Vita Aman

Unterstützung der Konsistenz zwischen Entscheidungen und ihrer Umsetzung durch Zusammenfassung von Codeänderungen

Bachelorarbeit

Wintersemester 2018/19
Betreuung: Prof. Dr. Barbara Paech, Anja Kleebaum
Institut für Informatik
Ruprecht-Karls-Universität Heidelberg

1. Februar 2019

Zusammenfassung

Ein zentraler Bestandteil bei der Entwicklung eines Softwaresystems ist die Dokumentation der zugehörigen Softwareartefakte. Die Dokumentation ist wichtig, damit EntwicklerInnen die Software gut evolvieren und so die Qualität der Software aufrecht erhalten können. Insbesondere wenn EntwicklerInnen nicht von Beginn an in einem Softwareprojekt tätig waren, fehlt ihnen das Wissen über die zuvor getroffenen Entscheidungen. Dieses Wissen wird als Entscheidungswissen bezeichnet. Neben der getroffenen Entscheidung umfasst das Entscheidungswissen auch das dahinterstehende Entscheidungsproblem, alternative Lösungsmöglichkeiten, sowie deren Vor- und Nachteile. Entscheidungswissen ist ein wichtiges Dokumentationsartefakt. Damit EntwicklerInnen das dokumentierte Entscheidungswissen gut nutzen können, sollte es möglichst vollständig und konsistent dokumentiert sein. Konsistenz heißt dabei zum einen, dass das Entscheidungswissen in sich richtig dokumentiert ist, das heißt, dass beispielsweise die Entscheidung das Entscheidungsproblem löst und nicht ein ganz anderes Problem. Zum anderen muss die dokumentierte Entscheidung konsistent zu ihrer Umsetzung sein, das heißt beispielsweise, dass Entwurfs- und Implementierungsentscheidungen im Code umgesetzt sein müssen. Um die Konsistenz zwischen Codeänderungen und den getroffenen Entscheidungen zu prüfen, müssen EntwicklerInnen bzw. der/die Rationale ManagerIn unter Umständen sehr viele Änderungen im Code überblicken, was sehr aufwändig sein kann. Daher ist es Ziel dieser Arbeit eine Zusammenfassung von Codeänderungen zu schaffen und somit EntwicklerInnen und den/die Rationale ManagerIn dabei zu unterstützen, die Konsistenz zwischen Entscheidungswissen und Code aufrecht zu erhalten. Dafür wird in dieser Arbeit das Jira Plugin ConDec erweitert. Eine erste Evaluation der in der Arbeit entwickelten Erweiterung zeigt, dass die Anzeige von zusammengefassten Codeänderungen nützlich sein kann, um die Konsistenz zwischen Entscheidungen und Code zu unterstützen.

Abstract

An essential part of the development of a software system is the documentation of its software artifacts. The documentation is important in order to support developers in evolving the software and in maintaining its quality. If developers are new to a software project, they lack knowledge about former decisions. This knowledge is called decision knowledge. Decision knowledge covers decisions, the issues that the decisions solve, alternatives, and arguments. For future use, decision knowledge needs to be documented and the documentation should be of a high quality, i.e., complete and consistent. On the one hand, the elements of the decision knowledge need to be consistent, e.g., the documented decision should solve the issue and not a different issue. On the other hand, the decisions need to be consistent with the software artifacts such as the source code. In order to check that the code changes are consistent with the decisions made, the developers and the rationale manager might have to scan many code changes, which can take a lot of time. In order to make this task easier, this thesis aims to support the developers and the rationale manager in checking the consistency of decisions and code changes by presenting them with summarized code changes. For this purpose, this thesis extends the ConDec Jira plugin. A first evaluation of this extension shows that the presentation of summarized code changes can support the consistency between decisions and code changes.

Inhaltsverzeichnis

1 Einleitung **3**
1.1 Motivation 3
1.2 Zielsetzung 4
1.3 Struktur der Arbeit 4

2 Grundlagen **5**
2.1 Modell zur Dokumentation von Wissen 5
2.2 Das ConDec-Projekt 6
2.3 Codezusammenfassung mit *ChangeScribe* 6
2.4 Technische Grundlagen 7
2.4.1 Issue-Tracking-System Jira 7
2.4.2 Versionskontrollsystem Git 7

3 Anforderungen **8**
3.1 Grobanforderungen 8
3.2 Personae 9
3.3 Funktionale Anforderungen 10
3.3.1 User Task 10
3.3.2 User Subtask 10
3.3.3 Systemfunktionen 11
3.4 Domänendatendiagramm 14
3.5 Nutzungsdiagramm 15
3.6 Arbeitsbereiche 15
3.7 Nichtfunktionale Anforderungen 16
3.7.1 Funktionale Angemessenheit 17
3.7.2 Benutzbarkeit 17
3.7.3 Wartbarkeit 17

4 Literaturrecherche **18**
4.1 Forschungsfrage 18
4.2 Methodik 18
4.3 Übersicht relevanter Literatur 19
4.4 Ergebnisse 22
4.4.1 Event Pipeline Pattern 25
4.4.2 Model-View-Controller 26
4.4.3 Model-View-View Model 26
4.4.4 Model-View-Presenter 27
4.4.5 Observer Pattern 29
4.4.6 Event Processing Pattern 31

4.5 Synthese 32
4.6 Erkenntnisse für die Arbeit 35

5 Entwurf und Implementierung **36**
5.1 Umsetzung erster Grobanforderungen 36
5.2 Umsetzung funktionaler Anforderungen 37
5.2.1 Systemfunktion 1 37
5.2.2 Systemfunktion 2 39
5.2.3 Systemfunktion 3 40
5.2.4 Systemfunktion 4 41
5.3 Addressierung nichtfunktionaler Anforderungen 43
5.3.1 Funktionale Angemessenheit 43
5.3.2 Benutzbarkeit 43
5.3.3 Wartbarkeit 44
5.4 UML-Klassendiagramm 44
5.5 Ergebnis 46

6 Qualitätsicherung **53**
6.1 Maßnahmen zur Qualitätssicherung 53
6.2 Komponententests 53
6.3 Systemtests 55
6.4 Statische Codeanalyse 57

7 Evaluation **59**
7.1 Evaluationsfragen 59
7.2 Vorgehen 60
7.3 Evaluationsergebnisse 60
7.4 Diskussion 64

8 Schlussfolgerung **67**
8.1 Zusammenfassung 67
8.2 Diskussion 67
8.3 Ausblick 68

9 Literatur **70**

Abbildungsverzeichnis **72**

Tabellenverzeichnis **73**

Abkürzungsverzeichnis **74**

1 Einleitung

Diese Bachelorarbeit wird mit einer Motivation im ersten Abschnitt dieses Kapitels eingeleitet. Abschnitt 2 erläutert die Zielsetzung dieser Arbeit und Abschnitt 3 beschreibt die entsprechende Struktur.

1.1 Motivation

Bei der agilen Softwareentwicklung führen EntwicklerInnen kontinuierlich Änderungen an einer Software durch und können auch mit neuen oder sich ändernden Anforderungen umgehen. Während sie die Software ändern, müssen sie gleichzeitig garantieren, dass die Software jederzeit auslieferbar ist. Dafür muss die Software eine hohe Qualität besitzen [9]. Um diese hohe Qualität zu gewähren, benötigen EntwicklerInnen Wissen über getroffene Entscheidungen. Im Laufe der Entwicklung von Software, sollten Entscheidungsprobleme, ihre alternativen Lösungsmöglichkeiten mit den Argumenten und die endgültige Entscheidung dokumentiert werden. Das dient dazu Wissen auch zukünftigen EntwicklerInnen zugänglich zu machen und den Entscheidungsprozess dadurch zu unterstützen. Dieses Wissen wird als Entscheidungswissen bezeichnet [8]. Der Entscheidungsprozess wird jedoch bisher häufig nicht dokumentiert. Ein Grund dafür könnte sein, dass EntwicklerInnen dafür keine gute Werkzeugunterstützung haben.
Während der Softwareentwicklung werden häufig Issue-Tracking- und Versionskontrollsysteme verwendet. Ein Issue-Tracking-System (ITS) dient zur Dokumentation von zu tätigenden Änderungen, Änderungen die bereits ausgeführt wurden, sowie von Entscheidungswissen. Ein Beispiel für ein Issue-Tracking-System ist Jira von Atlassian. Zudem unterstützt es Versionskontrollsysteme (Version Control System (VCS)), wie die Open-Source-Software Git, um Änderungen am Code in "Commits" zusammenzufassen und mittels "Commitnachrichten" zu dokumentieren.
EntwicklerInnen können sich Codeänderungen einzelner Commits, die zu einer Aufgabe gehören, anschauen, um z.B. nachzuvollziehen, ob die Aufgabe gut umgesetzt wurde und ob die Codeänderungen konsistent zu den getroffenen und dokumentierten Entscheidungen ist. Rationale ManagerInnen sind zusätzlich dafür verantwortlich, Softwareartefakte und Entscheidungswissen jederzeit aktuell und konsistent zu halten. Auch sie können mithilfe von Codeänderungen einzelner Commits die Aufgabenumsetzung nachvollziehen. Wenn die verlinkten Codeänderungen jedoch sehr groß sind, kann es sehr aufwändig sein, dies zu prüfen.
Zur Unterstützung von EntwicklerInnen beziehungsweise Rationale ManagerInnen könnte eine Zusammenfassung der zu einer Entwicklungsaufgabe zugehörigen Codeänderungen dienen. Somit können durch einen Blick alle Beteiligte die Konsistenz zwischen

dem zu der Entwicklungsaufgabe dokumentiertem Entscheidungswissen und den Codeänderungen überprüfen. Konsistenz heißt hierbei, ob im Code Änderungen durchgeführt wurden, die auch als Entscheidungen getroffen und dokumentiert sind. Wenn die Entscheidungen von der Umsetzung abweichen, liegt eine Inkonsistenz vor. Mit dieser Information können die Beteiligten entweder den Code noch einmal bearbeiten oder die Dokumentation des Entscheidungswissens aktualisieren.
Es gibt verschiedene Ansätze, die sich mit der Zusammenfassung von Codeänderungen beschäftigen. Ein Ansatz ist der *ChangeScribe*-Algorithmus [11]. *ChangeScribe* beschreibt die Codeänderungen in natürlicher Sprache.

1.2 Zielsetzung

Um die Konsistenz zwischen Entscheidungen und Codeänderungen zu erhalten, ist es Ziel dieser Arbeit, einen Prototypen zu entwickeln. Der Prototyp erstellt eine Zusammenfassung der zu einem Jira Issue zugehörigen Codeänderungen und soll es EntwicklerInnen und der Rationale ManagerIn ermöglichen, diese betrachten. Um eine möglichst generische Anzeige umzusetzen, wird hierfür durch eine Literaturrecherche ein Entwurfsmuster ausgesucht. Die Anzeige zur Codezusammenfassung soll ereignisbasiert sein, um EntwicklerInnen dazu zu bringen, es auch wirklich zu nutzen. Somit ist die Konsistenzprüfung im Entwicklungsprozess integriert. Anschließend wird darauf aufbauend ein Entwurf zur Entwicklung der Anzeige der Codezusammenfassung im Jira Issue entwickelt. Um die Eignung dieser Funktionalität zu bewerten, wird der Prototyp in dieser Arbeit mithilfe von NutzerInnen evaluiert.

1.3 Struktur der Arbeit

Die Bachelorarbeit ist in verschiedenen Kapitel untergliedert. Kapitel 2 beschreibt die Grundlagen der Arbeit. In Kapitel 3 sind die für das Plugin benötigten Anforderungen aus funktionaler und nicht funktionaler Sicht aufgelistet. Anschließend wird in Kapitel 4 mit Hilfe einer Literaturrecherche ein Entwurfsmuster für die ereignisbasierte Anzeige gesucht. Nach einem Vergleich wird das am besten geeignete Muster für den Entwurf ausgesucht. Daraufhin wird in Kapitel 5 mithilfe der in Kapitel 3 beschriebenen Anforderungen ein Entwurf für die Implementierung des Plugins entwickelt und die auf den Entwurf angepassten Implementierung mit einem Überblick über die Klassen sowie ihre Zusammenhänge erläutert. In Kapitel 6 wird aufgrund eines Testverfahrens die Qualitätssicherung für die Software beschrieben. In Kapitel 7 wird anknüpfend daran eine Evaluation auf Basis von Evaluationsfragen durchgeführt. Kapitel 8 enthält zum Abschluss eine Zusammenfassung mit einer Schlussfolgerung für die Arbeit, einer Diskussion und einem Ausblick auf zukünftige Weiterentwicklungen.

2 Grundlagen

In diesem Kapitel werden die nötigen Grundlagen für das Verständnis und die Weiterentwicklung des ConDec Plugins vorgestellt. Dafür wird im ersten Abschnitt das Modell zur Dokumentation von Wissen beschrieben. Anschließend wird im zweiten Abschnitt das ConDec-Projekt erläutert. Im dritten Abschnitt wird die Codezusammenfassung mit ChangeScribe beschrieben. Im vierten Abschnitt werden schließlich die technischen Grundlagen erläutert.

2.1 Modell zur Dokumentation von Wissen

Entscheidungswissen kann entweder in natürlicher Sprache oder in einem Modell festgehalten werden. Um Entscheidungswissen in einem Modell abzubilden, haben Hesse et al. [6] das Entscheidungsdokumentationsmodell erstellt (Abbildung 2.1).

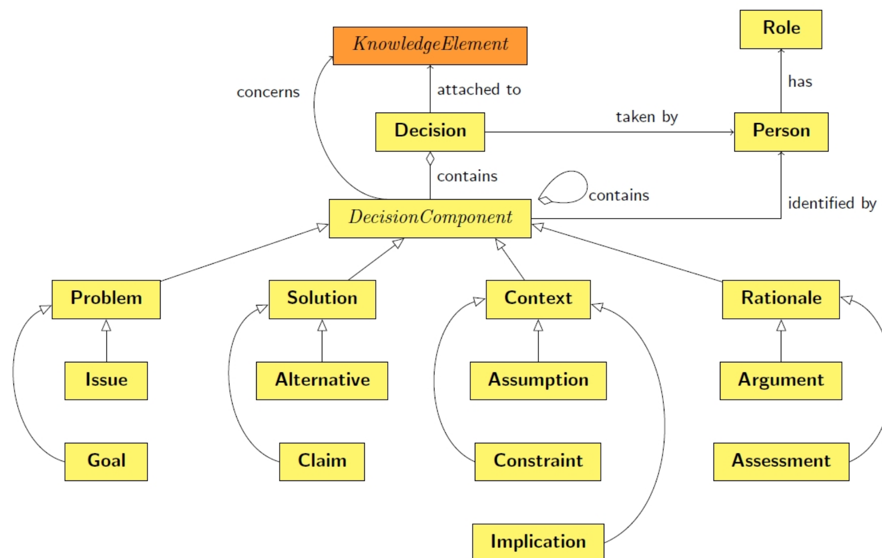


Abbildung 2.1: Entscheidungsdokumentationsmodell [6]

Dieses Modell unterscheidet zwischen dem Entscheidungsproblem, der Lösung, Kontextinformation und den Begründungen der Entscheidung (Rationale). Das Entscheidungsdokumentationsmodell ermöglicht es EntwicklerInnen Entscheidungswissen inkrementell und kollaborativ zu dokumentieren [6].

Kleebaum et al. [8] hat ein Modell für die Beziehungen zwischen dem Entscheidungsdokumentationsmodell von Hesse et al. zu den grundlegenden Softwareartefakten aufgezeigt (Abbildung 2.2). Entscheidungswissen kann sich auf Anforderungen (Features) und deren Umsetzung (Code) beziehen. Es kann beispielsweise in Entwicklungsaufgaben (Feature Tasks, z.B. in den Kommentaren von Jira Issues) und Commits festgehalten werden.

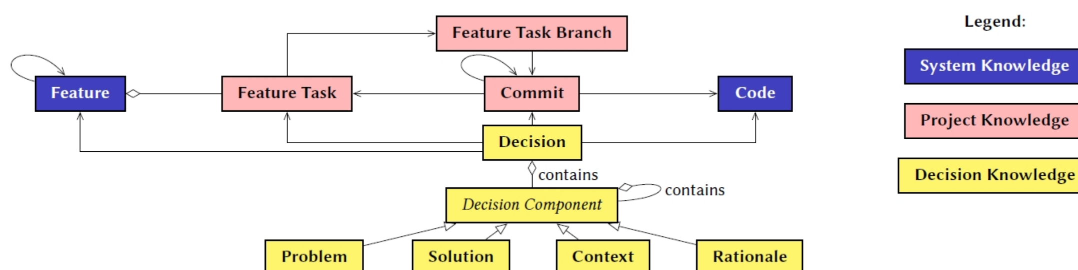


Abbildung 2.2: Beziehungen zwischen Softwareartefakten und Entscheidungswissen [8]

2.2 Das ConDec-Projekt

Damit EntwicklerInnen Entscheidungswissen während der agilen, kontinuierlichen Softwareentwicklung gut dokumentieren und nutzen können, sammeln Kleebaum et al. [9] Ideen und Anforderungen an eine Tool-Unterstützung zum Verwalten von Entscheidungswissen. Diese Ideen und Anforderungen werden im Forschungsprojekt CURES fortlaufend umgesetzt und als Continuous Decision Knowledge Management (ConDec) bezeichnet. ConDec ermöglicht es EntwicklerInnen Entscheidungswissen in Modellform festzuhalten (beispielsweise wie in dem in Abbildung 2.1 zu sehenden Entscheidungsdokumentationsmodell von Hesse et al.) und bietet verschiedene Möglichkeiten, Entscheidungswissen zu verwalten und zu betrachten.

2.3 Codezusammenfassung mit *ChangeScribe*

Während des Entwicklungsprozesses führen EntwicklerInnen Codeänderungen durch, jedoch werden diese nur teilweise dokumentiert. In Commitnachrichten können Änderungen des Codes beschrieben werden. Dabei kann direkt Entscheidungswissen in der Commitnachricht festgehalten werden und bei anschließender Dokumentation von Entscheidungswissen können diese Nachrichten als Unterstützung dienen. Das ist der Hintergrund des Plugins *ChangeScribe* für die integrierte Entwicklungsumgebung IDE Eclipse [11]. Es werden die Unterschiede zwischen zwei Code-Versionen extrahiert und analysiert. Anschließend stellt es eine Übersicht der Änderungen zur Verfügung, klassifiziert sie und beschreibt die Details aller Änderungen. Somit wird in natürlicher Sprache ein Text verfasst, der auch versucht die Gründe der Codeänderungen zu beschreiben.

Der Algorithmus hinter *ChangeScribe* könnte genutzt werden, um Zusammenfassungen von Codeänderungen in einem Issue-Tracking-System zu erzeugen.

2.4 Technische Grundlagen

2.4.1 Issue-Tracking-System Jira

Jira ist eine webbasierte Anwendung zur projektspezifischen Dokumentation von Softwareartefakten [1]. Es können Anforderungen an ein Softwareprojekt mithilfe von sogenannten "Jira Issues" dokumentiert werden. Es gibt verschiedene Typen von Jira Issues, wie zum Beispiel "System Functions" oder "WorkItems". Diese Jira Issue werden mit einer einmaligen Identifikationsnummer versehen, die den Kürzel des Projektnamens und eine sich inkrementierende Nummer beinhalten. Mit dieser "Issue-ID" und dem Kürzel des Projektes in Jira kann eine Verbindung zwischen den Jira Issues und den Codeänderungen hergestellt werden.

2.4.2 Versionskontrollsystem Git

Git ist ein verbreitetes Versionskontrollsystem und dient zur Verwaltung von Dateien[2]. Es dient dazu, größere Sicherheit vor Ausfällen von Speichersystemen zu gewährleisten und gemeinsames Arbeiten mehrerer EntwicklerInnen an einem Projekt zu vereinfachen. Dateien werden dafür im Gitsystem in Repositorien ("Repositories") verwaltet. Diese Verwaltung ist auf einem für alle beteiligten EntwicklerInnen zugänglichen Server gespiegelt. Durch das "Klonen" bekommt jede EntwicklerIn eine vollständige Kopie des Repositoriums und durch das Persistieren einer Änderung an dieser Datei mittels "Commit"- und "Push"-Befehl wird ein aktueller Stand der Datei auf den Server geladen. Hiermit ist sie für alle EntwicklerInnen verfügbar. Ein Commit wird üblicherweise mit einer "Commitnachricht" versehen, die Auskunft über getätigte Änderungen gibt. Wird in einer Commitnachricht eine Issue-ID angegeben, so entsteht eine Verlinkung zwischen Jira Issue und Commit.

[1] `https://de.atlassian.com/software/jira`, eingesehen am 31.01.2019
[2] `https://git-scm.com`, eingesehen am 31.01.2019

3 Anforderungen

In diesem Kapitel werden die Anforderungen für die Implementierungen zusammengefasst. Im ersten Abschnitt werden die Grobanforderungen beschrieben. Anschließend werden im zweiten Abschnitt mögliche NutzerInnen genannt. Der dritte Abschnitt erläutert die funktionalen Anforderungen. Des weiteren werden in Abschnitt vier das Domänendatendiagramm, in Abschnitt fünf das Nutzungsdiagramm und im Abschnitt sechs die erforderlichen Ansichten beschrieben. Zu guter Letzt werden im siebten Abschnitt die nicht funktionalen Anforderungen definiert.

3.1 Grobanforderungen

Es folgt eine Beschreibung der betrachteten Szenarien aus der Ausschreibung.

Szenario 1: EntwicklerInnen beginnen eine Aufgabe (Work Item, auch als Feature Task bezeichnet) im ITS. Ihnen werden das bei der Umsetzung vorangegangener Aufgaben dokumentierte Entscheidungswissen und der zusammengefasste Code präsentiert. Sie verwenden das präsentierte Wissen bei ihrer Aufgabe und können das Wissen editieren (Elemente einfügen/bearbeiten/löschen sowie Verlinkungen hinzufügen und löschen).

Szenario 2: EntwicklerInnen beenden eine Aufgabe im ITS. Ihnen wird der bei der Umsetzung dieser Aufgabe entstandene zusammengefasste Code präsentiert. Sie stellen sicher, dass wichtiges Entscheidungswissen dokumentiert und konsistent mit diesem Code ist. Sie können das Wissen editieren (Elemente einfügen/bearbeiten/löschen sowie Verlinkungen hinzufügen und löschen).

Dabei entsteht folgende Grobanforderung (R):

Grobanforderung R1: Eine generische ereignisbasierte Anzeige soll durch die von der EntwicklerIn ausgelösten Ereignisse "Beginnen" und "Beenden" angezeigt werden.

Grobanforderung R2: Die ereignisbasierte Anzeige soll die bei der Umsetzung vorangegangener Aufgaben dokumentierter Entscheidungswissen der EntwicklerIn angezeigt werden.

Grobanforderung R3: Das Entscheidungswissen vorangeganger Aufgaben soll von der EntwicklerIn in der Anzeige editiert werden können. Unter Editieren zählen einfügen, bearbeiten, löschen sowie Verlinkungen hinzufügen oder löschen.

Grobanforderung R4: Die ereignisbasierte Anzeige soll eine Zusammenfassung des bei der Umsetzung vorangegangener Aufgaben entstandenen Codes der EntwicklerIn angezeigt werden.

Durch die im Kapitel "5 Entwurf und Implementierung" beschriebenen Probleme und Entscheidungen, wurden die Grobanforderungen jedoch an die folgenden Szenarien und Grobanforderungen angepasst.

Szenario 1: EntwicklerInnen beenden eine Aufgabe im ITS. Ihnen wird der bei der Umsetzung dieser Aufgabe entstandener zusammengefasster Codeänderungen automatisch in die Kommentare der Aufgabe erzeugt. Sie stellen sicher, dass wichtiges Entscheidungswissen dokumentiert und konsistent mit der Codeänderung ist.

Szenario 2: EntwicklerInnen bearbeiten eine Aufgabe im ITS. Sie stellen sicher, dass wichtiges Entscheidungswissen dokumentiert und konsistent mit der Codeänderung ist.

Szenario 3: Die Rationale ManagerIn des Projektes prüft die Konsistenz zwischen Codeänderungen und Entscheidungswissen, um die Qualität zu sichern.

Es resultieren folgende neue Grobanforderungen.
Grobanforderung R1: Eine Zusammenfassung von allen geänderten Klassen und Methoden soll durch die von der EntwicklerIn ausgelöstes Ereignis "Beenden" ("Done") in der Kommentarfunktion des entsprechenden Jira Issues eingefügt werden.

Grobanforderung R2: Eine ereignisbasierte Anzeige soll mit der Zusammenfassung aller geänderten Klassen und Methoden von allen verlinkten Jira Issues zu sehen sein.

Grobanforderung R3: Die EntwicklerIn soll die in den Grobanforderungen R1 und R2 beschriebenen Funktionalität der Codezusammenfassung ein- und ausschalten können.

3.2 Personae

Im Folgenden werden die Personae vorgestellt, die die Eigenschaften der Grobanforderungen für das Plugin benötigen.

P.H. ist eine 22-Jährige Entwicklerin. Sie arbeitet seit zwei Monaten in einer Firma, die bereits seit 13 Jahren Jira als ITS nutzt. Sie selbst hat noch nie Jira vorher genutzt und weiß auch wenig über den bereits vorhandenen Code oder dessen Entscheidungswissen. Außerdem hat sie noch nie Entscheidungswissen dokumentiert. Sie braucht ein Plugin, welches einfach zu benutzen ist und keine zu große Anleitungen braucht. Außerdem hätte sie gerne ein Plugin, welches alle notwendige Informationen in einer Anzeige darstellt. Das Plugin sollte zudem noch ermöglichen, dass Entscheidungswissen einfach vor und nach dem Implementieren editiert werden kann. Sie wird frustriert, wenn die Suche von benötigten Informationen über mehrere Seiten navigiert wird. Sie möchte

also ein Plugin, welches mit einem Klick alle Informationen zu Codeänderungen und Entscheidungwissen anzeigt.

R.R. ist ein 31-jähriger Rationale Manager. Er arbeitet seit drei Jahren mit Jira und ist in mehreren Projekten als Rationale Manager tätig. Seine Aufgaben bestehen darin, eine Übersicht über alle Entscheidungen zu behalten und neues Entscheidungswissen, das verbal, über andere Medien oder auch während Scrum Meetings, entstanden ist, zu dokumentieren und zu verlinken. Dafür braucht er eine Übersicht über das gesamte Entscheidungswissen und deren Umsetzung. Es frustriert ihn, wenn die Stakeholder sich nicht an Entscheidungen bei der Umsetzung halten. Daher braucht er ein Plugin, welches den aktuellen Stand der Implementierung anzeigt und somit eine Analyse über die Konsistenz zwischen Codeänderungen und Entscheidung durchführen kann.

3.3 Funktionale Anforderungen

Im Folgenden sind die von den Grobanforderungen abgeleiteten funktionalen Anforderungen aufgezählt.

3.3.1 User Task

Aus den Grobanforderungen wurde folgende User Task (UT) abgeleitet.

User Task UT1: Software implementieren
Um Anforderungen umzusetzen, wird die Software entwickelt. Hierzu muss auf vorherigen Codeänderungen und Entscheidungswissen zurückgegriffen werden. EntwicklerInnen sollen beim Beenden einer Entwicklungsaufgabe einen Überblick über das Entscheidungswissen und den implementierten Codeänderung erhalten.

User Task UT2: Qualität der Software sichern
Um eine konstant laufende Software zu garantieren, muss die Qualität gesichert werden. Hierzu muss auf Codeänderung und Entscheidungswissen zurückgegriffen werden. Rationale ManagerInnen sollen während der Implementierung und beim Beenden einer Entwicklungsaufgabe einen Überblick über das Entscheidungswissen und den implementierten Codeänderung erhalten.

3.3.2 User Subtask

Die User Task lassen sich in folgende Unteraufgaben User Subtask (ST) aufgliedern.

User Subtask ST1: Software Entwicklungsprozess definieren
Die AdministratorInnen definieren den Software Entwicklungsprozess für das Projekt. Sie entscheiden, wie das Entscheidungswissen im Software Entwicklungsprojekt ver-

waltet wird. Dazu konfigurieren und verwalten die ProjektadministratorInnen und die SystemadministratorInnen die Eigenschaften des Plugins.

User Subtask ST2: Ändere den Zustand des Arbeitsablaufs
Vor dem entwickeln der Codeänderungen für die Software muss der Beginn dokumentiert werden. Dazu ändert man den Zustand des Arbeitsablaufs ("Workflow") in Jira von "Aufgaben" zu "Wird Ausgeführt". Somit kann die EntwicklerIn mit dem Bearbeiten der Aufgabe beginnen. Auch beim Beenden der Aufgabe, muss der Zustand von "Wird Ausgeführt" zu "Fertig" geändert werden.

User Subtask ST3: Code implementieren
Während die Aufgabe ausgeführt wird, implementiert die EntwicklerIn die Codeänderungen für diese Aufgabe. Die Codeänderungen wird Abschnittsweise commitet und auf dem Repository hochgeladen. In der Commitnachricht ist die Identifikationsnummer der Aufgabe in Jira enthalten, um eine Verlinkung zwischen Codeänderungen und Aufgabe zu ermöglichen.

User Subtask ST4: Qualität von dokumentierten Entscheidungswissen analysieren
Die Rationale ManagerIn und die EntwicklerIn wird während und nach der Bearbeitung der Aufgabe die Qualität des Entscheidungswissens analysieren. Dabei betrachtet er, ob eine Konsistenz zwischen dem bis zur Analyse entstandenen Codeänderungen zum Entscheidungswissen aufweist.

3.3.3 Systemfunktionen

Aus den Grobanforderungen haben sich somit folgende Systemfunktion (SF) gebildet.

Systemfunktion SF1: Extrahiere das Git Repository vom Software Entwicklungsprojekt
Bei einem vorhandenen Software Entwicklungsprojekt ist die Voraussetzung, dass ein für die Öffentlichkeit frei lesbares Git Repository verlinkt ist. Beim Einschalten der Funktionalität "Extraktion von Wissen aus Git?" entstehen keine Änderungen im System. Im Hintergrund wird das entsprechende Git Repository heruntergeladen. Sollte kein Git Repository verlinkt sein, so geschieht auch nichts im Hintergrund. Zu sehen sind die Vorbedingungen, die Eingaben, die Nachbedingungen, die Ausgabe und die Regel zusätzlich in Tabelle 3.1. Hierbei wird "User Subtask ST1: Software Entwicklungsprozess definieren" als Grundlage gesetzt.

Tabelle 3.1: SF1: Extrahiere das Git Repository vom Software Entwicklungsprojekt

Vorbedingung	Jira Projekt existiert, ConDec ist aktiviert
Eingabe	Git Extraktion Funktion wird an-/ausgeschaltet
Nachbedingung	Git Extraktion wurde an-/ausgeschaltet, Projekt-Repository wurde geklont
Ausgabe	Keine Ausgabe
Regel	Wenn kein Projekt-Repository vorhanden ist, so wird die Extraktion ohne das Klonen vom Repository an- und ausgeschaltet

Systemfunktion SF2: Erstelle Zusammenfassung von Codeänderungen
Ein Jira Issue enthält Codeänderungen welches auf Git commmitet wurde. Beim Öffnen des Kontextmenüs und Anklicken der Funktionalität zur Anzeige von den zusammengefassten Codeänderungen oder dem Schließen einer Aufgabe, ändert sich im System nichts. Die Ausgabe ist ein Dialog mit dem zusammengefassten Codeänderungen der ausgewählten verlinkten Aufgabe. Sollten keine Codeänderungen in der ausgewählten verlinkten Aufgabe vorhanden sein, so entsteht auch keine Codezusammenfassung. Tabelle 3.2 zeigt zusätzlich die Vorbedingungen, die Eingaben, die Nachbedingungen, die Ausgabe und die Regel auf. Betrachtet wird hier der Subtask "User Subtask ST4: Qualität von dokumentierten Entscheidungswissen analysieren".

Tabelle 3.2: SF2: Zeige zusammengefasste Codeänderungen von verlinkten Anforderungen, Aufgaben oder Entscheidungswissen im Entscheidungsbaum

Vorbedingung	Jira Issue existiert, Codeänderungen wurden in git commitet und gepusht, Codeänderungen sind mit dem Jira Issue verlinkt, ConDec ist aktiviert, Git Extraktion ist aktiviert, Entscheidungsbaum ist zu sehen oder Jira Issue kann geschlossen werden
Eingabe	Ereignis zur Anzeige für die Codezusammenfassung wird im Kontextmenü ausgeführt oder das Jira Issue wird beendet.
Nachbedingung	Keine Ausgabe
Ausgabe	Codezusammenfassung wird angezeigt
Regel	Sollte keine Codeänderungen commitet worden sein, so sollte auch keine Codezusammenfassung in der Anzeige zu sehen sein.

Systemfunktion SF3: Zeige zusammengefasste Codeänderungen von verlinkten Anforderungen, Aufgaben oder Entscheidungswissen im Entscheidungsbaum
Der Entscheidungsbaum enthält mindestens eine weitere Aufgabe, welcher mit der gerade zu bearbeitenden Aufgabe verlinkt ist. Beim Öffnen des Kontextmenüs und Anklicken der Funktionalität zur Anzeige vom zusammengefassten Codeänderungen, ändert sich im System nichts. Die Ausgabe ist ein Dialog mit dem zusammengefassten Code-

änderungen der ausgewählten verlinkten Aufgabe. Sollte keine Codeänderungen in der ausgewählten verlinkten Aufgabe vorhanden sein, so soll auch eine entsprechende Meldung angezeigt werden. In Tabelle 3.3 werden die Vorbedingungen, die Eingaben, die Nachbedingungen, die Ausgabe und die Regel für diese Systemfunktion aufgezeigt. Betrachtet werden hier die Subtasks "User Subtask ST3: Code implementieren" und "User Subtask ST4: Qualität von dokumentierten Entscheidungswissen analysieren".

Tabelle 3.3: SF3: Zeige zusammengefassten Codeänderungen von verlinkten Anforderungen, Aufgaben oder Entscheidungswissen im Entscheidungsbaum

Vorbedingung	Jira Issue existiert, Codeänderungen werden in git commitet und gepusht, Codeänderungen sind mit dem Jira Issue verlinkt, ConDec ist aktiviert, Git Extraktion ist aktiviert, Entscheidungsbaum ist zu sehen
Eingabe	Ereignis zur Anzeige für die Codezusammenfassung wird im Kontextmenü ausgeführt
Nachbedingung	Nichts ändert sich
Ausgabe	Anzeige mit Codezusammenfassung wird angezeigt
Regel	Sollte kein Codeänderungen commitet worden sein, so sollte auch keine Codezusammenfassung in der Anzeige zu sehen sein.

Systemfunktion SF4: Zeige zusammengefasste Codeänderungen von Anforderungen, Aufgaben oder Entscheidungswissen beim Beenden einer Aufgabe
Beim Beenden einer Aufgabe in Jira wird der Zustand des Workflows auf "Fertig" gesetzt. Dies wird vom System als Ereignis interpretiert. Beim Klick auf den entsprechenden Knopf soll somit ein Kommentar in der Kommentarfunktion der Aufgabe erscheinen, das eine Zusammenfassung von Codeänderungen der verlinkten Anforderungen, Aufgaben oder Entscheidungswissen anzeigt. Zusätzlich werden in Tabelle 3.4 die Vorbedingungen, die Eingaben, die Nachbedingungen, die Ausgabe und die Regel aufgelistet. Die User Subtasks "User Subtask ST2: Ändere den Zustand des Arbeitsablaufs", "User Subtask ST3: Code implementieren" und "User Subtask ST4: Qualität von dokumentierten Entscheidungswissen analysieren" werden hier erfüllt.

Tabelle 3.4: SF4: Zeige zusammengefasste Codeänderungen von Anforderungen, Aufgaben oder Entscheidungswissen beim Beenden einer Aufgabe nach dem Beenden

Vorbedingung	Jira Issue existiert, Codeänderungen wurden in git commitet und gepusht, Codeänderungen sind mit dem Jira Issue verlinked, ConDec ist aktiviert, Git Extraktion ist aktiviert
Eingabe	Das Jira Issue wird beendet
Nachbedingung	Nichts ändert sich
Ausgabe	Ein zusammengefasster Code wird in dem geschlossenen Jira Issue in den Kommentaren angezeigt
Regel	Sollte keine Codeänderungen commitet worden sein, so sollte auch keine Codezusammenfassung in den Kommentaren angezeigt werden.

3.4 Domänendatendiagramm

Es folgt ein Domänendatendiagramm, das den Zusammenhang der verwendeten Entitäten in Abbildung 3.1 erklärt. Zu sehen ist ein Zusammenhang zwischen den Commits, welche von Git hinzugefügt werden und den Jira Issues, also Aufgaben wie Tasks, WorkItems oder Features, als auch Entscheidungswissen.

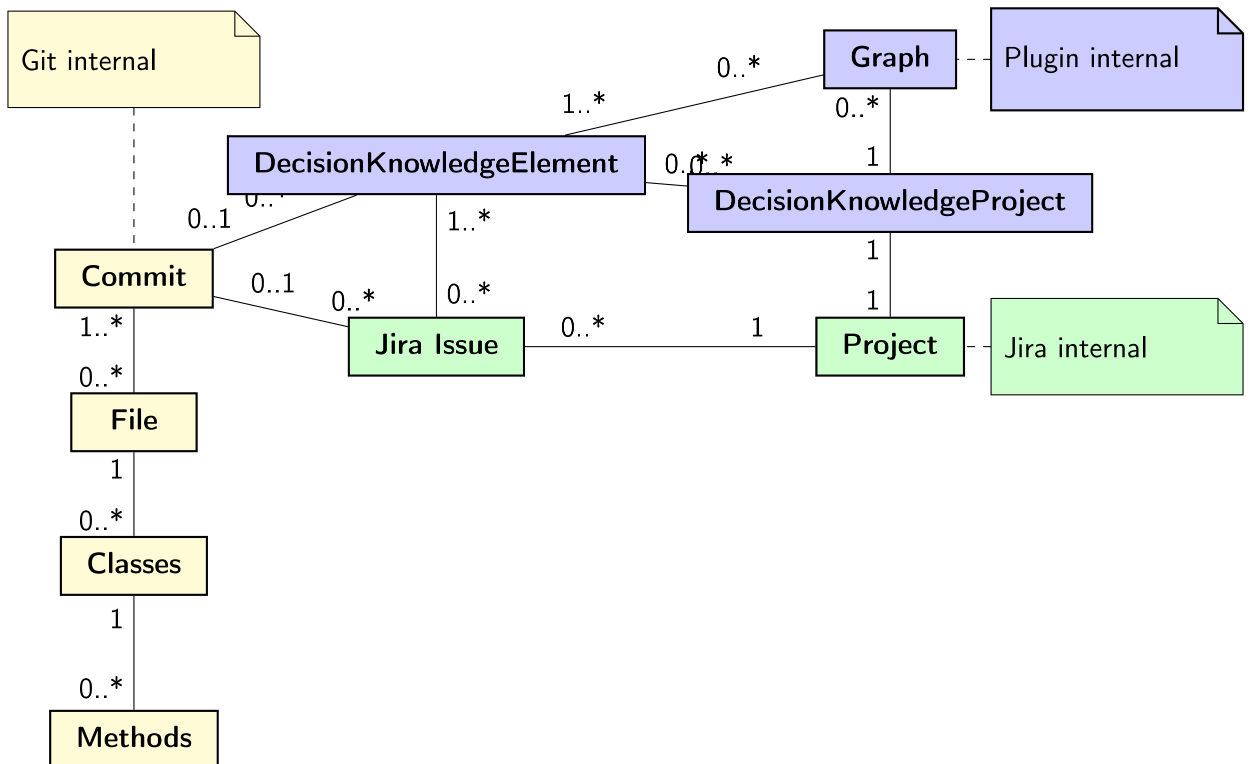


Abbildung 3.1: Domänendatendiagramm für die Anwendung des Prototypens

3.5 Nutzungsdiagramm

In Abbildung 3.2 ist das Nutzungsdiagramm für die Anwendung des Prototypens für das Plugin zu sehen. Hierbei hat die EntwicklerIn zunächst einmal die Git Extraktion zu konfigurieren, bei dem ein Klonen des verknüpften Git Repositorys ausgelöst wird. Die EntwicklerIn kann zudem auch die Codeänderungen implementieren, die beim Commit und Push eine Verknüpfung zwischen den Codeänderungen und des Jira Issues herstellt. Eine weitere Möglichkeit besteht, dass der Nutzer analysiert, ob die zu der Aufgabe verlinkten Jira Issues und ihre Codeänderungen konsistent sind, indem eine Anzeige mit einer Codezusammenfassung erzeugt wird. Sobald der Entwickler die Aufgabe beendet, wird ein Erzeugen der Codezusammenfassung in den Kommentaren ausgelöst, den er sich ansehen kann.

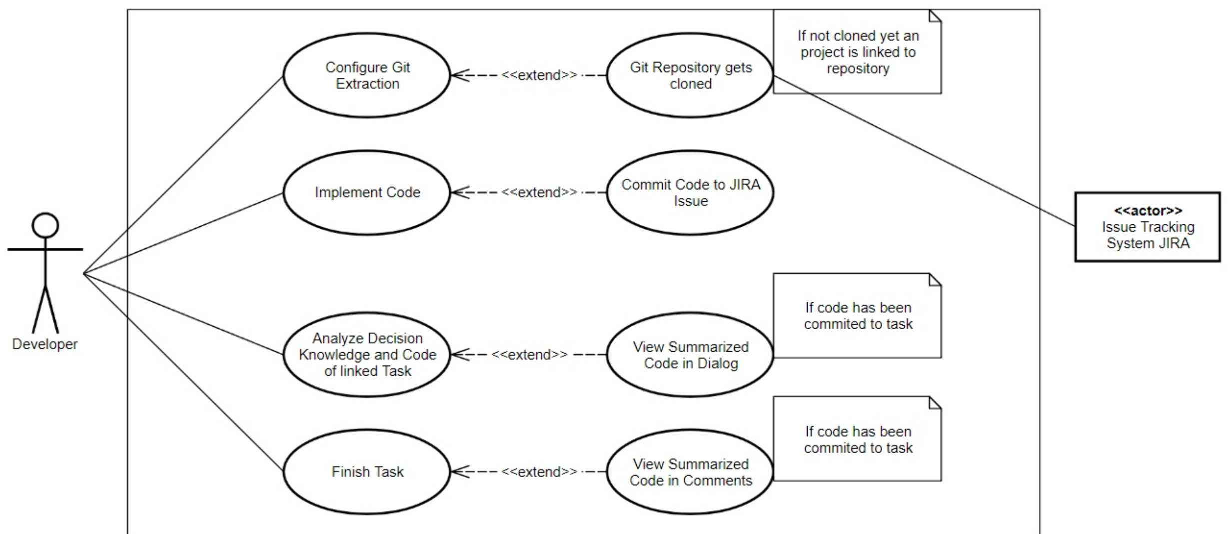


Abbildung 3.2: Nutzungsdiagramm für die Anwendung des Prototypens

3.6 Arbeitsbereiche

Das Arbeitsbereich-Strukturdiagramm in Abbildung 3.3 beschreibt alle für den Prototypen nötigen Arbeitsbereiche und deren Navigationsmöglichkeiten. Zudem enthält jeder Arbeitsbereich die zugehörigen Systemfunktionen und Daten.

Ansicht Workspace (WS) 1 wird von Jira bereitgestellt und beschreibt die Jira Oberfläche. Ansicht WS1.2 und Ansicht WS1.1 ermöglichen es, dass ProjektadministratorInnen das ConDec Plugin und einzelne Komponenten (de)aktivieren können. Zusätzlich stellt Ansicht WS1.2 weitere feinere Einstellungsmöglichkeiten für die vorhandenen Komponenten zur Verfügung. Beide Ansichten werden für die Systemfunktion SF3 benötigt.

Jira Issues werden in Ansicht WS1.4 dargestellt. Die NutzerIn kann hier Jira Issues erzeugen und verwalten. Ansicht WS1.4.1 erweitert Ansicht WS1.4 durch ein Modul und zeigt verlinktes Entscheidungswissen zum aktuellen Jira Issue in einer Graphenansicht. Von dort aus kann über ein Kontextmenü die Ansicht WS1.4.1.1 geöffnet und betrachtet werden. Hier wird die Systemfunktion SF2 erfüllt. In der weiteren Ansicht WS1.4.2 ist die Kommentarfunktion der Jira Issueansicht beschrieben, die für die Systemfunktion SF1 dient.

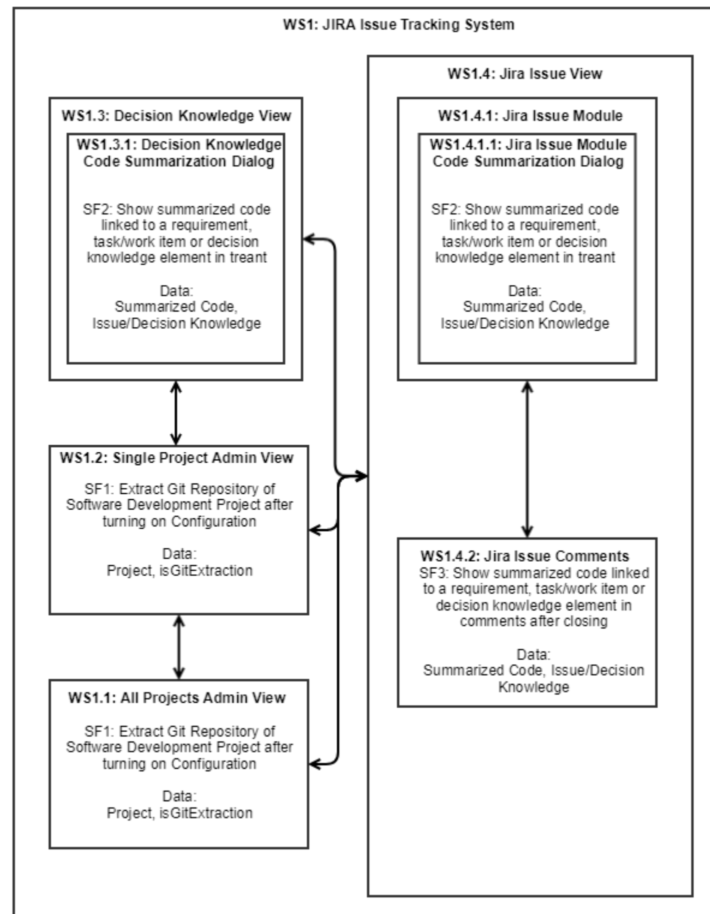


Abbildung 3.3: Arbeitsbereich für die Anwendung des Prototypens

3.7 Nichtfunktionale Anforderungen

Im Folgenden sind die nichtfunktionalen Anforderungen aufgelistet.

3.7.1 Funktionale Angemessenheit

Funktionale Angemessenheit setzt voraus, dass die funktionalen Anforderungen erfüllt wurden. Es wird mit der funktionaler Vollständigkeit und der funktionalen Korrektheit definiert. Dies wird mit abgeleiteten Testfälle überprüft, indem sie fehlerfrei durchlaufen. Eine genauere Beschreibung der Tests wird im Kapitel 6 Qualitätsicherung beschrieben.

3.7.2 Benutzbarkeit

Die Benutzbarkeit setzt voraus, dass durch eine leichte Bedienbarkeit des Plugins ein hoher Lernzuwachs der Benutzung der Funktionalität besteht. Zudem soll die Benutzeroberfläche ästhetisch sein. Im Kapitel 7 Evaluation werden anhand von ausgewählten Fragen NutzerInnen hinsichtlich dieser Anforderungen ausgefragt.

3.7.3 Wartbarkeit

Um eine generische ereignisbasierte Anzeige zu ermöglichen, soll die Wiederverwendbarkeit und die Veränderbarkeit des Codes ermöglicht werden. Dies wird mithilfe von statischen Tests überprüft. Genauer werden diese im Kapitel 6 Qualitätsicherung beschrieben.

4 Literaturrecherche

Um den in Kapitel 5 beschriebenen Entwurf des Prototypens zum Anzeigen der Zusammenfassung von Codeänderungen zu unterstützen, wird in diesem Kapitel eine Literaturrecherche zum Thema ereignisgesteuerte Anzeige durchgeführt. Dafür wird in Abschnitt 1 eine geeignete Forschungsfrage herausgearbeitet. Abschnitt 2 erläutert die Herangehensweise der Literaturrecherche. Relevante Literatur und ihre Kriterien werden in Abschnitt 3 aufgelistet und im Abschnitt 4 genauer beschrieben. Abschnitt 5 erläutert die Synthese gefundener Ergebnisse und wird dann in Abschnitt 6 als Erkenntnisse für die Arbeit zusammengefasst.

4.1 Forschungsfrage

Für die Anzeige von zusammengefassten Codeänderungen wie sie in Abschnitt 3.3 Funktionale Anforderungen durch Systemfunktion SF3 und Systemfunktion SF4 beschrieben wird, wird das Auslösen eines Ereignisses in Jira benötigt. Als Ereignis kann zum Beispiel das Ändern des Jira Issue Status. Ein Jira Issue Status beschreibt den momentanen Zustand einer Aufgabe im Arbeitsablauf ("workflow"). Ein Beispiel hierfür wäre das "Beenden" der Aufgabe.
Für die Implementierung dieser Anzeige ergibt sich die folgende Forschungsfrage: Welches Entwurfsmuster ist zur Implementierung einer generischen ereignisbasierten Anzeige geeignet?
Hierfür unterstützt die Literaturrecherche die Beantwortung dieser Forschungsfrage. Die Entwurfsmuster, die in den Artikeln vorgestellt werden, werden aus Sicht der Einsetzbarkeit des in Kapitel 3 beschriebenen Szenarios evaluiert. Weiterhin wird entschieden, welcher Ansatz für das Plugin am geeignetsten ist.

4.2 Methodik

Bei der Literaturrecherche werden die Datenbanken des IEEE[1] und des ACM[2] für die Suche geeigneter Literatur genutzt. Hierfür werden zwei Methoden angewandt. Dabei handelt es sich um die Suche ausgehend von Snowballing und die Suche ausgehend von Suchtermen. Dazu gibt es die zwei Unterscheidungen des "Forward"- und "Backward"-Snowballing. Mittels Forward-Snowballing werden jüngere Artikel gesucht, die den Aus-

[1] `https://ieeexplore.ieee.org/search/advsearch.jsp`, eingesehen am 31.01.2019
[2] `https://dl.acm.org/advsearch.cfm`, eingesehen am 31.01.2019

gangsartikel zitieren, während beim Backward-Snowballing die vom Ausgangsartikel zitierte Literatur untersucht wird [14].
Für die Literaturrecherche ausgehend von Suchtermen werden Begriffe im Zusammenhang mit der Forschungsfrage und den bis dato gefundenen Artikeln ausgewählt. Dabei handelt es sich um die folgenden Suchterme:

- event-based pattern
- observer pattern
- model view controller pattern
- PLoP event-driven
- observer event-based pattern
- observer pattern alternative
- event-based design pattern
- webhooks observer

Hierbei steht Pattern Language of Programs (PLoP) für eine jährlich stattfindende Konferenz für Entwurfsmuster.[3]
In dieser Arbeit wird hauptsächlich die Methodik der Suchtermen angewandt, da Snowballing zu Artikeln mit Anwendungsfällen für die gefundenen Muster führte und keine vergleichbaren oder andere Muster so zur Auswahl hätten stehen können.

4.3 Übersicht relevanter Literatur

Im Folgenden werden zwei Tabellen angezeigt. Bei Tabelle 4.1 werden die Ergebnisse der Suche mit Suchtermen vorgestellt und Tabelle 4.2 werden die Ergebnisse der Suche mit Snowballing angezeigt. Es werden hierbei die entsprechenden Suchterme beziehungsweise die Ausgangsartikel, das Datum des Suchvorgangs, Anzahl der Ergebnisse und relevanter Ergebnisse mit aufgezeigt. Die Relevanz der Ergebnisse wird überprüft, indem die Schlüsselwörter und der Titel des Artikels betrachtet wurden. Sobald diese mindestens einen weiteren Begriff beinhalteten, die entweder aus der Liste der Suchbegriffe stammt oder folgende sind

- design patterns,
- event based,
- ein bestimmtes Entwurfsmuster, wie observer pattern

[3] `https://www.hillside.net/`, eingesehen am 17.09.2018

wurde die Zusammenfassung (Abstract) des ausgewählten Artikels, den sogenannten "Abstract", durchgelesen und beurteilt. Wenn die oben angegebenen Suchterme oder ein Anwendungsbeispiel ähnlich dieser Arbeit genannt wurde, so wurde der Artikel in die Übersicht relevanter Literatur mit hinzugefügt. Nach betrachten der ersten 40 bis 50 Artikel, dies entspricht den ersten beiden Suchergebnisseiten, wurden zwischen dem 30. und 50. Ergebnis keine relevanten Artikel mehr gefunden und somit die weiteren Suchererergebnisse nicht mehr betrachtet. Dies führt auch dazu, dass einige Suchterme keine relevanten Ergebnisse ergaben.

Tabelle 4.1: Ergebnisse der Suche mit Suchtermen

Datenbank	Datum	Suchterm	Anzahl Ergebnisse	Anzahl relevanter Ergebnisse	relevante Ergebnisse
IEEE	15.08.2018	event-based pattern	331	1	Grushka et al. - 2010 - „A Design Pattern for Event-Based Processing of Security-Enriched SOAP Messages" [5]
IEEE	15.08.2018	model view controller pattern	181	2	Churi et al. - 2016 - „Model-View-Controller Pattern in BI Dashboards: Designing Best Practices" [3], Syromiatnikov et al. - 2014 - „A Journey through the Land of Model-View-Design Patterns" [17]
ACM	15.08.2018	event-based design	300.341	0	
ACM	27.08.2018	observer pattern	46.700	0	
IEEE	27.08.2018	observer pattern	1.179	1	Liu et al. - 2010 - „A Novel Implementation of Observer Pattern by Aspect Based on Java Annotation" [12]

ACM	03.09.2018	PLoP event-driven	531.105	0	
ACM	03.09.2018	PLoP event-based	190.250	0	
ACM	03.09.2018	PLoP event	20.148	0	
ACM	05.09.2018	observer pattern alternative	62.827	2	Springer et al. - 2016 - „Classes as Layers: Rewriting Design Patterns with COP - Alternative Implementations of Decorator, Observer, and Visitor“[16], Ampatzoglou et al. - 2013 - „Design Pattern Alternatives: What to do when a GOF pattern fails“ [1]
ACM	05.09.2018	+event-based +design +pattern	20.148	1	Paschke et al. - 2012 - „Tutorial on Advanced Design Patterns in Event Processing“[15]
IEEE	19.09.2018	webhooks observer	0	0	
IEEE	19.09.2018	webhook observer	0	0	
IEEE	19.09.2018	“Document Title”:webhooks observer OR “Author Keywords”: webhooks observer OR “Abstract”: webhooks observer	0	0	
ACM	19.09.2018	webhooks observer	23524	0	

Tabelle 4.2: Ergebnisse der Suche mit Snowballing

Ausgangs-Literatur	Datum	Snowballing-Art	Anzahl Zitierungen	Anzahl relevanter Ergebnisse	relevante Ergebnisse
Syromiatnikov und Weyns[17]	27.08.2018	Backward	21	2	Burbeck - 1987 - „Applications Programming in Smalltalk-80 (TM): How to use Model-View-Controller (MVC)“[2], Krasner et al. - 1988 - „A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80“[10]
Ampatzoglou, Charalampidou und Stamelos[1]	05.09.2018	Backward	34	1	Iaría et al. - 2001 - „Refining the Observer Pattern: The Middle Observer Pattern“[7]

4.4 Ergebnisse

Während der Literaturrecherche wurden zehn Artikel gefunden, die ein mögliches Entwurfsmuster für das Plugin beschreiben.
In Tabelle 4.3 ist eine Übersicht der Ergebnisse aller Artikel einzusehen. Dabei wird neben Titel, AutorInnen und Jahr der Erscheinung zudem noch Kontext und Motivation und die Art des Entwurfsmusters beschrieben. Zudem werden in den folgenden Abschnitten nocht die Anforderungen an das Muster, die Umsetzung, der Synchronisationsmechnanismus und noch eventuelle zusätzliche Informationen aufgezeigt.

Tabelle 4.3: Ergebnisse der Literatur

Titel	AutorInnen	Jahr	Kontext und Motivation	Entwurfsmuster
„A Design Pattern for Event-Based Processing of Security-Enriched SOAP Messages“	Grushka, Jensen und Lo Iacono	2010	Ereignis-gesteuerte XML Verarbeitung mit aufgeteilten Unteraufgaben	Event Pipeline Pattern
„Model-View-Controller Pattern in BI Dashboards: Designing Best Practices“	Churi, Wagh, Kalelkar u. a.	2016	Nutzung von Model-View-Controller im Business Intelligence mit Entscheidungen	Model-View-Controller
„A Journey through the Land of Model-View-Design Patterns“	Syromiatnikov und Weyns	2014	Vergleich aller Model-View-* Entwurfsmuster	• Model-View-Controller • Model-View-View Model • Model-View-Presenter
„A Novel Implementation of Observer Pattern by Aspect Based on Java Annotation“	Liu, Yin und Wang	2010	Implementierung und Erklärung von Observer Pattern	Observer Pattern

„Classes as Layers: Rewriting Design Patterns with COP - Alternative Implementations of Decorator, Observer, and Visitor“	Springer, Masuhara und Hirschfeld	2016	Implementierung und Erklärung von Observer Pattern und anderen nicht ereignisbasierten Entwurfsmustern	Observer Pattern
„Tutorial on Advanced Design Patterns in Event Processing“	Paschke, Vincent, Alves u. a.	2012	Ereignisverarbeitungen und dessen Entwurfsmuster	Event Processing Pattern
„Applications Programming in Smalltalk-80 (TM): How to use Model-View-Controller (MVC)“	Burbeck	1987	Aufbau und Verwendung von Model-View-Controller	Model-View-Controller
„A Cookbook for Using the Model- View-Controller User Interface Paradigm in Smalltalk-80“	Krasner und Pope	1988	Aufbau und Verwendung von Model-View-Controller	Model-View-Controller
„Refining the Observer Pattern: The Middle Observer Pattern“	Iaría und Chesini	2001	Veränderung des Observer Patterns, um Datenredundanzen zu vermeiden	MiddleObserver Pattern

Hierbei ergaben sich folgende Entwurfsmuster als Ergebnis:

- Ereignis-Leitungsmuster (Event Pipeline Pattern)
- Modell-Präsentation-Steuerung (Model-View-Controller)
- Modell-Präsentation-Präsentationsmodell (Model-View-View Model)

- Modell-Ansicht-Präsentierer (Model-View-Presenter)
- Beobachter-Muster (Observer Pattern)
- MittelBeobachter-Muster (MiddleObserver Pattern)
- Ereignisverarbeitungs Muster (Event Processing Pattern)

4.4.1 Event Pipeline Pattern

Anforderungen an das Entwurfsmuster: Eine Art Leitungsverarbeitung wird dazu verwendet, um Ereignisse von Modul zu Modul weiter zu kommunizieren. Die Aufteilung der Verarbeitung soll die Wartbarkeit erhöhen.

Umsetzung der Anforderungen: Für XML Verarbeitungsanweisung werden die Inhalt der XML Anweisungen in mehreren Modulen aufgeteilt, bei dem jeder eine Unteraufgabe ausführt. [5]. Wie in Abbildung 4.1 zu sehen, ist die
`AbstracEventHandler` Klasse als Basis zu betrachten. Diese abstrakte Klasse wird von den konkreten Klassen (Handler), die für einzelne Zwischenschritte auf der Leitung zuständig sind, implementiert. Vereinfacht heißt das, dass ein vorhandenes Ereignis zum nächsten Handler weitergereicht wird.

Synchronisationsmechanismus: Keine Informationen diesbezüglich.

Weitere Informationen: In diesem Modell kann jedes Modul Ereignisse erstellen und weitergeben um die Parameter eines vorhandenen Ereignisses zu ändern oder um es zu absorbieren.

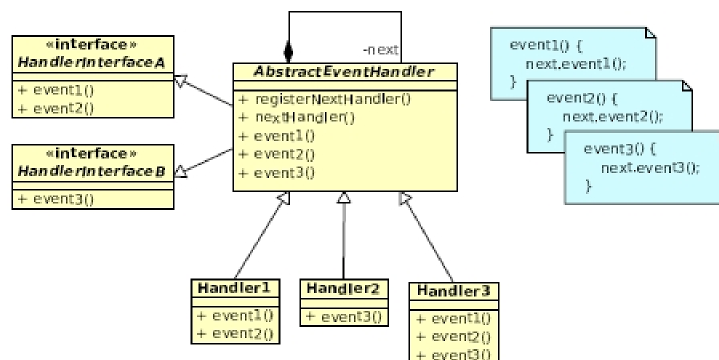


Abbildung 4.1: Event Pipeline Pattern [5]

4.4.2 Model-View-Controller

Anforderungen an das Entwurfsmuster: Die selbe Information wird unterschiedlich in verschiedenen Formaten angezeigt. Das heißt, dass die Informationen sich nicht ändern, die Anzeige hingegen schon. Außerdem müssen die Anzeige und das Verhalten Datenänderungen sofort reflektieren können.

Umsetzung der Anforderungen: Burbeck [2] beschreibt, dass der Model-View-Controller die Nutzereingabe, die Verhaltenskontrolle und die visuelle Rückmeldung als drei verschiedene Objekttypen mit spezialisierten Aufgaben implementiert sind. In Abbildung 4.2 werden die drei Objekttypen aufgezeigt. Das "Model" ist laut Syromiatnikov et al. [17] verantwortlich für die Domänendaten und die Logik. Dieses Objekt referenziert nicht auf die anderen Objekte dieses Modells. Bei der "View" Komponente werden die Daten des "Model" visualisiert. Der "Controller" verarbeitet die Informationen der Nutzereingabe und gibt sie an das "Model" weiter. View und Controller interagieren somit mit dem Nutzer.

Synchronisationsmechanismus: Der allgemeine Interaktionszyklus für den Model-View-Controller beinhaltet laut Krasner et al. [10] eine Aktionseingabe, bei dem der Controller dem Model die Meldung zur Zustandsänderung weitergibt. Das Model wiederum ändert, falls notwendig, seinen Zustand. Da View und Controller das Model dauerhaft beobachten, was in Abbildung 4.2 als "Observer" bezeichnet wird, werden die Anzeige und Interaktionsmethode geändert, falls dies notwendig ist.

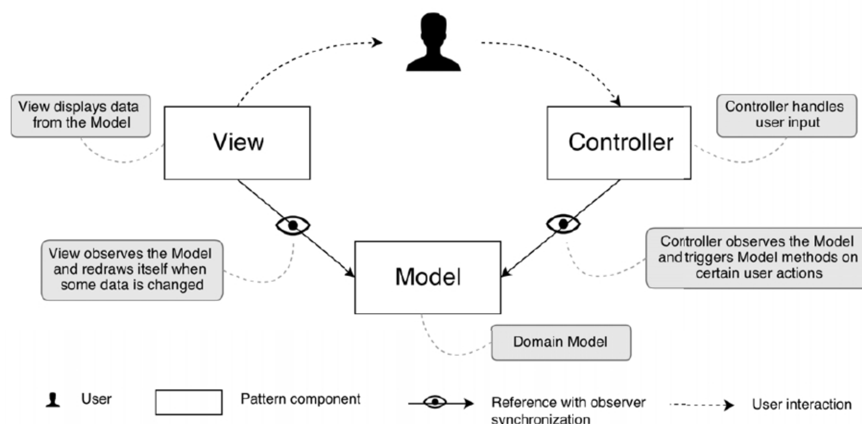


Abbildung 4.2: Model-View-Controller [17]

4.4.3 Model-View-View Model

Anforderungen an das Entwurfsmuster: Die Anzeigen müssen auch hier sofortige Änderungen der Daten anzeigen. Zusätzlich müssen Änderungen der Anzeigen, die auch direkt in der Anzeige erfolgt, verarbeitet und wieder angezeigt werden. So kann zum

Beispiel die Farbe der Anzeige geändert werden.

Umsetzung der Anforderungen: Anzeigenänderungen können im View Model gespeichert werden, welches in Abbildung 4.3 als "Application Model" angezeigt wird. Im Gegensatz zum Model-View-Controller interagieren View und Controller nicht direkt mit Model, sondern kommunizieren mit dem "Application Model", der den View Zustand und die Nutzereingabe an das Model weitergibt. Somit können neben Model-Daten auch visuelle Veränderungen angepasst werden.

Synchronisationsmechanismus: Veränderungen der Informationen zu Daten und Anzeige werden auch hier von Controller und View beobachtet. Es handelt sich wieder um die Observer Synchronisation.

Weitere Informationen: Das Model-View-View Model löst laut Syromiatnikov et al. [17] das Problem vom Model-View-Controller, dass View Zustände, die nicht zum Domänenmodel gehören und somit nicht zum Model gehören.

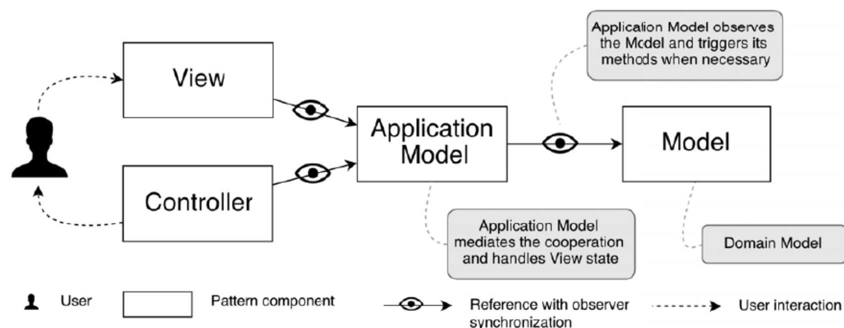


Abbildung 4.3: Model-View-View Model [17]

4.4.4 Model-View-Presenter

Anforderungen an das Entwurfsmuster: Wie beim Model-View-View Model, müssen auch hier sofortige Änderungen der Daten und der Anzeige angezeigt werden. Der Unterschied zu den bekannten Entwurfsmustern ist der Presenter, der dazu dient, Änderungen in der View direkt und ohne Umwege über das Model aufzurufen.

Umsetzung der Anforderungen: Für den Model-View-Presenter gibt es laut Syromiatnikov und Weyns mehrere Modelle. Betrachtet werden in diesem Artikel drei Modelle mit zwei unterschiedlichen Synchronisationsmethoden. In Abbildung 4.4 wird beispielsweise der sogenannte "Dolphin Smalltalk Model-View-Presenter" aufgezeigt. Hierbei hat der Presenter die Aufgabe, die Anwendung synchronisiert zu halten. Er beinhaltet die größte Logik, indem dort Nutzereingaben verarbeitet werden, Domänenmethoden aufgerufen, das Model konstant in Kommunikation gehalten wird und die View falls notwendig aktualisiert wird. In einem vergleichbaren Model-View-Presenter, dem "Supervising

Presenter", wird das Model nicht auf die Domänendaten eingeschränkt. Es kann auch Zustände für die Anzeige speichern und von der View beobachtet werden. Die View ist für die Anzeige und einfachen Nutzereingaben zuständig.
Beim "Passive View Pattern" hat die View eine leichte Implementierung mit einfachen Zugriffsfunktionen und Logik zur Ereignisverarbeitung. Das Model beinhaltet die Domänendaten und die Logik, während der Presenter die Nutzereingaben nimmt, sie in die Logik des Models einfügt und zur Aktualisierung in der View führt. Somit beobachtet keiner der Komponenten die anderen, sondern alle werden direkt vom Presenter gesteuert. Dies ist wiederrum eine flow synchronization.

Synchronisationsmechanismus: Im Model des "Dolphin Smalltalk Model-View-Presenter" beobachtet somit die View das Model mithilfe der Observer Synchronisation, um dessen Zustand anzuzeigen. Beim "Supervising Presenter" wird auch der Observer genutzt, damit der Presenter in diesem Model die View beobachtet und somit für die Kommunikation zwischen View und Model sorgt. Beim dritten Model, wird die Synchronisation nicht durch das Beobachten von Komponenten ermöglicht, sondern mithilfe eines Flusses. Die sogenannte "flow synchronization" beobachtet nicht im Gegensatz zum bisher schon genannten "observer synchronization". Es basiert auf einer sequentiellen Befehlsausführung. Anstelle des Beobachtens von Klassen, werden die direkten Methoden zwischen Schnittstellen und Domänen aufgerufen, wie in Abbildung 4.5 zu sehen ist.

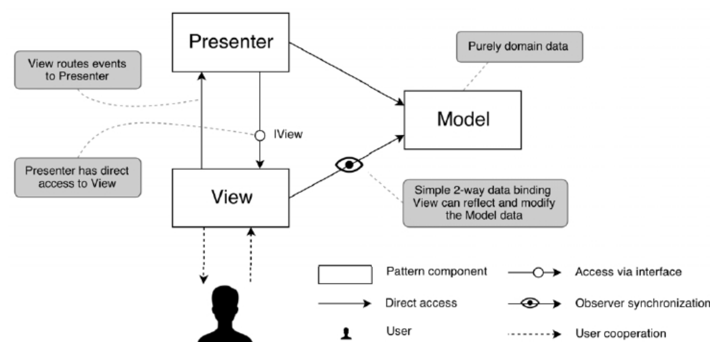


Abbildung 4.4: Dolphin Smalltalk Model-View-Presenter [17]

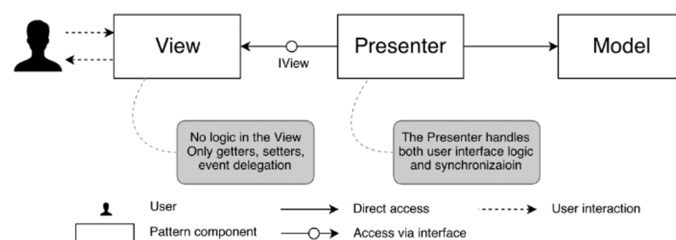


Abbildung 4.5: Passive View Pattern [17]

4.4.5 Observer Pattern

Anforderungen an das Entwurfsmuster: Das Observer Entwurfsmuster wird verwendet, um Veränderungen eines Subjekts an einen Beobachter weiter zu geben [16]. Ein Beobachter soll benachichtigt werden, wenn sich ein Subjekt, das er beobachtet ändert. [12].

Umsetzung der Anforderungen: Wie in Abbildung 4.6 zu sehen, wird der Observer aus vier Objekten gebaut. Das "Subject" kennt seine Beobachter, den "Observer". Der Observer aktualisiert die Schnittstelle für die Objekte, welche über Veränderungen im Subject benachrichtigt werden sollten. Das "ConcreteSubject" erbt die Methoden vom Subject. Es speichert den Zustand, der für den "ConcreteObserver" bedeutend ist und sendet Benachrichtigungen an seine Observer bei Zustandsänderungen. Der ConcreteObserver erbt vom Observer und speichert die Zustände, die im referenzierten ConcreteSubject konsistent bleiben sollten.
Iaría et al. [7] hat das Observer Entwurfsmuster erweitert und diese "Middle Observer pattern" genannt. Die genannten Bedingungen für die Nutzung diesen Patterns wären:

- Es sind mehrere Subjekte mit redundanten Informationen enthalten.
- Eine Konsistenz über den Observern mit Informationen, welche nicht zu dem Subjekt gehören, muss gehalten werden.
- Informationen oder Verhalten, die auf das Subjekt hinweisen, aber nicht vom Subjekt oder den Observern gehalten werden.

Beim MiddleObserver Pattern wird, wie in Abbildung 4.7 gezeigt, ein MiddleObserver eingefügt, der die Informationen behält, die weder zum Subjekt, noch zu den Observern gehören.

Synchronisationsmechanismus: Da es sich hierbei direkt um den Observer Pattern handelt, ist auch die Synchronisation die Observer Synchronisation.

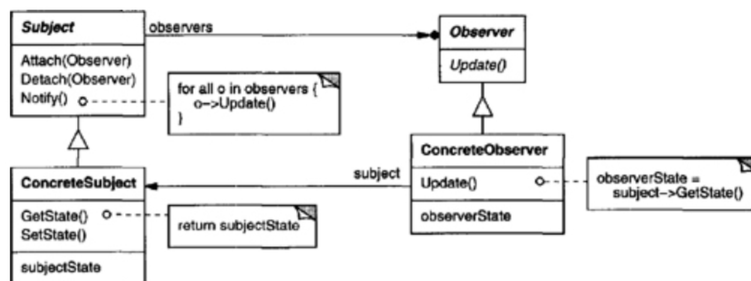


Abbildung 4.6: Observer Pattern [4]

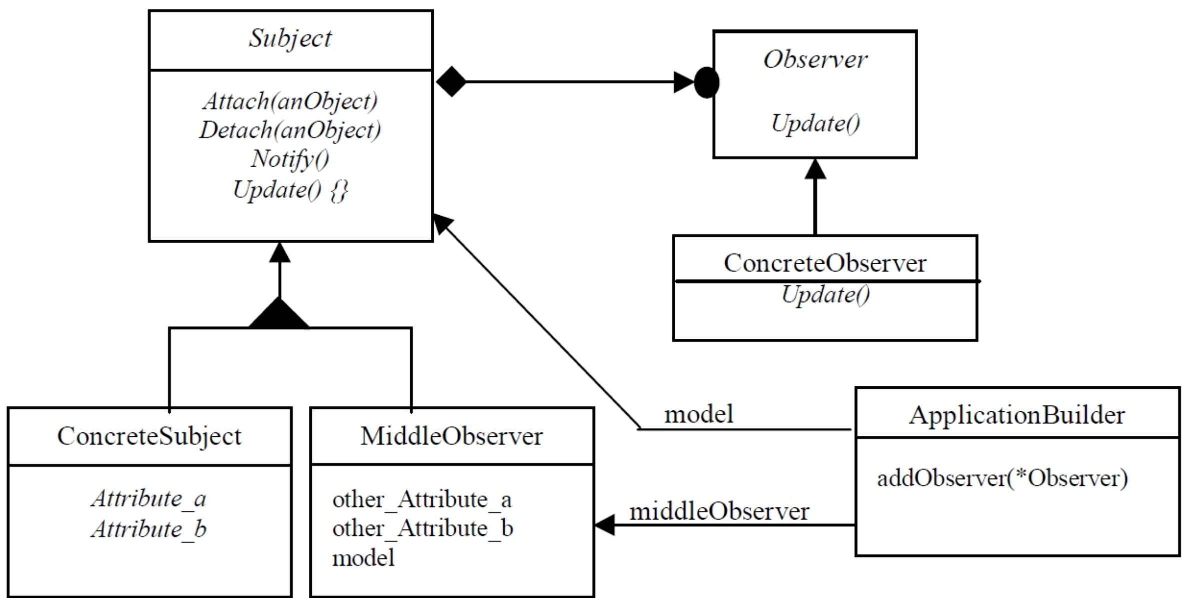


Abbildung 4.7: MiddleObserver Pattern [7]

4.4.6 Event Processing Pattern

Anforderungen an das Entwurfsmuster: Paschke et al. [15] beschreibt Entwurfsmuster zur Ereignisverarbeitung. Dabei werden die Ereignisse so verarbeitet, dass neue Ereignisse herausgegeben werden oder ein Filter mit ausgewählten Kriterien wählt ein geeignetes Ereignis und gibt es wieder aus.

Umsetzung der Anforderungen: Für die Umsetzung gibt es mehrere Möglichkeiten.

- das Identifikationsmuster (Identification Pattern), das Ereignis wird identifiziert nach vorhergegangen Ereignissen und Ereignisart.
- das Auswahlmuster (Selection Pattern), Ereignisse werden selektiert und zur weiteren Analyse gebracht.
- das Filtermuster (Filter Pattern), ein oder mehrere Ereignisse werden herausgefiltert und weiterverarbeitet.
- das Bereicherungsmuster (Enrichment Pattern), Ereignisse werden mit Informationen von vorhergegangen Ereignissen oder Daten "bereichert"
- das Überwachungsmuster (Monitoring Pattern), einige Ereignisse werden überwacht, um für weitere Abläufe selektiert zu werden.
- das Konsolidierungsmuster (Consolidation Pattern), mehrere Ereignisse werden zu einem "main" oder "primary" Ereignis zusammengefügt.
- die Zusammensetzung (Composition), Ereignisse werden mithilfe von vergangenen Informationen zusammengefügt zu einem Ereignis.
- das Anhäufungsmuster (Aggregation Pattern), zusammenfügen von Ereignisse, um ein Ereignis mit neuer Information zu allen Ereignissen gemeinsam zu erstellen.
- das Bewertungsmuster (Assessment Pattern), ein Ereignis welches bewertet wird und die Ergebnisse der Bewertung herausgibt.
- das Leitungsmuster (Routing Pattern), Ereignis wird durch eine Leitung geführt und soll als Ereignis erscheinen.

In Abbildung 4.8 wird hierfür beispielhaft das Identifikationsmuster aufgezeigt.

Synchronisationsmechanismus: Keine Informationen diesbezüglich.

Abbildung 4.8: Identification Pattern [15]

4.5 Synthese

In der Synthese werden die gefundenen Ansätze auf Hinsicht der Nützlichkeit für das Plugin beurteilt. In Tabelle 4.4 sieht man eine Zusammenfassung, in der neben den Artikeln für die Ansätze das entsprechende Entwurfsmuster, die Anforderungen, die Umsetzung und die Synchronisationsart der Ansätze zu finden sind, um die Unterschiede genauer zu betrachten. Benötigt wird eine generische ereignisbasierte Anzeige, die ausgelöst wird, wenn sich das Datenmodell ändert. In dieser Arbeit werden Ereignisse in einer Aufgabe (work item, feature task) betrachtet. Die Änderung des Datenmodells ist dabei durch eine Status-Änderung der Aufgaben gekennzeichnet. Die Anzeige soll eine Zusammenfassung der verknüpften Codeänderungen zu der beendeten Aufgabe beinhalten. Dies wird im Abschnitt 3.3 Funktionale Anforderungen erläutert und wird somit für die Systemfunktionen "SF3: Zeige zusammengefassten Code von verlinkten Anforderungen, Aufgaben oder Entscheidungswissen im Entscheidungsbaum" und "SF4: Zeige zusammengefassten Code von Anforderungen, Aufgaben oder Entscheidungswissen beim Beenden einer Aufgabe" benötigt. Eine Verarbeitung des Ereignisses in mehreren Modulen wäre für diese einmalige Anzeige somit nicht nötig. Dies bedeutet, dass die Event Pipeline von Gruschka et al. [5] in diesem Sinne zu komplex ist. Zudem entspricht das Event Processing Pattern von Paschke et al. [15] nicht den Vorgaben. Es handelt sich dabei um mehrere Entwurfsmuster, die Ereignisse analysieren und entweder eines auswählen, aufgrund von vorhandenen Kriterien, oder das Ergebnis der Analyse herausgeben. Eine mögliche Anzeige von anderem Inhalt wäre somit nicht möglich.
Betrachten wir nun die Observer bzw. MidObserver Pattern, so sehen wir, dass ein Subjekt ein oder mehrere Beobachter (Observer) besitzt. Diese führen dann bei Änderungen Methoden aus. Der MidObserver ist nur eine Erweiterung des Observer, um Redundanzen zu verringern. Zusammengefasst sind die Observer Pattern eventuell zu primitiv, da neben dem Beobachtern, in den Observern auch die Anzeige und die Logik enthalten sollte. Es wäre eine Option eine Klasse aus Observern, Anzeige und Logik zu implementieren, jedoch würde aus Sicht der Wiederverwendbarkeit und der Lesbarkeit dies nicht von Vorteil sein, da die Klassen eine zu große Komplexität und zu viele Methoden enthalten werden.
Somit bleibt für den Entwurf der generischen ereignisbasierten Anzeige ein Model-View-* Muster. Die Vorteile liegen vorallem darin, dass eine Aufteilung in Anzeige, Nutzereingabe und Logik vorhanden ist. Die Wartbarkeit ist durch die mehreren Klassen somit gewährt. Da nun Model-View-View Model und Model-View-Presenter Lösungen dafür sind, dass nicht Domänendaten, also anzeigebasierte Daten, auch geändert werden dür-

fen, und dies jedoch keine Anforderung, beziehungsweise keinen Nutzen in Jira haben würde, wird der Model-View-Controller für den Entwurf und die Entwicklung der ereignisbasierten Anzeige im Plugin genutzt. Außerdem enthält der Model-View-Controller Observer, welche zusätzlich die Logik und die Anzeige verwalten. Somit wird durch eine Änderung der Domänendaten eine Änderung in der Anzeige und der Nutzereingabe verändert, falls notwendig. Zusätzliche Aufrufe von Methoden sind somit redundant und werden weggelassen.

Tabelle 4.4: Synthese der Literatur

Ansatz	Entwurfsmuster	Anforderungen	Umsetzung	Synchronisation
[5]	Event Pipeline Pattern	Ereignisse müssen von Modul zu Modul über eine Leitungsverarbeitung transportiert werden.	AbstractEventHandler zur Weitergabe von Ereignisse an andere Handler; Mehrere Handler mit den ankommenenden Ereignissen; Mehrere Handler Interfaces zur Verarbeitung von ankommenenden Ereignissen	Keine Informationen
[3], [17], [2], [10]	Model-View-Controller	Informationen sollen bei Datenänderungen in der Anzeige sofort reflektiert werden. Mehrere Anzeigen für dieselben Daten soll möglich sein.	Model enthält Domändaten und Logik; View ist die Anzeige für das Model; Controller verarbeiten Informationen der Nutzereingabe	Observer Synchronisation
[17]	Model-View-View Model	Informationen sollen bei Datenänderungen und Anzeigenänderungen in der Anzeige sofort reflektiert werden.	Model enthält Domändaten und Logik; View ist die Anzeige für das Model; Controller verarbeiten Informationen der Nutzereingabe; Application Model enthält den Anzeigezustand	Observer Synchronisation

[17]	Model-View-Presenter	Informationen sollen bei Datenänderungen und Anzeigenänderungen in der Anzeige sofort reflektiert werden. Anzeigenänderungen soll unabhängig vom Model agieren.	Model enthält Domändaten; View ist die Anzeige für das Model; Presenter verarbeiten Informationen der Nutzereingabe und aktualisiert View	Observer Synchronisation
[17]	Model-View-Presenter	Informationen sollen bei Datenänderungen und Anzeigenänderungen in der Anzeige sofort reflektiert werden.	Model enthält Domändaten; View ist die Anzeige für das Model; Presenter verarbeiten Informationen der Nutzereingabe und aktualisiert View	Flow Synchronisation
[12], [16]	Observer Pattern	Veränderungen eines Subjekts werden an Beobachter weitergegeben, damit dieser es verarbeiten kann.	Subject kennt seine Observer; Observer aktualisiert Schnittstelle für Objekte, die vom Subject benachrichtigt werden sollen; ConcreteSubject sendet Benachrichtigungen an ConcreteObserver; ConcreteObserver speichert Zustände, die im referenzierten ConcreteSubject konsistent bleiben sollen	Observer Synchronisation

[7]	MidObserver Pattern	Erweiterung von Observer um Informationen zu speichern, die weder zum Subjekt noch zu Observer gehören	Subject kennt seine Observer; Observer aktualisiert Schnittstelle für Objekte, die vom Subject benachrichtigt werden sollen; ConcreteSubject sendet Benachrichtigungen an ConcreteObserver; ConcreteObserver speichert Zustände, die im referenzierten ConcreteSubject konsistent bleiben sollen; MidObserver enthält Informationen, die weder zum Subject noch zum Observer gehören	Observer Synchronisation
[15]	Event Processing Pattern	Ereignis wird verarbeitet und Ereignis oder Analyse von Ereignis wird herausgegeben	Eingabe Event; Verarbeitung, welches an das entsprechende Muster angepasst wird; Ausgabe Event	Keine Informationen

4.6 Erkenntnisse für die Arbeit

Eine generische ereignisbasierte Anzeige für das Plugin kann auf verschiedene Weisen realisiert werden. Jedoch betrachtet man die Wartbarkeit, die Lesbarkeit des Codes und die Komplexität, so ist es am sinnvollsten, mehrere Klassen zu nutzen. Doch sollte ein einfacher Prozess, welcher nur eine Anzeige aufzeigt, nicht in mehreren Modulen aufgeteilt werden, da die Weiterreichung von Ereignissen keine Rolle spielt. Somit sind unter anderem das Event Pipeline Pattern und das Event Processing Pattern nicht vorteilhaft für die Nutzung. Daher wird für den Entwurf und die anschließende Implementierung der Anzeige der Ansatz des Model-View-Controller in dieser Bachelorarbeit herangezogen.

5 Entwurf und Implementierung

Dieses Kapitel beschreibt den Entwurf und die Implementierung für den Prototypen, damit die an ihn gestellten Anforderungen erfüllt werden. Dabei wird im ersten Abschnitt die Umsetzung erster Grobanforderungen diskutiert. Anschließend wird die Umsetzung der funktionalen Anforderungen betrachtet. Im dritten Abschnitt werden nichtfunktionale Anforderungen addressiert. In den letzten beiden Abschnitten werden die Ergebnisse anhand des UML-Klassendiagramms und einer genauen Beschreibung aller implementierten Ansichten aufgezeigt.

5.1 Umsetzung erster Grobanforderungen

Bei der in Kapitel 3.1 Grobanforderungen ersten Grobanforderungen entstanden die Systemfunktion zur Entwicklung eines Dialogs nach dem Starten und Beenden der Jira Aufgabe. Hierbei entstand folgende Entscheidung:

Entscheidung 1: Das Auslösen des Ereignisses soll vom Backend beobachtet werden
Begründung: Es gab die Möglichkeit die Ereignisse "Starten" und "Beenden" entweder vom Backend im Java-Code oder vom Frontend im JavaScript-Code beobachten zu lassen. Die Entscheidung lief auf das Backend aus, da die GUI im Frontend nur einzelne Elemente beobachten kann. Das heißt, würde man im Frontend den Beobachter setzen, so müsste man den auf alle Knöpfe und Funktionalitäten setzen, die die Ereignisse auslösen. Im Backend hingegen kann eine Änderung des Status jeder Aufgabe beobachtet werden. Somit muss nur noch das "Starten" und "Beenden" herausgefiltert werden.

Mit dieser Entscheidung wurde folgender Entwurf entwickelt. Dabei sind die Farben unterteilt in grün für das Model, blau für die View und Gelb für den Controller.

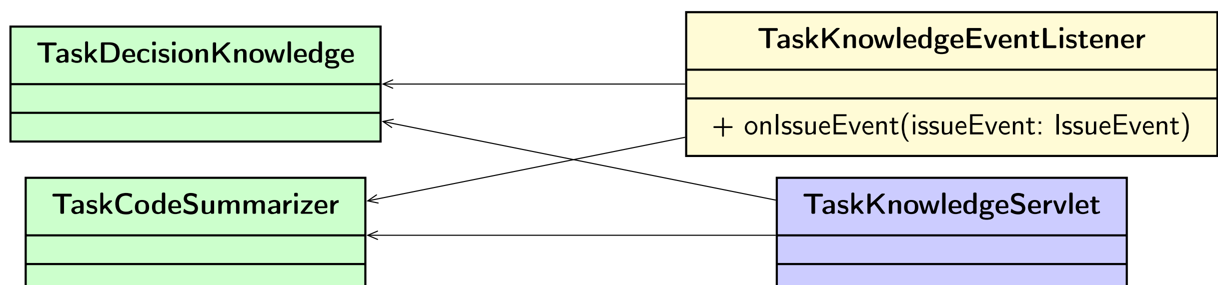


Abbildung 5.1: Entwurfsklassendiagramm der alten Grobanforderungen

Wie in Kapitel 4 Literaturrecherche beschrieben, sollte der Model-View-Controller verwendet werden. Zu sehen ist ein EventListener als Controller und ein Servlet als View. Zudem würden als Model die Klassen *TaskDecisionKnowledge* und *TaskCodeSummarizer* dienen.
Jedoch musste die Entwicklung dieses Entwurfs, aufgrund von Bedingungen in Jira, beenden. Ein Servlet kann in Jira kein Dialog aufrufen, ohne zuvor einen Knopf gedrückt zu haben, der vom GUI beobachtet wurde. Das heißt, es hätte ein zusätzlicher Knopf mit der schon vorhandenen Funktionalität zur Änderung des Aufgabenstatus eingefügt werden müssen. Es wurde dann folgende Entscheidung getroffen:

Entscheidung 2: Es wird nur eine Zusammenfassung von Codeänderungen nach dem Schließen der Aufgabe in den Kommentaren angezeigt
Begründung: Da bereits im Entscheidungsbaum alles verlinkte Entscheidungswissen zu sehen sind, kam es zu der Entscheidung, dass ein Dialog mit den verlinkten Entscheidungswissen zu Redundanz führen würde. Um dann keinen weiteren Knopf mit einer schon vorhandenen Funktionalität einzufügen, wurde entschieden, den zusammengefassten Codeänderungen in die Kommentare einzufügen. Beim Starten ist in dem Fall noch keine Codeänderungen vorhanden, daher wird nur beim Beenden der Aufgabe der zusammengefasste Codeänderungen in die Kommentare eingefügt.
Für die Übersicht aller zusammengefassten Codeänderungen, wird mithilfe des Kontextmenus im Entscheidungsbaum ein Dialog des entsprechend ausgewählten Jira Issues angezeigt.

Aufgrund dieser Entscheidungen, wurde wie im Kapitel 3.1 Grobanforderungen beschrieben, die Grobanforderungen verändert und neue Systemfunktionen eingeführt.

5.2 Umsetzung funktionaler Anforderungen

Dieses Kapitel beschreibt die Entscheidungsprobleme und ihre Entwurfsentscheidungen aller in Kapitel Abschnitt 3.3 Funktionale Anforderungen beschriebenen Systemfunktionen. Zudem werden die Entwurfsklassendiagramme für die anschließende Implementierung aufgezeichnet.

5.2.1 Systemfunktion 1

Entwurfsentscheidungen

In dieser Arbeit wird davon ausgegangen, dass Jira Projekte mithilfe von dem Plugin "Git Integration for Jira" mit Git Repositorys verlinkt sind. Werden in den Commitnachrichten die Issue-ID mit angegeben, so entsteht auch eine Verlinkung zwischen Jira Issue und Commit. JGit wird als Framework die Implementierung unterstützen, indem Commits aus den Issue ausgelesen werden und anschließend für den Prototypen dieser Arbeit weiterverwendet. Die Systemfunktion SF1 beschreibt die Extraktion des Git

Repositorys für das verwendete Software Entwicklungsprojekt in Jira mit JGit. Hierfür kann die Jira-Projekt AdministratorIn entscheiden, ob die Codezusammenfassung für das entsprechende Projekt verwendet wird oder nicht. Dazu dienen die in Kapitel Abschnitt 3.6 Arbeitsbereiche beschriebenen Arbeitsbereiche WS1.1 All Projects Admin View und WS1.2 Single Project Admin View. WS1.1 bietet eine Übersicht aller Projekte und ermöglicht die (De)Aktivierung einzelner Komponenten von ConDec. Ansicht WS1.2 beschreibt das aktuell ausgewählte Projekt detaillierter und bietet feinere Einstellungsmöglichkeiten wie z.B. welche Typen von Entscheidungswissen verwendet werden sollen.

In den Ansichten war bereits die (De)Aktivierungsmöglichkeit für eine Git Extraktion vorhanden. Jedoch hatte dies noch keine Funktionalität.
Es wurde entschieden, dass beim Aktivieren der Git Extraktion mithilfe des Projektschlüssels und der Git Integration für Jira, das Git Repository geklont wird. Sollte dieser bereits geklont sein, dann wird das Repository lediglich aktualisiert. Beim Abschalten der Git Extraktion, wird das Repository behalten, jedoch nicht regelmäßig aktualisiert.

Die Jira-Projekt AdministatorIn soll mit den in Abbildung 5.2 angezeigte Ansicht die Schiebeknöpfe nutzen, um die die (De)Aktivierung der Funktionalität auszuführen.

Project Name	ConDec Activated?	Store Knowledge in JIRA Issues?	Extract Knowledge From Git?	Extract Knowledge From Issue Comments?
ConDec JIRA	✓	✓	✓	✓

Abbildung 5.2: (De)Aktivierungsmöglichkeit der Git Extraktion in WS1.1

Entwurfsklassendiagramm

JIRA erzeugt WS1.1 und WS1.2 durch Templates. Diese existieren bereits in ConDec. Eine REST-Schnittstelle in der Klasse *ConfigRest* ermöglicht die Übertragung einer Änderung auf Seite der NutzerIn an den Server. Serverseitig wird die Auswahl durch Klasse *ConfigPersistenceManager* gespeichert und das Klonen des Git Repositorys wird in der Klasse *GitClient* ausgelöst. Dies wird dadurch ausgelöst, dass mithilfe der API des Plugins "Git Integration for Jira" die URL zum Git Repository ermittelt werden kann.

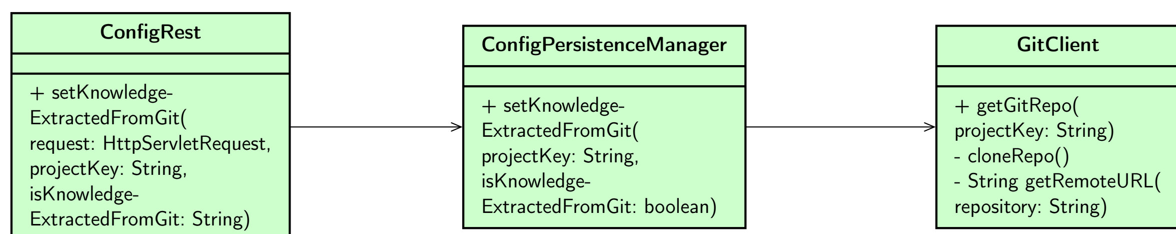


Abbildung 5.3: Entwurfsklassendiagramm für die Git Extraktion

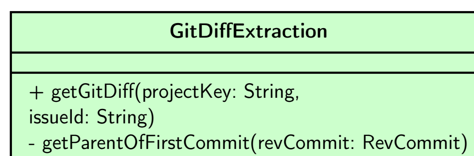
5.2.2 Systemfunktion 2

Entwurfsentscheidung

Die Systemfunktion SF2 beschreibt die Erstellung der Codezusammenfassung. Dazu ist die Ansicht erst einmal irrelevant. Benötigt wird hierfür die Liste aller Commits auf dem ausgewählten Jira Issue. Dabei hilft die API des Plugins "Git Integration for Jira". Durch die Information des Jira Issue Identifikationsnummer, kann auf die Commits zugegriffen werden. Um anschließend eine Zusammenfassung der Codeänderung zu erhalten, muss zunächst einmal mithilfe der Commits die Änderungen aller Dateien extrahiert werden. Dafür wird der letzte Commit und die Elter des ersten Commits benötigt. Diese werden verglichen und zu einem "Diff" zusammengeführt. Da dies eine etwas größere Aufgabe ist, wurde entschieden, dass das zusammenführen der "Diffs" und die Zusammenfassung der Codeänderungen in unterschiedlichen Klassen ausgeführt werden.
Außerdem wurde aufgrund der Komplikationen während der Entwicklung hinsichtlich der Diff-Extraktion und der Anzeige entschieden, dass vorerst *ChangeScribe* nicht angewendet wird und die Codezusammenfassung eine Liste von allen veränderten Klassen und ihre Methoden beinhaltet.

Entwurfsklassendiagramm

Wie in Abbildung 5.4 zu sehen ist, sind die zwei Klassen *GitDiffExtraction* und *TaskCodeSummarizer* zwei verschiedenen Aufgaben zugeteilt. In *GitDiffExtraction* wird wie beschrieben, zunächst einmal die Änderungen aller Dateien im Repository extrahiert. Im *TaskCodeSummarizer* werden die Java-Dateien herausgefiltert, sie zu lesbaren Dateien umformatiert, also "geparset", und anschließend werden die Klassen- und Methodennamen extrahiert.



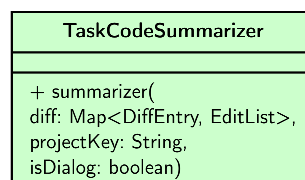


Abbildung 5.4: Entwurfsklassendiagramm für die Erstellung der Codezusammenfassung

5.2.3 Systemfunktion 3

Entwurfsentscheidungen

Die Systemfunktion SF3 beschreibt die Ansicht der Codezusammenfassung in der Ansicht WS1.3.1 Decision Knowledge Code Summarization Dialog und WS1.4.1.1 Jira Issue Module Code Summarization Dialog.

Um den Dialog anzeigen zu lassen, wurde entschieden, dass die NutzerIn in die Ansicht WS1.3 Decision Knowledge View beziehungsweise Ansicht WS1.4.1 Jira Issue Module über einen Rechtsklick auf einer der ausgewählten Jira Issues das Kontextmenü öffnet und über die neu hinzugefügte Funktionalität zur Anzeige vom Dialog führt. Der Dialog zeigt entsprechend die Codezusammenfassung des ausgewählten Jira Issues. Voraussetzung dafür wäre, dass Codeänderungen auch commited und gepusht wurden und dieser mit dem Jira Issue verlinkt.

Entwurfsklassendiagramm

Wie in Kapitel 4 Literaturrecherche beschrieben, wird eine generische Anzeige durch ein Ereignis ausgelöst. In diesem Fall handelt es sich bei dem Ereignis um das Klicken auf die Funktionalität der Codezusammenfassung im Kontextmenü.

Wie in Abbildung 5.5 zu sehen, wird ein Model-View-Controller als Entwurfsklassendiagramm. Dazu dienen die Klassen *CondecContextMenu* aus der JavaScript-Implementation als Controller in gelb markiert. Mithilfe der REST-Schnittstelle in der Klasse *KnowledgeRest* wird eine Übertragung der Änderungen an das Model ermöglicht, welches in der Farbe grün dargestellt wird. Das Model erweitert sich um die Klassen *GitDiffExtraction* und *TaskCodeSummarizer*. Die Klasse *CondecDialog* beobachtet hingegen die REST-Klasse und erhält die Codezusammenfassung, um diese dann in einem Dialog anzuzeigen. Dabei handelt es sich um die View und wird in blau dargestellt.

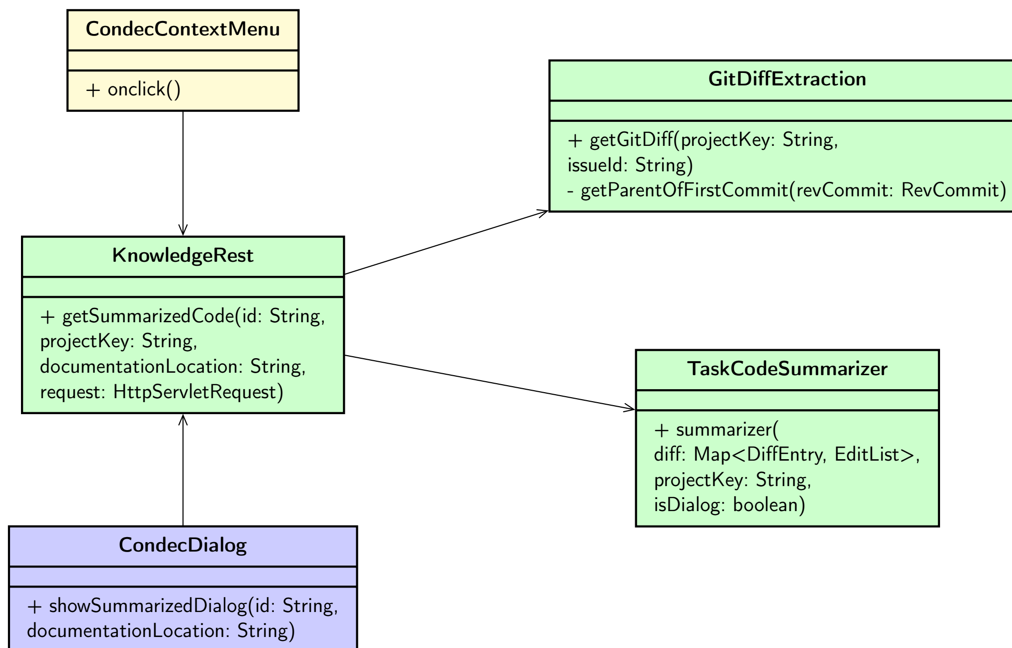


Abbildung 5.5: Entwurfsklassendiagramm für die Ansicht der Codezusammenfassung im Entscheidungsbaum

5.2.4 Systemfunktion 4

Entwurfsentscheidungen

Systemfunktion 4 aus Kapitel Abschnitt 3.3 Funktionale Anforderungen beschreibt die Codezusammenfassung in der Kommentaransicht des Jira Issues, sobald dieser geschlossen wird.
Um eine Codezusammenfassung in der Kommentaransicht anzeigen zu lassen, muss die NutzerIn das Jira Issue den Status auf "Beenden"("Done") setzen. Eine weitere Ausführung ist nicht nötig.

Da die Zeichenlänge in der Kommentarfunktion eingeschränkt ist, wurde entschieden, wie die vollständige Codezusammenfassung angezeigt werden sollte. Wie in Abbildung 5.6 angezeigt, wären die Optionen, zum Einen die Anzeige der vollständigen Codezusammenfassung in der Ansicht WS1.4.1.1 Jira Issue Module Code Summarization Dialog zu überlassen und somit keine weiteren Implementierungen einzuführen. Dies hieße aber auch, dass zur Ansicht der kompletten Codezusammenfassung, die Nutzerin das entsprechende Blatt im Entscheidungsbaum suchen und auswählen muss.
Eine weitere Option wäre, der Codezusammenfassung einen Makro zuzuteilen, und somit das Kontextmenü für diese zu aktivieren. Von dort aus wäre eine direkte Auswahl für den Dialog zur Anzeige der Codezusammenfassung vorhanden.

Da die zweite Option mit dem Makro eine leichtere Bedienbarkeit für die NutzerIn

garantiert, wird auch diese Option ausgewählt und implementiert.

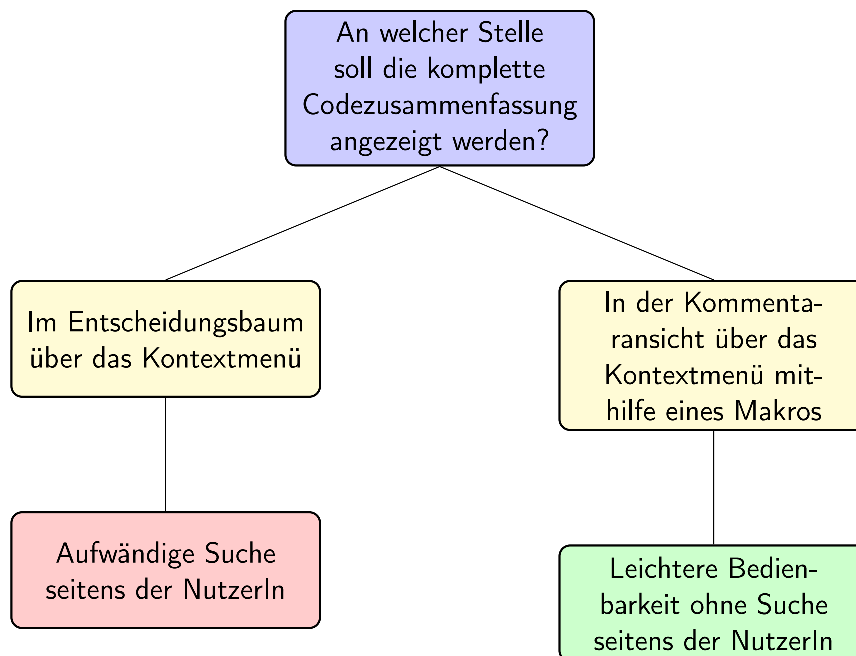


Abbildung 5.6: Entscheidungsbaum zur Anzeige der vollständigen Codezusammenfassung in der Kommentaransicht

Entwurfsklassendiagramm

In Abbildung 5.7 sieht man ein Entwurfsklassendiagramm zur Codezusammenfassung in der Kommentaransicht, Einfügen eines Makros und dem Öffnen des Dialogs mit der kompletten Codezusammenfassung.

Um die Codezusammenfassung nach der Statusänderung des Jira Issues zu ermöglichen, wird ein EventListener benötigt. Dieser erstellt mithilfe der Klassen *GitDiffExtraction* und *TaskCodeSummarizer* eine Codezusammenfassung. Anschließend wird die Zusammenfassung gekürzt, um die Zeichenbegrenzung der Kommentarfunktion von Jira nicht zu überschreiten.

Die gekürzte Codezusammenfassung wird mit der Klasse *CommentSplitterImpl* als einen Satz interpretiert, dem Entscheidungstyp CODESUMMARIZATION zugewiesen und so in die Datenbank der Entscheidungstypen gespeichert.

Anschließend wird mit der Klasse *JiraIssueCommentPersistenceManager* das Makro für den Satz hinzugefügt und als Kommentar in die Kommentaransicht hinzugefügt. Von dort aus wird über Rechtsklick ein Kontextmenü geöffnet, der so aggiert wie in Unterabschnitt 5.2.3 Systemfunktion 3 bereits beschrieben wurde. Hier liegt die Farbeinteilung in grün für die Logik, welches auch das Model des Model-View-Controll beinhaltet, blau für die View und gelb für Controller.

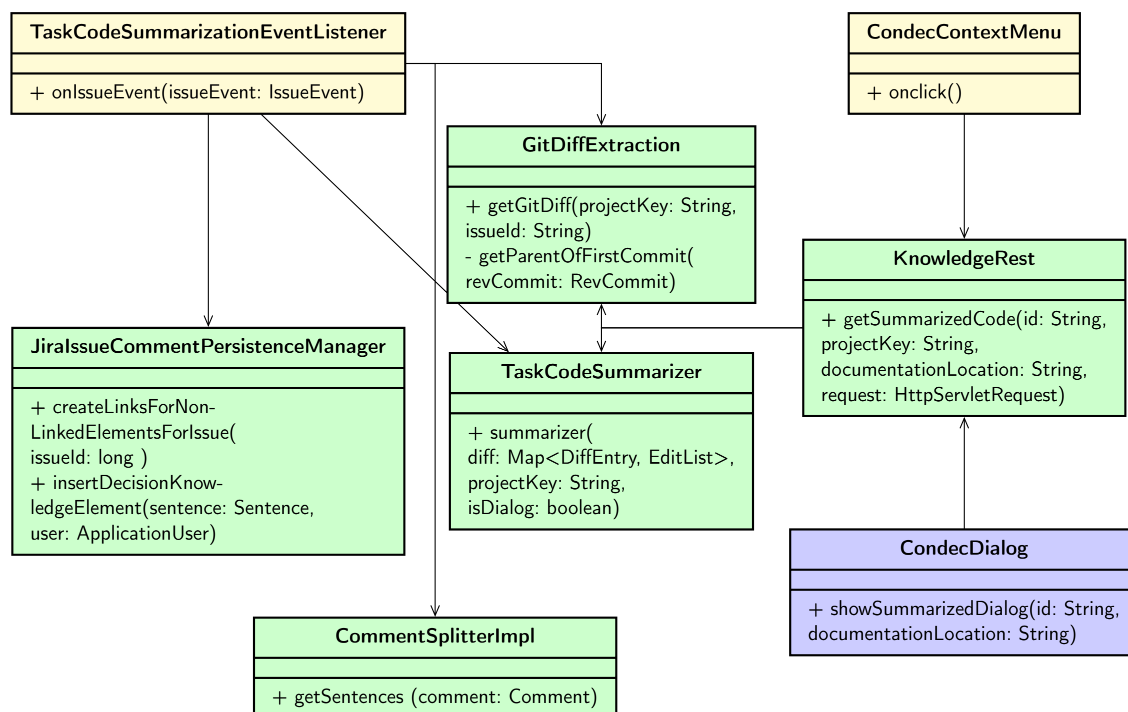


Abbildung 5.7: Entwurfsklassendiagramm für die Codezusammenfassung in der Kommentaransicht

5.3 Addressierung nichtfunktionaler Anforderungen

Nachfolgend wird beschrieben, wie der Entwurf die Erfüllung der nichtfunktionalen Anforderungen adressiert.

5.3.1 Funktionale Angemessenheit

Es werden Äquivalenzklassen abgeleitet und Komponententests, sowie Systemtests durchgeführt. Die Tests werden im Kapitel 6 Qualitätsicherung näher beschrieben.

5.3.2 Benutzbarkeit

Um das schnelle Einlernen zu gewährleisten, werden für die NutzerInnen bereits bekannte Funktionalitäten zum Auslösen der Mechanismen genutzt. So wurde beispielsweise entschieden, dass beim “Beenden” einer Aufgabe, also das setzen auf “Done”, die Codezusammenfassung hinzugefügt wird, da diese Tätigkeit nach der Implementierung auch durchgeführt wird. Auch wurde das bereits bekannte Kontextmenü für die Anzeige der Dialoge ausgewählt. Im Kapitel 7 Evaluation werden mithilfe von Testnutzern und einem Fragebogen die Nutzbarkeit sowie die Ästhetik beurteilt.

5.3.3 Wartbarkeit

Es wird zum Teil nach dem Model-View-Controller implementiert, sodass Logik, Regler und Sicht voneinander getrennt sind. So können alle drei Bereiche getrennt voneinander angepasst und ausgetauscht werden. Es wird darauf geachtet, keine übergroßen Klassen zu erstellen. Jede Sinneinheit bekommt ihre eigene Klasse. Klassen-, Methoden- und Variablennamen werden sinnvoll und verständlich ausgewählt. Der Code wurde für eine bessere Verständlichkeit mit "JavaDocKommentierungen versehen.
Weitere Metriken zur Sicherung von Wartbarkeit, werden mithilfe von statischen Tests im Kapitel 6 Qualitätsicherung ausgeführt.

5.4 UML-Klassendiagramm

In Abbildung 5.8 ist das bei der Entwicklung entstandene UML-Klassendiagramm gezeigt. Zu sehen sind hierbei die Klassen für die Logik in grün, die Anzeige in blau und die Regler in gelb.

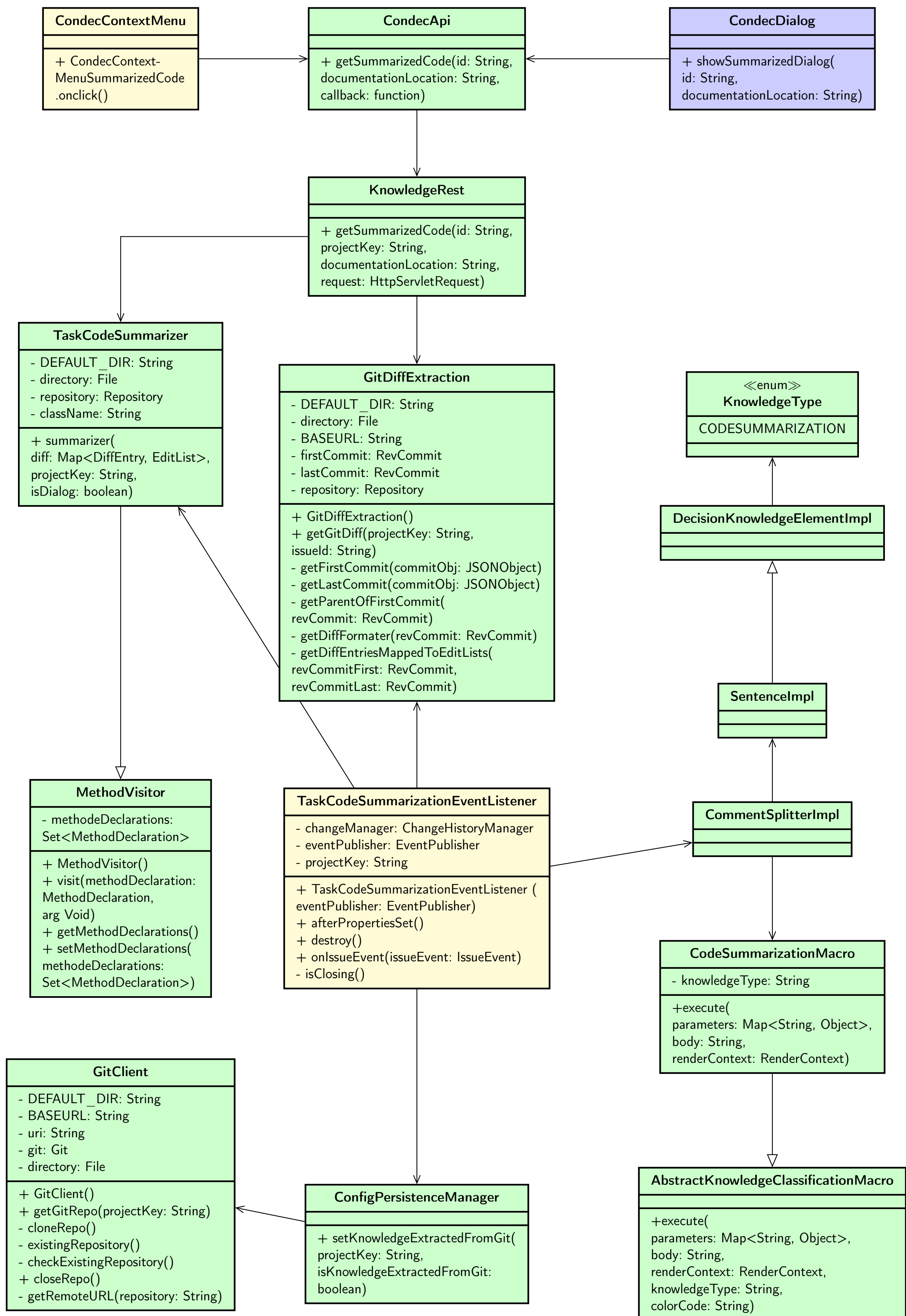


Abbildung 5.8: Klassendiagramm für die Anzeige von Codezusammenfassung

5.5 Ergebnis

In diesem Abschnitt wird anhand von den folgenden Abbildungen, die Ansicht und die Verwendung von den in Kapitel Abschnitt 3.3 Funktionale Anforderungen aufgezählten Systemfunktionen und den in Kapitel Abschnitt 3.6 Arbeitsbereiche aufgezeigten Ansichten.

In Abbildung 5.9 und Abbildung 5.10 sind die Einstellungsmöglichkeiten für die (De)Aktivierung von verschiedenen Modulen. Auch zu sehen ist hierbei die Option für die Git Extraktion. Die AdministratorIn kann hier somit einstellen, ob eine Codezusammenfassung in der Kommentaransicht erwünscht ist.

Abbildung 5.11 zeigt die Ansicht des Jira Issue. Zu sehen ist dabei zum einen die Ansicht WS1.4.1 Jira Issue Module View, also des Entscheidungsbaumes, aber auch die Ansicht der Kommentarfunktion WS1.4.2 Jira Issue Comments. Außerdem befindet sich in der oberen Leiste der Knopf zum Ändern des Status. In Abbildung 5.12 ist nach Änderung des Status auf "Doneëin Kommentar zu sehen. Die Farbe des Kommentars weist darauf hin, dass es mit einem Makro versehen ist und somit durch Rechtsklick auf diesem Kommentar ein Kontextmenü erscheint. Von dort aus, kann die vollständige Codezusammenfassung betrachtet werden.

In Abbildung 5.13 wird die Ansicht WS1.4.1.1 Jira Issue Module Code Summarization Dialog angezeigt. Durch Rechtsklick auf einem Knoten im Entscheidungsbaum, kann ein Jira Issue ausgewählt werden und dessen Codezusammenfassung im Dialog angezeigt werden.

Abbildung 5.14 zeigt die Ansicht WS1.3 Decision Knowledge View. Genauso wie bei Ansicht WS1.4.1 kann auch hier durch Rechtsklick die Codezusammenfassung für das ausgewählte Jira Issue in einem Dialog angezeigt werden. Dies zeigt die Abbildung 5.15 mit der Ansicht WS1.3.1 Decision Knowledge Code Summarization Dialog.

JIRA Dashboards Projects Issues Boards Git Create Search

Administration Search JIRA admin Back to project: LUCENE

Applications Projects Issues Add-ons User management Latest upgrade report System

ATLASSIAN MARKETPLACE
Find new add-ons
Manage add-ons

DECISION KNOWLEDGE
ConDec settings

Continuous Management of Decision Knowledge (ConDec) Settings

Settings for Open Authentication (OAuth)

You need to configure a new application link (with an arbitrary, non-existing URL) and generate a public/private key. Please read and follow these instructions about JIRA application links and public/private keys.

JIRA Home URL: http://localhost:2990/jira
URL to a JIRA Server.

Private Key: MIICeAIBADANBgkqhkiG9w0BAQEFAASCAmIwggJeAgEAAoGBAMhmjkK9M
Private key that matches the public key in Application links (under Incoming Authentication).

Consumer Key: OauthKey
Consumer key that matches the consumer key in Application links (under Incoming Authentication).

Request Request Token

Follow the shown link and allow the access. Copy the shown "secret" into the respective field.

Request Token: tfNOcvwbuLOKoLEJuAiCQmIPXKlrEWfM
The Request token is automatically provided after requesting it.

Secret: ZrdT0s
You need to manually copy the secret into this field.

Request Access Token

Access Token: AWvpOICXs96L7jg4ta23UZkCbggs0CPZ
The access token is automatically provided after requesting it.

Project Settings

Project Name	ConDec Activated?	Store Knowledge in JIRA Issues?	Extract Knowledge From Git?	Extract Knowledge From Issue Comments?
ConDec JIRA	✓	✓	✓	✓
DecXplore Test	✗	✗	✓	✗
LUCENE	✓	✓	✓	✗

Atlassian JIRA Project Management Software (v7.5.0#75005-sha1:fd8c849) · About JIRA · Report a problem

Powered by a free Atlassian JIRA evaluation license. Please consider purchasing it today.

Atlassian

Abbildung 5.9: Ansicht WS1.1 All Projects Admin View

Abbildung 5.10: Ansicht WS1.2 Single Project Admin View

Abbildung 5.11: Ansicht WS1.4 Jira Issue View

Abbildung 5.12: Ansicht WS1.4.2 Jira Issue Comments

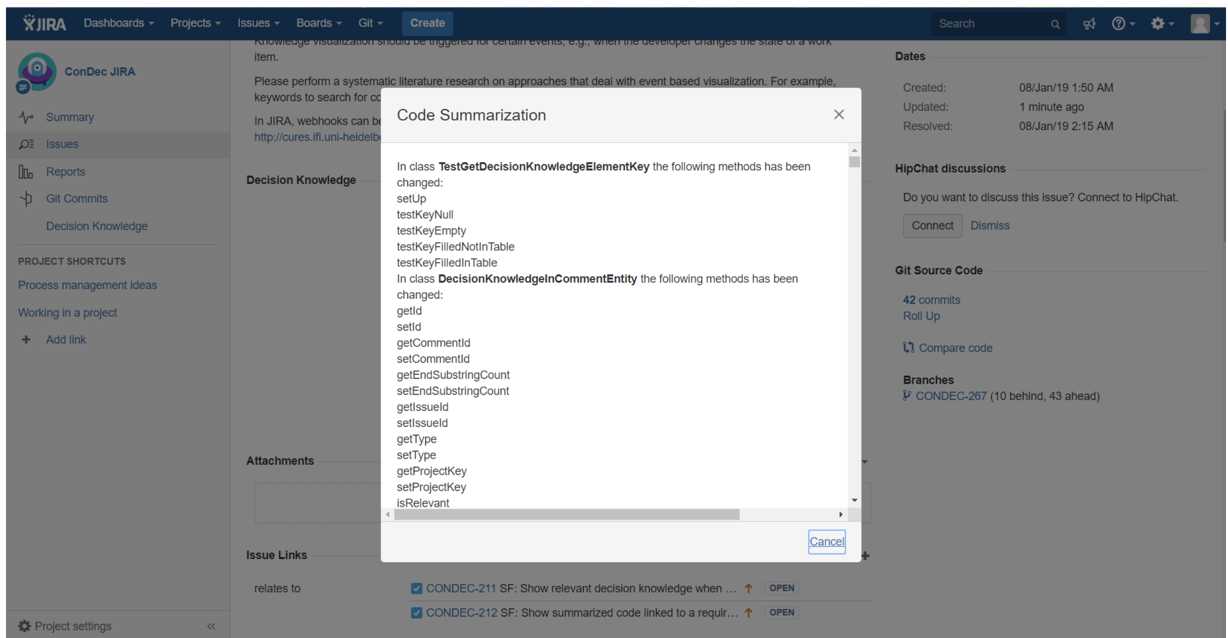


Abbildung 5.13: Ansicht WS1.4.1.1 Jira Issue Module Code Summarization Dialog

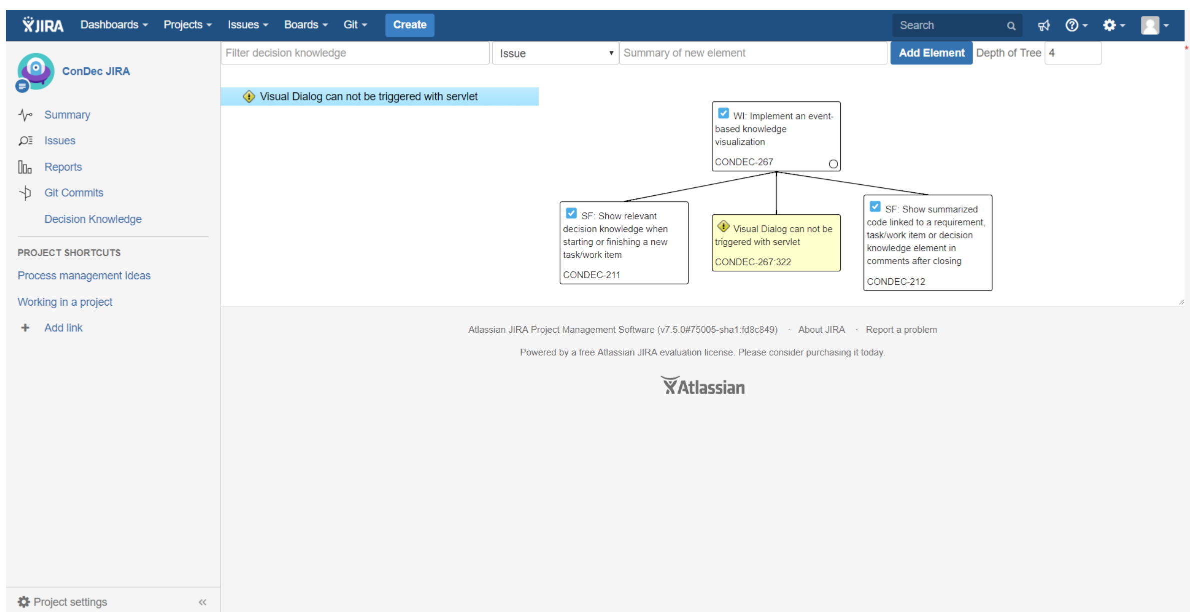


Abbildung 5.14: Ansicht WS1.3 Decision Knowledge View

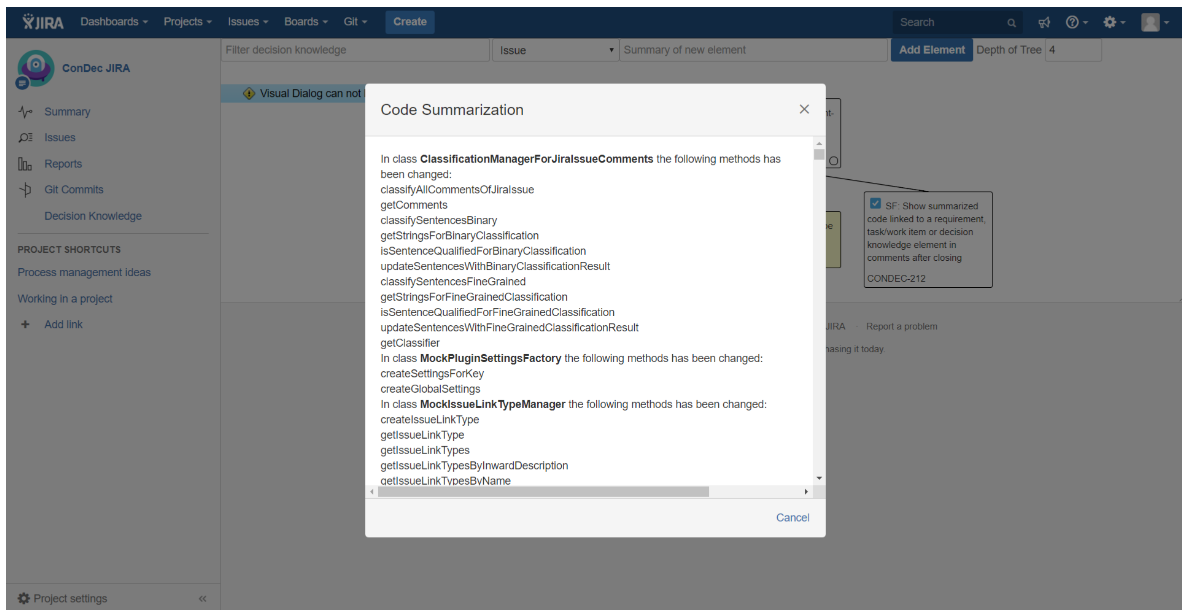


Abbildung 5.15: Ansicht WS1.3.1 Decision Knowledge Code Summarization Dialog

6 Qualitätsicherung

In diesem Kapitel wird die Sicherung der Qualität beschrieben. Dafür werden im ersten Abschnitt die Maßnahmen zur Qualitätssicherung erläutert. Anschließend werden in den folgenden Kapiteln die Komponententests, die Systemtests, die statische Codeanalyse und die kontinuierliche Integration beschrieben, wie sie bereits im ersten Abschnitt auch definiert wurden.

6.1 Maßnahmen zur Qualitätssicherung

Um die nichtfunktionalen Anforderungen aus Kapitel Abschnitt 3.7 zu erfüllen, werden folgende Teststufen ausgeführt:

Für die Betrachtung der Vollständigkeit und der Richtigkeit aller Systemfunktionen, werden Komponententests durchgeführt. Komponententests testen einzelne Funktionen und Module in einer Software. Hierbei wird nach geeigneten gültigen und ungültigen Äquivalenzklassen gesucht. Gültige Äquivalenzklassen testen das Systemverhalten mit validen Eingaben. Ungültige Äquivalenzklassen testen das Systemverhalten mit invaliden Eingaben.

Neben den Komponententests werden zur Sicherung der Vollständigkeit und Richtigkeit aller Systemfunktionen Systemtests durchgeführt. Systemtests prüfen das Plugin auf der Oberfläche. Diese Testfälle sind durch ein genaues Testdurchführungsprotokoll definiert und beschreiben eine erwartete Systemreaktion.

Um die Wartbarkeit zu garantieren, muss der Code durch eine statische Analyse betrachtet werden. Hierbei werden durch Tools im Eclipse IDE Fehler gefunden, sowie auch Metriken wie die zyklomatische Komplexität betrachtet.

6.2 Komponententests

Zur Entwicklung von Komponententests werden für die Funktionalitäten Äquivalenzklassen nach der Black-Box-Methode abgeleitet, um unterschiedliche Eingaben und die zu erwartenden Ausgaben zu testen. Sie werden sowohl lokal, als auch von dem Continuous-Integration-Server TravisCI1 ausgeführt. Dieses System ist in GitHub integriert und testet automatisiert jeden Commit.

Wichtig für das Plugin ist zunächst einmal das Auffinden eines gültigen Git Repositorys. Dafür müssen die Äquivalenzklassen aufgelistet werden, die aussagen, dass das Git Repository zunächst online erreichbar ist, das heißt es ist vorhanden und öffentlich lesbar. Außerdem kann das Git Repository nicht online erreichbar sein. Als nächstes wäre eine gültige Äquivalenzklasse, dass ein Git Repository in einem gültigen Pfad im Dateisystem vorhanden ist, beziehungsweise dass kein Git Repository in einem gültigen Pfad im Dateisystem vorhanden ist. Zu allerletzt wird noch eine ungültige Äquivalenzklasse aufgeführt, dass ein ungültiger Pfad im Dateisystem vorhanden ist.

- ÄK1: Git Repository ist online erreichbar
- ÄK2: Git Repository ist online nicht erreichbar
- ÄK3: Gültiger Pfad im Dateisystem und Git Repository ist vorhanden
- ÄK4: Gültiger Pfad im Dateisystem und Git Repository ist nicht vorhanden
- UK1: Ungültiger Pfad im Dateisystem

Tabelle 6.1: SF1: Extrahiere das Git Repository vom Software Entwicklungsprojekt beim einschalten der Konfiguration

Äquivalenzklasse	Ausgabe
ÄK1	Git Repository wird heruntergeladen
ÄK2	Git Repository wird nicht heruntergeladen
ÄK3	Git Repository wird aktualisiert
ÄK4	Git Repository wird nicht aktualisiert
UK1	Git Repository wird nicht aktualisiert

Eine weitere wichtige Funktionalität des Plugins, ist die Zusammenfassung von Codeänderungen. Dabei gibt es die gültige Äquivalenzklasse, dass keine Commits vorhanden sind, in denen Codeänderungen zusammengefasst werden können. Außerdem Wären Commits ohne den vorausgesetzten Java-Code-Dateien und mit den Java-Code-Dateien vorhanden.

- ÄK1: Keine Commits vorhanden
- ÄK2: Commits ohne Java-Datei vorhanden
- ÄK3: Commits mit Java-Dateien vorhanden

Tabelle 6.2: SF1: Extrahiere das Git Repository vom Software Entwicklungsprojekt beim einschalten der Konfiguration

Äquivalenzklasse	Ausgabe
ÄK1	Keine Codezusammenfassung wird erstellt
ÄK2	Keine Codezusammenfassung wird erstellt
ÄK3	Codezusammenfassung wird erstellt

Aus den genannten gültigen und ungültigen Äquivalenzklassen sind insgesamt 15 Komponententests entstanden.

6.3 Systemtests

In diesem Abschnitt wird der Prototyp gegen seine funktionalen sowie nicht funktionalen Anforderungen getestet. Hauptsächlich wird hierbei die Oberfläche und die korrekte Funktionalität der Komponenten getestet. In Tabelle 6.3 werden alle durchgeführten Systemtests aufgelistet und beschrieben.

Tabelle 6.3: Systemtests des Prototypens

Nr	Vorbedingung	Eingabe	Erwartete Ausgabe	Erfolgreich
1	Ansicht WS1.1, Git Extraktion ist ausgeschaltet	Git Extraktion wird eingeschaltet	Git Extraktion ist eingeschaltet	ja
2	Ansicht WS1.1, Git Extraktion ist einsgeschaltet	Git Extraktion wird ausgeschaltet	Git Extraktion ist ausgeschaltet	ja
3	Ansicht WS1.2, Git Extraktion ist ausgeschaltet	Git Extraktion wird eingeschaltet	Git Extraktion ist eingeschaltet	ja
4	Ansicht WS1.2, Git Extraktion ist eingeschaltet	Git Extraktion wird ausgeschaltet	Git Extraktion ist ausgeschaltet	ja
5	Ansicht WS1.3, Baum zeigt ein Jira Issue mit commit	Rechtsklick, Codezusammenfassung auswählen	Dialog mit Codezusammenfassung wird angezeigt	ja
6	Ansicht WS1.3, Baum zeigt zwei Jira Issues mit commit	Rechtsklick, Codezusammenfassung auswählen	Dialog mit Codezusammenfassung von richtigem Jira Issue wird angezeigt	ja

7	Ansicht WS1.3, Baum zeigt ein Jira Issue ohne commit	Rechtsklick, Codezusammenfassung auswählen	Dialog wird angezeigt, Inhalt keine Codezusammenfassung	ja
8	Ansicht WS1.4.1, Baum zeigt ein Jira Issue mit commit	Rechtsklick, Codezusammenfassung auswählen	Dialog mit Codezusammenfassung wird angezeigt	ja
9	Ansicht WS1.4.1, Baum zeigt zwei Jira Issues mit commit	Rechtsklick, Codezusammenfassung auswählen	Dialog mit Codezusammenfassung von richtigem Jira Issue wird angezeigt	ja
10	Ansicht WS1.4.1, Baum zeigt ein Jira Issue ohne commit	Rechtsklick, Codezusammenfassung auswählen	Dialog wird angezeigt, Inhalt keine Codezusammenfassung	ja
11	Ansicht WS1.4, Jira Issue auf einen Schritt vor "Done", Issue besitzt commits, Git Extraktion an	Jira Issue wird auf "Done"gesetzt	Jira Issue auf "Done", Codezusammenfassung mit Makro in Kommentaren	ja
12	Ansicht WS1.4, Jira Issue auf einen Schritt vor "Done", Issue besitzt commits, Git Extraktion aus	Jira Issue wird auf "Done"gesetzt	Jira Issue auf "Done", Codezusammenfassung nicht in Kommentaren	ja
13	Ansicht WS1.4, Jira Issue auf einen Schritt vor "Done", Jira Issue besitzt keine commits, Git Extraktion an	Jira Issue wird auf "Done"gesetzt	Jira Issue auf "Done", Codezusammenfassung nicht in Kommentaren	ja
14	Ansicht WS1.4, Jira Issue auf zwei Schritte vor "Done", Jira Issue besitzt commits, Git Extraktion an	Jira Issue wird auf einen Schritt vor "Done"gesetzt	Jira Issue auf einen Schritt vor "Done", Codezusammenfassung nicht in Kommentaren	ja
15	Ansicht WS1.4.2, Jira Issue auf "Done", Codezusammenfassung in Kommentar	Rechtsklick, Codezusammenfassung auswählen	Dialog mit Codezusammenfassung	ja

6.4 Statische Codeanalyse

Während der statischen Codeanalyse wird der Code mithilfe der Analysewerkzeuge “FindBugs” und “Metrics” betrachtet. Diese Werkzeuge sind Plugins für das Eclipse IDE. Somit kann man direkt den Code analysieren. Außerdem wird eine statische Codeanalyse von dem Continuous-Integration-Server TravisCI1 durchgeführt.

FindBugs
Mit FindBugs konnten 33 Fehler identifiziert werden, wobei lediglich einer zu den in dieser Arbeit beschriebenen Klassen gehört. Dieser Fehler wurden beseitigt, da ohne Behebung in Zukunft ein NullPointerException entstehen kann, und somit das Speichern eines Git Repositorys nicht mehr möglich. Abbildung 6.1 zeigt eine detaillierte Übersicht der Bugs und ordnet sie nach unterschiedlichen Fehlerklassifizierungen in verschiedenen Kategorien ein.

Abbildung 6.1: FindBugs Analyse

Metrics
Unter der McCabe-Metrik wird die zyklomatische Komplexität eines Software-Moduls verstanden. Laut McCabe darf diese Zahl nicht höher als zehn betragen, da ansonsten das zu testende Modul zu komplex ist und sich somit der Testaufwand stark erhöht [13]. In Tabelle 6.4 sind die für die Entwicklung relevante Klassen mit ihren Komplexitäten aufgezählt. Dort sieht man auch, dass die Komplexitätszahl in der Klasse *TaskCodeSummarizer* den höchstwert überschreitet. Nach einem Refaktorisieren wurde der Durchschnitt auf 4,333 und das Maximum auf 7 in der Methode *summarizer* gesenkt.

Tabelle 6.4: Komplexitätsmetriken

Klasse	Durchschnitt	Maximum	Methode mit der höchsten Komplexität
GitClient	3,429	5	existingRepository
GitDiffExtraction	2,833	5	getGitDiff
MethodVisitor	1	1	MethodVisitor
TaskCodeSummarizationEventListener	2,6	6	onIssueEvent
TaskCodeSummarizer	11	11	summarizer

7 Evaluation

In diesem Kapitel wird eine Evaluation für den Prototypen mit NutzerInnen beschrieben. Dafür werden in Abschnitt eins Fragen für die Evaluation ausformuliert. In Abschnitt zwei wird das Vorgehen der Evaluation beschrieben. Abschnitt drei zeigt die Ergebnisse der Evaluation auf. Anschließend werden in Abschnitt vier die Ergebnisse diskutiert.

7.1 Evaluationsfragen

In diesem Kapitel wird der Prototyp auf seine Gebrauchstauglichkeit für eine EntwicklerIn anhand eines Beispiels evaluiert. Die Aufgabe dieser Arbeit ist, Entscheidungswissen nicht nur zu dokumentieren, sondern auch die in Git gespeicherten Informationen zu nutzen. Der Prototyp soll die EntwicklerIn oder die Rationale ManagerIn dabei unterstützen, eine Konsistenz zwischen dem dokumentierten Entscheidungswissen und den Codeänderungen zu analysieren. Als Grundlage soll hierfür das Softwareprojekt ConDec Jira sein. Diese Arbeit ist ein Teil dieses Softwareprojekts. Dabei handelt es sich um ein Plugin für das ITS Jira zur Erstellung und Bearbeitung von Entscheidungswissen. In diesem Projekt würde zusätzlich das Plugin "Git Integration for Jira" eingebaut. Mithilfe von Jira Issue IDs, die während der Entwicklung in die commit-Nachrichten eingefügt werden, können Veränderungen im Code mit den entsprechenden Jira Issues verlinkt werden.
Dieser Prototyp des Plugins wird mithilfe von Entwicklern des ConDec Projektes und einem Fragekatalog evaluiert.
Um die in Kapitel Abschnitt 3.7 Nichtfunktionale Anforderungen beschriebene nichtfunktionale Anforderung der Nutzbarkeit zu prüfen, wurden 3 Kategorien von Fragen entwickelt.

1. Einfachheit der Nutzung
2. Nützlichkeit der Funktionalität
3. Zukünftiges Nutzungsverhalten

Dafür werden die Funktionalitäten aufgeteilt in Anzeige der Codezusammenfassung in der Kommentaransicht und der Codezusammenfassung im Dialog. Dabei entstanden folgende Aussagen, welche mit einer Zustimmung oder einem Widerspruch beurteilt werden kann:

1. Es ist einfach zusammengefassten Code in Jira Issue-Kommentaren zu erzeugen.
2. Es nutzt der besseren/verständlichenren Dokumentation von Entscheidungen zusammengefassten Code in den Jira Issue-Kommentaren zu erzeugen.
3. Es nutzt zur Reflexion der Konsistenz zwischen Entscheidungswissen und Code zusammengefassten Code in den Jira Issue-Kommentaren zu erzeugen.
4. Ich werde auch in zukünftigen Projekten Codezusammenfassung in Jira Issue-Kommentaren erzeugen lassen.
5. Es ist einfach zusammengefassten Code im Entscheidungsbaum anzuzeigen.
6. Es nutzt der besseren/verständlichenren Dokumentation von Entscheidungen zusammengefassten Code im Entscheidungsbaum anzuzeigen.
7. Es nutzt zur Reflexion der Konsistenz zwischen Entscheidungswissen und Code zusammengefassten Code im Entscheidungsbaum anzuzeigen.
8. Ich werde auch in zukünftigen Projekten Codezusammenfassung im Entscheidungsbaum anzeigen lassen.

7.2 Vorgehen

Zur Befragung sind insgesamt 6 Entwickler des Softwareprojektes gekommen. Ihnen wurde der Fragebogen mit den angegeben Fragen ausgeteilt. Es wurde ihnen zusätzlich erläutert, wie das Plugin zu verwenden ist. Dabei mussten sich die EntwicklerInnen jeweils in zwei Rollen versetzen. Für Frage 1 bis 4, sollten die Fragen aus Sicht einer EntwicklerIn beantwortet werden. Bei Frage 5 bis 8, wurde die Rolle der Rationale ManagerIn betrachtet. Dabei sollten sich die NutzerInnen ein Jira Issue ihrer Wahl mit commits aussuchen, die sie auch schließen wollen. Die Fragen werden mit den Optionen "Starker Widerspruch", "Widerspruch", "Neutral", "Zustimmung" oder "Starke Zustimmung" beantwortet. Zudem können die NutzerInnen eine Erklärung hinzufügen.

7.3 Evaluationsergebnisse

Sechs NutzerInnen wurden zur Evaluation des Prototypen befragt. Die NutzerIn mit der längsten Erfahrung im Projekt, ist seit 19 Monaten dabei. Im Durchschnitt wurde bereits 9 Monate pro NutzerIn am Projekt gearbeitet.

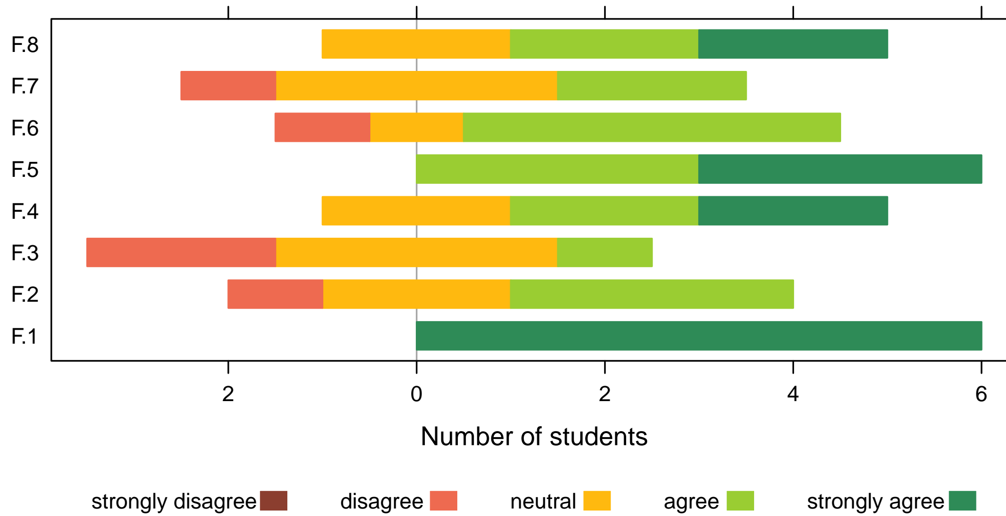


Abbildung 7.1: Evaluationsergebnisse

In Abbildung 7.1 ist zu sehen, dass sich die Meinung zu den Funktionalität sehr stark streut. Bei den Fragen der einfachen Bedienung, ist eine Tendenz zur starken Zustimmung zu sehen. Bei der Nützlichkeit sind sich die NutzerInnen jedoch nicht einig. Im Durchschnitt sehen die NutzerInnen den Prototypen neutral.

In Tabelle 7.1 werden zunächst einmal die von den NutzerInnen als positiv betrachteten Aspekte aufgelistet.
Tabelle 7.2 zeigt die von den NutzerInnen als negativ empfundene Kritik gegenüber dem Prototypen.
Eine Liste von Änderungsvorschlägen wurde dann in Tabelle 7.3 hinzugefügt.
Die Kritik und die Änderungsvorschläge werden im nächsten Kapitel diskutiert.

Tabelle 7.1: Ausgewählte positive Aussagen zu den Funktionalitäten

FrageNR.	Aussagen
1	Es ist sehr einfach, weil man keinen zusätzlichen Workflow einführen muss. Ich kann jede Möglichkeit nutzen ein Issue auf Done zu setzen.
2	Zeigt nochmal die Änderungen an. Insgesamt in Kombination mit dem Issue-Decision-Tree nützlich.
3	Die Anzeige von Codezusammenfassung in den Kommentaren ist hilfreich um Entscheidungen, die in den Kommentaren dokumentiert sind auf Vollständigkeit zu prüfen
4	Es ist ganz nützlich um vielleicht auch den Aufwand in Code-Zeilen zu reflektieren Für einen schnellen Überblick und vor allem für das Überprüfen der Konsistenz von Methodennamen werde ich es sicher benutzen.
5	Das Menü zu nutzen ist leicht und wenn man ConDec kennt sehr logisch die über das Kontextmenü zu machen.
6	Es ist gut geänderten Code mit Entscheidungswissen in Verbindung zu bringen. Durch die Möglichkeit Zusammenfassungen für alle Issues im Baum anzeigen zu können erleichtert es zu prüfen ob alle relevanten Entscheidungen zu dem entsprechenden Issue verknüpft sind.
7	Ein schneller Abgleich zwischen Namen im Code und in der Dokumentation ist möglich.
8	Ich kann als RM gut beurteilen, wie der Fortschritt zu einzelnen WI's ist.

Tabelle 7.2: Ausgewählte Kritiken zu den Funktionalitäten

FrageNR.	Aussagen
1	Vielleicht wäre eine andere Ansicht ansprechbarer.
2	Die Übersichtlichkeit ist gering Bei großen Änderungen ist die Liste sehr lang.
3	Bei sehr vielen Änderungen wird viel in der Zusammenfassung angezeigt. Die Frage ist, wer sich das alles nochmal anschaut um dann die Konsistenz zu den Entscheidungen prüft
4	Ich finde zu lange Zusammenfassungen helfen nicht, da es niemand anschaut. Zu kurze sind dann vermutlich zu abstrakt und helfen auch nicht weiter. Wenn die Daten und Darstellung verbessert werden, würde die Darstellung Sinn machen.
5	Das Aufrufen des Dialogs auf allen möglichen Kindern/Blättern im Baum ist m.M nicht sinnvoll, da es mich als Rationale ManagerIn nicht interessiert wie Code zu nicht verwendete Alternative zusammengefasst ist.
6	Die Übersichtlichkeit ist gering. Wenn in der Summary keine Dinge zusammengefasst werden, die für die Entscheidung ist, dann sehe ich den Nutzen nicht.
7	Ich finde die Anzeige zu grobgranular. Es werden alle Änderungen zu einem WI gezeigt.
8	Abhängig von der Qualität der Zusammenfassung, also der größe und relevanter Inhalt. Beides scheint mir verbesserungswürdig.

Tabelle 7.3: Änderungsvorschläge

FrageNR.	Aussagen
1	-
2	Evtl. kann man hier noch besser nach Klassen clustern bspw. durch aufklappen. Der Issue Kommentar müsste verkürzt werden und eher auf LOC oder ähnliches verweisen.
3	Eine Baumstruktur oder ein Graph mit grober Anzeige die nach Bedarf feiner werden kann würde helfen.
4	Wenn sich für mich irrelevante Änderungen einklappen lassen könnte ich mir eine häufigere und intensivere Nutzung vorstellen.
5	Man könnte noch ein generelles Menu-Item für JIRA-Issues anlegen, sodass das Auslösen auch außerhalb des Baumes geht.
6	Mehr Informationen über den geänderten Code wäre schön
7	Als Rationale Manager habe ich nicht so viel Ahnung von dem Code, vielleicht wäre eine Übersicht mit Anzahl geänderter Zeilen pro Klasse interessant, um den Aufwand zu schätzen. Eine große Übersicht oder Filter könnte die Arbeit erleichtern Eventuell möchte ich alle Änderungen zu einer Entscheidung betrachten, die aber auf ein WI commitet wurden. Eventuell kann man das zeitlich eingrenzen.
8	Ein Einklappen und somit eine bessere Übersicht der Zusammenfassung wären hilfreich und würden mich motivieren die Codezusammenfassung öfter zu benutzen.

7.4 Diskussion

In den Tabellen Tabelle 7.2 und Tabelle 7.3 werden die laut den NutzerInnen aufgezeigten Kritiken und Verbesserungsvorschläge aufgelistet.
In diesem Kapitel wird diskutiert, ob diese Verbesserungsvorschläge umsetzbar wären, beziehungsweise, ob die Funktionalitäten im Allgemeinen als nützlich zu sehen sind. Dabei teile ich die Diskussion wieder in die Nummern der Fragen auf.

Frage 1: Es ist einfach zusammengefassten Code in Jira Issue-Kommentaren zu erzeugen.

Das Erzeugen der Codezusammenfassung durch die Statusänderung auf "Done"wurde im Allgemeinen als einfach empfunden. Ein Vorschlag lautet, dass eventuell eine andere Ansicht ansprechbarer sei. Wie auch im Kapitel Abschnitt 3.1 aufgezählt wurde, bestanden die erste Anforderungen darin, bei einer Statusänderung auf "Done" einen Dialog mit der Codezusammenfassung anzuzeigen. Es ist aufgrund von bereits beschriebenen

Gründen jedoch nicht möglich, das zu implementieren. Eine weitere Möglichkeit wäre, einen zusätzlichen Tab einzufügen. In diesem könnte eine Baumstruktur mit Aufklapp- und Filtermöglichkeiten eingefügt werden. Somit würde dies auch einige Vorschläge der nächsten Fragen aufgreifen.

Frage 2: Es nutzt der besseren/verständlicheren Dokumentation von Entscheidungen zusammengefassten Code in den Jira Issue-Kommentaren zu erzeugen.

Hier wurde oft der Kritikpunkt gebracht, die Zusammenfassung seie unübersichtlich. Es wurde der Wunsch gehegt, zu filtern oder mehr zu clustern. Durch die in Frage 1 gestellte mögliche Option eines Tabs mit Baumstruktur, wäre es gut umzusetzen.
Auch wurde vorgeschlagen, dass eine Verkürzung des Kommentars und das eingrenzen auf die LOC ("Lines of Codes") eine größere Übersichtlichkeit hervorbringen würde. Die Verkürzung der Codezusammenfassung wird, wie auch in Kapitel Abschnitt 8.3 noch beschrieben wird, in Zukunft mithilfe des *ChangeScribe*-Algorithmus verkürzt. Somit sollen nicht wie bisher, Klassen- und Methodennamen aufgelistet werden, sondern eine Zusammenfassung aller neu erstellten Funktionalitäten, die aus den Namen der Klassen und Methoden generiert werden. Damit soll eine Übersichtlichkeit vorhanden sein. Jedoch wäre der Gedanke hinter diesem Prototypen nicht nur die LOC anzuzeigen. Mithilfe der LOC können keine Entscheidungen auf ihrer Konsistenz betrachtet werden.

Frage 3: Es nutzt zur Reflexion der Konsistenz zwischen Entscheidungswissen und Code zusammengefassten Code in den Jira Issue-Kommentaren zu erzeugen.

Auch hier wird die große und unstrukturierte Anzeige kritisiert. Wie bereits bei Frage 1 und 2, würde auch hier ein Tab mit Baumstruktur helfen

Frage 4: Ich werde auch in zukünftigen Projekten Codezusammenfassung in Jira Issue-Kommentaren erzeugen lassen.

Eine zu lange Codezusammenfassung wird hier kritisiert, die unbedeutend für die EntwicklerIn sein soll. Diese würde sich diese Zusammenfassung nicht anschauen wollen. Daher wird auch hier vorgeschlagen, wie bereits bei Frage 2 erwähnt, eine Funktionalitätenzusammenfassung mit *ChangeScribe* einzuführen. Dieser kann dann noch zusätzlich zu einer Baumstruktur in einem zusätzlich Tab hinzugefügt werden.

Frage 5: Es ist einfach zusammengefassten Code im Entscheidungsbaum anzuzeigen.

Hier wird erwähnt, dass eine Zusammenfassung von Codeänderungen in Entscheidungswissen, die nicht mehr verwendetet werden, bspw. Alternativen, als Rationale ManagerIn nicht sinnvoll seien. Jedoch könnte man als Rationale ManagerIn alte Codeänderungen betrachten und somit die Entscheidung nachvollziehen, weswegen es nur als Alternative vorgeschlagen wurde, falls durch beispielsweise neue Mitglieder des Entwicklerteams Fragen aufkommen.
Auch wurde ein zusätzlicher Knopf außerhalb des Kontextmenüs vorgeschlagen. Dies könnte jedoch die Ästhetik beeinflussen.

Frage 6: Es nutzt der besseren/verständlicheren Dokumentation von Entscheidungen zusammengefassten Code im Entscheidungsbaum anzuzeigen.

Wie auch in den vorherigen Fragen aufgezeigt, wurde hier die Übersichtlickkeit und die Menge an Informationen kritisiert. Wie auch in dem vorgeschlagenen Tab, kann auch hier eine Baumstruktur mit Klapp- und Filterfunktionen eingefügt werden.

Frage 7: Es nutzt zur Reflexion der Konsistenz zwischen Entscheidungswissen und Code zusammengefassten Code im Entscheidungsbaum anzuzeigen.

Hier wurde vorgeschlagen, eine Übersicht mit der Anzahl geänderter Zeilen pro Klasse, anstelle der Codezusammenfassung einzufügen. Als Rationale Manager würde es zwar eine Übersicht darüber bringen, ob an der Aufgabe gearbeitet wird, jedoch kann er so nicht analysieren, ob Entscheidungen umgesetzt werden.

Frage 8: Ich werde auch in zukünftigen Projekten Codezusammenfassung im Entscheidungsbaum anzeigen lassen.

Wie bereits in Frage 6 beschrieben, kann auch hier zur Kritik über die Übersichtlichkeit, eine Baumstruktur mit Klapp- und Filtermöglichkeiten in den Dialog eingefügt werden.

Zusammengefasst wurde die Verwendung und das Erlernen der Funktionalitäten als einfach empfunden. Somit wurden die in Kapitel Abschnitt 3.7 Nichtfunktionale Anforderungen aufgelistete nichtfunktionale Anforderung der Benutzbarkeit erfüllt. Jedoch wird oft die Übersicht der Codezusammenfassung kritisiert, die auch im Ausblick überarbeitet werden muss. Dies heißt, dass der Prototyp als gut empfunden wurde, jedoch viele Verbesserungen nötig sind.

8 Schlussfolgerung

In diesem Kapitel wird im ersten Abschnitt eine Zusammenfassung der Arbeit beschrieben. Im zweiten Abschnitt werden die Ergebnisse und die Ausarbeitung diskutiert. Der dritte Abschnitt gibt einen künftigen Ausblick für die Arbeit und der Dokumentation von Entscheidungen in Software-Projekten.

8.1 Zusammenfassung

Diese Arbeit thematisiert die Konsistenz von Entscheidungswissen und Codeänderungen. Dafür wurden zunächst einmal Anforderungen für den Prototypen aufgestellt. Dabei entstand die Forschungsfrage, welches Entwurfsmuster für eine ereignisbasierte Anzeige am besten geeignet wäre. Um dies zu beantworten wurde eine Literaturrecherche zur Findung eines Entwurfsmuster durchgführt. Das Model-View-Controller wurde ausgewählt und im Entwurf der Anzeige von Codezusammenfassung angewendet. Außerdem wurde ein Entwurf für eine ereignisbasierte Codezusammenfassung in der Kommentaransicht entwickelt. Im Anschluss wurde der Entwurf implementiert. Dafür wurde in das schon vorhandene Plugin ConDec für Jira weiterentwickelt. Um die Qualität diesen Prototypen zu garantieren, ist mithilfe von mehrere Teststufen eine Qualitätskontrolle durchgeführt worden. Für die anschließende Beurteilung der Nutzung des Prototypes, wurde eine abschließende Evaluation durchgeführt.

8.2 Diskussion

Zu Beginn wurde entschieden, dass eine Codezusammenfassung im Sinne von *ChangeScribe*[11] einzuführen sei. Aufgrund von Schwierigkeiten in der Entwicklung von der Kommunikation mit Git, konnte nur eine Auflistung aller geänderten Klassen- und Methodennamen als Codezusammenfassung eingeführt werden. Jedoch hat sich während der Evaluation herausgestellt, dass auch die Auflistung in einer übersichtlichen Ansicht, wie einer Baumstruktur mit Klapp- und Filtermöglichkeiten, wünschenswert gewesen wäre. Daher wäre eventuell eine Mischung aus Codezusammenfassung mit *ChangeScribe* in natürlicher Sprache und einer Auflistung in Baumstruktur eine bessere Option.

Außerdem hätte eine von der Statusänderung ausgelöste Dialogansicht eingebaut werden müssen. Mithilfe einer Literaturrecherche wurde hierfür ein Entwurfsmuster zur Entwicklung einer generischen ereignisbasierten Anzeige ausgewählt. Aufgrund von Kom-

plikationen bei der Kommunikation zwischen dem Backend und dem Frontend, wurde anschließend entschieden, einen Dialog auszulösen, sobald es aus dem in den Jira Issue vorhandenen Entscheidungsbaum ausgewählt wurde, welche eine Zusammenfassung von Codeänderungen anzeigt. Außerdem soll das Ereignis des Schließens eines Jira Issues eine Zusammenfassung der Codeänderungen in der Kommentaransicht des Jira Issues angezeigt werden.

Das Plugin kann ausschließlich Java-Projekte ausführen und benötigt ein für die Öffentlichkeit einsehbares Projekt. Zudem muss mit "Git Integration for Jira" eine Verbindung zwischen Git und Jira vorhanden sein. Sobald dann Commits mithilfe der Jira Issue IDs mit den Jira Issues verlinkt sind, können durch das Schließen oder das Öffnen des Dialogs eine Codezusammenfassung angezeigt werden.

Codeänderungen zusammenzufassen und sie in den Jira Issues anzuzeigen kann den Vorteil bringen, dass EntwicklerInnen und Rationale ManagerInnen einen Überblick über Konsistenz zwischen Codeänderungen und Entscheidungswissen erhalten können. Jedoch steht offen, wie verständlich dies für die einzelnen Beteiligten ist. Sollte beispielsweise eine Rationale ManagerIn wenig bis gar keine Ahnung zu dem ausgeführten Code haben, so können zu große oder zu kleine Zusammenfassungen von Codeänderungen nicht viel bringen, da ein Verständnis für diese fehlt. Ab welcher Größe zu groß oder zu klein gilt, ist noch zu untersuchen.

8.3 Ausblick

Neben den vorhandenen Funktionalitäten der Git-Integration und der eventbasierten Anzeige der Codezusammenfassung können noch weitere Funktionalitäten in Zukunft implementiert werden. Daher ist die Anzeige zunächst generisch und die Implementierung ist Erweiterbar.

Eine zukünftige Änderung wäre zunächst einmal die Ansicht der Codezusammenfassung. Dies wurde, wie in Kapitel Kapitel 7 Evaluation mit zwei Optionen aufgeführt. Eine Übersicht durch eine Baumstruktur, in denen die Klassen- und Methoden auf- und zugeklappt werden können und die Filtermöglichkeiten hätte, wäre zunächst eine Option. Eine weitere Option wäre die Einführung von *ChangeScribe* [11] um eine Zusammenfassung mit natürlicher Sprache einzuführen. Diese beide Optionen müssen untersucht werden und eventuell eines oder sogar beide implementiert werden.

Eine weitere Änderung im Plugin beinhaltet die Git-Integration. Zurzeit ist noch notwendig, dass ein öffentlich einsehbares Projekt auf Git liegt, und somit keine Logindaten benötigt werden. Eine Erweiterung der Implementierung soll die Abfrage von Logindaten ermöglichen und somit sollen alle Git-Projekte eingebunden werden.

Die Codezusammenfassung bezieht sich zur Zeit nur auf die Codeänderungen, die direkt mit dem Jira Issue verlinkt sind. So kann beispielsweise eine EntwicklerIn aber eine Codezusammenfassung von mehreren Jira Issues in einem übergeordneten Issue

fordern. Das bedeutet, das Plugin soll die Codeänderungen aller verlinkten Jira Issues extrahieren und zusammenfassen.

Diese Änderungen sollten sowohl der EntwicklerIn als auch der Rationale ManagerIn bei der Analyse der Konsistenz zwischen Entscheidungswissen und Codeänderungen in Jira in Zukunft helfen.

9 Literatur

[1] A. Ampatzoglou, S. Charalampidou und I. Stamelos, „Design Pattern Alternatives: What to do when a GOF pattern fails“, in *Proceedings of the 17th Panhellenic Conference on Informatics - PCI ’13*, ACM Press, 2013.

[2] S. Burbeck, „Applications Programming in Smalltalk-80 (TM): How to use Model-View-Controller (MVC)“, 1987.

[3] P. P. Churi, S. Wagh, D. Kalelkar und M. Kalelkar, „Model-View-Controller Pattern in BI Dashboards: Designing Best Practices“, IEEE, 2016.

[4] E. Gamma, *Design patterns, Elements of reusable object-oriented software*, eng, 37th printing, Ser. Addison-Wesley professional computing series. Boston: Addison-Wesley, 2009, Online–Ressource (xv, 395 p.)

[5] N. Grushka, M. Jensen und L. Lo Iacono, „A Design Pattern for Event-Based Processing of Security-Enriched SOAP Messages“, in *2010 International Conference on Availability, Reliability and Security*, IEEE, Februar 2010.

[6] T.-M. Hesse, A. Kuehlwein, B. Paech, T. Roehm und B. Bruegge, „Documenting implementation decisions with code annotations“, in *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering*, KSI Research Inc. und Knowledge Systems Institute Graduate School, Juli 2015.

[7] P. Iaría und U. Chesini, „Refining the Observer Pattern: The Middle Observer Pattern“, 2001.

[8] A. Kleebaum, J. O. Johanssen, B. Paech, R. Alkadhi und B. Bruegge, „Decision knowledge triggers in continuous software engineering“, in *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering - RCoSE ’18*, ACM Press, 2018.

[9] A. Kleebaum, J. O. Johanssen, B. Paech und B. Bruegge, „Tool support for decision and usage knowledge in continuous software engineering“.

[10] G. E. Krasner und S. T. Pope, „A Cookbook for Using the Model- View-Controller User Interface Paradigm in Smalltalk-80“, 1988.

[11] M. Linares-Vasquez, L. F. Cortes-Coy, J. Aponte und D. Poshyvanyk, „ChangeScribe: A tool for automatically generating commit messages“, in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, Mai 2015.

[12] J. Liu, H. Yin und Y. Wang, „A Novel Implementation of Observer Pattern by Aspect Based on Java Annotation“, IEEE, 2010.

[13] T. J. McCabe, „A complexity measure“, 4, Bd. SE-2, Dezember 1976, S. 308–320.

[14] B. Paech und A. Kleebaum. (2016). Richtlinien zur Literaturrecherche, Adresse: `http://confluence-se.ifi.uni-heidelberg.de/pages/viewpage.action?pageId=10225030`.

[15] A. Paschke, P. Vincent, A. Alves und C. Moxey, „Tutorial on Advanced Design Patterns in Event Processing“, 2012.

[16] M. Springer, H. Masuhara und R. Hirschfeld, „Classes as Layers: Rewriting Design Patterns with COP - Alternative Implementations of Decorator, Observer, and Visitor“, ACM Press, 2016.

[17] A. Syromiatnikov und D. Weyns, „A Journey through the Land of Model-View-Design Patterns“, in *2014 IEEE/IFIP Conference on Software Architecture*, IEEE, April 2014.

Abbildungsverzeichnis

2.1 Entscheidungsdokumentationsmodell [6] 5
2.2 Beziehungen zwischen Softwareartefakten und Entscheidungswissen [8] . 6

3.1 Domänendatendiagramm für die Anwendung des Prototypens 14
3.2 Nutzungsdiagramm für die Anwendung des Prototypens 15
3.3 Arbeitsbereich für die Anwendung des Prototypens 16

4.1 Event Pipeline Pattern [5] 25
4.2 Model-View-Controller [17] 26
4.3 Model-View-View Model [17] 27
4.4 Dolphin Smalltalk Model-View-Presenter [17] 28
4.5 Passive View Pattern [17] 28
4.6 Observer Pattern [4] 29
4.7 MiddleObserver Pattern [7] 30
4.8 Identification Pattern [15] 32

5.1 Entwurfsklassendiagramm der alten Grobanforderungen 36
5.2 (De)Aktivierungsmöglichkeit der Git Extraktion in WS1.1 38
5.3 Entwurfsklassendiagramm für die Git Extraktion 38
5.4 Entwurfsklassendiagramm für die Erstellung der Codezusammenfassung 39
5.5 Entwurfsklassendiagramm für die Ansicht der Codezusammenfassung im Entscheidungsbaum 41
5.6 Entscheidungsbaum zur Anzeige der vollständigen Codezusammenfassung in der Kommentaransicht 42
5.7 Entwurfsklassendiagramm für die Codezusammenfassung in der Kommentaransicht 43
5.8 Klassendiagramm für die Anzeige von Codezusammenfassung 45
5.9 Ansicht WS1.1 All Projects Admin View 47
5.10 Ansicht WS1.2 Single Project Admin View 48
5.11 Ansicht WS1.4 Jira Issue View 49
5.12 Ansicht WS1.4.2 Jira Issue Comments 50
5.13 Ansicht WS1.4.1.1 Jira Issue Module Code Summarization Dialog 51
5.14 Ansicht WS1.3 Decision Knowledge View 51
5.15 Ansicht WS1.3.1 Decision Knowledge Code Summarization Dialog . . . 52

6.1 FindBugs Analyse 57

7.1 Evaluationsergebnisse 61

Tabellenverzeichnis

3.1 SF1: Extrahiere das Git Repository vom Software Entwicklungsprojekt . 12
3.2 SF2: Zeige zusammengefasste Codeänderungen von verlinkten Anforderungen, Aufgaben oder Entscheidungswissen im Entscheidungsbaum . . 12
3.3 SF3: Zeige zusammengefassten Codeänderungen von verlinkten Anforderungen, Aufgaben oder Entscheidungswissen im Entscheidungsbaum . . 13
3.4 SF4: Zeige zusammengefasste Codeänderungen von Anforderungen, Aufgaben oder Entscheidungswissen beim Beenden einer Aufgabe nach dem Beenden 14

4.1 Ergebnisse der Suche mit Suchtermen 20
4.2 Ergebnisse der Suche mit Snowballing 22
4.3 Ergebnisse der Literatur 23
4.4 Synthese der Literatur 33

6.1 SF1: Extrahiere das Git Repository vom Software Entwicklungsprojekt beim einschalten der Konfiguration 54
6.2 SF1: Extrahiere das Git Repository vom Software Entwicklungsprojekt beim einschalten der Konfiguration 55
6.3 Systemtests des Prototypens 55
6.4 Komplexitätsmetriken 58

7.1 Ausgewählte positive Aussagen zu den Funktionalitäten 62
7.2 Ausgewählte Kritiken zu den Funktionalitäten 63
7.3 Änderungsvorschläge 64

Abkürzungsverzeichnis

ITS Issue-Tracking-System 3
VCS Version Control System 3
PLoP Pattern Language of Programs 19
R Grobanforderung 8
UT User Task 10
SF Systemfunktion 11
ST User Subtask 10
WS Workspace 15

Eidesstaatliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.
Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht.
Die Arbeit ist in gleicher oder vergleichbarer Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Vita Aman