11	The just-in-time prompt that is shown when changing the state of a Jira issue is useful (nudging mechanism).	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make the just-in-time prompt nudging mechanism more useful?
DG4.12	Keeping the documented rationale complete is easy. For example, completeness means that important issues and decisions are both captured.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make it easier to keep the documented rationale complete?
DG4.13	Keeping the documented rationale consistent with the other artifacts (requirements, design, code, tests, ...) is easy. For example, consistency means that the documentation is up-to-date.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make it easier to keep the documented rationale consistent?
DG4.14	Which of the features for creating and maintaining high documentation quality do you intend to use in the future?	
DG4.15	Do you have other feedback regarding "ConDec Design Goal 4: Support creating and maintaining high documentation quality"? Do you have feature requests regarding high quality.	
DG4.16 (E)	The rationale documented during the project has a high quality.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision. Why do you think that the quality is high or low? What could have improved the quality?
DG4.17 (E)	ConDec is useful for creating and maintaining high documentation quality.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is ConDec useful for creating and maintaining high documentation quality? Why is ConDec not useful for creating and maintaining high documentation quality? What would make it more useful?
DG4.18 (E)	I would use ConDec to create and maintain high documentation quality in the future.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision. Why would you use ConDec to create and maintain high documentation quality? Why would you not use ConDec to create and maintain high documentation quality? How can ConDec be improved so that you would use it in the future?

ConDec Design Goal 5: Enable documentation and exploitation of a high amount of knowledge

#	Question/Statement	Answer
DG5.1	Filtering the knowledge graph views in Jira is easy.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make filtering the knowledge graph views easier?
DG5.2	Filtering the knowledge graph views in Jira is useful.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make filtering the knowledge graph views more useful?

#	Question/Statement	Answer
DG5.3	Exploiting transitive links (to e. g. only see decisions for epics) in Jira is easy.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make the exploitation of transitive links easier?
DG5.4	Exploiting transitive links (to e. g. only see decisions for epics) in Jira is useful.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make the exploitation of transitive links more useful?
DG5.5	Marking of wrong/unuseful links so that they are excluded from presentation and knowledge graph traversal is easy.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. What is easy and why is it easy? What is difficult and why is it difficult? What would make marking of wrong/unuseful links easier?
DG5.6	Marking of wrong/unuseful links so that they are excluded from presentation and knowledge graph traversal is useful.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Why is it useful? Why is it not useful? What would make marking of wrong/unuseful links more useful?
DG5.7	Which of the features for supporting a high amount of documentation and exploitation do you intend to use in the future?	
DG5.8	Do you have other feedback regarding "ConDec Design Goal 5: Enable documentation and exploitation of a high amount of knowledge"? Do you have feature requests regarding high amount.	

3) General Questions II

#	Question/Statement
3.1	What other features do you expect for rationale management that are not implemented?
3.2 (E)	How useful could the rationale documentation be for the future of the project?
3.3 (E)	What exploitation/usage scenarios for the rationale documentation do you see?
3.4	What change scenario(s) can you imagine for the project in that the documented rationale would be very useful?
3.5 (E)	What documented decisions will be the most useful for the future development? Why will they be useful?
3.6	What other feedback do you have?

4) Usage Frequency of ConDec Features

Please provide an estimation how often you applied each ConDec feature.

Feature	Please select one option.			
Documentation of decision knowledge as entire Jira tickets	☐ not at all (0 decisions)	☐ rarely (1-3 decisions)	☐ sometimes (3-10 decisions)	☐ often (>10 decisions)
Documentation of decision knowledge in description and comments of Jira tickets	☐ not at all (0 decisions)	☐ rarely (1-3 decisions)	☐ sometimes (3-10 decisions)	☐ often (>10 decisions)
Documentation of decision knowledge in commit message using annotations during committing or afterward in Jira comment	☐ not at all (0 decisions)	☐ rarely (1-3 decisions)	☐ sometimes (3-10 decisions)	☐ often (>10 decisions)

Feature	**Please select one option.**			
Documentation of decision knowledge in code comments using annotations	☐ not at all (0 decisions)	☐ rarely (1-3 decisions)	☐ sometimes (3-10 decisions)	☐ often (>10 decisions)
Changing elements and links through interaction with the views of the knowledge graph via context menu or drag & drop	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Linking arguments to criteria in criteria matrix	☐ not at all (0 decisions)	☐ rarely (1-3 decisions)	☐ sometimes (3-10 decisions)	☐ often (>10 decisions)
Link recommendation and duplicate detection	☐ not at all (0 decisions)	☐ rarely (1-3 decisions)	☐ sometimes (3-10 decisions)	☐ often (>10 decisions)
Recommendation of solution options from knowledge sources (decision guidance)	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Automatic text classification to identify decision knowledge in Jira ticket text	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Decision grouping	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Knowledge dashboards with metrics	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Rationale backlog showing knowledge elements that violate the definition of done	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
definition of done checking result presentation in the quality check view	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Nudging mechanisms: ambient feedback and just-in-time prompt	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Stand-up table with decision knowledge in Confluence	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Semi-automatic release notes creation including decision knowledge	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Tree view: indented outline	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Tree view: node-link tree diagram	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Node-link diagram (graph view)	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Chronology	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Criteria matrix (also called decision table)	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Adjacency matrix	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
View for decision knowledge from git	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Change impact analysis	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)

Feature	Please select one option.			
Filtering and transitive linking in the views on the knowledge graph	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Marking links as wrong or useless	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)
Navigation from code in integrated development environment to knowledge graph view in Jira	☐ not at all (0 times)	☐ rarely (1-3 times)	☐ sometimes (3-10 times)	☐ often (>10 times)

5) Person-related Questions

#	Question/Statement	Answer
5.1	Before the project/workshop, I was experienced with rationale management.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Which experiences did you have regarding rationale management before the project/workshop?
5.2 (D)	Before the project, I was experienced in developing software for a customer.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Which experiences did you have regarding developing software for a customer before the project?
5.3 (D)	Before the project, I was experienced in developing software in a team.	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Which experiences did you have regarding developing software in a team before the project?
5.4 (E)	How long have you been working in the field of software engineering?	
5.5 (E)	I am experienced with Continuous Software Engineering (CSE).	☐ strongly disagree ☐ disagree ☐ neutral ☐ agree ☐ strongly agree Please justify your decision, e. g., by providing examples. Which experiences do you have regarding CSE?
5.6 (E)	What roles do you normally have?	
5.7 (E)	In how many projects have you documented rationale yourself?	
5.8 (E)	In how many projects have you exploited rationale documented by others?	

F.2. Detailed Ratings by Study Participants

We collected 16 filled-out questionnaires during the iPraktikum at the Technical University of Munich: six by the workplace-control app and ten by the car charging app developers. Table F.1 shows the answers from a five point Likert scale.

Table F.1.: Study participants' assessment of ConDec and the weighted means μ_w (iPraktikum).

ConDec Statement	Project	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	μ_w
Documenting rationale in Jira [as entire tickets] is easy.	iPraktikum	0	1	2	12	1	0.8
Documenting rationale in Jira is useful for decision making.	iPraktikum	2	2	6	4	2	0.1
I would document rationale in Jira in future projects.	iPraktikum	2	2	5	7	0	0.1
Presenting/visualizing rationale in Jira is useful.	iPraktikum	0	2	3	8	3	0.8
Searching and filtering the presented knowledge is easy.	iPraktikum	0	4	9	1	2	0.1
Presenting rationale in meeting agendas in Confluence is useful.	iPraktikum	1	2	6	4	3	0.4
Keeping the documented rationale consistent, complete, and correct is easy.	iPraktikum	0	8	2	6	0	-0.1
The documented rationale during this sprint is of high quality. It is complete, consistent, and correct.	iPraktikum	2	3	9	2	0	-0.3
Importing rationale into Confluence is easy.	iPraktikum	0	2	5	6	3	0.6
I would apply rationale-based meeting management in future projects.	iPraktikum	1	2	3	8	2	0.5

Table F.2 lists the answers by the study participants of two validation projects on how often they used the ConDec features. In the ISE 19/20 project, we did not ask for the usage frequencies of features that were not yet existing or mature.

Table F.3 lists the answers by the study participants of the validation projects on their perceived ease of use of the ConDec features.

Table F.2.: Study participants' answers on their usage frequencies of ConDec features and the weighted means μ_w.

ConDec Feature . . . how often used?	Project	Never	Rarely	Sometimes	Often	μ_w
Documentation of Decision Knowledge in . . .						
Description and comments of Jira tickets	ISE 19/20	0	0	3	4	1.6
	ISE 21/22 D	0	2	3	1	0.5
Entire Jira tickets	ISE 19/20	1	5	1	0	-0.9
	ISE 21/22 D	0	0	0	6	2
Commit messages using annotations during committing or afterwards in Jira comment	ISE 21/22 D	5	1	0	0	-1.8
Code comments using annotations	ISE 21/22 D	4	2	0	0	-1.7
Chat messages and exporting it to Jira	ISE 19/20	1	6	0	0	-1.1
Other Documentation Features						
Decision grouping	ISE 21/22 D	1	1	2	2	0.5
Automatic text classification to identify decision knowledge in Jira ticket text	ISE 19/20	0	6	1	0	-0.7
	ISE 21/22 D	2	4	0	0	-1.3
Link recommendation and duplicate detection	ISE 21/22 D	0	4	1	1	-0.2
Recommendation of solution options from external knowledge sources (decision guidance)	ISE 21/22 D	5	0	1	0	-1.5
Changing elements and links through interaction with views of knowledge graph	ISE 19/20	0	2	2	3	0.9
	ISE 21/22 D	0	2	3	1	0.5
Linking arguments to criteria in criteria matrix	ISE 21/22 D	0	0	1	5	1.8
Knowledge Graph Views						
Node-link diagram, V1	ISE 21/22 D	0	5	1	0	-0.7
Tree: indented outline, $V2_{ind}$	ISE 21/22 D	0	2	2	2	0.7
Tree: node-link tree diagram, $V2_{nld}$	ISE 21/22 D	1	1	1	3	0.7
Adjacency matrix, $V4_{adj}$	ISE 21/22 D	1	4	1	0	-0.8
Criteria matrix, $V4_{cri}$	ISE 21/22 D	0	0	1	5	1.8
Chronology view, V5	ISE 21/22 D	5	1	0	0	-1.8
Other Knowledge Exploitation Features						
Stand-up table with decision knowledge in Confluence	ISE 21/22 D	0	5	1	0	-0.7
Semi-automatic release notes creation including decision knowledge	ISE 21/22 D	0	5	1	0	-0.7
Change impact highlighting	ISE 21/22 D	3	3	0	0	-1.5
Navigation from code to knowledge graph view in Jira using VSCode or Eclipse ConDec	ISE 21/22 D	6	0	0	0	-2
Filtering the views on the knowledge graph	ISE 21/22 D	0	3	3	0	0
Exploiting transitive links	ISE 21/22 D	0	3	3	0	0
Decision knowledge presentation in pull requests	ISE 19/20	0	5	1	1	-0.3
View for decision knowledge from git	ISE 21/22 D	6	0	0	0	-2
Quality Assurance Features						
Knowledge dashboard with metrics, V6	ISE 21/22 D	0	2	1	3	0.8
Rationale backlog showing knowledge elements that violate the definition of done	ISE 21/22 D	0	1	3	2	1
definition of done checking result presentation in the quality check view	ISE 21/22 D	0	2	2	2	0.7
Just-in-time prompt nudging mechanism	ISE 21/22 D	2	1	1	2	0
Ambient feedback nudging mechanisms: coloring menu items and knowledge elements	ISE 21/22 D	2	1	1	2	0
Marking links as wrong or useless	ISE 21/22 D	2	2	2	0	-0.7
Merge check of decision knowledge in pull requests	ISE 19/20	0	5	1	1	-0.3

Table F.3.: Study participants' answers on their perceived ease of use of ConDec features and the weighted means μ_w.

ConDec Feature ... is easy to use?	**Project**	**Strongly Disagree**	**Disagree**	**Neutral**	**Agree**	**Strongly Agree**	μ_w
It is easy to document decision knowledge with ConDec.	ISE 21/22 C	0	0	0	2	0	1
	Workshop	0	0	4	0	0	0
It is easy to use/exploit decision knowledge with ConDec.	ISE 21/22 C	0	0	0	1	0	1
	Workshop	0	0	2	2	0	0.5
It is easy to create and maintain high documentation quality with ConDec.	ISE 21/22 C	0	0	2	0	0	0
	Workshop	0	1	2	1	0	0
Documentation of Decision Knowledge in ...							
Description and comments of Jira tickets	ISE 19/20	0	1	2	4	0	0.4
	ISE 21/22 D	0	0	1	3	2	1.2
Entire Jira tickets	ISE 19/20	0	1	5	1	0	0
	ISE 21/22 D	0	0	0	3	3	1.5
Chat messages and exporting it to Jira	ISE 19/20	0	1	6	0	0	-0.1
Other Documentation Features							
Decision grouping	ISE 21/22 D	0	0	0	4	2	1.3
Automatic text classification to identify decision knowledge in Jira ticket text	ISE 19/20	1	0	4	2	0	0
	ISE 21/22 D	0	0	3	3	0	0.5
Link recommendation and duplicate detection	ISE 21/22 D	0	0	1	1	4	1.5
Recommendation of solution options from knowledge sources (decision guidance)	ISE 21/22 D	0	0	1	4	1	1
Changing elements and links through interaction with views	ISE 19/20	0	0	3	4	0	0.6
	ISE 21/22 D	0	0	2	3	1	0.8
Linking arguments to criteria in criteria matrix	ISE 21/22 D	0	0	3	2	1	0.7
Knowledge Exploitation Features							
Stand-up table with decision knowledge in Confluence	ISE 19/20	0	0	5	2	0	0.3
	ISE 21/22 D	0	0	0	5	1	1.2
Semi-automatic release notes creation including decision knowledge	ISE 21/22 D	0	0	0	5	1	1.2
Change impact highlighting	ISE 21/22 D	0	0	3	2	1	0.7
Navigation from code to knowledge graph view in Jira	ISE 21/22 D	0	0	2	2	2	1
Filtering the views on the knowledge graph	ISE 21/22 D	0	0	2	3	1	0.8
Exploiting transitive links	ISE 21/22 D	0	0	2	3	1	0.8
Decision knowledge presentation in pull requests	ISE 19/20	0	2	2	3	0	0.1
Quality Assurance Features							
Knowledge dashboard with metrics, V6	ISE 21/22 D	0	0	1	2	3	1.3
Rationale backlog showing knowledge elements that violate the definition of done	ISE 21/22 D	0	0	1	5	0	0.8
Result presentation of definition of done checking in the quality check view	ISE 21/22 D	0	0	1	1	4	1.5
Ambient feedback nudging: coloring menu items and knowledge elements	ISE 21/22 D	0	0	1	1	4	1.5
Just-in-time prompt nudging mechanism	ISE 21/22 D	0	0	2	3	1	0.8
Marking links as wrong or useless	ISE 21/22 D	0	1	2	3	0	0.3
Merge check of decision knowledge in pull requests	ISE 19/20	1	0	1	4	1	0.6

Table F.4 lists the answers by the study participants of the validation projects on their perceived usefulness of the ConDec features.

Table F.4.: Study participants' answers on their perceived usefulness of ConDec features and the weighted means μ_w.

ConDec Feature ... is useful?	**Project**	**Strongly Disagree**	**Disagree**	**Neutral**	**Agree**	**Strongly Agree**	μ_w
ConDec is useful for decision making.	ISE 21/22 D	0	0	0	3	3	1.5
	ISE 21/22 C	0	0	0	2	0	1
	Workshop	0	1	0	1	2	1
Documentation of Decision Knowledge in ...							
Jira (in description and comments of tickets or as entire tickets)	ISE 19/20	0	1	3	3	0	0.3
	ISE 21/22 D	0	0	0	2	4	1.7
Commit messages using annotations during committing or afterward in Jira comment	ISE 21/22 D	0	2	3	2	0	0
Code comments using annotations	ISE 21/22 D	1	0	1	5	0	0.4
Chat messages and exporting it to Jira	ISE 19/20	0	3	2	2	0	-0.1
Wiki pages (not supported)	ISE 19/20	0	1	2	3	1	0.6
Pull requests (not supported)	ISE 19/20	0	1	4	1	1	0.3
Other Documentation Features							
Decision grouping	ISE 21/22 D	0	0	2	2	2	1
	ISE 21/22 C	0	0	0	1	1	1.5
Automatic text classification to identify decision knowledge in Jira ticket text	ISE 19/20	1	0	5	0	1	0
	ISE 21/22 D	2	1	2	1	0	-0.7
	ISE 21/22 C	0	0	1	0	1	1
Link recommendation and duplicate detection	ISE 21/22 D	1	1	2	1	1	0
	ISE 21/22 C	0	0	0	1	0	1
Recommendation of solution options from knowledge sources (decision guidance)	ISE 21/22 D	0	0	5	1	0	0.2
	ISE 21/22 C	0	0	0	0	1	2
Changing elements and links through interaction with views	ISE 19/20	0	0	2	4	1	0.9
	ISE 21/22 D	0	0	1	4	1	1
	ISE 21/22 C	0	0	0	1	1	1.5
Linking arguments to criteria in criteria matrix	ISE 21/22 D	0	0	4	2	1	0.6
	ISE 21/22 C	0	0	0	0	2	2
ConDec is useful for decision knowledge documentation.	ISE 21/22 C	0	0	0	2	0	1
	Workshop	0	0	0	2	2	1.5
Knowledge Exploitation Features							
Presenting/visualizing knowledge in Jira	ISE 19/20	0	0	2	4	1	0.9
	ISE 21/22 D	0	0	0	2	4	1.7
Stand-up table with decision knowledge in Confluence	ISE 21/22 D	0	2	0	3	1	0.5
	ISE 21/22 C	0	0	1	1	0	0.5
Semi-automatic release notes creation including decision knowledge	ISE 21/22 D	0	0	0	5	1	1.2
	ISE 21/22 C	1	0	1	0	0	-1
Change impact highlighting	ISE 21/22 D	0	0	4	2	0	0.3
	ISE 21/22 C	0	0	1	1	0	0.5
Navigation from code to knowledge graph view in Jira	ISE 21/22 D	0	1	5	0	0	-0.2
Filtering the views on the knowledge graph	ISE 21/22 D	0	0	0	2	4	1.7
	ISE 21/22 C	0	0	1	0	1	1
Exploiting transitive links	ISE 21/22 D	0	0	1	4	1	1
	ISE 21/22 C	0	0	1	1	0	0.5

Continued on next page

ConDec Feature . . . is useful?	Project	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	μ_w
Decision knowledge presentation in pull requests	ISE 19/20	0	2	2	2	1	0.3
ConDec is useful to use/exploit decision knowledge in context to other knowledge.	ISE 21/22 C	0	0	0	1	0	1
	Workshop	0	0	2	2	0	0.5
Quality Assurance Features							
Defining and checking of a definition of done for the knowledge documentation	ISE 21/22 D	1	1	0	4	0	0.2
	ISE 21/22 C	0	0	0	1	1	1.5
Knowledge dashboard with metrics, V6	ISE 21/22 D	0	0	0	3	3	1.5
	ISE 21/22 C	0	0	0	2	0	1
Rationale backlog showing knowledge elements that violate the definition of done	ISE 21/22 D	0	0	0	2	4	1.7
	ISE 21/22 C	0	0	1	0	1	1
Result presentation of definition of done checking in the quality check view	ISE 21/22 D	0	0	2	2	2	1
Ambient feedback nudging mechanisms: coloring menu items and knowledge elements	ISE 21/22 D	0	0	1	1	4	1.5
Just-in-time prompt nudging mechanism	ISE 21/22 D	0	1	2	0	3	0.8
Marking links as wrong or useless	ISE 21/22 D	1	0	2	3	0	0.2
	ISE 21/22 C	0	0	1	1	0	0.5
Merge check of decision knowledge in pull requests	ISE 19/20	1	3	0	3	0	-0.3
ConDec is useful to create and maintain high documentation quality.	ISE 21/22 C	0	0	0	2	0	1
	Workshop	1	0	1	2	0	0

Table F.5 lists the answers by the study participants of the validation projects on their intention to use the ConDec features in the future.

Table F.5.: Study participants' answers on their intention to use ConDec in the future and the weighted means μ_w.

ConDec Statement	Project	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	μ_w
I would use ConDec to support decision making in the future.	ISE 21/22 D	0	0	1	4	1	1
	ISE 21/22 C	0	0	1	1	0	0.5
	Workshop	0	0	3	1	0	0.2
I would use ConDec to document decision knowledge in the future.	ISE 21/22 C	0	0	2	0	0	0
	Workshop	0	0	3	1	0	0.2
I would use ConDec to exploit rationale in context to other knowledge in future.	ISE 21/22 C	0	0	0	1	0	1
	Workshop	0	0	3	1	0	0.2
I would use ConDec to create and maintain high documentation quality in the future.	ISE 21/22 C	0	0	2	0	0	0
	Workshop	0	0	4	0	0	0

Bibliography

Al Safwan, K., Elarnaoty, M., and Servant, F. (2022). "Developers' need for the rationale of code commits: An in-breadth and in-depth study". In: *Journal of Systems and Software* 189, p. 111320. DOI: 10.1016/j.jss.2022.111320.

Alexeeva, Z., Perez-Palacin, D., and Mirandola, R. (2016). "Design Decision Documentation: A Literature Overview". In: *Software Architecture*. Ed. by B. Tekinerdogan. Vol. 5292. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 84–101. DOI: 10.1007/978-3-319-48992-6_6.

Alkadhi, R. (2018). "Rationale in Developers' Communication". Dissertation. Technical University of Munich, Germany.

Alkadhi, R., Johanssen, J. O., Guzman, E., and Bruegge, B. (2017a). "REACT: An Approach for Capturing Rationale in Chat Messages". In: *11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'17)*. Toronto, Ontario, Canada: IEEE. DOI: 10.1109/esem.2017.26.

Alkadhi, R., Laţa, T., Guzman, E., and Bruegge, B. (2017b). "Rationale in Development Chat Messages: An Exploratory Study". In: *14th International Conference on Mining Software Repositories*. MSR '17. Buenos Aires, Argentina: IEEE Press, pp. 436–446. DOI: 10.1109/msr.2017.43.

Alkadhi, R., Nonnenmacher, M., Guzman, E., and Bruegge, B. (2018). "How do developers discuss rationale?" In: *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Campobasso, Italy: IEEE, pp. 357–369. DOI: 10.1109/saner.2018.8330223.

Aman, V. (2019). "Unterstützung der Konsistenz zwischen Entscheidungen und ihrer Umsetzung durch Zusammenfassung von Codeänderungen". Bachelor Thesis. Heidelberg University.

Ampatzoglou, A., Bibi, S., Avgeriou, P., and Chatzigeorgiou, A. (2020). "Guidelines for Managing Threats to Validity of Secondary Studies in Software Engineering". In: *Contemporary Empirical Methods in Software Engineering*. Vol. 106. Cham: Springer International Publishing, pp. 415–441. DOI: 10.1007/978-3-030-32489-6_15.

Anders, M. (2020). "Comprehensive and Targeted Access to and Visualization of Decision Knowledge". Master Thesis. Heidelberg University. DOI: 10.11588/heidok.00029025.

Arnold, R. S. and Bohner, S. A. (1993). "Impact Analysis - Towards A Framework for Comparison". In: *International Conference on Software Maintenance (ICSM)*. Montreal, Quebec, Canada: IEEE, pp. 292–301. DOI: 10.1109/ICSM.1993.366933.

Babar, M. A., de Boer, R. C., Dingsoyr, T., and Farenhorst, R. (2007). "Architectural Knowlege Management Strategies: Approaches in Research and Industry". In: *Second Workshop on Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent (SHARK/ADI'07: ICSE Workshops 2007)*. Minneapolis, MN, USA: IEEE, p. 7. DOI: 10.1109/SHARK-ADI.2007.3.

Babar, M. A., Dingsøyr, T., Lago, P., and van Vliet, H. (2009). *Software Architecture Knowledge Management: Theory and Practice*. Berlin, Heidelberg: Springer, p. 279. DOI: 10.1007/978-3-642-02374-3.

Bacher, I., Namee, B. M., and Kelleher, J. D. (2016). "On Using Tree Visualisation Techniques to Support Source Code Comprehension". In: *Working Conference on Software Visualization (VISSOFT)*. Raleigh, NC, USA: IEEE, 91–95. DOI: 10.1109/VISSOFT.2016.8.

Basili, V. R., Caldiera, G., and Rombach, D. H. (1994). "The Goal Question Metric Approach". In: *Encyclopedia of Software Engineering*. New York, NY, USA: John Wiley & Sons, Inc., pp. 528–532.

Bass, L., Clements, P., and Kazman, R. (2003). *Software architecture in practice*. Boston, US: Addison-Wesley.

Baum, J. (2021). "Support for Rationale Management with Nudging". Bachelor Thesis. Heidelberg University.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). *Manifesto for Agile Software Development*. URL: http://agilemanifesto.org.

Bendl, L. (2022). "Change Impact Analysis for Issue Tracking Systems". Bachelor Thesis. Heidelberg University.

Berry, D. M. (2017). "Evaluation of Tools for Hairy Requirements and Software Engineering Tasks". In: *25th International Requirements Engineering Conference Workshops (REW)*. Lisbon, Portugal: IEEE, pp. 284–291. DOI: 10.1109/REW.2017.25.

Bhat, M. (2020). "Tool support for architectural decision making in large software intensive agile projects". Dissertation. Technical University of Munich, Germany.

Bhat, M., Shumaiev, K., Biesdorf, A., Hassel, M., Hohenstein, U., and Matthes, F. (2017a). "An ontology-based approach for software architecture recommendations". In: *Twenty-third Americas Conference on Information Systems (AMCIS)*. Boston, Massachusetts, USA, p. 10.

Bhat, M., Shumaiev, K., Biesdorf, A., Hohenstein, U., and Matthes, F. (2017b). "Automatic Extraction of Design Decisions from Issue Management Systems: A Machine Learning Based Approach". In: *11th European Conference on Software Architecture (ECSA'17)*. Ed. by A. Lopes and R. de Lemos. Cham, Switzerland: Springer, pp. 138–154. DOI: 10.1007/978-3-319-65831-5_10.

Bhat, M., Shumaiev, K., Koch, K., Hohenstein, U., Biesdorf, A., and Matthes, F. (2018). "An Expert Recommendation System for Design Decision Making: Who Should be Involved in Making a Design Decision?" In: *International Conference on Software Architecture (ICSA)*. Seattle, WA, USA: IEEE, pp. 85–8509. DOI: 10.1109/ICSA.2018.00018.

Bhat, M., Tinnes, C., Shumaiev, K., Biesdorf, A., Hohenstein, U., and Matthes, F. (2019). "ADeX: A Tool for Automatic Curation of Design Decision Knowledge for Architectural Decision Recommendations". In: *International Conference on Software Architecture Companion (ICSA-C)*. Hamburg, Germany: IEEE, pp. 158–161. DOI: 10.1109/ICSA-C.2019.00035.

Boerner, M. (2021). "Qualitätssicherung von dokumentiertem Wissen mithilfe der Erhebung der Entscheidungsabdeckung". Bachelor Thesis. Heidelberg University.

Bosch, J. (2014). *Continuous Software Engineering: An Introduction*. Springer. DOI: 10.1007/978-3-319-11283-1.

Boz Kumru, Ö. (2019). "Analyse und Klassifikation von Entscheidungswissen in Jira-Issues". Bachelor Thesis. Heidelberg University.

Brandner, K. and Weinreich, R. (2019). "A Recommender System for Software Architecture Decision Making". In: *13th European Conference on Software Architecture (ECSA)*. Vol. 2. Paris, France: ACM, pp. 22–25. DOI: 10.1145/3344948.3344959.

Bratthall, L., Johansson, E., and Regnell, B. (2000). "Is a Design Rationale Vital when Predicting Change Impact? – A Controlled Experiment on Software Architecture Evolution". In: *2nd International Conference on Product Focused Software Process Improvement (PROFES)*. Vol. LNCS 1840. Oulu, Finland: Springer, pp. 126–139. DOI: 10.1007/978-3-540-45051-1_14.

Brooks, F. P. (1996). "The computer scientist as toolsmith II". In: *Communications of the ACM (CACM)* 39.3, pp. 61–68. DOI: 10.1145/227234.227243.

Brown, C. (2019). "Digital Nudges for Encouraging Developer Actions". In: *41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. Montreal, QC, Canada: IEEE, pp. 202–205. DOI: 10.1109/ICSE-Companion.2019.00082.

Brown, W. H., Malveau, R. C., McCormick, H. W. S., and Mowbray, T. J. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc.

Bruegge, B. and Dutoit, A. H. (2010). *Object-Oriented Software Engineering: Using UML, Patterns, and Java*. 3rd ed. Upper Sandle River, NJ, USA: Pearson Prentice Hall, p. 800.

Bruegge, B., Krusche, S., and Alperowitz, L. (2015). "Software Engineering Project Courses with Industrial Clients". In: *ACM Transactions on Computing Education* 15.4, 17:1–17:31.

Brunet, J., Murphy, G. C., Terra, R., Figueiredo, J., and Serey, D. (2014). "Do developers discuss design?" In: *11th Working Conference on Mining Software Repositories - MSR 2014*. Hyderabad, India: ACM, pp. 340–343. DOI: 10.1145/2597073.2597115.

Buchgeher, G. and Weinreich, R. (2011). "Automatic Tracing of Decisions to Architecture and Implementation". In: *Ninth Working IEEE/IFIP Conference on Software Architecture (WICSA)*. Boulder, CO, USA: IEEE, pp. 46–55. DOI: 10.1109/WICSA.2011.16.

Burge, J. E. (2008). "Design rationale: Researching under uncertainty". In: *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 22.4, pp. 311–324. DOI: 10.1017/S0890060408000218.

Burge, J. E. and Brown, D. C. (2008a). "Software Engineering Using RATionale". In: *Journal of Systems and Software* 81.3, pp. 395–413. DOI: 10.1016/j.jss.2007.05.004.

Burge, J. E. and Brown, D. C. (2008b). "SEURAT: Integrated Rationale Management". In: *International Conference on Software Engineering (ICSE)*. Leipzig: IEEE, pp. 835–838. DOI: 10.1145/1368088.1368215.

Burge, J. E., Carroll, J. M., McCall, R., and Mistrik, I. (2008). *Rationale-Based Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, p. 316. DOI: 10.1007/978-3-540-77583-6.

Capilla, R., Dueñas, J. C., and Nava, F. (2010). "Viability for codifying and documenting architectural design decisions with tool support". In: *Journal of Software Maintenance and Evolution: Research and Practice* 22.2, pp. 81–119. DOI: 10.1002/smr.419.

Capilla, R., Jansen, A., Tang, A., Avgeriou, P., and Babar, M. A. (2016). "10 years of software architecture knowledge management: Practice and future". In: *Journal of Systems and Software* 116, pp. 191–205. DOI: 10.1016/j.jss.2015.08.054.

Capilla, R., Jolak, R., Chaudron, M. R. V., and Carrillo, C. (2020a). "Design Decisions by Voice: The Next Step of Software Architecture Knowledge Management". In: *Human-Centered Software Engineering (HCSE)*. Vol. 12481 LNCS. Springer, pp. 166–177. DOI: 10.1007/978-3-030-64266-2_10.

Capilla, R., Nava, F., and Carrillo, C. (2008). "Effort Estimation in Capturing Architectural Knowledge". In: *23rd IEEE/ACM International Conference on Automated Software Engineering*. L'Aquila, Italy: IEEE, pp. 208–217. DOI: 10.1109/ASE.2008.31.

Capilla, R., Nava, F., Pérez, S., and Dueñas, J. C. (2006). "A Web-based Tool for Managing Architectural Design Decisions". In: *ACM SIGSOFT Software Engineering Notes* 31.5. DOI: 10.1145/1163514.1178644.

Capilla, R., Zimmermann, O., Carrillo, C., and Astudillo, H. (2020b). "Teaching Students Software Architecture Decision Making". In: *Software Architecture (European Conference*

on Software Architecture). Vol. 12292 LNCS. September. Cham: Springer, pp. 231–246. DOI: 10.1007/978-3-030-58923-3_16.

Caraban, A., Karapanos, E., Gonçalves, D., and Campos, P. (2019). "23 Ways to Nudge: A Review of Technology-Mediated Nudging in Human-Computer Interaction". In: *Conference on Human Factors in Computing Systems (CHI)*. Glasgow, UK: ACM, pp. 1–15. DOI: 10.1145/3290605.3300733.

Caraban, A., Konstantinou, L., and Karapanos, E. (2020). "The Nudge Deck: A Design Support Tool for Technology-Mediated Nudging". In: *Designing Interactive Systems Conference*. Eindhoven, Netherlands: ACM, pp. 395–406. DOI: 10.1145/3357236.3395485.

Carrillo, C. and Capilla, R. (2018). "Ripple Effect to Evaluate the Impact of Changes in Architectural Design Decisions". In: *12th European Conference on Software Architecture (ECSA'18)*. Madrid, Spain: ACM, pp. 1–8. DOI: 10.1145/3241403.3241446.

Chacon, S. and Straub, B. (2014). *Pro Git: Everything you need to know about Git*. Ed. by L. Corrigan. 2nd ed. New York, USA: Apress, p. 441. DOI: 10.1007/978-1-4842-0076-6.

Chawla, N. V., Japkowicz, N., and Kotcz, A. (2004). "Editorial: Special Issue on Learning from Imbalanced Data Sets". In: *ACM SIGKDD Explorations Newsletter* 6.1, pp. 1–6. DOI: 10.1145/1007730.1007733.

Chazette, L., Brunotte, W., and Speith, T. (2021). "Exploring Explainability: A Definition, a Model, and a Knowledge Catalogue". In: *29th International Requirements Engineering Conference (RE)*. Notre Dame, IN, USA: IEEE, pp. 197–208. DOI: 10.1109/RE51729.2021.00025.

Clarke, A. C. (1962). *Profiles of the Future: An Inquiry into the Limits of the Possible*. The Orion Publishing Group Ltd.

Cleland-Huang, J., Mirakhorli, M., Czauderna, A., and Wieloch, M. (2013). "Decision-Centric Traceability of architectural concerns". In: *7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*. San Francisco, CA, USA: IEEE, pp. 5–11. DOI: 10.1109/TEFSE.2013.6620147.

Clormann, J. (2018). "DecXtract: Dokumentation und Nutzung von Entscheidungswissen in Jira-Issue-Kommentaren". Master Thesis. Heidelberg University. DOI: 10.11588/heidok.00026059.

Codoban, M., Ragavan, S. S., Dig, D., and Bailey, B. (2015). "Software history under the lens: A study on why and how developers examine it". In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Bremen, Germany: IEEE, pp. 1–10. DOI: 10.1109/ICSM.2015.7332446.

Cohn, M. (2004). *User Stories Applied: For Agile Software Development*. Ed. by K. Beck and M. Fowler. 1st ed. Boston, MA, USA: Addison-Wesley, p. 286.

Cortés-Coy, L. F., Linares-Vásquez, M., Aponte, J., and Poshyvanyk, D. (2014). "On Automatically Generating Commit Messages via Summarization of Source Code Changes". In: *14th International Working Conference on Source Code Analysis and Manipulation*. Victoria, BC, Canada: IEEE, pp. 275–284. DOI: 10.1109/SCAM.2014.14.

Cursino, R., Ferreira, D., Lencastre, M., Fagundes, R., and Pimentel, J. (2018). "Gamification in Requirements Engineering: A Systematic Review". In: *11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, pp. 119–125. DOI: 10.1109/QUATIC.2018.00025.

Davis, F. D., Bagozzi, R. P., and Warshaw, P. R. (1989). "User Acceptance of Computer Technology: A Comparison of Two Theoretical Models". In: *Mangagement Science* 35.8, pp. 982–1002.

de Boer, R. C., Lago, P., Telea, A., and van Vliet, H. (2009). "Ontology-Driven Visualization of Architectural Design Decisions". In: *Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*. Cambridge, UK: IEEE, pp. 51–60. DOI: 10.1109/WICSA.2009.5290791.

de Boer, R. C. and van Vliet, H. (2009). "On the similarity between requirements and architecture". In: *Journal of Systems and Software* 82.3, pp. 544–550. DOI: 10.1016/j.jss.2008.11.185.

de Sombre, P. (2020). "Verlinkungsunterstützung und Duplikaterkennung von Wissenelementen". Master Thesis. Heidelberg: Heidelberg University.

Deshpande, G., Sheikhi, B., Chakka, S., Zotegouon, D. L., Masahati, M. N., and Ruhe, G. (2021). "Is BERT the New Silver Bullet? - An Empirical Investigation of Requirements Dependency Classification". In: *29th International Requirements Engineering Conference Workshops (REW)*. IEEE, pp. 136–145. DOI: 10.1109/REW53955.2021.00025.

Dhaouadi, M., Oakes, B. J., and Famelis, M. (2022). "End-to-End Rationale Reconstruction". In: *37th IEEE/ACM International Conference on Automated Software Engineering*. Rochester, MI, USA: ACM, pp. 1–5. DOI: 10.1145/3551349.3559547.

Dias, M., Bacchelli, A., Gousios, G., Cassou, D., and Ducasse, S. (2015). "Untangling fine-grained code changes". In: *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Montreal, QC, Canada: IEEE, pp. 341–350. DOI: 10.1109/SANER.2015.7081844.

Ding, W., Liang, P., Tang, A., and Vliet, H. van (2014). "Knowledge-Based Approaches in Software Documentation: A Systematic Literature Review". In: *Information and Software Technology* 56.6, pp. 545–567. DOI: 10.1016/j.infsof.2014.01.008.

Dingsøyr, T. and van Vliet, H. (2009). "Introduction to Software Architecture and Knowledge Management". In: *Software Architecture Knowledge Management*. Berlin, Heidelberg: Springer. Chap. 1, pp. 1–17. DOI: 10.1007/978-3-642-02374-3_1.

Doyle, M. and Straus, D. (1993). *How to Make Meetings Work: The New Interaction Method*. Berkley.

Dragomir, A., Lichter, H., and Budau, T. (2014). "Systematic Architectural Decision Management, A Process-Based Approach". In: *IEEE/IFIP Conference on Software Architecture*. Sydney, NSW, Australia: IEEE, pp. 255–258. DOI: 10.1109/WICSA.2014.39.

Durdik, Z. and Reussner, R. H. (2013). "On the Appropriate Rationale for Using Design Patterns and Pattern Documentation". In: *9th International ACM SIGSOFT Conference on the Quality of Software Architectures (QoSA)*. Vancouver, BC, Canada: ACM, pp. 107–116. DOI: 10.1145/2465478.2465491.

Dutoit, A. H., McCall, R., Mistrík, I., and Paech, B. (2006). *Rationale Management in Software Engineering: Concepts and Techniques*. Springer.

Dutoit, A. H., Wolf, T., Paech, B., Borner, L., and Rückert, J. (2005). "Using Rationale for Software Engineering Education". In: *18th Conference on Software Engineering Education and Training (CSEE&T)*. Ottawa, Canada: IEEE, pp. 129–136. DOI: 10.1109/CSEET.2005.1.

Easterbrook, S., Singer, J., Storey, M.-A., and Damian, D. (2008). "Selecting Empirical Methods for Software Engineering Research". In: *Guide to Advanced Empirical Software Engineering*. London: Springer. Chap. 11, pp. 285–311. DOI: 10.1007/978-1-84800-044-5_11.

Ebert, C., Gallardo, G., Hernantes, J., and Serrano, N. (2016). "DevOps". In: *IEEE Software* 33.3, pp. 94–100. DOI: 10.1109/MS.2016.68.

Falessi, D., Cantone, G., and Becker, M. (2006). "Documenting Design Decision Rationale to Improve Individual and Team Design Decision Making". In: *ACM/IEEE International Symposium on Empirical Software Engineering (ISESE)*. Rio de Janeiro, Brazil: ACM, pp. 134–143. DOI: 10.1145/1159733.1159755.

Fernandez, A., Garcia, S., Herrera, F., and Chawla, N. V. (2018). "SMOTE for Learning from Imbalanced Data: Progress and Challenges, Marking the 15-year Anniversary". In: *Journal of Artificial Intelligence Research* 61, pp. 863–905. DOI: 10.1613/jair.1.11192.

Filho, G. C. and Zisman, A. (2017). "D3TraceView: A Traceability Visualization Tool". In: *29th International Conference on Software Engineering and Knowledge Engineering (SEKE'17)*. Pittsburgh, PA, USA: KSI Research Inc., pp. 590–595. DOI: 10.18293/SEKE2017-038.

Fitzgerald, B. and Stol, K.-J. (2017). "Continuous software engineering: A roadmap and agenda". In: *Journal of Systems and Software* 123, pp. 176–189. DOI: 10.1016/j.jss.2015.06.063.

Fu, L., Liang, P., Li, X., and Yang, C. (2021). "A Machine Learning Based Ensemble Method for Automatic Multiclass Classification of Decisions". In: *Evaluation and Assessment in Software Engineering*. Trondheim, Norway: ACM, pp. 40–49. DOI: 10.1145/3463274.3463325.

García, F., Pedreira, O., Piattini, M., Cerdeira-Pena, A., and Penabad, M. (2017). "A framework for gamification in software engineering". In: *Journal of Systems and Software* 132, pp. 21–40. DOI: 10.1016/j.jss.2017.06.021.

Gaubatz, P., Lytra, I., and Zdun, U. (2015). "Automatic Enforcement of Constraints in Real-time Collaborative Architectural Decision Making". In: *Journal of Systems and Software* 103, pp. 128–149. DOI: 10.1016/j.jss.2015.01.056.

Gerdes, S., Soliman, M., and Riebisch, M. (2015). "Decision Buddy: Tool Support for Constraint-Based Design Decisions during System Evolution". In: *1st International Workshop on Future of Software Architecture Design Assistants (FoSADA)*. Montreal, QC, Canada: ACM, pp. 13–18. DOI: 10.1145/2751491.2751495.

Gerner, R. (2020). "Entwicklung eines Rationale Backlogs". Bachelor Thesis. Heidelberg University.

Gronert, F. (2019). "Unterstützung der Erstellung von Release-Beschreibungen durch dokumentiertes Entscheidungswissen". Bachelor Thesis. Heidelberg University.

Hamma, I. (2019). "Unterstützung der konsistenten Dokumentation von Entscheidungen im Software Engineering". Master Thesis. Heidelberg University.

Hansen, M. T., Nohria, N., and Tierney, T. (1999). "What's your strategy for managing knowledge?" In: *Harvard Business Review* 77.2, pp. 106–116.

Harari, Y. N. (2016). *Homo Deus: A brief history of tomorrow*. Harvill Secker.

Haselböck, S., Weinreich, R., and Buchgeher, G. (2019). "Using Decision Models for Documenting Microservice Architectures: A Student Experiment and Focus Group Study". In: *International Conference on Service-Oriented System Engineering (SOSE)*. San Francisco, CA, USA: IEEE, pp. 37–3709. DOI: 10.1109/SOSE.2019.00016.

Henze, D. (2020). "Dynamically Scalable Fog Architectures". Dissertation. Technical University of Munich, Germany.

Herzig, K. and Zeller, A. (2013). "The impact of tangled code changes". In: *10th Working Conference on Mining Software Repositories (MSR)*. San Francisco, CA, USA: IEEE, pp. 121–130. DOI: 10.1109/MSR.2013.6624018.

Hesse, T.-M. (2020). "Supporting Software Development by an Integrated Documentation Model for Decisions". Dissertation. Heidelberg University. DOI: 10.11588/heidok.00028713.

Hesse, T.-M., Kuehlwein, A., Paech, B., Roehm, T., and Bruegge, B. (2015). "Documenting Implementation Decisions with Code Annotations". In: *27th International Conference on Software Engineering and Knowledge Engineering (SEKE'15)*. Pittsburgh, PA, USA, pp. 152–157. DOI: 10.18293/SEKE2015-084.

Hesse, T.-M., Kuehlwein, A., and Roehm, T. (2016a). "DecDoc: A Tool for Documenting Design Decisions Collaboratively and Incrementally". In: *1st International Workshop on Decision Making in Software ARCHitecture (MARCH 2016)*. Venice, Italy: IEEE, pp. 30–37. DOI: 10.1109/MARCH.2016.9.

Hesse, T.-M., Lerche, V., Seiler, M., Knoess, K., and Paech, B. (2016b). "Documented decision-making strategies and decision knowledge in open source projects: An empirical study on Firefox issue reports". In: *Information and Software Technology* 79, pp. 36–51. DOI: 10.1016/j.infsof.2016.06.003.

Hesse, T.-M. and Paech, B. (2013). "Supporting the Collaborative Development of Requirements and Architecture Documentation". In: *3rd International Workshop on the Twin Peaks of*

Requirements and Architecture. Rio de Janeiro, Brazil: IEEE, pp. 22–26. DOI: 10.1109/TwinPeaks-2.2013.6617355.

Hesse, T.-M. and Paech, B. (2016). "Documenting Relations Between Requirements and Design Decisions: A Case Study on Design Session Transcripts". In: *Requirements Engineering: Foundation for Software Quality: 22nd International Working Conference, REFSQ 2016*. Ed. by M. Daneva and O. Pastor. Vol. LNCS 9619. Gothenburg, Sweden: Springer. Chap. Documentin, pp. 188–204. DOI: 10.1007/978-3-319-30282-9_13.

Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A., and America, P. (2007). "A general model of software architecture design derived from five industrial approaches". In: *Journal of Systems and Software* 80.1, pp. 106–126. DOI: 10.1016/j.jss.2006.05.024.

Hoorn, J. F., Farenhorst, R., Lago, P., and van Vliet, H. (2011). "The lonesome architect". In: *Journal of Systems and Software* 84.9, pp. 1424–1435. DOI: 10.1016/j.jss.2010.11.909.

Hübner, P. and Paech, B. (2020). "Interaction-based creation and maintenance of continuously usable trace links between requirements and source code". In: *Empirical Software Engineering (ESE)* 25.5, pp. 4350–4377. DOI: 10.1007/s10664-020-09831-w.

ISO/IEC 25010 (2011). *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. Geneva, Switzerland.

ISO/IEC/IEEE 24774 (2021). *Systems and software engineering – Life cycle management – Specification for process description*. Geneva, Switzerland.

Jacobson, I., Booch, G., and Rumbaugh, J. (1998). *The unified software development process*. Addison-Wesley.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2021). *An Introduction to Statistical Learning*. Springer Texts in Statistics. New York, NY: Springer, p. 618. DOI: 10.1007/978-1-0716-1418-1.

Jansen, A. G. J. and Bosch, J. (2005). "Software Architecture as a Set of Architectural Design Decisions". In: *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*. Pittsburgh, PA, USA: IEEE, pp. 109–120. DOI: 10.1109/WICSA.2005.61.

Jarczyk, A., Loffler, P., and Shipmann, F. (1992). "Design rationale for software engineering: a survey". In: *Twenty-Fifth Hawaii International Conference on System Sciences*. Vol. 2. Kauai, HI, USA: IEEE, pp. 577–586. DOI: 10.1109/HICSS.1992.183309.

Jesse, M. and Jannach, D. (2021). "Digital nudging with recommender systems: Survey and future directions". In: *Computers in Human Behavior Reports* 3. DOI: 10.1016/j.chbr.2020.100052.

Johanssen, J. O. (2019). "Continuous User Understanding in Software Evolution". Dissertation. Technical University of Munich, Germany.

Johanssen, J. O., Kleebaum, A., Bruegge, B., and Paech, B. (2017a). "Towards a Systematic Approach to Integrate Usage and Decision Knowledge in Continuous Software Engineering". In: *2nd Workshop on Continuous Software Engineering*. Hannover, Germany, pp. 7–11.

Johanssen, J. O., Kleebaum, A., Bruegge, B., and Paech, B. (2017b). "Towards the Visualization of Usage and Decision Knowledge in Continuous Software Engineering". In: *5th IEEE Working Conference on Software Visualization (VISSOFT 2017)*. Shanghai, China, pp. 104–108. DOI: 10.1109/VISSOFT.2017.18.

Johanssen, J. O., Kleebaum, A., Bruegge, B., and Paech, B. (2019a). "How do Practitioners Capture and Utilize User Feedback during Continuous Software Engineering?" In: *27th IEEE International Requirements Engineering Conference (RE'19)*. Jeju Island, South Korea: IEEE, pp. 153–164. DOI: 10.1109/RE.2019.00026.

Johanssen, J. O., Kleebaum, A., Paech, B., and Bruegge, B. (2018). "Practitioners' Eye on Continuous Software Engineering: An Interview Study". In: *International Conference on Software and System Process*. ICSSP '18. Gothenburg, Sweden: ACM, pp. 41–50. DOI: 10.1145/3202710.3203150.

Johanssen, J. O., Kleebaum, A., Paech, B., and Bruegge, B. (2019b). "The Eye of Continuous Software Engineering". In: *Software Engineering and Software Management (SE)*. Bonn, Germany: Gesellschaft für Informatik e.V., pp. 67–68. DOI: 10.18420/se2019-17.

Johanssen, J. O., Kleebaum, A., Paech, B., and Bruegge, B. (2019c). "Continuous software engineering and its support by usage and decision knowledge: An interview study with practitioners". In: *Journal of Software: Evolution and Process (JSEP)* 31.5, e2169. DOI: 10.1002/smr.2169.

Jonas, H. (1985). *The imperative of responsibility: In search of an ethics for the technological age*. University of Chicago Press.

Josephs, A., Gilson, F., and Galster, M. (2022). "Towards Automatic Classification of Design Decisions from Developer Conversations". In: *19th International Conference on Software Architecture Companion (ICSA-C)*. Honolulu, HI, USA: IEEE, pp. 10–14. DOI: 10.1109/ICSA-C54293.2022.00009.

Kahneman, D. (2011). *Thinking, Fast and Slow*. London, UK: Penguin Books.

Kano, N. (1984). "Attractive quality and must-be quality". In: *Hinshitsu. The Journal of Japanese Society for Quality Control* 14, pp. 39–48.

Kitchenham, B. A. and Charters, S. (2007). *Guidelines for Performing Systematic Literature Reviews in Software Engineering (Version 2.3)*. Tech. rep. EBSE 2007-001. Keele, Staffs, UK; Durham, UK: Keele University and Durham University Joint Report, p. 65.

Kleebaum, A., Johanssen, J. O., Paech, B., Alkadhi, R., and Bruegge, B. (2018a). "Decision knowledge triggers in continuous software engineering". In: *4th International Workshop on Rapid Continuous Software Engineering - RCoSE '18*. Gotheburg, Sweden: ACM Press, pp. 23–26. DOI: 10.1145/3194760.3194765.

Kleebaum, A., Johanssen, J. O., Paech, B., and Bruegge, B. (2018b). "Tool Support for Decision and Usage Knowledge in Continuous Software Engineering". In: *3rd Workshop on Continuous Software Engineering*, pp. 74–77. DOI: 10.11588/heidok.00024186.

Kleebaum, A., Johanssen, J. O., Paech, B., and Bruegge, B. (2019a). "Teaching Rationale Management in Agile Project Courses". In: *16. Workshop Software Engineering im Unterricht der Hochschulen (SEUH)*. Bremerhaven, Germany, pp. 125–132. DOI: 10.11588/heidok.00026358.

Kleebaum, A., Johanssen, J. O., Paech, B., and Bruegge, B. (2019b). "How do Practitioners Manage Decision Knowledge during Continuous Software Engineering?" In: *31st International Conference on Software Engineering and Knowledge Engineering*. SEKE'19. Lisbon, Portugal: KSI Research Inc., pp. 735–740.

Kleebaum, A., Johanssen, J. O., Paech, B., and Bruegge, B. (2019c). "Sharing and Exploiting Requirement Decisions". In: *Fachgruppentreffen Requirements Engineering (FGRE)*. Heidelberg, Germany: Gesellschaft für Informatik, pp. 19–20. DOI: 10.11588/heidok.00028596.

Kleebaum, A., Johanssen, J. O., Paech, B., and Bruegge, B. (2020). "Continuous Management of Requirement Decisions Using the ConDec Tools". In: *26th International Conference on Requirements Engineering (REFSQ20) Workshops, Doctoral Symposium, Live Studies Track, and Poster Track*. Pisa, Italy: CEUR-WS.org, p. 6. DOI: 10.11588/heidok.00028230.

Kleebaum, A., Johanssen, J. O., Paech, B., and Bruegge, B. (2021a). "Continuous Rationale Management Using the ConDec Tools". In: *Software Engineering 2021 Satellite Events*. Ed. by S. Götz, L. Linsbauer, I. Schaefer, and A. Wortmann. Braunschweig/Virtual: CEUR-WS, pp. 1–2. DOI: 10.11588/heidok.00029976.

Kleebaum, A., Konersmann, M., Langhammer, M., Paech, B., Goedicke, M., and Reussner, R. (2019d). "Continuous Design Decision Support". In: *Managed Software Evolution*. Ed. by R. Reussner, M. Goedicke, W. Hasselbring, B. Vogel-Heuser, J. Keim, and L. Märtin. Cham: Springer International Publishing. Chap. 6, pp. 107–139. DOI: 10.1007/978-3-030-13499-0_6.

Kleebaum, A., Paech, B., Johanssen, J. O., and Bruegge, B. (2021b). "Continuous Rationale Identification in Issue Tracking and Version Control Systems". In: *REFSQ-2021 Workshops, OpenRE, Posters and Tools Track, and Doctoral Symposium*. Essen/Virtual: CEUR-WS.org, p. 9. DOI: 10.11588/heidok.00029966.

Kleebaum, A., Paech, B., Johanssen, J. O., and Bruegge, B. (2021c). "Continuous Rationale Visualization". In: *Working Conference on Software Visualization (VISSOFT)*. Luxembourg: IEEE, pp. 33–43. DOI: 10.1109/VISSOFT52517.2021.00013.

Klepper, S., Krusche, S., and Bruegge, B. (2016). "Semi-automatic generation of audience-specific release notes". In: *International Workshop on Continuous Software Evolution and Delivery - CSED'16*. Austin, TX, USA: ACM, pp. 19–22. DOI: 10.1145/2896941.2896953.

Kopczyńska, S., Ochodek, M., Piechowiak, J., and Nawrocki, J. (2022). "On the benefits and problems related to using Definition of Done — A survey study". In: *Journal of Systems and Software* 193, p. 111479. DOI: 10.1016/j.jss.2022.111479.

Kopp, O. and Armbruster, A. (2019). "Generalized markdown architectural decision records: Capturing the essence of decisions". In: *11th Central European Workshop on Services and their Composition*. Vol. 2339. Bayreuth, Germany: CEUR, pp. 55–57.

Kopp, O., Armbruster, A., and Zimmermann, O. (2018). "Markdown architectural decision records: Format and tool support". In: *10th Central European Workshop on Services and their Composition*. Vol. 2072. Dresden, Germany: CEUR-WS.org, pp. 55–62.

Kretsou, M., Arvanitou, E.-M., Ampatzoglou, A., Deligiannis, I., and Gerogiannis, V. C. (2021). "Change impact analysis: A systematic mapping study". In: *Journal of Systems and Software* 174, p. 110892. DOI: 10.1016/j.jss.2020.110892.

Kruchten, P. (2004). "An Ontology of Architectural Design Decisions in Software-Intensive Systems". In: *2nd Groningen Workshop on Software Variability Management*, pp. 54–61.

Kruchten, P. (2009). "Documentation of Software Architecture from a Knowledge Management Perspective – Design Representation". In: *Software Architecture Knowledge Management*. Ed. by M. Ali Babar, T. Dingsøyr, P. Lago, and H. van Vliet. Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. 3, pp. 39–57. DOI: 10.1007/978-3-642-02374-3_3.

Kruchten, P., Capilla, R., and Dueñas, J. C. (2009). "The Decision View's Role in Software Architecture Practice". In: *IEEE Software* 26.2, pp. 36–42. DOI: 10.1109/MS.2009.52.

Kruchten, P., Lago, P., and van Vliet, H. (2006). "Building Up and Reasoning About Architectural Knowledge". In: *2nd International Conference on Quality of Software Architectures - Revised Papers (QoSA'06)*. Ed. by C. Hofmeister, I. Crnkovic, and R. Reussner. Vol. LNCS 4214. Lecture Notes in Computer Science. Västerås, Sweden: Springer Berlin Heidelberg, pp. 43–58. DOI: 10.1007/11921998.

Krusche, S. T. (2016). "Rugby – A Process Model for Continuous Software Engineering". Dissertation. Technical University Munich, Germany, p. 203.

Krusche, S. T., Alperowitz, L., Bruegge, B., and Wagner, M. O. (2014). "Rugby: An Agile Process Model Based on Continuous Delivery". In: *1st International Workshop on Rapid Continuous Software Engineering (RCoSE 2014)*, pp. 42–50. DOI: 10.1145/2593812.2593818.

Krusche, S. T. and Bruegge, B. (2017). "CSEPM - A Continuous Software Engineering Process Metamodel". In: *3rd International Workshop on Rapid Continuous Software Engineering (RCoSE)*. Buenos Aires, Argentina: IEEE, pp. 2–8. DOI: 10.1109/RCoSE.2017.6.

Krusche, S. T., von Frankenberg, N., and Afifi, S. (2017). "Experiences of a Software Engineering Course based on Interactive Learning". In: *15. Workshop Software Engineering im Unterricht der Hochschulen (SEUH)*. Hanover, Germany, pp. 32–40.

Kuchenbuch, T. (2019). "Darstellung der Evolution von Entscheidungswissen". Bachelor Thesis. Heidelberg University.

Kugele, S. and Antkowiak, D. (2016). "Visualization of Trace Links and Change Impact Analysis". In: *24th International Requirements Engineering Conference Workshops (REW)*. Beijing, China: IEEE, pp. 165–169. DOI: 10.1109/REW.2016.039.

Kunz, W. and Rittel, H. W. J. (1970). *Issues as elements of information systems*. Berkeley, California, US: Institute of Urban and Regional Development, University of California.

Kurtanović, Z. and Maalej, W. (2017). "Mining User Rationale from Software Reviews". In: *25th IEEE International Requirements Engineering Conference (RE)*. Ed. by A. Moeira and J. Araújo. Lisbon, Portugal: IEEE, pp. 53–62. DOI: 10.1109/RE.2017.86.

Kurtanović, Z. and Maalej, W. (2018). "On user rationale in software engineering". In: 23.3, pp. 357–379. DOI: 10.1007/s00766-018-0293-2.

Lauesen, S. (2002). *Software requirements: styles and techniques*. Pearson Education.

Lauesen, S. and Kuhail, M. A. (2012). "Task descriptions versus use cases". In: *Requirements Engineering* 17.1, pp. 3–18. DOI: 10.1007/s00766-011-0140-1.

Lee, J. (1991). "Extending the Potts and Bruns Model for Recording Design Rationale". In: *13th International Conference on Software Engineering (ICSE'91)*. Austin, TX, USA: IEEE, pp. 114–125. DOI: 10.1109/ICSE.1991.130629.

Lee, J. (1997). "Design Rationale Systems: Understanding the Issues". In: *IEEE Expert* 12.3, pp. 78–85. DOI: 10.1109/64.592267.

Lee, L. and Kruchten, P. (2008). "A Tool to Visualize Architectural Design Decisions". In: *Quality of Software Architectures. Models and Architectures: 4th International Conference on the Quality of Software-Architectures*. Ed. by S. Becker, F. Plasil, and R. Reussner. Springer, pp. 43–54. DOI: 10.1007/978-3-540-87879-7_3.

Lehman, M. M. (1980). "Programs, Life Cycles, and Laws of Software Evolution". In: *Proceedings of the IEEE (PROC)* 68.9, pp. 1060–1076. DOI: 10.1109/PROC.1980.11805.

Lehnert, S. (2011). "A taxonomy for software change impact analysis". In: *12th international workshop and the 7th annual ERCIM workshop on Principles on software evolution and software evolution - IWPSE-EVOL '11*. 1. Szeged, Hungary: ACM, p. 41. DOI: 10.1145/2024445.2024454.

Leite, L., Rocha, C., Kon, F., Milojicic, D., and Meirelles, P. (2019). "A Survey of DevOps Concepts and Challenges". In: *ACM Computing Surveys* 52.6, pp. 1–35. DOI: 10.1145/3359981.

Lester, M., Guerrero, M., and Burge, J. (2020). "Using evolutionary algorithms to select text features for mining design rationale". In: *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 34.2, pp. 132–146. DOI: 10.1017/S0890060420000037.

Li, X., Liang, P., and Li, Z. (2020). "Automatic Identification of Decisions from the Hibernate Developer Mailing List". In: *Evaluation and Assessment in Software Engineering*. December. Trondheim, Norway: ACM, pp. 51–60. DOI: 10.1145/3383219.3383225.

Liang, Y., Liu, Y., Kwong, C. K., and Lee, W. B. (2012). "Learning the "Whys": Discovering design rationale using text mining — An algorithm perspective". In: *Computer-Aided Design* 44.10, pp. 916–930. DOI: 10.1016/j.cad.2011.08.002.

Lopes Silva, I. C., Brito, P. H. S., dos S. Neto, B. F., Costa, E., and Silva, A. A. (2015). "A decision-making tool to support architectural designs based on quality attributes". In: *30th Annual ACM Symposium on Applied Computing (SAC)*. Salamanca, Spain: ACM, pp. 1457–1463. DOI: 10.1145/2695664.2695928.

López, C., Codocedo, V., Astudillo, H., and Cysneiros, L. M. (2012). "Bridging the gap between software architecture rationale formalisms and actual architecture documents: An ontology-driven approach". In: *Science of Computer Programming* 77.1, pp. 66–80. DOI: doi.org/10.1016/j.scico.2010.06.009.

Lytra, I., Tran, H., and Zdun, U. (2012). "Constraint-Based Consistency Checking between Design Decisions and Component Models for Supporting Software Architecture Evolution".

In: *16th European Conference on Software Maintenance and Reengineering*. Szeged, Hungary: IEEE, pp. 287–296. DOI: 10.1109/CSMR.2012.36.

Lytra, I., Tran, H., and Zdun, U. (2013). "Supporting Consistency between Architectural Design Decisions and Component Models through Reusable Architectural Knowledge Transformations". In: *Software Architecture: 7th European Conference, ECSA 2013, Montpellier, France, July 1-5, 2013, Proceedings*. Ed. by K. Drira. Vol. LNCS 7957. Lecture Notes in Computer Science. Montpellier, France: Springer, pp. 224–239. DOI: 10.1007/978-3-642-39031-9_20.

Lytra, I., Tran, H., and Zdun, U. (2015). "Harmonizing Architectural Decisions with Component View Models using Reusable Architectural Knowledge Transformations and Constraints". In: *Future Generation Computer Systems* 47, pp. 80–96. DOI: 10.1016/j.future.2014.11.010.

Lytra, I. and Zdun, U. (2014). "Inconsistency Management between Architectural Decisions and Designs Using Constraints and Model Fixes". In: *23rd Australian Software Engineering Conference*. Milsons Point, NSW, Australia: IEEE, pp. 230–239. DOI: 10.1109/ASWEC.2014.33.

MacLean, A., Young, R. M., Bellotti, V. M. E., and Moran, T. P. (1991). "Questions, Options, and Criteria: Elements of Design Space Analysis". In: *Human-Computer Interaction* 6.3-4, pp. 201–250.

Mahoney, M. S. (1990). "The roots of software engineering". In: *CWI Quarterly* 3.4, pp. 325–334.

Malloy, J. and Burge, J. E. (2016). "SEURAT_Edu: A Tool to Assist and Assess Student Decision-Making in Design". In: *47th Technical Symposium on Computing Science Education (SIGCSE'16)*. Memphis, Tennessee, USA: ACM, pp. 669–674. DOI: 10.1145/2839509.2844555.

Manteuffel, C., Tofan, D., Avgeriou, P., Koziolek, H., and Goldschmidt, T. (2015). "Decision architect - A decision documentation tool for industry". In: *Journal of Systems and Software* 112, pp. 181–198. DOI: 10.1016/j.jss.2015.10.034.

Marangunić, N. and Granić, A. (2015). "Technology acceptance model: a literature review from 1986 to 2013". In: *Universal Access in the Information Society* 14.1, pp. 81–95. DOI: 10.1007/s10209-014-0348-1.

Martinez-Fernandez, S., Jedlitschka, A., Guzman, L., and Vollmer, A. M. (2018). "A Quality Model for Actionable Analytics in Rapid Software Development". In: *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Prague, Czech Republic: IEEE, pp. 370–377. DOI: 10.1109/SEAA.2018.00067.

Maule, A. J. (2010). "Can Computers Help Overcome Limitations in Human Decision Making?" In: *International Journal of Human-Computer Interaction* 26.2-3, pp. 108–119. DOI: 10.1080/10447310903498684.

Merten, T., Mager, B., Hübner, P., Quirchmayr, T., Bürsner, S., and Paech, B. (2015). "Requirements Communication in Issue Tracking Systems in Four Open-Source Projects". In: *6th International Workshop on Requirements Prioritization and Communication (RePriCo)*. Essen, Germany, pp. 114–125.

Miesbauer, C. and Weinreich, R. (2012). "Capturing and Maintaining Architectural Knowledge Using Context Information". In: *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA)*. Helsinki, Finland: IEEE, pp. 206–210. DOI: 10.1109/WICSA-ECSA.212.30.

Miesbauer, C. and Weinreich, R. (2013). "Classification of Design Decisions - An Expert Survey in Practice". In: *7th European Conference on Software Architecture (ECSA'13)*. Ed. by K. Drira. Springer, pp. 130–145. DOI: 10.1007/978-3-642-39031-9_12.

Myers, M. D. and Newman, M. (2007). "The Qualitative Interview in IS Research: Examining the Craft". In: *Information and Organization* 17.1, pp. 2–26. DOI: 10.1016/j.infoandorg.2006.11.001.

Naur, P. and Randell, B. (1968). "Software Engineering". In: *Report of the 1968 conference in Garmisch, Germany*. NATO Science Committee.

Nazar, N., Hu, Y., and Jiang, H. (2016). "Summarizing Software Artifacts: A Literature Review". In: *Journal of Computer Science and Technology* 31.5, pp. 883–909. DOI: 10.1007/s11390-016-1671-1.

Nizenkov, K. (2019). "Design and implementation of a developer-centric quality web application". Bachelor Thesis. Heidelberg University.

Nonaka, I. and Takeuchi, H. (1995). "The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation". In: *Oxford University Press*.

Object Management Group (2017). *Unified Modeling Language Specification Version 2.5.1*.

Olsson, H. H., Alahyari, H., and Bosch, J. (2012). "Climbing the "Stairway to Heaven" – A Multiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software". In: *38th Euromicro Conference on Software Engineering and Advanced Applications*. Cesme, Turkey: IEEE, pp. 392–399. DOI: 10.1109/SEAA.2012.54.

Otchere, C. (2020). "Die Rolle von Qualitätsattributen während der Dokumentation und Nutzung von Entscheidungswissen". Bachelor Thesis. Heidelberg University.

Paech, B., Delater, A., and Hesse, T.-M. (2014). "Supporting Project Management Through Integrated Management of System and Project Knowledge". In: *Software Project Management in a Changing World*. Ed. by G. Ruhe and C. Wohlin. Heidelberg, Germany: Springer. Chap. 7, pp. 157–192. DOI: 10.1007/978-3-642-55035-5.

Paech, B. and Kohler, K. (2004). "Task-Driven Requirements in Object-Oriented Development". In: *Perspectives on Software Requirements*. Ed. by J. C. Sampaio do Prado Leite and J. H. Doorn. Vol. 753. Boston, MA: Springer US. Chap. 3, pp. 45–67. DOI: 10.1007/978-1-4615-0465-8_3.

Pan, S., Bao, L., Ren, X., Xia, X., Lo, D., and Li, S. (2021). "Automating Developer Chat Mining". In: *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Melbourne, Australia: IEEE, pp. 854–866. DOI: 10.1109/ASE51524.2021.9678923.

Paulk, M., Curtis, B., Chrissis, M., and Weber, C. (1993). "Capability maturity model, version 1.1". In: *IEEE Software* 10.4, pp. 18–27. DOI: 10.1109/52.219617.

Pedraza-García, G., Astudillo, H., and Correal, D. (2015). "DVIA: Understanding how software architects make decisions in design meetings". In: *European Conference on Software Architecture Workshops*. Dubrovnik/Cavtat, Croatia: ACM, pp. 1–7. DOI: 10.1145/2797433.2797486.

Pennington, J., Socher, R., and Manning, C. (2014). "Glove: Global Vectors for Word Representation". In: *Conf. on Empir. Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, pp. 1532–1543. DOI: 10.3115/v1/D14-1162.

Petersen, K., Feldt, R., Mujtaba, S., and Mattsson, M. (2008). "Systematic Mapping Studies in Software Engineering". In: *12th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. Bari, Italy: British Computer Society, pp. 68–77. DOI: 10.14236/ewic/EASE2008.8.

Pohl, K. and Rupp, C. (2016). *Requirements Engineering Fundamentals*. 2nd ed. Rocky Nook.

Popper, K. R. (1959). "The Logic of Scientific Discovery". In: *Hutchinson, London*.

R Core Team (2022). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria. URL: https://www.R-project.org.

Ralph, P. et al. (2020). "Empirical Standards for Software Engineering Research". In: *ACM Special Interest Group on Software Engineering (SIGSOFT)*. DOI: 10.48550/ARXIV.2010.03525.

Rastkar, S. and Murphy, G. C. (2013). "Why did this code change?" In: *35th International Conference on Software Engineering (ICSE)*. IEEE, pp. 1193–1196. DOI: 10.1109/ICSE.2013.6606676.

Rath, M., Rendall, J., Guo, J. L. C., Cleland-Huang, J., and Mäder, P. (2018). "Traceability in the wild". In: *40th International Conference on Software Engineering (ICSE)*. Gothenburg, Sweden: ACM, pp. 834–845. DOI: 10.1145/3180155.3180207.

Razavian, M., Paech, B., and Tang, A. (2023). "The vision of on-demand architectural knowledge systems as a decision-making companion". In: *Journal of Systems and Software* 198, p. 111560. DOI: 10.1016/j.jss.2022.111560.

Razavian, M., Tang, A., Capilla, R., and Lago, P. (2016). "In Two Minds: How Reflections Influence Software Design Thinking". In: *Journal of Software: Evolution and Process* 28.6, pp. 394–426. DOI: 10.1002/smr.1776.

Rittel, H. W. J. (1972). "On the Planning Crisis: Systems Analysis of the 'First and Second Generations'". In: *Bedriftsøkonomen* 8, pp. 390–396.

Robillard, M. P., Marcus, A., Treude, C., Bavota, G., Chaparro, O., Ernst, N., Gerosa, M. A., Godfrey, M., Lanza, M., Linares-Vásquez, M., Murphy, G. C., Moreno, L., Shepherd, D., and Wong, E. (2017). "On-Demand Developer Documentation". In: *International Conference on Software Maintenance and Evolution*, p. 5. DOI: 10.1109/ICSME.2017.17.

Robillard, M. P. and Walker, R. J. (2014). "An Introduction to Recommendation Systems in Software Engineering". In: *Recommendation Systems in Software Engineering*. Ed. by M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–11. DOI: 10.1007/978-3-642-45135-5_1.

Roeller, R., Lago, P., and van Vliet, H. (2006). "Recovering architectural assumptions". In: *Journal of Systems and Software* 79.4, pp. 552–573. DOI: 10.1016/j.jss.2005.10.017.

Rogers, B., Gung, J., Qiao, Y., and Burge, J. E. (2012). "Exploring techniques for rationale extraction from existing documents". In: *2012 34th International Conference on Software Engineering (ICSE)*. Zurich, Switzerland: IEEE, pp. 1313–1316. DOI: 10.1109/ICSE.2012.6227091.

Rogers, B., Justice, C., Mathur, T., and Burge, J. E. (2017). "Generalizability of Document Features for Identifying Rationale". In: *Design Computing and Cognition '16*. Cham: Springer International Publishing, pp. 633–651. DOI: 10.1007/978-3-319-44989-0_34.

Rogers, B., Qiao, Y., Gung, J., Mathur, T., and Burge, J. E. (2015). "Using Text Mining Techniques to Extract Rationale from Existing Documentation". In: *Design Computing and Cognition '14*. Springer, pp. 457–474. DOI: 10.1007/978-3-319-14956-1_26.

Runeson, P., Host, M., Rainer, A., and Regnell, B. (2012). *Case Study Research in Software Engineering: Guidelines and Examples*. 1st ed. Hoboken, NJ, USA: John Wiley & Sons, p. 256.

Saito, S., Iimura, Y., Massey, A. K., and Antón, A. I. (2017). "How Much Undocumented Knowledge is there in Agile Software Development? Case Study on Industrial Project using Issue Tracking System and Version Control System". In: *2017 IEEE 25th International Requirements Engineering Conference*. Lisbon, Portugal: IEEE Computer Society, pp. 186–195.

Saldaña, J. (2009). *The Coding Manual for Qualitative Researchers*. 2nd ed. SAGE Publications.

Sauerwein, E., Bailom, F., Matzler, K., and Hinterhuber, H. H. (1996). "The Kano model: How to delight your customers". In: *9th International Working Seminar on Production Economics*. Vol. 1. Innsbruck: Elsevier, pp. 313–327.

Schroeder, M. (1999). "A practical guide to object-oriented metrics". In: *IT Professional* 1.6, pp. 30–36. DOI: 10.1109/6294.806902.

Schubanz, M. (2014). "Design rationale capture in software architecture: What has to be captured?" In: *19th international doctoral symposium on Components and architecture - WCOP '14*. Marcq-en-Bareul, France: ACM, pp. 31–36. DOI: 10.1145/2601328.2601329.

Schubanz, M. (2021). *How Software Engineers Deal With Decisions in Scrum – An Analysis*. Tech. rep. Brandenburgische Technische Universität Cottbus-Senftenberg, p. 387. DOI: 10.26127/BTUOpen-5066.

Schubanz, M. and Lewerentz, C. (2020). "What Matters to Students – A Rationale Management Case Study in Agile Software Development". In: *17. Workshop Software Engineering im Unterricht der Hochschulen (SEUH)*. Innsbruck, Österreich: CEUR, pp. 17–26.

Schwaber, K. and Beedle, M. (2002). *Agile software development with Scrum*. Prentice Hall.

Seaman, C. (1999). "Qualitative Methods in Empirical Studies of Software Engineering". In: *IEEE Transactions on Software Engineering* 25.4, pp. 557–572. DOI: 10.1109/32.799955.

Seiler, M. (2017). "Dokumentation und Nutzung von Entscheidungen in Code". Bachelor Thesis. Heidelberg University.

Shahbazian, A., Kyu Lee, Y., Le, D., Brun, Y., and Medvidovic, N. (2018). "Recovering Architectural Design Decisions". In: *International Conference on Software Architecture (ICSA)*. Seattle, WA: IEEE, pp. 95–104. DOI: 10.1109/ICSA.2018.00019.

Shahin, M., Ali Babar, M., and Zhu, L. (2017). "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". In: *IEEE Access* 5.Ci, pp. 3909–3943. DOI: 10.1109/ACCESS.2017.2685629.

Shahin, M., Liang, P., and Khayyambashi, M. R. (2010). "Improving Understandability of Architecture Design through Visualization of Architectural Design Decision". In: *ICSE Workshop on Sharing and Reusing Architectural Knowledge (SHARK '10)*. Ed. by P. Avgeriou, P. Lago, and P. Kruchten. Cape Town, South Africa: ACM, pp. 88–95. DOI: 10.1145/1833335.1833348.

Shahin, M., Liang, P., and Li, Z. (2011). "Architectural Design Decision Visualization for Architecture Design: Preliminary Results of A Controlled Experiment". In: *5th European Conference on Software Architecture (ECSA)*. Ed. by I. Crnkovic, V. Gruhn, and M. Book. Vol. Companion. Essen, Germany: ACM, 2:1–2:8. DOI: 10.1145/2031759.2031762.

Sharma, P. N., Savarimuthu, B. T. R., and Stanger, N. (2021). "Extracting Rationale for Open Source Software Development Decisions — A Study of Python Email Archives". In: *43rd International Conference on Software Engineering (ICSE)*. Madrid, Spain: IEEE, pp. 1008–1019. DOI: 10.1109/ICSE43902.2021.00095.

Sharma, P. N., Savarimuthu, B. T. R., and Stanger, N. (2023). "How are decisions made in open source software communities? — Uncovering rationale from python email repositories". In: *Journal of Software: Evolution and Process* November 2022, pp. 1–29. DOI: 10.1002/smr.2526.

Shi, L., Jiang, Z., Yang, Y., Chen, X., Zhang, Y., Mu, F., Jiang, H., and Wang, Q. (2021). "ISPY: Automatic Issue-Solution Pair Extraction from Community Live Chats". In: *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021*, pp. 142–154. DOI: 10.1109/ASE51524.2021.9678894.

Soliman, M., Avgeriou, P., and Li, Y. (2021). "Architectural design decisions that incur technical debt — An industrial case study". In: *Information and Software Technology* 139, p. 106669. DOI: 10.1016/j.infsof.2021.106669.

Solis, C. and Ali, N. (2010). "Distributed Requirements Elicitation Using a Spatial Hypertext Wiki". In: *5th IEEE International Conference on Global Software Engineering*. Princeton, NJ, USA: IEEE, pp. 237–246.

Spiekermann, S. (2019). *Digitale Ethik: ein Wertesystem für das 21. Jahrhundert*. Droemer.

Spillner, A. and Linz, T. (2021). *Software Testing Foundations: A Study Guide for the Certified Tester Exam – Foundation Level – International Software Testing Qualifications Board (ISTQB) Compliant*. dpunkt.verlag.

Stol, K.-J. and Fitzgerald, B. (2018). "The ABC of Software Engineering Research". In: *ACM Transactions on Software Engineering and Methodology* 27.3, pp. 1–51. DOI: 10.1145/3241743.

Svensson, R. B., Feldt, R., and Torkar, R. (2019). "The Unfulfilled Potential of Data-Driven Decision Making in Agile Software Development". In: *Lecture Notes in Business Information Processing*. Vol. 355, pp. 69–85. DOI: 10.1007/978-3-030-19034-7_5.

Tang, A., Babar, M. A., Gorton, I., and Han, J. (2006). "A survey of architecture design rationale". In: *Journal of Systems and Software* 79.12, pp. 1792–1804. DOI: 10.1016/j.jss.2006.04.029.

Tang, A. and Lau, M. F. (2014). "Software Architecture Review by Association". In: *Journal of Systems and Software* 88.1, pp. 87–101. DOI: 10.1016/j.jss.2013.09.044.

Tang, A., Liang, P., Clerc, V., and van Vliet, H. (2011). "Traceability in the Co-evolution of Architectural Requirements and Design". In: *Relating Software Requirements and Architectures*.

Ed. by P. Avgeriou, J. Grundy, J. G. Hall, P. Lago, and I. Mistrík. Berlin, Heidelberg, Germany: Springer Berlin Heidelberg. Chap. 4, pp. 35–60. DOI: 10.1007/978-3-642-21001-3_4.

Tao, Y. and Kim, S. (2015). "Partitioning Composite Code Changes to Facilitate Code Review". In: *12th Working Conference on Mining Software Repositories*. Florence, Italy: IEEE, pp. 180–190. DOI: 10.1109/MSR.2015.24.

Thaler, R. H. and Sunstein, C. R. (2008). *Nudge: Improving Decisions about Health, Wealth, and Happiness*. New York, USA: Yale University Press.

Theunissen, T., van Heesch, U., and Avgeriou, P. (2022). "A mapping study on documentation in Continuous Software Development". In: *Information and Software Technology* 142, p. 106733. DOI: 10.1016/j.infsof.2021.106733.

Thurimella, A. K., Schubanz, M., Pleuss, A., and Botterweck, G. (2017). "Guidelines for Managing Requirements Rationales". In: *IEEE Software* 34.1, pp. 82–90. DOI: 10.1109/MS.2015.157.

Tofan, D., Galster, M., and Avgeriou, P. (2013). "Difficulty of Architectural Decisions – A Survey with Professional Architects". In: *7th European Conference on Software Architecture (ECSA'13)*. Ed. by K. Drira. Vol. LNCS 7957. Lecture Notes in Computer Science. Montpellier, France: Springer, pp. 192–199. DOI: 10.1007/978-3-642-39031-9_17.

Tralle, L. (2019). "Visualisierung und Verwaltung von Entscheidungswissen in Jira". Bachelor Thesis. Heidelberg University.

van der Ven, J. S. and Bosch, J. (2013). "Making the Right Decision: Supporting Architects with Design Decision Data". In: *Lecture Notes in Computer Science*. Vol. 7957. Berlin, Heidelberg: Springer, pp. 176–183. DOI: 10.1007/978-3-642-39031-9_15.

van Heesch, U., Avgeriou, P., and Hilliard, R. (2012). "A documentation framework for architecture decisions". In: *Journal of Systems and Software* 85.4, pp. 795–820. DOI: 10.1016/j.jss.2011.10.017.

Wagner, S., Goeb, A., Heinemann, L., Kläs, M., Lampasona, C., Lochmann, K., Mayr, A., Plösch, R., Seidl, A., Streit, J., and Trendowicz, A. (2015). "Operationalised product quality models and assessment: The Quamoco approach". In: *Information and Software Technology* 62.1, pp. 101–123. DOI: 10.1016/j.infsof.2015.02.009.

Wang, W. and Burge, J. E. (2010). "Using Rationale to Support Pattern-Based Architectural Design". In: *ICSE Workshop on Sharing and Reusing Architectural Knowledge - SHARK '10*. Cape Town, South Africa: ACM, pp. 1–8. DOI: 10.1145/1833335.1833336.

Weinreich, R. and Groher, I. (2016). "Software architecture knowledge management approaches and their support for knowledge management activities: A systematic literature review". In: *Information and Software Technology (IST)* 80, pp. 265–286. DOI: 10.1016/j.infsof.2016.09.007.

Weinreich, R., Groher, I., and Miesbauer, C. (2015). "An expert survey on kinds, influence factors and documentation of design decisions in practice". In: *Future Generation Computer Systems (FGCS)* 47, pp. 145–160. DOI: 10.1016/j.future.2014.12.002.

Wieringa, R. J. (2014). *Design Science Methodology for Information Systems and Software Engineering*. Springer Berlin Heidelberg, p. 332. DOI: 10.1007/978-3-662-43839-8.

Wieringa, R. J. and Moralı, A. (2012). "Technical Action Research as a Validation Method in Information Systems Design Science". In: *Design Science Research in Information Systems: Advances in Theory and Practice*. Vol. LNCS 7286. Lecture Notes in Computer Science. Las Vegas, Nevada, USA: Springer Berlin Heidelberg, pp. 220–238. DOI: 10.1007/978-3-642-29863-9_17.

Wisniowski, L. (2019). "Quality assurance of documented decision knowledge in feature branches". Master Thesis. Heidelberg University.

Wohlin, C. (2016). "Second-generation systematic literature studies using snowballing". In: *20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, 15:1–15:16. DOI: 10.1145/2915970.2916006.

Wohlin, C. and Aurum, A. (2015). “Towards a decision-making structure for selecting a research design in empirical software engineering”. In: 20.6, pp. 1427–1455. DOI: 10.1007/s10664-014-9319-7.

Yan, K. (2021). “Unterstützung der Analyse von Änderungsauswirkungen auf Graphen von Wissenelementen”. Master Thesis. Heidelberg University.

Yang, C., Liang, P., and Avgeriou, P. (2019). “Integrating Agile Practices into Architectural Assumption Management: An Industrial Survey”. In: *Evaluation and Assessment on Software Engineering (EASE)*. March. Copenhagen, Denmark: ACM, pp. 156–165. DOI: 10.1145/3319008.3319027.

Zannier, C., Chiasson, M., and Maurer, F. (2007). “A model of design decision making based on empirical results of interviews with software designers”. In: *Information and Software Technology* 49.6, pp. 637–653. DOI: 10.1016/j.infsof.2007.02.010.

Zdun, U., Capilla, R., Tran, H., and Zimmermann, O. (2013). “Sustainable Architectural Design Decisions”. In: *IEEE Software* 30.6, pp. 46–53. DOI: 10.1109/MS.2013.97.

Zhang, L., Sun, Y., Song, H., Chauvel, F., and Mei, H. (2011). “Detecting Architecture Erosion by Design Decision of Architectural Pattern”. In: *23rd International Conference on Software Engineering and Knowledge Engineering*. Skokie, IL, USA: Knowledge Systems Institute Graduate School, pp. 758–763.

Zhao, L., Alhoshan, W., Ferrari, A., Letsholo, K. J., Ajagbe, M. A., Chioasca, E. V., and Batista-Navarro, R. T. (2020). “Natural Language Processing (NLP) for requirements engineering: A systematic mapping study”. In: *arXiv* v. arXiv: 2004.01099.

Zhi, J., Garousi-Yusifoğlu, V., Sun, B., Garousi, G., Shahnewaz, S., and Ruhe, G. (2015). “Cost, benefits and quality of software development documentation: A systematic mapping”. In: *Journal of Systems and Software* 99, pp. 175–198. DOI: 10.1016/j.jss.2014.09.042.

Zhu, J., He, P., Fu, Q., Zhang, H., Lyu, M. R., and Zhang, D. (2015). “Learning to Log: Helping Developers Make Informed Logging Decisions”. In: *37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, pp. 415–425. DOI: 10.1109/ICSE.2015.60.

Zimmermann, O. (2011). “Architectural Decisions as Reusable Design Assets”. In: *IEEE Software* 28.1, pp. 64–69. DOI: 10.1109/MS.2011.3.

Zimmermann, O. and Miksovic, C. (2013). “Decisions Required vs. Decisions Made”. In: *Aligning Enterprise, System, and Software Architectures*. Ed. by I. Mistrik, A. Tang, R. Bahsoon, and J. A. Stafford. IGI Global. Chap. 10, pp. 176–208. DOI: 10.4018/978-1-4666-2199-2.ch010.

Zimmermann, O., Wegmann, L., Koziolek, H., and Goldschmidt, T. (2015). “Architectural Decision Guidance across Projects: Problem Space Modeling, Decision Backlog Management and Cloud Computing Knowledge”. In: *12th Working IEEE/IFIP Conference on Software Architecture (WICSA ’15)*. Ed. by L. Bass, P. Lago, and P. Kruchten. Montréal, Québec, Canada: IEEE, pp. 85–94. DOI: 10.1109/WICSA.2015.29.

Zubrod, P. (2017). “Dokumentation und Nutzung von Entscheidungen in Git”. Bachelor Thesis. Heidelberg University.

Zubrod, P. (2021). “Vorschlagsmechanismus für Lösungsoptionen zu Entscheidungsproblemen in der Softwareentwicklung”. Master Thesis. Heidelberg University.

List of Figures

List of Tables

List of Equations