Bericht zum Fortgeschrittenenpraktikum

Visualisierung von Wissen zu Code in Jira

durchgeführt
im Wintersemester 2020/21
am Lehrstuhl für Software Engineering
der Fakultät für Mathematik und Informatik
an der Ruprecht-Karls-Universität Heidelberg

Betreuerin: Anja Kleebaum
Bearbeiter: Marvin A. Ruder

Abgabedatum: 30. April 2021

Zusammenfassung

Zur Dokumentation und Verwaltung von Entscheidungswissen stehen die am Lehrstuhl für Software Engineering entwickelten ConDec-Plug-ins für das Issue Tracking System Atlassian Jira, die Entwicklungsumgebung (IDE) Eclipse sowie weitere bei der Entwicklung von Software häufig verwendete Tools (Atlassian Confluence, Slack, Atlassian Bitbucket) zur Verfügung. Sie erlauben die Dokumentation von Entscheidungswissen an verschiedenen Orten wie in eigenständigen Issues oder in Beschreibungstexten oder Kommentaren anderer Issues in Jira, aber auch in Code-Kommentaren oder Commit-Nachrichten des Versionskontrollsystems git. Das auf diese Weise dokumentierte Entscheidungswissen kann gemeinsam mit anderen Wissenselementen wie Anforderungen, Entwicklungsaufgaben oder Code auf verschiedene Arten visualisiert werden, wozu alle Wissenselemente in eine Graphstruktur innerhalb des ConDec-Jira-Plug-ins eingelesen und dort geeignet verlinkt werden. Jedoch ist das Auslesen und Verlinken von Codedateien sowie dem Entscheidungswissen aus Code-Kommentaren bisher nur ansatzweise implementiert und bietet Raum für Verbesserungen und Erweiterungen. In diesem Praktikum wurden die ConDec-Plug-ins in diesem Bereich um Funktionalitäten erweitert.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**

2.1 Wissensmodell in den ConDec-Tools . . . 3

2.2 Features der ConDec-Tools . . . 4

2.2.1 Dokumentationsorte . . . 4

2.2.2 Visualisierung . . . 5

2.2.3 Git-Anbindung . . . 5

2.2.4 Dashboards . . . 6

2.2.5 IDE-Unterstützung . . . 7

3 Anforderungen **8**

3.1 Grobanforderungen . . . 8

3.2 Detailanforderungen . . . 11

3.2.1 User Tasks und Subtasks . . . 11

3.2.2 Personae . . . 13

3.2.3 Systemfunktionen . . . 17

3.2.4 Arbeitsbereiche 26

3.2.5 Domänendaten 30

3.2.6 Nichtfunktionale Anforderungen 32

4 Entwurf und Implementierung 33

4.1 SF1: Filter knowledge graph 33

4.2 SF2: Configure decision knowledge extraction from git (commit messages and code comments) 38

4.3 SF3: List all code classes for a project 40

4.4 SF4: Automatically add code files from git repository into the knowledge graph 40

4.5 SF5: Automatically add and link decision knowledge from comments in code files into the knowledge graph 47

4.6 SF6: Show knowledge graph metrics with respect to code files . 49

4.7 SF7: Configure definition of done (DoD) for the decision knowledge documentation 51

4.8 SF8: Check definition of done (DoD) of the decision knowledge documentation related to a code file 53

4.9 SF9: Navigate to subgraph centered on a code file in Jira 54

4.10 SF10: Configure Jira Settings [in VS Code] 57

5 Qualitätssicherung 59

5.1 Kontiniuerliche Integration 59

5.1.1 Statische Codeanalyse 60

5.1.2 Testausführung 60

5.1.3 Testabdeckung 60
5.2 Komponententests 61
5.3 Systemtests 61
5.4 Prüfung der nichtfunktionale Anforderungen 62
5.4.1 Portability 62
5.4.2 Maintainability 64
5.4.3 Usability 65

6 Evaluation **66**

7 Fazit **75**

7.1 Zusammenfassung 75
7.2 Ausblick 76
7.3 Erfahrungen 78

8 Literatur **80**

Abbildungsverzeichnis **83**

Tabellenverzeichnis **85**

Kapitel 1

Einleitung

Requirements Engineers sowie Entwickler:innen treffen über die gesamte Entwicklungszeit einer Software hinweg Entscheidungen, etwa mit Bezug zu den Anforderungen, der Architektur, der Implementierung oder der Qualitätssicherung des zu entwickelnden Softwaresystems [3]. Das Wissen über diese Entscheidungen sowie ihr Kontext, darunter die durch sie gelösten Probleme, die mit ihnen zusammenhängenden Begründungen und Argumente, aber auch mögliche Alternativen (im Folgenden: Entscheidungswissen), stellen ein wichtiges Artefakt bei der Entwicklung von Softwaresystemen dar [1], [2].

Zur Dokumentation und Verwaltung von Entscheidungswissen stehen die am Lehrstuhl für Software Engineering entwickelten ConDec-Plug-ins für das Issue Tracking System *Atlassian Jira*, die Entwicklungsumgebung (IDE) *Eclipse* sowie weitere bei der Entwicklung von Software häufig verwendete Tools (*Atlassian Confluence*, *Slack*, *Atlassian Bitbucket*) zur Verfügung [2]. Sie erlauben die Dokumentation von Entscheidungswissen an verschiedenen Orten wie in eigenständigen Issues oder in Beschreibungstexten oder Kommentaren anderer Issues in Jira, aber auch in Code-Kommentaren oder Commit-Nachrichten des Versionskontrollsystems *git*. Das auf diese Weise dokumentierte Entscheidungswissen kann gemeinsam mit anderen Wissenselementen wie Anforderungen, Entwicklungsaufgaben oder Code auf verschiedene Arten visualisiert werden, wozu alle Wissenselemente in eine Graphstruktur innerhalb von ConDec Jira eingelesen und dort geeignet verlinkt werden [2], [3].

Jedoch ist das Auslesen und Verlinken von Codedateien sowie dem Entscheidungswissen aus Code-Kommentaren bisher nur ansatzweise implementiert und bietet Raum für Verbesserungen und Erweiterungen. Ziel des Praktikums ist es daher, die ConDec-Plug-ins in diesem Bereich um Funktionalitäten zu erweitern.

Konkret wird im Rahmen des Praktikums ermöglicht, an das ConDec-Jira-Plug-in auch passwortgeschützte Git-Repositorien anzubinden. Die Codedateien sowie Entscheidungswissen in Code-Kommentaren werden als Wissenselemente in den Wissensgraphen in Jira eingelesen. Weiterhin wird das Filtern des Wissensgraphen verbessert, indem herausgefilterte Elemente durch transitive Links ersetzt werden. So können den Nutzer:innen mehr gemäß den Filterkriterien relevante Wissenselemente angezeigt werden. Weiterhin wird die Rationale Coverage von Codedateien berechnet und visualisiert. Codedateien, deren Rationale Coverage gemäß einer konfigurierbaren *Definition of Done* nicht ausreichend ist, werden außerdem im *Rationale Backlog* angezeigt.

Auch wird für die IDEs *Eclipse* sowie *Visual Studio Code* eine einfache Navigationsmöglichkeit von einer Codedatei in der IDE zu einem auf diese Codedatei zentrierten Wissensgraphen in Jira geschaffen. Hierfür ist das ConDec-Eclipse-Plug-in zu erweitern sowie ein neues ConDec-Plug-in für *Visual Studio Code* zu erstellen.

Dieser Bericht ist wie folgt strukturiert: In Kapitel 2 werden grundlegende Begriffe und Konzepte eingeführt. Kapitel 3 beschreibt das Vorgehen bei der Ableitung geeigneter Detailanforderungen, die die Grundlage für Entwurf und Implementierung in Kapitel 4 bilden. Die zur Qualitätssicherung durchgeführten Schritte sind in Kapitel 5 beschrieben. Eine Evaluation der Ergebnisse anhand eines Nutzungsbeispiels erfolgt in Kapitel 6. Kapitel 7 fasst die Ergebnisse sowie das im Praktikum erlernte Wissen und die dort gesammelten Erfahrungen zusammen.

Kapitel 2

Grundlagen

Im folgenden Kapitel werden Grundlagen eingeführt, auf denen das Praktikum aufbaut. Hierzu gehören Features und Sichten der ConDec-Tools sowie das verwendete Modell zur Dokumentation von Entscheidungswissen.

2.1 Wissensmodell in den ConDec-Tools

Die ConDec-Tools verwenden ein Wissensmodell bestehend aus Wissenselementen, zwischen denen Beziehungen bestehen. Wissenselemente umfassen dabei Artefakte wie Anforderungen, Entwicklungsaufgaben, Codedateien und Entscheidungswissenselemente. Zu letzteren gehören die standardmäßig verwendbaren Typen von Entscheidungswissenselementen *Entscheidungsproblem* (engl. *Issue*), *Entscheidung* (engl. *Decision*), *Alternative* (engl. *alternative*) und *Pro-* sowie *Contra-Argument* (engl. *pro-*, *con-argument*) [2], die in folgendem Zusammenhang stehen:

Ein Entscheidungsproblem wird durch eine Entscheidung gelöst, wobei Alternativen das Entscheidungsproblem ebenfalls lösen würden. Für oder gegen eine Entscheidung oder eine Alternative sprechen Argumente der jeweiligen Art (Pro oder Contra).

Type: User Story Status: IN PROGRESS
Priority: High Resolution: Unresolved

Description
As a user, I want to choose a password so that I can securely log in to the system.

Should we check the security of the password?

Do not check the security of the password!

The users might choose a very simple password that can be easily cracked by brute-force attacks, such as their dog's name.

Use a library to check the password strength!

The users are assisted in choosing strong passwords that cannot be cracked.

Abbildung 2.1: Entscheidungswissenselemente dokumentiert in der Beschreibung eines Jira-Issues. [2]

Weitere vordefinierte Elementtypen können von Nutzer:innen aktiviert und verwendet werden, sind für dieses Praktikum jedoch nicht von Relevanz und werden daher hier nicht weiter betrachtet.

Beziehungen zwischen Wissenselementen sind gerichtet und tragen Bezeichnungen wie *steht in Beziehung zu* (engl. *relates*), *unterstützt* (engl. *supports*, besonders für Pro-Argumente) oder *greift an* (engl. *attacks*, besonders für Contra-Argumente). [2]

2.2 Features der ConDec-Tools

2.2.1 Dokumentationsorte

Wissenselemente können in ConDec an verschiedenen Stellen dokumentiert werden. Die im Praktikum relevanten Dokumentationsorte umfassen eigene Jira-Issues sowie Beschreibungen von (vgl. Abbildung 2.1) und Kommentare in Jira-Issues, Code-Kommentare sowie Commit-Nachrichten im Versionskontrollsystem *git* [2]. Die Dokumentation in Code-Kommentaren erfolgt in einer Javadoc-ähnlichen Syntax, die in Listing 2.1 beispielhaft dargestellt ist.

```java
/**
 * @issue What shall we do with the drunken sailor?
 * @alternative Put him in the long boat till he's sober!
 * @pro After this procedure, the sailor will be sober.
 * @con This procedure endangers the availability of the long boat in a case of
    shipwrecking or maritime emergency.
 * @alternative Put him in the scuppers with a hose-pipe on him!
 * @pro After this procedure, the sailor will probably not get drunk again.
 * @con This procedure is in violation of the United Nations Convention against
    Torture and Other Cruel, Inhuman or Degrading Treatment or Punishment.
 *
 */

public class DrunkenSailorDealer {
        // TODO Deal with the drunken sailor
}
```

Listing 2.1: Entscheidungswissenselemente dokumentiert in einer Java-Klasse.

2.2.2 Visualisierung

Das an den verschiedenen Stellen dokumentierte Entscheidungswissen wird gemeinsam mit weiteren Wissenselementen in einem Wissensgraphen organisiert, der auf verschiedene Weisen visualisiert werden kann. Hierzu gehören unter anderem die in Abbildung 2.2 dargestellte Baumstruktur. Je nach Kontext wird der gesamte Wissensgraph eines Softwareprojekts oder nur ein Teilgraph bezogen auf ein bestimmtes ausgewähltes Wurzelelement angezeigt.

Weiterhin ist eine Rationale-Backlog-Ansicht verfügbar, in der alle Wissenselemente gelistet sind, für die Entscheidungswissen nicht auf die vorgesehene Weise dokumentiert ist. Die Definition, ab wann ein Wissenselement in Bezug auf Entscheidungswissen als gut dokumentiert gilt (*Definition of Done*), lässt sich von Nutzer:innen anpassen.

2.2.3 Git-Anbindung

Das ConDec-Jira-Tool erlaubt eine Anbindung an Git-Repositorien, aus denen Codedateien sowie Commit-Nachrichten ausgelesen werden können. Mithilfe

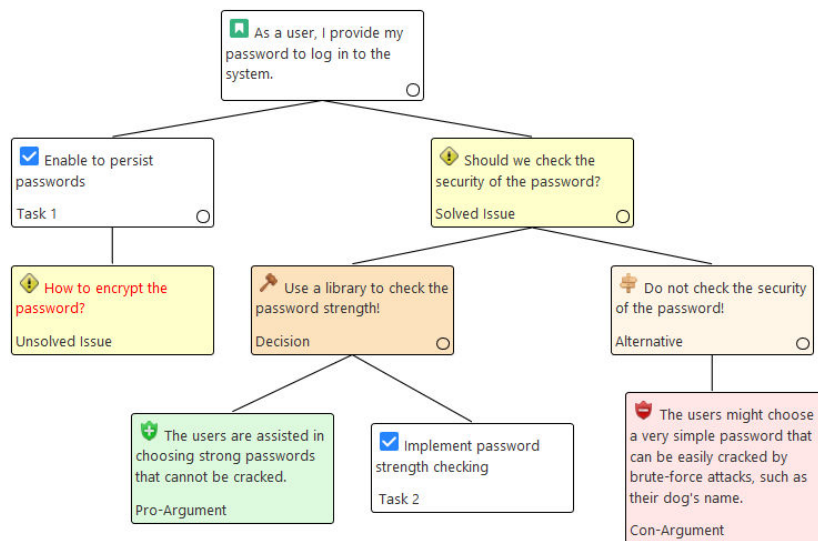

Abbildung 2.2: Visualisierung eines Wissensgraphen als Baum. [2]

von Jira-Issue-Keys innerhalb von Commit-Nachrichten werden Wissenselemente aus Git-Repositorien mit Jira-Issues (in der Regel des Typs *Entwicklungsaufgabe*, engl. *Work Item*) verlinkt. So wird jede Codedatei, die in einem Commit verändert wurde, mit jedem Jira-Issue verlinkt, dessen Key sich in der Nachricht des jeweiligen Commits befindet.

2.2.4 Dashboards

Metriken, die auf dem Wissensgraphen berechnet werden, etwa die Rationale Coverage[1] oder die Intra-Rationale-Completeness, werden in Dashboards visualisiert. So können etwa Tortendiagramme oder Boxplots Aufschluss über die Rationale Coverage von Jira-Issue eines bestimmten Typs geben. Ein Beispiel ist in Abbildung 2.3 gezeigt.

[1] Abdeckung von Anforderungen oder Code durch dokumentiertes Entscheidungswissen.

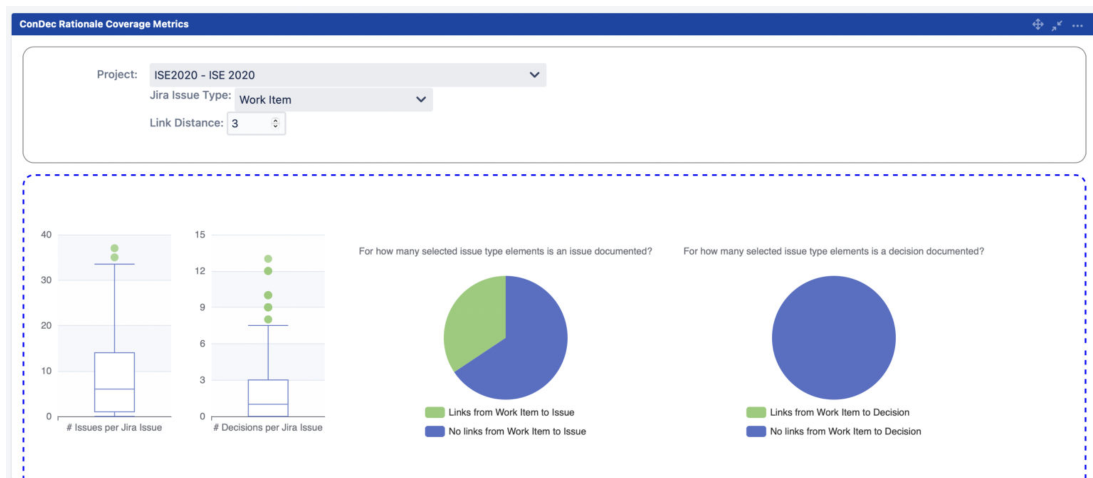

Abbildung 2.3: Visualisierung von Entscheidungswissen in Dashboard. [2]

2.2.5 IDE-Unterstützung

Für die Entwicklungsumgebung *Eclipse* ist ein ConDec-Plug-in verfügbar, das auf den Wissensgraphen in ConDec-Jira zugreift und diesen innerhalb Eclipse visualisiert.

Kapitel 3

Anforderungen

In diesem Kapitel werden ausgehend von den gegebenen Grobanforderungen [vgl. 3] die Detailanforderungen an die Erweiterungen der ConDec-Tools abgeleitet.

3.1 Grobanforderungen

Die folgend aufgeführten Grobanforderungen an das ConDec-Jira-Plug-in lagen zu Beginn vor:

R1: Anbindung von passwortgeschützten Git-Repositorien Das bestehende ConDec-Jira-Plug-in kann nur auf Repositorien zugreifen, die ohne Authentifizierung, etwa mittels Passwort, zugänglich sind. Das schränkt die Verwendung von ConDec besonders bei der Entwicklung proprietärer Software, die nicht quelloffen verfügbar ist, ein. Im Praktikum soll daher eine Anbindung passwortgeschützter Repositorien umgesetzt werden.

R2: Integration aller Codedateien aus Git in den Wissensgraphen Zurzeit werden nur Dateien mit der Dateiendung `.java` in den Wissensgraphen integriert. Entscheidungswissen, das in Code-Kommentaren anderer Programmiersprachen dokumentiert ist, wird dabei ebenso wie die Codedateien selbst

ignoriert. Im Praktikum soll eine Möglichkeit gefunden werden, Codedateien aller gängigen Programmiersprachen in den Wissensgraphen zu integrieren.

R3: Integration von Entscheidungswissen aus Code-Kommentaren in den Wissensgraphen In der bisherigen Implementierung des ConDec-Jira-Plugins werden Elemente des Entscheidungswissens in Code-Kommentaren nur für eine Feature-Branch-Ansicht extrahiert, nicht jedoch, um diese in den Wissensgraphen zu integrieren. Die Umsetzung dieser Integration der Wissenselemente aus Code-Kommentaren in den Wissensgraphen ist ein Ziel des Praktikums.

R4: Visualisierung von Wissen zu Code Da bisher keine Integration von Codedateien und darin enthaltenen Elementen des Entscheidungswissens in den Wissensgraphen erfolgte, bestand zugleich keine Möglichkeit, dieses Wissen zu visualisieren. Ein Ziel des Praktikums ist daher, eine Visualisierung des Wissensgraphen oder Subgraphen davon zu ermöglichen, die Codedateien sowie zu diesen verlinkte Wissenselemente anzeigt. Hierbei ist auf die bestehenden Visualisierungs-Frameworks zurückzugreifen.

R5: Filtern des Wissensgraphen durch Aufbau transitiver Links Die bestehenden Visualisierungsmöglichkeiten von ConDec Jira ermöglichen das Filtern von Subgraphen des Wissensgraph nach verschiedenen Kategorien (z.B. Typ des Wissenselements, Linkdistanz zu Wurzelelement des Subgraphen). Hierbei können jedoch Wissenselemente, die gemäß der Filterkriterien angezeigt werden sollen, bei der Visualisierung verloren gehen, wenn sie mit dem Wurzelelement des Subgraphen nur über Wissenselemente verlinkt sind, die gemäß der Filterkriterien nicht angezeigt werden sollen. Damit die anzuzeigenden Wissenselemente Teil des gefilterten Subgraphen sind, ist ein Ziel des Praktikums, herausgefilterte Wissenselemente durch transitive Links zu ersetzen.

So soll im beispielhaften Wissensgraphen

$$\text{Codedatei} \rightarrow \text{Work Item} \rightarrow \text{Entscheidungsproblem}$$

bei Entfernen des Wissenselements „Work Item“ ein transitiver Link zwischen „Codedatei“ und „Entscheidungsproblem“ erstellt werden. In der bisherigen Im-

plementierung würde im obigen Fall allein das Wissenselement „Codedatei“ angezeigt werden.

R6: Berechnung und Darstellung der Rationale Coverage von Codedateien Die zurzeit implementierten Dashboards des ConDec-Jira-Plug-ins erlauben es, Metriken in Bezug auf Jira-Issues bestimmter Typen in Relation zu verlinkten Entscheidungsproblemen und Entscheidungen anzuzeigen. Ein Ziel des Praktikums soll es sein, eine solche Visualisierungsmöglichkeit auch in Bezug auf Codedateien in Relation zu verlinkten Entscheidungsproblemen und Entscheidungen zu schaffen.

R7: Konfiguration der Definition of Done für Codedateien und Anzeige im Rationale Backlog Im bestehenden ConDec-Jira-Plug-in kann zu verschiedenen Artefakten konfiguriert werden, welche Voraussetzungen für ein bestimmtes Artefakt erfüllt sein müssen, damit dieses als gut dokumentiert im Hinblick auf die Dokumentation von Entscheidungswissen gilt. Artefakte, die diese Voraussetzungen nicht erfüllen, werden im *Rationale Backlog* gelistet. Ein Ziel des Praktikums ist es, dass Codedateien mit unzureichender Rationale Coverage im Rationale Backlog angezeigt werden. Damit festgelegt werden kann, ab wann eine Codedatei als gut dokumentiert gilt, wird eine Konfigurationsmöglichkeit der *Definition of Done* für Codedateien geschaffen.

Weiterhin ist Anforderung des Praktikums, das ConDec-Eclipse-Plug-in wie folgt zu erweitern und zugleich ein ConDec-Plug-in für die IDE *Visual Studio Code* mit ebendieser Funktionalität neu zu erstellen:

R8: Schaffung von Navigationsmöglichkeit zum ConDec-Jira-Plug-in Ist eine Codedatei innerhalb der IDE geöffnet, so soll eine Möglichkeit geschaffen werden, zu einem auf diese Codedatei zentrierten Subgraphen des Wissensgraphens innerhalb des ConDec-Jira-Plug-ins zu navigieren.

Im Übrigen wurden an die Plug-ins die folgenden nichtfunktionalen Anforderungen gestellt:

R9: Erweiterbarkeit, Änderbarkeit, Benutzbarkeit Entwurf und Implementierung sollen leicht erweiterbar, änderbar und benutzbar sein. Neue Implementierungen sollen der bestehenden Implementierung sowie einander möglichst

ähnlich sein. Neuer Code soll mithilfe von Komponententests mit einer Abdeckung von mindestens 85 Prozent getestet werden. Die Plug-ins sollen leicht installierbar sein.

3.2 Detailanforderungen

Aus den in Abschnitt 3.1 formulierten Grobanforderungen wurden Detailanforderungen abgeleitet.

Soweit Detailanforderungen in mehreren Projektdokumentationen (für das Jira-, Eclipse- sowie Visual-Studio-Code-Plug-in) Verwendung finden, sind sie mit wortgleichen Beschreibungen mehrfach in den verschiedenen Projektdokumentationen in Jira angelegt.

3.2.1 User Tasks und Subtasks

Für die Erweiterungen an den ConDec-Plug-ins wurden im Praktikum keine neuen User Tasks oder Subtasks spezifiziert. Stattdessen wurde auf die bereits bestehenden User Tasks und Subtasks im ConDec-Jira-Projekt oder im ConDec-Eclipse-Projekt verwiesen und neu erstellte Wissenselemente mit diesen verlinkt.

Die folgenden User Tasks und Subtasks fanden im Praktikum Verwendung:

UT1: Rationale management

Rationale Manager haben die Aufgabe, den Prozess zum Management von Entscheidungswissen zu definieren sowie das dokumentierte Entscheidungswissen zu prüfen und dessen hohe Qualität zu sicherzustellen.

Dieser User Task sind die folgenden Subtasks zugeordnet:

UT1S1: Set up rationale management process Rationale Manager definieren den Prozess zum Management von Entscheidungswissen. Hierfür spezifizieren sie etwa die zu verwendenden Typen von Entscheidungswissen oder die *Definitions of Done*.

UT1S2: Analyze quality of documented decision knowledge and other knowledge (requirements, code) Rationale Manager prüfen und verbessern die Qualität des dokumentierten (Entscheidungs-)Wissens, sodass dieses eine hohe Qualität etwa hinsichtlich Konsistenz, Vollständigkeit oder Korrektheit aufweist.

UT2: Development of software

Requirements Engineers leiten Anforderungen für eine Software ab, Entwickler:innen implementieren diese Anforderungen. Während dieser Prozesse möchten Requirements Engineers und Entwickler:innen gute Entscheidungen treffen und verwalten dazu Entscheidungswissen, das in einem Wissensgraphen gespeichert und visualisiert wird.

Dieser User Task sind die folgenden Subtasks zugeordnet:

UT2S1: Understand the former decisions and the software evolution Requirements Engineers und Entwickler:innen möchten die im Entwicklungsprozess getroffenen Entscheidungen verstehen und wesentliche Entscheidungen, die die Softwareevolution geprägt haben, kennen, um bei zukünftigen Entscheidungen oder bei Neubeurteilungen früherer Entscheidungen auf Wissen und Erfahrungen aus der Vergangenheit zurückgreifen zu können. Sie möchten ebenso die damit verbundenen Anforderungen, das Entscheidungswissen sowie den Code verstehen.

UT2S2: Analyze impact of change Requirements Engineers und Entwickler:innen möchte die Auswirkungen von Änderungen analysieren, um Entscheidungen konsistent zu vorigen Entscheidungen zu treffen und das dokumentierte Wissen konsistent zu halten.

UT2S3: Continuously document decision knowledge in code comments and commit messages Entwickler:innen setzen Entwicklungsaufgaben in Codedateien um, die in Git-Repositorien organisiert sind. Hierbei dokumentieren sie Entscheidungen und zugehöriges Entscheidungswissen in Commit-Nachrichten und Code-Kommentaren.

3.2.2 Personae

Die in den User Tasks und Subtasks in Unterabschnitt 3.2.1 genannten Rollen werden durch die bereits bestehenden Personas in Tabelle 3.1, Tabelle 3.2 sowie Tabelle 3.3 konkretisiert.

Tabelle 3.1: Beispielhafte Persona für die Rolle „Entwickler:in“.

Name	Lori Bell
Biographie	22 Jahre alt, ausgebildete Software-Entwicklerin. Arbeitet für eine große Softwarefirma, die Anforderungen und Aufgaben seit 2005 in Jira dokumentiert.
Wissen	Benutzt Eclipse, IntelliJ und VS Code als IDE, git als Versionskontrollsystem und Jira als Issue Tracking System. […] Weiß, wie Entscheidungswissen in Commit-Nachrichten und Jira-Issue-Beschreibungen und -Kommentaren explizit dokumentiert wird.
Bedürfnisse	Möchte einfach und leicht Entscheidungswissen ohne Kontextwechsel dokumentieren. Möchte während des Softwareentwicklungsprozesses einfach und leicht die Dokumentationsqualität des Entscheidungswissens prüfen. Möchte einfache, leichte und nicht aufdringliche Kontextwechsel, wenn sie erforderlich sind.
Frustrationen	Kontextwechsel bei der Dokumentation. Über verschiedene Plattformen verteilte Dokumentation.
Ideale Features	Klassifizierung von Entscheidungswissen in natürlicher Sprache (Jira-Issue-Kommentare, -Beschreibungen, Commit-Nachrichten, Codekommentare). Visualisierung von Code, der von einer Entscheidung betroffen ist. Aggregations des Entscheidungswissens an einem Ort. Navigation zwischen Tools mit höchstens zwei Klicks.

Tabelle 3.2: Beispielhafte Persona für die Rolle „Requirements Engineer“.

Name	Phillip Hill
Biographie	40 Jahre alt, männlich, Masterabschluss in Informatik. Arbeitet für eine große Softwarefirma, die Anforderungen und Aufgaben seit 2005 in Jira dokumentiert. Arbeitet meist mit Jira, Word und Excel.
Wissen	Arbeitet seit zwölf Jahren als Requirements Engineer, davon seit acht Jahren in dieser Firma. Benutzt Jira seit 5 Jahren.
Bedürfnisse	Möchte vorige Entscheidungen im Requirements Engineering verstehen. Möchte die Auswirkungen von Änderungen analysieren. Möchte Entscheidungen dokumentieren, die mit der Erhebung und Umsetzung von Anforderungen zusammenhängen. […]
Frustrationen	Lange Ladezeiten. Komplizierte Benutzung (zu viele Klicks für bestimmte Funktionalitäten).
Ideale Features	[…] Einfache Möglichkeit, Entscheidungswissen in Dokumenten (etwa Jira-Issue- oder Code-Kommentaren) zu dokumentieren und dieses Wissen in einen Wissensgraphen zu integrieren. Möglichkeit, Entscheidungswissenselemente in weniger als drei Klicks zu verlinken. Möglichkeit, den Wissensgraphen zu filtern, um nur bestimmte Wissenstypen anzuzeigen.

Tabelle 3.3: Beispielhafte Persona für die Rolle „Rationale Manager“.

Name	Ralph Reed
Biographie	31 Jahre alt, männlich, Masterabschluss in Informatik, Bachelorabschluss in Kommunikationswissenschaften. Arbeitet für eine große Softwarefirma. Ist in viele verschiedene Projekte eingebunden. Benutzt Jira-Dashboards und Git-Konsolen, um alle neuen Jira-Issues und Entscheidungselemente im Blick zu behalten.
Wissen	Benutzt Jira seit 3 Jahren. Betreut viele Projekte als Rationale Manager. Ist ein Experte für kontinuierliches Rationale Management. […]
Bedürfnisse	Möchte den Entscheidungswissens-Managementprozesses im Projekt einfach definieren. Möchte Metriken, die die Qualität der Entscheidungswissensdokumentation beschreiben, einfach erhalten. Möchte Wissenselementen aus Artefakten wie Issue-Kommentaren, Code-Kommentaren sowie Commit-Nachrichten an einem Ort sammeln. Möchte Entwickler:innen eine einfache Weise zur Dokumentation von Entscheidungswissen bereitstellen, sodass sie motiviert sind, ein gutes Wissensmanagement zu betreiben. […]
Frustrationen	Projektbeteiligte, die ihre Entscheidungen nicht dokumentieren. Inkorrektes oder unvollständiges Entscheidungswissen. […]
Ideale Features	Konfigurationsmöglichkeiten für den Entscheidungswissensprozess, hierunter für die Git-Anbindung, des Rationale Backlogs, […]. Jira-Dashboards, um Entscheidungswissenselemente und Metriken zur Dokumentationsqualität zu überwachen. Anzeige und direkte Navigation zu unvollständigen Wissenselementen. […]

3.2.3 Systemfunktionen

Die User Tasks und Subtasks in Unterabschnitt 3.2.1 werden durch die folgenden Systemfunktionen unterstützt. Hierbei wurden die bereits bestehenden Systemfunktionen 1–3 in ihrem Umfang erweitert sowie die übrigen Systemfunktionen neu formuliert.

Eine Zuordnung von Systemfunktionen zu User Tasks und Subtasks zeigt Tabelle 3.4.

Tabelle 3.4: Übersicht über Systemfunktionen und Grobanforderungen innerhalb der User Tasks und Subtasks.

UT1					**UT2**						
UT1S1		**UT1S2**			**UT2S1**				**UT2S2**	**UT2S3**	
SF2 (R1)	SF7 (R7)	SF1 (R5)	SF6 (R6)	SF8 (R7)	SF1 (R5)	SF3 (R4)	SF9 (R8)	SF10 (R8)	SF1 (R5)	SF4 (R2)	SF5 (R3)

SF1: Filter knowledge graph Diese Systemfunktion setzt die Grobanforderung R5 um. Sie liefert zu gegebenen Filterkriterien einen Subgraphen des Wissensgraphen, der nur die Wissenselemente enthält, die den Filterkriterien entsprechen. In diesem Praktikum wurde die Systemfunktion um die Möglichkeit erweitert, im gefilterten Subgraphen transitive Links zu generieren.

Tabelle 3.5: SF1: Filter knowledge graph.

Vorbedingung	Jira-Projekt existiert und ConDec-Plug-in ist für dieses aktiviert. Wissensgraph mit Wissenselementen und Verlinkungen existiert.
Eingabe	WS1.3 oder WS1.4: Filterkriterien, Wissensgraph.
Nachbedingung	*keine Änderungen*
Ausgabe	WS1.3 oder WS1.4: Wissenselemente und Verlinkungen, auf die die Filterkriterien zutreffen, in gefiltertem Subgraph.
Ausnahme	*keine*
Regeln	Filterkriterien: […] transitive Links sollen erstellt werden (standardmäßig nein, nur auswählbar, wenn Wurzelelement verfügbar ist, nicht bei Anzeige des globalen Wissensgraphen); […]

SF2: Configure decision knowledge extraction from git (commit messages and code comments) Diese Systemfunktion setzt die Grobanforderung R1 um. Sie erhält von Nutzer:innen Zugangsdaten für Git-Repositorien und weitere Einstellungen im Zusammenhang mit der Extraktion von Wissen aus Git-Repositorien. Diese speichert sie für das jeweilige Jira-Projekt und überschreibt dabei bestehende Einstellungen.

Tabelle 3.6: SF2: Configure decision knowledge extraction from git (commit messages and code comments)

Vorbedingung	Jira-Projekt existiert und ConDec-Plug-in ist für dieses aktiviert. Git-Repositorium existiert.
Eingabe	WS1.2: Projektschlüssel, Aktivierungsstatus der Wissensextraktion aus Git, URIs der Git-Repositorien sowie zugehörige Authentifizierungsdaten, […]
Nachbedingung	Projekteinstellungen sind aktualisiert. Repositorien werden geklont.
Ausgabe	WS1.2: Bestätigungsnachricht “Git configuration has been changed”.
Ausnahme	Das Klonen der Repositorien schlägt fehl.
Regeln	Wenn das Klonen der Repositorien fehlschlägt, wird die Wissensextraktion aus Git deaktiviert.

SF3: List all code classes for a project Diese Systemfunktion setzt die Grobanforderung R4 um. Sie zeigt sämtliche Code-Klassen an, die aus Git in den Wissensgraphen eines Projekts integriert wurde.

Tabelle 3.7: SF3: List all code classes for a project

Vorbedingung	Jira-Projekt existiert und ConDec-Plug-in ist für dieses aktiviert. Git-Anbindung ist konfiguriert und aktiviert.
Eingabe	WS1.3.2: Projektschlüssel, Auswahl Wissenstyp: “Code”.
Nachbedingung	*keine Änderungen*
Ausgabe	WS1.3.2: Liste aller Codedateien.
Ausnahme	*keine*

SF4: Automatically add code files from git repository into the knowledge graph Diese Systemfunktion setzt die Grobanforderung R2 um. Sie liest aus

den in den Einstellungen hinzugefügten Git-Repositorien (vgl. SF2) Codedateien aus und integriert diese als Wissenselemente in den Wissensgraphen.

Tabelle 3.8: SF4: Automatically add code files from git repository into the knowledge graph

Vorbedingung	Jira-Projekt existiert und ConDec-Plug-in ist für dieses aktiviert. Git-Anbindung ist konfiguriert und aktiviert.
Eingabe	Liste an Codedateien im Repositorium, Dateiendungen von Codedateien.
Nachbedingung	Codedateien sind in den Wissensgraphen eingebunden und verlinkt.
Ausgabe	WS1.3.2, WS1.3.3, WS1.3.6 oder WS1.4.2: Codedateien werden in der entsprechen Visualisierung des Wissensgraphen angezeigt.
Ausnahme	*keine*
Regeln	Die Systemfunktion wird jedes Mal aufgerufen, wenn das Git-Repositorium auf dem Jira-Server aktualisiert wird.

SF5: Automatically add and link decision knowledge from comments in code files into the knowledge graph Diese Systemfunktion setzt die Grobanforderung R3 um. Sie liest aus den Codedateien im Wissensgraphen (vgl. SF4) Entscheidungswissen aus, das in Code-Kommentaren notiert ist. Diese werden anschließend als Wissenselemente in den Wissensgraphen integriert.

Tabelle 3.9: SF5: Automatically add and link decision knowledge from comments in code files into the knowledge graph

Vorbedingung	Jira-Projekt existiert und ConDec-Plug-in ist für dieses aktiviert. Git-Anbindung ist konfiguriert und aktiviert. Codedateien sind als Wissenselemente Teil des Wissensgraphen und mit Work-Items verlinkt.
Eingabe	Codedateien im Repositorium mit darin dokumentiertem Entscheidungswissen, Zuordnung von Dateiendungen zu Codekommentar-Stilen.
Nachbedingung	Codedateien sind in den Wissensgraphen eingebunden und verlinkt.
Ausgabe	WS1.3.2, WS1.3.3 oder WS1.3.6: Entscheidungswissen wird in der entsprechen Visualisierung des Wissensgraphen angezeigt.
Ausnahme	*keine*
Regeln	Entscheidungswissen in Codekommentaren soll im Stil von Javadoc formatiert werden. Die Systemfunktion wird jedes Mal aufgerufen, wenn das Git-Repositorium auf dem Jira-Server aktualisiert wird.

SF6: Show knowledge graph metrics with respect to code files Diese Systemfunktion setzt die Grobanforderung R6 um. Sie liest Code-Dateien, die in den Wissensgraphen integriert wurden (vgl. SF4), sowie Entscheidungswissenselemente im Wissensgraphen und visualisiert aus diesen Rohdaten Metriken mithilfe von Tortendiagrammen und Boxplots. Es wird angezeigt, wie viele Entscheidungen oder Entscheidungsprobleme mit Codedateien über eine gewisse Linkdistanz verlinkt sind.

Tabelle 3.10: SF6: Show knowledge graph metrics with respect to code files

Vorbedingung	Jira-Projekt existiert und ConDec-Plug-in ist für dieses aktiviert. Git-Anbindung ist konfiguriert und aktiviert. Entscheidungswissenselemente existieren im Wissensgraphen.
Eingabe	WS1.5.3: Entscheidungswissenselemente und Codedateien aus dem Wissensgraphen.
Nachbedingung	*keine Änderungen*
Ausgabe	WS1.5.3: Metriken zu Entscheidungswissen im Zusammenhang mit Codedateien sind mithilfe von Diagrammen visualisiert.
Ausnahme	*keine*
Regeln	Es wird die Rationale Coverage von Codedateien bezüglich Entscheidungen sowie Entscheidungsproblemen über eine bestimmte Linkdistanz angezeigt.

SF7: Configure definition of done (DoD) for the decision knowledge documentation Diese Systemfunktion setzt die Grobanforderung R7 um. Sie erhält von Nutzer:innen Kriterien für die *Definition of Done* von Wissenselementen eines bestimmten Typs und speichert diese für das jeweilige Jira-Projekt, wobei bestehende Einstellungen überschrieben werden.

Tabelle 3.11: SF7: Configure definition of done (DoD) for the decision knowledge documentation

Vorbedingung	Jira-Projekt existiert und ConDec-Plug-in ist für dieses aktiviert. Codedateien sind in den Wissensgraphen eingebunden und verlinkt.
Eingabe	WS1.2: Parameter für die *Definition of Done*.
Nachbedingung	Eingegebene Parameter sind gespeichert.
Ausgabe	Bestätigungsnachricht "Eingaben gespeichert".
Ausnahme	*keine*
Regeln	Die *Definition of Done* für Codedateien ist erfüllt wenn die Codedatei eine Testklasse ist, weniger als eine vorgegebene Zahl an Codezeilen enthält, oder über eine vorgegebene Linkdistanz mit mindestens einer Entscheidung im Wissensgraphen verbunden ist.

SF8: Check definition of done (DoD) of the decision knowledge documentation related to a code file Diese Systemfunktion setzt die Grobanforderung R7 um. Sie liest Code-Dateien, die in den Wissensgraphen integriert wurden (vgl. SF4), und zeigt diejenigen davon an, die die *Definition of Done* für Codedateien (vgl. SF7) nicht erfüllen.

Tabelle 3.12: SF8: Check definition of done (DoD) of the decision knowledge documentation related to a code file

Vorbedingung	Jira-Projekt existiert und ConDec-Plug-in ist für dieses aktiviert. Codedateien sind in den Wissensgraphen eingebunden und verlinkt.
Eingabe	WS1.3.1: Rationale-Backlog-Ansicht ist ausgewählt, Filter enthält den Wissenstyp “Code”.
Nachbedingung	*keine Änderungen*
Ausgabe	WS1.3.1: Alle Codedateien, die die *Definition of Done* nicht erfüllen, werden angezeigt.
Ausnahme	*keine*
Regeln	Die *Definition of Done* für Codedateien ist erfüllt wenn die Codedatei eine Testklasse ist, weniger als eine vorgegebene Zahl an Codezeilen enthält, oder über eine vorgegebene Linkdistanz mit mindestens einer Entscheidung im Wissensgraphen verbunden ist.

SF9: Navigate to subgraph centered on a code file in Jira Diese Systemfunktion setzt die Grobanforderung R8 um. Sie liest den Namen der in einer IDE geöffneten oder ausgewählten Codedatei sowie die Jira-URL und den Jira-Projektidentifier aus der IDE-Konfiguration und navigiert mithilfe dessen zu einem Subgraphen des Wissensgraphens in Jira, der auf die ausgewählte Codedatei zentriert ist.

Tabelle 3.13: SF9: Navigate to subgraph centered on a code file in Jira in Eclipse

Vorbedingung	Eclipse IDE ist geöffnet. Codedatei ist in IDE geöffnet. ConDec Eclipse ist mit dem Jira-Projekt verknüpft. Jira-Projekt existiert und ConDec-Plug-in ist für dieses aktiviert. Codedateien sind in den Wissensgraphen eingebunden und verlinkt.
Eingabe	WS2.2: Codedatei
Nachbedingung	*keine Änderungen*
Ausgabe	WS1.3.2: Codedatei ist ausgewählt, Subgraph der Codedatei wird angezeigt.
Ausnahme	*keine*

Tabelle 3.14: SF9: Navigate to subgraph centered on a code file in Jira in Visual Studio Code

Vorbedingung	Visual Studio Code IDE ist geöffnet. Codedatei ist in IDE geöffnet. ConDec Visual Studio Code ist aktiviert. Jira-Projekt existiert und ConDec-Plug-in ist für dieses aktiviert. Codedateien sind in den Wissensgraphen eingebunden und verlinkt.
Eingabe	WS6.1: Codedatei
Nachbedingung	*keine Änderungen*
Ausgabe	WS1.3.2: Codedatei ist ausgewählt, Subgraph der Codedatei wird angezeigt.
Ausnahme	Jira-Projektinformationen fehlen
Regeln	Wenn Jira-Projektinformationen (Server-URL, Projektschlüssel) nicht in den Arbeitsbereicheinstellungen gespeichert sind, werden Nutzer:innen gebeten, diese einzugeben. Wenn die Informationen eingegeben werden, werden sie in den Arbeitsbereicheinstellungen gespeichert.

SF10: Configure Jira Settings [in VS Code] Diese Systemfunktion setzt die Grobanforderung R8 um. Sie erhält von Nutzer:innen die URL sowie den Projektidentifier zu einem Jira-Projekt und speichert diese in den Arbeitsbereich-Einstellungen der IDE, wobei bestehende Einstellungen überschrieben werden.

Tabelle 3.15: SF10: Configure Jira Settings [in VS Code]

Vorbedingung	Visual Studio Code IDE ist geöffnet. ConDec Visual Studio Code ist aktiviert für den geöffneten Arbeitsbereich.
Eingabe	WS6.1 oder WS6.2: Jira-Projektinformationen (Server-URL, Projektschlüssel)
Nachbedingung	Jira-Projektinformationen werden in den Arbeitsbereicheinstellungen gespeichert.
Ausgabe	Wenn von WS6.1 aufgerufen: Pop-up-Information: “You can always change the Jira project information in the workspace settings.”
Ausnahme	Wenn von WS6.1 aufgerufen: Leere Eingabe führt zu Fehlermeldung: “The Jira project information is not specified. Please provide the Jira project information. You can always change the Jira project information in the workspace settings.”

3.2.4 Arbeitsbereiche

Die ConDec-Plug-ins für Jira, Eclipse und Visual Studio Code enthalten verschiedene Arbeitsbereiche, von denen die folgenden erweitert sowie im Fall von ConDec Visual Studio Code neu erstellt wurden.

Die Zuordnung von Systemfunktionen zu den Arbeitsbereichen ist in den UI-Strukturdiagrammen in Abbildung 3.1, Abbildung 3.2 sowie Abbildung 3.3 dargestellt.

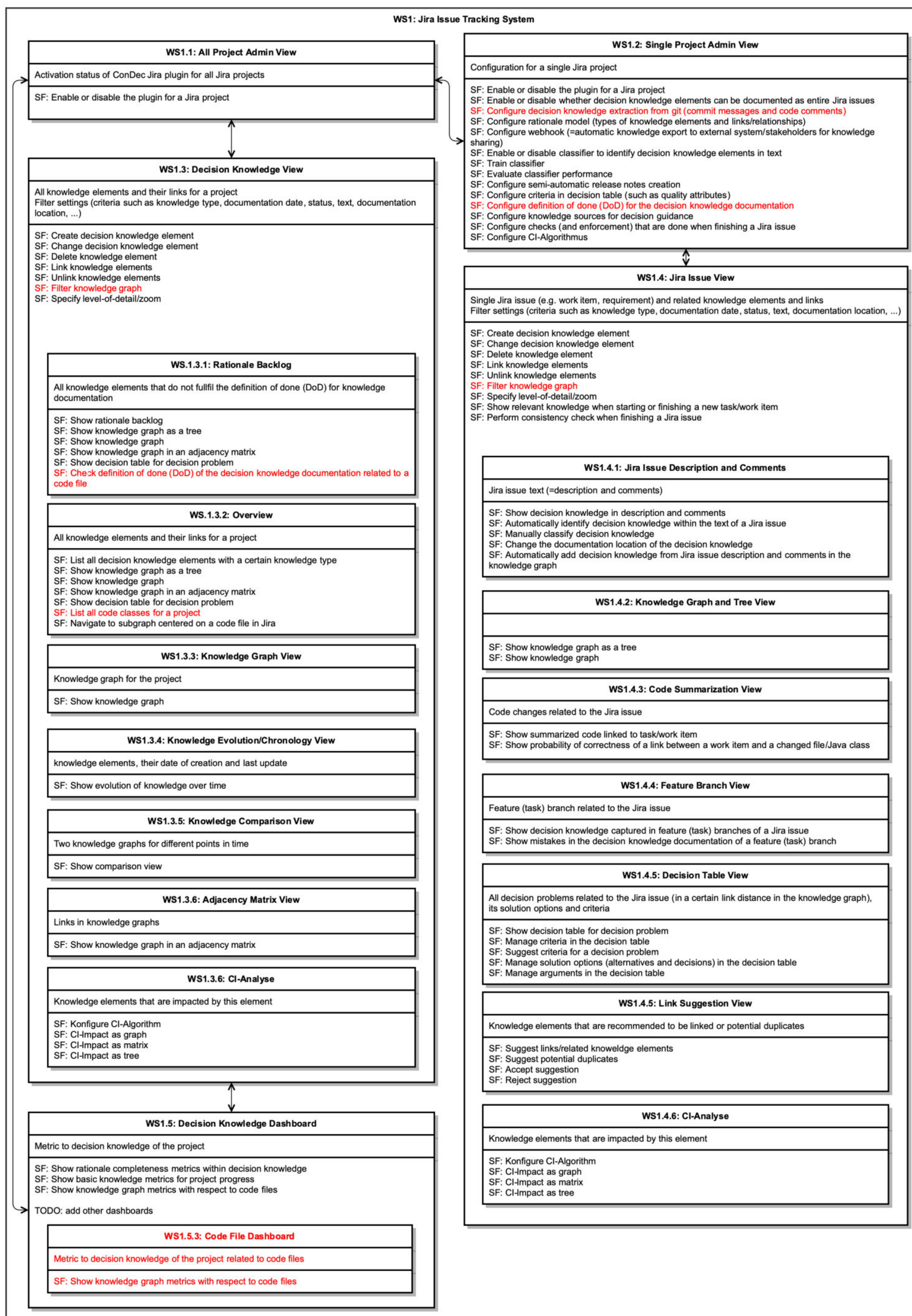

Abbildung 3.1: UI-Strukturdiagramm für ConDec Jira. Rot markierte Elemente wurden im Praktikum hinzugefügt.

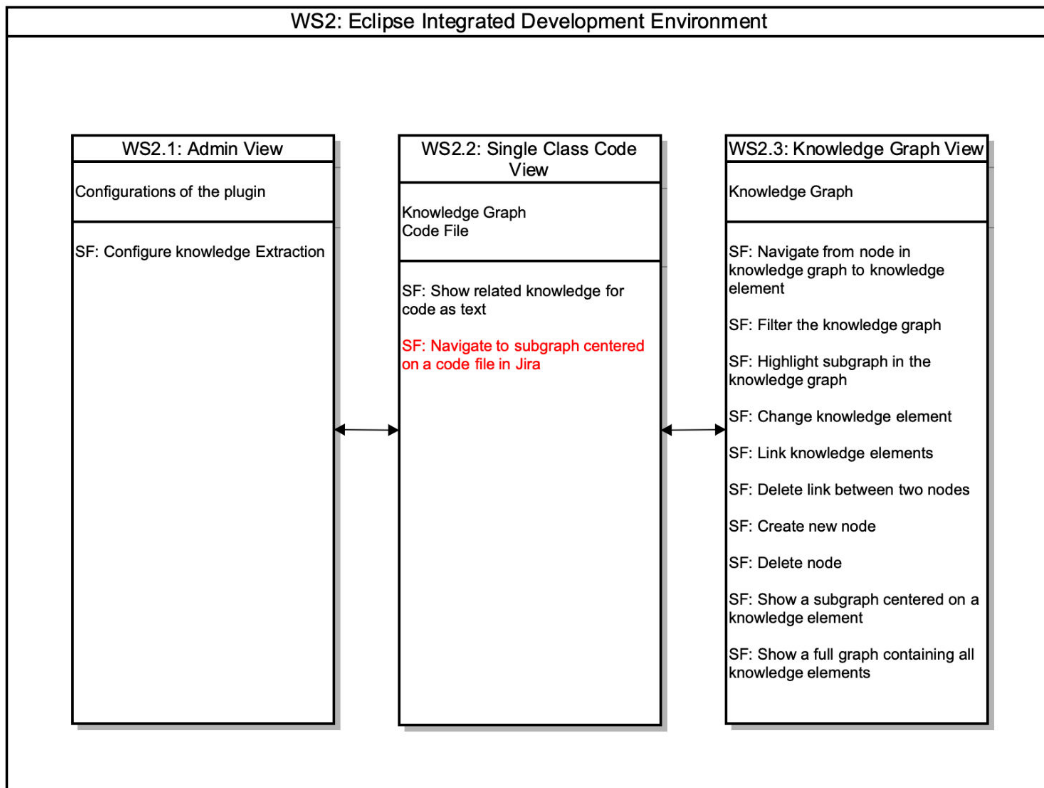

Abbildung 3.2: UI-Strukturdiagramm für ConDec Eclipse. Rot markierte Elemente wurden im Praktikum hinzugefügt.

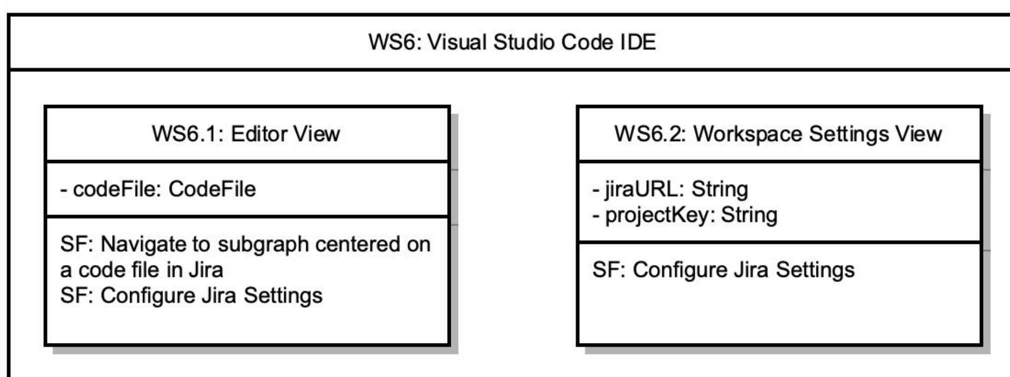

Abbildung 3.3: UI-Strukturdiagramm für das neu erstellte Plug-in ConDec Visual Studio Code.

ConDec Jira

WS1.2: Single Project Admin View In diesem Arbeitsbereich werden projektspezifische Einstellungen für ConDec verwaltet. Beim Aufruf werden die bestehenden Einstellungen angezeigt. Nutzer:innen können hier neue Einstellungen eintragen und diese speichern. Im Praktikum sind hier insbesondere die Einstellungen zur Git-Anbindung sowie zur *Definition of Done* relevant.

WS1.3: Decision Knowledge View Dieser Arbeitsbereich zeigt den gesamten Wissensgraphen in einem Projekt. Zur Visualisierung werden verschiedene Frameworks verwendet, wobei jede Art der Visualisierung einem eigenen untergeordneten Arbeitsbereich entspricht. Hervorzuheben sind hier die Anzeige des Rationale Backlogs im Arbeitsbereich **WS1.3.1: Rationale Backlog/Backlog Items Overview**, die Überblicksansicht **WS1.3.2: Decision Knowledge Overview**, die Graphansicht **WS1.3.3: Knowledge Graph View** sowie die Adjazenzmatrixansicht **WS1.3.6: Adjacency Matrix View**.

WS1.4: Jira Issue View Dieser Arbeitsbereich zeigt die Detailansicht eines Jira-Issues. Innerhalb dieser bestehen wieder untergeordnete Arbeitsbereiche, die einen auf das jeweilige Jira-Issue zentrierten Subgraphen des Wissensgraphen mithilfe verschiedene Frameworks visualisieren. Nennenswert ist hier insbesondere die Baum- und Graphansicht **WS1.4.2: Knowledge Graph (and Tree) View**.

WS1.5: Jira Dashboards Dieser Arbeitsbereich erlaubt es, Statistiken zu Entscheidungswissen anzuzeigen, das in einem Projekt dokumentiert ist. Hier wurde im Praktikum der untergeordnete Arbeitsbereich **WS1.5.3: Code File Dashboard** neu angelegt, der visualisiert, wie viele Entscheidungen oder Entscheidungsprobleme mit Codedateien über eine gewisse Linkdistanz verlinkt sind.

ConDec Eclipse

WS2.2: Single Class Code View Dieser Arbeitsbereich zeigt Codedateien innerhalb der IDE *Eclipse* an. Ausgehend von hier lässt sich zur Anzeige des

Wissensgraphen in Jira navigieren, der auf die in Eclipse ausgewählte Codedatei zentriert ist.

ConDec Visual Studio Code

WS6.1: Editor View Dieser Arbeitsbereich zeigt Codedateien innerhalb der IDE *Visual Studio Code* an. Ausgehend von hier lässt sich zur Anzeige des Wissensgraphen in Jira navigieren, der auf die in Eclipse ausgewählte Codedatei zentriert ist.

WS6.2: Workspace Settings View In diesem Arbeitsbereich werden projektspezifische Einstellungen für ConDec verwaltet. Beim Aufruf werden die bestehenden Einstellungen angezeigt. Nutzer:innen können hier neue Einstellungen eintragen und diese speichern. Diese umfassen die URL sowie den Projektidentifier zu einem Jira-Projekt.

3.2.5 Domänendaten

Das in Abbildung 3.4 gezeigte Domänendatendiagramm zeigt die Entitäten und ihre Zusammenhänge innerhalb der ConDec-Projekte. Es wurde im Praktikum geringfügig ergänzt. Folgend aufgeführte Entitäten und Zusammenhänge sind hervorzuheben:

Zwei Wissenselemente (*Knowledge Element*s) können zueinander in Beziehung (*Relationship*) stehen, wobei die Beziehung einem Beziehungstyp (*Relationship Type*) zugeordnet werden kann. Im Praktikum wurde hier der Beziehungstyp „TRANSITIVE“ ergänzt.

Es gibt verschiedene Arten von Wissenselementen. Hierzu zählen Elemente des Systemwissens (*System Knowledge Element*s) oder Elemente des Entscheidungswissens (*Decision Knowledge Element*s). Im Praktikum relevante Elemente des Systemwissens sind Codedateien. Die im Praktikum betrachteten Elemente des Entscheidungswissens sind die in Abschnitt 2.1 beschriebenen.

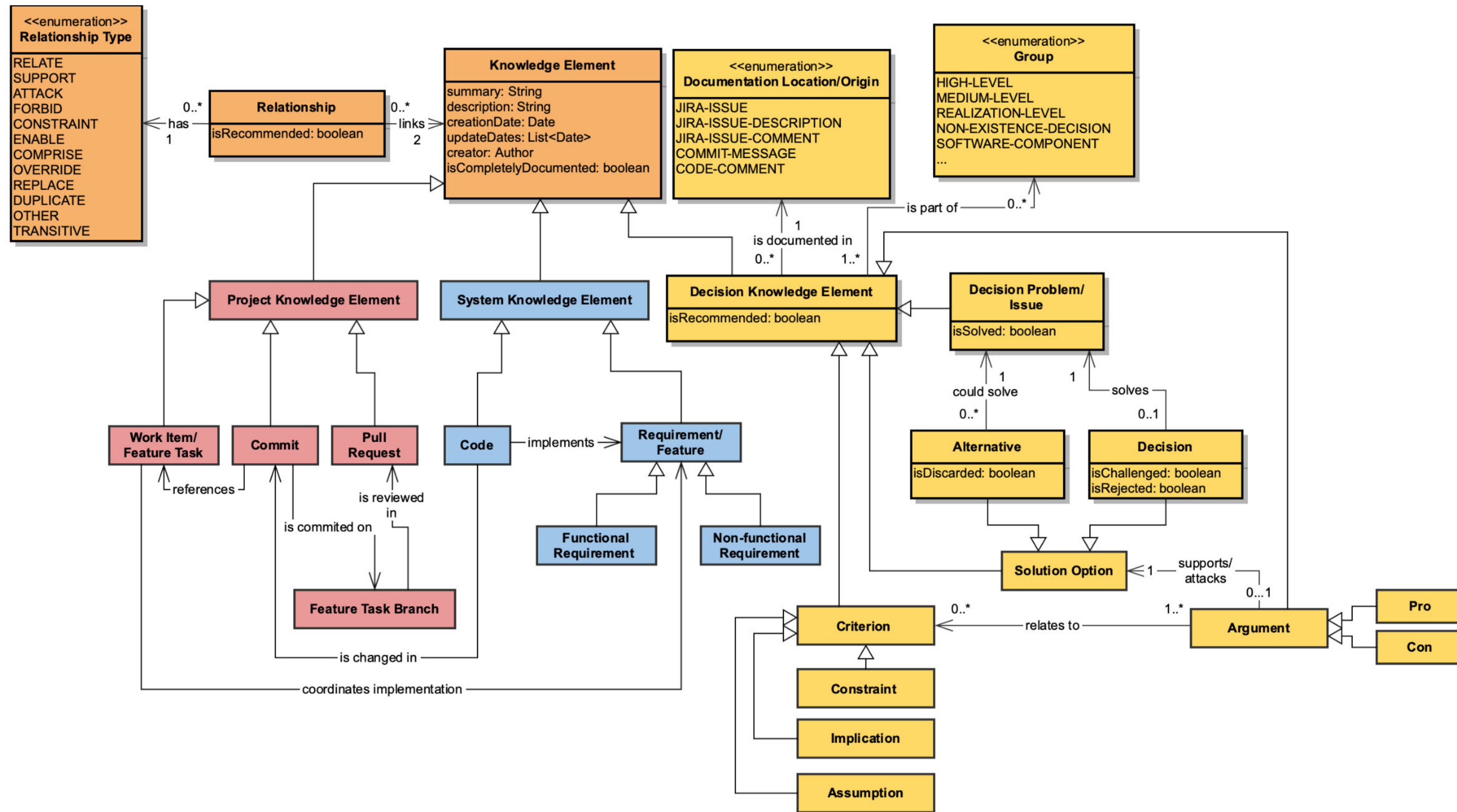

Abbildung 3.4: Domänendatendiagramm der ConDec-Plug-ins.

Entscheidungswissenselemente sind an einem Dokumentationsort (*Documentation Location/Origin*) dokumentiert. Im Praktikum war insbesondere der Dokumentationsort „CODE-COMMENT“ relevant.

3.2.6 Nichtfunktionale Anforderungen

Die an das Praktikum gestellten nichtfunktionalen Anforderungen finden sich in den folgenden bereits im Jira-Projekt des ConDec-Jira-Plug-ins dokumentierten nichtfunktionalen Anforderungen wieder.

NFR: Portability Die Software soll unter anderem an neue Bedürfnisse leicht anpassbar sein. Sie soll außerdem einfach zu installieren sein.

NFR: Maintainability Die Software soll einfach zu ändern sein. Zur Beurteilung dessen können Metriken wie Komplexität oder die Einhaltung von Code Styles herangezogen werden. Hierfür sind statische Codeanalysen geeignet.

NFR: Usability Die Software soll einfach zu verstehen sein und möglichst wenige Benutzeranleitungen erfordern. Sie soll weiterhin leicht zu erlernen und zu verwenden sein. Schließlich soll die Software ansprechend aussehen.

Kapitel 4

Entwurf und Implementierung

Auf der Grundlage der Anforderungen in Kapitel 3 wurden die Erweiterungen der ConDec-Plug-ins wie nachfolgend beschrieben entworfen und sogleich implementiert. Beides wird gemeinsam in diesem Kapitel beschrieben.

4.1 SF1: Filter knowledge graph

Für die Umsetzung dieser Systemfunktion muss zunächst entschieden werden, ob und inwieweit die Richtung von Verlinkungen Einfluss auf die Erstellung transitiver Links haben soll.

Wird die Richtung von Verlinkungen berücksichtigt, so ist insbesondere beim Linktyp "is related to" bei der Verlinkung von Jira-Issues oft nicht erkennbar, in welche Richtung die Verlinkung aufgebaut wird. Entsprechend ist es für Nutzer:innen schwierig, die Richtung von Verlinkungen wie gewünscht festzulegen.

Wird hingegen die Richtung von Verlinkungen berücksicht, so werden verschiedentlich viele unnütze Links erstellt. So würden in einem Wissensgraphen, in dem mehrere Codedateien sowie mehrere Entscheidungprobleme mit einer Entwicklungsaufgabe verlinkt sind, bei Herausfilterung von Wissenselementen des Typs „Entwicklungsaufgabe" alle Codedateien sowie alle Entscheidungs-

probleme miteinander verlinkt werden. Derartige Links dürften jedoch gemeinhin keinen Nutzen haben.

Als Kompromisslösung wurde innerhalb der Klasse `FilteringManager` die Funktion `transitiveLinkShallBeAdded(…)` implementiert. Sie gibt zu zwei Wissenselementen, die beim Entfernen eines weiteren Wissenselement möglicherweise verlinkt werden sollen, den Wahrheitswert `true` zurück, wenn alle folgend genannten Bedingungen *nicht* zutreffen:

1. beide Wissenselemente sind identisch oder bereits zueinander verlinkt;
2. beide Wissenselemente sind Entscheidungswissenselemente und es ist nicht das erste Wissenselement Startelement[1] des entfernten Wissenselements sowie das zweite Wissenselement Zielelement[2] des entfernten Wissenselements;
3. das erste Wissenselement ist kein Entscheidungswissenselement, das zweite Wissenselement ist ein Entscheidungswissenselement;
4. beide Wissenselemente haben zum Wurzelelement des Subgraphen dieselbe Distanz.

Ist eine der genannten Bedingungen erfüllt, wird der Wahrheitswert `false` zurückgegeben. Bedingung 4 erfordert, dass im Graphen ein Wurzelelement bestimmt ist. Somit können transitive Links mit dem implementierten Algorithmus nur im Kontext eines Subgraphen des Wissensgraphens erzeugt werden, der auf ein bestimmtes Wissenselement zentriert ist.

Insgesamt führt die Implementierung dazu, dass die Richtung von Verlinkungen genau dann berücksichtigt wird, wenn mindestens eines der zu verlinkenden Wissenselemente ein Entscheidungswissenselement ist. In diesen Fällen werden die Links automatisch von ConDec und nicht von Nutzer:innen erstellt, somit ist auch ihre Richtung bekannt. Links zwischen einem Entscheidungswissenselement und einem anderen Wissenselement werden dabei mit dem Entscheidungswissenselement als Start- und dem anderen Wissenselement als

[1] Führt in einem Graphen eine gerichtete Kante von A nach B, so wird A als Startelement bezeichnet.

[2] Führt in einem Graphen eine gerichtete Kante von A nach B, so wird B als Zielelement bezeichnet.

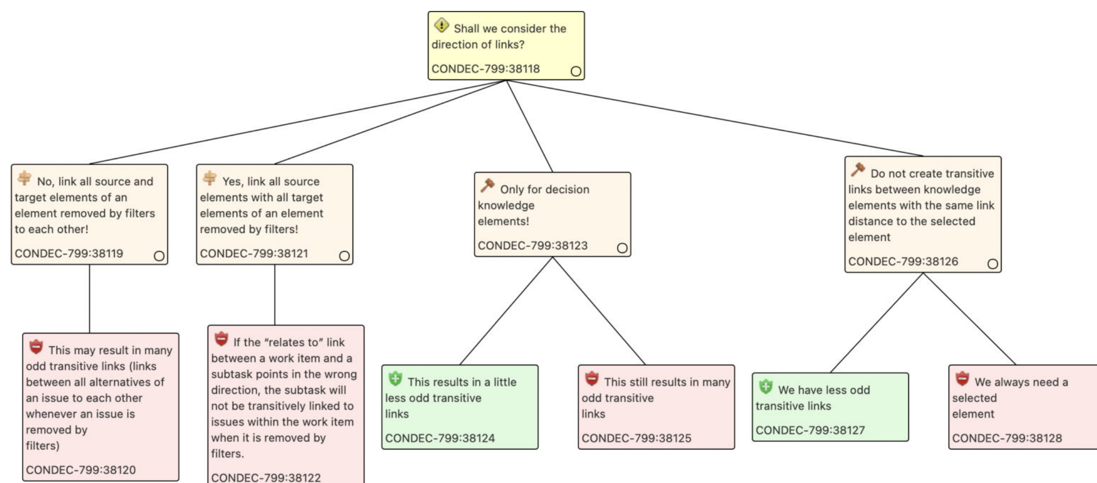

Abbildung 4.1: Entscheidungsbaum für Entscheidungsproblem zur Beachtung der Richtung von Links.

Zielelement erstellt. Weiterhin wird wie beschrieben ein transitiver Link nicht erstellt, wenn beide Wissenselemente zum Wurzelelement des Subgraphen dieselbe Distanz haben. Dies führt dazu, dass das oben beschriebene Problem, bei dem sehr viele Links ohne tatsächlichen Nutzen erstellt würden, nicht auftritt. Das Entscheidungsproblem ist in Abbildung 4.1 als Entscheidungsbaum dargestellt.

Zum Erstellen der transitiven Links wurde innerhalb des `FilteringManager`s die Funktion `addTransitiveLinksToSubgraph(...)` erstellt. In dieser wird zunächst ein temporärer Graph erstellt, der als Zwischenspeicher für neu erstellte transitive Links dient. Dies ist erforderlich, damit im beispielhaften Wissensgraphen aus den Elementen

$$A \to B \to C \to D,$$

wenn B und C durch Filtereinstellungen entfernt werden, trotzdem ein transitiver Link zwischen A und D erstellt wird.

Nun werden mithilfe der Funktion `getElementsNotMatchingFilterSettings()` sämtliche Wissenselemente, die gemäß der eingestellten Filterkriterien aus dem Wissensgraphen entfernt werden sollen, bestimmt und in einer Menge

gespeichert. Anschließend wird über jedes dieser zu entfernenden Elemente iteriert.

Zu jedem zu entfernenden Wissenselement werden die zu ihm verlinken Elemente ermittelt, wobei auch Links aus dem temporären Graphen, mithin frisch erstellte transitive Links, berücksichtigt werden. Anschließend wird über jedes mögliche Tupel aus verlinkten Wissenselemente iteriert.

Dabei wird zunächst geprüft, ob innerhalb der Funktion bereits die Linkdistanz von den Wissenselementen des Tupels zum Wurzelelement berechnet und in einer Zuordnungstabelle gespeichert wurde, was zur Prüfung der oben genannten Bedingung 4 erforderlich ist. Ist eine Linkdistanz noch nicht berechnet, so wird diese berechnet und in der genannten Zuordnungstabelle von Wissenselement zu Linkdistanz gespeichert. Dies reduziert den Berechnungsaufwand der Funktion und erhöht ihre Performance enorm, da das Berechnen von Linkdistanzen in der vorliegenden Implementierung des Wissensgraphen von ConDec sehr rechenaufwändig ist. Schließlich wird mithilfe der oben beschriebenen Funktion `transitiveLinkShallBeAdded(...)` festgestellt, ob ein transitiver Link vom ersten zum zweiten Wissenselement des Tupels erstellt werden soll. Ist das der Fall, so wird der Link unter Verwendung des Linktyps `TRANSITIVE` erstellt und in den Subgraphen des Wissensgraphen eingefügt. Da die Links nicht in den (persistenten) Wissensgraphen eingefügt werden, sondern nur in einen nicht persistenten Subgraphen, müssen transitive Links bei Änderung der Filtereinstellung nicht wieder aus dem Wissensgraphen entfernt werden; sie müssen jedoch bei jeder Änderung der Filtereinstellung neu erzeugt werden.

Um das Erzeugen transitiver Links aktivieren oder deaktivieren zu können, wurde in der Klasse `FilterSettings` das Wahrheitswertsattribut `createTransitiveLinks` ergänzt, welches über die Benutzeroberfläche mithilfe einer Auswahlbox verändert werden kann und standardmäßig den Wert `false` trägt. Dieses Attribut wird im `FilteringManager` in der Funktion `getSubgraphMatchingFilterSettings()` ausgelesen, ist dessen Wert `true`, so wird die beschriebene Funktion `addTransitiveLinksToSubgraph(...)` aufgerufen.

Der Entwurf der beschriebenen Implementierung ist im Klassendiagramm in Abbildung 4.2 gezeigt.

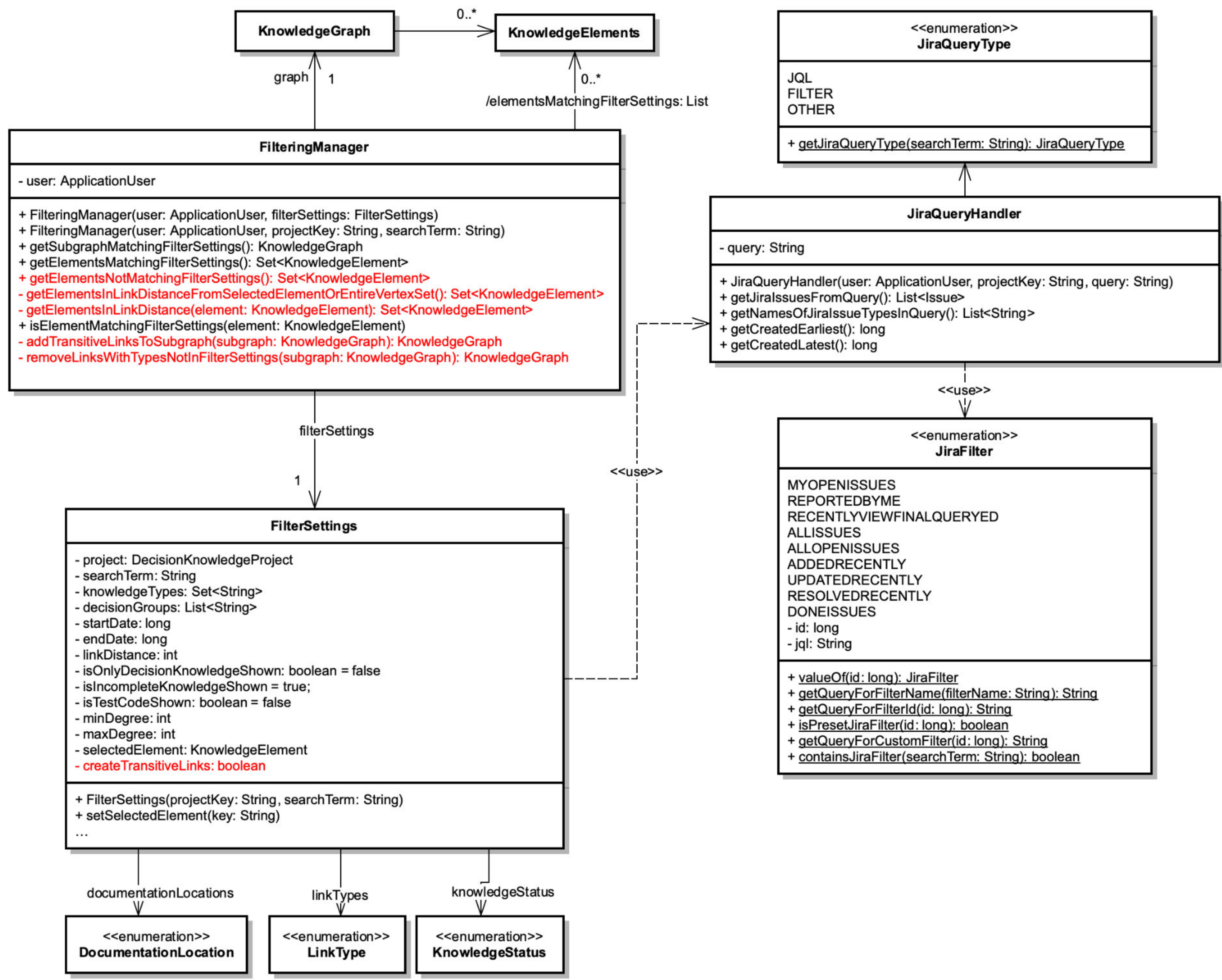

Abbildung 4.2: Klassendiagramm der Wissensgraph-Filterung von ConDec Jira.

- Which authentication methods shall be implemented?
  - Use GitHub OAuth access token.
    - GitHub Token authentication is easy to implement.
    - Only supported by GitHub.
    - Personal GitHub credentials do not need to be stored in Jira.
  - Use GitLab Personal Access Token.
    - GitLab Token authentication is easy to implement.
    - Only supported by GitLab.
    - Personal GitLab credentials do not need to be stored in Jira.
  - Use HTTPS authentication with credentials (username+password).
    - Appears to be supported by all major git services (including GitHub, GitLab, gitolite).
    - HTTPS authentication is easy to implement.
    - Personal HTTPS credentials need to be stored in Jira, or separate git user account needs to be set up.
  - Use SSH authentication with credentials (username+password).
    - Personal SSH credentials need to be stored in Jira, or separate git user account needs to be set up.
    - SSH authentication with credentials is difficult to implement.
  - Use SSH public key authentication.
    - Difficult to set up (SSH key must be generated, uploaded to git and Jira).
    - Personal credentials do not need to be stored in Jira for SSH public key authentication.
    - SSH public key authentication is difficult to implement.

Abbildung 4.3: Entscheidungsbaum für Entscheidungsproblem zur Wahl der unterstützten Authentifizierungsmethoden.

4.2 SF2: Configure decision knowledge extraction from git (commit messages and code comments)

Für die Umsetzung dieser Systemfunktion standen zunächst verschiedene Authentifizierungsmöglichkeiten zur Auswahl; über SSH mithilfe von SSH-Schlüsseldateien oder Anmeldedaten aus Benutzername und Passwort, über HTTPS mit Anmeldedaten aus Benutzername und Passwort, mit GitHub OAuth Access Token oder mit GitLab Personal Access Token. Aufgrund des geringeren Implementierungsaufwands und der breiten Kompatibilität über viele verschiedene Git-Server hinweg wurde zunächst entschieden, nur HTTPS-Varianten, nicht jedoch SSH-Varianten, zu implementieren. Um bei den weit verbreiteten Git-Servern GitHub und GitLab zusätzlich zu ermöglichen, allein projektbezogene Tokens und nicht personenbezogene Tokens wie Passwörter zu hinterlegen, wurde zusätzlich entschieden, neben der Anmeldung mit Benutzername und Passwort auch die Access Tokens von GitHub und GitLab zu unterstützen. Das Entscheidungsproblem ist in Abbildung 4.3 als Entscheidungsbaum dargestellt.

Nach Festlegung dieser Authentifizierungsmethoden wurde der Aufzählungstyp `AuthMethod` definiert, der die Werte `NONE` (für Git-Repositorien ohne Passwort), `HTTP` (für HTTPS-Authentifizierung über Benutzername und Passwort), `GITHUB` (für GitHub OAuth Access Token) und `GITLAB` (für GitLab Personal Access Token) annehmen kann. Die Implementierung als Aufzählungstyp erlaubt hier ein Hinzufügen weiterer Authentifizierungsmethoden in der Zukunft und führt zur einfachen Änderbarkeit der Software.

Um die Anmeldedaten zu speichern, wurde die Klasse `GitRepositoryConfiguration` implementiert, die die Repositoriums-URL, den standardmäßig auszulesenden Zweig, die Authentifizierungsmethode (vom Typ `AuthMethod`), den Benutzernamen sowie das Token (Passwort oder Access Token) enthält. Eine Methode `getCredentialsProvider()` gibt einen `UsernamePasswordCredentialsProvider` aus der Bibliothek *JGit* zurück, die in ConDec zum Auslesen von Git-Repositorien verwendet wird.

Eine Instanz der Klasse `GitRepositoryConfiguration` wird in der bestehenden Klasse `GitClientForSingleRepository` verwaltet und zum Zugriff auf das Repositorium verwendet. Die bestehende Klasse `ConfigPersistenceManager` speichert zu jedem von ConDec verwalteten Projekt die zugehörigen `GitRepositoryConfiguration`s und sorgt damit für die Persistenz der Authentifizierungsdaten.

Um mit der Jira-Benutzeroberfläche kommunizieren zu können, wurde die `ConfigRest`-Schnittstelle um den Endpunkt `/setGitRepositoryConfigurations` erweitert, die empfangene Authentifizierungsdaten mithilfe des `ConfigPersistenceManager`s speichert.

Die Jira-Benutzeroberfläche zur Verwaltung der Git-Repositorien wurde wie in Abbildung 4.4 gezeigt um Eingabemöglichkeiten von Authentifizierungsdaten erweitert. Hierbei zeigt eine Auswahlliste zunächst alle verfügbaren Authentifizierungsmethoden an. Wird eine Eingabe getroffen, so werden abhängig von der Auswahl die Felder zur Eingabe von Benutzername oder Passwort bzw. Token angezeigt oder ausgeblendet.

Der Entwurf der beschriebenen Implementierung ist im Klassendiagramm in Abbildung 4.5 gezeigt.

Continuous Management of Decision Knowledge (ConDec)

Decision Knowledge Extraction from Git

Extract from Git?

Enables or disables whether decision knowledge is extracted from git for this project. When enabled, all code files are extracted and decision knowledge elements from code comments are added to the knowledge graph.

Git Repositories URI: https://git-dbs.ifi.uni-heidelberg.de/practicals/2020-ise.git Delete entry

Default Branch: master

Authentication Method: GitLab Personal Acce

User Name: akleebaum

Password or Token: ••••••••••

+ Add New Repository Save Addresses

URI: Uniform resource identifier of the git repository that will be cloned (Example: https://github.com/cures-hub/cures-condec-jira.git).

Default Branch: Name of default branch for the Repository (Example: develop).

Authentication Method: Method of Authentication for accessing non-public repositories.

User Name: The git service user name for authentication.

Password or Token: The git service password or token for authentication.

The git repository/repositories are cloned to JiraHome/data/condec-plugin/git/*project-key*.

Abbildung 4.4: Eingabemaske für Git-Repositorien mit Authentifizierungsdaten.

4.3 SF3: List all code classes for a project

Durch die Implementierung des Wissenstyps `CODE` und des Integrierens von Codedateien in den Wissensgraphen in Abschnitt 4.4 ist diese Systemfunktion bereits fertig implementiert. Im Arbeitsbereich WS1.3.2: Decision Knowledge Overview kann durch Auswahl des Wissenstyps „Code“ eine Liste der Codedateien im Projekt sowie die auf diese zentrierten Subgraphen des Wissensgraphen angezeigt werden. Ein Beispiel dessen zeigt Abbildung 4.6.

4.4 SF4: Automatically add code files from git repository into the knowledge graph

Für die Umsetzung dieser Systemfunktion muss zunächst festgelegt werden, welche Dateien eines Repositoriums als Codedateien interpretiert und als solche in den Wissensgraphen integriert werden sollen. Hier gab es die Möglichkeiten, entweder bestimmte Dateiendungen direkt im Code aufzuzählen, oder diese durch Nutzer:innen festlegen zu lassen. Im ersten Fall haben Nutzer:innen weder die Möglichkeit, weitere Code-Dateiendungen zu definieren, noch vordefinierte Code-Dateiendungen zu ignorieren. Damit Nutzer:innen die Definitionen von Codedateien an ihre Bedürfnisse anpassen können, wurde die

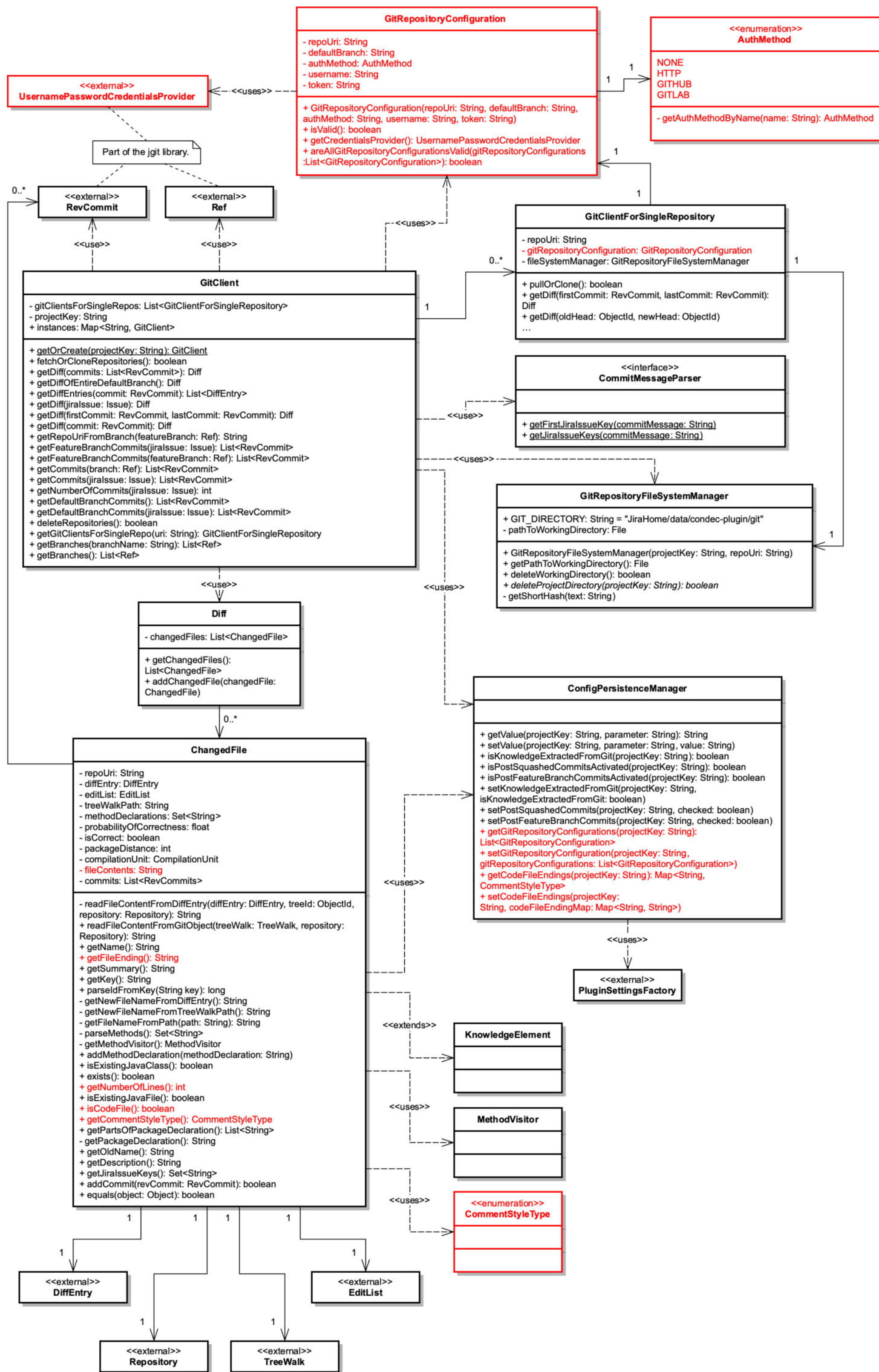

Abbildung 4.5: Klassendiagramm der Git-Anbindung von ConDec Jira.

Rationale Backlog | Overview | Decision Graph | Decisi

Contains text | Code | Summary of nev

Linked Elements Min 0 Max 100 Doc

- app-routing.module.ts
- app.component.spec.ts
- app.component.ts
- app.e2e-spec.ts
- app.module.ts
- app.po.ts
- dbHandler.ts
- environment.dev.ts
- environment.prod.ts
- environment.ts
- fileType.ts
- formatters.test.ts
- formatters.ts
- hei-search.component.spec.ts
- hei-search.component.ts
- insertionHandler.test.ts
- insertionHandler.ts
- language.ts
- main.ts
- mockSolrSearchResponse.ts
- mockSolrSpellcheckForSuggestionResponse.ts
- mockSolrSpellcheckResponse.ts
- mockSolrSuggestResponse.ts
- pageParams.ts
- polyfills.ts

Abbildung 4.6: Eine Liste der Codedateien im Projekt.

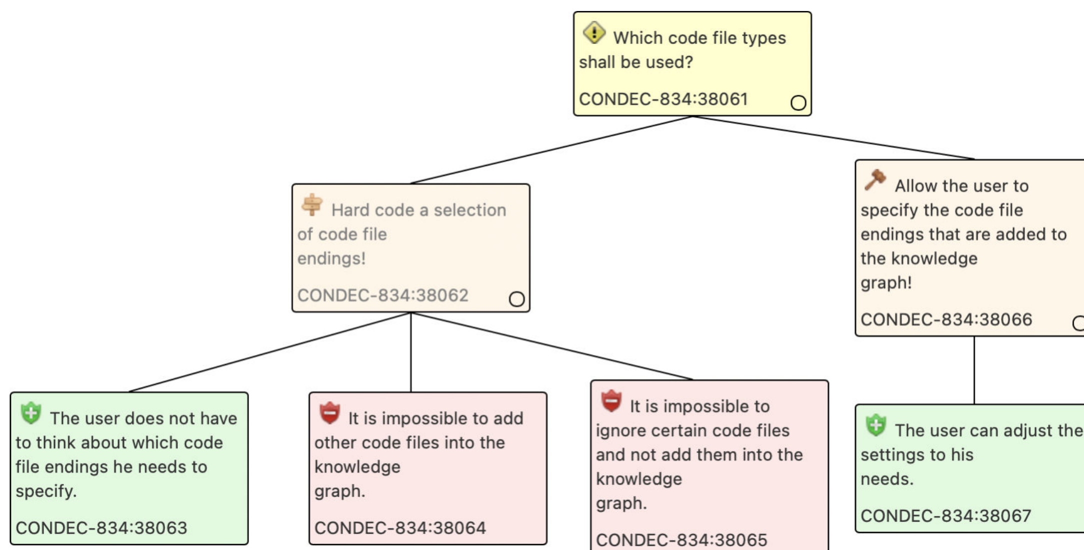

Abbildung 4.7: Entscheidungsbaum für Entscheidungsproblem zur Festlegung der Dateiendungen, die als Codedateien zu interpretieren sind.

letztgenannte Alternative als Entscheidung getroffen. Das Entscheidungsproblem ist in Abbildung 4.7 als Entscheidungsbaum dargestellt.

Die Dateiendungen von Codedateien werden für jedes Projekt im `ConfigPersistenceManager` verwaltet. Hier gibt die neu implementierte Funktion `getCodeFileEndings(...)` eine Zuordnungstabelle von Dateiendungs-Strings auf Instanzen des in Abschnitt 4.5 näher beschriebenen Aufzählungstyps `CommentStyleType` zurück. An dieser Stelle wird zwischen verschiedenen Arten von Kommentarstilen unterschieden, damit in Abschnitt 4.5 Entscheidungswissen aus Code-Kommentaren korrekt ausgelesen werden kann. Innerhalb der bestehenden Klasse `ChangedFile` wurde nun die Funktion `getFileEnding()`, die die Dateiendung der Datei zurückgibt, sowie die Funktion `getCommentStyleType()` implementiert, die prüft, ob die Dateiendung in der Zuordnungstabelle aus dem `ConfigPersistenceManager` auftaucht. Ist das der Fall, wird der jeweilige `CommentStyleType` zurückgegeben, andernfalls der `CommentStyleType NONE`. Die weiterhin implementierte Funktion `isCodeFile()` gibt nun als Wahrheitswert zurück, ob der `CommentStyleType` ein anderer als `NONE` ist.

Bei jedem Abrufen des Git-Repositoriums wird nun innerhalb der Funktion `extractAllChangedFiles(...)` der Klasse `CodeFileExtractorAndMaintainer` über sämtliche Dateien hierin iteriert und mithilfe der Funktion `isCodeFile()` geprüft,

Code File Endings
Java/C code comment style: Example: java, c, h, cpp, hpp
Python code comment style: Example: py, sh
HTML code comment style: Example: html, htm, xml, css, php, vn
TeX code comment style: Example: tex
Save File Endings
Java/C code comment style: Single-line comments with //, multi-line comments with /* ... */.
Python code comment style: Single-line comments with #.
HTML code comment style: Multi-line comments with <!-- ... -->.
TeX code comment style: Single-line comments with %.

Abbildung 4.8: Eingabemaske für Dateiendungen.

ob es sich um eine Codedatei handelt. Ist dies der Fall, wird sie mithilfe der entsprechenden Funktion des `codeFilePersistenceManager`s in den Wissensgraphen eingefügt. Dabei sorgen die Konstruktoren der Klasse `ChangedFile` dafür, dass der Typ des Wissenselements stets `CODE` ist; hierfür wurde der Aufzählungstyp `KnowledgeType` um ebendiesen Typ erweitert.

Um mit der Jira-Benutzeroberfläche kommunizieren zu können, wurde die `ConfigRest`-Schnittstelle um den Endpunkt `/setCodeFileEndings` erweitert, die empfangene Dateiendungen pro Kommentarstil mithilfe des `ConfigPersistenceManager`s speichert.

Die Jira-Benutzeroberfläche zur Verwaltung der Git-Repositorien wurde wie in Abbildung 4.8 gezeigt um Eingabemöglichkeiten von Dateiendungen erweitert. Hierfür wird ein Textfeld pro Kommentarstil angeboten, darunter werden alle Kommentartypen beschrieben. Ein leeres Eingabefeld zeigt ein Beispiel für mögliche Dateiendungen, um Nutzer:innen die zulässige Syntax zu demonstrieren.

Der Entwurf der beschriebenen Implementierung ist in den Klassendiagrammen in Abbildung 4.5, Abbildung 4.9 sowie Abbildung 4.10 gezeigt.

Abbildung 4.9: Klassendiagramm des Wissensmodells von ConDec Jira.

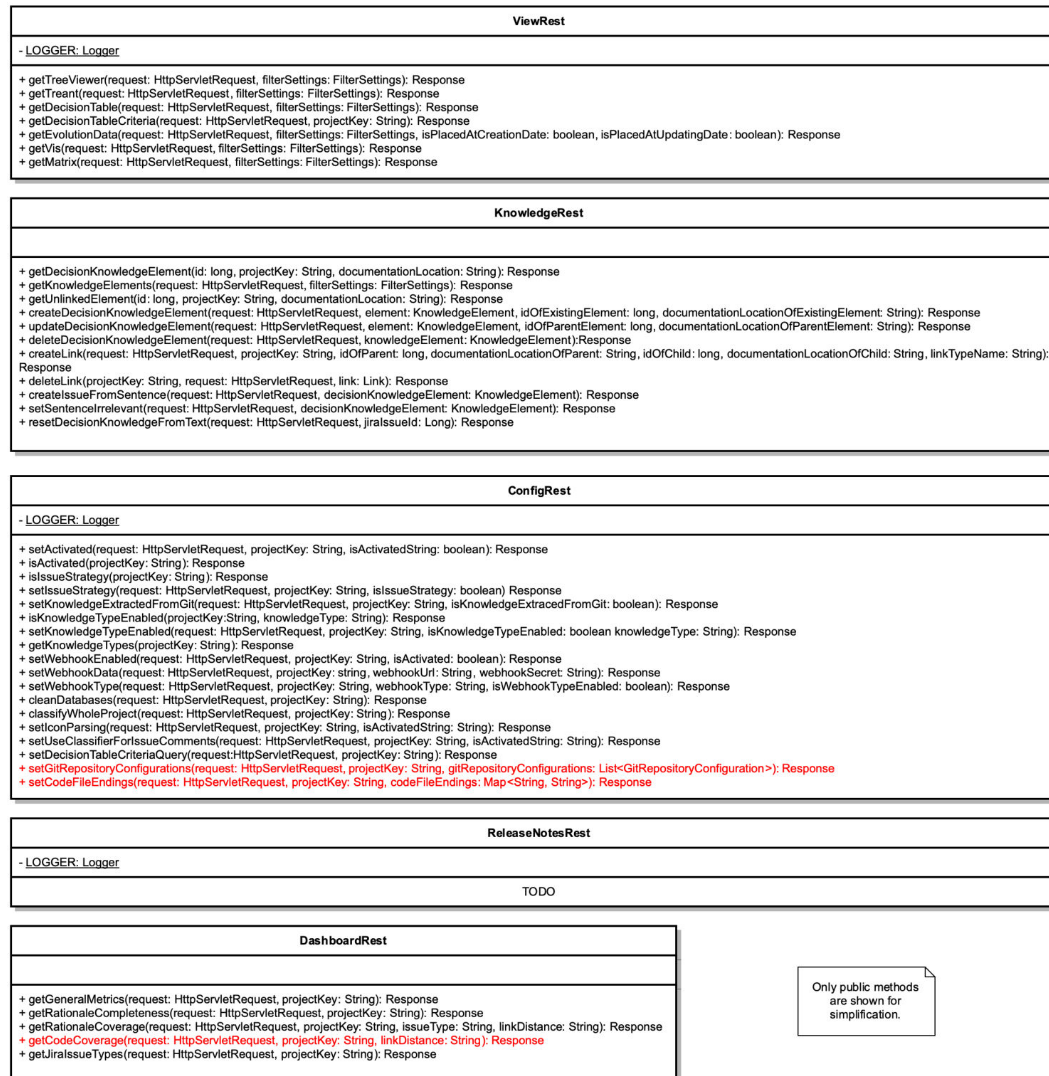

Abbildung 4.10: Klassendiagramm der REST-Schnittstellen von ConDec Jira.

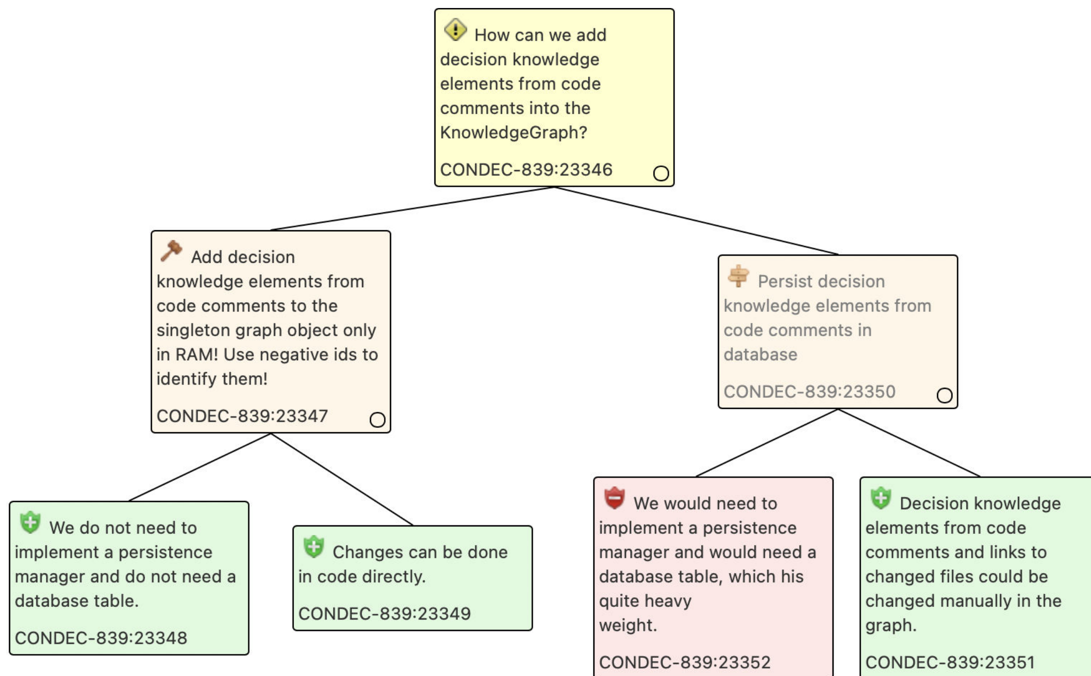

Abbildung 4.11: Entscheidungsbaum für Entscheidungsproblem zur Festlegung der Persistenzstrategie für Entscheidungswissenselemente aus Code-Kommentaren.

4.5 SF5: Automatically add and link decision knowledge from comments in code files into the knowledge graph

Für die Umsetzung dieser Systemfunktion muss zunächst eine Persistenzstrategie für Entscheidungswissenselemente aus Code-Kommentaren festgelegt werden. Hier bestehen die Möglichkeiten, einen weiteren Persistenzmanager zu implementieren, der mithilfe einer weiteren Datenbanktabelle die Entscheidungswissenselemente speichert, oder die Entscheidungswissenselemente nicht persistent zu speichern und stets zur Laufzeit des Plug-ins neu auszulesen und in den Wissensgraphen zu integrieren. Da das Implementieren eines weiteren Persistenzmanagers sehr aufwändig, jedoch für unsere Zwecke nicht erforderlich ist, wurde die Entscheidung getroffen, die Entscheidungswissenselemente nicht persistent zu speichern. So können Änderungen direkt im Code durchgeführt werden. Das Entscheidungsproblem ist in Abbildung 4.11 als Entscheidungsbaum dargestellt.

Tabelle 4.1: Kommentartypen und zugehörige Kommentarsyntax.

Bezeichnung	**Beginn eines einzeiligen Kommentars**	**Beginn eines mehrzeiligen Kommentars**	**Ende eines mehrzeiligen Kommentars**
`JAVA_C`	`//`	`/*`	`*/`
`PYTHON`	`#`	`null`	`null`
`HTML`	`null`	`<!--`	`-->`
`TEX`	`%`	`null`	`null`

Zum Auslesen von Wissenselementen aus Codekommentaren wurde die Klasse `CodeCommentParser` angelegt, die aus einem `ChangedFile` zunächst die Zeichen ausliest, die einen Codekommentar begrenzen. Dies geschieht mithilfe des Aufzählungstyps `CommentStyleType` und der darin enthaltenen Funktionen `getSingleLineCommentChar()`, `getMultiLineCommentCharStart()` and `getMultiLineCommentCharEnd()`, die für die vorgegebenen Kommentartypen die entsprechenden Zeichen zurückgibt. Eine Auflistung der Kommentartypen und zugehörigen Zeichen findet sich in Tabelle 4.1.

Anschließend wird im Inhalt der Codedatei abhängig von den in Tabelle 4.1 gelisteten Zeichen mithilfe von regulären Ausdrücken nach Kommentaren gesucht und diese extrahiert. Hierbei werden gleichzeitig bei mehrzeiligen Kommentaren Zeilenumbrüche, Leerzeichen sowie die in Javadoc-Kommentaren üblichen $*$ am Zeilenanfang durch ein einzelnes Leerzeichen ersetzt.

Die bereits existierenden Klassen `GitDecXtract` und `GitDiffedCodeExtractionManager` werden nun verwendet, um aus den Javadoc-Kommentaren Wissenselemente auszulesen. Die ausgelesenen Wissenselemente werden in der Funktion `extractAllChangedFiles(...)` der Klasse `CodeFileExtractorAndMaintainer` miteinander und mit der Codedatei selbst verlinkt. Hierbei wird ein Argument zu einer darüber stehenden Alternative oder Entscheidung verlinkt, eine Alternative oder Entscheidung wird zu einem darüber stehenden Entscheidungsproblem verlinkt, und ein Entscheidungsproblem wird mit der Codedatei selbst verlinkt. Weiterhin werden die sonstigen Entscheidungswissenstypen Ziel und Problem direkt mit der Codedatei verlinkt, andere anderen sonstigen

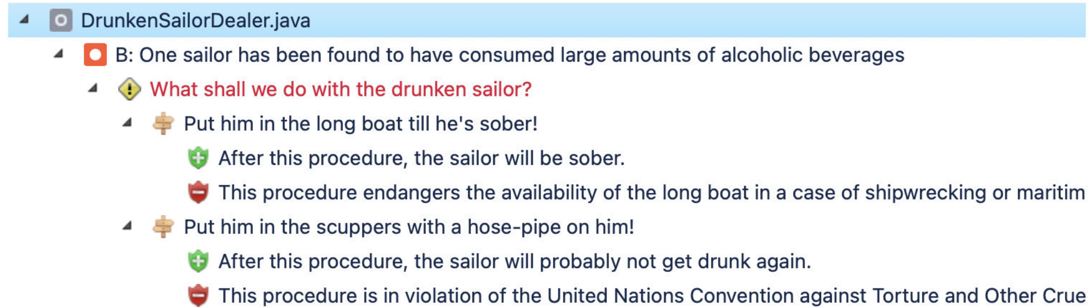

Abbildung 4.12: Subgraph des Wissensgraphen mit Entscheidungswissenselementen aus Code-Kommentaren.

Entscheidungswissenstypen werden mit einem darüber stehenden Entscheidungsproblem verlinkt.

Als Beispiel für als Code-Kommentaren ausgelesene Elemente des Entscheidungswissens ist in Abbildung 4.12 der auf die Codedatei in Listing 2.1 zentrierte Wissensgraph visualisiert.

Der Entwurf der beschriebenen Implementierung ist in den Klassendiagrammen in Abbildung 4.5 und Abbildung 4.9 gezeigt.

4.6 SF6: Show knowledge graph metrics with respect to code files

Zur Berechnung der Rationale Coverage von Codedateien wurde die Klasse `CodeCoverageCalculator` erstellt. Hierin iteriert die Funktion `getCodeFilesWithNeighborsOfOtherType(…)` über alle Codedateien im Wissensgraphen und fügt deren Dateinamen an einen der Zeichenketten `withLink` oder `withoutLink`, abhängig davon, ob die jeweilige Codedatei über eine vorgegebene Linkdistanz mit mindestens einem Wissenselement eines vorgegebenen Typs, etwa Entscheidungsproblem oder Entscheidung, verlinkt ist. Anschließend werden beide Zeichenketten zurückgegeben.

Weiterhin gibt die Funktion `getNumberOfDecisionKnowledgeElementsForCodeFiles(…)` auf analoge Weise eine Zuordnungstabelle zurück, die einer Codedatei die Zahl an Wissenselementen eines vorgegebenen Typs zuordnet, die mit der Codedatei über eine vorgegebene Linkdistanz verlinkt sind.

Da auch hier das Berechnen von Linkdistanzen sehr rechenaufwändig ist, werden in den beschriebenen Funktionen bereits berechnete Linkdistanzen in einer verschachtelten Zuordnungstabelle gespeichert, in der einer Codedatei eine Zuordnungstabelle zugeordnet wird; in diesen Zuordnungstabellen wird einem Elementtyp eine Ganzzahl zugeordnet, die angibt, wie viele Wissenselemente dieses Typs in der vorgegebenen Linkdistanz mit der jeweiligen Codedatei verlinkt sind. Bereits berechnete Informationen werden dann, wenn sie während der Lebenszeit des `CodeCoverageCalculator`s erneut verwendet werden sollen, nicht erneut berechnet, sondern aus den Zuordnungstabellen gelesen. So werden insgesamt Informationen, die in der bestehenden Implementierung vier Mal benötigt werden, nur ein Mal berechnet, was die Performance etwa um den Faktor 4 verbessert.

Um mit der Jira-Benutzeroberfläche kommunizieren zu können, wurde die `DashboardRest`-Schnittstelle um den Endpunkt `/codeCoverage` erweitert, die einen Projektschlüssel sowie eine Linkdistanz empfängt und mithilfe des `CodeCoverageCalculator`s die Ergebnisse der Funktionen `getCodeFilesWithNeighborsOfOtherType(…)` sowie `getNumberOfDecisionKnowledgeElementsForCodeFiles(…)` mit der empfangenen Linkdistanz und jeweils für die Wissenselementtypen Entscheidungsproblem sowie Entscheidung berechnet und sogleich zurückgibt.

Die Jira-Dashboard-Arbeitsbereich wurde um ein in Abbildung 4.13 gezeigtes Dashboard erweitert, das die Auswahl eines Projekts sowie die Eingabe einer Linkdistanz erlaubt. Anschließend werden die Anzahlen der Entscheidungsprobleme sowie Entscheidungen, die über die Linkdistanz mit Codedateien verlinkt sind, in zwei Boxplots visualisiert. Die Anteile an Codedateien, die über die Linkdistanz mit mindestens einem Entscheidungsproblem sowie mindestens einer Entscheidung verlinkt sind oder nicht verlinkt sind, werden in zwei Tortendiagrammen visualisiert. Beim Klick auf die Quantile der Boxplots sowie die Tortenstücke des Tortendiagramms werden die darin enthaltenen Code-

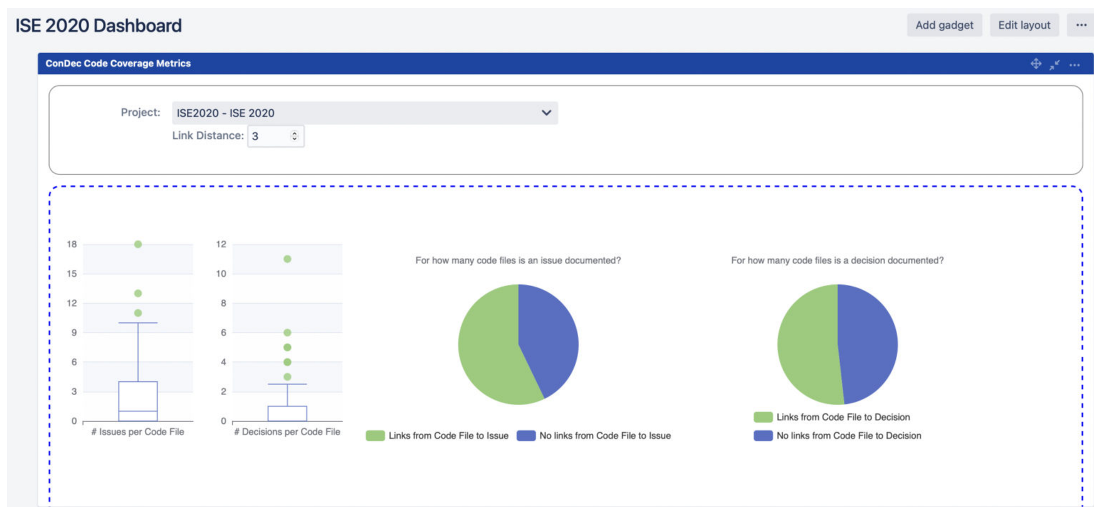

Abbildung 4.13: Darstellung der Rationale Coverage von Codedateien in einem Dashboard.

dateien gelistet, mit einem Klick auf diese wird zu einem auf sie zentrierten Wissensgraphen navigiert.

Der Entwurf der beschriebenen Implementierung ist im Klassendiagramm in Abbildung 4.14 gezeigt.

4.7 SF7: Configure definition of done (DoD) for the decision knowledge documentation

Die Jira-Benutzeroberfläche zur Verwaltung der Definition-of-Done-Einstellungen wurde wie in Abbildung 4.15 gezeigt um die Definition of Done von Codedatein erweitert. Hier werden die folgenden drei Bedingungen aufgelistet, von denen von einer Codedatei mindestens eine erfüllt sein muss, damit diese als gut dokumentiert gilt:

1. Der (case-insensitive) Dateiname der Datei beginnt mit `test`
2. Die Anzahl der Codezeilen in der Datei ist kleiner als ein von Nutzer:innen einstellbarer Schwellwert (Standardwert: 50)

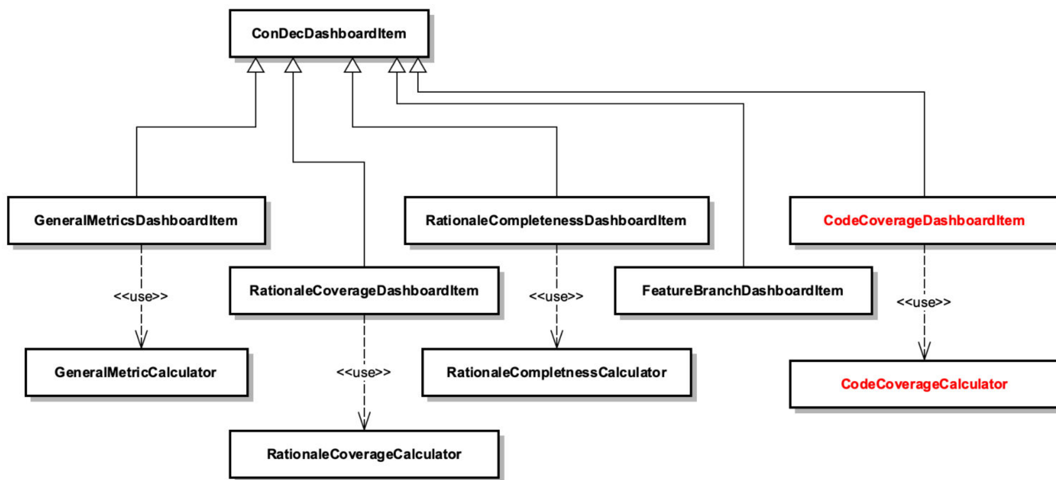

Abbildung 4.14: Überblicksklassendiagramm der Berechnung der in Dashboards visualisierten Daten von ConDec Jira.

Continuous Management of Decision Knowledge (ConDec)

Rationale Backlog Configuration

Settings for the criteria of the definition of done

Set the *definition of done* for each type of decision knowledge.

All elements that do **not** match the definition of done are shown in the Rationale Backlog View. Some of these Criteria are default and can not be changed, others can be configured.

Criteria to match the definition of done:

Issue (Decision Problem):
- is *resolved*
- is linked to a decision
- is linked to an alternative

Decision (Solution):
- is **not** *challenged*
- is linked to an issue (=decision problem)
- is linked to a pro-argument

Alternative:
- is linked to an issue
- is linked to an argument (either pro or con)

Argument (Pro or Con):
- is linked to a solution option (decision or alternative)

Code File:
- is a test file (i.e., its name starts with "test")
- contains less than 50 lines of code
- is linked to a decision with a maximum link distance of 4

Save Definition of Done: Save

Saves all criteria for the definition of done.

Abbildung 4.15: Konfigurationsmöglichkeiten der Definition of Done.

3. Die Codedatei ist im Wissensgraphen zu mindestens einer Entscheidung über eine Linkdistanz von n verlinkt, wobei n ein von Nutzer:innen einstellbarer Schwellwert ist (Standardwert: 4)

Auch können mithilfe zweier Textfelder die Schwellwerte der Bedingungen 2 und 3 geändert werden. Die geänderten Schwellwerte werden mithilfe einer bereits implementierten REST-Schnittstelle dem `ConfigPersistenceManager` übergeben.

4.8 SF8: Check definition of done (DoD) of the decision knowledge documentation related to a code file

Zur Prüfen der Definition of Done für Codedateien wurde die Klasse `CodeCompletenessCheck` als weitere Implementierung der bestehenden Schnittstellenklasse `CompletenessCheck` geschaffen. Die Funktion `execute(...)` gibt dabei den Wahrheitswert `true` zurück, wenn mindestens eine der in Abschnitt 4.7 beschriebenen Bedingungen erfüllt ist. Ist keine der Bedingungen erfüllt, so wird der Wahrheitswert `false` zurückgegeben. Die ersten beiden Bedingungen sorgen dafür, dass kleine Codedateien oder Testklassen die für die Rationale Coverage relevante Bedingung 3 nicht erfüllen müssen. Wird nun in Jira das Rationale Backlog aufgerufen, so wird für jede Codedatei die Funktion `execute(...)` aufgerufen. Ist der Rückgabewert `false`, so erfüllt die Codedatei die Definition of Done nicht und wird im Rationale Backlog angezeigt.

In der Klasse `DefinitionOfDone`, die die von Nutzer:innen einstellbaren Parameter der Definitions of Done verschiedener Wissenselementtypen verwaltet, wurden die entsprechende Zuordnungstabelle um die Parameter `linkDistanceFromCodeFileToDecision` und `lineNumbersInCodeFile` ergänzt sowie triviale Getter- und Setter-Funktionen hinzugefügt. Die bestehende Klasse `ConfigPersistenceManager` speichert zu jedem von ConDec verwalteten Projekt die zugehörige `DefinitionOfDone` und sorgt damit für die Persistenz der eingegebenen Schwellwerte.

Der Entwurf der beschriebenen Implementierung ist im Klassendiagramm in Abbildung 4.16 gezeigt.

4.9 SF9: Navigate to subgraph centered on a code file in Jira

Um die Navigation von IDEs zu einem bestimmten Subgraphen des Wissensgraphen in ConDec Jira zu ermöglichen, muss zunächst eine Möglichkeit geschaffen werden, bei Aufrufen einer URL diesen Subgraphen anzuzeigen. Hierfür eignen sich URL-Parameter. In den Velocity-Templates und den zugehörigen JavaScript-Dateien wird daher der URL-Parameter codeFileName ausgelesen. Ist dieser beim Aufruf der Seite https://<Jira-Server-URL>/projects/-<Projektschlüssel>?selectedItem=decision-knowledge-page leer, so wird wie bisher der Arbeitsbereich WS1.3.1: Rationale Backlog geöffnet. Enthält jedoch der URL-Parameter einen Dateinamen (etwa https://<Jira-Server-URL>/projects/<Projektschlüssel>?selectedItem=decision-knowledge-page&codeFileName=KnowledgeGraph.java), so wird direkt zum Arbeitsbereich WS1.3.2: Decision Knowledge Overview gewechselt, der einen auf die Codedatei KnowledgeGraph.java zentrierten Subgraphen des Wissensgraphen anzeigt.

In ConDec Eclipse wurde anschließend die Klasse OpenWebbrowser um die Funktion openWebpage(…) erweitert, die einen Dateipfad entgegennimmt und dessen letzten Teil, den Dateinamen, ermittelt, aus dem ConfigPersistenceManager die Jira-Server-URL sowie den Projektschlüssel ausliest und daraus eine wie oben dargestellte URL erstellt und sogleich im Webbrowser öffnet. Weiterhin wurde die Klasse ShowGraphInJira als Erweiterung der abstrakten Eclipse-eigenen Klasse AbstractHandler erstellt, deren Funktion execute(…) die eben beschriebene Funktion openWebpage(…) aufruft, wenn im in Abbildung 4.17 gezeigten ConDec-Kontextmenü zu einer Codedatei im Projektexplorer der neue Menüpunkt "Show Graph in Jira" ausgewählt wird. Der Pfad zu dieser Codedatei wird dabei an die Funktionen execute(…) und openWebpage(…) weitergereicht.

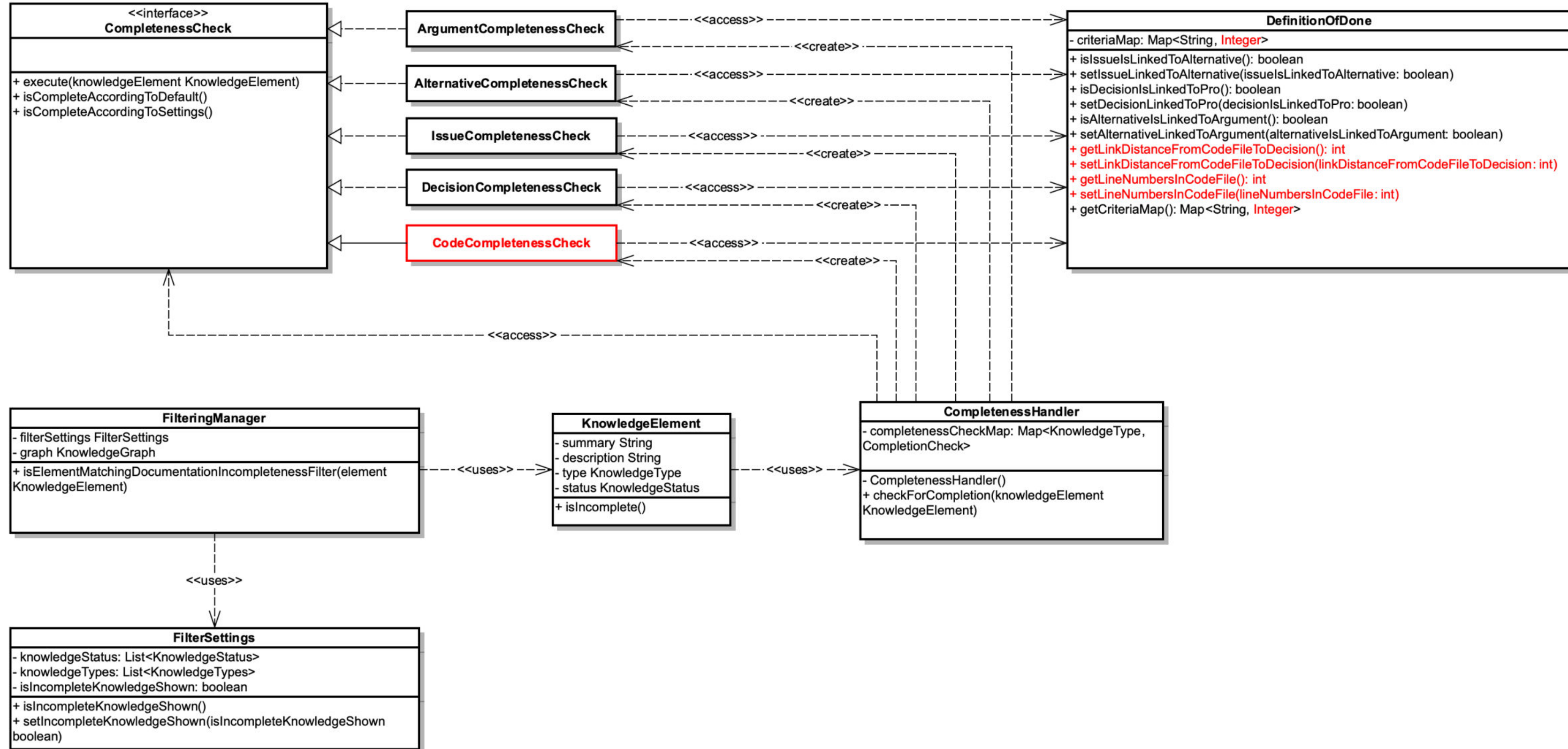

Abbildung 4.16: Klassendiagramm der Prüfungen auf Vollständigkeit von ConDec Jira.

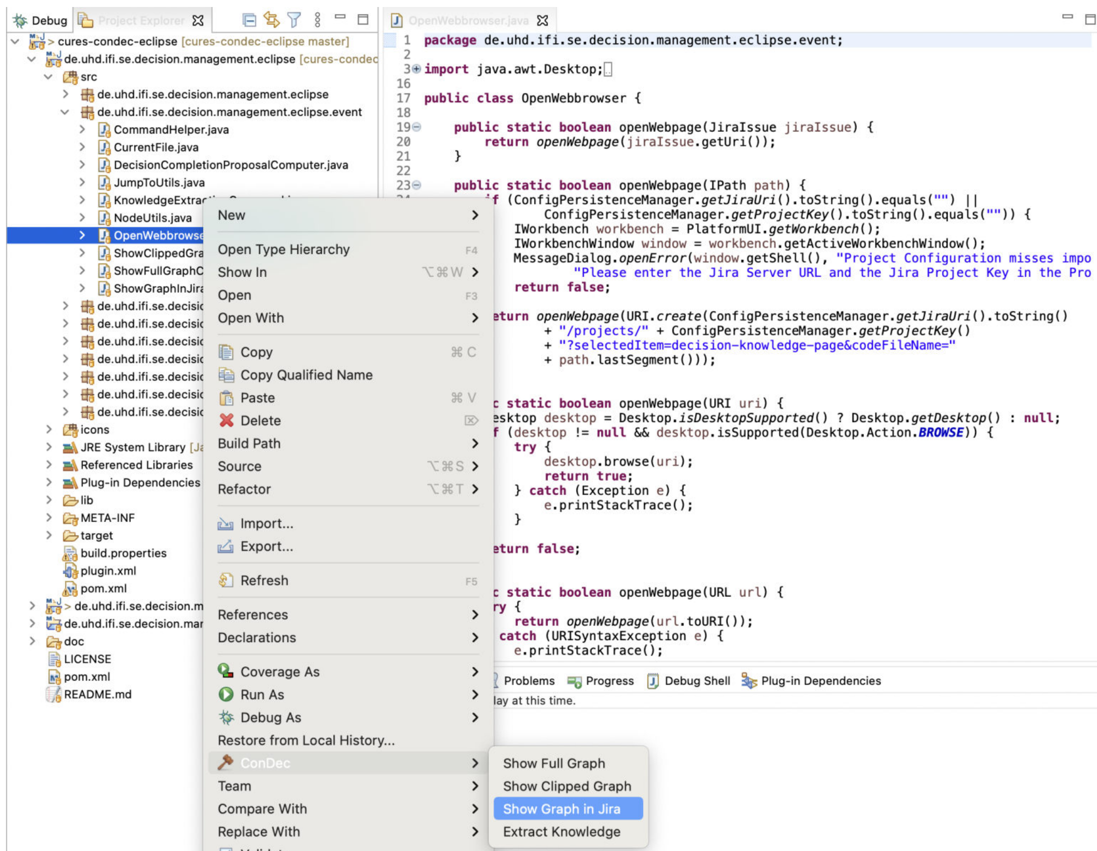

Abbildung 4.17: Kontextmenüpunkt in Eclipse zur Navigation zum ConDec-Jira-Plug-in.

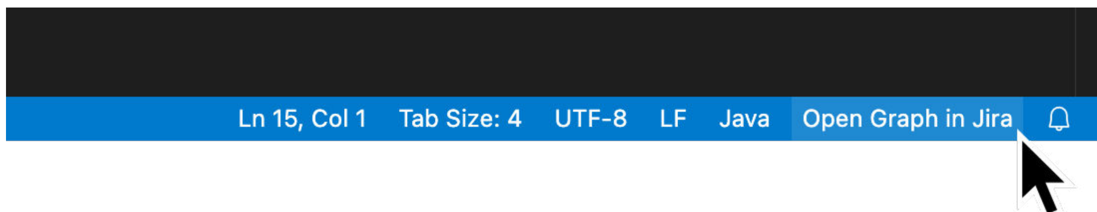

Abbildung 4.18: Statusleistenelement in Visual Studio Code zur Navigation zum ConDec-Jira-Plug-in.

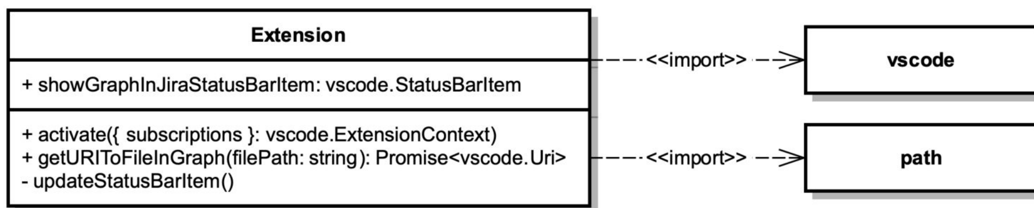

Abbildung 4.19: Klassendiagramm von ConDec Visual Studio Code.

Für Visual Studio Code wurde das neue Plug-in ConDec Visual Studio Code erstellt. Es besteht im Wesentlichen aus der Datei `extension.ts`, die ein Statusleistenelement wie in Abbildung 4.18 definiert. Das Statusleistenelement wird angezeigt, wenn eine Codedatei geöffnet ist. Je nachdem, ob in den Arbeitsbereicheinstellungen Jira-Server-URL sowie Projektschlüssel bereits gespeichert sind oder nicht, lautet der Beschreibungstext des Statusleistenelements "Enter Jira project information…" oder "Open Graph in Jira". Wird auf das Element geklickt, während in den Arbeitsbereicheinstellungen die genannten Informationen nicht gespeichert sind, so öffnen sich Eingabefelder, die zur Eingabe dieser Informationen auffordern. Eingegebene Informationen werden in den Arbeitsbereicheinstellungen gespeichert. Wurden die Informationen eingegeben, oder waren beim Klick auf das Element die Informationen bereits in den Arbeitsbereicheinstellungen gespeichert, so wird aus den Informationen sowie des Dateinamens der geöffneten Codedatei eine wie oben dargestellte URL erstellt und sogleich im Webbrowser geöffnet.

Der Entwurf der beschriebenen Implementierung ist im Klassendiagramm in Abbildung 4.19 gezeigt.

4.10 SF10: Configure Jira Settings [in VS Code]

Diese Systemfunktion ist mit der Implementierung in Abschnitt 4.9 bereits weitgehend implementiert. Die dort beschriebene Speicherung von eingegebenen Jira-Projektinformationen ist eine Möglichkeit, die Arbeitsbereicheinstellungen zu ändern. Eine weitere Möglichkeit besteht darin, die Datei der Arbeitsbereich-

einstellungen unter `.vscode/settings.json` im Editor von Visual Studio Code direkt zu verändern und diese zu speichern.

Kapitel 5

Qualitätssicherung

Nach der in Kapitel 4 beschriebenen Implementierung wurden die Erweiterungen mithilfe von Komponenten- (Abschnitt 5.2) und Systemtests (Abschnitt 5.3) auf Korrektheit getestet. Auch wurde die Erfüllung der nichtfunktionalen Anforderungen (Abschnitt 5.4) überprüft. Zunächst werden in Abschnitt 5.1 die bei der Entwicklung der ConDec-Plug-ins verwendeten Techniken der kontiniuerlichen Integration (CI) beschrieben.

5.1 Kontiniuerliche Integration

Der Code der ConDec-Plug-ins wird in GitHub-Repositorien verwaltet. Bevor neuer Code, der auf Featurezweigen des Repositoriums verwaltet wird, mit dem Code auf den Hauptzweig zusammengeführt werden kann, wird der neue Code von verschiedenen Diensten analysiert und dessen Qualität beurteilt. Nur wenn die Analyse eine hohe Qualität des Codes festgestellt hat, kann eine Zusammenführung erfolgen.

5.1.1 Statische Codeanalyse

Mithilfe des Dienstes *Codacy* wird der Code des Repositoriums einer statischen Codeanalyse unterzogen. Hierbei kommen verschiedene Analysetools zur Anwendung, für Java-Code etwa *PMD*. Dabei wird geprüft, ob Stilkonventionen und Standards eingehalten werden, und das Zusammenführen mit dem Hauptbranch solange verhindert, bis alle Fehler behoben sind. Beispiele für nicht eingehaltene Standards und Konventionen sind etwa nicht verwendete `import`-Befehle, mehrfach implementierter identischer Code oder nicht verwendete lokale Variablen.

5.1.2 Testausführung

Nach jedem Commit werden die Komponententests von dem CI-Dienst *Travis CI* ausgeführt. Dabei wird geschaut, ob und welche Tests fehlschlagen. Ist dies der Fall, so wird das Zusammenführen mit dem Hauptbranch solange verhindert, bis alle Tests bestehen. Während der Testausführung wird die Testabdeckung ermittelt und an den Dienst *Codecov* übermittelt.

5.1.3 Testabdeckung

Nach jeder erfolgreichen Testausführung des CI-Dienstes *Travis CI* wird die dabei festgestellte Testabdeckung von dem Dienst *Codecov* analysiert. Hierbei wird erzwungen, dass die Testabdeckung der in Featurezweigen durchgeführten Änderungen mindestens 65 Prozent betragen muss, insgesamt darf jedoch die Testabdeckung des gesamten Projekts nicht abnehmen. Aktuell beträgt die Testabdeckung im ConDec-Jira-Projekt 87 Prozent, sodass dort insgesamt die geforderte Testabdeckung von 85 Prozent umgesetzt ist.

5.2 Komponententests

Für während der Erweiterung der ConDec-Plug-ins neu hinzugekommenen oder veränderten Code wurden Komponententests geschrieben, die die verschiedenen Funktionalitäten und Verhaltensweisen der getesteten Klassen und Funktionen innerhalb einer Testumgebung auf Korrektheit prüfen. Testdaten wie Commits eines Git-Repositoriums oder Jira-Issues sowie Elemente der Testumgebung wie Eingabefelder von Visual Studio Code wurden in dieser Testumgebung mithilfe von beispielhaften Daten nachgebildet, wobei geeignete Klassen von *JGit*, des Pakets `com.atlassian.mock` sowie die Test-Frameworks *JUnit*, *Mocha* und *Sinon.JS* zur Anwendung kamen.

Wie in Unterabschnitt 5.1.3 und Unterabschnitt 5.1.2 beschrieben wurden die Komponententests ohne Fehler ausgeführt und weisen in den meisten Fällen die geforderte Testabdeckung von 85 Prozent auf.

Im ConDec-Visual-Studio-Code-Projekt wurde hingegen eine Testabdeckung von lediglich 60 Prozent erreicht. Dies ist jedoch auf den verhältnismäßig großen Anteil von UI-Code zurückzuführen, der durch Komponententests nicht ohne eine komplizierte Nachbildung von weiteren UI-Elementen testbar ist. Die Codezeilen, die die wesentliche Funktionalität des Plug-ins (Erfragen und Speichern von fehlenden Einstellungen, Generieren und Aufrufen einer URL) implementieren, weisen eine deutlich hörere Testabdeckung auf.

5.3 Systemtests

Für die in Unterabschnitt 3.2.3 abgeleiteten Systemfunktionen wurden Systemtestfälle erstellt, die die Korrektheit der Implementierung und insbesondere deren korrekten Aufruf über die verschiedenen Sichten prüfen. Die insgesamt 12 Systemtests sind derart gewählt, dass für jede Systemfunktion das erwartete Verhalten sowie die Behandlung von Ausnahmen und besonderen Regeln mindestens ein Mal getestet werden.

Die Tests sind in Jira-Issues des Typs „Test“ beschrieben und in einem Jira-Issue des Typs „Test Concept“[1] gebündelt. Sie sind als logische Testfälle formuliert, die durch die Angabe von einzugebenden Daten konkretisiert sind. Ihre Ausführung ist in einem Jira-Issue des Typs „Test Execution“[2] dokumentiert. Zuvor erforderliche Vorbedingungen sind in den „Test“-Issues enthalten oder in separaten Issues des Typs „Pre-Condition“ beschrieben. Die konkreten Testfälle wurden innerhalb des ConDec-Projekts selbst oder innerhalb des ISE-Projekts 2020 ausgeführt.

Weiterhin wurde während der Ausführung der Systemtestfälle die nichtfunktionale Anforderung „Usability“ wie in Abschnitt 5.4 näher beschrieben getestet.

5.4 Prüfung der nichtfunktionale Anforderungen

5.4.1 Portability

Installierbarkeit

Bei der Prüfung der Portabilität der ConDec-Plug-ins wurde zunächst deren Installierbarkeit untersucht.

ConDec Jira Das ConDec-Jira-Plug-in lässt sich installieren, indem eine vorkompilierte `.jar`-Datei aus dem Repositorium heruntergeladen und in dem vorgesehenen Bereich in Jira hochgeladen wird. Alternativ lässt sich das Plug-in mithilfe eines einzigen Kommandozeilenbefehls lokal kompilieren, sofern zuvor die im Repositorium angegebenen vorausgesetzten Entwicklungssoftwares installiert wurden. Beide Methoden der Installation waren gut dokumentiert, wurden im Praktikum gemäß der Dokumentation durchgeführt und verliefen problemlos. Die Installationsmethoden oder -anleitungen wurden daher im Praktikum nicht weiter verändert.

[1] https://jira-se.ifi.uni-heidelberg.de/browse/CONDEC-916
[2] https://jira-se.ifi.uni-heidelberg.de/browse/CONDEC-929

ConDec Eclipse Das ConDec-Eclipse-Plug-in lässt sich installieren, indem eine vorkompilierte `.jar`-Datei aus dem Repositorium heruntergeladen und in einen vorgegebenen Ordner innerhalb der Eclipse-Installation kopiert wird. Alternativ lässt sich auch dieses Plug-in mithilfe eines einzigen Kommandozeilenbefehls lokal kompilieren, sofern zuvor die im Repositorium angegebenen vorausgesetzten Entwicklungssoftwares installiert wurden. Voraussetzung ist hier zudem, dass Eclipse selbst mit einer vorgegebenen Version des Java Development Kits ausgeführt werden muss. Auch hier waren beide Methoden der Installation gut dokumentiert, wurden im Praktikum gemäß der Dokumentation durchgeführt und verliefen problemlos. Die Installationsmethoden oder -anleitungen wurden auch hier im Praktikum nicht weiter verändert.

ConDec Visual Studio Code Das ConDec-Visual-Studio-Code-Plug-in lässt sich installieren, indem eine vorkompilierte `.vsix`-Datei aus dem Repositorium heruntergeladen und mithilfe eines einzigen Kommandozeilenbefehls in Visual Studio Code installiert wird. Alternativ lässt sich das Plug-in mithilfe zweier Kommandozeilenbefehle lokal kompilieren, sofern zuvor die im Repositorium angegebenen vorausgesetzten Entwicklungssoftwares installiert wurden. Beide Methoden der Installation wurden im Praktikum ausführlich Schritt für Schritt dokumentiert und hiernach gemäß dieser Dokumentation ohne Probleme durchgeführt.

Anpassbarkeit

Auch die Anpassbarkeit der Plug-ins war zu untersuchen. Hierbei wurde geprüft, wie einfach sich die Plug-ins in einer integrierten Entwicklungsumgebung öffnen und ausführen ließen.

ConDec Jira Die Entwicklung des ConDec-Jira-Plug-ins wurde in Visual Studio Code durchgeführt. Alle Kommandozeilenbefehle zum Herunterladen von Dependencies, zum Kompilieren des Plug-ins sowie zum Starten einer lokalen Demoversion von Jira, in der das Plug-in sofort installiert ist, waren wohl dokumentiert, sodass sich das Einrichten der IDE sehr einfach gestaltete. Auch das Anbinden eines Debuggers konnte rasch umgesetzt werden, da in dem an-

gezeigten Log der Jira-Demoversion die hierfür erforderlichen Informationen leicht einsehbar preisgegeben wurden.

ConDec Eclipse Die Entwicklung des ConDec-Eclipse-Plug-ins wurde in Eclipse durchgeführt. Bei der Einrichtung der IDE traten zunächst Probleme auf, da viele Dependencies von Eclipse trotz erfolgreicher Installation derer nicht gefunden werden konnten. In enger Zusammenarbeit mit der Betreuerin konnte dieses Problem gelöst und die Beschreibung der Einrichtung von Eclipse im Repositorium ergänzt und detailliert werden, sodass für zukünftige Erweiterungen die einfache Anpassbarkeit sichergestellt ist. Nach Durchführen der so dokumentierten Einrichtungsschritte konnte das Plug-in ohne Probleme sofort aus der IDE heraus kompiliert, in einer neuen Eclipse-Instanz ausgeführt und ein Debugger angebunden werden.

ConDec Visual Studio Code Die Entwicklung des ConDec-Visual-Studio-Code-Plug-ins wurde in Visual Studio Code durchgeführt. Die Befehle zum Kompilieren, Ausführen sowie Testen des Plug-ins sind wie bei Node.js-Projekten üblich in der Datei `package.json` enthalten. Zusätzlich wurden sie als *Launch Configurations* oder *Tasks* innerhalb von Visual Studio Code eingerichtet, indem entsprechende Konfigurationsdateien im Ordner `.vscode` angelegt wurden. Beim Öffnen des Projekts in Visual Studio Code stehen Entwickler:innen diese Konfigurationen sofort zur Verfügung, sodass sie das Plug-in etwa mit einem Klick ausführen können. Das Vorhandensein dieser Konfigurationsdatei ist im Repositorium dokumentiert, sodass die leichte Anpassbarkeit des Plug-ins insgesamt gewährleistet ist.

5.4.2 Maintainability

Die einfache Änderbarkeit einer Software kann mithilfe von statischen Codeanalysen überprüft werden. Für die statischen Codeanalysen wird wie in Unterabschnitt 5.1.1 der Dienst Codacy verwendet. Die Codequalität aller drei Plug-ins wurde dabei mit der Bestnote A bewertet. Sämtliche Verstöße gegen Stilkonventionen oder Standards, die von Codacy festgestellt wurden, wurden vor dem Zusammenführen mit dem Hauptzweig des Repositoriums behoben.

Bei Entwurf und Implementierung wurde stets darauf geachtet, Erweiterungen nach Möglichkeit in die bestehenden Klassen zu integrieren. Mussten Komponenten trotzdem neu erstellt werden, so wurden sie vielerorts, etwa bei `AuthMethod` oder `CommentStyleType`, als Aufzählungstyp implementiert, was es erlaubt, diese später auf einfache Weise zu erweitern.

5.4.3 Usability

Während der Ausführung der in Abschnitt 5.3 beschriebenen Systemtests wurde die Benutzbarkeit der Erweiterungen der ConDec-Plug-ins geprüft. Hierbei wurde geprüft, ob die Erweiterungen leicht verständlich ohne Anleitung oder Erlernung benutzbar sind sowie ob sie ansprechend aussehen.

Es konnte festgestellt werden, dass die Benutzbarkeit der Erweiterungen insgesamt gegeben ist. Beispiele (wie etwa in Abbildung 4.8) und Erläuterungen (wie etwa in Abbildung 4.4 oder die in Tabelle 3.15 beschriebenen Hinweis- und Fehlertexte) unterstützen das bessere Verständnis von Features und Konfigurationsoptionen. Das *Look and Feel* der Benutzeroberflächen wurde stets an die bestehenden Implementierungen, etwa an verwendete Visualisierungs-Frameworks, und an Jira, Eclipse sowie Visual Studio Code angepasst, die hierfür bereitgestellten Interfaces der jeweiligen Softwarehersteller wurden stets verwendet. Beispiele hierfür sind die Verwendung von Jira-eigenen CSS-Klassen wie `aui-button`, sichtbar in Abbildung 4.4, oder das Einbetten von UI-Elementen in die Statusleiste von Visual Studio Code in Abbildung 4.18.

Kapitel 6

Evaluation

In diesem Kapitel werden innerhalb eines zusammenhängenden Nutzungsbeispiels im ConDec-Jira-Projekt die im Praktikum umgesetzten Erweiterungen angewendet.

Anbindung des Git-Repositoriums Um Codedateien aus dem ConDec-Jira-Git-Repositorium auslesen zu können, muss es in den Git-Einstellungen von ConDec hinzugefügt werden. Hierzu wird dessen URL in das vorgegebene Feld eingetragen und der Standardzweig "develop" angegeben, auf dem der Code hauptsächlich verwaltet wird. Eine Authentifizierung ist hier nicht erforderlich, da das Repositorium öffentlich zugänglich ist. Im entsprechenden Feld wird daher "None" ausgewählt.

Der Code im Repositorium besteht im Wesentlichen aus Java-Dateien im Backend und aus JavaScript-Dateien sowie Velocity-Templates im Frontend. In diesem Nutzungsbeispiel sollen nur die Java-Dateien des Backends analysiert werden. Hierzu wird bei der Festlegung der Codedateiendungen nur `java` eingetragen. Die Eintragung erfolgt in das mit "Java/C code comment style" bezeichnete Feld, damit das in Java-Code-Kommentaren dokumentierte Entscheidungswissen ausgelesen wird.

Nach der Speicherung beider Eingaben wie in Abbildung 6.1 wird die Git-Extraktion eingeschaltet. Hiernach wurde auf die Bestätigung des Jira-Servers, dass die Extraktion der Dateien aus Git erfolgreich war, gewartet. Da das Con-

Abbildung 6.1: Konfiguration der Git-Extraktion.

Dec-Jira-Projekt jedoch sehr viele Codedateien enthält und das Auslesen viel Zeit in Anspruch nimmt, wurde eine solche Bestätigung nicht angezeigt. Stattdessen meldete der Server nach exakt 5 Minuten den Fehler “502 Proxy Error”. Hier wird vermutet, dass ein Timeout innerhalb des Backends für diesen Fehler verantwortlich ist.

Anzeige der ausgelesenen Codedateien und des ausgelesenen Entscheidungswissens Trotzdem scheint das Auslesen der Codedateien auch nach dieser Rückmeldung des Servers fortgesetzt zu werden. Nach einigen weiteren Minuten wird in die Überblicksansicht gewechselt und dort eine Liste aller Codedateien im Projekt angezeigt. Die Liste enthält mehrere Hundert Einträge und scheint dabei vollständig zu sein. Um im Code dokumentierte Entscheidungswissenselemente anzuzeigen, wird die Auswahlbox “Decision Knowledge Only” aktiviert und die “Link Distance” bei 3 belassen. Anschließend wird die Codedatei `GitClient.java` ausgewählt. Im anschließend angezeigten Wissensgraphen in Abbildung 6.2 werden die im Code dokumentierten Entscheidungswissenselemente angezeigt.

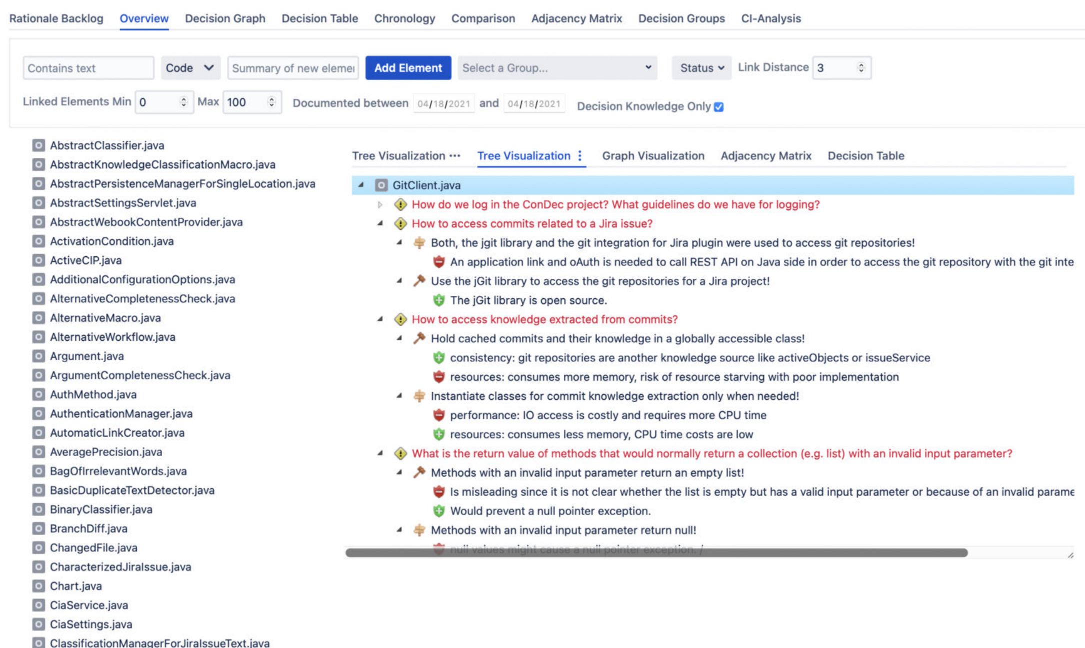

Abbildung 6.2: Wissensgraph mit Entscheidungswissen, zentriert auf die Codedatei `GitClient.java`.

Auffällig waren hier erneut die langen Ladezeiten. Die Liste an Codedateien wurde nach etwa einer Minute angezeigt, der Wissensgraph zur Codedatei nach etwa 30 Sekunden. Zwischenzeitlich nahm die angezeigte Webseite mehr als 2 GB Arbeitsspeicher des Clients in Anspruch. Weiterhin fiel auf, dass nach Auswahl der Codedatei zeitgleich Daten für alle verfügbaren Visualisierungsframeworks abgerufen wurden.

Anzeige von Metriken zur Rationale Coverage Nun soll geprüft werden, welche Codedateien nicht mit Entscheidungswissen verlinkt sind. Hierfür wird das Dashboard “ConDec Code Coverage Metrics” aufgerufen und das Projekt “CONDEC – ConDec Jira” ausgewählt. Die vorausgewählte Linkdistanz von 3 wird beibehalten. Nach etwa 3 Sekunden werden die Metriken in vier Diagrammen wie in Abbildung 6.3 angezeigt. Aus dem linken Tortendiagramm ist abzulesen, dass insgesamt 535 Codedateien ausgelesen werden, was der Anzahl an Java-Dateien im ConDec-Projekt entspricht. Von den 535 Codedateien ist für 526 ein Entscheidungsproblem dokumentiert, für 9 Codedateien ist dies nicht der Fall. Durch Klick auf das entsprechende Element im Tortendiagramm werden die 9 Codedateien ohne dokumentiertes Entscheidungsproblem wie in Abbildung 6.4 angezeigt. Hierunter befindet sich etwa die Da-

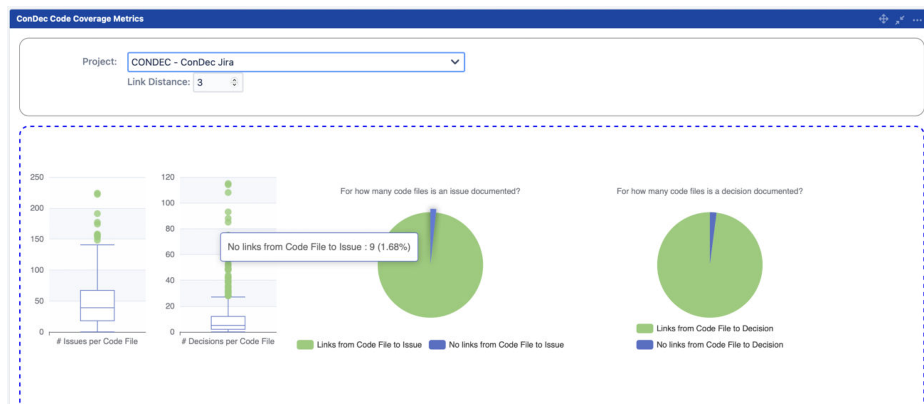

Abbildung 6.3: Dashboard mit Metriken zur Rationale Coverage von Codedateien.

Navigate to Detail View of:

CONDEC-CodeCoverageCalculator.java CONDEC-JiraIssueMetric.java CONDEC-TargetGroup.java CONDEC-CodeCoverageDashboardItem.java CONDEC-TestCodeCoverageCalculator.java CONDEC-TestJiraIssueMetric.java CONDEC-TestTargetGroup.java CONDEC-TestGetCodeCoverage.java CONDEC-TestCodeCoverageDashboardItem.java .

Abbildung 6.4: Liste an Codedateien ohne dokumentiertes Entscheidungsproblem als Detailansicht des Dashboards.

tei `CodeCoverageCalculator.java`. Durch Klick auf diese wird der zugehörige Wissensgraph geöffnet. Der Wissensgraph zeigt, dass die Codedatei in der Arbeitsaufgabe “Add dashboard with metrics related to code files” geändert wurde. Die Arbeitsaufgabe wird aufgerufen und ein geeignetes Entscheidungsproblem in ihrer Beschreibung ergänzt. Anschließend wird das Dashboard erneut aufgerufen. Nun wird angezeigt, dass nur noch vier Codedateien ohne dokumentiertes Entscheidungsproblem bestehen.

Festlegen und Prüfen der Definition of Done von Codedateien Es soll festgestellt werden, welche Codedateien die Definition of Done nicht erfüllen. Hierfür wird zunächst an den Einstellungen für das Rationale Backlog nichts geändert, die Standardeinstellungen (weniger als 50 Codezeilen, Entscheidung in Distanz von maximal 4) sind daher für das Projekt konfiguriert. Das Rationale Backlog wird aufgerufen, das Startdatum wird, um alle Wissenselemente einzuschließen, auf den 1. Januar 1980 gesetzt, und in der Auswahlliste für den Wissenstyp wird ausschließlich “Code” ausgewählt. Das angezeigte Rationale Backlog ist wie in Abbildung 6.5 oben leer, was bei der Betrachtung der vier Codedateien ohne dokumentiertes Entscheidungsproblem bei der Anzeige der Metriken realistisch und konsistent erscheint.

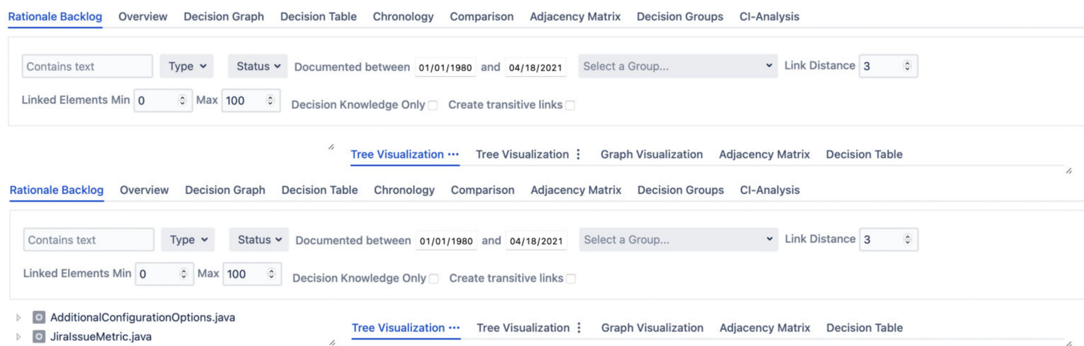

Abbildung 6.5: Rationale Backlog vor (oben) und nach (unten) Anpassung der Definition of Done.

Um trotzdem einige Codedateien im Rationale Backlog anzuzeigen, wurde die Definition of Done für Codedateien wie in Abbildung 6.6 verändert, sodass Entscheidungen in einer Distanz zur Codedatei von maximal 3 verlinkt sein müssen. Hiernach wurden im wie zuvor gefilterten Rationale Backlog zwei Codedateien angezeigt, dargestellt in Abbildung 6.5 unten.

Anzumerken ist auch hier, dass die Anzeige des Rationale Backlogs nach Änderung der Filterkriterien etwa eine Minute benötigte.

Generieren transitiver Links Im nächsten Schritt soll festgestellt werden, in welchen Codedateien eine bestimmte Funktionalität implementiert ist. Gegenstand der Untersuchung ist die Systemfunktion "Automatically add and link decision knowledge from comments in code files into the knowledge graph". Beim Aufruf dieser ist zunächst erkennbar, dass sie nicht direkt mit Codedateien verlinkt ist. Stattdessen ist sie mit mehreren Arbeitsaufgaben verlinkt, die wiederum mit Codedateien verlinkt sind. Um nun eine Liste aller Codedateien, die die Systemfunktion implementieren, anzuzeigen, werden im Abschnitt "Decision Knowledge" des Jira-Issues der Systemfunktion Filter verwendet: Als Wissenstyp wird lediglich "Code" ausgewählt, die Linkdistanz wird auf 2 bestimmt, schließlich wird die Auswahlbox "Create transitive links" aktiviert. Im so gefilterten Wissensgraphen wird nach etwa 800 ms eine Liste an 27 Codedateien wie in Abbildung 6.7 angezeigt. Bei genauer Betrachtung dieser Codedateien lässt sich feststellen, dass für genau die Codedateien, die mit mindestens einer der drei Arbeitsaufgaben der Systemfunktion verlinkt sind, transitive Links erstellt wurden.

Continuous Management of Decision Knowledge (ConDec)

Rationale Backlog Configuration

Settings for the criteria of the definition of done

Success
The definition of done is updated.

Set the *definition of done* for each type of decision knowledge.

All elements that do **not** match the definition of done are shown in the Rationale Backlog View. Some of these Criteria are default and can not be changed, others can be configured.

Criteria to match the definition of done:

Issue (Decision Problem):
- is *resolved*
- is linked to a decision
- is linked to an alternative

Decision (Solution):
- is **not** *challenged*
- is linked to an issue (=decision problem)
- is linked to a pro-argument

Alternative:
- is linked to an issue
- is linked to an argument (either pro or con)

Argument (Pro or Con):
- is linked to a solution option (decision or alternative)

Code File:
- is a test file (i.e., its name starts with "test")
- contains less than 50 lines of code
- is linked to a decision with a maximum link distance of 3

Save Definition of Done: Save

Saves all criteria for the definition of done.

Abbildung 6.6: Anpasste Definition of Done für Codedateien.

Tree Visualization | Tree Visualization | Graph Visualization | Decision Table | Feature branch(es) | Adjacency Matrix | Related knowledge | Duplicate knowledge | Chronology | CI-Analyse

Contains text | Type | Status | Select a Group... | Documented between 04/18/2021 and 04/18/2021

Linked Elements Min 0 Max 100 | Link Distance 2 | Decision Knowledge Only

Sentences without Decision Knowledge | Create transitive links

SF: Automatically add and link decision knowledge from comments in code files into the knowledge graph
- ChangedFile.java
- CharacterizedJiraIssue.java
- CiaService.java
- CodeCommentParser.java
- CodeFileExtractorAndMaintainer.java
- CommentMetricCalculator.java
- CommentStyleType.java
- CommitMessageParser.java
- ConfigPersistenceManager.java
- ConfigRest.java
- DecisionGroupManager.java
- DecisionKnowledgeProject.java
- Diff.java
- FilterSettings.java
- FilteringManager.java
- GeneralMetricCalculator.java
- GeneralMetricsDashboardItem.java
- GenericLinkManager.java
- GitDecXtract.java
- GitDiffedCodeExtractionManager.java
- JiraIssueTextPersistenceManager.java
- KnowledgeElement.java
- KnowledgeGraph.java
- KnowledgeRest.java
- Link.java
- MockJiraHomeForTesting.java
- RationaleFromCodeCommentExtractor.java

Abbildung 6.7: Systemfunktion mit transitiven Links zu Codedateien.

Navigation von IDE zu Wissensgraph in Jira Zuletzt soll demonstriert werden, wie aus einer IDE zum Wissensgraphen einer Codedatei in Jira gewechselt werden kann. Hierfür wird die in Abbildung 6.5 angezeigte Codedatei `AdditionalConfigurationOptions.java` betrachtet, sodass analysiert werden kann, warum sie die Definition of Done nicht erfüllt.

Eclipse In Eclipse wird angenommen, dass das ConDec-Plug-in bereits eingerichtet wurde. Es wurde dann die genannte Codedatei geöffnet und sogleich mithilfe eines Rechtsklicks aus dem Dateiexplorer ausgewählt. Im Kontextuntermenü "ConDec" wurde der Menüpunkt "Show Graph in Jira" wie in Abbildung 6.8 ausgewählt.

Visual Studio Code In Visual Studio Code wurde die genannte Codedatei geöffnet, anschließend wurde das Statusleistenelement "Enter Jira project information…" ausgewählt. Da das ConDec-Plug-in für dieses Projekt noch nicht eingerichtet wurde, wird zur Eingabe von Jira-Server-URL und Projektschlüssel aufgefordert. Nach erfolgreicher Eingabe der entsprechenden Werte wies zunächst eine Informationsnachricht auf die Möglichkeit hin, die eingegebenen Informationen jederzeit in den Arbeitsbereicheinstellungen anzupassen. Nach Eingabe aller Informationen wird ein Fenster angezeigt, in dem das Öffnen einer URL zu bestätigen ist. Die verschiedenen Anzeigen von ConDec Visual Studio Code zeigt Abbildung 6.9.

Anzeige in Jira Beide IDE-Eingaben führen nun zum Aufruf derselben URL in Jira. Dabei wird wie in Abbildung 6.10 eine Liste aller Codedateien angezeigt, nebenstehend ist ein auf die genannte Codedatei zentrierter Subgraph des Wissensgraphen gezeigt. Bei Betrachtung des Subgraphen lässt sich schnell feststellen, dass die Codedatei die Definition of Done nicht erfüllt, weil innerhalb der vorgegebenen Linkdistanz von 3 keine Entscheidung zu ihr verlinkt ist.

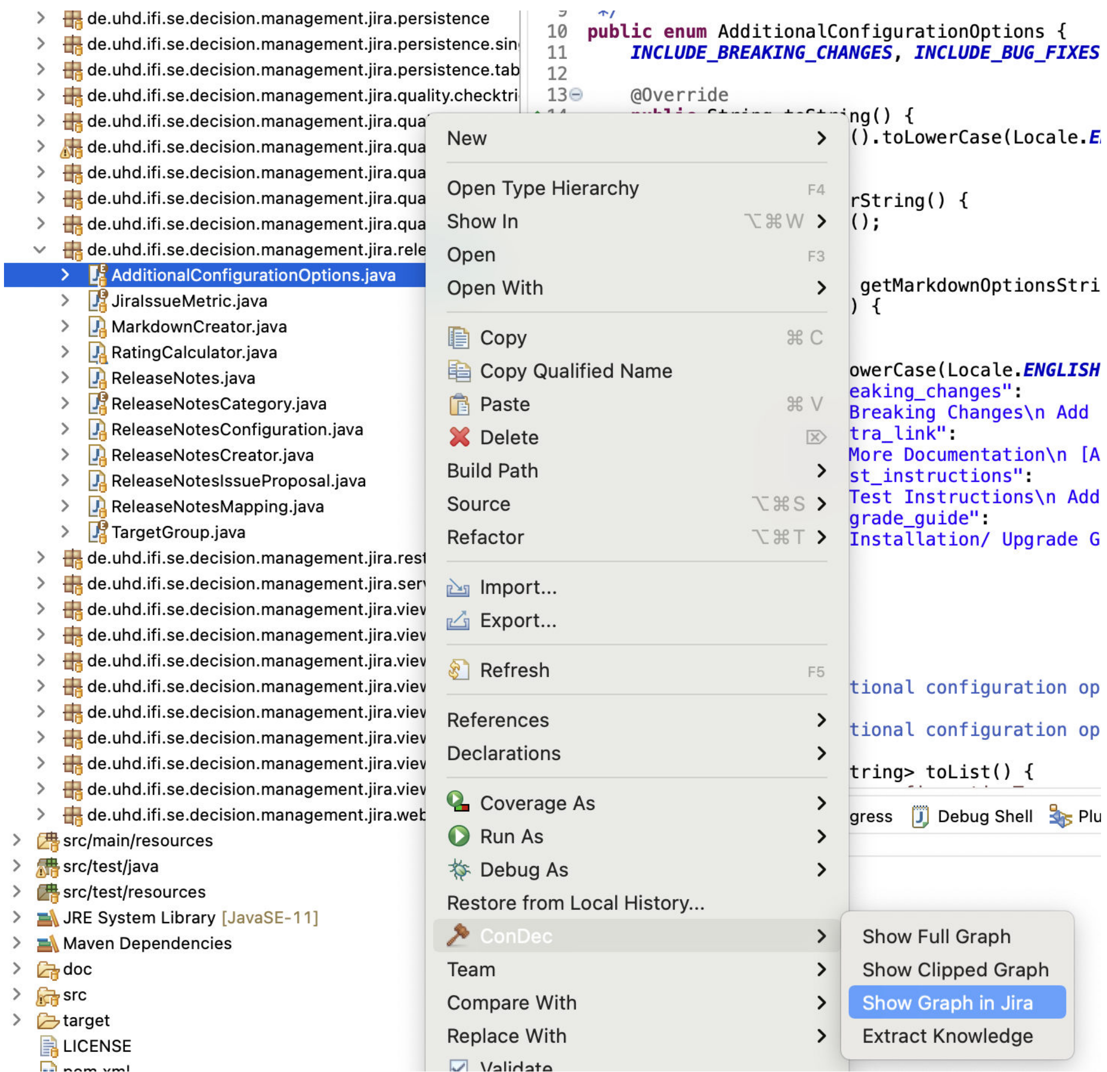

Abbildung 6.8: Schaltfläche zur Navigation von Eclipse zu Jira.

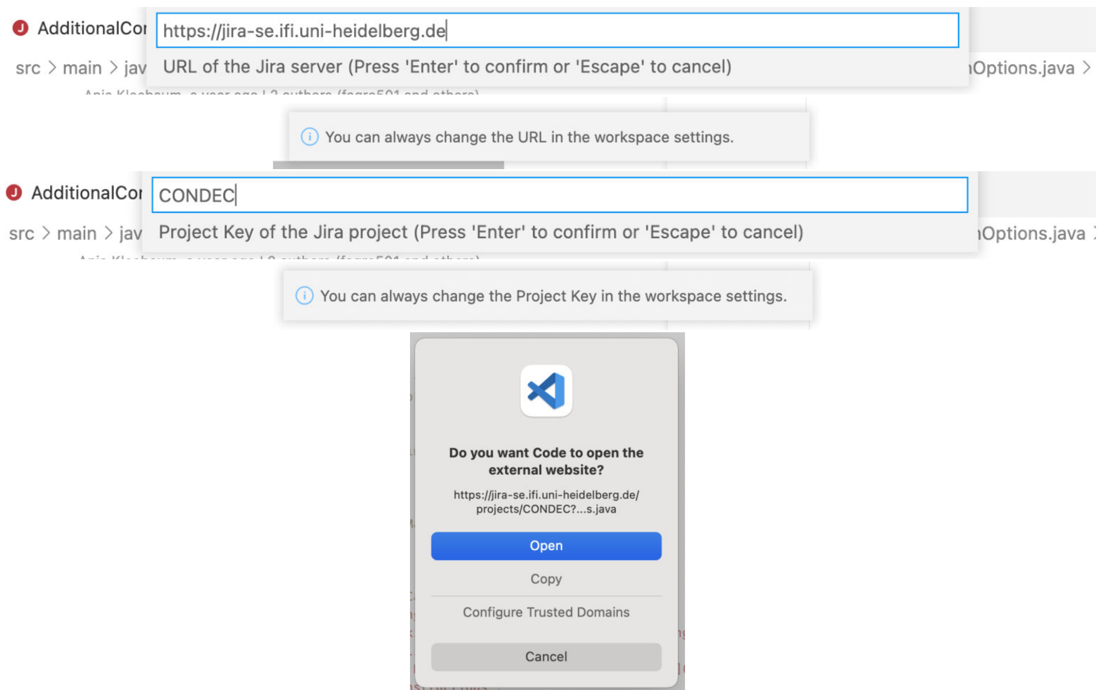

Abbildung 6.9: Eingabe- und Informationsfenster bei der Navigation von Visual Studio Code zu Jira.

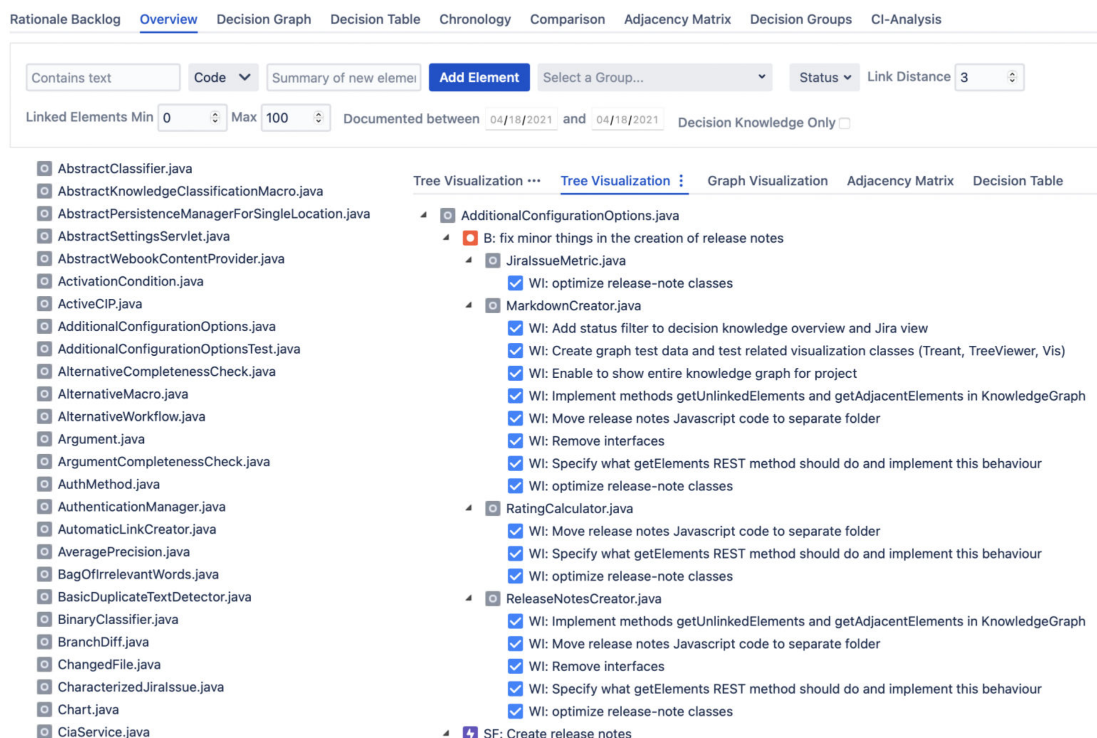

Abbildung 6.10: Subgraph des Wissensgraphen zur Codedatei `AdditionalConfigurationOptions.java`.

Kapitel 7

Fazit

Im letzten Kapitel werden die Ergebnisse des Praktikums zusammengefasst sowie die Erfahrungen bei dessen Bearbeitung dokumentiert. Schließlich werden im Ausblick mögliche zukünftige Erweiterungen und Verbesserungen der ConDec-Plug-ins skizziert.

7.1 Zusammenfassung

Im Praktikum wurden Möglichkeiten umgesetzt, Codedateien und Entscheidungswissen aus Codekommentaren in den Wissensgraphen zu integrieren, zu visualisieren, zu filtern sowie Metriken zu ihnen zu erheben.

Die Anbindung von Code-Repositorien an das ConDec-Jira-Plug-in wurde auch für authentifizierungspflichtige Repositorien ermöglicht. Es wurde ermöglicht, Codedateien sowie in ihren Code-Kommentaren dokumentiertes Entscheidungswissen in den Wissensgraphen einzubinden, geeignet zu verlinken und anzeigen zu lassen. Hierbei können Dateitypen von Codedateien sowie darin verwendete Kommentarsyntaxen von Nutzer:innen gewählt werden. Die Visualisierung von Entscheidungswissen wurde durch die Möglichkeit, transitive Links zu generieren, erweitert. Codedateien können mithilfe von Dashboards sowie des Rationale Backlogs hinsichtlich ihrer Rationale Coverage ausgewertet und gefiltert werden. Eine Navigationsmöglichkeit von einer Codedatei innerhalb

einer IDE zum Wissensgraphen in Jira, der auf diese Codedatei zentriert ist, wurde geschaffen.

Die korrekte Funktionalität der Erweiterungen wurde sowohl in Komponenten- als auch in Systemtests festgestellt. Zugleich wurde gezeigt, dass die nichtfunktionalen Anforderungen erfüllt wurden. In einem zusammenhängenden Nutzungsbeispiel wurden schließlich die im Praktikum umgesetzten Erweiterungen angewendet.

7.2 Ausblick

Die vorliegende Umsetzung bietet eine Grundlage, um die Zusammenhänge zwischen Entscheidungswissen und Code zu erkunden, zu visualisieren und zu analysieren. Gleichwohl bietet sie an verschiedenen Stellen Möglichkeiten zur Verbesserung, die Grundlage zukünftiger Forschungsprojekte sein können.

Ein häufig auftretendes Problem waren lange Berechnungs- und hieraus resultierende Ladezeiten, etwa bei der Berechnung transitiver Links oder der Metriken für die Dashboard-Ansichten. Grund war häufig die sehr rechenintensive und rekursiv implementierte Berechnung von Linkdistanzen und Elementen, die sich innerhalb einer vorgegebenen Linkdistanz zu einem anderen Element befinden. Ebenfalls verantwortlich war das sofortige Berechnen verschiedener Ansichten, die Nutzer:innen nicht sofort, sondern nur nach manueller Auswahl angezeigt werden. Hier kann ein Einsatz effizienterer Algorithmen und gegebenenfalls geeigneterer Graph-Bibliotheken die Wartezeit für Nutzer:innen deutlich reduzieren und so die Nützlichkeit der genannten Features verbessern. Selbiges gilt für die Berechnung von Ansichten erst nach Auswahl durch Nutzer:innen.

Die langen Berechnungs- und Ladezeiten führten überdies häufig zu Timeouts, sodass etwa Bestätigungsmeldungen nicht angezeigt wurden. Es ist daher zu empfehlen, die Schwellwerte für Timeouts anzupassen, wenn und soweit lange Berechnungs- und Ladezeiten nicht verringert werden können.

Ein weiteres Performanceproblem stellte teilweise der Arbeitsspeicherbedarf mancher Sichten dar. Hier könnte überprüft werden, ob alle an den Client übertragenen Daten tatsächlich für die Visualisierung erforderlich sind; gegebenenfalls können Optimierungen vorgenommen werden. Insbesondere ließen sich eventuell Berechnungen von der Client- auf die Serverseite verschieben.

Außerdem wurde festgestellt, dass der Schlagwortfilter des Arbeitsbereichs *WS1.3.2: Decision Knowledge Overview* nicht nur die Wissenselemente in der Liste der Wissenselemente auf der linken Seite filtert, sondern auch den zum dort ausgewählten Wissenselement gehörigen Wissensgraphen auf der rechten Seite. Das führt dazu, dass es nicht möglich ist, den Schlagwortfilter zu benutzen, um ein Wissenselement zusammen mit dem vollständigen zugehörigen Wissensgraphen anzuzeigen. Je nach gewähltem Schlagwort enthält der angezeigte gefilterte Wissensgraph neben dem Wurzelelement nur sehr wenige oder gar keine anderen Wissenselemente. Eine Anpassung der Anwendung der Filter lediglich auf die Liste der Wissenselemente könnte hier die Nützlichkeit dieser Funktionalität steigern. Auch könnte der Schlagwortfilter durch Ergänzungen wie die Anzeige von Filtervorschlägen während des Tippens oder eine Rechtschreibkorrektur auf der Basis der Wissenselemente im Wissensgraphen erweitert werden.

Das im Praktikum erweiterte Rationale Backlog zeigt zurzeit an, welche Wissenselemente die Definition of Done nicht erfüllen, jedoch besteht die Definition of Done für viele Wissenstypen aus mehreren einzelnen Bedingungen. Um die Dokumentation der Wissenselemente so anzupassen, dass sie die Definition of Done erfüllen, müssen Nutzer:innen allerdings wissen, warum genau die Definition of Done nicht erfüllt ist und welche Bedingungen zur Erfüllung fehlen. Eine Anzeige der nicht erfüllten einzelnen Bedingungen zu jedem Wissenselement im Rationale Backlog würde dessen Nützlichkeit steigern.

Im Praktikum wurden Codedateien weitgehend als atomare Objekte behandelt und deren Inhalt bis auf die Extraktion von Entscheidungswissen aus Codekommentaren nicht weiter betrachtet. Eine mögliche Verfeinerung dieses Modells könnte Klassen und Funktionen innerhalb von Codedateien beinhalten, die als eigenständige Wissenselemente behandelt werden können. Das Entscheidungswissen aus Codekommentaren könnte so präziser zu einer bestimm-

ten Klasse oder Funktion zugeordnet werden. Auch könnten `import`-Befehle oder Funktionsaufrufe analysiert und auf dieser Grundlage Codedateien, Klassen und Funktionen geeignet verlinkt werden, was ein detaillierteres Analysieren der Implementierung ermöglicht. Eine solche Funktionalität erfordert jedoch eine weitaus umfangreichere Analyse des Inhaltes von Codedateien als bisher umgesetzt und bleibt daher zukünftigen Projekten vorbehalten.

Auch ist eine weitere Unterstützung von Entwickler:innen bei der Erfüllung der Definition of Done denkbar. Hierfür ließen sich etwa die ConDec-IDE-Plug-ins um eine Anzeige der Definition of Done der jeweils geöffneten Codedatei erweitern. Das Vergeben des Status "Done" an eine Arbeitsaufgabe in Jira kann von der Erfüllung der Definition of Done aller darin veränderten Codedateien abhängig gemacht werden. Weiterhin ließe sich ein Dienst oder ein Plug-in für gängige Git-Server wie GitHub oder GitLab entwickeln, die ein Zusammenführen mit dem Hauptbranch nur erlauben, wenn die Definition of Done aller veränderter Codedateien erfüllt ist. Schließlich ist zur Unterstützer aller Entwickler:innen auch ein Entwickeln von ConDec-Plug-ins für weitere IDEs wie XCode oder IntelliJ vorstellbar.

7.3 Erfahrungen

Während des Entwicklungsprozesses der Erweiterungen der ConDec-Plug-ins konnten verschiedene Erfahrungen gesammelt werden.

Bei der Ermittlung der Anforderungen an die Erweiterungen wurde festgestellt, dass die bestehende Anforderungsdokumentation an verschiedenen Stellen Inkonsistenzen aufwies. Hierbei sind besonders unterschiedliche Nummerierungen der Workspaces, die fehlende Kennzeichnung veralteter Implementierungen oder die verschiedenartige Benennung von Verlinkungen gleicher Issue-Typen und gleicher Bedeutung. Bereits früh im Praktikum wurden diese Feststellungen kommuniziert, Fehler nach Möglichkeit ausgebessert und langfristige Lösungsansätze entworfen. So könnte es hilfreich sein, beim Verweisen auf Arbeitsbereiche oder andere Issues stets den nicht veränderbaren Jira-Issue-Schlüssel zu verwenden. Das bestehende Modell zur „Dokumentation

der Projektartefakte in Jira“[1] ließe sich um Vorgaben zur Linkbezeichnung erweitern. Auch könnte eine Checkliste zur Anforderungsdokumentation für jeden Issue-Typ sicherstellen, dass etwa die richtigen Verlinkungen bestehen, Namenskonventionen eingehalten werden und in den Übersichtstabellen in Confluence sowie in Diagrammen das Issue manuell hinzugefügt wird. Schließlich gefährdet die bestehende Praxis, funktionale Anforderungen und Arbeitsbereiche in separaten Tabellen in Confluence zu listen und diese Tabellen manuell zu pflegen, die Konsistenz der Dokumentation, da das zusätzliche Eintragen oder Ändern von Wissenselementen nach der Dokumentation in Jira an dieser Stelle leicht vergessen werden kann. Hier wird empfohlen, derartige Tabellen von geeigneten Plug-ins basierend auf dem in Jira-Issues gespeicherten Wissen laufend automatisiert generieren zu lassen.

Weiterhin konnte der Bearbeiter im Praktikum viele wichtige Fähigkeiten und Kenntnisse neu kennenlernen oder anwenden. Hierzu gehören das Einarbeiten in bereits bestehende größere Softwareprojekte sowie das kollaborative Arbeiten daran. Das Verwenden von Tools zur kontinuierlichen Integration sowie zur kontinuierlichen Auslieferung wurde im Praktikum erstmals erlernt. Auch das Entwickeln von Jira-Plug-ins und das Zusammenspiel von Front- und Backend über REST-Schnittstellen wurden im Praktikum kennengelernt. Schließlich erlangte der Bearbeiter bei der Entwicklung des Visual-Studio-Code-Plug-ins erstmals Einblicke in die Programmiersprache *TypeScript*, das Verwenden des Paketmanagers *npm* sowie das Testen und dabei auch das Nachbilden etwa von Elementen einer grafischen Benutzeroberfläche mithilfe von Frameworks wie *Mocha* oder *Sinon.JS*.

[1] https://confluence-se.ifi.uni-heidelberg.de/pages/viewpage.action?pageId=10225005#AllgemeineRichtlinienfürFoPras,BachelorundMasterarbeiten-DokumentationderProjektartefakteinJIRA

Kapitel 8

Literatur

[1] A. Kleebaum, J. O. Johanssen, B. Paech und B. Bruegge, „Tool Support for Decision and Usage Knowledge in Continuous Software Engineering,“ in *Combined Proceedings of the Workshops of the German Software Engineering Conference 2018 (SE 2018), Ulm, Germany, March 06, 2018*, S. Krusche, K. Schneider, M. Kuhrmann, R. Heinrich, R. Jung, M. Konersmann, E. Schmieders, M. Striewe, S. Strickroth, U. Lucke, H. Lichter, D. Riehle, A. Steffens, R. Höttger, J. Teßmer und J. Steghöfer, Hrsg., Ser. CEUR Workshop Proceedings, Bd. 2066, CEUR-WS.org, 2018, S. 74–77. DOI: 10.11588/heidok.00024186.

[2] ——, „Continuous Management of Requirement Decisions Using the ConDec Tools,“ in *Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 26th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2020), Pisa, Italy, March 24, 2020*, M. Sabetzadeh, A. Vogelsang, S. Abualhaija, M. Borg, F. Dalpiaz, M. Daneva, N. Condori-Fernández, X. Franch, D. Fucci, V. Gervasi, E. C. Groen, R. S. S. Guizzardi, A. Herrmann, J. Horkoff, L. Mich, A. Perini und A. Susi, Hrsg., Ser. CEUR Workshop Proceedings, Bd. 2584, CEUR-WS.org, 2020. DOI: 10.11588/heidok.00028230.

[3] B. Paech und A. Kleebaum, „Visualisierung von Wissen zu Code in Jira,“ Aufgabenbeschreibung zum Fortgeschrittenenpraktikum, September 2020.

Abbildungsverzeichnis

2.1 Entscheidungswissenselemente dokumentiert in der Beschreibung eines Jira-Issues. [2] 4

2.2 Visualisierung eines Wissensgraphen als Baum. [2] 6

2.3 Visualisierung von Entscheidungswissen in Dashboard. [2] . . . 7

3.1 UI-Strukturdiagramm für ConDec Jira. Rot markierte Elemente wurden im Praktikum hinzugefügt. 27

3.2 UI-Strukturdiagramm für ConDec Eclipse. Rot markierte Elemente wurden im Praktikum hinzugefügt. 28

3.3 UI-Strukturdiagramm für das neu erstellte Plug-in ConDec Visual Studio Code. 28

3.4 Domänendatendiagramm der ConDec-Plug-ins. 31

4.1 Entscheidungsbaum für Entscheidungsproblem zur Beachtung der Richtung von Links. 35

4.2 Klassendiagramm der Wissensgraph-Filterung von ConDec Jira. 37

4.3 Entscheidungsbaum für Entscheidungsproblem zur Wahl der unterstützten Authentifizierungsmethoden. 38

4.4 Eingabemaske für Git-Repositorien mit Authentifizierungsdaten. 40

4.5 Klassendiagramm der Git-Anbindung von ConDec Jira. 41

4.6 Eine Liste der Codedateien im Projekt. 42

4.7 Entscheidungsbaum für Entscheidungsproblem zur Festlegung der Dateiendungen, die als Codedateien zu interpretieren sind. . 43

4.8 Eingabemaske für Dateiendungen. 44

4.9 Klassendiagramm des Wissensmodells von ConDec Jira. 45

4.10 Klassendiagramm der REST-Schnittstellen von ConDec Jira. . . 46

4.11 Entscheidungsbaum für Entscheidungsproblem zur Festlegung der Persistenzstrategie für Entscheidungswissenselemente aus Code-Kommentaren. 47

4.12 Subgraph des Wissensgraphen mit Entscheidungswissenselementen aus Code-Kommentaren. 49

4.13 Darstellung der Rationale Coverage von Codedateien in einem Dashboard. 51

4.14 Überblicksklassendiagramm der Berechnung der in Dashboards visualisierten Daten von ConDec Jira. 52

4.15 Konfigurationsmöglichkeiten der Definition of Done. 52

4.16 Klassendiagramm der Prüfungen auf Vollständigkeit von ConDec Jira. 55

4.17 Kontextmenüpunkt in Eclipse zur Navigation zum ConDec-Jira-Plug-in. 56

4.18 Statusleistenelement in Visual Studio Code zur Navigation zum ConDec-Jira-Plug-in. 56

4.19 Klassendiagramm von ConDec Visual Studio Code. 57

6.1 Konfiguration der Git-Extraktion. 67

6.2 Wissensgraph mit Entscheidungswissen, zentriert auf die Codedatei `GitClient.java`. 68

6.3 Dashboard mit Metriken zur Rationale Coverage von Codedateien. 69

6.4 Liste an Codedateien ohne dokumentiertes Entscheidungsproblem als Detailansicht des Dashboards. 69

6.5 Rationale Backlog vor (oben) und nach (unten) Anpassung der Definition of Done. 70

6.6 Anpasste Definition of Done für Codedateien. 71

6.7 Systemfunktion mit transitiven Links zu Codedateien. 71

6.8 Schaltfläche zur Navigation von Eclipse zu Jira. 73

6.9 Eingabe- und Informationsfenster bei der Navigation von Visual Studio Code zu Jira. 74

6.10 Subgraph des Wissensgraphen zur Codedatei `AdditionalConfigurationOptions.java`. 74

Tabellenverzeichnis

3.1 Beispielhafte Persona für die Rolle „Entwickler:in“. 14

3.2 Beispielhafte Persona für die Rolle „Requirements Engineer“. . . 15

3.3 Beispielhafte Persona für die Rolle „Rationale Manager“. 16

3.4 Übersicht über Systemfunktionen und Grobanforderungen innerhalb der User Tasks und Subtasks. 17

3.5 SF1: Filter knowledge graph. 18

3.6 SF2: Configure decision knowledge extraction from git (commit messages and code comments) 19

3.7 SF3: List all code classes for a project 19

3.8 SF4: Automatically add code files from git repository into the knowledge graph 20

3.9 SF5: Automatically add and link decision knowledge from comments in code files into the knowledge graph 21

3.10 SF6: Show knowledge graph metrics with respect to code files . 22

3.11 SF7: Configure definition of done (DoD) for the decision knowledge documentation 23

3.12 SF8: Check definition of done (DoD) of the decision knowledge documentation related to a code file 24

3.13 SF9: Navigate to subgraph centered on a code file in Jira in Eclipse 25

3.14 SF9: Navigate to subgraph centered on a code file in Jira in Visual Studio Code 25

3.15 SF10: Configure Jira Settings [in VS Code] 26

4.1 Kommentartypen und zugehörige Kommentarsyntax. 48