Jochen Clormann
Matrikelnummer: 2934569

Masterarbeit

Ruprecht-Karls-Universität Heidelberg
Lehrstuhl für Software Engineering

DecXtract: Dokumentation und Nutzung von Entscheidungswissen in JIRA-Issue-Kommentaren

Gutachterin:
Prof. Dr. Barbara Paech

Betreuerin:
Anja Kleebaum

Abgabe: 18.12.2018
Bearbeitungszeit: 6 Monate

Zusammenfassung

[**Kontext und Motivation**] Zur Entwicklung eines Softwaresystems benötigen EntwicklerInnen Wissen über die Evolution der zugehörigen Softwareartefakte. EntwicklerInnen müssen beispielsweise vorangegangene Quellcodeänderungen verstehen, um eine neue Anforderung bzw. eine Änderung an einer bestehenden Anforderung zu implementieren. Dabei spielt Entscheidungswissen eine zentrale Rolle für die erfolgreiche Evolution eines Softwaresystems: EntwicklerInnen benötigen Wissen zu bereits getroffenen Entscheidungen, um eigene Entscheidungen zu treffen. Um neuen EntwicklerInnen in einem Softwareprojekt den Zugriff auf dieses Entscheidungswissen zu ermöglichen, ist es wichtig, dass Projektbeteiligte Entscheidungen dokumentieren. Dafür nutzen sie verschiedene Dokumentationsmöglichkeiten und halten Entscheidungswissen häufig informell in Commitnachrichten, Chatnachrichten oder Kommentaren fest.

[**Beiträge**] Diese Masterarbeit umfasst eine systematische Literaturrecherche, die Entwicklung des JIRA-Plug-Ins DecXtract sowie einen Datensatz als Goldstandard zur Evaluation. Die systematische Literaturrecherche beantwortet die Frage, welches Wissen EntwicklerInnen für das Verständnis von Quellcodeänderungen benötigen und welche Rolle Entscheidungswissen dafür spielt. Neben einer Liste mit konkreten Fragen, die EntwicklerInnen zu einer Quellcodeänderung stellen, zeigt die Literaturrecherche, dass EntwicklerInnen explizites Entscheidungswissen benötigen, dieses aber nicht dokumentiert oder auffindbar ist. Um Entscheidungswissen in Kommentaren explizit zu machen und seine Auffindbarkeit zu verbessern, wird das Entscheidungsdokumentationstool ConDec für JIRA um die Komponente DecXtract erweitert. DecXtract ermöglicht EntwicklerInnen, Text in Kommentaren von JIRA-Issues automatisch als Entscheidungswissen zu klassifizieren sowie eigene Kommentare manuell als explizites, klassifiziertes Entscheidungswissen zu dokumentieren. EntwicklerInnen können Entscheidungswissen in Kommentaren zu bestehenden Softwareartefakten wie Anforderungen, Entwicklungsaufgaben oder anderem Entscheidungswissen verlinken. So entsteht ein integriertes Wissensmodell mit explizit dokumentiertem Entscheidungswissen. Dieses integrierte Modell wird als Graph visualisiert und bietet NutzerInnen verschiedene Möglichkeiten zur Verwaltung.
Eine Befragung von EntwicklerInnen, die ebenfalls am ConDec-Projekt beteiligt sind, zeigt die Eignung von DecXtract zur Dokumentation von Entscheidungswissen. Zur Evaluation wird ein JIRA-Projekt mit 90 JIRA-Issues des Apache LUCENE Projektes erzeugt. Mit Hilfe von DecXtract wird informelles Entscheidungswissen aus den JIRA-Issue-Kommentaren explizit angelegt und miteinander verlinkt. Anhand der identifizierten Fragen von EntwicklerInnen an eine Quellcodeänderung wird das von DecXtract bereitgestellte Wissen genutzt um das Verständnis von Quellcodeänderungen zu evaluieren.

[**Schlussfolgerung**] DecXtract verbessert die Entscheidungsdokumentation durch die Klassifikation von Entscheidungselementen in JIRA-Issue-Kommentaren. Durch das präsentierte Entscheidungswissen können EntwicklerInnen immer den aktuellen Stand einer Diskussion und des dazugehörigen Entscheidungsproblems beobachten. Wenn eine Quellcodeänderung mit einem JIRA-Issue verlinkt ist, können EntwicklerInnen das Entscheidungswissen zu einer Quellcodeändeurng betrachtet. Dieses Wissen unterstützt EntwicklerInnen beim Verständnis der Quellcodeänderung.

Abstract

[**Context and motivation**] To develop a software system, developers need knowledge about the evolution of its software artifacts. For example, developers need to understand previous source code changes to implement a new requirement or changes to an existing requirement. Decision knowledge is important to successfully evolve a software system: Developers need knowledge about the decisions made to make new decisions. In order to make decision knowledge accessible to new developers, it is important that project participants document the decision knowledge. For this purpose, they use various documentation locations and often informally capture decision knowledge in commit messages, chat messages, or comments.

[**Contributions**] This master thesis presents a systematic literature review, the development of the JIRA plug-in DecXtract and a dataset as the gold standard for the evaluation. The systematic literature review answers the question what knowledge developers need to understand source code changes and what role decision knowledge plays. In addition to a list of specific questions developers ask about source code changes, the literature review shows that developers need explicit decision knowledge, but this is not documented or traceable. To make decision knowledge in comments explicit and to improve its accessibility, the decision documentation tool ConDec for JIRA is extended by the component DecXtract. DecXtract allows developers to automatically classify text in comments from JIRA Issues as decision knowledge, and manually document new comments as explicit and classified decision knowledge. Developers can link decision knowledge in JIRA Issue comments to existing software artifacts such as requirements, development tasks, or other decision knowledge. This results in an integrated knowledge model with explicitly documented knowledge. This integrated model is visualized as a graph and offers users various ways to manage it. A survey of developers who are also involved in the ConDec project demonstrates the suitability of DecXtract to document decision knowledge. For evaluation, a JIRA project with 90 JIRA issues of the Apache LUCENE project is created. With the help of DecXtract, informal decision knowledge from the JIRA Issue comments is explicitly created and linked to each other. Based on the identified questions from developers to a source code change, the knowledge provided by DecXtract is used to evaluate the understanding of source code changes.

[**Conclusion**] DecXtract improves decision making by classifying explicit decision knowledge elements in JIRA issue comments. Through the presented decision knowledge, developers can observe the current status of a discussion and the associated decision problem. If a source code change is linked to a JIRA issue, developers may look upon decision knowledge about a source code change. This knowledge helps developers understand the source code change.

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Erstellung dieser Arbeit unterstützt haben.

Das größte Dankeschön geht an meine Betreuerin Anja Kleebaum. Danke für deine hervorragende Betreuung, deine hilfreichen Anregungen, dein konstruktives Feedback, und jede Stunde Zeit du für mich und meine Arbeit investiert hast. Danke, für alles was ich von dir lernen konnte!

Natürlich möchte ich mich auch bei Frau Prof. Dr. Barbara Paech bedanken. Für sämtliches Wissen, dass ich in zwei Abschlussarbeiten, vier HiWi Stellen sowie zahlreichen Vorlesungen und Lehrveranstaltungen erlernen durfte.

Also, a big thank you goes to Rana Alkadhi from Munich, who shared her knowledge and data set on text classification with me.

Ein Dankeschön geht auch das ConDec-Entwicklungsteam: Leffi, Lars, Lukaz, Vita, Fabian und natürlich Tim, der mir sehr bei der Einarbeitung in die JIRA Plug-In Entwicklung geholfen hat und bei JIRA-Problemen immer zur Stelle war.

Das tiefste Dankeschön geht an meine Eltern Rainer und Sabine. Die mir mein Studium überhaupt ermöglicht haben, mich jederzeit Unterstützt und Ermutigt haben und bis zuletzt einige meiner Aufgaben übernommen haben, sodass ich Zeit für diese Arbeit aufbringen konnten.

Gleiches gilt natürlich auch für meine Großeltern, die immer ein offenes Ohr für mich haben und jederzeit ihre Hilfe anbieten. Und natürlich auch meinen Bruder Stefan, der dieses Jahr ebenfalls ein neues Lebenskapitel anfängt.

Ohne meinen Onkel Christian und Tante Silke, hätte ich wahrscheinlich nie angefangen zu studieren. Ich danke euch für eure Unterstützung und Hilfsbereitschaft auf meinen gesamten Bildungsweg. Es hat sehr lange gedauert, aber ich habe verstanden das fordern auch immer fördern heißt. Danke dafür!

Besonders möchte ich mich auch bei meiner Freundin Pia bedanken, für deine Geduld, deine Nachsicht, deinen emotionalen Rückhalt und vor allem die vielen Stunden Zeit in denen du meine Pflichten übernommen hast, damit ich arbeiten kann, danke < 3!

Ein großes Dankeschön geht auch meine beste Freundin Linda, mit der ich die besten Zeiten meines Studiums verbracht habe, die immer Zeit für mich hat und immer eine konstruktive fachgebietsunabhängige Problemlösung beigetragen hat.

Zu guter Letzt möchte ich mich bei meinen Korrektur-Leserinnen Colin, Christian, Linda, Pia und natürlich dem besten Lateinlehrer Chris, für eure Zeit bedanken!

Inhaltsverzeichnis

Eidesstaatliche Erklärung zur Masterarbeit

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

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

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

Heidelberg, den
Ort, Datum

Jochen Clormann

1 Einleitung

Bei der Entwicklung eines Softwaresystems treffen EntwicklerInnen und andere Projektbeteiligte viele Entscheidungen. Das können Design-, Architektur- oder Implementierungsentscheidungen sein. Jede Entscheidung geht dabei aus einem Entscheidungsproblem hervor.

Um Entscheidungsprobleme zu lösen, bewerten und vergleichen Projektbeteiligte vorliegende Alternativen. Beispielsweise ist es ein Entscheidungsproblem, welche Bibliothek ein Softwaresystems zur graphischen Visualisierung von Entscheidungswissen verwendet. Sämtliches Wissen, das ein Entscheidungsproblem beeinflusst, wird als Entscheidungswissen bezeichnet. Damit EntwicklerInnen Entscheidungen und Entscheidungsprobleme zu einem späteren Zeitpunkt nachvollziehen können, müssen alle Projektbeteiligten ihre Entscheidungen, die dazugehörigen Entscheidungsprobleme, Alternativen und Argumente dafür bzw. dagegen, dokumentieren. Undokumentiertes Entscheidungswissen ist nur Projektbeteiligten bekannt, die an der Lösung des Entscheidungsproblems beteiligt waren. Stehen die beteiligten Personen zu einem späteren Zeitpunkt nicht mehr zur Verfügung, ist das entsprechende Wissen verloren [8].

1.1 Motivation

Ko et al. führten eine Studie bei Microsoft durch und beobachteten 17 Entwicklungsteams für 90 Minuten in ihrem Arbeitsalltag [31]. Die Autoren identifizieren die verschiedenen Aufgaben und Arbeitsabläufe der Teams. Anschließend werden die beteiligten Personen zu ihrer Arbeit und ihrem Alltag befragt. Die Frage „Wieso wurde dieser Code auf diese Weise entwickelt?“ konnten 44% der befragten Personen nicht beantworten. Dieser Wert ist auf unzureichende Entscheidungsdokumentation im Entwicklungsprozess und den Verlust von Entscheidungswissen zurückzuführen.

Lougher und Rodden identifizieren drei Ursachen, die zum Verlust von Entscheidungswissen führen [12]. 1) Entscheidungen werden unstrukturiert im Quellcode-Kommentaren gespeichert. 2) Entscheidungswissen wird nicht erfasst oder gespeichert. 3) Entscheidungswissen wird in externen Dokumenten festgehalten, aber nicht zum Quellcode verlinkt. Durch mangelhafte Entscheidungsdokumentation können Projektbeteiligte zu späteren Zeitpunkten Entscheidungen nicht mehr nachvollziehen. Dieser Wissensverlust wird auch als Erosion bezeichnet.

Es existieren verschiedene Modelle, Entscheidungswissen strukturiert zu dokumentieren. Kunz et al. beschreiben das IBIS Modell mit drei Wissenstypen: *Issue*, *Position* und *Argument* [9]. Gemeinsam bilden diese Wissenstypen ein Entscheidungsproblem mit verschiedenen Positionen. Argumente können Positionen unterstützen oder in Frage stellen. Hesse et al. beschreiben ein detailliertes Entscheidungsdokumentationsmodell [1, 10]. Sämtliches Entscheidungswissen ist hier als *DecisionComponent* festgehalten. Diese Instanz definiert sich durch weitere Komponenten wie: *Problem*, *Solution*, *Context* und *Argument*.

Open-Source-Projekte (z.B. Apache Lucene, Ubuntu oder Mozilla Thunderbird) nutzen Versionskontrollsysteme wie Git oder Subversion, um den Quellcode zu teilen und zu versionieren. Weiterhin nutzen diese Open-Source-Projekte ein Issue-Tracking-System (ITS) wie JIRA. ITS ermöglichen die Dokumentation von Anforderungen und Entwickleraufgaben sowie Diskussionen zu Entwicklungsartefakten. Bei diesen Diskussionen halten EntwicklerInnen informelles Entscheidungswissen bspw. in JIRA-Issue-Kommentaren fest. Möchte eine EntwicklerIn beispielsweise ein konkretes Entscheidungsproblem und dessen Lösungen verstehen, muss sie alle vorhandenen Diskussionen lesen und versuchen das Entscheidungsproblem und dessen Lösung darin zu erkennen. Es ist dabei nicht klar, ob die Dokumentation wirklich vollständig ist, was das Verständndis der Kommentare für die EntwicklerIn erschwert.

Es existieren verschiedene Ansätze natürlichsprachliche Texte nach gewissen Kriterien zu klassifizieren [7, 16]. R. Alkadhi befasst sich in ihrer Arbeit mit der Klassifikation von natürlicher Sprache in Wissenstypen. Ihr Ansatz erkennt entscheidungsrelevante Sätze in natürlicher Sprache und kann anschließend einen Wissenstyp klassifizieren [7]. Es wurde jedoch nicht erforscht, wie einzelne Elemente eines Entscheidungsdokumentationsmodells miteinander in Verbindung gebracht werden können, nachdem sie aus natürlicher Sprache klassifiziert wurden. Zudem ist es offen, welchen Mehrwert das identifizierte Entscheidungswissen für das Verständnis und die Weiterentwicklung der Software bietet.

Diese Arbeit beschäftigt sich mit der Frage, wie Entscheidungswissen in Fließtexten erkannt und in ein Entscheidungsdokumentationsmodell überführt werden kann und welchen Nutzen die Betrachtung von informellem Entscheidungswissen zum Verständnis von Quellcodeänderungen bietet.

1.2 Ziele der Arbeit

Diese Arbeit führt eine systematische Literatursuche durch, um die Frage zu beantworten, welches Wissen und welches Entscheidungswissen EntwicklerInnen zum Verständnis von Quellcodeänderungen benötigen.

Das JIRA-Plug-In ConDec zu Entscheidungsdokumentation soll um die Komponente DecXtract erweitert werden. DecXtract soll NutzerInnen die Möglichkeit zur Entscheidungsdokumentation in JIRA-Issue-Kommentaren bieten. Die Anforderungsdokumentation definiert die nötigen Funktionalitäten von DecXtract.

Die NutzerInnen sollen mit DecXtract Entscheidungswissen manuell in Wissenstypen klassifizieren oder eine automatische Klassifikation nutzen. Anschließend kann dieses Entscheidungswissen mit weiterem Entscheidungswissen und Projektwissen mit ConDec verlinkt werden.

Eine umfangreiche Qualitätssicherung soll die Qualität von DecXtract durch Komponententests, Integrationstests und Systemtests sicherstellen.

Die Evaluation soll anhand einer Umfrage die Eignung von DecXtract zur Entscheidungsdokumentation zeigen. Des Weiteren soll ein Evaluationsdatensatz erzeugt werden, der Entscheidungswissen aus JIRA-Issues dokumentiert und mit einer Quellcodeänderung verlinkt ist. Mit diesem Datensatz soll gezeigt werden, dass DecXtract benötigtes Wissen von EntwicklerInnen bereitstellt, um eine Quellcodeänderung zu verstehen.

1.3 Struktur

Kapitel 2 beschreibt die nötigen Grundlagen und Begriffe für diese Arbeit. Kapitel 3 dokumentiert die Methoden und Ergebnisse der systematischen Literaturrecherche. Kapitel 4 definiert Nutzerrollen und die umzusetzenden Anforderungen für DecXtract durch Personae, User Tasks, Systemfunktionen und User Stories. Kapitel 5 dokumentiert den Entwurf von DecXtract sowie alle bei der Implementierung aufgetretenen Entscheidungsprobleme. Kapitel 6 erläutert die Qualitätssicherung von DecXtract mit Komponententests, Integrationstests und Systemtests. Kapitel 7 definiert zwei Evaluationsfragen und beschreibt die Evaluationsmethoden, Durchführung und Ergebnisse. Kapitel 8 fasst diese Arbeit zusammen und beschreibt die Ergebnisse der einzelnen Kapitel.

2 Grundlagen

Dieses Kapitel definiert die nötigen Grundlagen und Begriffe für diese Arbeit. Abschnitt 2.1 definiert den Softwareentwicklungsprozess *Continuous Software Engineering*. Abschnitt 2.2 beschriebt zwei Dokumentationsmodelle für Entscheidungswissen. Abschnitt 2.3 präsentiert die Motivation und Ansichten des ConDec-Projekts. Abschnitt 2.4 erläutert technische Grundlagen zur Klassifikation von natürlichsprachlichen Texten. Abschnitt 2.5 gibt einen Überblick und eine statistische Analyse der verwendeten Trainingsdaten.

2.1 Kontinuierliche Softwareentwicklung

Lehman definiert neun Gesetze zur Softwareevolution, die verdeutlichen, dass sich Softwaresysteme permanent an ihre Umgebung anpassen müssen. Diese Anpassungen sind nötig, wenn sich die Anforderungen an das Softwaresystem ändern, neue Technologien eingeführt werden oder sich die Umgebung des Softwaresystems auf andere Weise ändert. Eine entsprechende Quellcodeänderung implementiert die Anpassung eines Softwaresystems. Tabelle 2.1 zitiert die Gesetze eins, zwei und fünf, die sich auf Änderungen in Softwaresystemen beziehen [23, 21].

Tabelle 2.1: Lehmans Gesetzte zur Softwareevolution eins, zwei und fünf [21, 23].

Nr.	Lehmans Gesetz
1	Softwaresysteme müssen ständig angepasst werden, sonst werden sie immer weniger zufriedenstellend.
2	Mit zunehmender Entwicklung eines Softwaresystems nimmt die Komplexität zu, es sei denn, es wird an der Wartung oder Reduzierung gearbeitet.
5	Der Funktionsumfang von Softwaresystemen muss kontinuierlich erweitert werden, um die Benutzerzufriedenheit über die gesamte Lebensdauer zu erhalten.

EntwicklerInnen eines Softwaresystems müssen mit Unsicherheiten und sich ändernden Anforderungen umgehen. Agile Methoden bieten die nötige Flexibilität, solche Unsicherheiten bzw. sich ändernde Anforderungen zu adressieren. Continuous Software Engineeringist ein agiler Softwareentwicklungsprozess, der EndnutzerInnen durch schnelles Ausliefern der Software (Continuous Deployment) stark am Entwicklungsprozess beteiligt. Sie können regelmäßig Feedback zu einer aktuellen Version der Software geben. Um Continuous Deployment möglich zu machen, muss die Software zu jeder Zeit fehlerfrei und auslieferbar sein, das heißt sie muss eine hohe Softwarequalität besitzen. Dafür ist auch das Entscheidungswissen der EntwicklerInnen wichtig. Entscheidungs- und Nutzungswissen sind daher zwei wichtige Wissenstypen im kontinuierlichen Software-Engineering.

Krusche und Bruegge beschreiben das Prozessmodell „CSEPM“, das Entwicklungsaktivitäten als parallel laufende Workflows behandelt und eine individuelle Anpassung des Softwareprozesses ermöglicht [27]. Das Prozessmodell CSEPM beinhaltet statische Aspekte, die die Beziehungen zwischen spezifischen CSE-Konzepten beschreiben, einschließlich Reviews, Releases und Feedback. Es beschreibt auch den dynamischen Aspekt von CSE, z.B. wie Änderungsvorschläge neue Entwicklungsabläufe starten.

2.2 Modelle zur Dokumentation von Entscheidungswissen

Die Systemarchtitektur spielt im Entwicklungsprozess eines Softwaresystems eine wichtige Rolle und ist entscheident für den Erfolg des Projektes [10]. Designentscheidungen beschreiben die Systemarchitektur auf einer groben Ebene und sind besonders wichtig, da sie im späteren Verlauf nur sehr aufwändig zu ändern sind. Die Systemarchitektur kann als Menge aller Designentscheidungen betrachtet werden [8]. Eine Voraussetzung ist eine vollständige und verfolgbare Entscheidungsdokumentation [4].

Entscheidungswissen ist oft nur mangelhaft oder gar nicht dokumentiert [12]. Eine Schwachstelle ist die unstrukturierte Entscheidungsdokumentation. So wird Entscheidungswissen oft nur in Quellcode-Kommentaren oder in externen Systemen festgehalten, ohne eine Verlinkung zwischen zwei Artefakten herzustellen [1]. Es existieren verschiedene Entscheidungsdokumentationsmodelle, die Entscheidungswissen in strukturierter Form darstellen.

Kunz und Rittel definieren das *Issue-Based Information Systems* – Modell (IBIS), das drei Wissenstypen umfasst [9] (Abbildung 2.1). Ein *Issue* beschreibt das untersuchte Entscheidungsproblem. Eine *Position*[1] bezieht eine Stellung oder Meinungsäußerung zum verlinkten Entscheidungsproblem. Ein *Argument* spricht entweder für oder gegen eine *Position* und/oder *Issue*.

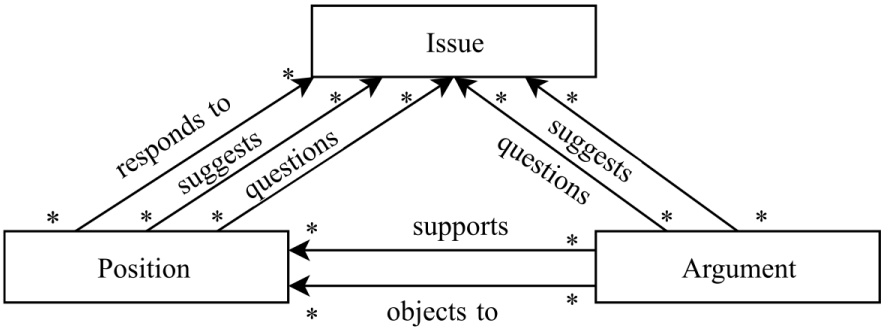


Abbildung 2.1: IBIS Dokumentationsmodell von Kunz und Rittel [9, 7].

Hesse et al. beschreiben ein ausführlicheres Dokumentationsmodell [1, 10] (Abbildung 2.2). Das Basiselement ist eine Entscheidung (*Decision*). Sie dient als Container für weitere Entscheidungselemente und ist einer *Person* zugewiesen. Weiterhin können Entscheidungen anderen Wissenselementen (*KnowledgeElement*) wie beispielsweise anderen Entscheidungen oder Anforderungen zugeordnet werden. Dabei sind alle Wissenstypen durch ein *DecisionComponent* beschrieben, die folgende Wissenstypen annehmen kann:

- Ein *Problem* wird durch einen Zustand beschrieben, der nicht zufriedenstellend ist bzw. sich nicht mit den Anforderungen vereinen lässt. Dieser Zustand ist zu beheben. Das Problem wird in einem *Issue* beschrieben. *Goal* definiert den Wunschzustand ohne Kenntnis des Problems.
- Eine *Solution* bezieht sich auf ein konkretes *Problem* und wird durch *Alternative* ausgedrückt. Gegebenenfalls sind *Claims* an eine Lösung gestellt, die über eine Problemlösung hinaus erfüllt werden müssen.
- Der *Context* definiert zusätzliche Bedingungen an eine Entscheidung. Diese werden durch *Assumtion*, *Constraint* und *Implication* einer Entscheidung beschrieben.

[1] Position: wird in dieser Arbeit als Alternative bezeichnet.

- *Rationale* werden durch ein Pro- und Kontra- *Argument* ausgedrückt, eine Meinung für oder gegen ein *DecisionComponent* vertreten.

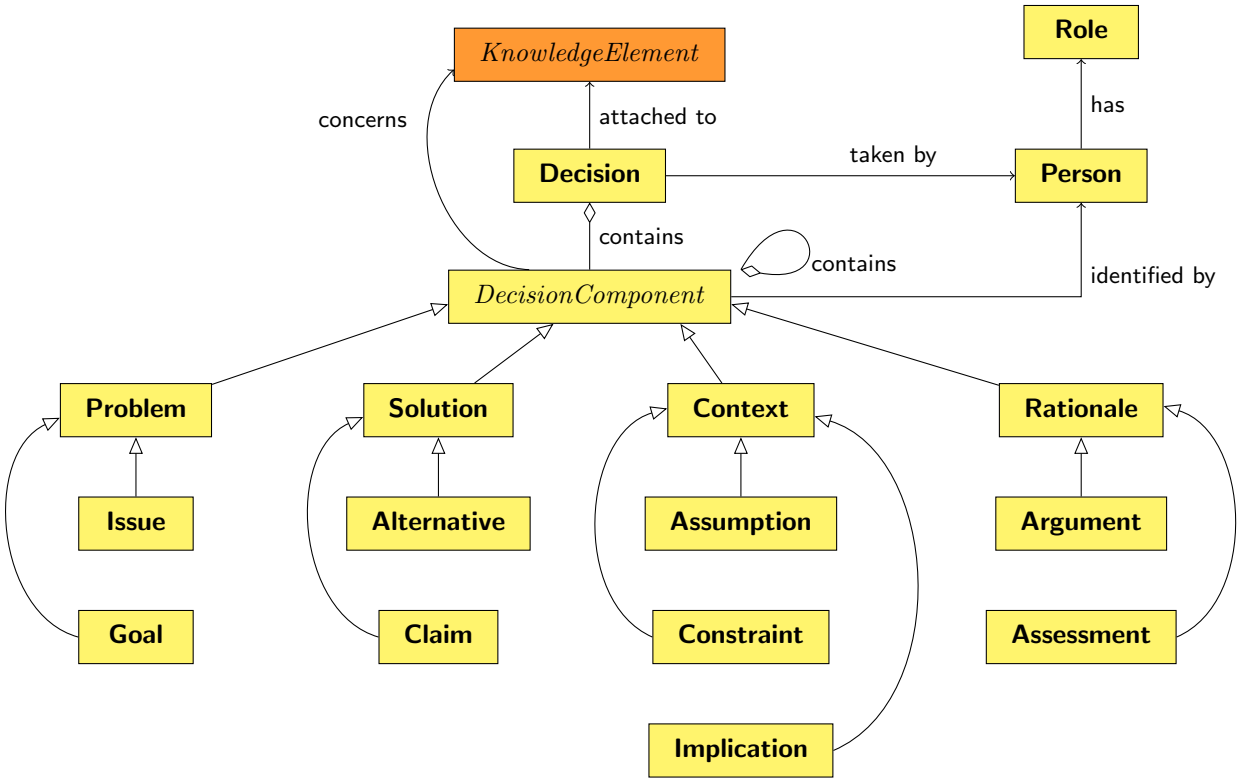


Abbildung 2.2: Entscheidungsdokumentationsmodell von Hesse et. al. [1]

2.3 Das ConDec-Projekt

Issue-Tracking-Systeme und Versionskontrollsysteme sind weit verbreite Werkzeuge, verfügen aber nicht über einen strukturierten Ansatz zur Integration von Entscheidungswissen. Kleebaum et al. sammeln Ideen und Anforderungen an eine Tool-Unterstützung zum Verwalten von Entscheidungswissen im kontinuierlichen Software-Engineering [5, 17]. Diese Ideen und Anforderungen werden im Forschungsprojekt CURES[2] fortlaufend umgesetzt und als Continuous Decision Knowledge Management (ConDec) bezeichnet. Dabei benutzt ConDec das Entscheidungsdokumentationsmodell von Hesse et al. und bietet verschiedene Möglichkeiten, Entscheidungswissen zu verwalten und zu betrachten [1].

ConDec stellt verschiedene Ansichten zur Verfügung, um Entscheidungswissen zu betrachten und zu verwalten. Ansicht WS1.3: Decision Knowledge View beinhaltet alle Elemente des Entscheidungswissens für ein Projekt. Diese Ansicht ermöglicht es einen Wissenstypen auszuwählen. Bei der Auswahl zeigt die Listenansicht (Abbildung 2.3 links) alle im Projekt vorhandenen Entscheidungselemente mit dem gewählten Wissenstypen. Bei Selektion eines Listenelementes zeigt die Baumansicht (Abbildung 2.3 rechts) den Entscheidungsbaum des gewählten Entscheidungselementes. Durch Drag and Drop können Entscheidungselemente untereinander verlinkt werden. Ein Kontextmenü auf Entscheidungselementen bietet weitere Funktionalitäten. Ansicht WS1.4.1: JIRA Issue Module (Abbildung 2.4) zeigt Entscheidungswissen zu vorhandenen JIRA Issues wie Work Items oder Systemfunktionen (Anforderungen). Die NutzerIn kann Wissen in dieser Ansicht ebenfalls verwalten.

[2]CURES-Projekt: http://www.dfg-spp1593.de/cures/ Zuletzt aufgerufen am 17.12.2018

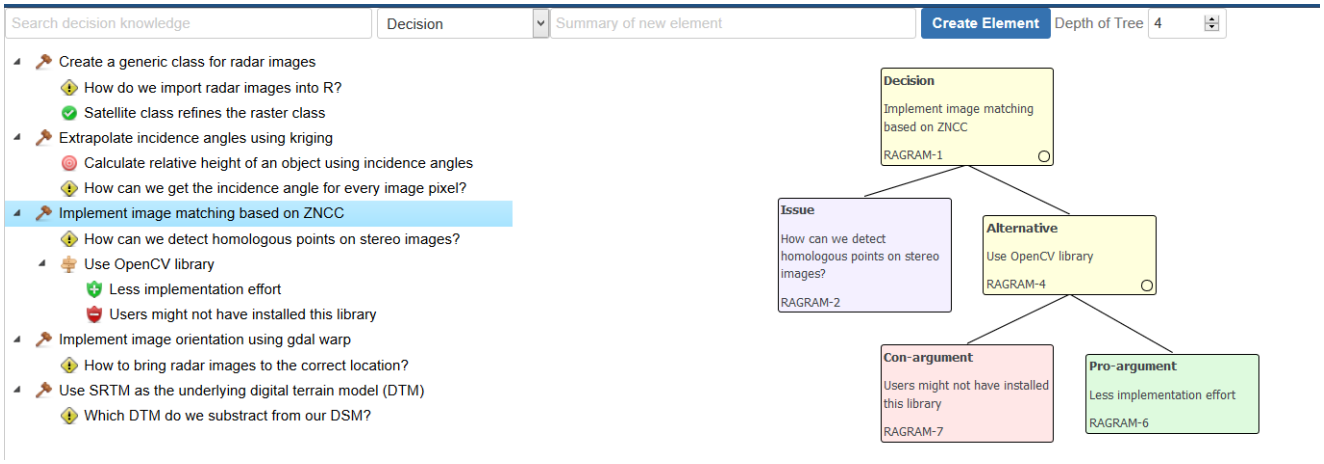


Abbildung 2.3: Ansicht WS1.3: Decision Knowledge Page in ConDec. Links: Listenansicht. Rechts: Baumansicht.

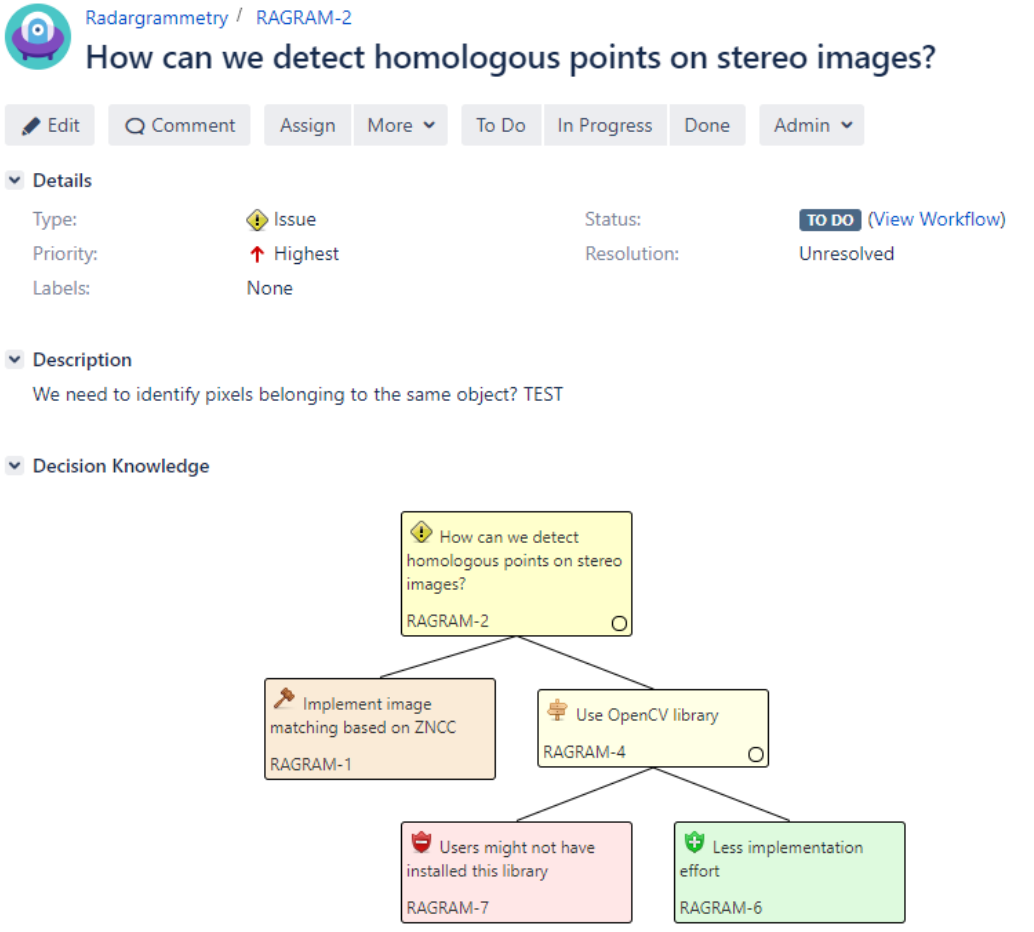


Abbildung 2.4: Ansicht WS1.4: Issue View und WS1.4.1: Issue Module im Abschnitt *Decision Knowledge*

2.4 Klassifikation von natürlichsprachlichen Texten

Im Bereich der Datenanalyse gruppiert ein Klassifikationsalgorithmus Objekte einer Datenmenge anhand korrelierender Attribute. Der Algorithmus sammelt Datenpunkte mit ähnlichen Attributen und fasst diese in einer Klasse zusammen. Ein Beispiel der Textklassifikation ist die Anwendung von Spamfiltern. Ein Spamfilter nutzt den Volltext einer E-Mail als Eingabe und klassifiziert diesen in die Klassen: *Spam* und *Kein Spam* [16].

Klassifikationsalgorithmen können entweder von EntwicklerInnenn mit definierten Regeln entwickelt werden oder orientieren sich am Prozess von Supervised-Machine-Learning (SML). SML Algorithmen nutzen eine Trainingsphase, um Regeln und Abhängigkeiten in einem bereits klassifizierten Datensatz zu finden. Im Beispiel des Spamfilters findet der SML-Algorithmus in der Trainingsphase Textmuster und Schlagwörter, deren Anwesenheit auf eine Spam-Nachricht hindeutet.

Nach Abschluss der Trainingsphase folgt die Evaluation des SML Algorithmus and von Evaluationsdaten. Dafür werden klassifizierte Daten, die nicht in der Menge von Trainingsdaten enthalten sind, erneut klassifiziert [11]. Die Evaluation quantifiziert die Güte des Algorithmus anhand von Metriken, um seine Mächtigkeit mit anderen Algorithmen zu vergleichen.

Zur Berechnugn der Metriken wird jedes Objekt vom SML Algorithmus klassifiziert und anschließend die tatsächliche Klasse (wahre Bedingung) mit der Klassifikation (vorhergesagte Bedingung) verglichen. Ist der Vergleich korrekt, wird die Klassifikation mit „true“ bezeichnet, ist sie inkorrekt mit „false“. Liegt eine korrekte Klassifikation einer positiven Klasse[3] (hier „kein Spam“) vor, wird die Klassifikation als „true positive“ bezeichnet. Die korrekte Klassifikation einer negativen Klasse[3] (hier „Spam“) analog als „True negative“. Tabelle 2.2 veranschaulicht diese Bezeichnungen sowie die Fälle einer inkorrekten Klassifikation: „False negative“ und „False positive“.

Tabelle 2.2: Darstellung der Ergebnistabelle eines binären Klassifikators

		Wahre Bedingung	
		positiv	negativ
Vorhergesagte Bedingung	positiv	True positive (TP)	False positive (FP)
	negativ	False negative (FN)	True negative (TN)

Anhand der vier möglichen Klassifikationsergebnisse lassen sich verschiedene Metriken ableiten, um die Güte eines Klassifikators zu beschreiben [14]. Tabelle 2.3 beschreibt drei Metriken, die in dieser Arbeit verwendet werden.

Ein trainierter und evaluierter Algorithmus wird auch als Modell bezeichnet. Diese Bezeichnung ist zutreffend, da alle Klassen und Attribute der Trainingsdaten bzw. des anzugehenden Problems abgebildet sind. Ändern sich die Trainingsdaten oder liegt eine neue größere Datenmenge vor, ist es angebracht den Algorithmus neu zu trainieren.

Die Java-Bibliothek Weka[4] von der University of Waikato stellt mehrere SML Algorithmen zur Verfügung. Weka kann über eine Oberfläche ausgeführt werden oder in ein Java System eingebunden werden. Es ist möglich, Daten aus verschiedenen Quellen (Datenbanken, Online Tabel-

[3] Positive/Negative Klasse: Die Unterscheidung zwischen positiver und negativer Klasse unterliegt der NutzerIn. So kann bei der Suche nach Spam E-Mails die Klasse positiv definiert werden, wenn eine E-Mail keinen Spam enthält. Welche Klasse letztendlich als positiv definiert ist, spielt allerdings keine Rolle. Diese muss nur konsistent gehalten werden.

[4] WEKA: https://www.cs.waikato.ac.nz/ml/weka/ Zuletzt aufgerufen am 17.12.2018

Tabelle 2.3: Metriken zur Beurteilung der Qualität eines Klassifikators

Name	Berechnung	Beschreibung
Precision	$\frac{TP}{TP+FP}$	Anteil der korrekt als positiv klassifizierten Objekte an allen als positiven klassifizierten Objekten.
Recall	$\frac{TP}{TP+FN}$	Anteil der korrekt als positiv klassifizierten Objekte an allen tatsächlich positiven Objekte.
F1-Score	$\frac{2}{\frac{1}{recall}+\frac{1}{precision}}$	Harmonisches Mittel zwischen Precision und Recall

len, etc.) einzubinden und zu verwalten. Es stehen verschiedene Algorithmen zur Verfügung die trainiert und evaluiert werden können. Als Ergebnis werden Metriken (Tabelle 2.3) berechnet.

2.5 Trainingsdatensatz

Um im späteren Projektverlauf eine natürliche Sprache in Wissenstypen zu klassifizieren, wird das JIRA-Projekt von Apache LUCENE als Trainingsdatensatz eingesetzt. Open-Source Projekte wie LUCENE haben die Eigenschaft, dass EntwicklerInnen und andere Projektbeteiligte, Probleme und Aufgaben in öffentlichen Foren (bspw. in JIRA-Issue-Kommentaren) diskutieren. Es wird angenommen, dass durch die öffentlichen Diskussionen viele Probleme besprochen, Meinungen ausgetauscht, bewertet und darüber hinaus viele Entscheidungen getroffen werden. Zudem erfüllt LUCENE die Bedingung, dass alle Commits mit dem betreffenden JIRA-Issue verlinkt sind. Dies erhöht die Verfolgbarkeit einzelner Entwicklungsartefakte. Alkadhi et al. nutzen das JIRA-Projekt von Apache LUCENE ebenfalls zum Training eines Klassifikators, der natürliche Sprache in Wissentypen klassifizieren kann [3, 7]. Der erzeugte Datensatz dieser Arbeiten erfüllt die Anforderungen wissenschaftlicher Veröffentlichung und wird auch für diese Arbeit benutzt.

2.5.1 LUCENE Projekt

LUCENE[5] ist eine von Apache entwickelte, frei verfügbare Bibliothek zur Volltextsuche. Die Bibliothek verspricht leistungsstarke Suchalgorithmen, die ressourcenschonend arbeiten. Die EntwicklerInnen von LUCENE verwenden GitHub[6] zum verteilten Arbeiten am Quellcode, sowie JIRA[7] zur Projektdokumentation. Seit 1997 wird LUCENE stetig weiterentwickelt und besteht derzeit[8] aus über 31.000 Commits in 113 Branches von 73 EntwicklerInnen sowie über 8.500 Issues in JIRA.

[5] Apache LUCENE: https://lucene.apache.org/core/ Zuletzt aufgerufen am 17.12.2018
[6] GitHub LUCENE: https://github.com/apache/lucene-solr Zuletzt aufgerufen am 17.12.2018
[7] LUCENE JIRA: https://issues.apache.org/jira/projects/LUCENE/issues Zuletzt aufgerufen am 17.12.2018
[8] Nobvember, 2018

2.5.2 Erzeugung des Trainingsdatensatzes

Der Datensatz von Alkadhi et al. besteht aus dem Text der Kommentare von 100 JIRA-Issues des JIRA-Projektes von LUCENE [7, 3]. Die Autoren teilten die Kommentare in Sätze und bestimmten den Wissenstyp für jeden Satz. Zusätzlich wird jedem Satz ein binäres Attribut *isRelevant* zugeordnet, das angibt, ob der Satz entscheidungsrelevante Informationen besitzt. Relevante Sätze besitzen zwingend einen Wissenstyp.

Dieser Datensatz liegt als SQL-Datenbank vor und enthält 2446 Sätze mit zugeordnetem Wissenstyp. Jedes Objekt besteht aus einem *Satz* sowie sechs Wahrheitswerten zur Identifikation des Wissenstyps. Tabelle 2.4 beschreibt die Wahrheitswerte der Datenbank. Die Wissenstypen dieses Datensatzes nutzen das IBIS Modell [9]. „Position“ wird im Datensatz mit *Alternative* bezeichnet. „Argument“ wird durch *Pro* und *Con* dargestellt. Der Präfix „is“ wird nur in Verwendung des Klassifikators benutzt.

Tabelle 2.4: Mögliche Klassen des Trainingsdatensatzes

Attribute	Beschreibung
isRelevant	Der Satz ist relevant für das aktuelle Entscheidungsproblem.
isAlternative	Alternativer Lösungsansatz zu einem Entscheidungsproblem.
isPro	Befürwortender Satz für ein Argument.
isCon	Widersprechender Satz gegen ein Argument.
isIssue	Beschreibt ein Entscheidungsproblem.
isDecision	Finale Entscheidung für eine Alternative.

Tabelle 2.5 beschreibt Beispielsätze des Datensatzes. Die ausgewählten Sätze sind ausschließlich den aufgeführten Wissenstypen zugeordnet. Darüber hinaus enthält der Datensatz auch Sätze, die mehr als einem Wissenstyp zugeordnet sind.

Tabelle 2.5: Klassifizierte Sätze aus dem LUCENE Projekt

Wissenstyp	Beispielsatz
Relevant	„Tests seem to pass, unfortunately Solr trunk is very unstable.“
Alternative	„Here is a patch.“
Pro	„Looks good to me.“
Con	„I tried, but then a test fails.“
Issue	„Are all the changes in this patch necessary to get the build to pass?“
Decision	„I committed this.“

2.5.3 Statistische Beschreibung des Trainingsdatensatzes

Tabelle 2.6 beschreibt die Anzahl der vorkommenden Wissenstypen im Trainingsdatensatz. Zeilen und Spalten drücken die vorliegenden Wissenstypen aus. Die Hauptdiagonale beschreibt die Anzahl des jeweiligen Wissenstyps. Der Datensatz enthält auch Sätze, die mehr als einen Wissenstypen enthalten. Diese bilden die Einträge unter der Hauptdiagonalen. Durch die Eigenschaften der einzelnen Wissenstypen scheint es logisch, dass gewisse Kombinationen nicht auftreten können. Bspw. *isPro* und *isCon* im selben Satz. Dennoch zeigt die Suche nach diesen Kombinationen, dass auch solche Sätze existieren (In Tabelle 2.6 gelb markiert).

Tabelle 2.6: Anzahl der jeweiligen Attribute von 2446 Sätzen im LUCENE Projekt

		isRelevant		isIssue		isAlternative		isDecision		isPro		isCon	
		0	1	0	1	0	1	0	1	0	1	0	1
isRelevant	0	666											
	1		1780										
isIssue	0	665	1522	2187									
	1	1	258		259								
isAlternative	0	663	955	1371	247	1618							
	1	3	825	815	12		828						
isDecision	0	665	1574	1982	257	1414	825	2239					
	1	1	206	205	2	204	3		207				
isPro	0	666	1237	1645	258	1261	642	1703	200	1903			
	1	0	543	542	1	357	186	536	7		543		
isCon	0	666	1496	1906	256	1422	740	1959	203	1670	492	2162	
	1	0	284	281	3	196	88	280	4	233	51		284

Tabelle 2.7 zeigt Beispiele fragwürdiger Attribut-Kombinationen. Die fraglichen Attribute sind rot gekennzeichnet. Drei Alternativen, eine Entscheidung und ein Issue sind als nicht relevant klassifiziert. Inhaltlich entsprechen diese Sätze jedoch dem zugeordneten Wissenstyp. Hier wurde der Eintrag im Attribut *isRelevant* falsch gesetzt. Die falsch gesetzten Attribute wurden für den weiteren Projektverlauf korrigiert. Zudem scheint es verwunderlich, dass 51 Sätze gleichzeitig Pro als auch Kontra Elemente darstellen. Eine genaue Betrachtung dieser Sätze zeigt sehr ausführliche Kommentare. Oft sind mehrere Sachverhalte in einem Satz kommentiert.

Tabelle 2.7: Sätze mit nicht logischen Attibut-Kombinationen.

Zuweisung von Wissenstyp zu Sätzen in LUCENE						Beispielsatz
isRelevant	isAlternative	isIssue	isPro	isCon	isDecision	
0	1	0	0	0	0	„Patch.“
0	1	0	0	0	0	„Here is a Patch.“
0	0	1	0	0	0	„Not that it's especially my business, but how did the snapshots get pushed historically to nexus if you didn't have access to nexus?“
0	0	0	0	0	1	„Committing shortly.“
1	1	0	1	1	0	„I wanted to avoid introducing another class (facet collections already use this primitive IntIterator), but maybe a ChildrenIterator with next() is simplest.“

3 Literaturrecherche

Dieses Kapitel beschreibt zwei Forschungsfragen und deren Beantwortung durch eine systematische Literaturrecherche. Abschnitt 3.1 definiert zwei genutzte Methoden zur Literaturrecherche. Abschnitt 3.2 beschreibt die Durchführung der Literaturrecherche und präsentiert die Ergebnisse. Die Synthese in Abschnitt 3.3 vergleicht die gefundenen Artikel anhand von vier Kategorien. Abschnitt 3.4 präsentiert die Erkenntnisse aus der Literaturrecherche für diese Arbeit.

Diese Arbeit betrachtet Quellcodeänderungen in Softwaresystemen. Eine systematische Literaturrecherche nach Kitchenham und Charters soll die Forschungsfrage beantworten, welches Wissen EntwicklerInnen benötigen, um Quellcodeänderungen zu verstehen [25]. Tabelle 3.1 beschreibt diese und daraus folgende Forschungsfragen (Ff) für die Literaturrecherche. Aufgrund der Ausrichtung dieser Arbeit wird besonders darauf geachtet, wie dokumentiertes Entscheidungswissen für das Verständnis von Quellcodeänderungen von EntwicklerInnen genutzt wird.

Tabelle 3.1: Forschungsfragen für die Literaturrecherche

Nr.	Forschungsfrage
Ff.1	Welches Wissen benötigen EntwicklerInnen zum Verständnis von Quellcodeänderungen?
Ff.1.2	Welches Entscheidungswissen benötigen EntwicklerInnen zum Verständnis von Quellcodeänderungen?

3.1 Methode

Die Literaturrecherche ermöglicht einen Überblick zum aktuellen Stand der Forschung der definierten Forschungsfragen. Die Methoden „schlagwortbasierte Suche“ und „Snowballing“ sollen geeignete Artikel finden [25, 26].

Die schlagwortbasierte Suche nutzt die Datenbanken ACM Digital Library[1] und IEEExplore[2]. Tabelle 3.2 definiert Suchterme zu jeder Forschungsfrage für eine Datenbankabfrage. Dabei nutzen alle Abfragen die Suchterme „code“ und „change“. Diese Suchterme sollen zu Artikeln führen, die speziell Quellcodeänderung untersuchen. Suchterm S.1 ergänzt diese Terme um „information“ und „needs“, um von EntwicklerInnen benötigtes Wissen abzufragen. Suchterm S.2 und S.3 nutzen die Terme „rationale“ bzw. „decision“ und „knowledge“, um zu erforschen, welche Rolle Entscheidungswissen beim Verständnis von Quellcodeänderungen spielt.

Snowballing bezeichnet die Verwendung der Referenzliste eines Artikels oder der Zitate eines Artikels zur Identifizierung zusätzlicher Artikel. Forward Snowballing bezeichnet dabei die Suche nach jüngeren Artikeln die einen vorliegenden Artikel zitieren. Backward Snowballing bezeichnet die Suche nach im vorliegenden Artikel zitierten Artikeln. Diese Methode hat den

[1] ACM Digital Library: https://dl.acm.org/ Zuletzt aufgerufen am 17.12.2018
[2] IEEExplore: https://ieeexplore.ieee.org/ Zuletzt aufgerufen am 17.12.2018

Tabelle 3.2: Suchterme der Abfragen für die schlagwortbasierte Suche

Id.	Ff	Suchterm	Zweck
S.1	1	(+code +change +information +needs)	Benötigtes Wissen zum Verständnis für Quellcodeänderungen
S.2	2	(+code +change +rationale)	Rolle von Entscheidungswissen beim Verständnis von Quellcodeänderungen
S.3	2	(+code +change +decision +knowledge)	

Vorteil, dass einzelne Artikel bereits im Kontext ähnlicher Forschungsfragen analysiert und bewertet wurden. Um Artikel durch Snowballing zu finden, wird zusätzlich die Datenbank Google Scholar[3] benutzt.

Tabelle 3.3 definiert acht Kriterien, um einen Artikel in die Auswahl relevanter Literatur für diese Arbeit einzuschließen. Kriterium Klr.1 bis Klr.5 prüfen die Verfügbarkeit und den Themenbereich der Artikel. Entspricht ein Artikel diesen Kriterien, wird dieser in die erste Auswahl eingeschlossen. Von dieser Auswahl wird die Zusammenfassung, die Ergebnisse und die Einleitung betrachtet. Führt der Artikel eine Befragung mit EntwicklerInnen durch (Kriterium Klr.6) und beschrieben diese drei gelesenen Kapitel relevante Sachverhalte zur Forschungsfrage (Kriterium Klr.7), wird diese in die erweiterte Auswahl eingeschlossen.

Die Artikel der erweiterten Auswahl werden gelesen und markiert. Trifft Kriterium Klr.8 zu, wird der Artikel der finalen Auswahl hinzugefügt. Erfüllt ein Artikel die geforderten Kriterien nicht, wird dieser von der Auswahl ausgeschlossen.

Tabelle 3.3: Relevanzkriterien für Artikelauswahl in der Literaturrecherche

Id.	Beschreibung
Klr.1	Der Artikel wurde einem Peer Review unterzogen.
Klr.2	Der Artikel ist in deutscher oder englischer Sprache verfügbar.
Klr.3	Der Artikel ist durch die genutzten Datenbanken frei verfügbar.
Klr.4	Der Artikel hat einen erkennbaren Bezug zur Softwareentwicklung.
Klr.5	Der Titel deutet auf eine Relevanz zur Forschungsfrage hin.
Klr.6	Der Artikel führt eine Befragung mit EntwicklerInnen zu Quellcodeänderungen durch.
Klr.7	Zusammenfassung, Einleitung und Ergebnis des Artikels beschreiben relevante Erkenntnisse zur Forschungsfrage.
Klr.8	Der Artikel liefert einen Beitrag zur Beantwortung der Forschungsfrage.

[3]Google Scholar: https://scholar.google.de/ Zuletzt aufgerufen am 17.12.2018

3.2 Durchführung und Ergebnisse

Tabelle 3.4 beschreibt die Ergebnisse der schlagwortbasierten Suche. Die Abfragen nutzen die Suchterme aus Tabelle 3.2. Abfrage A.1 resultiert in 387 Suchtreffern bei ACM und 156 bei IEEE. Die Überprüfung mit Kriterien Klr.1 bis Klr.5 resultiert in neun relevanten Artikeln. Nach Prüfung mit Klr.6 bis Klr.7 bleiben noch vier Artikel mit Relevanz für Ff.1. Abfrage A.3 resultiert in 28 Suchtreffern bei ACM und 20 bei IEEE. Die Überprüfung mit Kriterien Klr.1 bis Klr.5 resultiert in einem relevanten Artikel, der auch nach Prüfung mit Klr.6 bis Klr.7 Relevanz für Ff.1.2 zeigt. Abfrage A.4 resultiert in 54 Suchtreffern bei ACM und 48 bei IEEE. Die Überprüfung mit Kriterien Klr.1 bis Klr.5 resultiert in einem relevanten Artikel, der auch nach Prüfung mit Klr.6 bis Klr.7 Relevanz für Ff.1.2 zeigt.

Tabelle 3.4: Ergebnisse der Suche mit Suchtermen.

Id	Datum	Ff	Suchterme	Quelle	#Ergebnisse	#Relevant[a]	Referenzen[b]
A.1	Nov. 2018	1	S.1	ACM	387	7	[30] [32] [34]
				IEEE	156	2	[34]
A.3	Nov. 2018	2	S.2	ACM	28	1	[33]
				IEEE	20	0	–
A.4	Nov. 2018	2	S.3	ACM	54	4	–
				IEEE	48	0	–

[a]Erfüllt Kriterium Klr.1 bis Klr.6
[b]Erfüllt Kriterium Klr.1 bis Klr.7

Die gefundenen Artikel wurden anschließend mit Snowballing untersucht. Tabelle 3.5 zeigt die Durchführung und Ergebnisse. Die Suche in Backward Richtung zeigt dabei mehr Erfolg als die Forward Richtung. Die Betrachtung der gefundenen Artikel in Forward Richtung mit Kriterium Klr.5 und Klr.6 zeigt viele Artikel, die Ergebnisse der Studien zur Motivation eigener Thesen nutzen, aber keine eigene Studie durchführen. Beispielsweise untersuchen viele Artikel Fehlervorhersagen oder Änderungszusammenfassungen. Die Backward Richtung ist erfolgreicher und findet drei neue Artikel.

Tabelle 3.6 listet alle sieben für diese Arbeit gefundenen Artikel mit Autor und Titel auf. Tabelle 3.7 vergleicht diese Artikel mit Berücksichtigung auf Schlagworte, Kontext, Forschungsfrage, Ergebnisse und Wissenschaftliche Methodik.

Tabelle 3.5: Durchführung von Snowballing

Quelle	Richtung	# Zitate	# Relevant[a]	Referenzen[b]
[30]	Backward	35	3	–
	Forward	57	1	–
[32]	Backward	59	6	[36] [31]
	Forward	102	2	–
[33]	Backward	28	2	[35] [36]
	Forward	93	1	–
[34]	Backward	19	1	[31]
	Forward	11	0	–

[a]Erfüllt Kriterium Klr.1 bis Klr.6
[b]Erfüllt Kriterium Klr.1 bis Klr.7

Tabelle 3.6: Ergebnis der schlagwortbasierten Suche und Snowballing

Autor	Ref.	Titel
Shihab et al.	[30]	An industrial study on the risk of software changes.
Ko et al.	[31]	Information Needs in Collocated Software Development Teams.
Tao et al.	[32]	How do software engineers under stand code changes?
LaToza und Meyers	[33]	Hard-to-answer questions about code.
Kim	[34]	An exploratory study of awareness interests about software modifications
Fritz und Murphy	[35]	Using information fragments to answer the questions developers ask.
LaToza et al.	[36]	Maintaining mental models.

Tabelle 3.7: Vergleich aller gefundenen Artikel

Artikel	Schlagworte	Kontext & Motivation	Forschungsfrage	Idee / Ergebnisse	Methode
[30]	Change Risk, Change Metrics, Code Metrics, Bug Inducting Changes	Identifikation von Änderungen die zusätzliche Aufmerksamkeit durch Reviews oder Tests brauchen	Welche Faktoren haben Einfluss auf die Qualität einer Quellcodeänderung?	Klassifikator beurteilt Quellcodeänderung durch Betrachtung messbarer Faktoren	Befragung von über 450 EntwicklerInnen
[31]	Information Analysis, Information Needs, Requirements Tracing	Identifikation von benötigtem Wissen im Alltag von EntwicklerInnen	1) Welches Wissen suchen EntwicklerInnen? 2) Wo finden sie dieses Wissen? 3) Welche Situationen verhindern die Wissensbeschaffung?	Identifikation von 21 Wissenstypen, die EntwicklerInnen brauchen	Beobachtung und Befragung von 17 EntwicklerInnen
[32]	Code change, code review, information needs, tool support	Identifikation von benötigtem Wissen zum Verständnis von Quellcodeänderungen	Welches Wissen benötigten EntwicklerInnen zum Verständnis von Quellcodeänderungen?	Identifikation von 23 Fragen von EntwicklerInnen	Befragung von 180 EntwicklerInnen
[34]	Empirical study, code change, awareness interests	Identifikation von jüngsten Informationen zu Quellcodeänderung	Welche jüngsten Informationsbedürfnisse von Quellcodeänderungen haben EntwicklerInnen?	Identifikation und Bewertung von 20 Fragen von EntwicklerInnen	Befragung von 25 EntwicklerInnen
[33]	program comprehension, developer questions	Anforderungsanalyse für Toolsupport zur Unterstützung von EntwicklerInnen	Welche Fragen müssen EntwicklerInnen in ihrem Alltag beantworten?	21 Kategorien und 94 Fragen. Entscheidungswissen als häufigste Frage	Befragung von 179 EntwicklerInnen
[35]	human-centric software engineering, information fragments	Identifikation von benötigtem Wissen im Alltag von EntwicklerInnen	1) Welches Wissen suchen EntwicklerInnen im Alltag? 2) Wie kann die Suche nach Wissen unterstützt werden?	1) 78 Fragen die EntwicklerInnen stellen. 2) Information Retrieval Methoden zur Erkennung verwandter Artefakte	Interviews mit elf EntwicklerInnen zu fehlendem Tool Support
[36]	Design, Documentation, Experimentation, Human Factors	Identifikation von Kommunikationswegen von Wissen	1) Wie kommunizieren EntwicklerInnen Design Entscheidungen? 2) Welche Informationen tauschen EntwicklerInnen zu Design Entscheidungen aus?	1) 6 Kategorien mit 19 Face-to-Face Aussagen von EntwicklerInnen. 2) Entscheidungswissen als wichtigstes Wissen identifiziert.	Befragung und Interviews mit EntwicklerInnen

3.3 Synthese

Die Synthese sammelt gewonnene Erkenntnisse aus den gefunden Artikeln. Zu diesem Zweck beschreibt Tabelle 3.8 Fragen aus den gefundenen Artikeln.

Tabelle 3.8: Fragen von EntwicklerInnen zum Verständnis von Quellcodeänderungen.

	Thema	Frage	Begründung
[30]	AQ	Wie viele LOC wurden geändert?	Große Quellcodeänderungen sind Fehleranfällig
	AQ[a], AT	Wie viele Dateien wurden geändert?	Oft geänderte Dateien sind Fehleranfällig
	AQ, AT[b]	Wann wurde die Quellcodeänderung durchgeführt?	Quellcodeänderungen in der Nacht oder vor Deadlines sind Fehleranfällig
	AA[c], AT	Welche Schnittstellen sind betroffen?	API Änderungen können Auswirkungen auf Client Seite haben
	AQ, AA	Fixt diese Quellcodeänderung einen Bug?	Bugfixes sind aufwändig und Fehleranfällig
[31]	AQ	Hat meine/diese Quellcodeänderung Fehler?	Prüfung der Funktionalität bei Änderungsbetrachtung
	EW, AT	Gibt es Implikationen die von dieser Quellcodeänderung ausgehen?	Design Entscheidung zur Quellcodeänderung
	AA	Wie haben sich meine Komponenten geändert?	Informationen über Änderungen anderer Projektbeteiligter
	AA	Wie schwer ist es dieses Problem zu beheben?	Abschätzung der Notwendigkeit einer Änderung im aktuellen Projektstatus
	EW[d]	Warum wurde der Quellcode auf diese Weise entwickelt?	Erläuterung der Entscheidung zu dieser Quellcodeänderung
[32]	EW, AT	Welche Dokumentation existiert zu dieser Quellcodeänderung?	Notwendigkeit von Entscheidungswissen für Quellcodeänderungen
	EW	Was ist die Begründung für diese Quellcodeänderung?	Notwendigkeit von Entscheidungswissen für Quellcodeänderungen
	AQ, EW	Ist diese Änderung korrekt? / Wird der Fehler behoben?	Fehleranfälligkeit von Quellcodeänderungen
	AT, AW	Wie passen sich Methoden an, die die geänderte Methode aufrufen?	Korrektheit von Änderungsauswirkungen
	AQ, AT	Welche Testfälle sollen zur Verifikation dieser Quellcodeänderung ausgeführt werden?	Korrektheit der Änderungsauswirkung
[33]	EW	Warum wurde dieser Code auf diese Weise entwickelt?	Erläuterung der Entscheidung zu dieser Quellcodeänderung
	EW	Warum wurde dieser Code nicht anders entwickelt?	Erläuterung der Entscheidung zu dieser Quellcodeänderung
	EW, AA	War diese Änderung Gewollt, Versehentlich oder ein Hack?	Erläuterung der Umsetzung dieser Quellcodeänderung
	AA	Gibt es Implikationen die von dieser Quellcodeänderung ausgehen?	Design Entscheidung zur Quellcodeänderung
[34]	AA	Welche jüngsten Quellcodeänderung überschneiden sich mit meinen Quellcodeänderung?	Abschätzung der Auswirkung der Quellcodeänderung
	AA	Wie hat sich die Nutzung der API durch die Quellcodeänderung verändert?	Nötige Anpassung durch eine Quellcodeänderung
	EW	Warum wurde diese Quellcodeänderung auf diese Weise durchgeführt?	Erläuterung der Entscheidung zu dieser Quellcodeänderung
[35]	EW, AT	Warum wurde die Quellcodeänderung durchgeführt?	Notwendigkeit der Quellcodeänderung
	AT, AA	Welche Klassen haben sich geändert?	Abschätzung der Auswirkung der Quellcodeänderung
	AA	Wer benutzt diese API, die ich ändern möchte?	Abschätzung der Auswirkung der Quellcodeänderung
	AA	Welche jüngsten Quellcodeänderungen sind relevant für mich?	Informationen über Änderungen anderer Projektbeteiligen
[36]	EW, AT	Gibt es Implikationen die von dieser Quellcodeänderung ausgehen?	Design Entscheidung zur Quellcodeänderung
	AA	Welche Auswirkungen hat eine Quellcodeänderung an dieser Stelle?	Abschätzung der Auswirkung der Quellcodeänderung

[a] AQ: Qualität der Quellcodeänderung
[b] AT: Verlinkung der Quellcodeänderung
[c] AA: Auswirkungen der Quellcodeänderung
[d] EW: Entscheidungswissen zur Quellcodeänderung

Aus den Fragen in Tabelle 3.8 lassen sich zwei Kategorien ableiten. Kategorie K.1 stellt Fragen einer EntwicklerIn zum Verständnis von Quellcodeänderungen. Dafür finden sich die Unterkategorien Qualität, Änderungsauswirkung und Verbundenheit. Kategorie K.2 definiert die Rolle von Entscheidungswissen zum Verständnis von Quellcodeänderungen. Tabelle 3.9 beschreibt diese Kategorien und Unterkategorien.

Tabelle 3.9: Kategorien zur Betrachtung der gefundenen Artikel

Nr.	Beschreibung
K.1	Fragen einer EntwicklerIn zum Verständnis von Quellcodeänderungen
AQ	Die erste Kategorie ist die Qualität der Quellcodeänderung. Je besser die Qualität der Änderung ist, desto leichter ist diese nachvollziehbar.
AA	Die zweite Kategorie ist die Änderungsauswirkung einer Quellcodeänderung. Um eine Quellcodeänderung zu verstehen, kann es nötig sein die Kern-Änderung und die von ihr verursachten Änderungsauswirkungen zu identifizieren.
AT	Die dritte Kategorie ist die Verbundenheit des dokumentierten Wissens. Für eine Quellcodeänderung soll ein Link zu einer Aufgabenbeschreibung oder Fehlerbericht vorliegen.
K.2	Rolle von Entscheidungswissen zum Verständnis von Quellcodeänderungen. Dabei soll sowohl auf den Inhalt, die Präsentation und Verfügbarkeit von Entscheidungswissen geprüft werden.

Es ist auffällig, dass manche Fragen mehreren Kategorien zugeordnet werden können. Bei genauerer Betrachtung der einzelnen Kategorien finden sich zudem Überschneidungen. So ist die Qualität einer Quellcodeänderung auch mit der Qualität der durchgeführten Änderungsauswirkung betroffen. Entscheidungswissen und Verlinkung überschneiden sich ebenso, da Entscheidungswissen zu einer Quellcodeänderung verlinkt sein muss.

3.4 Erkenntnisse für diese Arbeit

Der folgende Abschnitt beantwortet die Forschungsfragen (Tabelle 3.1 Seite 13) und leitet Erkenntnisse für diese Arbeit ab.

Die Literaturrecherche liefert sieben Artikel, die Umfragen mit EntwicklerInnen zum Informationsbedarf zum Verständnis von Quellcodeänderungen haben. Die Ergebnisse werden in Fragen präsentiert, die EntwicklerInnen zu einer Quellcodeänderung stellen. Aus diesen Fragen lassen sich die Unterkategorien: Qualität, Änderungsauswirkung und Verlinkung ableiten. Tabelle 3.10 beschreibt alle Fragen unter Berücksichtigung der einzuordnenden Kategorie.

Aus allen gelesenen Artikeln wird deutlich, dass EntwicklerInnen Entscheidungswissen benötigen, dieses aber nicht dokumentiert bzw. nicht auffindbar ist. Daraus lassen sich zwei Herausforderungen an diese Arbeit ableiten:

1. Eine Quellcodeänderung ist mit vorhandenem Entscheidungs- und Projektwissen verlinkt.

2. EntwicklerInnen können Entscheidungswissen zu Quellcodeänderungen leicht dokumentieren.

Tabelle 3.10: Fragen einer EntwicklerIn zum Verständnis von Quellcodeänderungen

Nr.	Kategorie	Frage
1	AQ[a]	Wie viele LOC wurden geändert? [30]
2	AQ, AT[b]	Wie viele Dateien wurden geändert? [30]
3	AA[c], AT	Welche Schnittstellen sind betroffen? [30]
4	AQ, AA	Fixt diese Quellcodeänderung einen Bug? [30]
5	AQ	Hat meine/diese Quellcodeänderung Fehler? [31]
6	AA	Wie haben sich meine Komponenten geändert? [31]
7	AA	Wie schwer ist es, dieses Problem zu beheben? [31]
8	AT, AA	Wie passen sich Methoden an, die die geänderte Methode aufrufen? [32]
9	AQ, AT	Welche Testfälle sollen zur Verifikation dieser Quellcodeänderung ausgeführt werden? [32]
10	AA	Überschneiden sich jüngste Quellcodeänderungen mit meinen Quellcodeänderung? [34][35]
11	AA	Wie hat sich die Nutzung der API durch die Quellcodeänderung verändert?[34][35]
12	AT, AA	Welche Klassen haben sich geändert? [35]
13	AA	Welche Auswirkungen hat eine Quellcodeänderung an dieser Stelle? [36]
14	EW[d], AQ, AT	Wann & Warum wurde die Quellcodeänderung durchgeführt? [30][35]
15	EW, AA	War diese Änderung Gewollt, Versehentlich oder ein Hack? [33]
16	EW	Warum wurde dieser Code nicht anders entwickelt? [33]
17	EW, AQ	Ist diese Änderung korrekt? / Wird der Fehler behoben? [32]
18	EW	Was ist die Begründung für diese Quellcodeänderung?[32]
19	EW, AT	Welche Dokumentation existiert zu dieser Quellcodeänderung? [32]
20	EW	Warum wurde der Quellcode auf diese Weise entwickelt? [31][33][34]
21	EW, AT	Gibt es Implikationen die von dieser Quellcodeänderung ausgehen? [36][31][33]

[a] **AQ** : Qualität der Quellcodeänderung
[b] **AT** : Verlinkung der Quellcodeänderung
[c] **AA** : Auswirkungen der Quellcodeänderung
[d] **EW** : Entscheidungswissen zur Quellcodeänderung

4 Anforderungen

Das folgende Kapitel beschreibt Anforderungen an DecXtract. Abschnitt 4.1 benennt Personae in den Rollen der EndnutzerInnen. Abschnitt 4.2 beschreibt funktionale Anforderungen durch die Aufgaben der NutzerInnen (User Task (UT) und Subtask (ST)) sowie die Systemfunktionen (SF), die DecXtract anbietet, um die NutzerIn zu unterstützen. Abschnitt 4.3 definiert nichtfunktionale Anforderungen durch messbare Werte, die DecXtract erfüllen muss. Abschnitt 4.4 beschreibt alle Systemfunktionen im Oberflächenkontext mit einem UI Strukturdiagramm. Abschnitt 4.5 listet alle Anforderungen tabellarisch auf.

Die Anforderungserhebung erfolgte durch Betrachtung des Ausschreibungsdokumentes. Zudem wurden iterative Experimente mit Prototypen und Mock Ups durchgeführt. Diese simulierten die Verwendung von DecXtract der einzelnen Persona und zeigten weitere Anforderungen für die jeweiligen Rollen.

4.1 Personae

Die nachfolgenden Personae beschreiben die Rollen der NutzerInnen durch eine konkrete fiktive Person. DecXtract soll die Person in ihrer Rolle in ihrem Arbeitsalltag unterstützen. Dafür sind typische Frustrationen und Bedürfnisse beschrieben. Personae helfen der EntwicklerIn während der Anforderungsanalyse und Entwicklung, sich in den Alltag der EndnutzerInnen zu versetzen [19].

Tabelle 4.1: Persona: Projekt-AdministratorIn

Job	Projekt-AdministratorIn
Biographie	35 Jahre. Master in Computer Science. Arbeitet für eine kleine Firma, deren Softwareentwicklungsprozess schnell neue Versionen an die EndnutzerInnen ausliefert und deren Feedback stark einbezieht. Die Firma betreibt Continuous Software Engineering (CSE). Benutzt Eclipse als IDE, Git als Versionskontrollsystem und JIRA als Issue Tracking System.
Wissen	Arbeitet seit 5 Jahren mit JIRA. Hat bereits in agilen und anderen CSE Projekten gearbeitet. Weiß, dass Entscheidungswissen ein entscheidender Faktor für landlebende Softwaresysteme sind.
Bedürfnisse	Einfache Möglichkeit Entscheidungswissen in einem Projekt zu verwalten.
Frustration	Verlust von Entscheidungswissen durch fehlende Toolunterstützung. Widerwillige EntwicklerInnen, die keine Dokumentation anfertigen.
Wünsche	Möglichkeit Entscheidungswissen in typischen CSE Artefakten wie Commit-Nachrichten und Issue-Kommentaren für JIRA-Issues festzuhalten.

Tabelle 4.2: Persona: Dokumentationsbeauftragte für Entscheidungswissen

Job	Dokumentationsbeauftragte für Entscheidungswissen (Rationale-ManagerIn)
Biographie	31 Jahre. Master in Kommunikationswissenschaften. Arbeitet für eine große Software Firma. Ist in viele verschiedene Projekte eingebunden. Benutzt JIRA Dashboards und Git-Konsolen, um alle neuen JIRA-Issues und Entscheidungselemente im Blick zu behalten.
Wissen	Arbeitet seit 5 Jahren mit JIRA. Betreut(e) viele Projekte als Rationale-ManagerIn.
Bedürfnisse	Entscheidungswissen in Issue Kommentaren mit Entscheidungswissen in Issues zu verbinden. Sämtliches Entscheidungswissen zu einem JIRA-Issue untersuchen.
Frustration	Projektbeteiligte, die ihre Entscheidungen nicht in JIRA-Issues und Commit-Nachrichten dokumentieren.
Wünsche	Möchte Entscheidungswissen in JIRA-Issue-Kommentaren und Commit-Nachrichten in JIRA-Issues exportieren zu können.

Tabelle 4.3: Persona: EntwicklerIn

Job	EntwicklerIn
Biographie	22 Jahre. Ausgebildete Software Entwicklerin. Arbeitet für eine große Softwarefirma, die Anforderungen und Aufgaben seit 2005 in JIRA dokumentiert. Benutzt IntelliJ als IDE und ist sehr erfahren in Java und JavaScript.
Wissen	Arbeitet seit 2 Monaten als EntwicklerIn in dieser Firma. Hat vorher nie JIRA benutzt, Entscheidungen dokumentiert oder Commit-Nachrichten geschrieben.
Bedürfnisse	Einfach nutzbare Plug-Ins, mit guter Dokumentation und Tutorials.
Frustration	Keine Hilfestellung zu Plug-Ins.
Wünsche	Eine Wiki Seite mit ausführlicher Dokumentation zum Plug-In und allen Features.

Tabelle 4.4: Persona: Anforderungsbeauftragte

Job	Anforderungsbeauftragte
Biographie	40 Jahre. Master of Science in Informatik. Arbeitet für ein großes Softwareunternehmen, das seit 2005 alle Anforderungen und Entwicklungsaufgaben in JIRA verwaltet. Arbeitet viel mit JIRA, Word und Excel.
Wissen	Arbeitet seit 12 Jahren als Anforderungsbeauftragte. Nutzt JIRA seit 5 Jahren
Bedürfnisse	Muss Entscheidungen zur Erhebung und Umsetzung von Anforderungen dokumentieren.
Frustration	Lange Ladezeiten. Komplizierte Nutzung (zu viele Klicks für die Verwaltung von Anforderungen und Entscheidungswissen)
Wünsche	Entscheidungswissen in informellen Dokumenten markieren,und verknüpfen.

4.2 Funktionale Anforderungen an DecXtract

Dieser Abschnitt beschreibt alle Anforderungen an die Funktionalität von DecXtract. Der dafür definierte User Task beschreibt die Aufgabe, bei der NutzerInnen unterstützt werden sollen. Dieser wird zusätzlich mit Sub Tasks, Systemfunktionen und User Storys verfeinert.

4.2.1 Aufgaben der NutzerInnen

UT1 – Software entwickeln
Anforderungsingenieure erheben und beschreiben Anforderungen für eine Software. Diese Anforderungen werden von EntwicklerInnen implementiert. Während der Erhebung und Implementierung der Anforderungen treffen alle projektbeteiligten Personen Entscheidungen. Diese sind entweder explizit in entsprechenden Entscheidungsartefakten angelegt oder werden implizit ohne konkrete Dokumentationsvorschrift in JIRA-Issue-Kommentaren und Commit-Nachrichten festgehalten.

ST1 – Software-Entwicklungsprozess für das Projekt definieren
ProjektadministratorInnen definieren den Softwareentwicklungsprozess für das Projekt. Sie entscheiden darüber, wie Entscheidungswissen im Softwareentwicklungsprojekt verwaltet wird. Zu diesem Zweck konfigurieren und verwalten die Projektadministratoren und SystemadministratorInnen die Funktionen des Plug-Ins.

ST2 – Anforderungen erheben, dokumentieren, ändern, verwalten
Anforderungsbeauftragte ermitteln und dokumentieren Anforderungen an die Software. Dabei treffen sie Entscheidungen.

ST3 – Code implementieren
EntwicklerInnen implementieren Quellcode anhand der definierten Anforderungen. Alle Design- und Entwicklungsentscheidungen werden entsprechend des festgelegten Prozesses dokumentiert.

ST4 – Änderungsauswirkungen analysieren
Die EntwicklerIn plant eine Änderung anhand einer neuen oder geänderten Anforderungen bzw. einer geänderten Entwurfsentscheidung. Dabei identifiziert sie neben den direkt zu ändernden Softwareartefakten auch die Softwareartefakte (Code, Anforderungen, Entscheidungen), die von der Änderung betroffen sind.

ST5 – Qualität des dokumentierten Entscheidungswissens analysieren
Die Dokumentationsbeauftragte für Entscheidungswissen pflegt dokumentiertes Entscheidungswissen sodass es konsistent in hoher Qualität vorliegt.

4.2.2 Systemfunktionen und User Stories

Die nachfolgenden Systemfunktionen beschreiben die Aufgaben von DecXtract aus Sicht des Systems. Dafür werden die notwendigen Ein- und Ausgaben, Vor- und Nachbedingungen sowie Ausnahmen und Regeln definiert. Des Weiteren wird die Motivation und Rolle der NutzerIn durch eine User Story für jede Systemfunktion beschrieben.

Die Reihenfolge der Systemfunktionen hat keine Aussage über Priorität und Ablauf.

4.2.2.1 DecXtract aktivieren

Als eine Projekt-AdministratorIn möchte ich in den Projekteinstellungen DecXtract aktivieren und deaktivieren, sodass ich entscheiden kann, wann das Plug-In für mein Projekt genutzt wird. Tabelle 4.5 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask ST1.

Tabelle 4.5: SF1: DecXtract aktivieren

Vorbedingung	JIRA-Projekt existiert, ConDec ist aktiviert.
Input	Projekt Schlüssel, Extraktion aktiviert?
Nachbedingung	DecXtract entweder aktiviert oder deaktiviert
Ausgabe	Erfolgs- oder Fehlermeldung
Ausnahmen	Projektschlüssel existiert nicht in der Datenbank.

4.2.2.2 Entscheidungswissen in JIRA-Issue-Kommentaren automatisch klassifizieren

Als eine EntwicklerIn möchte ich Entscheidungswissen in JIRA-Issue-Kommentaren automatisiert klassifizieren lassen, um selbst keine Sätze in Wissenstypen klassifizieren zu müssen. Tabelle 4.6 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask ST3.

Tabelle 4.6: SF2: Entscheidungswissen in JIRA-Issue-Kommentaren automatisch klassifizieren

Vorbedingung	Issue existiert mit mindestens einem Kommentar.
Input	JIRA-Issue
Nachbedingung	Klassifikation ist intern gespeichert.
Ausgabe	Text ist als relevantes Entscheidungselement mit bestimmten Wissenstypen oder als nicht relevant gekennzeichnet.
Regeln	Wird ein Satz als relevant klassifiziert, wird der Wissenstyp klassifiziert. Speziell formatierter Text wird nicht klassifiziert. Bereits klassifizierter Text wird nicht klassifiziert.

4.2.2.3 Entscheidungswissen in JIRA-Issue-Kommentaren manuell klassifizieren

Als eine EntwicklerIn möchte ich Entscheidungswissen in JIRA-Issue-Kommentaren markieren, um Sätze in entsprechende Wissenstypen zu klassifizieren. Tabelle 4.7 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask ST3.

Tabelle 4.7: SF3: Entscheidungswissen in JIRA-Issue-Kommentaren manuell klassifizieren

Vorbedingung	Issue existiert mit mindestens einem Kommentar.
Input	Satz bzw. Sätze/Teile aus JIRA-Issue-Kommentar und Wissenstyp.
Nachbedingung	Klassifikation ist intern gespeichert
Ausgabe	Entscheidungswissen ist im Text gekennzeichnet.
Ausnahmen	Die Klassifikation ist unzulässig.
Regeln	Markierte Sätze werden nicht klassifiziert. Die Markierung muss nach der Vorschrift des Systems erfolgen.

4.2.2.4 Entscheidungswissen in JIRA-Issue-Kommentaren anzeigen

Als eine EntwicklerIn oder Rationale-ManagerIn möchte ich Entscheidungswissen in JIRA-Issue-Kommentaren einsehen, sodass ich einen Überblick über alle Entscheidungselemente zu diesem JIRA-Issue habe. Tabelle 4.8 beschreibt die Systemfunktion zu dieser Anforderung.

Tabelle 4.8: SF4: Entscheidungswissen in JIRA-Issue-Kommentaren anzeigen

Vorbedingung	JIRA-Issue existiert mit in Kommentaren klassifiziertem Entscheidungswissen.
Input	JIRA-Issue
Nachbedingung	Entscheidungswissen ist angezeigt.
Ausgabe	Entscheidungswissen in übersichtlicher Darstellung
Ausnahmen	Projekt existiert nicht. Keine Kommentare vorhanden.
Regeln	Nur lesende Ansicht

4.2.2.5 Entscheidungswissen in JIRA-Issue-Kommentaren im Kontext von weiterem Entscheidungswissen, Projektwissen und Systemwissen anzeigen

Als eine EntwicklerIn oder Rationale-ManagerIn möchte ich Entscheidungswissen in JIRA-Issue-Kommentaren im Kontext von weiterem Wissen einsehen, sodass ich alle vorhanden Entscheidungselemente ansehen und verwalten kann. Tabelle 4.9 beschreibt die Systemfunktion zu dieser Anforderung.

Tabelle 4.9: SF5: Entscheidungswissen in JIRA-Issue-Kommentaren im Kontext von weiterem Entscheidungswissen, Projektwissen und Systemwissen anzeigen

Vorbedingung	Projekt existiert, Entscheidungswissen ist vorhanden
Input	JIRA-Projekt
Nachbedingung	Entscheidungswissen ist angezeigt
Ausgabe	Entscheidungswissen in übersichtlicher Darstellung
Ausnahmen	Projekt existiert nicht. Keine Entscheidungselemente vorhanden.

4.2.2.6 Automatisch sowie manuell klassifiziertes Entscheidungswissen bearbeiten

Als eine EntwicklerIn möchte ich klassifiziertes Entscheidungswissen bearbeiten, um Fehler der automatischen und manuellen Klassifikation zu korrigieren. Damit erzeuge ich textuell korrektes Entscheidungswissen, das vollständig dokumentiert und verlinkt ist. Der Wissenstyp gibt die Aussage des Textes wieder. Tabelle 4.10 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask ST5.

Tabelle 4.10: SF6: Automatisch sowie manuell klassifiziertes Entscheidungswissen bearbeiten

Vorbedingung	JIRA-Issue existiert mit in Kommentaren klassifiziertem Entscheidungswissen
Input	Ausgewähltes Entscheidungselement, gewünschte Änderung
Nachbedingung	Klassifikation ist intern gespeichert. Es wird vermerkt, wenn Entscheidungswissen durch die NutzerIn markiert wurde.
Ausgabe	Erfolgs oder Fehlermeldung.

4.2.2.7 Entscheidungswissen in JIRA-Issue-Kommentaren verlinken, Verlinkung lösen

Als eine EntwicklerIn möchte ich Entscheidungswissen in JIRA-Issue-Kommentaren mit anderem Wissen verlinken, um eine Entscheidung vollständig und korrekt zu dokumentieren. Tabelle 4.11 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask ST5.

Tabelle 4.11: SF7: Entscheidungswissen in JIRA-Issue-Kommentaren verlinken, Verlinkung lösen

Vorbedingung	JIRA-Issue existiert mit in Kommentaren klassifiziertem Entscheidungswissen
Input	Startelement mit Dokumentationsort, Zielelement mit Dokumentationsort.
Nachbedingung	Verlinkung ist intern gespeichert.
Ausgabe	Verlinkung ist auf der Oberfläche sichtbar.
Ausnahmen	Start oder Ziel Element existiert nicht.

4.2.2.8 Explizites Entscheidungswissen zu JIRA-Issue hinzufügen

Als eine Entwicklerin möchte ich existierendes Entscheidungswissen mit neuen Entscheidungselementen verknüpfen. Dabei möchte ich bestimmen, ob neues Entscheidungswissen als JIRA-Issue oder als JIRA-Issue-Kommentar gespeichert wird, sodass neue Entscheidungselemente entsprechend ihres Kontext verfügbar sind. Tabelle 4.12 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask ST4.

Tabelle 4.12: SF8: Explizites Entscheidungswissen zu JIRA-Issue hinzufügen

Vorbedingung	JIRA-Issue existiert.
Input	JIRA-Issue, Beschreibung, Wissenstyp, Dokumentationsort
Nachbedingung	Neues Entscheidungselement ist erstellt und verlinkt.
Ausgabe	Neues Entscheidungselement, entsprechend gewähltem Dokumentationsort.
Ausnahmen	JIRA-Issue existiert nicht.

4.2.2.9 Dokumentationsort von Entscheidungswissen verwalten

Als eine Dokumentationsbeauftragte für Entscheidungswissen möchte ich den Dokumentationsort von Entscheidungswissen verändern, sodass es möglich ist verknüpftes Entscheidungswissen in JIRA zu exportieren. Tabelle 4.13 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask ST5.

Tabelle 4.13: SF9: Dokumentationsort von Entscheidungswissen verwalten

Vorbedingung	Projekt existiert, Entscheidungswissen ist vorhanden
Input	JIRA-Projekt
Nachbedingung	neues JIRA-Issue mit Entscheidungswissen
Ausgabe	neues JIRA-Issue Projekt
Ausnahmen	Quelle des Entscheidungselements existiert nicht.

4.2.2.10 Metriken zur Beurteilung der Vollständigkeit des Entscheidungswissens berechnen

Als eine Rationale ManagerIn möchte ich Metriken zur Verbundenheit des Entscheidungswissens einsehen, um zu beurteilen, wie vollständig die getroffenen Entscheidungen in meinem Projekt dokumentiert sind. Tabelle 4.14 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask ST5.

Tabelle 4.14: SF10: Metriken zur Beurteilung der Vollständigkeit des Entscheidungswissens berechnen

Vorbedingung	Projekt existiert, Entscheidungswissen ist vorhanden
Input	Alle Entscheidungselemente
Nachbedingung	Metriken sind berechnet.
Ausgabe	Metriken in geeigneter Form
Ausnahmen	Projekt existiert nicht. Keine Entscheidungselemente vorhanden.

4.2.3 Domänendaten

Das Domänendatenmodell in Abbildung 4.1 beschreibt die Beziehung zwischen JIRA-Issues, Quellcode, Kommentaren und Sätzen. Dabei stellen JIRA-Issues in Sätze in Kommentaren von JIRA-Issues ein Entscheidungselement dar. Jedes Entscheidungselement besitzt einen Wissenstyp nach dem IBIS Modell. DecXtract behandelt Sätze in einem JIRA-Issue-Kommentar als Entscheidungselemente. Zur Identifizierung bekommen Sätze eine Id-Nummer. Die Beziehung zwischen JIRA-Issue und Entscheidungselement ist bereits in ConDec realisiert. JIRA-Entitäten sind Blau dargestellt. Die ConDec Implementierung in Orange. Grün bezeichnet Information aus einem Git Repository. Die hier aufgezeigten Wissenstypen orientieren sich am IBIS Modell aus Abbildung 2.1 und [9].

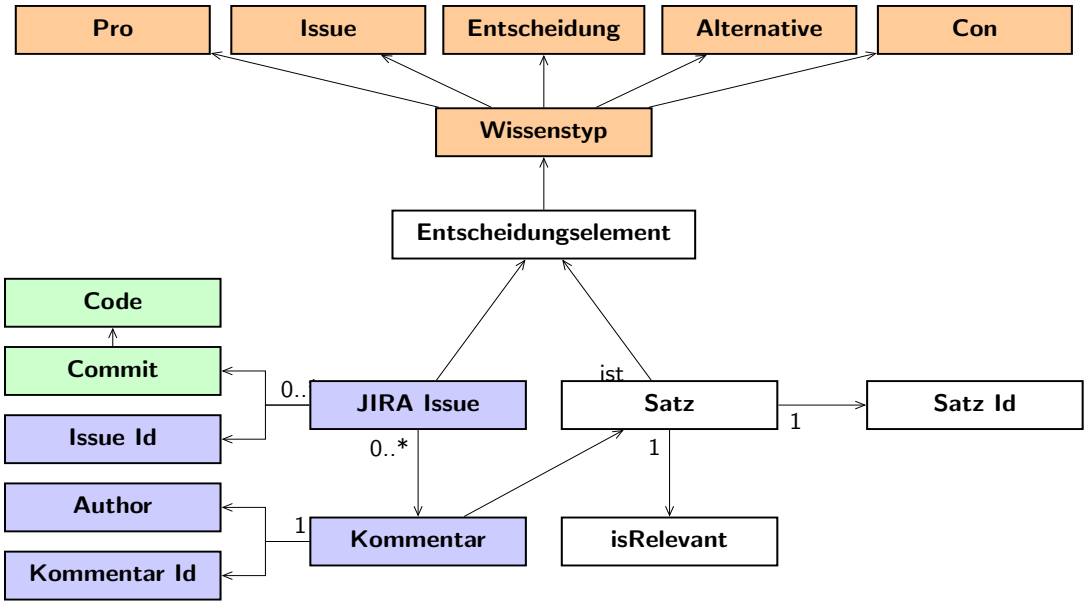


Abbildung 4.1: Domänendatenmodell zur Beziehung zwischen JIRA-Issues und Sätzen mit Entscheidungswissen

4.3 Nichtfunktionale Anforderungen an DecXtract

Nichtfunktionale Anforderungen (NFR) beschreiben, *wie gut* eine funktionale Anforderung entwickelt werden bzw. arbeiten muss. Die Kategorien der NFR orientieren sich an [18]. Die Beschreibung stellt eine Einschränkung bei der Umsetzung der Anforderungen dar. Der Zeitpunkt beschreibt den Projektabschnitt, in dem die NFR zu beachten ist. Die Tabellen 4.15, 4.16, 4.17, 4.18, 4.19, 4.20, 4.21 und 4.22 definieren die nichtfunktionalen Anforderungen.

Tabelle 4.15: NFR: Funktionale Eignung: Funktionale Vollständigkeit

Kategorie	Funktionale Eignung: Funktionale Vollständigkeit (Functional Suitability: Functional Completeness)
Beschreibung	Das entwickelte System bildet alle dokumentierten Anforderungen ab. Die Anforderungen decken alle Aspekte des Ausschreibungsdokumentes ab. Die dokumentierten Anforderungen machen keine Impliziten annahmen.
Zeitpunkt	Entwurfs- und Entwicklungsphase

Tabelle 4.16: NFR: Funktionale Eignung: Funktionale Korrektheit

Kategorie	Funktionale Eignung: Funktionale Korrektheit (Functional Suitability: Functional correctness)
Beschreibung	Das entwickelte System wird durch Testfälle überprüft. Dabei sollen 80% aller Methoden durch Unit Tests abgedeckt sein und ein korrektes Ergebnis liefern. Systemtests sollen alle möglichen Nutzer Interaktionen abdecken.
Zeitpunkt	Entwurfs- und Testphase

Tabelle 4.17: NFR: Leistungseffizienz: Zeitverhalten

Kategorie	Leistungseffizienz: Zeitverhalten (Performance efficiency: Time-behaviour)
Beschreibung	Das Klassifizieren von Entscheidungswissen in JIRA-Issue-Kommentaren dauert für 10 Kommentare mittlerer Länge (<100 Wörter) nicht länger als 5 Sekunden auf einem aktuellen System (Intel i5 7200U, 16 GB Ram)
Zeitpunkt	Entwurfs- und Entwicklungsphase

Tabelle 4.18: NFR: Kompatibilität: Interoperabilität

Kategorie	Kompatibilität: Interoperabilität (Compatibily: Interoperability)
Beschreibung	Die entwickelte Komponente fügt sich in ConDec ein. Dabei entstehen keine Integrationsprobleme.
Zeitpunkt	Entwurfs- und Entwicklungsphase

Tabelle 4.19: NFR: Benutzerfreundlichkeit: Schutz vor Benutzerfehlern

Kategorie	Benutzerfreundlichkeit: Ästhetik der Benutzeroberfläche (Usability: User interface aestetics)
Beschreibung	Farbliche Kennzeichnung verschiedener Wissenstypen.
Zeitpunkt	Entwicklungsphase

Tabelle 4.20: NFR: Zuverlässigkeit: Fehlertoleranz

Kategorie	Zuverlässigkeit: Fehlertoleranz (Reliability: Fault tolerance)
Beschreibung	Das System verhält sich stabil gegenüber fehlerhaften Nutzereingaben.
Zeitpunkt	Entwicklungs- und Testphase

Tabelle 4.21: NFR: Instandhaltbarkeit: Modularität

Kategorie	Instandhaltbarkeit: Modularität (Maintainability: Modularity)
Beschreibung	Das System verwendet geeignete Design Patterns, Seperation of Concerns sowie geeignete Klassen und Paketstrukturen.
Zeitpunkt	Entwurfs- und Entwicklungsphase

Tabelle 4.22: NFR: Instandhaltbarkeit: Wiederverwendbarkeit

Kategorie	Instandhaltbarkeit: Wiederverwendbarkeit (Maintainability: Reusability)
Beschreibung	Die Systemarchitektur erlaubt das spätere hinzufügen von anderen Quellen mit Entscheidungswissen
Zeitpunkt	Entwurfs- und Entwicklungsphase

4.4 User Interface Strukturdiagramm

Das User Interface (UI) Strukturdiagramm in Abbildung 4.2 beschreibt alle für DecXtract nötigen Arbeitsbereiche und deren Navigationsmöglichkeiten. Zudem enthält jeder Arbeitsbereich die zugehörigen Systemfunktionen und Daten.

Ansicht Work Space (WS) 1 wird von JIRA bereitgestellt und beschreibt die JIRA-Oberfläche. JIRA-Issues werden in Ansicht WS1.4 dargestellt. Die NutzerIn kann hier JIRA-Issues erzeugen und verwalten. Ansicht WS1.4.1 erweitert Ansicht WS1.4 durch ein Modul und zeigt verlinktes Entscheidungswissen zum aktuellen JIRA-Issue in einer Graphenansicht. Die von ConDec bereitgestellten Systemfunktionen werden in dieser Ansicht durch die DecXtract Systemfunktionen 4 und 8 erweitert. Ansicht WS1.4.2 erzeugt ein Tab Panel am unteren Ende von Ansicht WS1.4. Das Tab Panel befindet sich neben dem Panel „Kommentare“ und zeigt Entscheidungswissen aus Kommentaren zum aktuellen JIRA-Issue. In Verbindung mit dem Tab Panel „Kommentare“ stehen die Systemfunktionen 2,3,4,5 und 7.

Über die von JIRA bereitgestellte seitliche Navigation kann Ansicht WS1.3 und Ansicht WS1.5 erreicht werden. Ansicht WS1.3 ermöglicht es NutzerInnen Entscheidungswissen zum aktuellen Projekt einzusehen und zu verwalten. Ansicht WS1.5 erzeugt eine Auswertung über Entscheidungswissen aus JIRA-Issue-Kommentaren. Systemfunktion 6 berechnet hier entsprechende Kennzahlen für das ausgewählte Projekt. Ansicht WS1.2 und Ansicht WS1.1 ermöglichen es, ProjektadministratorInnen ConDec und einzelne Komponenten zu (de)aktivieren. Zusätzlich bietet Ansicht WS1.2 weitere feinere Einstellungsmöglichkeiten für die vorhandenen Komponenten zur Verfügung.

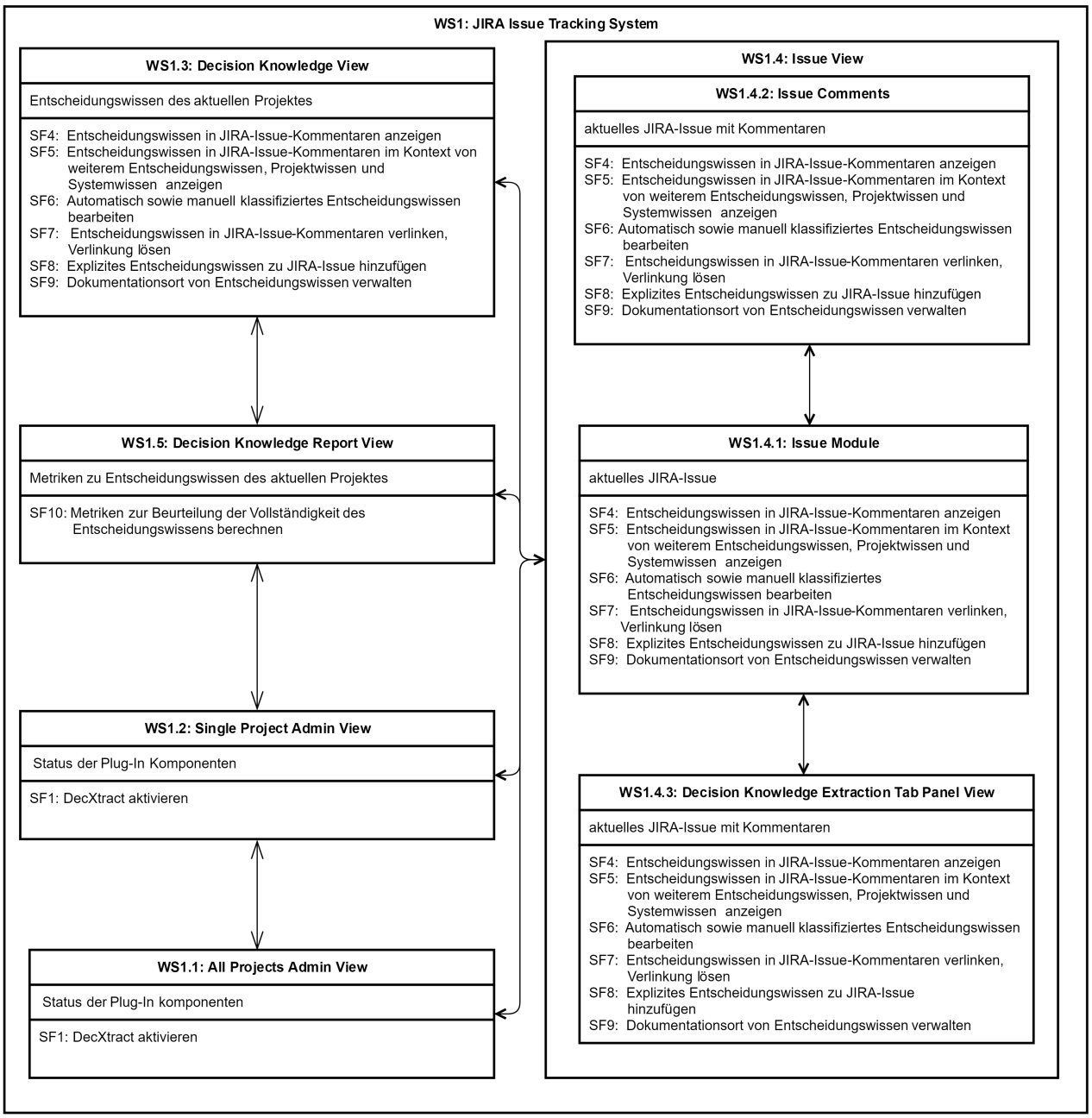


Abbildung 4.2: UI Strukturdiagramm zu ConDec und DecXtract

4.5 Zusammenfassung

Dieses Kapitel definiert die erwarteten NutzerInnen in den Rollen einer Projekt-AdministratorIn, einer Rationale-ManagerIn, einer EntwicklerInnen und einer Anforderungsbeauftragten. Ein User Task und fünf User Sub Tasks beschreiben die Aufgaben dieser Rollen. Zehn funktionale Anforderungen werden mit Systemfunktionen und User Stories an DecXtract gestellt. Neun weitere nichtfunktionale Anforderungen beschreiben wie gut die funktionalen Anforderungen umgesetzt werden müssen. Tabelle 4.23 beschreibt alle, in diesem Kapitel definierten, Anforderungen.

Tabelle 4.23: Auflistung aller Anforderungen mit Identifikationsnummer

Id	Typ	Name
1	SF	DecXtract aktivieren
2	SF	Entscheidungswissen in JIRA-Issue-Kommentaren automatisch klassifizieren
3	SF	Entscheidungswissen in JIRA-Issue-Kommentaren manuell klassifizieren
4	SF	Entscheidungswissen in JIRA-Issue-Kommentaren anzeigen
5	SF	Entscheidungswissen in JIRA-Issue-Kommentaren im Kontext von weiterem Entscheidungswissen, Projektwissen und Systemwissen anzeigen
6	SF	Automatisch sowie manuell klassifiziertes Entscheidungswissen bearbeiten
7	SF	Entscheidungswissen in JIRA-Issue-Kommentaren verlinken, Verlinkung lösen
8	SF	Explizites Entscheidungswissen zu JIRA-Issue hinzufügen
9	SF	Dokumentationsort von Entscheidungswissen verwalten
10	SF	Metriken zur Beurteilung der Vollständigkeit des Entscheidungswissens berechnen
11	NFR	Funktionale Eignung: Funktionale Vollständigkeit
12	NFR	Funktionale Eignung: Funktionale Korrektheit
13	NFR	Leistungseffizienz: Zeitverhalten
14	NFR	Kompatilität: Interoperabilität
15	NFR	Benutzerfreundlichkeit: Ästhetik der Benutzeroberfläche
16	NFR	Zuverlässigkeit: Fehlertoleranz
17	NFR	Instandhaltbarkeit: Modularität
18	NFR	Instandhaltbarkeit: Wiederverwendbarkeit

5 Entwurf und Implementierung

Dieses Kapitel beschreibt die wichtigsten Entwurfsentscheidungen zur Umsetzung der definierten Anforderungen sowie Vorgehen und Herausforderungen bei der Implementierung. Abschnitt 5.1 erläutert das existierende JIRA-ConDec Plug-In und dessen Architektur. Abschnitt 5.2 benennt alle Entscheidungen, die zur Erweiterung der ConDec Systemarchitektur getroffen wurden. Dafür werden zunächst Modelle und anschließend die Integration in ConDec beschrieben. Abschnitt 5.3 erklärt alle Entwurfsentscheidungen zu Systemfunktionen sowie deren Implementierung.

5.1 Architektur des ConDec JIRA-Plug-In's

Das CURES ConDec JIRA-Plug-In ermöglicht der NutzerIn, Entscheidungswissen in JIRA zu erfassen und zu erforschen. Dokumentiertes Entscheidungswissen kann in JIRA-Issues mit Features, Aufgaben zur Implementierung oder Fehlerberichten verlinkt werden. Dafür existieren bereits verschiedene Ansichten und Module (Kapitel 4.4 Seite 29), die DecXtract erweitert.

ConDec stellt derzeit zwei Möglichkeiten bereit, um Entscheidungswissen zu persistieren. Die Issue-Strategie stellt Entscheidungselemente in JIRA-Issues dar. JIRA-Issue-Links werden verwendet, um Entscheidungselemente miteinander und mit anderen JIRA-Issues, wie beispielsweise Feature-Aufgaben, zu verlinken. Der Vorteil dieser Strategie besteht darin, dass alle für JIRA-Issues verfügbaren Funktionen genutzt werden können, um Entscheidungswissen zu verwalten, z.B. die Suche nach einer Entscheidung in der Liste der JIRA-Issues. Der Nachteil ist, dass das dedizierte Issue-Type-Schema dem JIRA-Projekt zugeordnet werden muss.

Um diesen Nachteil zu überwinden, verwendet die Active-Objects-Strategie unterschiedliche Modellklassen für Entscheidungselemente und deren Verknüpfungen. Diese Strategie nutzt *Object Relational Mapping*, um mit der internen Datenbank von JIRA zu kommunizieren. Instanzen dieser Datenbank werden mit *Active Objects* (AO) bezeichnet. Die AO-Strategie nutzt zwei Datenbanktabellen. Die Tabelle *ConDec_Element* speichert Entscheidungselemente. Die Tabelle *ConDec_Link* speichert Links zwischen zwei Entscheidungselementen der AO-Strategie.

REST Schnittstellen stellen einen wichtigen Bestandteil für die JIRA-Plug-In Entwicklung dar. Diese ermöglichen es, außerhalb von JIRA, Daten aus der JIRA-Datenbank anzufordern oder neue Daten an den JIRA-Server zu übertragen.

Der Quellcode von ConDec ist auf öffentlich GitHub[1] verfügbar.

[1] ConDec GitHub: https://github.com/cures-hub/cures-condec-jira Zuletzt aufgerufen am 17.12.2018

5.2 Architekturentscheidungen

Die Architekturbeschreibungen enthalten alle Änderungen und Erweiterungen an der ConDec Systemarchitektur. Tabelle 5.1 beschreibt wichtige Entscheidungsprobleme und deren Lösung.

Die Firma Atlassian entwickelt JIRA als Webservice, NutzerInnen auf der Clientseite können diesen Dienst einfach in ihrem Webbrowser nutzen. Deshalb können alle gängigen Web-Programmiersprachen für die Clientseite eingesetzt werden. Diese Arbeit nutzt JavaScript (mit JQuery) und Apache Velocity. Die Serverseite beschränkt die Entwicklung auf Java als Programmiersprache.

Bei der Projektplanung musste entschieden werden, ob DecXtract direkt in das ConDec Plug-In integriert wird oder als eigenständiges, von ConDec abhängiges JIRA-Plug-In entwickelt werden soll. Da DecXtract von einigen ConDec Komponenten abhängig ist, wurde entschieden DecXtract direkt zu integrieren.

Mit DecXtract werden neue Modellklassen anhand des Domänendatendiagramms in Abbildung 4.1 auf Seite 27 eingefügt. Abbildung 5.1 zeigt das Entwurfsklassendiagramm der DecXtract Modellklassen (gelb) im Kontext der ConDec Modellklassen (blau). Die neue Klasse *Sentence* bildet einen Textabschnitt ab und speichert benötigte Attribute. Ein Satz repräsentiert ein Entscheidungselement als Bestandteil eines Entscheidungsproblems. Daher erbt *Sentence* von *DecisionKnowledgeElement*, um die Integration in ConDec zu ermöglichen. Sätze sind Teile eines JIRA-Issue-Kommentars, diese werden durch die Klasse *Comment* abgebildet.

Um DecXtract optimal in ConDec zu integrieren, muss die Klasse *Sentence* im restlichen Plug-In etabliert werden. Dies betrifft die erzeugten Oberflächenstrukturen von ConDec in Ansicht WS1.3: Decision Knowledge View. Da *Sentence* Objekte auch *DecisionKnowledgeElement* Objekte sind, benötigen Methoden keine Fallunterscheidung. Das Entwurfsklassendiagramm in Abbildung 5.2 zeigt die geplante Beziehung der Klassen zur Erzeugung der Listenansicht *TreeViewer* und der Graphenansicht *GraphImpl*. Die Modellklasse *Sentence* soll dabei so entwickelt werden, dass alle Zuständigkeiten getrennt sind (engl.: Seperation of Concerns).

Da *Sentence* von *DecisionKnowledgeElement* erbt, müssen keine Änderungen am Zugriff auf Informationen (bspw. das Attribut *description*) angepasst werden. Die Klasse *GraphImpl* muss um eine Methode ergänzt werden, die alle verlinkten Sätze und deren Links einbindet.

Tabelle 5.1: Architektur Entscheidungsprobleme

Entscheidungsproblem	Entscheidung
Welche Programmiersprachen sollen genutzt werden?	Java, JavaScript, Apache Velocity
Wie soll DecXtract mit ConDec Funktionalitäten verknüpft werden?	Integriere DecXtract in das ConDec Plug-In
Wie sollen die neuen Entitäten in ConDec eingebunden werden?	Nutze die Klasse *Sentence* für einen Satz in einem JIRA-Issue-Kommentar *Comment*
Wie soll DecXtract mit ConDec interagieren?	Nutze *DecisionKnowledgeElement* als Super-Klasse für *Sentence*

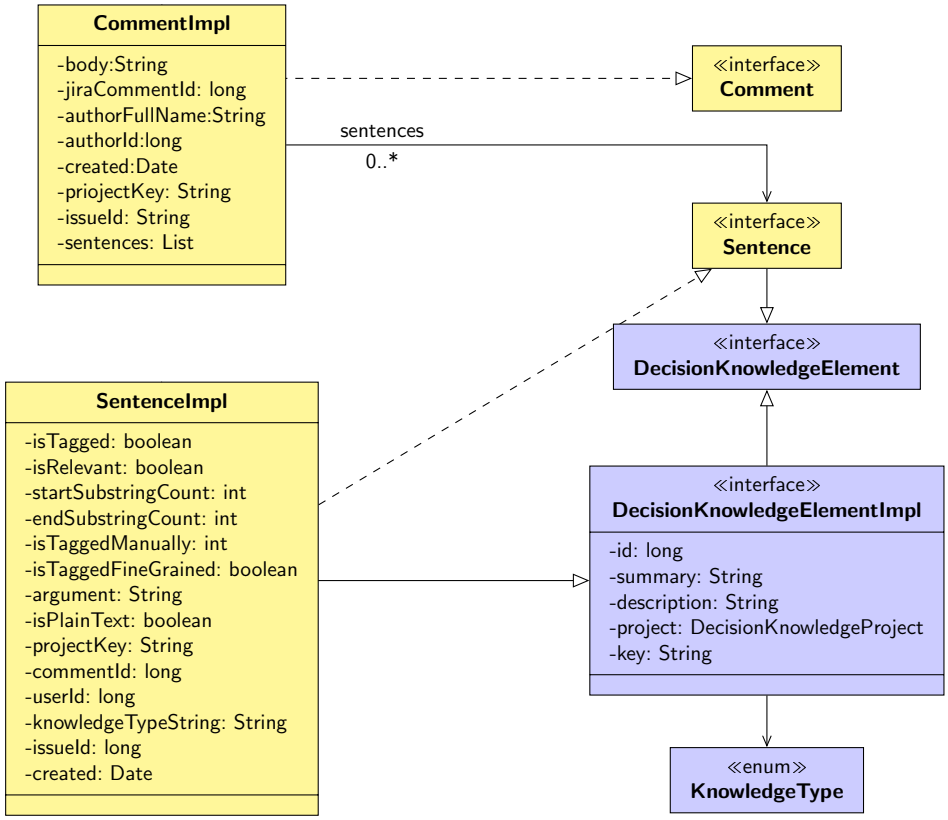


Abbildung 5.1: Entwurfsklassendiagramm zur Integration der Modellklassen in ConDec

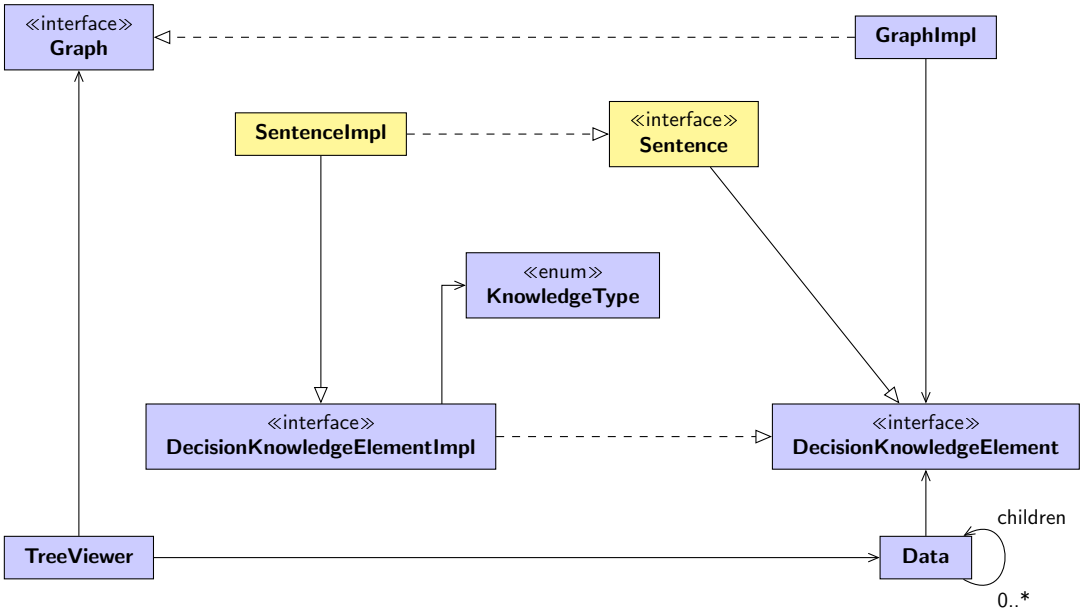


Abbildung 5.2: Entwurfsklassendiagramm zur Integration der Modellklassen von DecXtract in die Oberflächenklassen von ConDec

5.3 Entwurfsentscheidungen zu Systemfunktionen

Aus den Anforderungen ergeben sich drei Möglichkeiten Entscheidungswissen in JIRA-Issue-Kommentaren in eine Wissenstyp zu klassifizieren. Die manuelle Klassifizierung mit Tags und Icons in Systemfunktion 3 aus Abschnitt 4.2.2.3, die automatische Klassifizierung in Systemfunktion 2 aus Abschnitt 4.2.2.2, sowie die Korrektur der vorangegangenen Möglichkeiten in Systemfunktion 6 aus Abschnitt 4.2.2.6.

Diese und weitere Systemfunktionen werden nachfolgend anhand ihrer Entwurfsentscheidungen und Implementierungsarbeiten beschrieben.

5.3.1 DecXtract aktivieren

Dieser Abschnitt beschreibt Systemfunktion 1 aus Anforderungskapitel 4.2.2.1 auf Seite 24.

Die JIRA-Projekt AdministratorIn kann entscheiden, ob sie DecXtract im Rahmen von ConDec benutzen möchte. ConDec bietet bereits zwei Ansichten für Einstellungen an Projekten an: WS1.1 All Projects Admin View und WS1.2 Single Project Admin View. WS1.1 bietet eine Übersicht aller Projekte und ermöglicht die (De)aktivierung einzelner Komponenten von ConDec. Ansicht WS1.2 beschreibt das aktuell ausgewählte Projekt detaillierter und bietet feinere Einstellungsmöglichkeiten wie z.B. welche Typen von Entscheidungswissen verwendet werden sollen.

Es wurde entschieden, diese Ansichten um eine (De)aktivierungsmöglichkeit für DecXtract zu erweitern. Somit sind alle ConDec-Einstellungen für die NutzerIn an einer Stelle verfügbar.

Die Projekt-AdministratorIn soll in den Ansichten WS1.1 All Projects Admin View und WS1.2 Single Project Admin View die Möglichkeit haben DecXtract mit einem Schalter zu (De)aktivieren. Beide Ansichten bieten in ConDec bereits Möglichkeiten an um einzelne Komponenten zu konfigurieren. Abbildung 5.3 zeigt die Möglichkeit zur Aktivierung von DecXtract in WS1.1 All Projects Admin View in der letzten Spalte.

Abbildung 5.3: Aktivierungsmöglichkeit zu DecXtract in WS1.1 All Projects Admin View

JIRA erzeugt WS1.1 und WS1.2 durch Templates. Diese existieren bereits in ConDec und werden durch einen Schalter für die DecXtract Komponente erweitert. Eine REST-Schnittstelle in der Klasse *ConfigRest* ermöglicht die Übertragung einer Änderung auf Seite der NutzerIn an den Server. Serverseitig wird die Auswahl durch Klasse *ConfigPersistence* gespeichert.

5.3.2 Entscheidungswissen in JIRA-Issue-Kommentaren automatisch klassifizieren

Dieser Abschnitt beschreibt den Entwurf und die dabei getroffenen Entscheidungen zu Systemfunktion 2 aus Kapitel 4.2.2.2 auf Seite 24. Abbildung 5.4 illustriert das wichtigste Entscheidungsproblem zu dieser Systemfunktion. Tabelle 5.5 listet weitere Entscheidungsprobleme auf. Abbildung 5.6 zeigt alle nötigen Klassen für diese Systemfunktion.

R. Alkadhi entwickelt in ihrer Arbeitet einen Klassifikator, der Entscheidungswissen in natürlicher Sprache erkennt und in einen Wissenstyp klassifiziert. Dieser Algorithmus kann einem Satz mehr als ein Wissenstyp zuweisen [7]. Eine Entscheidung, die bei der Umsetzung dieser Systemfunktion getroffen werden musste, war die Frage, ob DecXtract mehr als einen Wissenstyp für ein Entscheidungselement zulässt. Abbildung 5.4 adressiert dieses Entscheidungsproblem.

Die ConDec-Klasse *DecisionKnowledgeElement* speichert Wissenstypen im Attribut *knowledgeType*. Die möglichen Wissenstypen sind in der *KnowledgeType* Enumeration gespeichert. Technisch ist die Umsetzung von mehreren Wissenstypen für ein Entscheidungselement möglich, erhöht allerdings den Integrationsaufwand in ConDec-Klassen. Zudem entsteht eine logische Inkonsistenz gegenüber der bestehenden Implementierung von ConDec. Um DecXtract gut in ConDec zu integrieren und eine gute Wartbarkeit zu ermöglichen wurde entschieden einem Satz maximal einen Wissenstyp zuzuordnen.

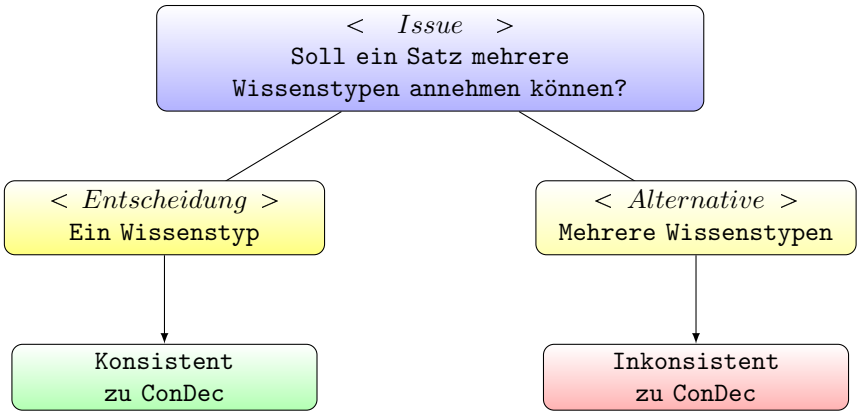


Abbildung 5.4: Entscheidung über die Zuordnung von Sätzen zu Wissenstypen.

Ein weiteres Entscheidungsproblem, das bei der Umsetzung dieser Systemfunktion gelöst werden muss ist, wie Fließtext in Sätze geteilt werden kann. Dabei muss auch entschieden werden, wie mit speziellen Formatierungen, wie in Tabelle 5.2 gezeigt, umgegangen wird. Es wurde entscheiden die JAVA Bibliothek BreakIterator[2] Zum Teilen von Sätzen einzusetzen. Diese Bibliothek Fließtexte in Sätze unter Berücksichtigung von Satztrennung, Grammatik sowie multipler Satzzeichen wie bspw. „?!“ und „...“. Somit hat diese Bibliothek einen großen Vorteil gegenüber Methoden wie bspw. *String.split()*.

Des Weiteren wurde entschieden, dass spezielle Formatierungen nicht geteilt werden, um ihren Kontext nicht zu zerstören. Da diese Textbausteine nicht in den Trainingsdaten vorkommen, wird formatierter Text auch von der Klassifikation ausgeschlossen.

[2] Java Break Iterator: https://docs.oracle.com/javase/7/docs/api/java/text/BreakIterator.html Zuletzt aufgerufen am 17.12.2018

Tabelle 5.2: Formatierungsmöglichkeiten, die vom Klassifikator nicht berücksichtigt werden.

Textbaustein	Beispiel
Zitat	{quote} This is a quote {quote}
Quellcode	{code:java} int i = 0; {code}
Logfile	{noformat} ...some logs...{noformat}
Panel	{panel:title=Title}This is a big text {panel}

Die Klasse *CommentSplitter* sucht in Fließtexten nach speziellen Formatierungen und teilt ein Kommentar in einzelne Sätze. Die Klasse *Sentence* persistiert alle Sätze, und speichert die Start- und Endpositionen des Satzes in einer AO-Tabelle. Die Alternative, den Text eines Satzes ebenfalls in einer AO-Tabelle zu speichern, stellt eine Redundanz gegenüber der JIRA-Persistierung von Kommentaren dar.

Nach der Satzteilung durch die Klasse *CommentSplitter*, beginnt die Textklassifikation. Abbildung 5.5 zeigt diesen Prozess. Der binärer Klassifikator prüft, ob ein Satz relevantes Wissen für ein Entscheidungsproblem besitzt. Die Klasse *DecisionKnowledgeClassifier* klassifiziert die Sätze in die Klassen *relevant* und *irrelevant*. Anschließend klassifiziert der feingranulare Klassifikator den Wissenstyp relevanter Sätze. Die verfügbaren Wissenstypen orientieren sich am IBIS Model (Abschnitt 2.2, Seite 6).

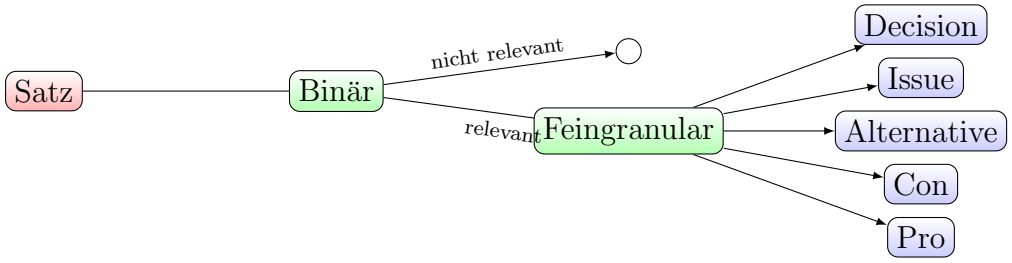


Abbildung 5.5: Prozessbaum im Klassifikationsprozess von Sätzen

Für die Implementierung des Klassifikators wird die Bibliothek Weka (Kapitel 2.4, Seite 2.4) genutzt. Um Texte zu klassifizieren, müssen diese initial gefiltert werden. Dabei stehen in WEKA mehrere Techniken zur Verfügung: Term Frequency & Inverse Term Frequency (TF-IDF) errechnet die Relevanz eines Terms[3] gegenüber einem Dokument. Gleichung 5.1 beschreibt die Berechnung der Term Frequency für einen Term i in einem Dokument j, wobei $n_{k,j}$ die Anzahl der Dokumente bezeichnet die den Term j beinhalten.

Anschließend wird die inverse term frequency in Gleichung 5.2 berechnet. Dabei bezeichnet i den jeweiligen Term und doc_i die Dokumente in denen der Term i vorkommt. Die Anwendung von TF-IDF resultiert in einer Tabelle von allen enthaltenen Wörtern, sowie aus dem Produkt von TF und IDF für jeden Term.

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \qquad (5.1) \qquad\qquad idf_i = log(\frac{n}{doc_i}) \qquad (5.2)$$

Die Anwendung der Methode *toLowerCase* vollzieht die Transformation aller Terme in Kleinbuchstaben. Die Methode *NGramTokenizing* bezeichnet die Anwendung von TF-IDF auf Wortsequenzen mit einer Länge von n.

[3]Term: Hier: Ein Wort, mehrere Wörter oder ein Satz

Der binäre Klassifikator nutzt die WEKA-Klasse *FilteredClassifier*. Diese kombiniert einen Filter mit einem beliebigen Klassifikationsalgorithmus. Die Filterklasse *StringToWordVector* nutzt die beschriebenen Filter und erzeugt Vektoren von Termen mit ihrer Relevanz im gewählten Dokument[4]. Experimente mit verschiedenen Klassifikationsalgorithmen zeigen, das *Sequential Minimal Optimization* (SMO) die besten Ergebnisse liefert. Tabelle 5.3 enthält die Ergebnisse des Trainingsprozess.

Tabelle 5.3: Trainingsmetriken verschiedener Klassifikationsalgorithmen zur binären Textklassifikation

Algorithmus	Precision	Recall	F1-Score
SMO	0,861	0,864	0,862
SGD	0,859	0,861	0,860

Der feingranulare Klassifikator nutzt die Klasse LC[5] als Klassifikator. Zusätzlich wird erneut die Klasse *FilteredClassifier* mit den beschriebenen Textfiltern eingesetzt. Tabelle 5.4 beschreibt gerundete Metriken für getestete Klassifikationsalgorithmen. Aufgrund der guten Ergebnisse und der schnellen Laufzeit wird der NaïveBayes Multinomial (NBM) Algorithmus eingesetzt [13].

Tabelle 5.4: Testmetriken feingranularer Klassifikationalgorithmen zur feingranularen Textklassifikation

Algorithmus	Alternative			Issue			Decision			Pro			Con		
Metrik	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
NBM	0,6	0,8	0,7	0,4	0,3	0,3	0,7	0,7	0,7	0,5	0,7	0,6	0,3	0,5	0,4
SMO	0,7	0,7	0,7	0,4	0,3	0,3	0,8	0,7	0,8	0,7	0,6	0,6	0,4	0,4	0,4
SGD	0,7	0,7	0,7	0,3	0,4	0,4	0,6	0,7	0,7	0,6	0,6	0,6	0,4	0,4	0,4
J48	0,7	0,7	0,7	0,4	0,1	0,2	0,8	0,6	0,7	0,6	0,5	0,5	0,5	0,3	0,3

Aufgrund der beschränkten Menge an Testdaten (Tabelle 2.6) wird die Trainingsmethode *cross validation* eingesetzt [15]. Diese Methode ist nützlich, wenn Test und Evaluationsdaten in geringer Menge vorliegen oder sehr verschieden sind. Zuerst wird die Anzahl der Validierungsdurchläufe mit n definiert. Anschließend wird der vorliegende Datensatz in n gleichgroße Teile geteilt. Teilmenge 1 bis $n-1$ stellen nun Trainingsdaten dar, Teilmenge n die Evaluationsdaten. Iterativ wird jede Teilmenge einmal vom Trainingsdatensatz ausgeschlossen und zur Evaluation des trainierten Modells verwendet. Anschließend wird das Modell mit den besten Ergebnissen ausgewählt.

Die Klassifikation wird ebenfalls in der AO-Tabelle gespeichert. Abschließend wird der klassifizierte Text im JIRA-Issue-Kommentar mit Tags (entsprechend Systemfunktion 3) markiert.

Zusätzlich wurde Ansicht WS1.2 Single Project Admin View um einen Schalter erweitert, der die automatische Klassifizierung ermöglicht oder unterdrückt. Die Systemfunktion prüft diese Einstellung vor jeder Ausführung des Klassifikators.

[4] Während des Trainingsprozess ist ein Dokument durch die Menge der Trainingsdaten einer Klasse beschrieben.
[5] LC als Abkürzung für Label Powerset.

Tabelle 5.5: Entscheidungsprobleme zu Systemfunktion 2

Entscheidungsproblem	Entscheidung
Darf ein Satz mehr als einen Wissenstyp annehmen?	Nein, da inkonsistent zu ConDec.
Wie kann Fließtext in einzelne Sätze geteilt werden?	Die BreakIterator Bibliothek trennt Fließtexte mit Rücksicht auf Grammatik und Satzzeichen.
Sollen speziell formatierte Testbausteine in einzelne Sätze geteilt werden?	Nein, da dies sonst den Kontext und die Aussage des Testbausteine zerstört.
Soll die NutzerIn entscheiden können, ob sie die automatische Klassifikation benutzen möchte?	Ja, mit einem Schalter in Ansicht WS1.2, da die permanente automatische Klassifizierung störend wirken kann.
Welche Trainingsmethode soll eingesetzt werden?	Training mit Cross Validation.
Welcher Klassifizierungsalgorithmus soll eingesetzt werden?	SGD (Binär) & NBM (Feingranular), aufgrund der (im Vergleich) besten Ergebnisse.

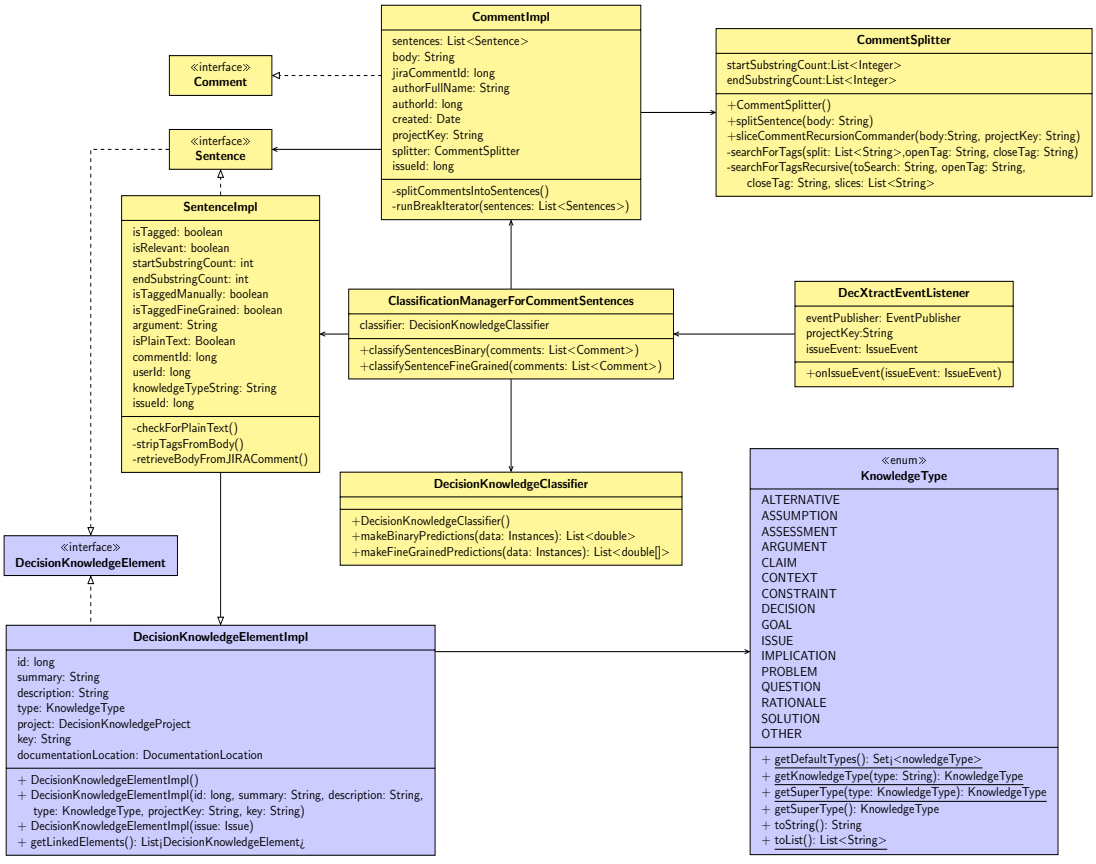


Abbildung 5.6: Klassendiagramm zu allen relevanten Klassen für Systemfunktion 2

5.3.3 Entscheidungswissen in JIRA-Issue-Kommentaren manuell klassifizieren

Dieser Abschnitt beschreibt den Entwurf und die dabei getroffenen Entscheidungen zu Systemfunktion 3 in Abschnitt 4.2.2.3 auf Seite 24. Abbildung 5.7 illustriert das wichtigste Entscheidungsproblem zu dieser Systemfunktion. Tabelle 5.7 listet weitere Entscheidungsprobleme auf. Abbildung 5.9 zeigt alle nötigen Klassen für diese Systemfunktion.

Die NutzerIn von DecXtract soll Sätze in JIRA-Issue-Kommentaren eigenständig als Entscheidungswissen klassifizieren können. Gegenüber der automatischen Klassifikation in Systemfunktion 2 wird dieser Vorgang als „manuelle Klassifikation“ bezeichnet.

Es musste entschieden werden, wie die NutzerIn Bereiche im Fließtext als Entscheidungswissen kennzeichnen kann. Abbildung 5.7 zeigt dieses Entscheidungsproblem. JIRA bietet die Verwendung von Makros im Text-Editor an. Makros erzeugen spezielle Formatierungen, wie in Tabelle 5.2 gelistet. Makros erlauben eine Kennzeichnung mit einem Icon und einer Hintergrundfarbe für Sätze eines Entscheidungselementes. Tabelle 5.6 zeigt ein Beispiel eines Makros.

Alternativ kann eine Markupsprache ähnlich wie HTML verwendet werden, die Sätze mit eindeutigem Start- und Endpunkt klassifiziert. Diese Variante erlaubt keine Formatierung im Kommentarbereich eines JIRA-Issues durch Farben. Die verwendeten Tags bleiben im Text sichtbar, was den Lesefluss der NutzerIn stört. Tabelle 5.6 zeigt ein Beispiel zur Verwendung einer Markupsprache.

Beide Alternativen wurden in einem Prototyp implementiert. Durch die farbliche Hervorhebung mit Makros werden aktuelle Entscheidungsprobleme in Ansicht WS1.4.1 Issue Comments (bspw. im vorangegangenen Kommentar) gut deutlich. Zudem können neue Projektbeteiligte schnell den Kontext aktueller Entscheidungselemente erkunden. Aufgrund dieser Argumente wurde entschieden, Makros zur manuellen Klassifikation zu implementieren. Des Weiteren soll es der NutzerIn möglich sein, Icons im Fließtext zu benutzen, um Entscheidungswissen zu markieren. Dafür sollen die in JIRA existierenden Icons mit einem Wissenstyp kodiert werden. Ein Icon markiert den Text bis zum nächsten Zeilenumbruch als Entscheidungselement.

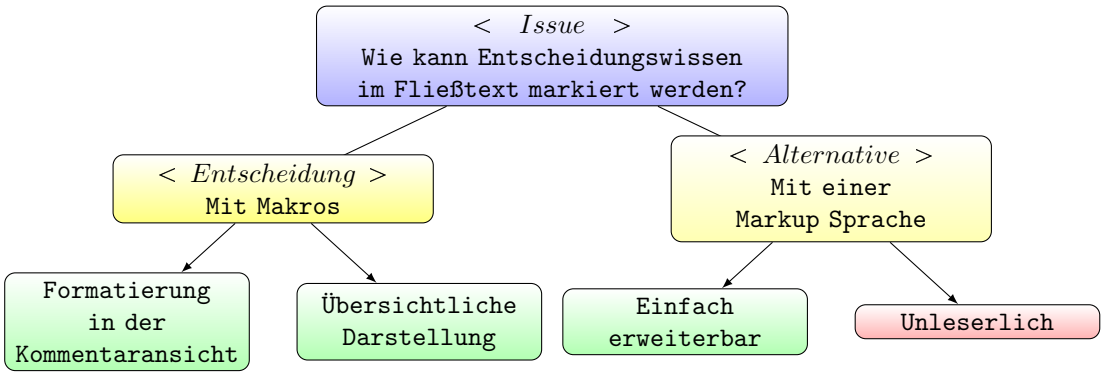


Abbildung 5.7: Entscheidungsbaum zur Entscheidung zur Markierung von Entscheidungswissen im Fließtext

Tabelle 5.6: Beispiele zur Markierung von Entscheidungswissen in Fließtexten

Makro	Markup
{pro}I like this idea {pro}	[pro] I like this idea [/pro]

Durch Vererbung und Überschreiben der Klasse *BaseMacro*[6] können neue Makros in JIRA hinzugefügt werden. Dabei benötigt jedes Makro (bzw. jeder Wissenstyp), eine eigene Klasse. Das Entwurfsklassendiagramm in Abbildung 5.8 beschreibt diese Beziehung. In jeder Klasse kann somit eine eigene HTML-Formatierung definiert werden. Die nachfolgenden fehlerhaften Eingabe werden nicht behandelt.

{pro} I like this idea {con} {pro} I like this idea {Pro}

Das Klassendiagramm in Abbildung 5.9 beschreibt u.a. die Klasse *DecXtractEventListener*. Diese Klasse registriert Änderung an einem JIRA-Issue, bspw. wenn die NutzerIn einen neuen Kommentar verfasst. Bei diesem Event startet die Klasse *ViewConnector*, die alle Kommentare des aktuellen JIRA-Issues einliest und analog zu Systemfunktion 2 in Sätze teilt und von der Klasse *ActiveObjectsManager* in die AO-Datenbank schreibt.

Tabelle 5.7: Entscheidungsprobleme zu Systemfunktion 3

Entscheidungsproblem	Entscheidung
Wie kann Entscheidungswissen im Fließtext markiert werden?	Makros, erlauben die Markierung in der Kommentaransicht.
Wann sollen Kommentare/Sätze in die AO-Datenbank geschrieben werden?	Event Listener fängt Speichern neuer Kommentare ab.
Soll die NutzerIn die Möglichkeit haben Markierungen durch Makros zu (De)aktivieren?	Anbindung an DecXtract Status in Systemfunktion 1.

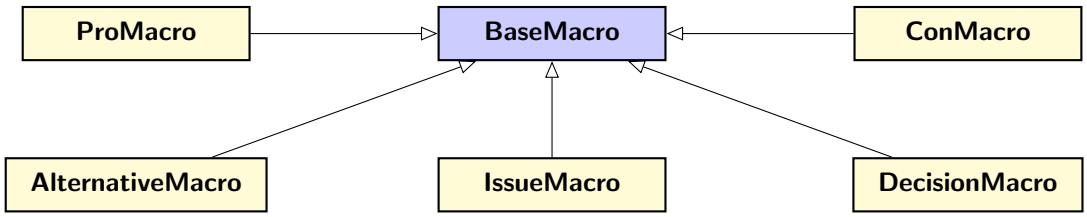


Abbildung 5.8: Entwurfsklassendiagramm für Makro Klassen

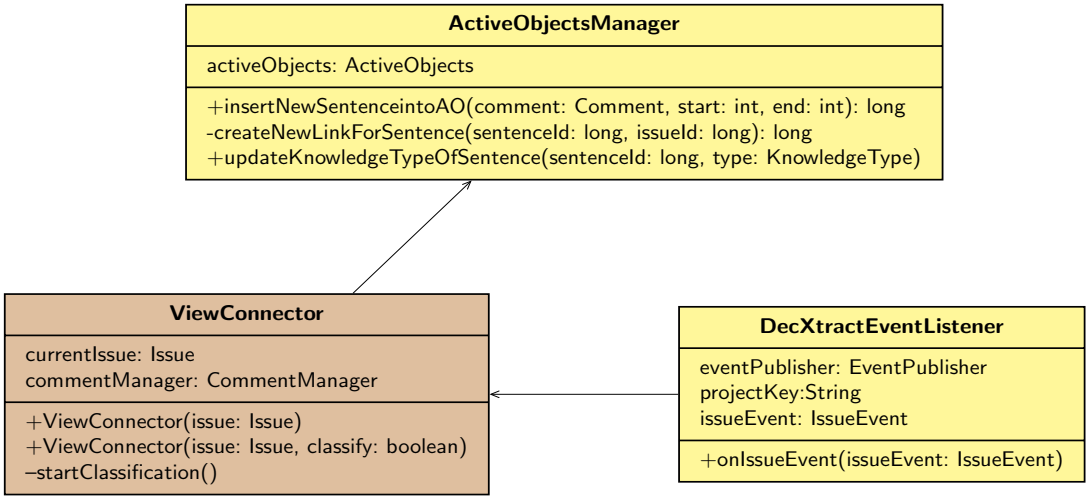


Abbildung 5.9: Klassendiagramm zu allen relevanten Klassen für Systemfunktion 2

[6]Base Macro: https://docs.atlassian.com/atlassian-renderer/5.0/apidocs/com/atlassian/renderer/v2/macro/BaseMacro.html Zuletzt aufgerufen am 17.12.2018

5.3.4 Entscheidungswissen in JIRA-Issue-Kommentaren anzeigen

Dieser Abschnitt beschreibt den Entwurf und die dabei getroffenen Entscheidungen zu Systemfunktion 4 in Abschnitt 4.2.2.4 auf Seite 25. Abbildung 5.10 illustriert das wichtigste Entscheidungsproblem zu dieser Systemfunktion. Tabelle 5.8 listet weitere Entscheidungsprobleme auf. Abbildung 5.11 zeigt alle nötigen Klassen für diese Systemfunktion.

Systemfunktion 2 und Systemfunktion 3 visualisieren Entscheidungswissen in JIRA-Issue-Kommentaren durch Makros im Kommentarbereich. Diese Systemfunktion soll diese Ansicht durch weitere Funktionalitäten erweitern.

Die Dokumentationsbeauftragte für Entscheidungswissen muss für ihre Arbeit sämtliches Entscheidungswissen eines JIRA-Issues untersuchen. Es wurde entschieden, dafür auch nicht relevant gekennzeichnete Sätze in Betracht zu ziehen. Abbildung 5.10 beschreibt das Entscheidungsproblem, wie sämtliches Entscheidungswissen zu einem JIRA-Issue angezeigt und verwaltet werden kann.

Eine Möglichkeit ist die Kommentaransicht um eine Filterfunktionalität zu erweitern. Diese Filter können ausgewählte Wissenstypen ein- und ausblenden. Ein Issue-Tabl-Panel kann Kommentare ähnlich zur Kommentaransicht abbilden und dabei die Filtermöglichkeit anbieten.

Eine Alternative ist eine Listenansicht aller Entscheidungselemente, die mit dem aktuellen JIRA-Issue verlinkt sind. Diese Darstellung erlaubt ebenfalls die Erweiterung einer Filtermöglichkeit.

Beide Alternativen wurden in einem Prototyp implementiert. Es wurde deutlich, dass die Listenansicht mehr Interaktionsmöglichkeiten für die NutzerIn bietet. Beispielsweise kann Systemfunktion 6 und Systemfunktion 7 hier eingesetzt werden. Zudem können durch verschachtelte Listen auch verlinktes Entscheidungswissen gezeigt werden. Die Darstellung in der erweiterten Kommentaransicht erzeugt sehr aufwändig HTML Code, wobei viele Sonderfälle beachtet werden müssen. Daher wurde entscheiden, die Listenansicht zu implementieren. Diese Ansicht wird mit WS1.4.3 Issue Tab Panel View bezeichnet.

Die Listenansicht in WS1.4.3 zeigt dabei identisches Wissen wie Ansicht WS:1.4.1 Issue Module. Jedoch wird die Listenansicht um Entscheidungselemente ergänzt, die als *nicht relevant* klassifiziert wurden. Ansicht WS1.4.1 zeigt nur relevantes Wissen, das mit dem aktuellen JIRA-Issue verknüpft ist.

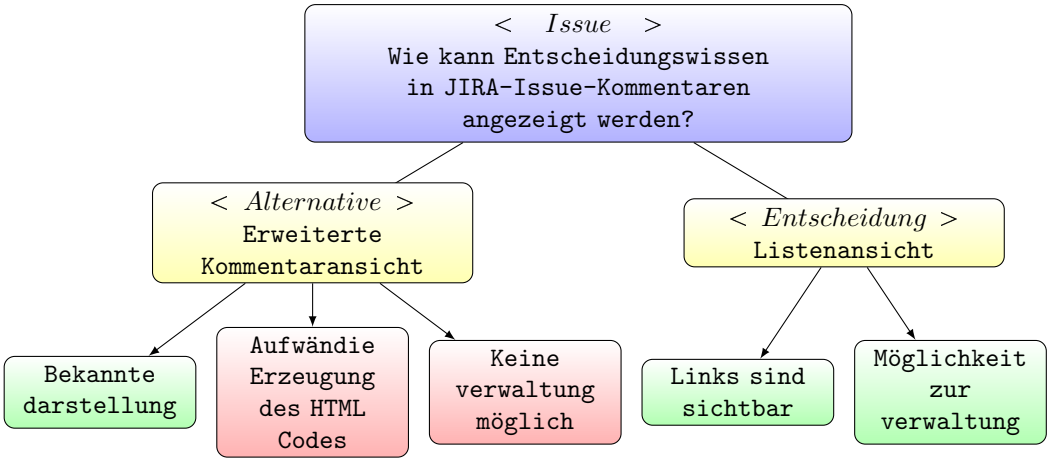


Abbildung 5.10: Entscheidungsbaum zur Entscheidung von Links zwischen verschiedenen Elementen des Entscheidungswissens.

Ansicht WS1.3: Decision Knowledge View stellt eine Listenansicht aller Entscheidungselemente zu einem gewählten Wissenstyps dar. Der Konstruktor der Klasse *TreeViewer* erzeugt diese Liste anhand eines übergebenen Wissenstyps. Ein neuer Konstruktor für DecXtract ermöglicht den Aufbau einer Liste anhand eines Entscheidungselementes. Dafür werden rekursiv alle verlinkten Entscheidungselemente in die Liste eingefügt. Es wurde entschieden, die Liste auf Nutzerseite mit der JavaScript Bibliothek *JsTree* zu präsentieren. Diese erzeugt eine verschachtelte Liste und ermöglicht weitere Funktionalitäten wie Kontextmenüs und Drag and Drop.

Tabelle 5.8: Entscheidungsprobleme zu Systemfunktion 4

Entscheidungsproblem	Entscheidung
Wie soll Entscheidungswissen in JIRA-Issue-Kommentaren angezeigt werden?	Listenansicht
Sollen nicht relevante Sätze in der Listenansicht gezeigt werden?	Ja, bietet einfachere Möglichkeit zur Verwaltung gegen Bearbeitung der Makros.
Wie soll die Filtermöglichkeit nach Wissenstypen realisiert werden?	Liste wird nach gefilterten Typen durchsucht und entsprechend entfernt.
Wie soll die Listenansicht auf Nutzer Seite realisiert werden?	JSTree Bibliothek bietet geforderte Funktionalität, bereits in ConDec realisiert.
Wie soll die Listenansicht erzeugt werden?	Erweiterung der ConDec Implementierung.
Wie sollen Wissentstypen in der Listenansicht kodiert werden?	Typ-Icon vor jedem Eintrag.

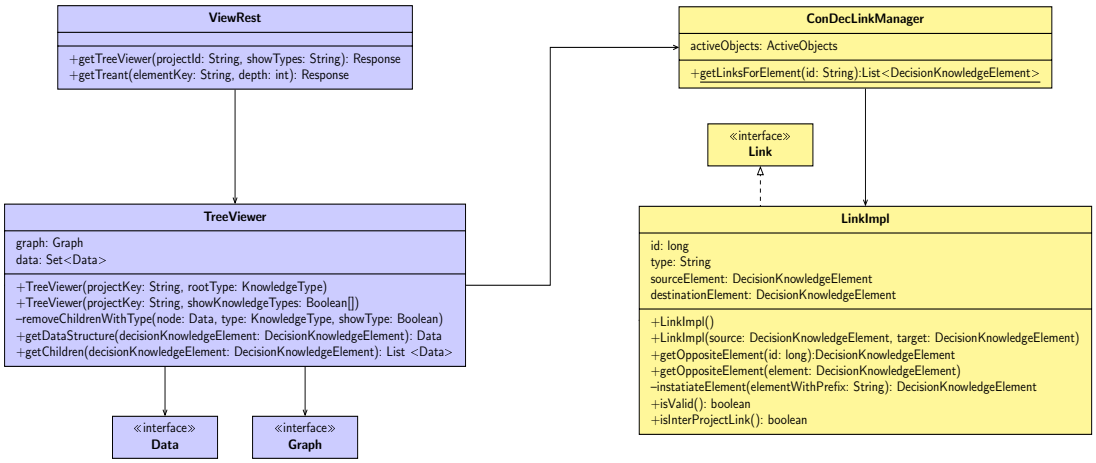


Abbildung 5.11: Klassendiagramm zu allen relevanten Klassen für Systemfunktion 4

5.3.5 Entscheidungswissen in JIRA-Issue-Kommentaren im Kontext zu weiterem Entscheidungswissen, Projektwissen und Systemwissen anzeigen

Dieser Abschnitt beschreibt den Entwurf und die dabei getroffenen Entscheidungen zu Systemfunktion 5 in Abschnitt 4.2.2.5 auf Seite 25. Abbildung 5.12 illustriert das wichtigste Entscheidungsproblem zu dieser Systemfunktion. Tabelle 5.11 listet weitere Entscheidungsprobleme auf. Abbildung 5.13 zeigt alle nötigen Klassen für diese Systemfunktion.

Die NutzerIn hat in Ansicht WS1.3: Decision Knowledge View und WS1.4.1 Issue Module die Möglichkeit Entscheidungswissen im Kontext von Projekt und Systemwissen zu betrachten und zu verwalten. Bei der Umsetzung dieser Systemfunktion musste entschieden werden, wie Entscheidungswissen in JIRA-Issue-Kommentaren zusammen mit Entscheidungswissen in JIRA-Issues in diese Ansichten eingebunden wird. DecXtract ist durch die Vererbung von *DecisionKnowledgeElement* bereits gut in ConDec integriert. Allerdings sind Verlinkungen zwischen Entscheidungselementen in ConDec durch JIRA-Links repräsentiert, die nicht geeignet sind, um Instanzen der Klasse *Sentence* mit einem JIRA-Issue zu verlinken. Abbildung 5.12 beschreibt das Entscheidungsproblem, wie Instanzen der Klasse *Sentence* mit anderen Wissenstypen verlinkt werden können. Dabei benutzen alle Alternativen eine AO-Tabelle, um Links zu speichern.

Die erste Möglichkeit nutzt eine AO-Tabelle für jede Link-Kombination verschiedener Dokumentationsorte. Bsp: Satz → Issue, Satz → Satz. Diese Möglichkeit ist schnell implementiert, da sie keine Änderungen, sondern nur Erweiterungen in ConDec erfordert. Allerdings entstehen viele Tabellen bei zukünftiger Integration neuer Dokumentationsorte.

Eine Alternative ist die Verwendung einer Tabelle für ausgehende Links jedes Dokumentationsorts. Dabei ist immer der Dokumentationsort der Quelle bekannt, der Dokumentationsort des Zielelements ist in der Tabelle gespeichert. Diese Variante erfordert einige Änderungen am existierenden System. Auch diese Variante erfordert Änderungen bei der Integration neuer Dokumentationsorte.

Die dritte Variante beschreibt eine Tabelle für alle Links. Hier werden die Dokumentationsorte für Start- und Zielelement jedes Links gespeichert. Diese Variante nutzt nur eine Tabelle und ist zukünftig einfach um neue Dokumentationsorte erweiterbar. Es muss durch eine Fallunterscheidung sicher gestellt werden, dass jeder Link auf den korrekten Dokumentationsort verweist.

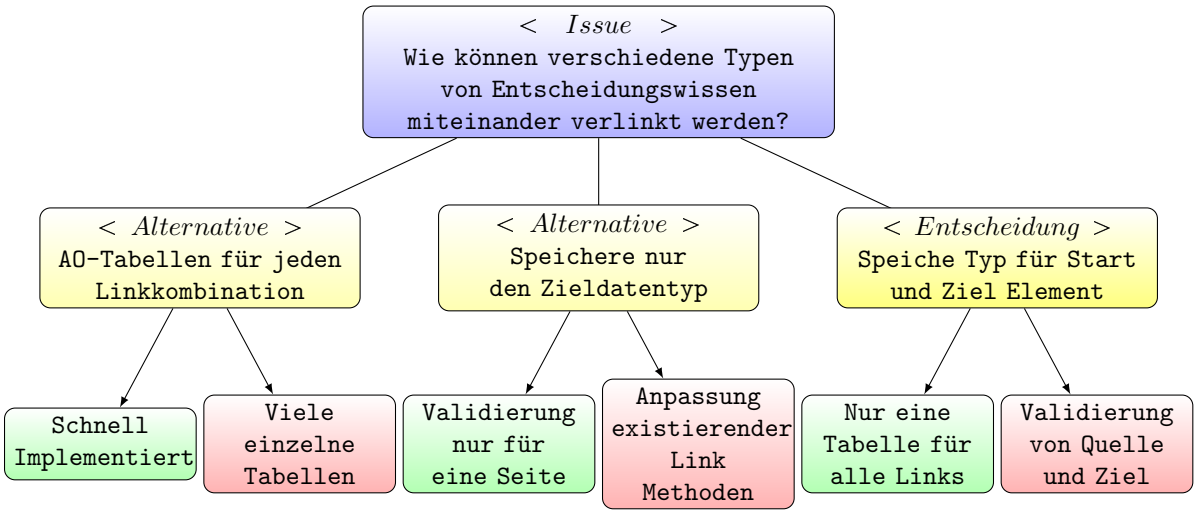


Abbildung 5.12: Entscheidungsbaum zur Entscheidung von Links zwischen verschiedenen Elementen des Entscheidungswissens.

Aufgrund der guten Integrierbarkeit neuer Dokumentationsorte wurde entschieden, die dritte Alternative zu implementieren. Diese Variante speichert Links zwischen verschiedenen Dokumentationsorten in einer AO-Tabelle. Dafür wurde entschieden, den Dokumentationsort mit einem eindeutigen Präfix vor der Id eines Entscheidungselementes zu identifiziert. Des Weiteren soll dafür die Enumeration *DocumentationLocation* eingeführt werden. Tabelle 5.9 listet alle eingesetzten Präfixe mit Dokumentationsort auf.

Die ConDec Active-Objects Strategie speichert ebenfalls Links in AO-Tabellen. Es wurde entscheiden, beide Möglichkeiten zur Verlinkung in einer AO-Tabelle zu vereinen. Dies ermöglicht eine konsistente und leicht erweiterbare Methode zur Verlinkung zwischen Entscheidungswissen aller Dokumentationsorte. Tabelle 5.10 zeigt einen Link zwischen einem JIRA-Issue und einem Satz eines JIRA-Issues. Die *id* weißt dem Link-Objekt eine eindeutige Identifikationsnummer zu, *type* beschreibt die Beziehung zwischen den verlinkten Objekten. Dabei kann *type* die Werte *contain*, *supports* und *attacks* annehmen.

Tabelle 5.9: Kodierung der Abkürzungen zur Speicherung von Links verschiedener Dokumentationsorte

Präfix	Dokumentationsort
i	JIRA-Issue
s	Satz eines JIRA-Issue-Kommentars
a	Active Objects Objekt
c	Commit
p	Pull Request

Die Klasse *ConDecLinkManager* verwaltet die Zugriffe auf die AO-Tabellen mit gespeicherten Links. Diese Klasse bietet die üblichen Datenbankoperationen, einfügen, löschen und aktualisieren. Zudem kann für eine Instanz von *DecisionKnowledgeElement* nach möglichen (ein- und ausgehenden) Links gesucht werden.

Tabelle 5.10: Linkbeispiel zwischen Issue mit Id „345“ und Satz mit Id „678“

id	source	target	type
123	i345	s678	contain

Tabelle 5.11: Entscheidungsprobleme zu Systemfunktion 5

Entscheidungsproblem	Entscheidung
Wie können verschiedene Typen von Wissen miteinander verlinkt werden	AO-Tabelle für Links speichert Dokumentationsorte und Ids von Elementen
Wie sollen Dokumentationsorte in der AO-Tabelle identifiziert werden?	Präfix für die Id des Entscheidungselements
Wie soll zwischen ein- und ausgehenden Links unterschieden werden?	Speichern von Start- und Zielelement.
Wie soll der Dokumentationsort (Linkpräfix) gespeichert werden?	Enumeration *DocumentLocation* in *DecisionKnowledgeElement*
Sollen Links der AO-Strategie in einer eigenen Tabelle gespeichert sein?	Links der AO-Strategie sind in der gleichen Tabelle wie Links zwischen Sätzen gespeichert.

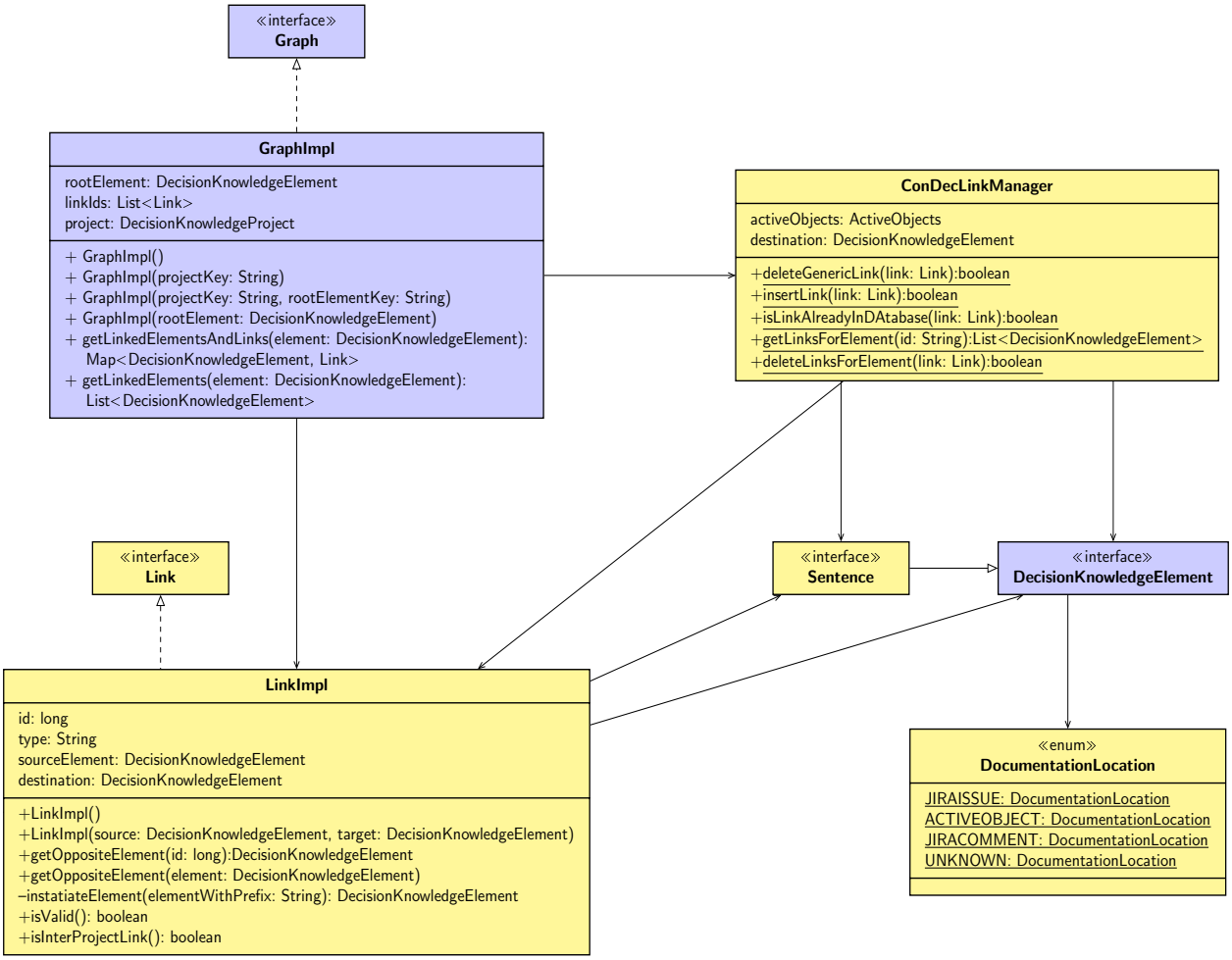


Abbildung 5.13: Klassendiagramm zu allen relevanten Klassen für Systemfunktion 5

5.3.6 Klassifiziertes Entscheidungswissen in JIRA-Issue-Kommentaren bearbeiten

Dieser Abschnitt beschreibt den Entwurf und die dabei getroffenen Entscheidungen zu Systemfunktion 6 aus Kapitel 4.2.2.6 auf Seite 25. Abbildung 5.14 illustriert das wichtigste Entscheidungsproblem zu dieser Systemfunktion. Tabelle 5.12 listet weitere Entscheidungsprobleme auf. Abbildung 5.15 zeigt alle nötigen Klassen für diese Systemfunktion.

Aus den Ergebnissen des Trainings- und Evaluationsprozesses des Klassifikators (Tabelle 5.4, Seite 39) ist ersichtlich, dass der Klassifikator auch falsche Klassifikationen vornimmt. Die NutzerIn soll eine Möglichkeit bekommen, Klassifikationen nachträglich anzupassen ohne Makros in JIRA-Issue-Kommentaren ändern zu müssen.

Es wurde entschieden, die Möglichkeit zur Bearbeitung in die Listenansicht aus Ansicht WS1.4.2 Issue Comments und WS1.3: DecisionKnowledgeView, sowie in die Graphenansicht in Ansicht WS1.3 und WS1.4.1: Issue Module einzufügen. Des Weiteren musste entschieden werden, welche Möglichkeit zur Bearbeitung von Entscheidungswissen aus JIRA-Issue-Kommentaren die NutzerIn bekommen soll. Abbildung 5.14 beschreibt dieses Entscheidungsproblem.

Eine Möglichkeit ist die Einbindung eines Dialogs mit einer Drop Down Liste aller Wissenstypen, der bei Auswahl eines Listenelements aufgerufen wird. Die NutzerIn muss für die Bearbeitung die Schritte: Dialog öffnen, Drop Down Liste öffnen, Wissenstyp wählen, Dialog schließen ausführen.

Eine Alternative ist die Einbindung von Drag and Drop Bereichen über der Listenansicht. So kann die NutzerIn Entscheidungselemente auf neue Wissenstypen schieben. Mit fünf Wissenstypen und einer Sektion für *nicht relevant* überlädt dies allerdings die Ansicht oder fordert sehr kleine Drop-Bereiche.

Eine weitere Möglichkeit ist die Nutzung eines Kontextmenüs für Elemente der Liste. Für jeden Wissenstyp existiert ein Eintrag im Kontextmenü bspw.: *setze Alternative*. Durch die Selektion des gewünschten Eintrags im Kontextmenü wird das Entscheidungselement neu klassifiziert. Diese Variante lässt sich zudem leicht um weitere Funktionalitäten erweitern.

Aufgrund der Erweiterbarkeit und der leichten Zugänglichkeit für die NutzerIn wurde entschieden, ein Kontextmenü zu implementieren. Zusätzlich stellt das Kontextmenü den Eintrag *Bearbeite Satz* bereit, der es der NutzerIn ermöglicht den Inhalt des selektierten Entscheidungselementes zu bearbeiten.

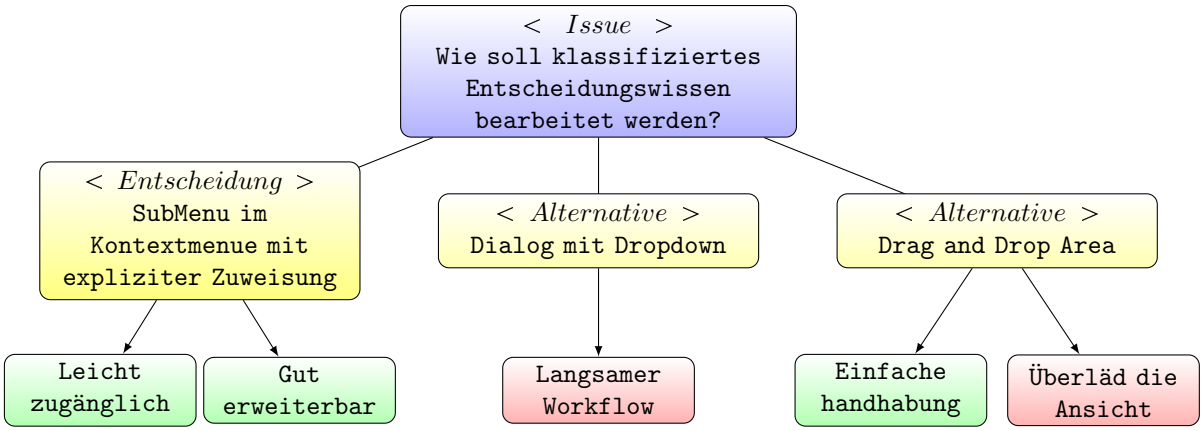


Abbildung 5.14: Entscheidungsbaum zur Entscheidung von Links zwischen verschiedenen Elementen des Entscheidungswissens.

Entscheidet sich die NutzerIn den Wissenstyp eines Satzes über das Kontextmenü zu bearbeiten, ändert die Klasse *KnowledgeRest* den AO-Eintrag des entsprechenden Elements. Zudem wurde entschieden, das Makro im Kommentar anzupassen, wenn die NutzerIn den Wissenstyp ändert. Dies erhält die Konsistenz zwischen den Dokumentationsorten. Des Weiteren stehen verschiedene Möglichkeiten zur Verfügung das Kontextmenü zu implementieren. Für ein Kontextmenü mit *jQuery* muss in der Implementierung zwischen Graphen und Listenansicht unterschieden werden. Eine Implementierung mit der Klasse *Dropdown* und dem CSS von JIRA ermöglicht eine identische Implementierung für alle Ansichten. Da diese Alternative weniger Quellcode produziert und durch das CSS von JIRA dargestellt werden kann, wurde diese Variante implementiert.

Tabelle 5.12: Entscheidungsprobleme zu Systemfunktion 6

Entscheidungsproblem	Entscheidung
Wie kann klassifiziertes Entscheidungswissen bearbeitet werden?	Kontextmenü für Entscheidungselemente
In welcher Ansicht soll die Bearbeitung ermöglicht werden?	Listen- und Graphenansicht mit Kontextmenü
Wie sollen geänderte Wissenstypen persistiert werden?	Wissenstyp in Makro im Kommentar und in AO ändern zur Erhaltung der Konsistenz
Welches Kontextmenü soll benutzt werden?	JIRA AUI Kontextmenü durch Dropdown Implementierung
Sollen Wissenstypen über das Kontextmenü geändert werden können?	Expliziter Eintrag in Kontextmenü

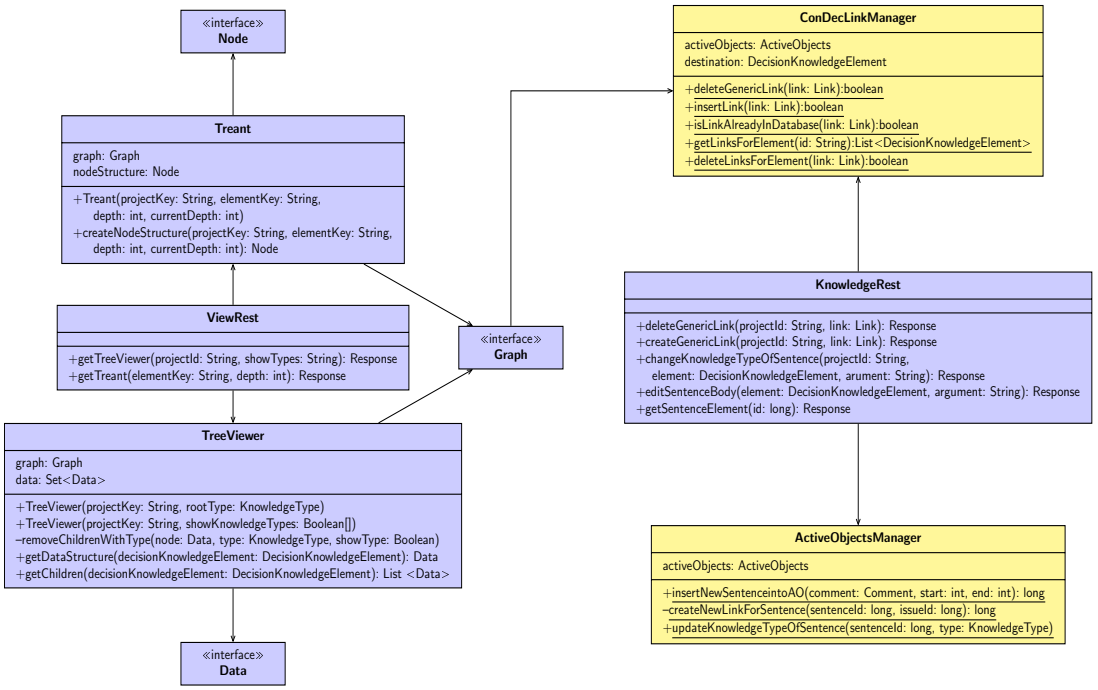


Abbildung 5.15: Klassendiagramm zu allen relevanten Klassen für Systemfunktion 6

5.3.7 Entscheidungswissen in JIRA-Issue-Kommentaren verlinken, Verlinkung lösen

Dieser Abschnitt beschreibt den Entwurf und die dabei getroffenen Entscheidungen zu Systemfunktion 7 aus Kapitel 4.2.2.7 auf Seite 26. Abbildung 5.16 illustriert das wichtigste Entscheidungsproblem zu dieser Systemfunktion. Tabelle 5.15 listet weitere Entscheidungsprobleme auf. Abbildung 5.17 zeigt alle nötigen Klassen für diese Systemfunktion.

In Entwurfsabschnitt 5.3.5 auf Seite 45 zu Systemfunktion 5 wird die Methode zum Speichern von Links zwischen Entscheidungselementen unterschiedlicher Dokumentationsort beschrieben. Diese Systemfunktion befasst sich mit der Erzeugung von Links durch die NutzerIn.

Bei der Entwicklung dieser Systemfunktion wurde entschieden, die Verlinkung von Entscheidungswissen in JIRA-Issue-Kommentaren in Ansicht WS1.3: Decision Knowledge Page, WS1.4.1: Issue Module und WS1.4.3: Issue Tab Panel View zu ermöglichen.

Abbildung 5.16 beschreibt das Entscheidungsproblem zur Frage, wie die NutzerIn Links in den genannten Ansichten herstellen kann. In Ansicht WS1.3 werden Links zwischen Entscheidungselementen durch Drag and Drop hergestellt. Ein Dialog über das Kontextmenü ermöglicht es zudem weitere JIRA-Issues zu verlinken.

Bei der Verwendung eines Dialogs muss die NutzerIn die folgenden Schritte durchführen: Entscheidungselement selektieren, Dialog öffnen, Link-Zielelement suchen, auswählen und bestätigen. Diese Aktionen verlangsamen den Arbeitsablauf der NutzerIn. Alternativ kann die Drag and Drop Funktionalität der Listen- und Graphenansicht genutzt werden. Dafür selektiert die Nutzerin ein Entscheidungselement und zieht es auf ein anderes Entscheidungselement. Somit ist ein Link zwischen zwei Elementen mit einer Aktion hergestellt.

Da Drag and Drop einen schnelleren Arbeitsablauf der Nutzerin ermöglicht sowie in allen Ansichten konsistent zu ConDec ist, wurde entschieden diese Alternative zu implementieren.

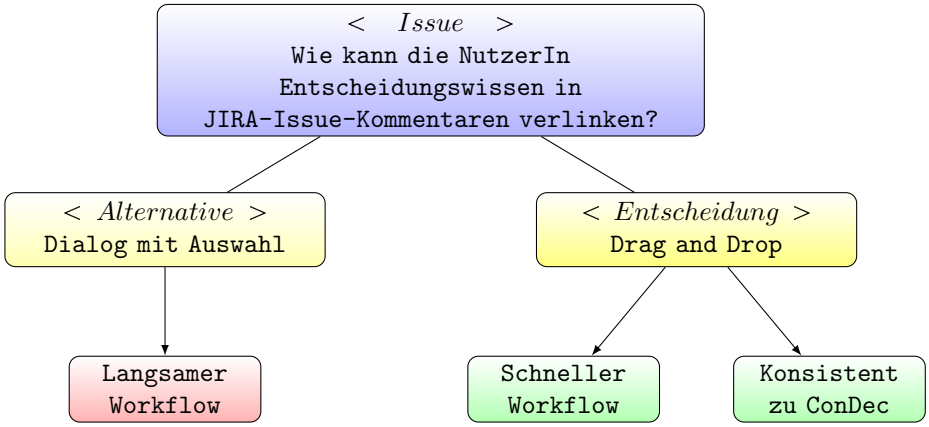


Abbildung 5.16: Entscheidungsbaum zur Entscheidung zur Erzeugung von Links zwischen Entscheidungselementen

Bei der Implementierung stellte sich die Frage, ob der NutzerIn eine Erfolgs- oder Fehlermeldung gezeigt werden soll, nachdem sie Entscheidungselemente verlinkt hat. ConDec nutzt die JIRA-Methode *showFlag(.)*, um der NutzerIn Rückmeldungen zu präsentieren. Um die Konsistenz zu ConDec zu erhalten, wird diese Methode genutzt um Rückmeldungen zu erzeugten Links zu präsentieren.

ConDec nutzt eine REST-Schnittstelle um Links zwischen Entscheidungselementen serverseitig herzustellen. Der Prototyp zu dieser Systemfunktion nutzt die Klasse *ConDecLinkManager* und eine eigene REST-Schnittstelle. Somit würden zwei REST-Schnittstellen zum Erzeugen von Links existieren. Daher wurde entschieden, beide Schnittstellen zu vereinen. Diese Schnittstellen-Methode prüft serverseitig, ob ein JIRA-Issue-Link erzeugt werden soll oder ein Link in der AO-Tabelle. Diese Entscheidung ermöglichte zudem eine deutliche Reduzierung des Quellcodes der Graphen- und Listenansicht.

Mit beiden vorangegangen Entscheidungen wurde entschieden, die Graphen- und Listenansicht nach jeder Verlinkung durch die NutzerIn neu vom Server zu laden. Dies ist ggf. mit Ladezeiten verbunden, allerdings sieht die NutzerIn stets das aktuell gespeicherte Entscheidungsproblem und es entstehen keine Inkonsistenzen zwischen Client- und Serverseite.

Nach Abschluss der Implementierung dieser Systemfunktion fiel auf, dass alle Entscheidungselemente initial zu ihrem JIRA-Issue verlinkt werden. Daher wurde die Möglichkeit geprüft, neue Entscheidungselemente direkt zu einem verwandten Entscheidungselement zu verlinken. Ein Experiment prüfte die Verlinkung anhand der textuellen Ähnlichkeit eines neuen Entscheidungselements zu den existierenden Entscheidungselementen. Die Levenshtein-Distanz ermöglicht es die Ähnlichkeit zweier Terme numerisch auszudrücken [20].

Das nachfolgende Beispiel fügt einen neuen Link für das neue Pro-Argument: „I agree, Uwe - I'll fold that into the patch.“ in des Entscheidungsproblems aus Abbildung 7.2 auf Seite 82 hinzu. Der Wortlaut dieses Satzes ist Typisch für Pro-Argumente im LUCENE- Projekt. Tabelle 5.13 beschreibt die Levenshtein-Distanz dieses Satzes zu anderen Entscheidungselementen dieses JIRA-Issues. Die Anwendung der Levenshtein-Distanz kann in diesem Fall kein eindeutiges Ergebnis liefern.

Eine weitere Möglichkeit ist eine regelbasierte Verlinkung von Entscheidungswissen. Ein neues Entscheidungselement wird dabei anhand seines Wissenstyps verlinkt. Tabelle 5.14 beschreibt die identifizierten Regeln, die Entscheidungselemente verlinken. Dabei wird ein neues Entscheidungselement immer mit dem letzten erstellten Entscheidungselement verlinkt, das auf eine Regel zutrifft. Bspw. wird eine neue Entscheidung mit dem letzten erzeugten Issue verlinkt.

Da die Anwendung der Levenshtein-Distanz schlechte Ergebnisse für diese Aufgabe erzeugt, wurde entschieden neue Entscheidungselemente regelbasiert zu verlinken. Die Regeln aus Tabelle 5.14 dienen als Grundlage für die Implementierung. Die Verlinkung durch Regeln ermöglicht es der NutzerIn ein vollständig verlinktes Entscheidungsproblem in einem einzelnen JIRA-Issue-Kommentar zu beschreiben.

Tabelle 5.13: Levenshtein Distanz der Entscheidungselemente in LUCENE-2387

Wissenstyp	Satz	L. Dist.
Entscheidung	Attached patch nulls out the Fieldable reference.	1
Entscheidung	As Tokenizers are reused, the analyzer holds also a reference to the last used Reader. The easy fix would be to unset the Reader in Tokenizer.close(). If this is the case for you, that may be easy to do. So Tokenizer.close() looks like this	1
Entscheidung	29x version of this patch.	1
Issue	Is there a chance that this can also be applied to 3.0.2 / 3.1?	1
Pro	It would be really helpful to get this as soon as possible in the next Lucene version	1
Entscheidung	OK I'll backport	1

Tabelle 5.14: Regeln für Links zwischen Entscheidungselementen.

Quelle	Ziel
Argument	Entscheidung oder Alternative
Entscheidung	Issue
Alternative	Issue
Issue	JIRA-Issue

Tabelle 5.15: Entscheidungsprobleme zu Systemfunktion 7

Entscheidungsproblem	Entscheidung
Wie kann die NutzerIn Entscheidungswissen in JIRA-Issue-Kommentaren verlinken?	Drag and Drop der Entscheidungselemente in Listen- und Graphenansicht
In welcher Ansicht soll Entscheidungswissen in JIRA-Issue-Kommentaren verlinkt werden?	Alle Ansichten mit Entscheidungswissen: WS1.3, WS1.4.1, WS1.4.2, WS1.4.3
Soll die NutzerIn ein visuelles Feedback zur Verlinkung bekommen?	Zeige Erfolgs- oder Fehlermeldung bei neuer Verlinkung.
Soll Entscheidungswissen in JIRA-Issue-Kommentaren durch eine eigene Schnittstelle verlinkt werden?	Nein, nutze eine REST-Schnittstelle für alle Links.
Soll die Listen- und Baumansicht nach einer Verlinkung neu geladen werden?	Lade Ansicht neu vom Server nach Verlinkung durch die NutzerIn.
Wie soll neues Entscheidungswissen automatisch verlinkt werden?	Suche anhand von Regeln das letzte erzeugte Element mit geeignetem Wissenstyp.

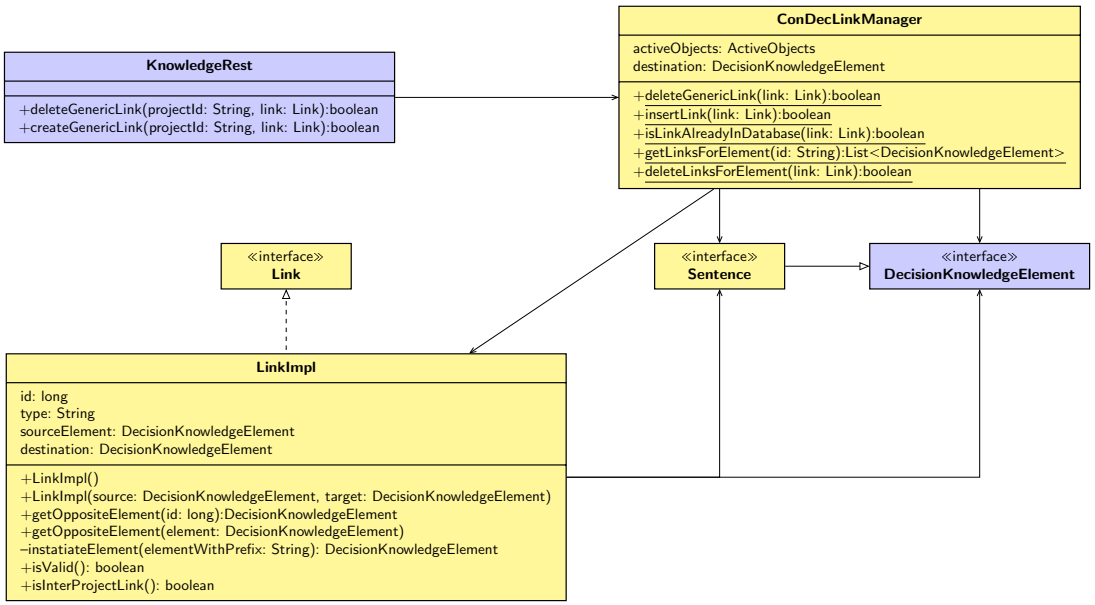


Abbildung 5.17: Klassendiagramm zu allen relevanten Klassen für Systemfunktion 7

5.3.8 Explizites Entscheidungswissen zu JIRA-Issue hinzufügen

Dieser Abschnitt beschreibt den Entwurf und die dabei getroffenen Entscheidungen zu Systemfunktion 8 aus Kapitel 4.2.2.8 auf Seite 26. Abbildung 5.18 illustriert das wichtigste Entscheidungsproblem zu dieser Systemfunktion. Tabelle 5.16 listet weitere Entscheidungsprobleme auf. Abbildung 5.19 zeigt alle nötigen Klassen für diese Systemfunktion.

Ansichten WS1.3: Decision Knowledge View und WS1.4.1: Issue Module bieten der NutzerIn die Möglichkeit neue verlinkte Entscheidungselemente zu erzeugen. Die NutzerIn selektiert dabei ein Entscheidungselement, öffnet das Kontextmenü und gibt in einem Dialog die relevanten Informationen (Beschreibung und Wissenstyp) ein. Das neue Entscheidungselement ist anschließend mit dem selektierten Entscheidungselement verlinkt.

Diese Systemfunktion soll ein neues Entscheidungselement als JIRA-Issue-Kommentar zum selektierten JIRA-Issue hinzufügen. Dafür musste entscheiden werden wie die NutzerIn Entscheidungswissen in Ansichten WS1.3, WS1.4.1 und WS1.4.3: Issue Tab Panel in JIRA-Issue-Kommentaren zu einem JIRA-Issue hinzufügen kann.

Eine Möglichkeit ist die Erweiterung des existierenden Dialogs zur Erzeugung eines Entscheidungselements um die Auswahl des Dokumentationsort. Die NutzerIn kann hier in einer Drop Down Auswahl den Dokumentationsort des neuen Entscheidungselements auszuwählen.

Alternativ kann die NutzerIn einen eigenen Dialog *Add Decision Knowledge In Comment* nutzen. Dieser Dialog ist identisch zum existieren Dialog *Add Decision Knowledge Element*, fügt das neue Entscheidungselement aber in einem JIRA-Issue-Kommentar hinzu.

Da beide Dialoge die selbe Aufgabe haben, wurde entschieden, den existierenden Dialog zu erweitern. Bei der Implementierung musste entschieden werden, ob auch die REST-Schnittstelle zur Erzeugung eines Entscheidungselements wiederverwendet werden kann. Bei einer Wiederverwendung prüft die Serverseite welcher Dokumentationsort ausgewählt wurde. Diese Möglichkeit überträgt allerdings keine Pro- und Contra- Argumente als Wissenstyp. Deshalb wurde entschieden eine eigene REST Schnittstelle zur Erzeugung von JIRA-Issue-Kommentaren zu implementieren.

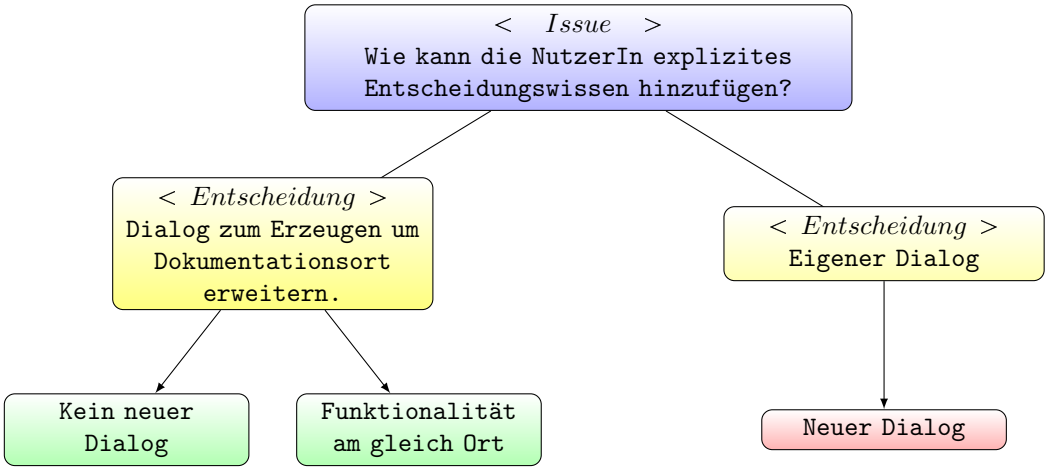


Abbildung 5.18: Entscheidungsbaum zur Entscheidung zur Erzeugung von Entscheidungswissen in JIRA-Issue-Kommentaren

Durch die Implementierung einer eigenen REST-Schnittstelle prüft die Serverseite welchen Dokumentationsort die NutzerIn ausgewählt hat. Die Klasse *ActiveObjectManager* erzeugt einen neuen Kommentar im ausgewählten JIRA-Issue. Es wurde entschieden neue Entscheidungselement mit dem ausgewählten Entscheidungselement zu verlinken. Alternativ ist eine regelbasierte Verlinkung wie in Systemfunktion 7 möglich. Diese Variante kann die NutzerIn aber verwirren, da sie explizit ein JIRA-Issue selektiert hat, zu dem sie ein neues Entscheidungselement verlinken möchte.

Tabelle 5.16: Entscheidungsprobleme zu Systemfunktion 8

Entscheidungsproblem	Entscheidung
Wie kann die NutzerIn explizites Entscheidungswissen hinzufügen?	Dialog zum Erzeugen eines Entscheidungselements um Dokumentationsort erweitern.
Wie soll die Clientseite die Erzeugung behandeln?	Nutzung einer eigenen REST Schnittstelle für Dokumentationsort in JIRA-Issue-Kommentaren
Wie soll die Serverseite die Erzeugung behandeln?	Nutzung einer eigenen REST Schnittstelle. Erzeugung und Verlinkung in *ActiveObjectsManager*
Wie soll das neue Entscheidungselement verlinkt werden?	Verlinkung auf das ausgewählte Element der NutzerIn

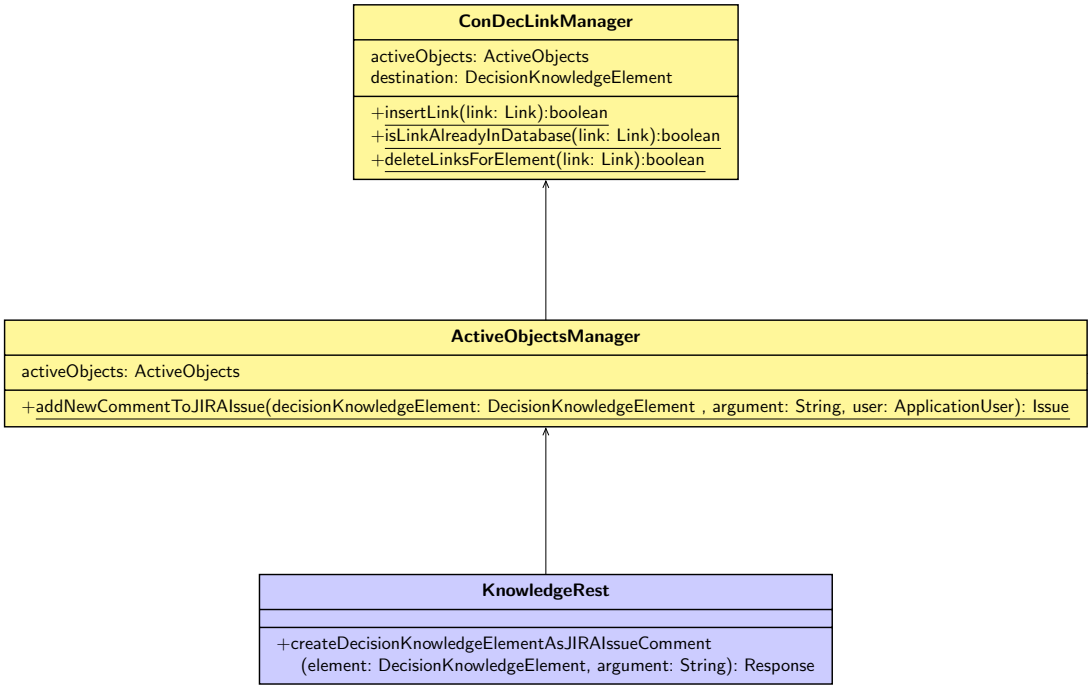


Abbildung 5.19: Klassendiagramm zu allen relevanten Klassen für Systemfunktion 9

5.3.9 Verwalte den Dokumentationsort von Entscheidungswissen

Dieser Abschnitt beschreibt den Entwurf und die dabei getroffenen Entscheidungen zu Systemfunktion 9 aus Kapitel 4.2.2.9 auf Seite 26. Abbildung 5.20 illustriert das wichtigste Entscheidungsproblem zu dieser Systemfunktion. Tabelle 5.17 listet weitere Entscheidungsprobleme auf. Abbildung 5.21 zeigt alle nötigen Klassen für diese Systemfunktion.

Einige von JIRA bereitgestellte Funktionalitäten lassen sich nur auf JIRA-Issues anwenden (bspw. JIRA-Issue Export), nicht aber auf Entscheidungselemente die in anderen Dokumentationsorten persistiert sind. Diese Systemfunktion soll es der NutzerIn ermöglichen, den Dokumentationsort von Entscheidungswissen aus JIRA-Issue-Kommentaren zu ändern. Dafür soll DecXtract Sätze aus JIRA-Issue-Kommentaren in explizite JIRA-Issues konvertieren.

Bei der Implementierung trat das Entscheidungsproblem auf, ob Sätze aus dem ursprünglichen Dokumentationsort gelöscht werden, wenn die NutzerIn den ausgewählten Satz in ein JIRA-Issue-Kommentar konvertieren möchte. Abbildung 5.20 beschreibt dieses Entscheidungsproblem.

Wenn ein Satz nicht aus dem ursprünglichen Dokumentationsort gelöscht wird, können Inkonsistenzen entstehen, sobald das konvertierte JIRA-Issue geändert wird. Allerdings bleibt der Kontext des Kommentars erhalten.

Wird ein Satz aus dem ursprünglichen Dokumentationsort gelöscht, ist die NutzerIn angehalten das konvertierte JIRA-Issue mit weiterem Entscheidungswissen, Systemwissen und Projektwissen zu verlinken. Somit überträgt sich der Kontext des ursprünglichen Dokumentationsorts auf das neue Entscheidungselement. Diese Annahme wird positiv bewertet.

Da Inkonsistenzen vermieden werden, wurde entscheiden Sätze aus dem ursprünglichen Dokumentationsort zu entfernen. Da neue JIRA-Issues auch kommentiert werden können, ist es möglich, dass sich deren Kontext ändert. Dies kann im ursprünglichen Dokumentationsort nicht nachverfolgt werden. Daher ist die NutzerIn hier angehalten, das neue JIRA-Issue entsprechend zu verlinken.

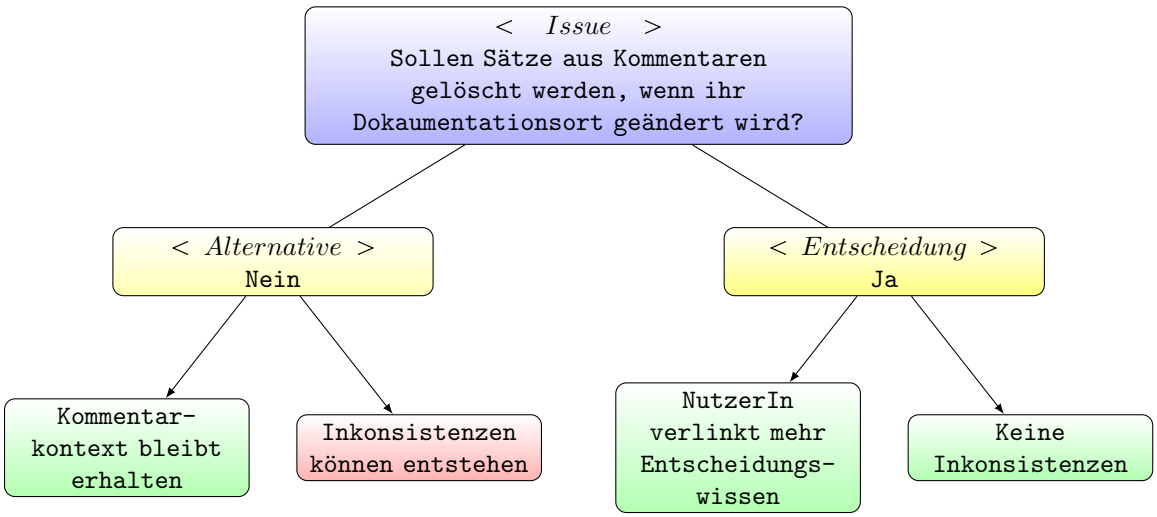


Abbildung 5.20: Entscheidungsbaum zur Entscheidung zur Persistierung von Sätzen mit geändertem Dokumentationsort.

Es wurde entschieden, diese Funktionalität in das Kontextmenü aus Entwurfsabschnitt 5.3.6 auf Seite 48 einzubinden.

ConDec stellt bereits die Methode *insertDecisionKnowledgeElement* in der Klasse *IssueStrategy* zur Verfügung, um JIRA-Issue-Instanzen zu ereugen. Es ist naheliegend, diese Methode zu Nutzen um JIRA-Issues zu erzeugen.

Zudem stellte sich die Frage, ob die aktuelle JIRA-Persistenzstrategie benutzt werden soll, um JIRA-Issue Instanzen zu erzeugen. Ist jedoch die AO-Strategie aktiv, wird neues Entscheidungswissen nur in eine andere AO-Tabelle geschrieben. Dies entspricht nicht der Absicht dieser Systemfunktion. Daher wurde entschieden immer die Methode *insertDecisionKnowledgeElement* der JIRA-Issue-Strategie zu nutzen.

Tabelle 5.17: Entscheidungsprobleme zu Systemfunktion 9

Entscheidungsproblem	Entscheidung
Sollen Sätze, deren Dokumentationsort geändert wird, aus Kommentaren gelöscht werden?	Ja, NutzerIn hat volle Kontrolle über Kontext.
Wie soll der Dokumentationsort eines Entscheidungselement angepasst werden?	Kontextmenüeintrag für gewünschtes Element.
Wie soll ein neues JIRA-Issue erzeugt werden?	Wiederverwendung der JIRA-Issue Strategie
Welche Änderungsrichtung soll von DecXtract behandelt werden?	Kommentar → JIRA-Issue
Sollen neue JIRA-Issues entsprechend der aktuellen JIRA-Persistenz-Strategie erzeugt werden?	Nein, benutze immer die JIRA-Issue Strategie

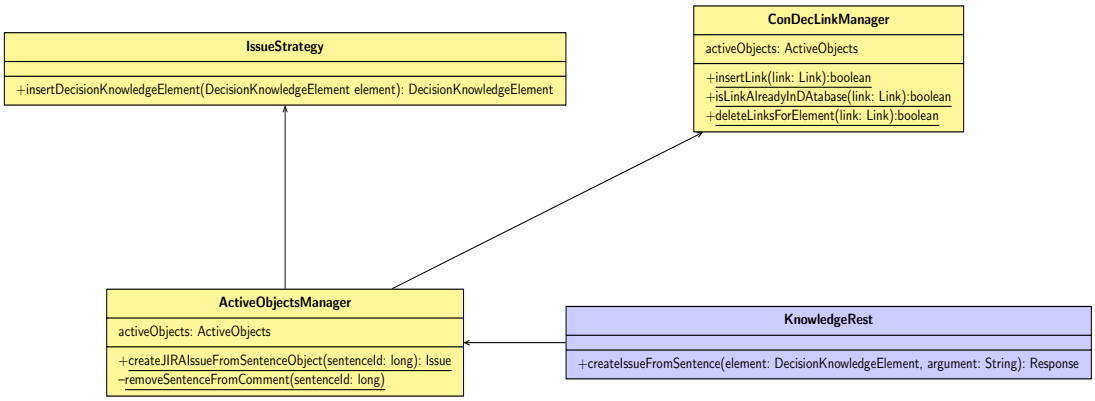


Abbildung 5.21: Klassendiagramm zu allen relevanten Klassen für Systemfunktion 9

5.3.10 Metriken zur Beurteilung der Vollständigkeit des Entscheidungswissens berechnen

Dieser Abschnitt beschreibt den Entwurf und die dabei getroffenen Entscheidungen zu Systemfunktion 10 aus Kapitel 4.2.2.10 auf Seite 27. Abbildung 5.22 illustriert das wichtigste Entscheidungsproblem zu dieser Systemfunktion. Tabelle 5.18 listet weitere Entscheidungsprobleme auf. Abbildung 5.23 zeigt alle nötigen Klassen für diese Systemfunktion.

JIRA bietet verschiedene Berichte (hier: Reports) an, um Auswertungen wie bspw. Burndown Charts von ausgewählten Projekten zu erstellen. Es ist möglich, neue Reports in diese Auswertungen einzufügen. Die NutzerIn soll eine Möglichkeit bekommen, verschiedene Kennzahlen zum Entscheidungswissen im ausgewählten Projekt einzusehen. Es wurde entschieden, diese Kennzahlen mit Boxplots und Kuchendiagrammen zu visualisieren. In diesem Zuge stellte sich das Entscheidungsproblem, ob diese Grafiken auf Serverseite oder Nutzerseite erzeugt werden sollen. Abbildung 5.22 beschreibt dieses Entscheidungsproblem.

Die erste Möglichkeit erzeugt Grafiken auf Serverseite. Die Grafiken werden in Binärdateien konvertiert und mit einem Velocity Template an die Clientseite übertragen. Diese Aufgabe ist technisch anspruchsvoll und erhöht die Netzwerklast, da viele Bilder an die Nutzerseite übertragen werden.

Eine Alternative ist der Aufbau von Grafiken auf Nutzerseite mit JavaScript. Dafür werden lediglich die zugrunde liegenden Kennzahlen an die Nutzerseite übertragen. Diese Möglichkeit nutzt JavaScript Bibliotheken für ansehnliche Grafiken die Nutzerinteraktionen erlauben. Aus diesem Grund wurde entschieden die Grafiken auf Nutzerseite mit JavaScript zu erzeugen.

Aufgrund der Open-Source Lizenz und der guten Dokumentation, wurde entschieden die Java Script Bibliothek ECharts[7] einzusetzen.

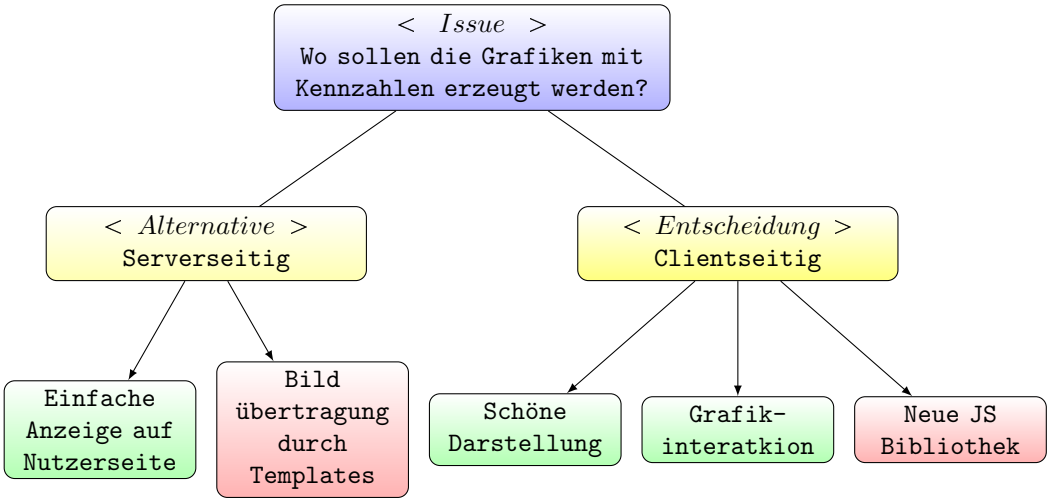


Abbildung 5.22: Entscheidungsbaum zur Entscheidung von Links zwischen verschiedenen Elementen des Entscheidungswissens.

[7] ECharts: https://ecomfe.github.io/echarts-doc/public/en/index.html Zuletzt aufgerufen am 17.12.2018

Folgende Metriken sollen berechnet und dargestellt werden:

- Kommentare pro JIRA-Issue
  Beschreibt das Diskussionsverhalten der NutzerInnen
- Commits pro Issue
  Beschreibt die Anzahl der nötigen Änderungen für ein JIRA-Issue
- Link Distanz
  Beschreibt die Komplexität einer Entscheidung. Wird für die Entscheidungstypen Issue, Alternative, Entscheidung berechnet
- Verteilung der Wissenstypen
  Beschreibt die Quantität der jeweiligen Wissenstypen.
- Vollständigkeit der Entscheidungen
  Beschreibt für je Issue, Alternative, Entscheidung, welche Entscheidungselemente verlinkt sind
- Anzahl Sätze mit Entscheidungselementen
  Beschreibt die Nutzung von Kommentaren zu Entscheidungsdokumentation.

Die Metrik für die Anzahl der Commits pro Issue stellte eine besondere Herausforderung dar. Die Voraussetzung zum Einsatz dieser Metrik ist das JIRA-Plug-In „Git Integration for JIRA“[8]. Dieses muss installiert und eingerichtet (Das gewünschte JIRA-Projekt muss mit einem GIT Server verbunden) sein. Dieses Plug-In bietet eine REST-Schnittstelle, um Commits im JSON-Format zu gewünschten JIRA-Issues anzufragen. Jedoch sind REST-Abfragen von Serverseite standardmäßig nicht authentifiziert. Eine Möglichkeit dieses Problem zu lösen ist die explizite Aufforderung an die NutzerIn ihren Nutzernamen und Passwort einzugeben. Eine weitere Möglichkeit ist die Verwendung der OAuth-Methode[9]. Dieses Verfahren ermöglicht es den JIRA-Server mit anderen Anwendungen zu verbinden und über ein Public und Private Key Paar eine sichere Verbindung herzustellen. Da Oauth nur einmalig eingerichtet werden muss und einen höheren Sicherheitsstandard sowie Komfort bietet als die Eingabe der Nutzeranmeldeinformationen, wurde entschieden OAuth zu verwenden. Um OAuth auch für zukünftige Nutzungsfälle zugänglich zu machen, wurden entsprechende Methoden entwickelt die beliebige REST-Schnittstellen ansprechen können. Die entsprechenden Sicherheitsmechanismen muss die NutzerIn einmalig in Ansicht WS1.1 All Projects Admin View einrichten.

Zudem wurde entschieden der NutzerIn die Möglichkeit zu bieten, die zugrundeliegenden Kennzahlen einzusehen. Die NutzerIn kann die Zahlen anschließend bspw. in Excel kopieren oder LaTeX-Tabellen erzeugen. Zur Anzeige muss die NutzerIn die gewünschte Grafik selektieren und erhält anschließend die Kennzahlen in einem Pop-Up.

Für diese Anzeige wurde entschieden, den jeweiligen JIRA-Issue-Schlüssel zu jeder Kennzahl zu präsentieren. Dies ermöglicht es der NutzerIn einzusehen, welche Entscheidungselemente weitere Aufmerksamkeit benötigen.

[8] Git Integration for JIRA: https://bigbrassband.com/api-doc.html Zuletzt aufgerufen am 17.12.2018
[9] JIRA OAuth:https://developer.atlassian.com/server/jira/platform/oauth/ Zuletzt aufgerufen am 17.12.2018

Tabelle 5.18: Entscheidungsprobleme zu Systemfunktion 10

Entscheidungsproblem	Entscheidung
In welcher Ansicht sollen Metriken gezeigt werden?	Ansicht WS1.5: Decision Knowledge Report View
Wie sollen Metriken präsentiert werden?	Boxplots und Kuchendiagramme bieten verständliche und ansehnliche Darstellung.
Welche Bibliothek soll zur Grafikerzeugung benutzt werden?	*Echarts.js* bietet geeignete Funktionalität und ist frei zugänglich.
Wie können die zugrunde liegenden Datenreihen eingebunden werden?	JIRA-Issue Schlüssel mit entsprechendem Wert.
Wie können die zugrunde liegenden Datenreihen angezeigt werden?	Datenreihe bei Selektion der Grafik in Popup anzeigen.
Welche Technologie soll zur Authentifizierung genutzt werden?	OAuth bietet besten Sicherheitsstandard nach einmaliger Registrierung durch Application Link.

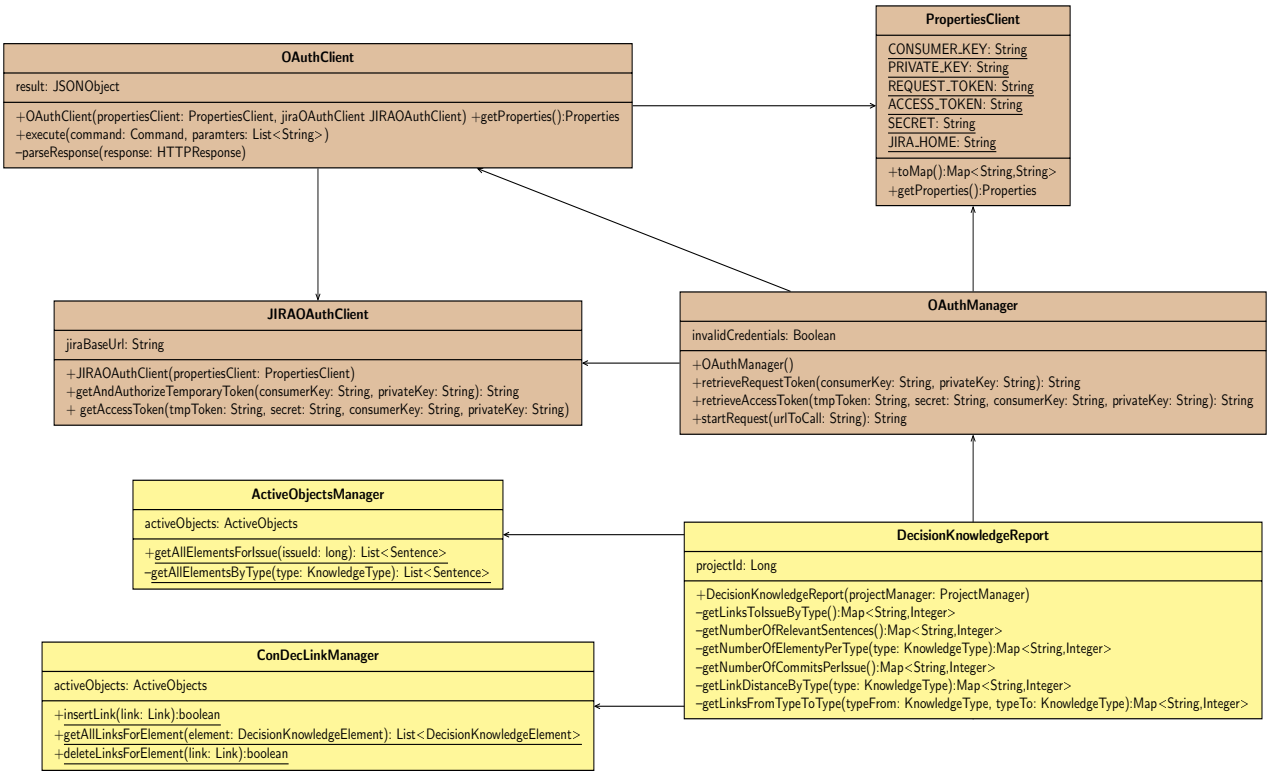


Abbildung 5.23: Klassendiagramm zu allen relevanten Klassen für Systemfunktion 10

5.4 Ergebnisse

Dieser Abschnitt zeigt die Ergebnisse der Implementierung anhand von Abbildungen der Ansichten in denen DecXtract arbeitet.

Abbildung 5.24 zeigt Ansicht WS1.1: All Projects Admin View. Die Projekt AdministratorIn nutzt hier Systemfunktion 1 um DecXtract in ausgewählten Projekten zu (De)aktivieren. Des Weiteren bietet diese Ansicht die Aktivierung der OAuth Authentifizierung für Systemfunktion 10.

Abbildung 5.25 zeigt Ansicht WS1.2: Single Projects Admin View. Die Projekt AdministratorIn kann ConDec und DecXtract hier konfigurieren. Durch die Konfigurationsmöglichkeiten kann sie hier den Klassifikator für JIRA-Issue-Kommentare und Icons zur manuellen Klassifikation (De)aktivieren. Der Button *Validate Sentence Database* führt eine Validierung der AO-Datenbank durch. Diese Methode prüft alle Elemente der AO-Datenbank auf Konsistenz zu ihrem Dokumentationsort. Der Button *Classify all Comments* klassifiziert mit Systemfunktion 2 alle JIRA-Issue-Kommentare des aktuellen JIRA-Projektes.

Abbildung 5.26 zeigt Ansicht WS1.4.1: Issue Module View. NutzerInnen betrachten hier explizit angelegtes Entscheidungswissen aller Dokumentationsorte. Systemfunktion 5 erzeugt die Datenstruktur für diese Ansicht. Die NutzerIn kann zu einem bestimmten JIRA-Issue mit Systemfunktion 6 das Entscheidungswissen verwalten, mit Systemfunktion 7 das Entscheidungswissen verlinken, mit 8 neues, explizites Entscheidungswissen in einem gewünschten Dokumentationsort hinzufügen, oder mit Systemfunktion 9 den Dokumentationsort verwalten.

Abbildung 5.27 zeigt Ansicht WS1.4.2: Issue Comments View. EntwicklerInnen und Rational ManagerInnen schreiben in dieser Ansicht Kommentare zu einem JIRA-Issue. Diese Ansicht hinterlegt Entscheidungswissen farblich und platziert ein Icon zur Identifikation des Wissenstyps. Im unteren Teil dieser Ansicht können NutzerInnen Kommentare schreiben und mit Systemfunktion 3 Entscheidungswissen durch Tags klassifizieren.

Abbildung 5.28 zeigt Ansicht WS1.4.3: Decision Knowledge Extraction Tab Panel View. Diese Ansicht nutzt Systemfunktion 4 um Entscheidungswissen in den Kommentaren des aktuellen JIRA-Issues und verlinkter JIRA-Issues anzuzeigen. NutzerInnen können dieses Entscheidungswissen mit Systemfunktion 6 verwalten und mit Systemfunktion 7 verlinken.

Abbildung 5.29 zeigt Ansicht WS1.5: Issue Report View. NutzerInnen können hier Kennzahlen zum Entscheidungswissen im aktuellen JIRA-Projekt einsehen. Systemfunktion 10 erzeugt diese Ansicht.

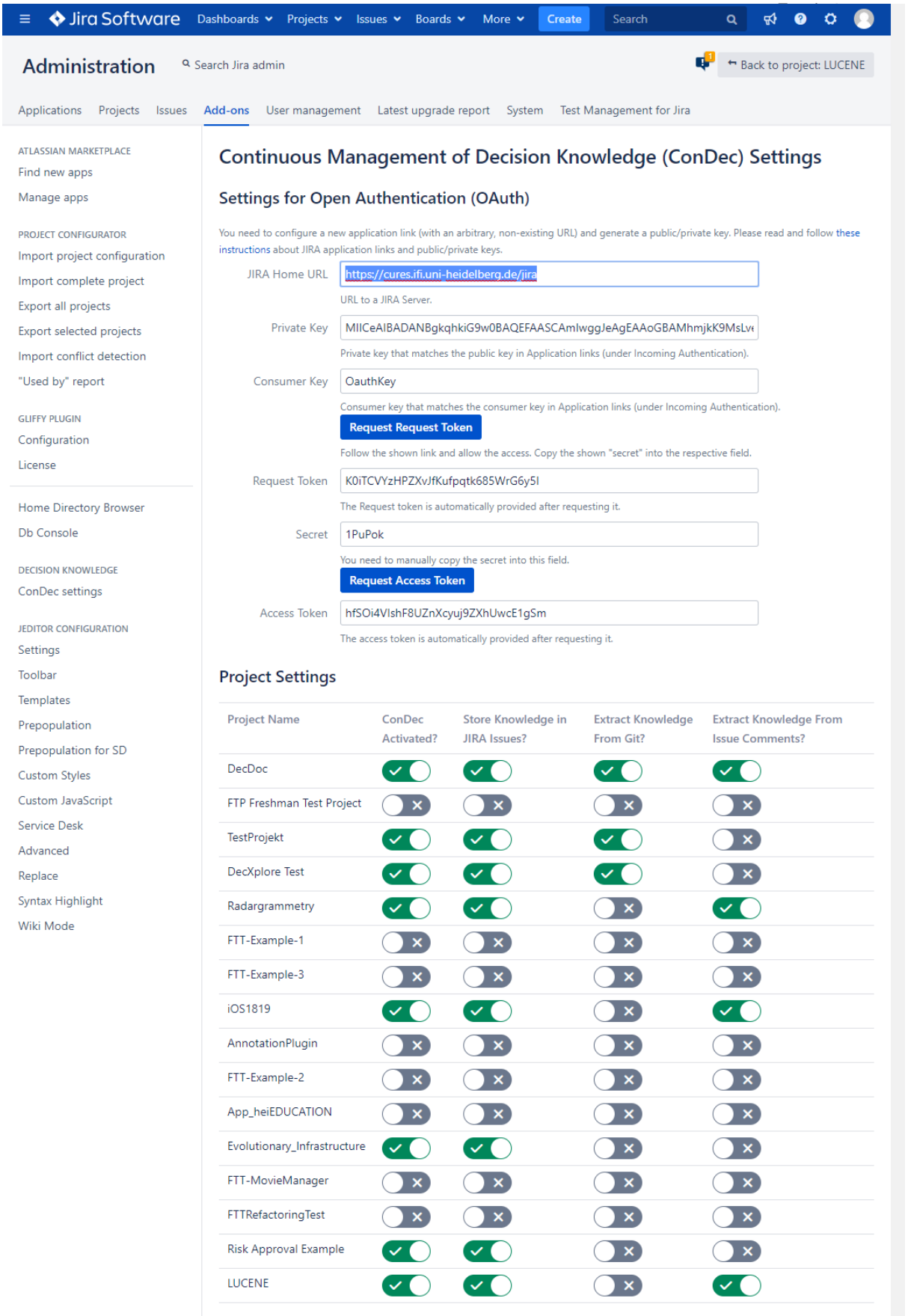


Abbildung 5.24: Ansicht WS1.1: All Projects Admin View

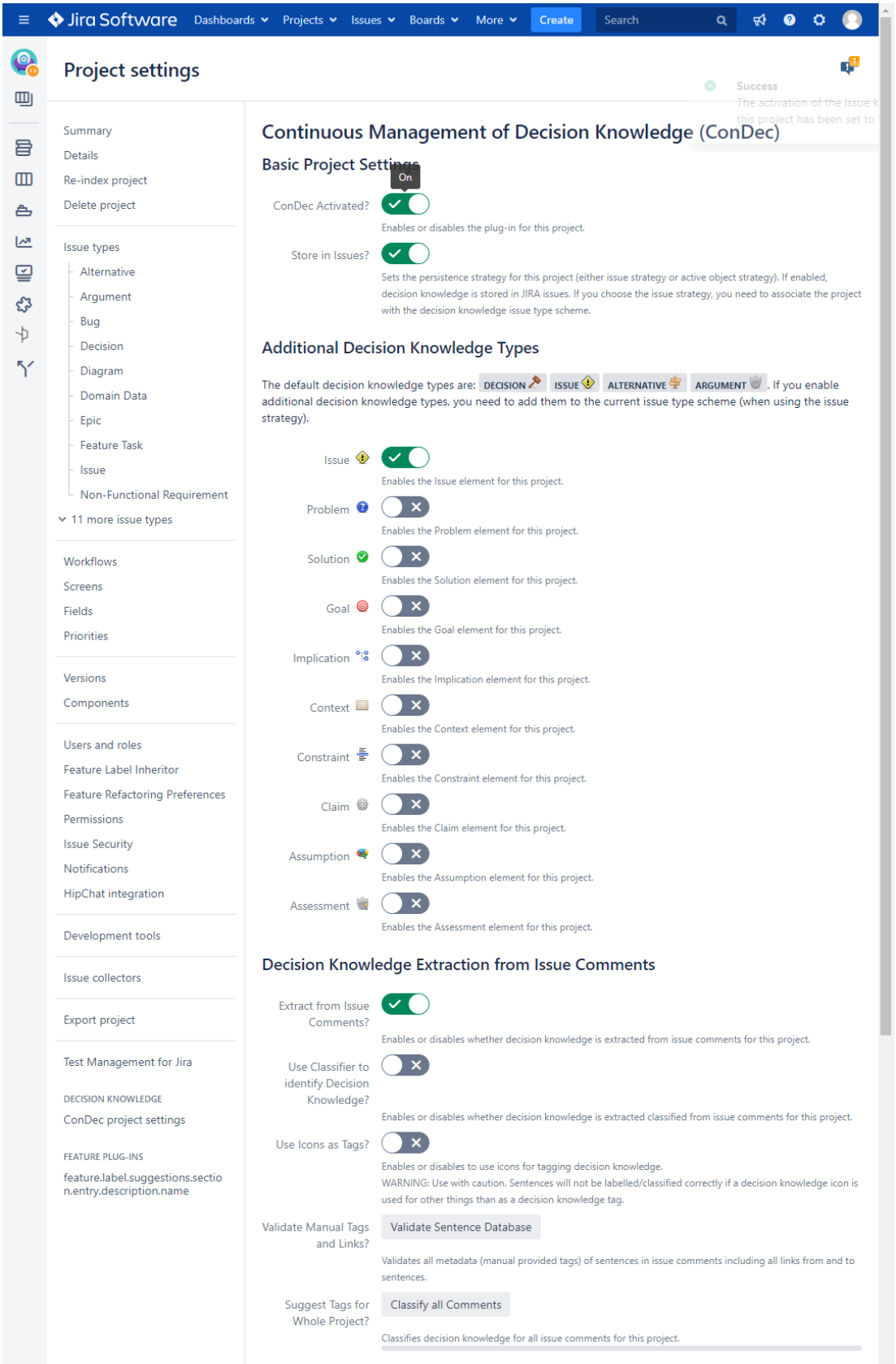


Abbildung 5.25: Ansicht WS1.2: Single Projects Admin View

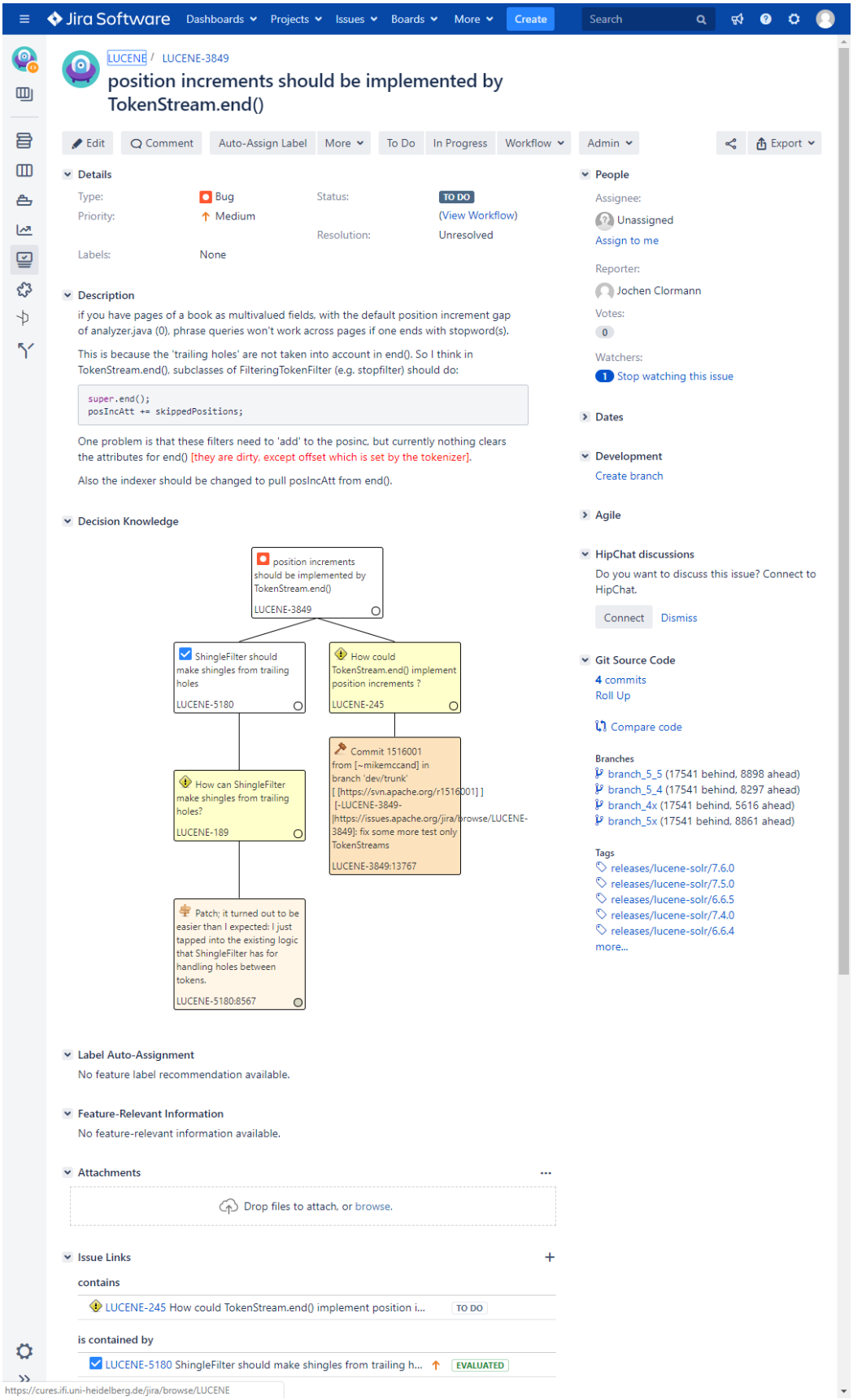


Abbildung 5.26: Ansicht WS1.4.1: Issue Module View

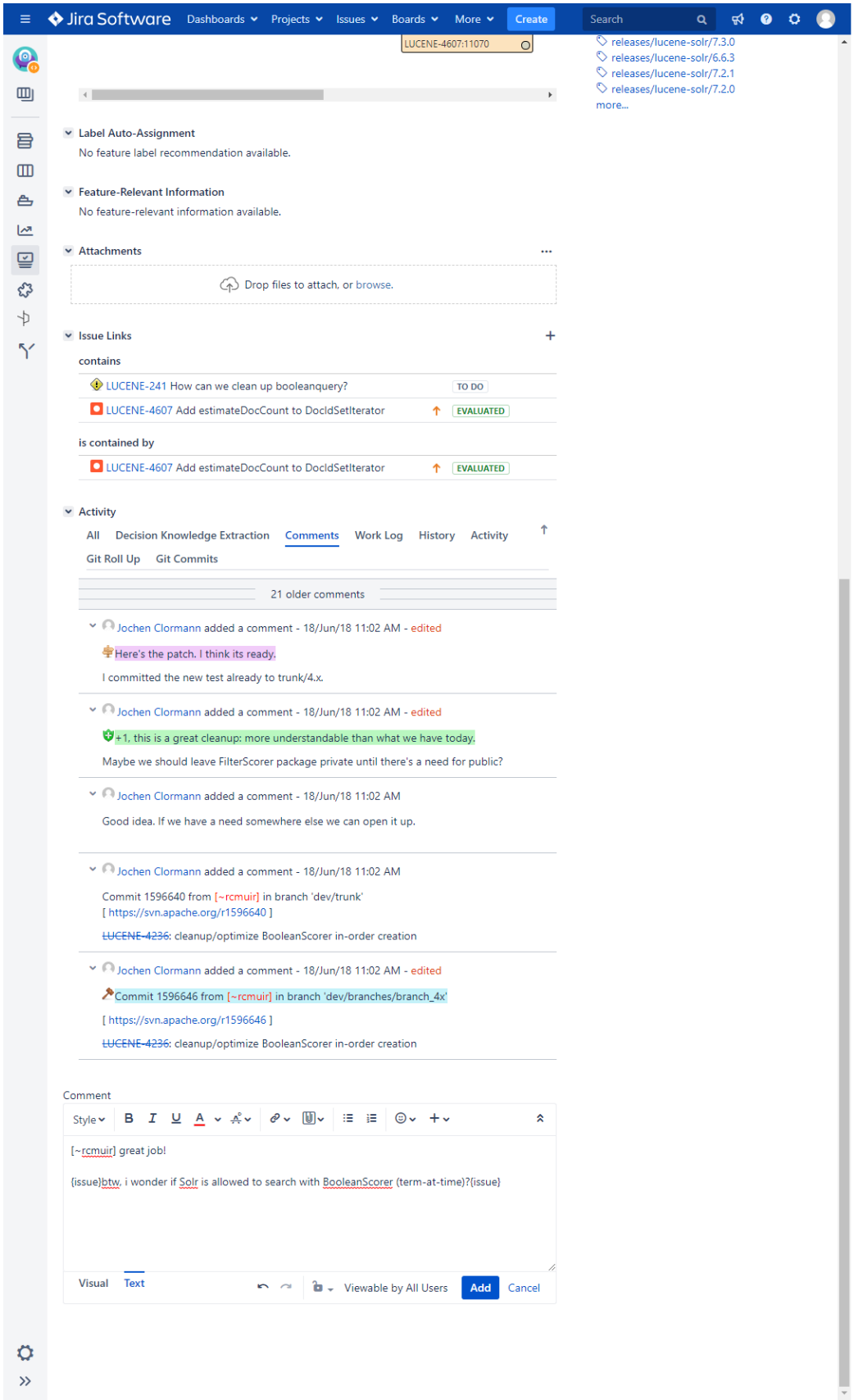


Abbildung 5.27: Ansicht WS1.4.2: Issue Comments View

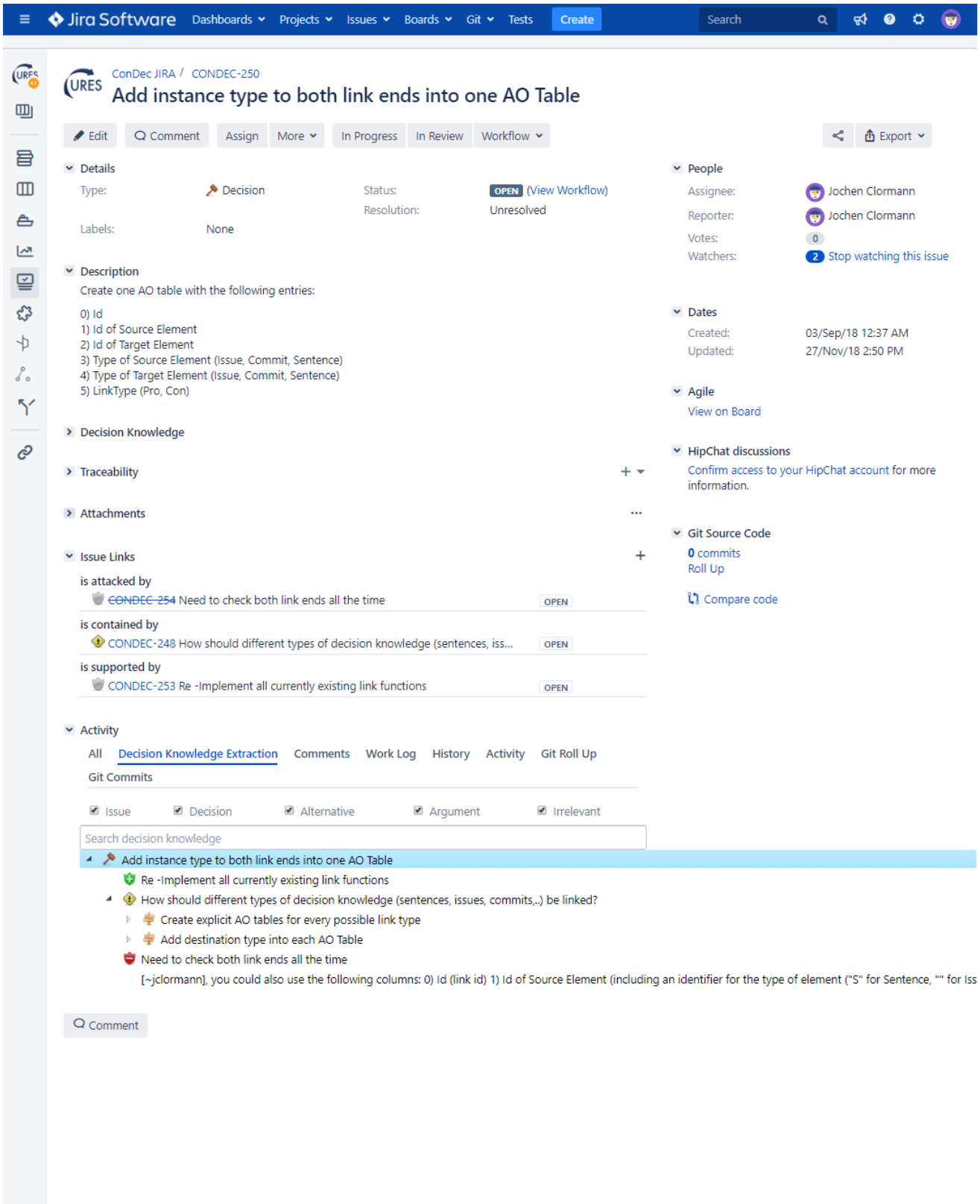


Abbildung 5.28: Ansicht WS1.4.3: Decision Knowledge Extraction Tab Panel View

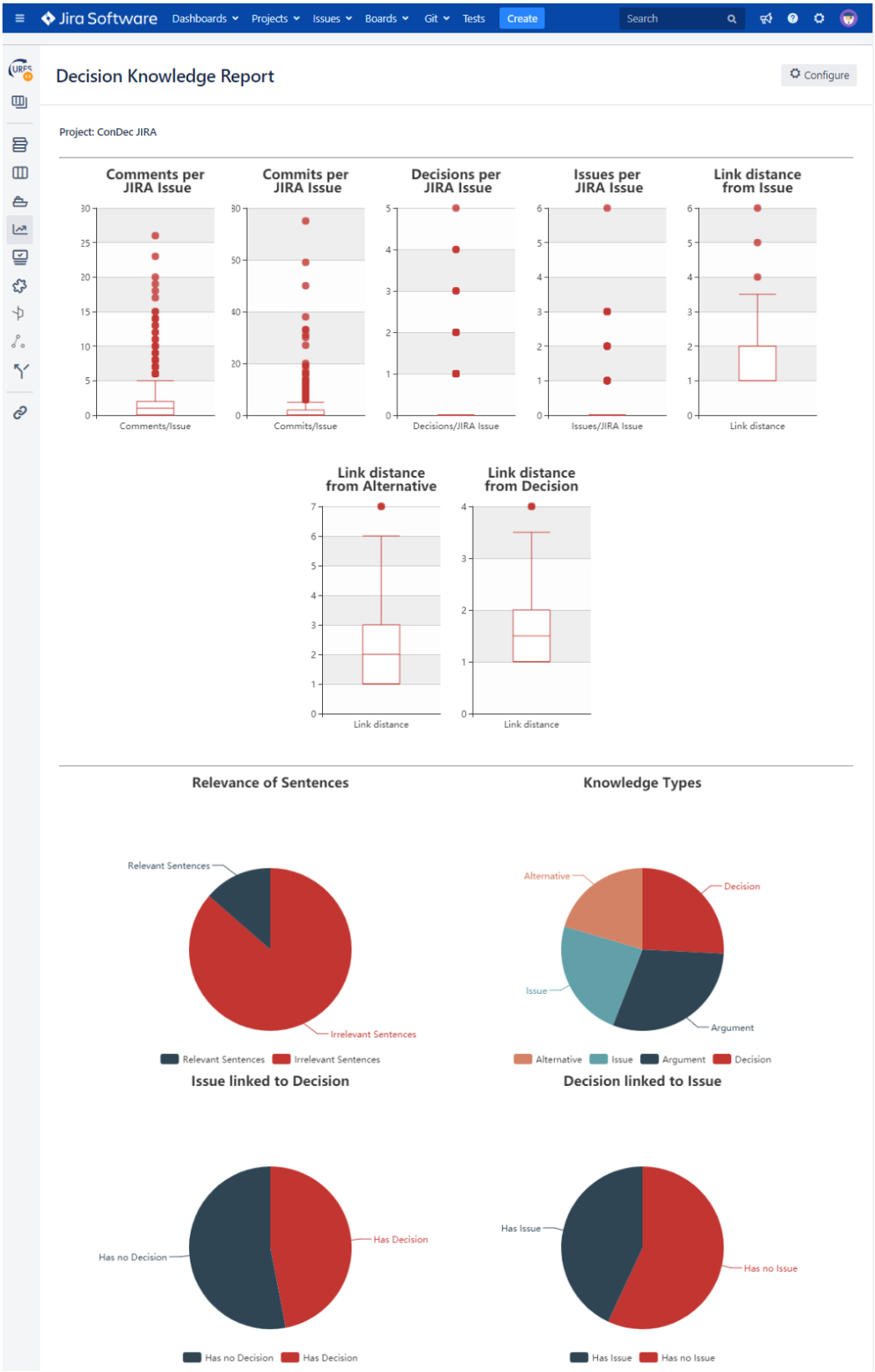


Abbildung 5.29: Ansicht WS1.5: Issue Report View

6 Qualitätssicherung

Dieses Kapitel beschreibt die Planung, Durchführung und Auswertung von Testfällen. Die gefundenen Fehler werden kurz mit Lösungsansatz beschrieben. Die Qualitätssicherung wird für jede Systemfunktion in drei Teststufen durchgeführt. Abschnitt 6.1 beschreibt die Planung und Durchführung der qualitätssichernden Maßnahmen. Abschnitt 6.4 erläutert die Überprüfung der Güte des Klassifikators. Abschnitt 6.5 definiert die durchgeführten Testfälle für alle Teststufen und Systemfunktionen.

6.1 Planung qualitätssichernder Maßnahmen

Die Qualitätssicherung wird nach Spillner und Linz in den Teststufen Komponententest, Integrationstest und Systemtest durchgeführt [29].

Komponententests dienen der Überprüfung einzelner Methoden von DecXtract mit JUnit. Als Testumgebung wird die lokale Eclipse-Instanz genutzt. Die Tests werden sowohl lokal ausgeführt, als auch von dem Continuous-Integration-Server TravisCI[1]. Dieses System ist in GitHub integriert und testet automatisiert jeden Commit. Komponententestfälle werden durch gültige Äquivalenzklassen und ungültige Äquivalenzklassen definiert. Gültige Äquivalenzklassen testen das Systemverhalten mit validen Eingaben. Ungültige Äquivalenzklassen testen das Systemverhalten mit invaliden Eingaben. Tabelle 6.3 definiert alle durchgeführten Komponententests. Ein Eintrag in der Tabelle entspricht dabei mehreren JUnit Tests. Dafür wurden die Testdaten in Menge und Eingabereihenfolge variiert. Komponententests sind mit *Kt.* nummeriert.

Integrationstests prüfen die Integration von DecXtract in ConDec. Diese Testfälle testen ConDec-Komponenten, die mit DecXtract geändert wurden. Dies betrifft die Klassen *GraphImpl* zur Erzeugung der Listen und Baumansicht. Um das Systemverhalten optimal zu prüfen, werden DecXtract und ConDec typische Testdaten genutzt. Integrationstests nutzen ebenfalls Äquivalenzklassen zur Testfalldefinition. Tabelle 6.4 definiert die Testfälle der Integrationstests.

Systemtests prüfen DecXtract und ConDec auf der Oberfläche. Diese Testfälle sind durch ein genaues Testdurchführungsprotokoll definiert und beschreiben eine erwartete Systemreaktion. Da DecXtract in ConDec integriert ist, prüfen Systemtests auch die Integration von DecXtract in ConDec über die Oberfläche. Tabelle 6.5 zeigt die Testfälle der Systemtests.

6.2 Tests funktionaler Anforderungen

Für jede Systemfunktion werden entsprechende Testfälle entworfen. Aufgrund der Eigenheit der Systemfunktionen, durchlaufen diese nicht immer alle drei Teststufen.

[1] TravisCI: https://travis-ci.org/ Zuletzt aufgerufen am 17.12.2018

SF1: DecXtract aktivieren
Diese Systemfunktion wird ausschließlich von Systemtests getestet. Die Testfälle St.63 bis St.66 prüfen diese Funktionalität. Die Testfälle prüfen die Funktionalität DecXtract in ConDec zu (De)aktivieren. Außerdem wird geprüft, ob die gewählte Einstellung gespeichert und bei einem neuen Besuch der Seite entsprechend angezeigt wird. Diese Systemtests werden zusätzlich in Ansicht WS1.2: Single Project Admin View durchgeführt.

Im Projektverlauf kam es mehrmals zur Überarbeitung des JavaScript-Cods von Ansicht WS1.2. Dies sorgte dafür, dass Testfall St.66 mehrmals fehlschlug. Zu jedem Zeitpunkt wurden die Probleme im JavaScript-Code korrigiert.

SF2: Entscheidungswissen in JIRA-Issue-Kommentaren automatisch klassifizieren
Testfälle Kt.2 bis Kt.10 und St.71 bis St.81 definieren die durchgeführten Komponenten- und Systemtests für diese Systemfunktion. Da das Verhalten der Weka Klassen in der JUnit Umgebung unerklärliche Exceptions hervorruft, wurden die Klassifikator Klassen durch einen Mock ersetzt.

Komponententest KT.5 deckt auf, dass der Klassifikator das Attribut *isTagged* der Klasse *Sentence* auf *true* setzt, was eine weitere binäre Klassifikation unterbindet. Das Attribut verhindert allerdings auch den Einsatz des feingranularen Klassifikators. Dieses Problem wurde durch das neue Attribut *isTaggedFineGrained* gelöst.

SF3: Entscheidungswissen in JIRA-Issue-Kommentaren manuell klassifizieren
Die Komponententests Kt.12 bis Kt.24 testen die Trennung von Sätzen und deren manuelle Klassifikation mit verschiedenen Kombinationen aller Textbausteine (Tabelle 5.2 Seite 38). Alle Testdaten variieren Textbausteine in Menge und Reihenfolge. Sytemtests St.71 bis St.81 testen diese Funktionalität in Ansicht WS1.4.2.

Komponententest Kt.12 deckt auf, dass zwei aneinanderhängende Entscheidungselemente wie z.B.:„{pro}+1{pro} {con}-1{con}“, drei Sätze produzieren. Der zweite Satz besteht aus einem Leeerzeichen und hat den Wissenstyp *Pro*. Dies wurde behoben, indem die rekursive Funktion eindeutige Start- und Endpunkte für jedes Macro sucht.
Komponententest Kt.17 und Kt.23 decken auf, dass die Kombination von Groß- und Kleinschrift in Tags (bspw.: {pro}+1{Pro}) nicht erkannt wird. Dies wurde behoben indem Tags fortan ohne Beachtung der Groß- und Kleinschreibung geprüft werden.
Systemtest St.74 deckt auf, dass die Klassifizierung mit Icons für den Wissenstyp *Alternative* nicht funktioniert aber für alle anderen Wissenstypen. Dies blieb vom Komponententest unbemerkt da Icons durch Makros ersetzt werden. Hier war das Icon der Alternative falsch definiert.

SF4: Entscheidungswissen in JIRA-Issue-Kommentaren anzeigen
Die Integreationstests It.52 bis It.57 prüfen das Verhalten der Listenansicht in WS.1.4.2 und WS1.3 mit Testdaten von DecXtract und ConDec. Die Systemtests St.71 bis St.75 testen die Funktionalität der Listenansicht auf der Oberfläche. Beide Teststufen erzeugen Sätze in JIRA-Issue-Kommentaren und prüfen deren Existenz in der Listenansicht.

Systemtest St.75 deckt auf, dass irrelevante Sätze nicht bei Selektion der Filtercheckbox ausgeblendet werden. Dieser Zustand folgt aus der inkorrekten Benennung der HTML Elemente.

SF5: Entscheidungswissen in JIRA-Issue-Kommentaren im Kontext zu weiterem Entscheidungswissen, Projektwissen und Systemwissen anzeigen
Die Integreationstests It.52 bis It.62 prüfen das Verhalten der Baum- und Listenansicht in Ansicht WS1.4.1, WS1.4.2 und WS1.3. Die Systemtests St.71 bis St.75 und St.67 bis St.70 testen die Funktionalität der Baum- und Listenansicht auf der Oberfläche.

Systemtest St.70 deckt auf, dass Links von Sätzen zu JIRA-Issues nicht gelöscht werden können, da die REST Methode *deleteLink* nur JIRA-Issue-Links löscht. Dies wurde mit einer einheitlichen REST-Schnittstelle gelöst, die auf Serverseite überprüft, in welcher Persistenzmethode ien Link gelöscht wrid.

SF6: Klassifiziertes Entscheidungswissen in JIRA-Issue-Kommentaren bearbeiten
Die Komponententests Kt.26 bis Kt.34 und Systemtests St.76 bis St.81 testen die Möglichkei,t Entscheidungswissen in JIRA-Issue-Kommentaren zu bearbeiten. Dafür wird der Wissenstyp und die Beschreibung geändert.

Testfall St.78 deckt auf, dass Makros doppelt in ein Kommentar eingefügt werden. Dieser Zustand wurde bereits durch Einführung des Attributes *isTagged* aufgelöst. Testfall St.80 deckt auf, dass Exceptions auftreten, wenn der Text gekürzt wird. Dieser Zustand wird verhindert, indem Strings auf ihre Länge geprüft werden, bevor die Methode *substring(.)* darauf zugreift.

SF7: Entscheidungswissen in JIRA-Issue-Kommentaren verlinken, Verlinkung lösen
Die Komponententests Kt.35 bis Kt.44 und Systemtests St.67 bis St.70 sowie St.82 bis St.84 testen die Funktionalität, Entscheidungswissen verschiedener Dokumentationsorte miteinander zu verlinken. Die Testfälle prüfen dabei alle möglichen Links zwischen den Dokumentationsorten herzustellen und zu löschen.

Komponententest Kt.44 deckt auf, dass nicht geprüft wird ob beide Elemente eines Links existieren. Die eingeführte Funktion *isValid()* prüft ob eine Instanz der Klasse *Link* an jedem Ende eine Instanz von *DecisinoKnowledgeElement* besitzt.

SF8: Explizites Entscheidungswissen zu JIRA-Issue hinzufügen
Die Komponententests Kt.45 bis Kt.48 definieren Testfälle, um explizites Entsheidungswissen in den Kommentaren hinzuzufügen. Diese Testfälle konnten keine Fehler finden.

SF9: Verwalte den Dokumentationsort von Entscheidungswissen
Die Komponententests Kt.49 bis Kt.51 testen die Funktionalität, den Dokumentationsort eines Entscheidungselements in JIRA-Issue-Kommentaren anzupassen.

Testfall Kt.51 deckt auf, das keine Überprüfung stattfindet, ob ein Objekt der Klasse *Sentence* mit übergebenen Id tatsächlich existiert. Eine neue Methode *doesSentenceExist()* in der Klasse *ActiveObjectsManager* prüft diesen Zustand.

SF10: Metriken zur Beurteilung der Vollständigkeit des Entscheidungswissens berechnen
Die Systemtests St.85 bis St.88 testen die Funktionalitäten von Ansicht WS.1.5. Die Testfälle konnten keine Fehler feststellen.

6.3 Tests nichtfunktionaler Anforderungen

NFR9: Funktionale Eignung: Funktionale Vollständigkeit
Alle Systemtestfälle wurden aus dem Ausschreibungsdokument sowie dem Anforderungskapitel abgeleitet. Alle Anforderungen sind durch erfüllte Systemtestfälle abgedeckt.

NFR10: Funktionale Eignung: Funktionale Korrektheit
Alle Komponenten sind durch JUnit Tests abgedeckt. Die Gesamtabdeckung beträgt 74.7%. Für jeden Eingabeparameter einer Methode sind alle möglichen Äquivalenzklassen abgedeckt.[2]

NFR11: Leistungseffizienz: Zeitverhalten
Die Messungen wurden ineiner lokalen JIRA-Instanz mit Intel Core i5 7200 U Prozessor und 16Gb RAM durchgeführt. Der Test wurde mit mehreren Kommentaren mehrmals durchgeführt. Dabei wurde die Anzahl der Sätze sowie die Länge der Sätze variiert. Tabelle 6.1 zeigt die durchschnittlich gemessenen Zeiten. Es ist erkennbar, dass die Satzlänge mehr Einfluss auf das Zeitverhalten hat als die Anzahl der Sätze. Dies ist dem *Tokenizer* geschuldet, der Wortsequenzen aus drei aufeinanderfolgenden Worten bildet. Je länger ein Satz, desto mehr Daten werden generiert.

Tabelle 6.1: Ergebnisse der Messung zum Zeitverhalten.

Anzahl Sätze	∅ Satzlänge	∅ Dauer [s]
5	5	1.21
5	10	2.29
5	20	3.90
10	5	1.34
10	10	2.53
10	20	4.45
20	5	1.59
20	10	2.88
20	20	4.42

NFR12: Kompatilität: Interoperabilität
19 JUnit Integrationstests stellen die Integration von DecXtract in ConDec sicher. Dabei erreichen die DecXtract Testfälle eine Abdeckung von 74,4 % in ConDec Klassen.

NFR13: Benutzerfreundlichkeit: Ästhetik der Benutzeroberfläche
Die Wissenstypen der Entscheidungselemente in JIRA-Issue-Kommentaren sind farblich gekennzeichnet. Die Farben orientieren sich an der Farbgebung der Wissenstypen in ConDec.

[2] Aktuelle Testabdeckung einsehbar in CodeCov: https://codecov.io/gh/cures-hub/cures-condec-jira/tree/master/src/main/java/de/uhd/ifi/se/decision/management/jira Zuletzt aufgerufen am 17.12.2018

NFR14: Zuverlässigkeit: Fehlertoleranz
Komponententests decken durch Äquivalenzklassen verschiedene Möglichkeiten ab, in denen die NutzerIn fehlerhafte Eingaben tätigen kann.

NFR15: Instandhaltbarkeit: Modularität
Das entwickelte System benutzt die Design Patterns: Model-View-Controller, Observer und Strategy.

NFR16: Instandhaltbarkeit: Wiederverwendbarkeit
Die Beziehung zwischen den Klassen *Comment* und *Sentence* ist so entworfen, dass Instanzen von *Sentence* auch ohne zugehöriges JIRA-Issue-Kommentar erzeugt werden können. Funktional arbeiten alle Funktionen mit Kommentaren als einzige Quelle.

6.4 Güte des Klassifikators

Zur Berechnung der Güte des Klassifikators wird der Datensatz aus Kapitel 7.3.1, Seite 80 genutzt. Dieser Datensatz enthält 1176 manuell klassifizierte Sätze aus JIRA-Issue-Kommentaren des JIRA LUCENE Projekts. Zur Evaluation wurden diese Sätze heruntergeladen und in einer lokalen MEKA Instanz evaluiert. Tabelle 6.2 zeigt die Ergebnisse der einzelnen Wissenstypen.

Tabelle 6.2: Evaluationsmetriken des feingranularen Klassifikators

Alternative			Issue			Decision			Pro			Con		
P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
0.37	0.57	0.24	0.15	0.57	0.45	0.49	0.62	0.41	0.31	0.71	0.41	0.09	0.58	0.15

Die Metriken zeigen schlechtere Werte im Vergleich zu den Testmetriken in Tabelle 5.4 Seite 39. Ein Grund ist der Kontext der Sätze in Trainingsdaten und Evaluationsdaten. Der Trainingsdatendatensatz ist manuell ohne den Kontext eines Entscheidungsproblems klassifiziert. Für die manuelle Klassifikation eines Satzes wird lediglich dieser Satz betrachtet. Ein Satz im Trainingsdatensatz stellt zudem nur einen grammatikalischen Satz der englischen Sprache dar. Die Sätze des Evaluationsdatensatz wurden im Kontext eines konkreten Entscheidungsproblems klassifiziert. Zudem kann ein Satz im Evaluationsdatensatz aus mehreren grammatikalischen Sätzen bestehen. Absatz 8.2 Seite 94 diskutiert Möglichkeiten zur Verbesserung des Klassifikators.

6.5 Auflistung der Testfälle

Tabelle 6.3 zeigt die Testfälle der Komponententests, Tabelle 6.4 zeigt die Testfälle der Integrationstests und Tabelle 6.5 zeigt die Testfälle der Systemtests.

Tabelle 6.3: Komponententestfälle

SF	ÄQ	id	Test Objekt	Testdaten	Erwartetes Ergebnis	Ergebnis
SF2	Gültig	Kt.1	BinaryPrediction	gefüllte Liste	klassifizierte Liste	✓
		Kt.2	BinaryPrediction	klassifizierte Liste	unveränderte Liste	✓
		Kt.3	BinaryPrediction	markierte Liste	unveränderte Liste	✓
		Kt.4	FineGrainedPrediction	gefüllte Liste	klassifizierte Liste	✓
		Kt.5	FineGrainedPrediction	klassifizierte Liste	unveränderte Liste	✗→ ✓
		Kt.6	FineGrainedPrediction	markierte Liste	unveränderte Liste	✓
	Ungültig	Kt.7	BinaryPrediction	leere List	leere Liste	✓
		Kt.8	BinaryPrediction	null Liste	leere Liste	✓
		Kt.9	FineGrainedPrediction	leere Liste	leere Liste	✓
		Kt.10	FineGrainedPrediction	null Liste	leere Liste	✓
SF3	Gültig	Kt.11	CommentSplitter	Sätze	Getrennte Sätze	✓
		Kt.12	CommentSplitter	Sätze	Korrekter Wissenstyp	✗→ ✓
		Kt.13	CommentSplitter	Zitate	Getrennte Sätze	✓
		Kt.14	CommentSplitter	Noformat	Getrennte Sätze	✓
		Kt.15	CommentSplitter	Quellcode	Getrennte Sätze	✓
		Kt.16	CommentSplitter	Tags	Getrennte Sätze	✓
		Kt.17	CommentSplitter	Tags	Korrekter Wissenstyp	✗→ ✓
		Kt.18	CommentSplitter	Icons	Getrennte Sätze	✓
		Kt.19	CommentSplitter	Icons	Korrekter Wissenstyp	✓
	Ungültig	Kt.20	CommentSplitter	Zitate	Kompletter Satz	✓
		Kt.21	CommentSplitter	Noformat	Kompletter Satz	✓
		Kt.22	CommentSplitter	Quellcode	Kompletter Satz	✓
		Kt.23	CommentSplitter	Tags	Kompletter Satz	✗→ ✓
		Kt.24	CommentSplitter	Icons	Kompletter Satz	✓
SF6	Gültig	Kt.25	KR[a].changeKT[b]	id, type	neuer Type	✓
		Kt.26	KR.editText	Id, text	korrektes Kommentar	✓
		Kt.27	KR.editText	Id, text	korrekte AO Tabelle	✓
		Kt.28	KR.irrelevant	Id	Kommentar ohne Tags	✓
		Kt.29	KR.irrelevant	Id	korrekte AO Tabelle	✓
	Ungültig	Kt.30	KR.changeKT	null Id, type	keine Änderung	✓
		Kt.31	KR.changeKT	Id, null	keine Änderung	✓
		Kt.32	KR.editText	null, text	keine Änderung	✓
		Kt.33	KR.editText	Id, null	keine Änderung	✓
		Kt.34	KR.irrelevant	null	keine Änderung	✓
SF7	Gültig	Kt.35	KR.insert	Projekt, Link	valider Link	✓
		Kt.36	KR.delete	Projekt, Link	gelöschter Link	✓
	Ungültig	Kt.37	KR.insert	null, Link	kein Link erzeugt	✓
		Kt.38	KR.insert	Projekt, null	kein Link erzeugt	✓
		Kt.39	KR.insert	null, null	kein Link erzeugt	✓
		Kt.40	KR.delete	null, Link	kein Link gelöscht	✓
		Kt.41	KR.delete	Projekt, null	kein Link gelöscht	✓
		Kt.42	KR.delete	null, null	kein Link gelöscht	✓
		Kt.43	KR.insert	Projekt, inv. Link	kein Link erzeugt	✓
		Kt.44	KR.insert	inv. Projekt, Link	kein Link erzeugt	✗→ ✓
SF8	Gültig	Kt.45	KR.cDKE[c]	Dke[d]	neues Kommentar	✓
	Ungültig	Kt.46	KR.cDKE	null	kein Kommentar erzeugt	✓
		Kt.47	KR.cDKE	Dke ohne Id	kein Kommentar erzeugt	✓
		Kt.48	KR.cDKE	Dke ohne Beschreibung	kein Kommentar erzeugt	✓
SF9	Gültig	Kt.49	KR.changeLoc[e]	Id	valider Issue	✓
	Ungültig	Kt.50	KR.changeLoc	null	kein Issue erzeugt	✓
		Kt.51	KR.changeLoc	- 1	kein Issue erzeugt	✗→ ✓

[a]KR: KnowledgeRest
[b]KT: KnowledgeType
[c]cDKE: createDecisionKnowledgeElement
[d]dke: DecisionKnowledgeElement
[e]changeLoc: ChangeDocumentationLocation

Tabelle 6.4: Integrationstestfälle

Äquivalenzklasse	Id	Test Objekt	Testdaten	Erwartetes Ergebnis	Ergebnis
Gültig	It.52	TVCD[a]	JIRA-Issue	Liste mit 2 Elementen ohne irrelevantem Satz	✓
	It.53	TVCD	JIRA-Issue	Liste mit 1 Element	✓
	It.54	TVDX[b]	JIRA-Issue	Liste mit 3 Elementen und relevantem Satz	✓
	It.55	TVDX	JIRA-Issue ohne Links	Liste mit 1 Element	✓
Ungültig	It.56	TVCD	null	Leere Liste	✓
	It.57	TVDX	null	Leere Liste	✓
Gültig	It.58	Treant	JIRA-Issue	Baum mit 3 Elementen	✓
	It.59	Treant	JIRA-Issue ohne Links	Baum mit 1 Element	✓
Ungültig	It.60	Treant	null Projekt	Leerer Baum	✓
	It.61	Treant	leeres Projekt	Leerer Baum	✓
	It.62	Treant	null Issue	Leerer Baum	✓

[a]TVCD: TreeViewer ConDec
[b]TVDX: TreeViewer DecXtract

Tabelle 6.5: Systemtestfälle

	Id	Vorbedingung	Schritte	Erwartetes Ergebnis	Ergebnis
WS1.1	St.63	DecXtract deaktiviert	1. Aktiviere DecXtract	DecXtract aktiviert	✓
	St.64	DecXtract deaktiviert	1. Aktiviere DecXtract 2. Lade Seite neu	DecXtract aktiviert	✓
	St.65	DecXtract aktiviert	1. Deaktiviere DecXtract	DecXtract deaktiviert	✓
	St.66	DecXtract aktiviert	1. Deaktiviere DecXtract 2. Lade Seite neu	DecXtract deaktiviert	✗→ ✓
WS1.3	St.67	DKV geöffnet. Entscheidungselement 1 enthält Kommentare	1. Wähle Entscheidungstyp 2. Klick auf Entscheidungselement 1	Graph mit Entscheidungswissen aus gewähltem JIRA-Issue-Kommentaren wird gezeigt	✓
	St.68	DKV geöffnet.	1. Wähle Entscheidungstyp 2. Klick auf EEJIK[a]	Graph mit Entscheidungswissen des zugehörigen JIRA-Issues wird gezeigt	✓
	St.69	DKV geöffnet.	1. Wähle EEJIK 2. Ziehe EEJIK auf EEJI[b]	EEJIK ist Kind des EEJI	✓
	St.70	DKV geöffnet.	1. Wähle EEJI 2. Ziehe EEJI auf EEJIK	EEJI ist Kind des EEJIK	✓
WS1.4.2	St.71	Kommentar vorhanden	1. Gehe zu DKETP[c]	Klassifiziertes Entscheidungswissen angezeigt	✓
	St.72	Kein Kommentar vorhanden	1. Gehe zu DKETP	Leere Ansicht	✓
	St.73	Manuell klassifiziertes Kommentar vorhanden	1. Gehe zu DKETP	Klassifiziertes Entscheidungswissen angezeigt	✓
	St.74	Mit Icon klassifiziertes Kommentar vorhanden	1. Gehe zu DKETP	Klassifiziertes Entscheidungswissen angezeigt	✗→ ✓
	St.75	Klassifiziertes Kommentar vorhanden	1. Gehe zu DKETP 2. Wähle Entscheidungstyp ab	Entscheidungswissen ausgeblendet	✗→ ✓
	St.76	JIRA-Issue geöffnet	1. Klicke auf „Decision Knowledge Extraction“ Panel	DKETP geöffnet	✓
	St.77	DKETP geöffnet	1. Rechtsklick auf Element 2. Wähle neuen Entscheidungstyp	Entscheidungselement besitzt neuen Entscheidungstyp	✓
	St.78	DKETP geöffnet, MKE[d] vorhanden	1. Rechtsklick auf Element 2. Wähle neuen Entscheidungstyp	Entscheidungselement besitzt neuen Entscheidungstyp, auch in Kommentaransicht	✗→ ✓
	St.79	DKETP geöffnet	1. Rechtsklick auf Element 2. Wähle „Edit Sentence“ 3. Ändere Text beliebig	Entscheidungselement besitzt neuen Text, auch in Kommentaransicht	✓
	St.80	DKETP geöffnet, MKE vorhanden	1. Rechtsklick auf Element 2. Wähle „Edit Sentence“ 3. Ändere Text beliebig	Entscheidungselement besitzt neuen Text, auch in Kommentaransicht	✗→ ✓
	St.81	DKETP geöffnet	1. Rechtsklick auf Element 2. Klicke „Set Irrelevant“	Entsprechendes Entscheidungselement ohne Entscheidungstyp	✓
WS1.4.2	St.82	DKETP geöffnet	1. Wähle Element 1 2. Ziehe auf Element 2	Entscheidungselement 1 besitzt Entscheidungselement 2 als Kind	✓
	St.83	DKETP geöffnet	1. Wähle Element 1 2. Ziehe auf Element 2 3. Wähle Element 2 4. Ziehe auf Element 3	Entscheidungselement 3 besitzt Entscheidungselement 2 und 1 als Kind	✓
	St.84	DKETP geöffnet	1. Wähle Kindelement 1 2. Ziehe JIRA-Issue	Entscheidungselement 1 ist Kindelement von JIRA-Issue	✓
WS1.5	St.85	Report geöffnet		Alle Kennzahlen werden angezeigt	✓
	St.86	Report geöffnet	1. Klicke auf jede Grafik	Dialog mit Liste der Kennzahlen öffnet sich.	✓
	St.87	Report geöffnet	1. Bewege Mauszeiger über Kennzahlen	Tooltip erscheint und zeigt genaue Informationen	✓
	St.88	Report geöffnet	1. Rechtsklick auf Grafik 2. Wähle „Bild speichern unter“	Grafik wird als Bild gespeichert	✓

[a]EEJIK: Entscheidungselement aus JIRA-Issue-Kommentaren
[b]EEJI: Entscheidungselement als Jira-Issue
[c]DKETP: Decision Knowledge Extraction Tab Panel
[d]MKE: Manuell klassifiziertes Entscheidungselement

7 Evaluation

Dieses Kapitel zeigt die Durchführung und die Ergebnisse der Evaluation von DecXtract. Abschnitt 7.1 definiert die Ziele der Evaluation und beschreibt die eingesetzten Methoden. Abschnitt 7.2 untersucht die Eignung von DecXtract, um Entscheidungswissen in JIRA-Issue-Kommentaren zu dokumentieren. Abschnitt 7.3 beschreibt die Untersuchung des Nutzens von Entscheidungswissen zum Verständnis von Quellcodeänderungen. Abschnitt 7.4 fasst die Ergebnisse der Evaluation zusammen.

7.1 Ziele und Durchführung der Evaluation

Die Literaturrecherche (Kapitel 3 Seite 13) beantwortet die Frage, welches Wissen und insbesondere welches Entscheidungswissen EntwicklerInnen zum Verständnis von Quellcodeänderungen benötigen. Anhand dieser Fragen werden zwei Evaluationsfrage (Ef) für DecXtract abgeleitet.

Ef.1: Ist DecXtract für die Dokumentation und Nutzung von Entscheidungswissen geeignet?

Das Technology Acceptance Model (TAM) beschreibt drei Variablen zur Akzeptanz eines Softwaresystems [24]. Diese sind einfache Benutzbarkeit eines Softwaresystems, Nützlichkeit eines Softwaresystems für eine Aufgabe und Intention zur Wiederverwendung. Diese Variablen messen die Akzeptanz der NutzerInnen für ein Softwaresystem. Die Eignung von DecXtract zur Dokumentation in JIRA-Issue-Kommentaren wird durch eine Umfrage mit EntwicklerInnen festgestellt. Der Fragebogen für diese Umfrage ist entsprechend der im TAM definierten Variablen entworfen. Die Menge der befragten Personen setzt sich aus Studierenden und EntwicklerInnen des ConDec-Projektes zusammen.

Ef.2: Reicht das Wissen aus JIRA-Issue-Kommentaren um Quellcodeänderungen zu verstehen?

Zur Beantwortung dieser Frage wird ein Evaluationsdatensatzerzeugt, der Quellcodeänderungen mit einem JIRA-Issue verknüpft. Ein Projekt, das die definierten Anforderungen an den Evaluationsdatensatzerfüllt, ist Apache LUCENE. Entscheidungswissen der JIRA-Issue-Kommentare von LUCENE wird mithilfe von DecXtract explizit angelegt und verlinkt. Anhand der gefundenen Fragen von EntwicklerInnen zu Quellcodeänderungen (Tabelle 3.10 Seite 19) wird ein Protokoll erstellt, das alle JIRA-Issues des Evaluationsdatensatz prüft. Das Protokoll wird bei der Betrachtung der Quellcodeänderung mit dem verfügbaren Entscheidungswissen jedes JIRA-Issues ausgefüllt. Anhand der identifizierten Werte lässt sich das Verständnis von Quellcodeänderungen bewerten.

7.2 Eignung von DecXtract

Das Ziel der Evaluationsfrage Ef.1 ist die Verifizierung der Eignung von DecXtract zur Dokumentation von Entscheidungswissen in JIRA-Issue-Kommentaren. Um diese Frage zu beantworten, wird ein Fragebogen nach den drei Variablen des TAM entworfen [24]. Dieser Fragebogen soll in einer Umfrage die Akzeptanz der NutzerInnen von DecXtract feststellen. Die drei Variablen werden folgendermaßen eingesetzt:

1. *Ease of Use* – Die einfache Benutzbarkeit beschreibt, inwieweit einE NutzerIn DecXtract mühelos benutzen kann.
2. *Usefullness* – Die Nützlichkeit beschreibt die subjektive Wahrnehmung eines Nutzers/einer Nutzerin darüber, inwieweit DecXtract sie bei ihrer Arbeit unterstützen kann.
3. *Intention* – Die Absicht beschreibt die voraussichtliche Nutzung von DecXtract der NutzerIn in zukünftigen Projekten.

Der Fragebogen besteht aus den vier Kategorien in Tabelle 7.1. Jede Kategorie stellt drei Fragen zu je einer Variable des TAM. Vor der Beantwortung einer Kategorie bearbeiten die TeilnehmerInnen der Umfrage eine Aufgabe, um sich mit dem Thema der Kategorie vertraut zu machen. Für Kategorie Ke.1 und Ke.2 dokumentieren die TeilnehmerInnen ein eigenes Entscheidungsproblem mit der Hilfe von DecXtract. Für Kategorie Ke.3 diskutieren die TeilnehmerInnen gemeinsam ein Entscheidungsproblem in JIRA-Issue-Kommentaren. Für Kategorie Ke.4 betrachten die TeilnehmerInnen ihr jüngst dokumentiertes Entscheidungswissen in den verfügbaren Ansichten von ConDec.

Tabelle 7.1: Kategorien des Fragebogens zur Feststellung der Eignung von DecXtract

Kat.	Komponente	Aufgabe
Ke.1	Manuelle Klassifikation	Durchführung der manuellen Klassifikation von Entscheidungswissen in JIRA-Issue-Kommentaren zu einem Entscheidungsproblem
Ke.2	Automatische Klassifikation	Durchführung der automatischen Klassifikation von Entscheidungswissen in JIRA-Issue-Kommentaren zu einem Entscheidungsproblem
Ke.3	Einsatz im Team	Gemeinsame Diskussion und Lösungsfindung eines Entscheidungsproblems
Ke.4	Nutzen der Ansichten	1) WS1.3 Decision Knowledge View (Abbildung 2.3) 2) WS1.4.1 Issue Module (Abbildung 5.26) 3) WS1.4.2 Issue Comments (Abbildung 5.27)

Tabelle 7.2 beschreibt den erstellten Fragebogen. Die TeilnehmerInnen der Umfrage beantworten die Fragen anhand einer fünfstufigen Likert-Skala mit den Werten *starker Widerspruch* bis *starke Zustimmung*. Des Weiteren sollen die TeilnehmerInnen ihre Antworten im Freitextbereich begründen und ausführen, was ihnen gut oder schlecht gefallen hat.

Tabelle 7.2: Fragebogen zur Eignung von DecXtract nach TAM [24]

Nr.	Kat.	Variable	Frage
F.1	Ke.1	Benutzbarkeit	Es ist einfach Entscheidungswissen in JIRA-Issue-Kommentaren manuell zu klassifizieren.
F.2	Ke.1	Nützlichkeit	Es nutzt der Reflexion und der besseren/verständlicheren Dokumentation Entscheidungswissen in JIRA-Issue-Kommentaren manuell zu klassifizieren.
F.3	Ke.1	Absicht	Ich werde auch in zukünftigen Projekten Entscheidungswissen manuell in JIRA-Issue-Kommentaren klassifizieren.
F.4	Ke.2	Benutzbarkeit	Es ist einfach Entscheidungswissen in JIRA-Issue-Kommentaren automatisch zu klassifizieren.
F.5	Ke.2	Nützlichkeit	Es nutzt der Reflexion und der besseren bzw. verständlicheren Dokumentation von Entscheidungswissen in JIRA-Issue-Kommentaren automatisch zu klassifizieren.
F.6	Ke.2	Absicht	Ich werde auch in zukünftigen Projekten Entscheidungswissen automatisch in JIRA-Issue-Kommentaren klassifizieren.
F.7	Ke.3	Benutzbarkeit	Es ist einfach JIRA-Issue Kommentare zur Entscheidungsdokumentation im Team zu nutzen.
F.8	Ke.3	Nützlichkeit	Es nutzt der kollaborativen Entscheidungsfindung JIRA-Issue Kommentare zur Entscheidungsdokumentation im Team zu nutzen.
F.9	Ke.3	Absicht	Ich werde mich auch in zukünftigen Projekten an der Entscheidungsfindung in JIRA-Issue-Kommentaren in anderen Projekten beteiligen.
F.10	Ke.4	Nützlichkeit	Es ist nützlich Entscheidungswissen in JIRA-Issue-Kommentaren in Ansicht WS1.3 Decision Knowledge View zu betrachten.
F.11	Ke.4	Nützlichkeit	Es ist nützlich Entscheidungswissen in JIRA-Issue-Kommentaren in Ansicht WS1.4.1 Issue Module zu betrachten.
F.12	Ke.4	Nützlichkeit	Es ist nützlich Entscheidungswissen in JIRA-Issue-Kommentaren in Ansicht WS1.4.2 und WS1.4.3 Issue Comments zu betrachten.

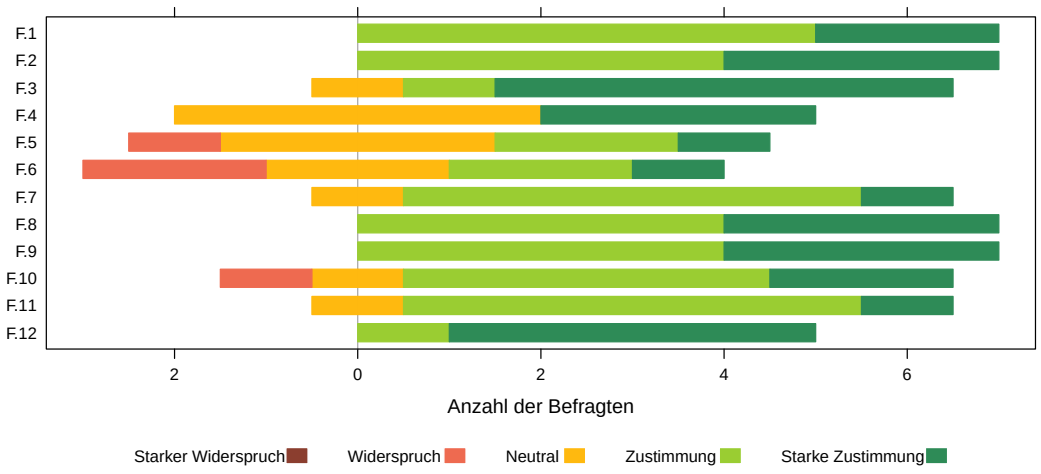


Abbildung 7.1: Ergebnisse der Befragung von sieben Studierenden

Sieben Studierende haben an der Befragung teilgenommen. Die erfahrenste Nutzerin nutzt ConDec seit 18 Monaten, die unerfahrenste nutzt ConDec seit einem Monat. Im Mittelwert nutzen die TeilnehmerInnen ConDec seit sechs Monaten.

Abbildung 7.1 zeigt die Ergebnisse der Befragung. Tabelle 7.3 zeigt die häufigsten Antworten im Freitextbereich. Die Rückmeldungen fallen durchaus positiv aus. Die TeilnehmerInnen halten DecXtract zur Dokumentation von Entscheidungswissen für geeignet.

Die TeilnehmerInnen stimmen zu, dass die manuelle Klassifikation einfach zu benutzen ist, und stimmen der Nützlichkeit von DecXtract zur Entscheidungsdokumentation zu. Sie bevorzugen die manuelle Klassifizierung gegenüber der automatischen Klassifizierung. Nur drei TeilnehmerInnen stimmen zu, dass sie die automatische Klassifizierung auch in zukünftigen Projekten nutzen. Im Freitextbereich wird vermerkt, dass der Klassifikator eine schlechte Präzision aufweist, wodurch Entscheidungselemente manuell neu klassifiziert werden müssen. Eine TeilnehmerIn beschreibt diesen Umstand als zeitaufwändig, sieht die nötige Betrachtung des Entscheidungselements zur Bestimmung des tatsächlichen Wissenstyps aber als positiv zur Reflexion der Dokumentation. Alle sieben TeilnehmerInnen stimmen zu, dass die manuelle Klassifikation einfach zu benutzen ist und die Dokumentation von Entscheidungswissen verbessert.

Die Diskussion eines Entscheidungsproblems im Team zeigt, dass alle TeilnehmerInnen die Entscheidungsdokumentation in JIRA-Issue-Kommentaren auch in zukünftigen Projekten benutzen würden. Dabei loben sie die automatische Verlinkung von neuen Entscheidungselementen sowie eine schnelle Übersicht des aktuellen Status der Entscheidungsprobleme der Teammitglieder.

Die Betrachtung der Visualisierungen von ConDec zeigt, dass die TeilnehmerInnen die farbliche Markierung im Kommentarbereich eines JIRA-Issues gut finden. Die TeilnehmerInnen kritisieren allerdings die lange Ladezeit in Ansicht WS1.3. Zwei Befragte bemängeln Redundanzen zwischen WS1.4.1, WS1.4.3, zwei weitere loben jedoch den Vorteil der Übersichtlichkeit bei großen Entscheidungsproblemen von Ansicht WS1.4.2. Die TeilnehmerInnen wünschen sich in den Visualisierungen von ConDec noch einen Bezug auf den/die AutorIn eines Entscheidungselements.

Tabelle 7.4 beschreibt Kritikpunkte der TeilnehmerInnen mit Begründung des Autors.

Tabelle 7.3: Häufige Aussagen der TeilnehmerInnen

Kat.	# Antworten	Feedback
Ke.1	6	Gute Dokumentationsmöglichkeit durch einfache Markierung im Text
	4	Reflexion der eigenen Aussage wird unterstützt
	2	Schnelle Dokumentationsmöglichkeit von Entscheidungswissen
Ke.2	5	Einfach nutzbar
	4	Klassifizierung ungenau, manuelle Korrektur nötig
	2	Zwingt Autor zur Reflexion
Ke.3	5	Gute Übersicht zu aktuellen Alternativen
	2	Gute Übersicht zur aktuellen Arbeit der KollegInnen
	2	AutorIn des Entscheidungselements in der Visualisierung
Ke.4	4	WS1.3 hat lange Ladezeiten
	3	WS1.4.1 wird schnell unübersichtlich bei großen Bäumen
	2	WS1.4.2 Bearbeitung nicht klassifizierter Kommentare gut möglich

Tabelle 7.4: Kritikpunkte der befragten Personen mit Begründung des Autors

Kat.	Aussage der TeilnehmerInnen	Begründung des Autors
Ke.1	Die farbliche Hinterlegung im Kommentarbereich ist sehr angenehm, es wäre gut auf diesen Einträgen noch ein Kontextmenü zur Verwaltung der Sätze zu haben.	Dieses ist technisch aufwendig umzusetzen. Dennoch scheint ein Bedarf für solch eine Funktionalität zu existieren. Diese Funktionalität wurde nachträglich in Systemfunktion 4 integriert.
Ke.2	Nach der automatischen Klassifikation ist es aufwändig den Wissenstyp zu ändern, insbesondere bei großen Kommentaren mit vielen Sätzen, da jedes mal die Ansicht neu geladen wird.	Neuladen ist notwendig, um dem/der NutzerIn Rückmeldung zur Verlinkung und Klassifikation zu geben. Dennoch ist dieser Aufwand eine Einschränkung der Nutzbarkeit.
Ke.3	Wenn einE andereR NutzerIn einen Kommentar verfasst, muss ich meine Ansicht eigenständig neu laden.	Dies ist ein Multi-User-JIRA-Problem. Als beteiligteR NutzerIn an einer Diskussion erhält man Benachrichtigungen per Mail.
Ke.4	Bei vielen Elementen brauchte die Seite lange, um zu laden, hier wäre ein Ladebalken angebracht, damit nicht der Eindruck entsteht, die Seite sei eingefroren.	Das ist ein bekanntes Problem und auf den Aufbau der vielen Baumstrukturen zurück zu führen. Ein Work Item für einen Ladebalken wurde erstellt.

Gültigkeit und Einflussfaktoren der Studie

Nach Runeson et al. müssen die folgenden vier Gültigkeitskriterien für eine Studie erfüllt sein [28]. Die interne Gültigkeit untersucht, inwieweit die untersuchten Fragen mit anderen Variablen korrelieren. Eine Variable, die bei der Beantwortung der Fragen eine entscheidende Rolle spielen könnte, ist die Erfahrung der Studierenden. Um dies auszugleichen stellte der Umfragenleiter die Funktionalität von ConDec und DecXtract vor jeder Aufgabe vor. Des Weiteren kennen die befragten TeilnehmerInnen DecXtract und können eigene bereits vorgeprägte Meinung besitzen.

Die externe Gültigkeit untersucht, inwieweit die gefundenen Ergebnisse verallgemeinert werden können. Ein Kriterium, das die externe Gültigkeit einschränken könnte, ist, dass das eingesetzte JIRA-Projekt zur Umfrage einen kleineren Umfang haben könnte als industrielle Projekte, bezogen auf die Anzahl der Entwicklungsartefakte und Anzahl der Projektbeteiligten (vgl. Abschnitt 2.5.1). Dieser Faktor kann die Variable *Usefulness* beeinflussen.

Die Konstruktionsgültigkeit untersucht, in wie weit die beobachteten Ergebnisse wirklich die Eignung von DecXtract beschreiben, d.h. die Evaluationsfrage beantworten können. Um die Konstruktionsgültigkeit zu gewährleisten, basiert der entworfene Fragebogen auf dem wissenschaftlich anerkannten TAM.

Die Zuverlässigkeit beschreibt, inwieweit die Ergebnisse dieser Evaluation von den durchführenden Autoren abhängen. Das Ergebnis könnte dadurch beeinflusst sein, dass die Umfrage zur Feststellung der Eignung von DecXtract von seinem Entwickler durchgeführt wurde, jedoch wurden die Antworten so objektiv wie möglich ausgewertet.

7.3 Entscheidungswissen in JIRA-Issue-Kommentaren zum Verständnis von Quellcodeänderungen im LUCENE Projekt

Dieser Abschnitt beschreibt die Frage, ob DecXtract einer EntwicklerIn helfen kann, Quellcodeänderungen besser zu verstehen. Abschnitt 7.3.1 beschreibt Anforderungen und Aufbau eines Evaluationsdatensatzes. Abschnitt 7.3.2 zeigt ein JIRA-Issue des Evaluationsdatensatz mit Quellcodeänderung. Abschnitt 7.3.3 betrachtet Quellcodeänderungen im Evaluationsdatensatzmit dem Entscheidungswissen in JIRA-Issue-Kommentaren. Abschnitt 7.3.4 interpretiert die Ergebnisse.

7.3.1 Aufbau des Evaluationsdatensatzes

Zur Beantwortung der Evaluationsfragen wird ein JIRA-Projekt als Evaluationsdatensatz benötigt. Tabelle 7.5 beschreibt die Anforderungen zur Auswahl eines geeigneten JIRA-Projektes.

Tabelle 7.5: Anforderungen an den Evaluationsdatensatz

Anforderung
JIRA-Projekt ist öffentlich zugänglich.
JIRA-Projekt dokumentiert die Entwicklungsartefakte eines Open-Source-Projektes.
Das Projekt wird regelmäßig weiterentwickelt.
Der Quellcode ist in einer gängigen Programmiersprache entwickelt (Java, C, ...).
Der Quellcode ist in einem GIT-Repository versioniert.
Die Commits sind mit JIRA-Issues über die Angabe der JIRA-Issue-ID verlinkt.

Alle Anforderungen treffen auf das JIRA-Projekt des Apache LUCENE Projektes (Abschnitt 2.5 Seite 10) zu. Die Nutzung dieses Projekts zur Evaluation bietet des Weiteren die Möglichkeit den Klassifikator auf dem Projekt der Trainingsdaten zu evaluieren (Abschnitt 6.4 Seite 71).

Aus dem LUCENE JIRA-Projekt wurden zufällig 100 JIRA-Issues und deren verlinkte JIRA-Issues zur Evaluation ausgewählt, sodass der Ausgangsdatensatz 131 JIRA-Issues umfasst. Zur Evaluation müssen diese JIRA-Issues in ein lokales JIRA-Projekt importiert werden. Eine Möglichkeit ist die manuelle Übertragung durch Kopieren und Einfügen aller JIRA-Issues und Kommentare. Diese Möglichkeit ist sehr robust, da alle speziellen Formatierungen und Hyperlinks übernommen und geprüft werden können. Diese Methode ist sehr zeitaufwendig.

Eine weitere Möglichkeit ist der Export eines JIRA-Issues aus dem LUCENE-Projekt im XML oder JSON Format. Über eine JIRA-REST-Schnittstelle können so neue JIRA-Issues im lokalen LUCENE Projekt erzeugt werden. Allerdings ist diese Methode sehr anfällig gegen Sonderzeichen, spezielle Formatierungen und Hyperlinks. Zudem muss jedes Kommentar mit einem JIRA-REST-Aufruf angelegt werden.

Aufgrund der Robustheit gegen Sonderzeichen und spezielle Formatierungen wurde entschieden, alle JIRA-Issues und Kommentare per Hand in ein lokales[1] JIRA-LUCENE Projekt zu kopieren.

[1] Lokales LUCENE JIRA-Projekt: Das JIRA-Projekt ist auf dem CURES-Server der Universität Heidelberg gespeichert. Link: https://cures.ifi.uni-heidelberg.de/jira/projects/LUCENE/issues Zuletzt aufgerufen am 17.12.2018

7.3.2 Beispiel eines Entscheidungsproblems in LUCENE

Dieser Abschnitt beschreibt ein JIRA-Issue des LUCENE Projekts mit zugehöriger Quellcodeänderung. Abbildung 7.2 stellt den Entscheidungsbaum zum JIRA-Issue LUCENE-2387 dar. Das Entscheidungsproblem ist die Frage, wie *Memory Leaks* in der Klasse *IndexWriter* verhindert werden können. Die/der AutorIn der ersten Alternative schlägt eine Änderung vor. Im LUCENE Projekt scheint es üblich, eine Diff-Datei[2] hochzuladen und dies mit dem Kommentar „Patch“ zu signalisieren. Das nachfolgende Entscheidungselement beschreibt die Änderung in diesem Patch. Durch das Pro-Argument zu diesem Entscheidungselement beschließen die beteiligten EntwicklerInnen diesen Änderungsvorschlag zu akzeptieren. Das Entscheidungselement „29x version of this patch“ gibt an, dass dieser Patch akzeptiert und bereits zum Quellcode hinzugefügt wurde.

Quellcode 7.1 zeigt die Quellcodeänderung zu diesem Entscheidungsproblem. Die EntwicklerIn hat grün hinterlegte Zeilen hinzugefügt und rot hinterlegte Zeilen entfernt. Die Zeile

```
input = null;
```

implementiert die Quellcodeänderung zum dokumentierten Entscheidungswissen. Diese Quellcodeänderung ist konsistent zum Entscheidungswissen der JIRA-Issue Kommentare.

```
--- lucene/dev/trunk/lucene/src/java/org/apache/lucene/index/DocInverterPerField.java
+++ lucene/dev/trunk/lucene/src/java/org/apache/lucene/index/DocInverterPerField.java
@@ -199,6 +199,10 @@
public void close() throws IOException {
-     input.close();
+     if (input != null) {
+       input.close();
+       // LUCENE-2387: don't hold onto Reader after close, so GC can reclaim
+       input = null;
+ }
```

Listing 7.1: Quellcodeänderung in LUCENE-2387

[2]Diff Datei: Ähnlich Quellcode 7.1, Zeigt durchgeführte Änderungen am Quellcode.

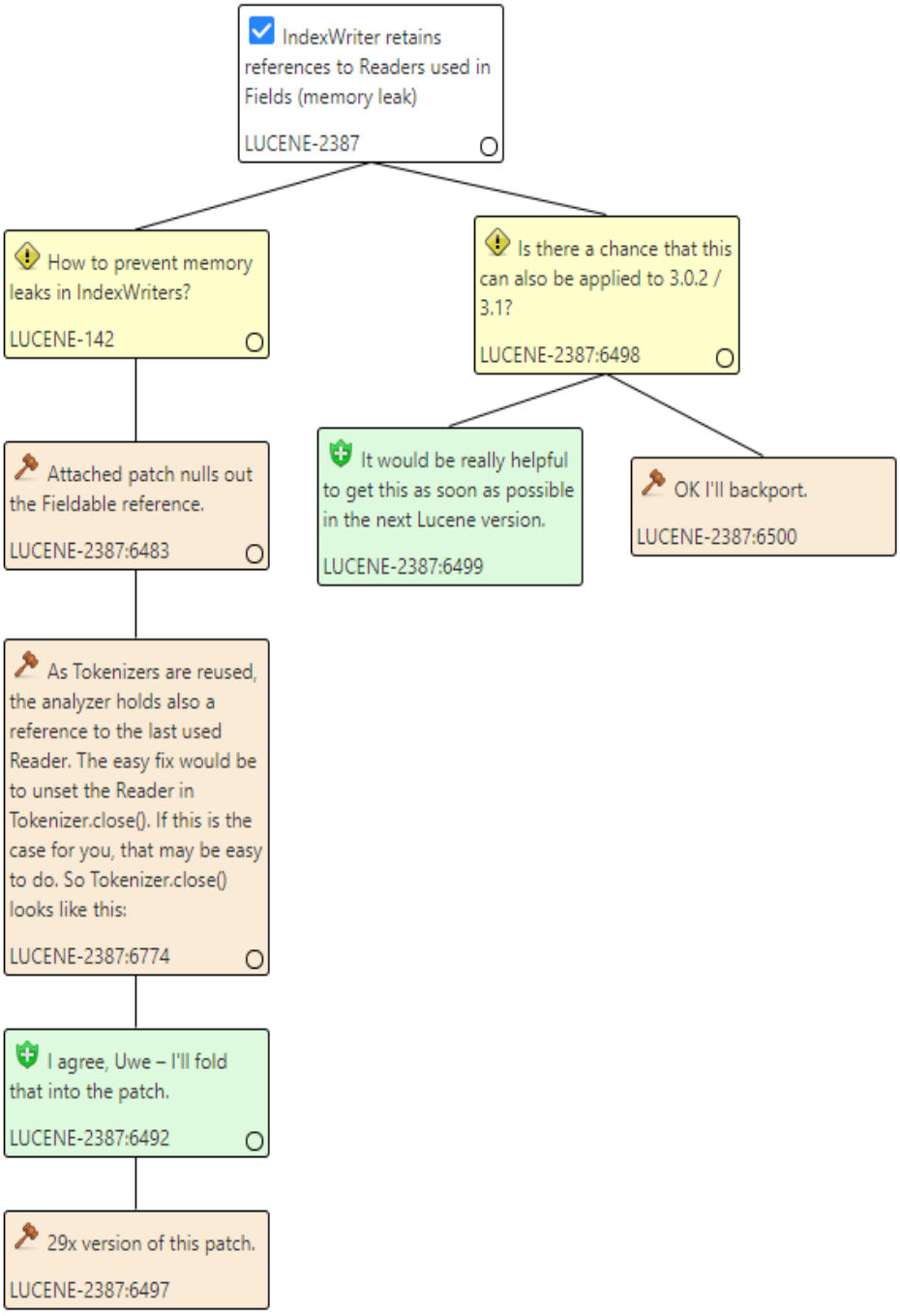


Abbildung 7.2: Entscheidungsproblem zu LUCENE-2387

7.3.3 Planung und Durchführung der Analyse

Um die Evaluationsfrage Ef.2 dieser Arbeit zu beantworten, werden alle JIRA-Issues des Evaluationsdatensatzes analysiert. Die Analyse betrachtet die folgenden fünf Kategorien, abgeleitet aus dem Ergebnis der Literaturrecherche (Tabelle 3.10 Seite 19). Tabelle 7.6 beinhaltet das Protokoll zur Analyse mit 31 Fragen, die zeigen sollen, inwieweit Entscheidungswissen aus JIRA-Issue-Kommentaren EntwicklerInnen beim Verständnis von Quellcodeänderungen unterstützt.

1) Allgemein – Frage F.1 bis F.7
   Diese Kategorie beschreibt die Metadaten des JIRA-Issue. Der JIRA-Issue-Key sorgt für die Nachverfolgbarkeit, um das Protokoll mit dem JIRA-Issue zu verbinden. Der JIRA-Issue-Typ nimmt die Werte *Work Item* oder *Bug* an. Status beschreibt, ob der Sachverhalt des JIRA-Issue gelöst wurde oder nicht. Des Weiteren wird die Anzahl der Commits, Kommentare und gefundenen Entscheidungselemente dokumentiert. Diese Kennzahlen ermöglichen statistische Bewertungen gegen die Verständlichkeit der betrachteten Artefakte.

2) Vollständigkeit – Frage F.8 bis F.16
   Diese Kategorie beschreibt die Vollständigkeit des Entscheidungsproblems eines JIRA-Issue. Für jeden Wissenstyp wird die Anzahl der existierenden Entscheidungselemente gezählt. Zudem wird geprüft, ob sich ein eindeutiges Entscheidungsproblem, Lösung und Alternative finden lassen. Lösungen und Alternativen müssen eindeutige Argumente für oder gegen sie besitzen.

3) Quellcodeänderung– F.17 bis F.20
   Diese Kategorie beschreibt die Quellcodeänderungdes JIRA-Issue. Shihab et al. klassifiziert große Quellcodeänderungen als risikoanfälliger als kleine Quellcodeänderungen [30]. Die Kennzahlen zur Änderung der LOC berechnet das „Git Integration for JIRA“ Plug-In. Der betrachtetet Wert summiert sich aus allen Commits.

4) Konsistenz – Frage F.21 bis F.27
   Diese Kategorie beschreibt die Konsistenz zwischen dokumentiertem Entscheidungswissen und Quellcodeänderung. Dafür muss die implementierte Quellcodeänderungmit dem Entscheidungswissen konsistent sein. Es wird auch betrachtet, ob Entscheidungsprobleme dokumentiert sind, die nicht implementiert wurden. Die Kernänderung beschreibt genau die Quellcodeänderung, die nötig ist, um das Entscheidungsproblem zu lösen. Änderungsauswirkungen[3] resultieren aus der Kernänderungen und passen Quellcode für die Kernänderung an.

5) Entscheidungsfindungsstrategie – Frage F.28 bis F.31
   Diese Kategorie untersucht nach Hesse et al. die sprachliche Dokumentation der Argumente für Entscheidungen und Alternativen [22]. Die Begründung einer rationalen Argumentation basiert auf Tatsachen und Erfahrungen, wogegen eine nicht-rationale Argumentation eine persönliche Meinung, Erfahrung in der Vergangenheit oder Gefühle nutzt.

[3] Beispiel einer Änderungsauswirkung: Die Kernänderung ändert den Rückgabetyp einer Methode von *String* auf *Integer*. Die Änderungsauswirkungen betreffen alle Methodenaufrufe, die ebenfalls von *String* auf *Integer* geändert werden müssen.

Tabelle 7.6: Analyseprotokoll der Quellcodeänderungen

1) Allgemein		**2) Vollständigkeit**	
F.1	JIRA-Issue Key	F.8	Anzahl Issues [4]
F.2	JIRA-Issue Typ [30]	F.9	Anzahl Entscheidungen [4]
F.3	Status	F.10	Anzahl Alternativen [4]
F.4	Anzahl Commits	F.11	Anzahl Argumente [4]
F.5	Anzahl Kommentare	F.12	Eindeutiges Entscheidungsproblem [31, 32]
F.6	Anzahl Entscheidungselemente	F.13	Eindeutige Lösung [31]
F.7	Hinzugefügte Entscheidungselemente	F.14	Eindeutige Begründung zur Lösung [31, 32]
		F.15	Eindeutige Alternativlösung [32]
		F.16	Eindeutiges Gegenargument [31, 32]
3) Quellcodeänderung		**4) Konsistenz**	
F.17	Anzahl LOC hinzugefügt [30]	F.21	Inkonsistenz im Entscheidungswissen
F.18	Anzahl LOC gelöscht [30]	F.22	Codeänderung konsistent zu Entscheidung
F.19	Anzahl LOC geändert [30]	F.23	Kernänderung ist identifizierbar
F.20	Anzahl Dateien geändert [30]	F.24	Änderungsauswirkung erwartet [33]
		F.25	Änderungsauswirkung identifizierbar? [33]
		F.26	Commit enthält unerwähnte Änderungen
		F.27	Entscheidung nicht umgesetzt
5) Entscheidungsfindungsstrategie			
	Argumente sind rational beschrieben [22]		
F.28	Pro-Argument		
F.29	Con-Argument		
	Argumente sind nicht rational beschrieben [22]		
F.30	Pro-Argument		
F.31	Con-Argument		

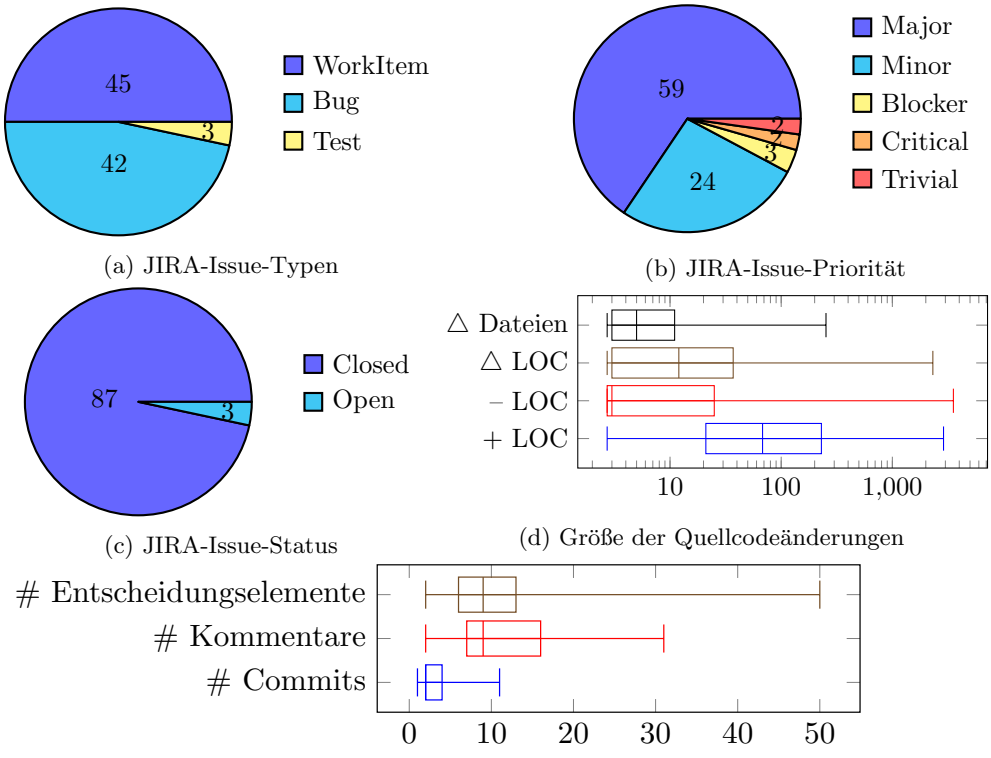
7.3.3.1 Ergebnisse: Allgemein und Quellcodeänderung

Dieser Abschnitt betrachtet die Ergebnisse der Kategorie Allgemein und Quellcodeänderung.

Im lokalen JIRA-Projekt von LUCENE wurde Entscheidungswissen in allen JIRA-Issues mit Hilfe von Systemfunktion 2 und 3 klassifiziert und verlinkt. 31 JIRA-Issues sind für die Evaluation nicht geeignet, da sie keine GIT-Commits besitzen. Abbildung 7.3 zeigt Kennzahlen zu den verbleibenden 90 JIRA-Issues.

Der Datensatz besteht aus 90 JIRA-Issues, mit 45 Work Items, 42 Bugs und 3 Tests. 59 JIRA-Issues besitzen die neutrale Priorität *Major*. 24 JIRA-Issues sind niedriger mit *Minor* priorisiert. Sieben weitere JIRA-Issues sind auf die Prioritäten *Trivial*, *Critical* und *Blocker* aufgeteilt. Lediglich drei JIRA-Issues besitzen den Status *Open*, alle weiteren sind *Closed*. Abbildung 7.3d zeigt die Verteilung der Größe der Quellcodeänderungen. Commits fügen häufig neuen Quellcode hinzu. Meist sind nur wenige (<10) Dateien von einer Quellcodeänderung betroffen.

Abbildung 7.3e zeigt Kennzahlen zu JIRA-Issues. Die Anzahl der Entscheidungselemente in JIRA-Issues liegt im Median bei acht Entscheidungselementen pro JIRA-Issue. Die Anzahl der Kommentare liegt im Median bei neun Kommentaren pro JIRA-Issue. Im Maximum existieren weniger Kommentare als Entscheidungselemente pro JIRA-Issue. Die Anzahl der Commits liegt zwischen einem und elf Commits, der Median liegt bei zwei Commits.



(a) JIRA-Issue-Typen

(b) JIRA-Issue-Priorität

(c) JIRA-Issue-Status

(d) Größe der Quellcodeänderungen

(e) Umfang der JIRA-Issues

Abbildung 7.3: Statistische Analyse des Evaluationsdatensatz im lokalen LUCENE Projekt

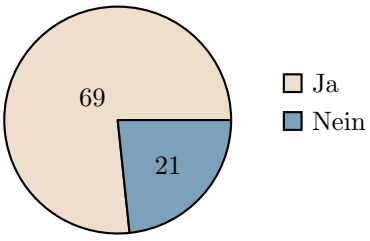
7.3.3.2 Ergebnisse: Vollständigkeit

Die Qualität des Entscheidungswissens wird nach Alkadhi et al. durch die Vollständigkeit des Entscheidungswissens (Rationale Completeness) festgestellt [4]. Rationale Completeness tritt ein, wenn alle Entscheidungsproblem mit Alternativen diskutiert wurden und alle Alternativen mit Argumenten bewertet wurden, zudem ist eine Entscheidung zu identifizieren. Um die Rationale Completeness zu bestimmen, wird jedes Entscheidungsproblem auf vorhandene Entscheidungselemente mit entsprechenden Wissenstypen und deren Verlinkung untersucht. Alle Entscheidungselemente müssen dabei eindeutig erkennbar sein und aus den Kommentaren des untersuchten JIRA-Issues stammen.

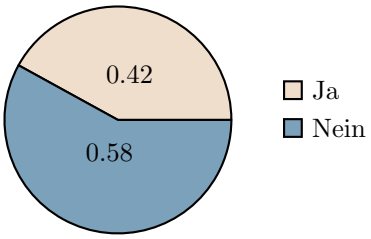
Abbildung 7.4a beschreibt die Existenz der finalen Entscheidung. Die Aussage dieses Entscheidungselements muss mit der Quellcodeänderung konsistent sein (vgl. Abschnitt 7.3.3.3). Abbildung 7.4b zeigt, wie viele dieser Entscheidungen begründet sind.

Abbildung 7.4c beschreibt die Existenz eines alternativen Vorschlags zur Entscheidung eines Entscheidungsproblems. Dies kann sowohl ein Teilaspekt als auch ein Gegenvorschlag sein. Abbildung 7.4d zeigt, wie viele der vorhandenen Alternativen widerlegt sind. Ein Widerspruch spricht eine deutliche Aussage gegen eine Alternative aus.

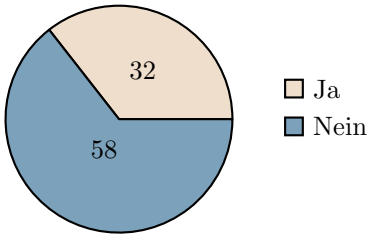
Die ausgewählten JIRA-Issues des LUCENE Projektes zeigen mit 76 % (69) eine akzeptable Dokumentation der umgesetzten Entscheidungen. Diese sind jedoch nur zu 42 % (29) begründet. Alternativen zu Entscheidungen sind nur in 35 % (32) der untersuchten JIRA-Issues dokumentiert. Diese sind nur zu 38 % (12) widerlegt.



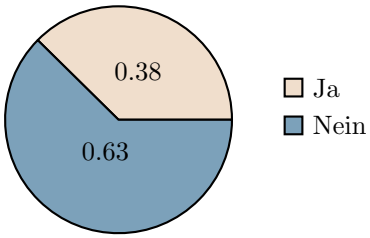
(a) JIRA-Issue enthält eine eindeutige Lösung zum Entscheidungsproblem



(b) Die vorhandene Lösung des Entscheidungsproblems ist eindeutig begründet



(c) JIRA-Issue enthält mindestens eine Alternative zum Entscheidungsproblem



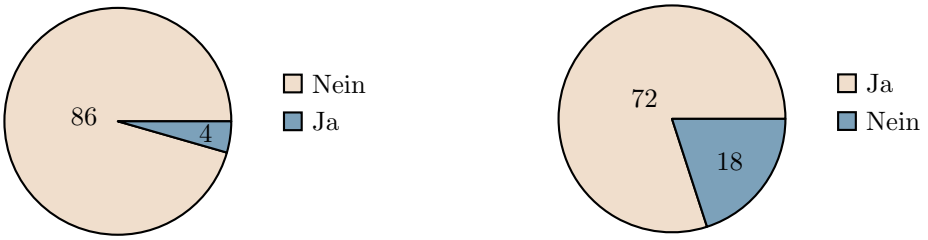
(d) Die vorhandene Alternative ist eindeutig Widerlegt

Abbildung 7.4: Rationale Completeness der untersuchten JIRA-Issues im Lucene Projekt

7.3.3.3 Ergebnisse: Konsistenz und Entscheidungsfindungsstrategie

Abbildung 7.5a zeigt die Ergebnisse zu Frage F.21. Nur vier JIRA-Issues haben Inkonsistenzen innerhalb des dokumentierten Entscheidungswissens. Ein JIRA-Issue besitzt Alternativen, die nicht durch Argumente bewertet sind, die implementierte Alternative besitzt zwei Con-Argumente. Die EntwicklerIn begründet nicht, wieso diese Entscheidung gegenüber der Alternativen implementiert wurde. Die Diskussionen der drei weiteren JIRA-Issues behandeln nicht das zu lösende Entscheidungsproblem. Es lassen sich Entscheidungselemente identifizieren, diese stehen allerdings in keinem Bezug zur Quellcodeänderung.

Abbildung 7.5b zeigt die Ergebnisse zu Frage 22. Von 90 geprüften JIRA-Issues wurde die Konsistenz zwischen Entscheidungswissen und Quellcodeänderung festgestellt. Der Hauptgrund für die fehlende Konsistenz ist nicht dokumentiertes Entscheidungswissen. In vier Fällen kommt es vor, dass Quellcode (Klassen und Methoden) gelöscht wird, dies aber nicht im Entscheidungswissen begründet ist. In sieben Fällen wird nach der Quellcodeänderung eine kleine (< 5 LOC) Korrektur durchgeführt. Diese Korrekturen sind nicht im Entscheidungswissen begründet oder dokumentiert. Diese 18 JIRA-Issues haben im Mittelwert 10 Kommentare und 7.5 Entscheidungselemente. Es ist nicht darauf zurückzuführen, dass die EnwicklerInnen zu wenig Wissen dokumentiert haben. Allerdings ändern diese 18 JIRA-Issues im Mittelwert 141 LOC, fügen 328 LOC hinzu und löschen 285 LOC. Ein Anzeichen für die schlechte Konsistenz kann die Größe der Quellcodeänderung sein.



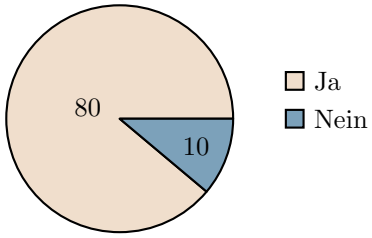
(a) F.21: Inkonsistenzen innerhalb des Entscheidungswissens

(b) F.22: Code Änderung ist konsistent zu Beschreibung in Entscheidungsdokumentation

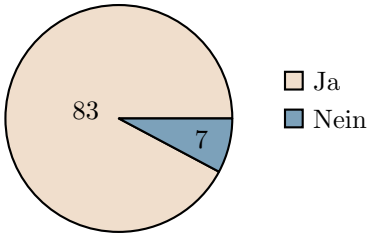
Abbildung 7.5: Konsistenzanalyse ausgewählter JIRA-Issues des LUCENE Projekts I

Abbildung 7.6a beantwortet Frage 23 und zeigt die Anzahl der JIRA-Issues, deren Kernänderung identifizierbar ist. Zehn JIRA-Issues ermöglichen keine Identifizierung der Kernänderung. In acht Fällen ist die gesamte Quellcodeänderung zu groß, um die Kernänderung zu identifizieren. In diesen Fällen sind auch keine oder zu viele Methoden- und Klassennamen im Entscheidungswissen dokumentiert. Im Mittelwert ändern diese zehn JIRA-Issues 310 LOC, fügen 990 LOC hinzu und löschen 206 LOC. Dies ist ein Indiz für die Größe der Quellcodeänderung als Ursache zur schlechten Identifizierbarkeit von Kernänderungen.

Abbildung 7.6b zeigt den Vergleich zwischen Frage 24 und 25. Diese Fragen untersuchen, ob eine Änderungsauswirkung erwartet wird, und ob diese identifizierbar ist. Bei sieben JIRA-Issues stimmen die Antworten nicht überein. In allen sieben Fällen behandelt das JIRA-Issue das Entscheidungsproblem einer neuen Funktionalität (z.B. neue Klasse oder Methode), die als Kernänderung betrachtet wird. Der Aufruf der neuen Klasse oder Methode wird als Änderungsauswirkung betrachtet. Diese ist in keinem der sieben Fälle zu identifizieren. Die LUCENE EntwicklerInnen trennen ihre Aufgaben (hier: Entwicklung einer neuen Klasse und Einsatz einer neuen Klasse) in verschiedenen JIRA-Issues. Daher ist die Änderungsauswirkung in diesem Commit nicht zu identifizieren.



(a) Kernänderung ist identifizierbar

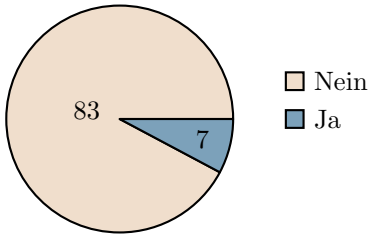


(b) Erwartete Änderungsauswirkung identifizierbar

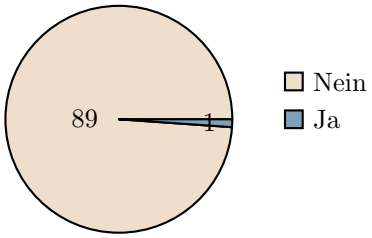
Abbildung 7.6: Konsistenzanalyse ausgewählter JIRA-Issues des LUCENE Projekts II

Abbildung 7.7a zeigt die Antwort auf Frage 26 und gibt an, wie viele Quellcodeänderungen nicht in einem Entscheidungsproblem diskutiert wurden. Dies trifft auf sieben JIRA-Issues zu. Fünf der sieben Fälle beschreiben Bug Fixes, die in der zugehörigen Commit Nachricht beschrieben werden. In zwei Fällen ist die Begründung hinter der Änderung nicht klar.

Abbildung 7.7b zeigt die Antwort auf Frage 27, die Entscheidungsprobleme sucht, die nicht im Quellcode umgesetzt wurden. Nur ein JIRA-Issue konnte gefunden werden, dessen Quellcodeänderung ein Entscheidungsproblem nicht umsetzt.



(a) Änderung durchgeführt die nicht im Entscheidungsproblem beschrieben ist



(b) Entscheidene Entscheidungsprobleme wurden nicht umgesetzt

Abbildung 7.7: Konsistenzanalyse ausgewählter JIRA-Issues des LUCENE Projekts III

Abbildung 7.8 zeigt das Ergebnis der Fragen 28 bis 31 zur Entscheidungsfindungsstrategie. Es ist deutlich, dass Con-Argumente fast ausschließlich rational formuliert sind. Dies belegt die ursprüngliche Annahme, dass EntwicklerInnen in Open-Source-Projekten viele Meinungen austauschen, insbesondere wenn die EntwicklerInnen Gegenargumente begründen.

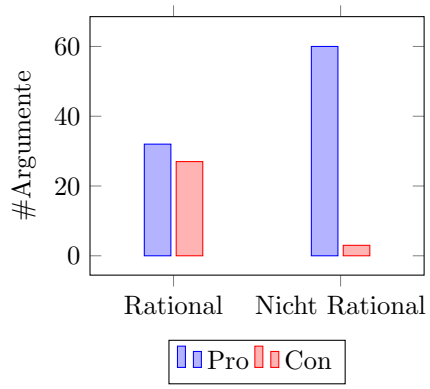


Abbildung 7.8: Ergebnis Entscheidungsfindungsstrategie Frage 28 bis 31

7.3.4 Interpretation korrelierender Attribute

Durch ein R Script wurden die erfassten Daten aus dem Protokollbogen in Tabelle 7.6 analysiert und verglichen. Dabei wird jedes JIRA-Issue als Vektor betrachtet, der die Ergebnisse der Fragen enthält. Die numerische Antwort[4] auf eine Frage wird als Attribut im Vektor gespeichert.

Das R Script nutzt die Methode *cor(.)*, um eine Menge von Vektoren auf korrelierende Attribute zu vergleichen. Korrelierende Attribute werden sowohl in positiver als auch negativer Richtung betrachtet. Eine positive Korrelation ist ein gleichzeitiger Anstieg zweier Attribute, z.B.: Mit steigender Regenmenge, steigt der Wasserpegel. Bei negativer Korrelation sinkt ein Attribut bei gleichzeitigem Anstieg des zweiten Attributs, z.B.: Bei steigender Regenmenge, sinkt die Temperatur. Der Korrelationswert k nimmt Werte zwischen -1 und 1 an. Tabelle 7.7 betrachtet ausgewählte Korrelationen der Attribute.

Tabelle 7.7: Korrelierende Attribute zum Verständnis

Id	Nr.	Attribut	k	Attribut	Nr
K.1	F.3	Status	-0.31	unerwartete Änderung	F.26
K.2	F.4	# Commits	0.31	# Kommentare	F.5
K.3	F.5	# Kommentare	0.43	Alternative vorhanden	F.15
K.4	F.5	# Kommentare	0.41	Rationales Con-Argument	F.29
K.5	F.5	# Kommentare	-0.02	Nicht Rationales Con-Argument	F.31
K.6	F.10	# Alternativen	0.46	Rationales Con-Argument	F.29
K.7	F.13	Eindeutige Lösung vorhanden	0.64	Codeänderung ist konsistent	F.22
K.8	F.14	Begründung zur Lösung	0.23	Codeänderung ist konsistent	F.22
K.9	F.17	# LOC hinzugefügt	-0.41	Kernänderung ist identifizierbar	F.23
K.10	F.19	# LOC geändert	0.29	unerwartete Änderung	F.26
K.11	F.23	Kernänderung ist identifizierbar	0.46	Eindeutige Lösung vorhanden	F.13
K.12	F.23	Kernänderung ist identifizierbar	0.46	Codeänderung ist Konsistent	F.22
K.13	F.25	Auswirkung identifizierbar	0.87	Auswirkung erwartet	F.24

Korrelationswert K.1 beschreibt eine leichte Abhängigkeit zwischen dem Status und vorhanden Quellcodeänderungen, die nicht im Entscheidungsproblem diskutiert wurden. Von vier offenen JIRA-Issues besitzen zwei unerwartete Änderungen. Dieser Wert ist nicht aussagekräftig.

Korrelationswert K.2 beschreibt eine leichte Abhängigkeit zwischen der Anzahl der Commits und der Anzahl der Kommentare zu einem JIRA-Issue. Daraus folgt, dass große Diskussionen mehrere Commits produzieren. Subjektiv entsteht allerdings der Eindruck, dass die Anzahl der Commits von den Autoren abhängt und davon, ob eine Lösung zu einem Entscheidungsproblem noch auf weiteren Branches commitet wird. Abbildung 7.9d beschreibt dieses Verhältnis.

Korrelationswert K.3 beschreibt eine Korrelation zwischen der Anzahl der Kommentare zu einem JIRA-Issue und den vorhandenen Alternativen zum Entscheidungsproblem des JIRA-Issue. Daraus folgt, dass LUCENE EntwicklerInnen in JIRA-Issues mit großen Diskussionen viele Alternativen und Verbesserungen vorgeschlagen.

Die Korrelationswerte K.4, K.5 und K.6 beschreiben die Korrelationen zwischen rational argumentierten Con-Argumenten und der Anzahl der Kommentare im JIRA-Issue und Alternativen im Entscheidungsproblem. Die Werte zeigen, dass LUCENE EntwicklerInnen in großen Diskussionen ihre Gegenargumente meist rational formulieren. Dies deckt sich mit Abbildung 7.8

[4] Numerische Antwort: Textfragen wie bspw. F.13 zur eindeutigen Lösung eines Entscheidungsproblems werden in die Aussage „beantwortet“ bzw „nicht beantwortet“ überführt.

Der Korrelationswert K.7 beschreibt die Abhängigkeit der Konsistenz von Quellcodeänderungmit Entscheidungswissen zu einer dokumentierten Lösung des Entscheidungsproblems. Daraus folgt, dass eine eindeutige Lösung zu einem Entscheidungsproblem das Verständnis einer Quellcodeänderung unterstützt. Abbildung 7.9b beschreibt dieses Verhältnis.

Der Korrelationswert K.8 zeigt, dass die Begründung zu einer Lösung weniger Einfluss auf das Verständnis einer Quellcodeänderung hat als die Lösung des Entscheidungsproblems. Diese Aussage deckt sich nicht mit den Ergebnissen der Literaturrecherche in Kapitel 3 ab Seite 13.

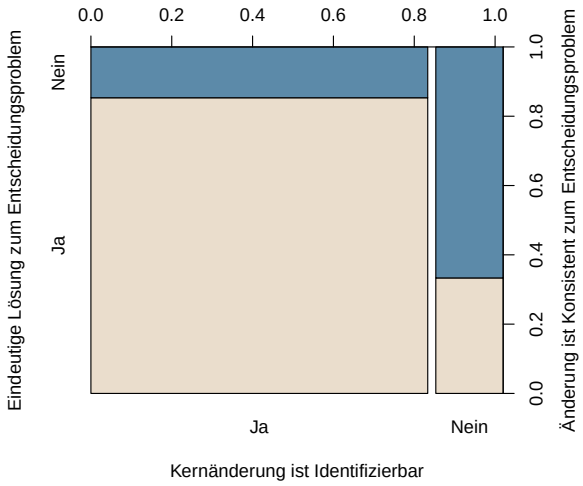
Der Korrelationswert K.9 zeigt, dass sich eine hohe Anzahl an LOC pro Quellcodeänderung negativ auf die Identifizierbarkeit der Kernänderung auswirkt.

Der Korrelationswert K.10 beschreibt eine leichte Abhängigkeit zwischen Commits mit vielen geänderten LOC und Quellcodeänderungen, die nicht im Entscheidungsproblem dokumentiert sind.

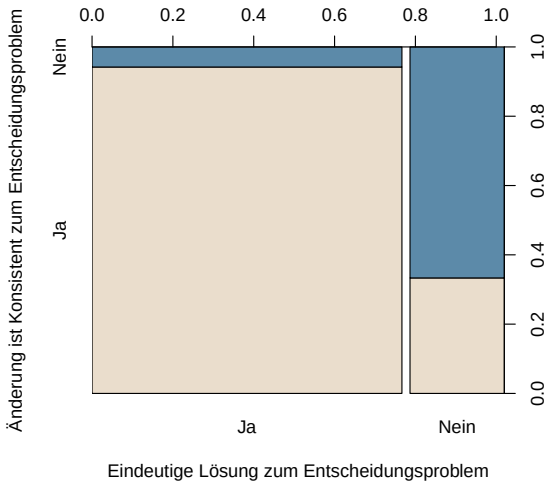
Der Korrelationswert K.11 identifiziert eine Abhängigkeit zwischen der Identifizierbarkeit einer Quellcodeänderung und der Existenz einer eindeutigen Lösung des Entscheidungsproblems. Daraus folgt, dass die Beschreibung der Lösung im dokumentierten Entscheidungswissen die/-den BetrachterIn unterstützt eine Quellcodeänderung zu identifizieren. Abbildung 7.9a zeigt dies Verhältnis.

Der Korrelationswert K.12 identifiziert eine Abhängigkeit zwischen der Identifizierbarkeit einer Quellcodeänderung und der Konsistenz der Quellcodeänderung zum dokumentierten Entscheidungswissen. Dies bestätigt das Ergebnis und die Interpretation von K.11 und K.7. Abbildung 7.9c zeigt dies Verhältnis.

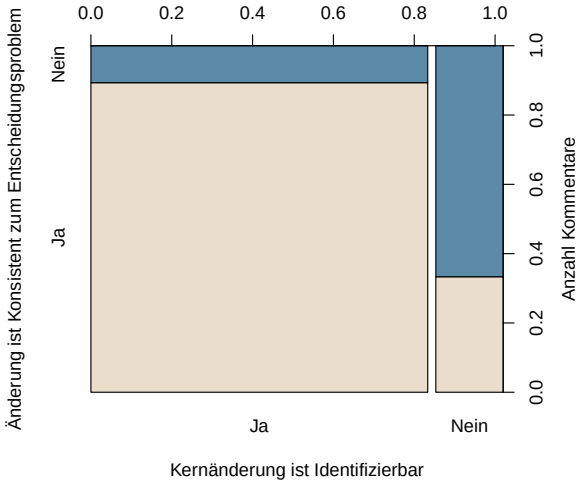
Der Korrelationswert K.13 bestätigt, dass in 87 % der untersuchten JIRA-Probleme eine Änderungsauswirkung identifiziert werden konnte, sofern diese erwartet wurde.



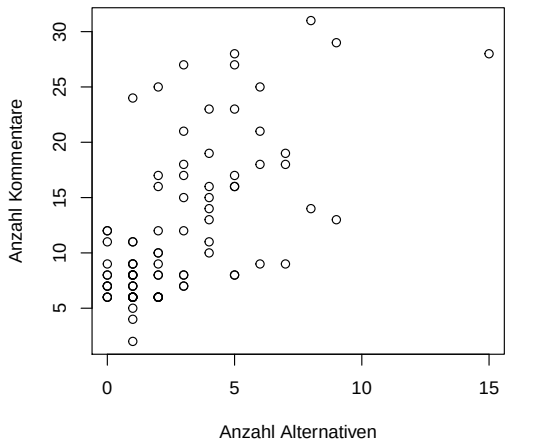
(a) Vergleich F.23 - F.13:
Korrelation einer identifizierbaren Kernänderung und einer eindeutig dokumentierten Entscheidung.



(b) Vergleich F.13 - F.22
Korrelation einer eindeutig dokumentierten Entscheidung und einer zu dem Entscheidungswissen konsistenten Quellcodeänderung.



(c) Vergleich F.23 - F.22
Korrelation einer identifizierbaren Kernänderung und einer zu dem Entscheidungswissen konsistenten Quellcodeänderung.



(d) Vergleich F.5 - F.15
Korrelation zwischen der Anzahl der Alternativen und Anzahl der Kommentare eines JIRA-Issues

Abbildung 7.9: Visualisierung der Korrelationen einzelner Attribute

7.4 Zusammenfassung

Die Evaluation zeigt die Eignung von DecXtract zur Entscheidungsdokumentation mit zwei Fragen. Durch Ef.1 wird die Eignung von DecXtract zur Entscheidungsdokumention durch eine Umfrage mit Studierenden untersucht. Die Umfrage betrachtet die Variablen: Benutzbarkeit, Nützlichkeit und Absicht in den Systemfunktionen 2, 3, 4, 5, 6 und 7. Die Umfrage zeigt gute Resultate für alle drei Variablen mit weiteren Vorschlägen zur Verbesserung und Erweiterung. Die TeilnehmerInnen der Umfrage bewerten jedoch die Genauigkeit des Klassifikators für Entscheidungswissen als zu ungenau und wünschen sich hier noch eine Verbesserung.

Mit Ef.2 wird der Mehrwert von Entscheidungswissen in JIRA-Issue-Kommentaren zum Verständnis von Quellcodeänderungen analysiert. Dafür wird das in Kapitel 3 ab Seite 13 gesammelte Wissen genutzt, um Fragen zum Verständnis einer Quellcodeänderung anhand Entscheidungswissen aus JIRA-Issue-Kommentaren zu definieren. Für diese Evaluationsfrage wurde ein Evaluationsdatensatz in einem lokalen JIRA-Projekt erzeugt. Alle enthaltenen JIRA-Issues wurden mit dem definierten Fragenkatalog untersucht und die Quellcodeänderung mit dem Entscheidungswissen verglichen. Die Beantwortung des Fragenkatalogs prüft ob das aus den JIRA-Issue-Kommentaren extrahierte Entscheidungswissen beim Verständnis der Quellcodeänderung hilft.

Die Untersuchung zeigt, dass bei ca. drei Viertel der untersuchten JIRA-Issues Entscheidungswissen in deren Kommentaren festgehalten ist. Enthalten die JIRA-Issue-Kommentare Entscheidungswissen, hilft dies auch beim Verständnis der Quellcodeänderung. Die Größe der Quellcodeänderung wird dabei als Verständnis beeinflussender Faktor identifiziert. Je größer eine Quellcodeänderung ist, desto schwieriger ist diese zu verstehen, auch mit vorliegendem Entscheidungswissen.

8 Schlussfolgerung

Die Zusammenfassung beschreibt die wesentlichen Ergebnisse dieser Arbeit. Die Aussagen beziehen Stellung zu den wichtigsten Entscheidungsproblemen der einzelnen Kapitel und beantworten definierte Fragen. Der Ausblick beschreibt Ansatzpunkte zur Verbesserung dieser Arbeit sowie Ideen für das CURES–ConDec Projekt.

8.1 Zusammenfassung

Die Rolle der Rationale ManagerIn hat die Aufgabe, Entscheidungswissen in Software-Projekten explizit zu dokumentieren und zu verwalten. EntwicklerInnen halten Entscheidungswissen oft nur informell in Kommentaren des genutzten Issue-Trackers oder in Commit-Nachrichten fest [3]. Diese Arbeit befasst sich mit der Frage, wie informelles Entscheidungswissen in Kommentaren eines Issue-Trackers, in explizites Entscheidungswissen überführt werden kann. Für diesen Zweck wird das JIRA-Plug-In ConDec um die Komponente DecXtract erweitert, die es Rationale ManagerInnen und EntwicklerInnen ermöglicht, Entscheidungswissen automatisiert und manuell zu klassifizieren.

Eine systematische Literaturrecherche beantwortet die Fragen, welches Wissen EntwicklerInnen zum Verständnis von Quellcodeänderungen benötigen und welches Entscheidungswissen EntwicklerInnen zum Verständnis von Quellcodeänderungen benötigen. Die Recherche liefert 21 Fragen von EntwicklerInnen aus ihrem Alltag im Umgang mit Quellcodeänderungen.

EntwicklerInnen können DecXtract nutzen, um Entscheidungswissen explizit in Kommentaren zu klassifizieren und mit anderen Wissens Artefakten zu verbinden. Die explizite Darstellung von informellem Entscheidungswissen in JIRA-Issue-Kommentaren soll die Verständlichkeit von Quellcodeänderungen erhöhen.

Eine Umfrage mit Studierenden zeigt die Eignung von DecXtract durch Befragung zur Benutzbarkeit, Einfachheit und Intention. Die Befragten loben die gute Integration in JIRA und die Visualisierungen von DecXtract und ConDec.

Die Evaluation von DecXtract nutzt die gewonnene Erkenntnis aus der Literaturrecherche zur Definition von Fragen zur Analyse von Quellcodeänderungen. Das JIRA-Projekt von Apache LUCENE wurde als Evaluationsdatensatz ausgewählt. 90 JIRA-Issues wurden in ein lokales JIRA-Projekt überführt. Mithilfe von DecXtract wurde Entscheidungswissen in den JIRA-Issue-Kommentaren klassifiziert und miteinander verlinkt. Jede Quellcodeänderung wurde mithilfe des expliziten Entscheidungswissens verglichen. Dabei wurde für jedes JIRA-Issue ein Protokoll aufgezeichnet und Fragen zur Konsistenz, Vollständigkeit und der Entscheidungsfindungsstrategie gestellt.

Die Auswertung des Protokolls für alle 90 JIRA-Issues zeigt, dass explizites Entscheidungswissen aus JIRA-Issue Kommentaren EntwicklerInnen beim Verständnis von Quellcodeänderungen unterstützt. Die Größe der Quellcodeänderung, sowie schlecht dokumentiertes Entscheidungswissen werden als negativ auswirkende Faktoren identifiziert.

8.2 Ausblick und Diskussion

Die Umfrage sowie die Testergebnisse zeigen schlechte Ergebnisse des Klassifikators in der Anwendung außerhalb der Trainingsdaten. Dieses Problem bietet noch einige Möglichkeiten zur Verbesserung.

Eine Ontologie und die Nutzung von natürlichsprachiger Texterkennung mit Natural Language Processing-Ansätzen können die Eingabedaten auf Synonyme und häufig auftretende Wörter in ähnlichem Kontext prüfen. Zudem sollte ein Klassifikator nur auf dem Projekt angewendet werden, auf dem er trainiert wurde. Dafür sollte eine Funktionalität entworfen werden, die es ermöglicht einen Klassifikator nach einer Trainingsphase (manueller Klassifikation durch EntwicklerInnen) auf dem eigenen Projekt anzuwenden und regelmäßig neu zu trainieren (*Active Learning)*. Die Anwendung von *Explainable Machine Learning* kann dabei helfen, nach der Klassifikation Schlagworte zu identifizieren, die der Machine Learning Algorithmus für die Wissenstypen gefunden hat. Die Anwendung von *Active Learning* ermöglicht zudem die schnelle Einbindung von neuen Wissenstypen.

Um die Verbindung zwischen Quellcodeänderungen und Entscheidungswissen noch zu verstärken, kann ConDec auf das angebundene Versionierungssystem zugreifen und Commit Nachrichten als Quelle für Entscheidungswissen nutzen. Zudem können Quellcodeausschnitte aus JIRA-Issue-Kommentaren mit einer Quellcodeänderung durch Analyse der textuellen Ähnlichkeit in Verbindung gebracht werden. Dies ermöglicht eine schnelle Identifizierung der Kernänderung.

Literaturverzeichnis

[1] Hesse, T.-M., Kuehlwein, A., Paech, B., Roehm, T., & Bruegge, B. (2015). *Documenting Implementation Decisions with Code Annotations.* In 27th International Conference on Software Engineering and Knowledge Engineeringpp. pp. 152–157. Pittsburgh, PA , USA: KSI Research Inc. https://doi.org/10.18293/SEKE2015-084

[2] Hesse, T.-M., & Paech, B. (2013). *Supporting the Collaborative Development of Requirements and Architecture Documentation.* In 3rd International Workshop on the Twin Peaks of Requirements and Architecture pp. 22–26. Rio de Janeiro, Brazil: IEEE. https://doi.org/10.1109/TwinPeaks-2.2013.6617355

[3] Alkadhi, R., Nonnenmacher, M., Guzman, E., & Bruegge, B. (2018). *How do developers discuss rationale?* In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER) pp. 357–369. Campobasso, Italy: IEEE. https://doi.org/10.1109/SANER.2018.8330223

[4] Alkadhi, R., Lata, T., Guzmany, E., & Bruegge, B. (2017). *Rationale in Development Chat Messages: An Exploratory Study.* In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR) pp. 436–446. Buenos Aires, Argentina: IEEE. https://doi.org/10.1109/MSR.2017.43

[5] Kleebaum, A., Johanssen, J. O., Paech, B., & Alkadhi, R. (2018). *Decision Knowledge Triggers in Continuous Software Engineering.* In 4th International Workshop on Rapid Continuous Software Engineering (RCoSE'18). Gothenburg, Sweden: ACM. https://doi.org/10.1145/3194760.3194765

[6] Ko, A. J., DeLine, R., & Venolia, G. (2007). *Information Needs in Collocated Software Development Teams.* In 29th International Conference on Software Engineering (ICSE'07) pp. 344–353. IEEE. https://doi.org/10.1109/ICSE.2007.45

[7] Alkadhi, R. M. A. (2018). *Rationale in Developers ' Communication.* Doctoral dissertation. Technische Universität München.

[8] Jansen, A., & Bosch, J. (2005). *Software architecture as a set of architectural design decisions.* In 5th Working IEEE/IFIP Conference on Software Architecture, WICSA 2005, pp. 109–120. https://doi.org/10.1109/WICSA.2005.61

[9] Kunz, W. & Rittel, H. (1970). *Issues as Elements of Information Systems.* Institute of Urban and Regional Development, University of California, Berkeley, California, Working Paper 131,

[10] Paech, B., Delater, A., & Hesse, T.-M. (2014). *Supporting Project Management Through Integrated Management of System and Project Knowledge.* In G. Ruhe & C. Wohlin (Eds.), Software Project Management in a Changing World pp. 157–192. Berlin, Heidelberg, Germany: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-55035-5

[11] Bishop, C., & Nasrabadi, N. (2007). *Pattern recognition and machine learning.* In Journal of Electronic Imaging, 16(4), 738. https://doi.org/10.1117/1.2819119

[12] Lougher, R., & Rodden, T. (1993). *Supporting long-term collaboration in software maintenance.* In Conference on Organizational Computing Systems (COCS'93) pp. 228–238. http://doi.org/10.1145/168555.168581

[13] Rish, I. (2001). *An empirical study of the naive Bayes classifier.* In Empirical Methods in Artificial Intelligence Workshop, IJCAI, 22230(JANUARY 2001), pp. 41–46. https://doi.org/10.1039/b104835j

[14] Powers, M. W. (2011). *Evaluation: from Precision, Recall and F-measure to ROC, Informedness, Markedness and Correlation.* In Journal of Machine Learning Technologies, 2(1), pp. 37–63. https://doi.org/10.1.1.214.9232

[15] Kohavi, R. (1995). *A study of cross-validation and bootstrap for accuracy estimation and model selection.* In 14th International Joint Conference on Artificial Intelligence - Volume 2, 2(0), pp. 1137–1143. https://doi.org/10.1067/mod.2000.109031

[16] Androutsopoulos, I., Koutsias, J., Chandrinos, K. V., Paliouras, G., & Spyropoulos, C. D. (2000). *An Evaluation of Naive Bayesian Anti-Spam Filtering.* In Machine Learning in the New Information Age,pp. 9–17. https://doi.org/10.1109/IAW.2007.381951

[17] Kleebaum, A., Johanssen, J. O., Paech, B., & Bruegge, B. (2018). *Tool Support for Decision and Usage Knowledge in Continuous Software Engineering.* In S. Krusche, H. Lichter, D. Riehle, & A. Steffens (Eds.), Workshop on Continuous Software Engineering (CSE'18) pp. 74–77. Ulm, Germany: CEUR-WS.org. 10.11588/heidok.00024186

[18] International Organization for Standardization/ International Electrotechnical Commission. (2011). Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models ISO/IEC 25010:2011(E). Geneva, Switzerland: International Organization for Standardization, International Electrotechnical Commission.

[19] Pohl, K., & Rupp, C. (2015). *Basiswissen Requirements Engineering: Aus-und Weiterbildung nach IREB-Standard zum Certified Professional for Requirements Engineering Foundation Level.* dpunkt. verlag.

[20] Levenshtein, V. I. *Binary codes capable of correcting deletions, insertions, and reversals.* In Soviet physics doklady. Vol. 10. No. 8. 1966 pp. 707-710

[21] Lehman, M. M. (1980). *Programs, Life Cycles, and Laws of Software Evolution.* In IEEE, 68(9), pp. 1060–1076. http://doi.org/10.1109/PROC.1980.11805

[22] Hesse, T.-M., Lerche, V., Seiler, M., Knoess, K., & Paech, B. (2016). *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. http://doi.org/10.1016/j.infsof.2016.06.003

[23] Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997). *Metrics and laws of software evolution-the nineties view.* In Fourth International Software Metrics Symposium pp. 20–32. IEEE Comput. Soc. http://doi.org/10.1109/METRIC.1997.637156

[24] Fred D. Davis, Richard P. Bagozzi, P. R. W. (1989). *User Acceptance of Computer Technology : A Comparison of Two Theoretical Models.* Management Science, 35(8), pp. 982–1003.

[25] Kitchenham, B. A., & Charters, S. (2007). *Guidelines for Performing Systematic Literature Reviews in Software Engineering* (Version 2.3) (No. EBSE Technical Report 2007-001) (Vol. 2). Keele, Staffs, UK; Durham, UK: Citeseer. http://doi.org/10.1145/1134285.1134500

[26] Jalali, S., & Wohlin, C. (2012). *Systematic Literature Studies: Database Searches vs. Backward Snowballing*. In ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'12) pp. 29–38. Lund, Sweden: ACM. http://doi.org/10.1145/2372251.2372257

[27] Krusche, S., & Bruegge, B. (2017). *CSEPM - A Continuous Software Engineering Process Metamodel*. In 3rd International Workshop on Rapid Continuous Software Engineering, RCoSE 2017, pp 2–8. http://doi.org/10.1109/RCoSE.2017.6

[28] P. Runeson, M. Höst, A. Rainer, & B. Regnell. *Case Study Research in Software Engineering. Guidelines and Examples*. Wiley, 1st edition, 2012.

[29] Spillner, A., Linz, T., *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester; Foundation Level nach ISTQB-Standard*. Heidelberg: dpunkt-Verl., 2012.

[30] Shihab, E., Hassan, A. E., Adams, B., & Jiang, Z. M. (2012). *An industrial study on the risk of software changes*. In ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12 https://doi.org/10.1145/2393596.2393670

[31] Ko, A. J., DeLine, R., & Venolia, G. (2007). *Information Needs in Collocated Software Development Teams*. In 29th International Conference on Software Engineering pp. 344-353. https://doi.org/10.1109/ICSE.2007.45

[32] Tao, Y., Dang, Y., Xie, T., Zhang, D., & Kim, S. (2012). *How do software engineers understand code changes?* In ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12 pp. 1-11. Cary, North Carolina, USA: ACM Press. https://doi.org/10.1145/2393596.2393656

[33] LaToza, T. D., & Myers, B. A. (2010). *Hard-to-answer questions about code*. In Evaluation and Usability of Programming Languages and Tools on - PLATEAU '10, pp. 1–6. https://doi.org/10.1145/1937117.1937125

[34] Kim, M. (2011). *An exploratory study of awareness interests about software modifications*. In 4th international workshop on Cooperative and human aspects of software engineering - CHASE '11 p. 80. New York, New York, USA: ACM Press. https://doi.org/10.1145/1984642.1984662

[35] Fritz, T., & Murphy, G. C. (2010). *Using information fragments to answer the questions developers ask*. In 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10, 1, 175. https://doi.org/10.1145/1806799.1806828

[36] LaToza, T. D., Venolia, G., & DeLine, R. (2006). *Maintaining mental models*. In 28th International Conference on Software Engineering - ICSE '06, 492. https://doi.org/10.1145/1134285.1134355

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

Backward Snowballing Literaturrecherche nach im vorliegenden Artikel zitierten Artikeln. 13

Berichte JIRA-Berichte sammeln Informationen und erstellen Übersichten zum aktuellen JIRA-Projekt. Bspw. Burndown Charts. 57

Bug Fehler im Quellcode. 88

Button Deutsch: Knopf. Hier: Interaktionsmöglichkeit für die NutzerIn, um auf der Oberfläche eine Funktion auszulösen. 60

Commit Speichern von lokalen Änderungen in einem Versionskontrollsystem. 80, 84, 85, 87, 89, 90

Continuous Software Engineering Softwareentwicklungsprozess, der NutzerInnen stark in den Entwicklungsprozess mit einbezieht. 5

CSS Cascading Style Sheets: Stylesheet Sprache für HTML Systeme. 49

Diff Unterschied zwischen Quellcode vor und nach einer Änderung. 80

Entscheidungsproblem Entscheidung, deren Alternativen im Projektumfeld abgewogen und entschieden wird. 1, 37, 38, 40, 43, 45, 48, 50, 51, 53, 55, 57

Entscheidungswissen Vorhandenes Wissen über eine Entscheidungsfrage. Argumente für oder gegen mögliche Alternativen sowie der Kontext der Entscheidung. 1

Forward Snowballing Literaturrecherche nach jüngeren Artikeln, die einen vorliegenden Artikel zitieren. 13

Icon Kleine Grafik. Hier: Symbol das einen Wissenstyp identifiziert. 60, 68

JSON JavaScript Object Notation. Kompaktes Datenformat zum Datenaustausch. 58

LOC Lines of Code. 17, 19, 83

LUCENE LUCENE ist eine von Apache entwickelte frei verfügbare Programmbibliothek zur Volltextsuche. Sie ist komplett in JAVA entwickelt und in vielen Applikationen einsetzbar. Kontext: Die Daten aus dem LUCENE-GIT und JIRA werden zur Evaluation dieser Arbeit benutzt. v, 10, 75, 80, 81, 80, 83, 85, 86, 87, 89, 91

Machine Learning Fähigkeit neues Wissen aus erlernten Daten zu generieren. 94

Natürliche Sprache Def. laut Sprachwissenschaft: Eine von Menschen gesprochene Einzelsprache die aus historischer Entwicklung entstanden ist. Im Kontext von Tools und EntwicklerInnen kommt es vor das neue Wörter eine eigene kontextbezogene Bedeutung haben. Bspw. +1 wird als Ich unterstütze diese Aussage gedeutet. 2

Patch Korrektur eines (fehlerhaften) Softwaresystems. 80

Rationale Completeness Rationale Completeness beschreibt die Vollständigkeit eines Entscheidungsproblem. Sie liegt vor wenn für jedes Problem eine Entscheidung mit Argumenten dokumentiert ist. 86

REST REST Schnittstelle. 33, 36, 50, 51, 53, 54, 58, 69, 80

Snowballing Snowballing bezeichnet die Verwendung eines vorliegenden Artikels um weitere Artikel zu identifiziere. 13, 15

Abkürzungsverzeichnis

CSE Continuous Software Engineering 21
Ef Evaluationsfrage 75
Ff Forschungsfrage 13
ITS Issue-Tracking-System 2
SML Supervised-Machine-Learning 9
TAM Technology Acceptance Model 75
UI User Interface 29
WS Work Space 29