Friedrich-Alexander-Universität Erlangen-Nürnberg

Technische Fakultät, Department Informatik

JOHANNES PFANN

MASTER THESIS

# MEASURING THE PATCH REVIEW PROCESS IN OPEN AND INNER SOURCE

Eingereicht am 21. April 2017

Betreuer: Prof. Dr. Dirk Riehle, M.B.A., M.Sc. Maximilian Capraro
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

# Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

---

Erlangen, 21. April 2017

# License



---

Erlangen, 21. April 2017

# Abstract

Inner source development is the application of open source practices for a company's internal software development. One of the practice is called review process. This process separates the code contribution from it's integration. In inner source, the review process is not researched.
Therefore, a suitable software for measuring this process is required for research purposes. The measuring instruments for inner source development used today are not capable of examining the review process.
This thesis develops an extension of an existing application for analysing review processes in Inner Source.
To evaluate the functionality of this application, it is applied to selected projects. The collected data is used to demonstrate that they are suitable for answering typical questions for review processes.
For further research the extension allows mesearument of the review processes in inner source projects.

# Zusammenfassung

Inner-Source-Entwicklung ist die Anwendung von Open-Source-Praktiken in Unternehmensstrukturen. Eine Praktik wird Review-Prozess bezeichnet. Dieser Prozess trennt den Codebeitrag von dessen Integration. In Inner Source ist der Review-Prozess nicht erforscht.
Daher wird für Forschungszwecke eine geeignete Software zum Messen dieses Prozesses benötigt. Die bisher eingesetzten Messinstrumente für Inner-Source-Entwicklung sind nicht in der Lage den Review-Prozess zu untersuchen.
Die Arbeit beschäftigt sich mit der Erweiterung einer bestehenden Anwendung um Informationen über die Review-Prozesse in Inner Source zu erfassen.
Um die Funktionalität dieser Erweiterung zu evaluieren, wird diese auf ausgewählten Projekten angewendet. Mit den gesammelten Daten wird demonstriert, dass diese zur Beantwortung typischer Fragestellungen geeignet sind.
Die Erweiterung ermöglicht für weitere Forschungen die Messung der Review-Prozesse in Inner-Source-Projekten.

# Inhaltsverzeichnis

# 1 Einleitung

Inner Source ist die Verwendung von Open-Source-Praktiken innerhalb von Unternehmensgrenzen (Capraro and Riehle (2016)). Anders als bei Open-Source-Projekten wird der Source Code nicht weltweit zugänglich gemacht, sondern lediglich für Mitarbeiter des Unternehmens geöffnet. Des Weiteren werden existierende Softwareentwicklungsprozesse um Methoden aus der Open-Source-Entwicklung ergänzt und erweitert.
Ziele von Inner Source sind die Verbesserung der Wissensweitergabe und -verteilung über Abteilungsgrenzen hinweg, sowie die Verbesserung der Wiederverwendung von Code innerhalb der Entwicklungsorganisationen (Riehle et al. (2016)).
Eine dieser Methoden wird als Review-Prozess bezeichnet. Bei dieser Methode wird eine Codeänderung, ein sogenannter Patch, und dessen Integration getrennt. Jeder Entwickler erhält die Möglichkeit einen Patch zu verfassen, jedoch können nur ausgewählte Entwickler diese Patches annehmen und in die Software integrieren oder ablehnen. Der Review-Prozess erlaubt anhand der angenommen und abgelehnten Patches einen Einblick auf die Effizienz der Entwicklung und letztlich auch auf die Qualität der Umsetzung von Inner Source in Unternehmensorganisationen.
Die bisher eingesetzten Messinstrumente zur Untersuchung von Inner-Source-Projekte sind nicht in der Lage den Review-Prozess zu untersuchen. Daher wird in dieser Arbeit eine existierende Software dahingehend erweitert, dass diese den Review-Prozess berücksichtigt. Der Beitrag dieser Arbeit besteht aus folgenden Punkten:

- Implementierung einer Komponente zum Erfassen von Reviews für verschiedene Review-Werkzeuge.
- Implementierung von Clients für die Review-Werkzeuge GitHub und GitLab.
- Aufbereitung der Review-Daten zur Untersuchung eines Review-Prozesses;
- Integration dieser Komponente in eine bestehende Softwarelösung;
- Evaluierung der Komponente durch Ausführung von ausgewählten Projekten;

Zur Evaluation wird die erweiterte Software an ausgewählte Projekte angewendet um mit den gewonnenen Informationen typische Fragestellungen zu beantworten.
Der Aufbau der Arbeit ist wie folgt gegliedert: In Kapitel 2 werden Forschungen des Review-Prozess in Open Source und ähnliche Softwarelösung zum Vermessen von Projektdaten vorgestellt. Ergänzend wird darüber diskutiert, welchen Beitrag diese Arbeit leisten dazu leisten kann. In Kapitel 3 wird der Zweck der Arbeit vorgestellt und Anforderungen für die Erweiterung definiert. In Kapitel 4 wird die Architektur und das Design der Software erklärt. Kapitel 5 beschäftigt sich mit der konkreten Implementierung. Die entwickelte Software wird in Kapitel 6 evaluiert und anhand ausgewählter Projekte getestet, um daraus typische Fragestellungen über den Review-Prozess zu beantworten.

# 2 Related Work

In Open Source ist die Bedeutung von Review-Prozessen bereits gut erforscht. Anhand dessen wurde diese Arbeit von vier früheren Untersuchungen der Review-Prozessen in Open Source inspiriert, um ähnliche Untersuchungen für Inner-Source-Projekte durchzuführen. Außerdem werden bestehende Softwarelösungen zur Untersuchung von Projekten vorgestellt und diskutiert.

## 2.1 Literatur

Nachfolgend werden Arbeiten aufgeführt und für diese Arbeit relevante Erkenntnisse diskutiert. Die Faktoren, die aus diesen Arbeiten Einfluss auf den Review-Prozess nehmen, werden bei der Implementierung berücksichtigt, damit diese später auf Inner-Source-Projekte portiert werden können.
Die erste Arbeit (Georgios Gousios (2014a)) beschäftigt sich mit Pull Requests und untersucht diese anhand ausgewählter Projekte auf GitHub. Das Ergebnis dieser Studie zeigt, dass Pull Requests die Zusammenarbeit der Community verbessert und die Zeit zur Integration neuer Softwarebausteine verkürzt. Jedoch werden nur Projekte untersucht die sich auf GitHub befinden. Mit der Software dieser Arbeit kann diese Forschung auf weitere Plattformen wie GitLab ausgedehnt werden.
Eine weitere (Georgios Gousios (2014b)) betrachtet die Patches eines Reviews. Die Annahme der Arbeit ist, je kleiner ein Patch, desto schneller

werden diese bearbeitet und angenommen. Beide haben sich durch eine Untersuchung von zwei Open-Source-Projekten bestätigt. Die Softwarelösung aus dieser Arbeit betrachtet ebenfalls Charakteristiken eines Patches und verknüpft diese zu den dazugehörigen Reviews. Es ist also möglich die gleichen Forschungsfragen auf mehr als zwei Projekten zu untersuchen und diese auf Inner-Source-Projekte zu erweitern.
Die Arbeit Jiang et al. (2013) befasst sich mit der Frage, aus welchen Gründen Patches akzeptiert bzw. abgelehnt werden. Die Untersuchung befasst sich mit der Entwicklung des Linux Kernel und fanden heraus, dass Patches von bekannten Entwickler öfters und schneller akzeptiert wurden. Die Ergebnisse beziehen sich nur auf den Linux Kernel und können daher nicht allgemein auf andere Projekte übertragen werden. Mit der Erweiterung des Patch-Flow Crawlers kann diese Untersuchung auf andere Projekte ausgedehnt werden.
Die letzte Arbeit (Olga Baysal (2013)) beschäftigt sich mit Faktoren wie der Größe der Patches, der Priorität, aus welchen Komponenten diese stammen und welche Organisationseinheiten bzw. Personen diese beigetragen haben. Diese Faktoren wurden wieder an Open-Source-Projekten untersucht. Das Ergebnis zeigt, dass alle Faktoren Einfluss auf die Bearbeitungszeit und für die Akzeptanz der Reviews nehmen. Die Arbeit beschränkt sich jedoch nur auf Open-Source-Projekte. Da alle aufgelisteten Kriterien auch mit dieser Arbeit erfasst werden können, ist es möglich die Ergebnisse mit Inner-Source-Projekten zu vergleichen.

## 2.2 Existierende Softwarelösungen

Es gibt ähnliche Softwarelösungen zur Untersuchung von Projekten, nämlich GHTorrent und Produkte von der Firma Bitergia.
GHTorrent ist eine Open Source Lösung zur Analyse von GitHub Projekten (Gousios (2013)). Für den Zweck, Inner-Source-Projekte zu untersu-

chen ist GHTorrent ungeeignet, da GHTorrent nur Daten von Projekten die auf GitHub gehostet werden bereitstellt.
Die andere Alternative sind die Produkte von dem Unternehmen Bitergia. Dieses bietet Bitergia Dashboards (Bitgeria (a)) und Bitergia Reports (Bitergia (2015)) an. Beide Produkte sind zur Analyse und Auswertung von Metriken der Softwareentwicklung gedacht. Analysiert werden unter anderem verschiedene Versionskontrollsysteme, Review Tools, Foren sowie Chats. Außerdem verspricht das Unternehmen auf kundenspezifische Bedürfnisse einzugehen. Beide Lösungen bauen auf das Open-Source-Projekt GrimoireLab auf (Bitgeria (b)). Hier ergeben sich zwei Möglichkeiten. Die erste könnte sein, die Software der Firma zu nutzen. Der Nachteil dieser ist, dass für Inner-Source-Projekte die aktuelle Software von dem Dienstleister bei Bedarf angepasst werden muss. Die zweite Möglichkeit wäre, die Open-Source-Lösung auf bestimmte Anforderungen anzupassen. Dieses erfordert allerdings eine Einarbeitungszeit der Software. Dagegen spricht, dass diese Einarbeitungszeit auch für eine Eigenentwicklung genutzt werden kann. Außerdem ist das Verständnis der Software höher bei einer eigenen Lösung.

# 3 Artefakt

In diesem Kapitel geht es um die Anforderungen für eine Komponente zur Untersuchung von Review-Prozessen in Open- und Inner-Source-Projekten. Zuerst wird der Zweck der Erweiterung dargelegt um daraus Anforderungen zu erstellen. Die Entwicklung erweitert den Patch-Flow Crawler um eine Komponente, Reviews in einem Projekt zu erfassen und dessen Daten aufzubereiten.

## 3.1 Zweck

Die Entwicklung zur Untersuchung von Review-Prozessen erweitert den bereits entwickelten Patch-Flow Crawler. Dieser erfasst alle Patches eines Projekts und analysiert die Zusammenarbeit der Organisationseinheiten mit dessen Projektmitgliedern. Um den Review-Prozess als einen weiteren Aspekt der Zusammenarbeit einzubeziehen, müssen zusätzlich Reviews innerhalb eines Projektes betrachtet werden.
Ziel dieser Untersuchung von Inner-Source-Projekten ist, einen Einblick auf die Zusammenarbeit und Effizienz der Projektmitglieder und letztlich auch auf die Qualität der Umsetzung von Inner-Source-Praktiken in Unternehmensstrukturen zu erhalten.
Das Ergebnis dieser Arbeit ist die Erweiterung des Patch-Flow Crawlers um eine Komponente zur Erfassung der Reviews.
Es ist notwendig, dass die Software in unterschiedlichen Unternehmen eingesetzt werden kann. In diesen soll ein einheitliches Datenmodell erstellt

werden, dass Informationen über Personen, deren Abteilungen, den Patches und den dazugehörigen Reviews verbindet. Mit diesen Daten sollen Inner-Source-Projekte untersucht werden. Um die Erweiterung zu evaluieren, werden verschiedene Projekte auf folgende typische Fragestellungen zu einem Review-Prozess beantwortet.

1. Wie viele Patches werden in Open-Source- und Inner-Source-Projekten abgelehnt?
2. Wie lange benötigt ein Review durchschnittlich?
3. Gibt es Unterschiede zwischen Open Source und Inner Source Reviews?
4. Welche Unterschiede gibt es zwischen Review-Prozessen bei Inner Source und Open Source?

Da kein Zugriff auf ein Inner-Source-Projekt möglich war, wurde im Rahmen dieser Arbeit die Fragestellungen abgeändert und ausschließlich auf Open-Source-Projekten getestet.

1. Wie viele Patches werden in Open-Source-Projekten abgelehnt?
2. Wie lange benötigt ein Review durchschnittlich?
3. Wie viel Patches werden durchschnittlich gereviewed?
4. Wie viel Personen sind Durchschnittlich an einem Review beteiligt?

Hintergrund der ersten Frage ist die Annahme, dass die Anzahl der abgelehnten Patches auf die Codequalität der eingereichten Patches zurückzuführen ist. Die Dauer einer Bearbeitung eines Reviews und die Anzahl der Patches die gereviewed werden, lassen möglicherweise Rückschlüsse auf die Umsetzung des Review-Prozesses schließen. Abschließend kann mit der letzten Frage ein Einblick auf die Zusammenarbeit in dem Review-Prozess geworfen werden.

## 3.2 Anforderungen

Nachdem der Leistungsumfang der neuen Software vorgestellt wurde, können daraus Anforderungen generiert werden. Im Folgenden werden diese in funktionale und nichtfunktionale Anforderungen unterteilt.

### Nichtfunktionale Anforderungen

Aus der Zielsetzung in Abschnitt 3.1 ergeben sich zwei grundlegend technische Anforderungen.

**NF1** Der Patch-Flow Crawler muss um eine neue Komponente erweitert werden und die Informationen der Patches mit den Reviews verbinden;

**NF2** Um Forschungen in verschiedenen Unternehmen zu realisieren, muss die Erweiterung für verschiedene Systeme angepasst werden können;

### Funktionale Anforderungen

In Abschnitt 3.1 wurden typische Fragestellungen zu Review-Prozessen vorgestellt. Um die Fragestellung zu beantworten, wie viel Patches nicht in einem Review enthalten sind, muss es eine Zuordnung von Patches zu Reviews geben. Zusätzlich müssen die Reviews die Information über dessen Bearbeitungszustand enthalten. Diese Zustände müssen angenommen, geschlossen oder noch in Bearbeitung umfassen, um festzustellen, wie viel Patches in einem Projekt abgelehnt werden. Außerdem bedarf es Informationen zum Erstellen und Schließen eines Reviews um die Dauer einer Bearbeitung zu messen. Um festzustellen, wie viel Personen an einem Review beteiligt sind, müssen die Personen jeder Aktion erfasst und gespeichert werden.
Zusammengefasst ergeben sich folgende Abfragen zu den aufbereiteten Daten.

**F1** Abfrage eines Patches, ob dieser direkt oder über ein Review committed wurde;

**F2** Abfrage der Patches, ob dieser angenommen, abgelehnt oder sich noch in einem Review befindet;

**F3** Abfrage über Aktivitäten eines Reviews und wer diese durchführt;

**F4** Abfrage, wann ein Review begonnen und geschlossen wurde;

# 4 Architektur und Design

Im folgenden Kapitel wird die Architektur und das Design der Komponente zur Erfassung von Review-Daten vorgestellt. Dabei muss eine bereits existierende Software, der Patch-Flow Crawler erweitert werden. Der Patch-Flow Crawler ist eine Software zur Erfassung aller Patches eines Projekts. Neben den Patches aus den Versionsverwaltungen erfasst der Crawler auch Daten bezüglich der Autoren, Unternehmensstrukturen und Projektinformationen von projektspezifischen Softwaresystemen. Damit wird das Ziel verfolgt, einen Einblick der Zusammenarbeit innerhalb eines Projekts zu erhalten.
Ein weiterer Aspekt der Zusammenarbeit ist der Review-Prozess. Ziel der Erweiterung ist, zusätzlich Informationen des Review-Prozesses einzubinden.
In den nachfolgenden drei Abschnitten werden die notwendigen Anpassungen des Patch Flow Crawlers aufgezählt. Darunter fällt die Anpassung der Datenbank, der CrawlEngine und des Plugins. Danach wird die Architektur der neuen Komponente erklärt.

## 4.1 Datenbankmodell

Die Erweiterung des Patch-Flow Crawlers benötigt weitere Objekte, die in der Datenbank gespeichert werden. Diese neuen Objekte müssen sinnvoll mit den Objekten des Patch-Flow Crawlers verknüpft werden.
In dem folgenden Abschnitt wird zuerst das Datenbankmodell des Patch-

Flow Crawlers vorgestellt, um die entsprechenden Erweiterungen zu ergänzen.

### 4.1.1 Datenbankmodell des Patch-Flow Crawlers

Wie bereits erwähnt werden neben den Informationen der Patches auch Informationen über deren Autoren, Unternehmensstrukturen und Projektinformationen erfasst. Wie die einzelnen Objekte aufgebaut und miteinander verbunden sind, wird in Abbildung 4.1 verdeutlicht.

#### Patch

Die zentrale Entität ist der *Patch*. Dieser speichert neben den Informationen wann und wie viel verändert wurde, den Ort und wer diesen erstellt hat. Damit können die *Person*, *PatchRepository* sowie die *Component* zugeordnet werden.

#### OrgUnit

Die Personen und Komponenten sind einer sog. *OrgUnit* zugeordnet. OrgUnits sind firmeninterne Organisationseinheiten, wie beispielsweise eine Abteilung aus ein oder mehreren Personen.

#### Person

Das Modell *Person* speichert einen eindeutigen Schlüssel eines Mitarbeiters, den Vor- und Nachnamen sowie dessen E-Mail-Adresse. Für einen Patch ist eine Person ein Commiter oder Autor. Zusätzlich sind Personen einer *OrgUnit* zugeordnet.

#### Component

Eine *Component* stellt für den Patch Flow Crawler eine Einheit dar, die mehrere *PatchRepositories* beinhaltet. So können zu einer Komponente verschiedene Module existieren, die ihrerseits auf verschiedenen Orten liegen.

#### PatchRepository

Ein *PatchRepository* repräsentiert ein Versionskontrollsystem. Ein *Patch* wird genau einem *PatchRepository* zugeordnet.

### 4.1.2 Einführung neuer Objekte in das Datenbankmodell

Abbildung 4.2 stellt das gesamte Datenmodell mit der Erweiterung der Reviewtickets dar. Neben dem Repository für Patches wird ein neues Repository für die Reviewtickets eingeführt, nämlich das *ReviewTicketRepository*. Dieses beinhaltet die Informationen zu dem Ort des Repository. Die gesammelten Reviewtickets werden dann als *ReviewTickets* in der Datenbank gespeichert. Die Tickets wiederum sind mit dem *ReviewTicketRepository* verknüpft und beinhalten eine Liste aus *AbstractAcitivies*. Eine *AbstractActivity* ist eine Aktivität, die innerhalb eines Reviewtickets auftritt. Dieses hat einen eindeutigen Token innerhalb eines *ReviewTicketRepository*, eine eindeutige Id für die Datenbank, den Autor und ein Datum der Erstellung des Tickets. Für den Zweck dieser Arbeit werden sechs verschiedene Aktivitäten unterschieden und nachfolgend vorgestellt.

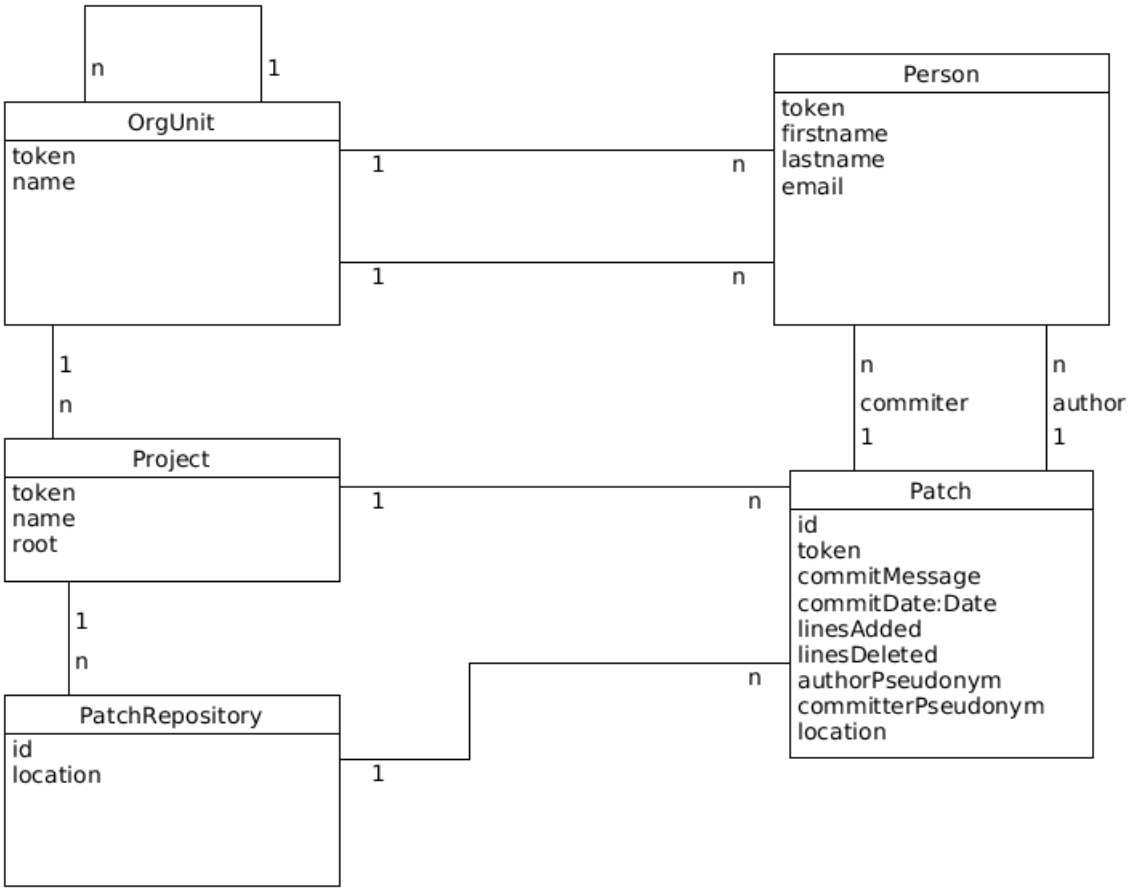


Abbildung 4.1: Vereinfachtes Datenmodell des Patch-Flow Crawlers

### AcceptActivity

Die *AcceptActivity* repräsentiert den Zeitpunkt, wenn ein Reviewticket akzeptiert wurde. Als akzeptiert gilt ein Reviewticket, wenn deren Patches in den Quellcode aufgenommen werden. In Verbindung mit dem Erstellungsdatum des Reviewtickets kann die zeitliche Varianz der Daten erfasst werden. Mit dieser Information kann die Anwendung die Dauer eines erfolgreichen Reviewticket messen.

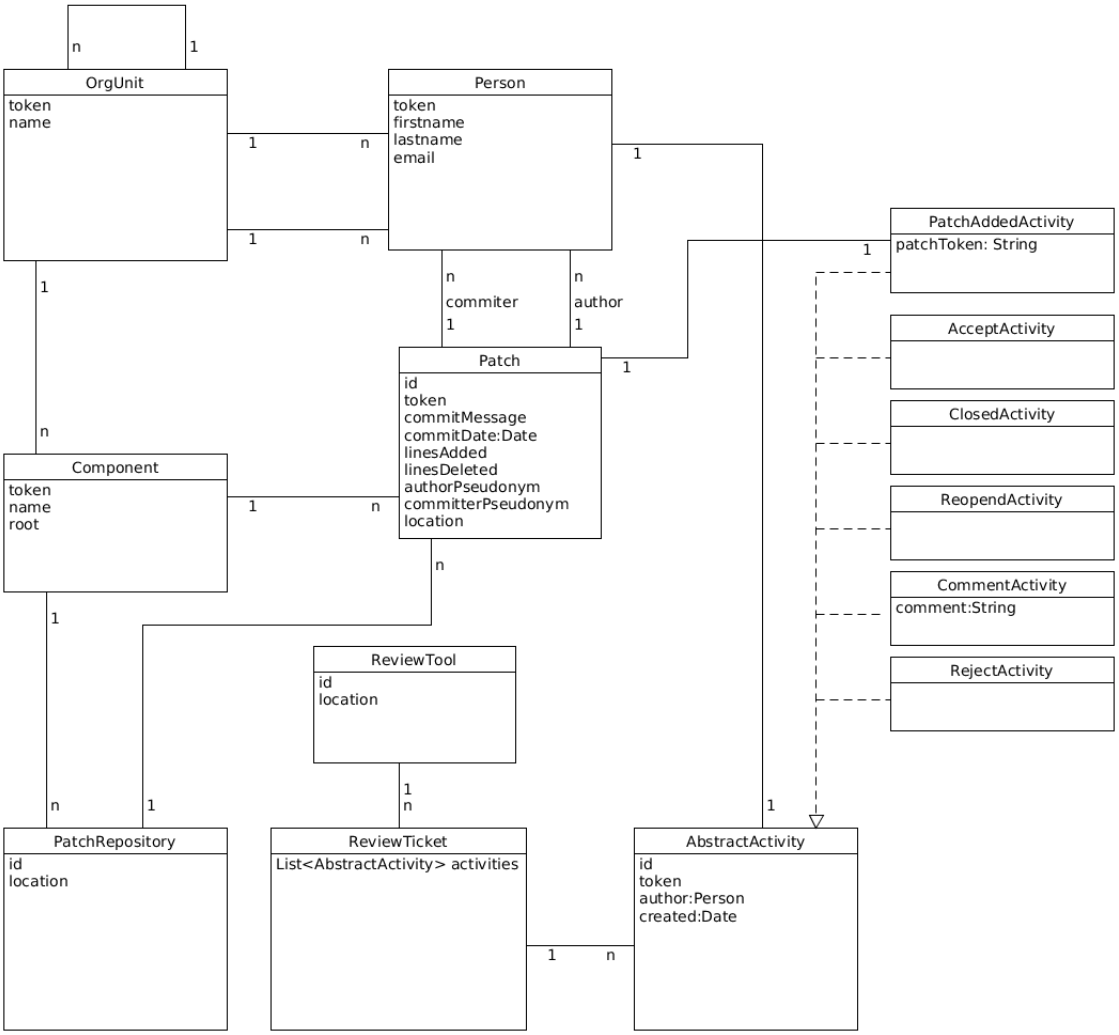


Abbildung 4.2: Erweitertes Datenmodell

### RejectActivity

Wenn ein Ticket nicht angenommen wird, kann es abgelehnt werden. Die *RejectActivity* stellt diesen Fall da. Analog zur *AcceptActivity* ist auch hier die Dauer vom Erstellen eines Tickets und dem erfolglosen Schließen für die Auswertung wichtig.

### CommentActivity

In einem Review werden die Patches auf den Plattformen diskutiert. Für jeden Kommentar, der zu einem Review existiert, wird als *CommentActivity* repräsentiert. Auf manchen Plattformen gibt es die Möglichkeit, entweder direkt zu einem Review oder auf eine bestimmte Zeile Quellcode zu kommentieren. Beide Fälle werden in dieser Anwendung als *CommentAcitivity* behandelt. Grund dafür ist, dass möglicherweise in anderen Systemen beide Varianten nicht unterstützt werden. Das *CommentAcitivty* besitzt ein zusätzliches Datenfeld zum Speichern eines Kommentars.

### ReopenedActivity

Die *ReopenedActivity* wird erstellt, wenn ein Ticket wieder nach einem Schließen des Reviewtickets geöffnet wurde. Mit dieser Activity wird für die Auswertung erkenntlich gemacht, ab wann ein Reviewticket wieder bearbeitet wird.

### PatchAddedActivity

Sobald ein Patch zu einem Reviewticket hinzugefügt wird, symbolisiert dies eine *PatchAddedActivity*. In den hier vorgestellten Reviewplattformen GitHub und GitLab (siehe 4.8) können Reviewtickets nur zu einem Branch erstellt werden. Für den Fall, dass in anderen Systemen einzelne Patches gereviewed werden, wird zu jedem Patch ein *PatchAddedActivity* erstellt. Um die Zuordnung von *Patch* und *PatchAddedActivity* zu gewährleisten, erhält die Activity ein zusätzliches Datenfeld `token` der ein eindeutiger Schlüssel für einen *Patch* darstellt. Außerdem besteht zwischen Patch und Activity eine 1-zu-1 Abhängigkeit.

## 4.2 CrawlEngine

Um die Komponente zur Erfassung von Review-Daten in den Patch-Flow Crawler zu integrieren, müssen diese an das bestehende Architekturkonzept eingefügt werden. Der Patch-Flow Crawler bestimmt in dessen Architektur einen fest definierten Ablauf, der sich in der Klasse *CrawlEngine* widerspiegelt. Zuerst wird der Ablauf der CrawlEngine vorgestellt, um dann den Lösungsansatz für die Erweiterung zu erklären.

### 4.2.1 CrawlEngine des Patch-Flow Crawlers

In der Klasse *CrawlEngine* wird der eigentliche Programmablauf vorbereitet und durchgeführt. In Abbildung 4.3 wird der Ablauf der Bearbeitungsschritte illustriert. Im ersten Schritt werden sogenannte PreSteps ausgeführt. Diese können beliebige Vorbereitungen ausführen. Beispielsweise könnte die Datenbank mit initialen Daten eines Projektes bestückt werden. Im zweiten Schritt werden die PatchCrawlJobs vorbereitet. Die Klasse *PatchCrawlJob* hat die Aufgabe alle Patches aus einem Versionskontrollsystem zu laden und aufzubereiten. Für jedes Repository, das untersucht werden soll, wird genau ein *PatchCrawlJob* erstellt. Im darauf folgenden Schritt werden diese Jobs ausgeführt. Abschließend sorgen PostSteps für beliebige Nachbereitungen.

### 4.2.2 Erweiterung der CrawlEngine

Für das Laden und Bearbeiten der Reviewtickets wird das Konzept der PatchCrawlJobs übernommen. Dazu wird eine neue Klasse, nämlich der *ReviewTicketCrawlJob*, eingeführt. Dieser hat die Aufgabe alle Reviewtickets zu laden und aufzubereiten. Die Aufbereitung umfasst die Umwandlung der gewonnenen Review-Daten in entsprechende Objekte der Anwendung und die Verknüpfung der Reviews mit den Patches. Da

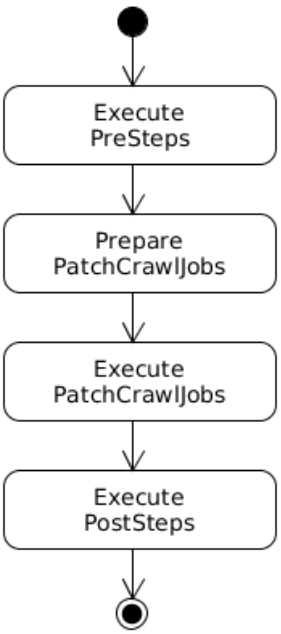


Abbildung 4.3: Bearbeitungsschritte der CrawlEngine als Aktivitätsdiagramm

die Reviewtickets mit den Patches verknüpft werden, müssen die Patches bereits bekannt sein. Aus diesem Grund muss die Bearbeitung der Reviewtickets nach dem Erfassen aller Patches erfolgen. In Abbildung 4.4 wird der neue Ablauf der CrawlEngine dargestellt. Die Vorbereitung und Ausführung der ReviewTicketCrawlJobs wird nach der Ausführung der PatchCrawlJobs eingefügt.

## 4.3 Plugin

Der Patch-Flow Crawler besitzt einen Mechanismus um für verschiedene Systeme eingesetzt zu werden. Während die CrawlEngine den Programmablauf bestimmt, wird mit einem Plugin die konkreten Ausführungsschritte übergeben. Da in Abschnitt 4.2.2 die ReviewTicketCrawlJobs in

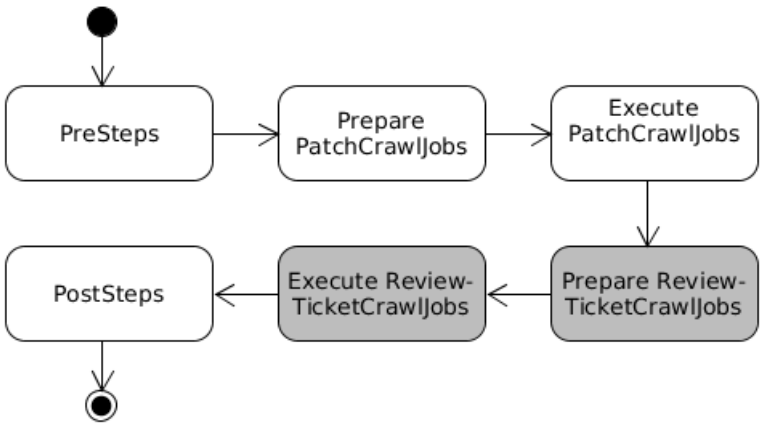


Abbildung 4.4: Bearbeitungsschritte der erweiterten CrawlEngine als Aktivitätsdiagramm

die CrawlEngine eingefügt wurden, muss auch das Plugin entsprechend erweitert werden.

### 4.3.1 Das Interface Plugin

Abbildung 4.5 zeigt das Interface Plugin. Dieses enthält eine Methode `getPatchProcessors()`. Mit dieser wird eine Liste aus Objekten von dem Typ *PatchProcessor* übergeben. PatchProcessoren sind Bearbeitungsstrategien für einen PatchCrawlJob. In diesen werden die Patches bearbeitet, nachdem diese erfasst wurden. Die Methode `getScmAdapter(Repository)` muss eine Klasse *Repository* übergeben werden und liefert einen Adapter für das Versionskontrollsystem zurück. Dieser Adapter wird auch dem PatchCrawlJob übergeben, damit dieser sich die Patches aus dem jeweiligen Versionskontrollsystem laden kann.

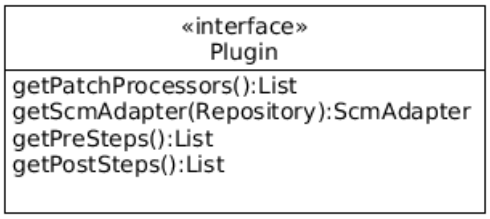


Abbildung 4.5: Klassendiagramm des Plugin

### 4.3.2 Erweiterung des Plugins

Die Anpassung des Plugins wird in Abbildung 4.6 dargestellt. Hier befinden sich zwei neue Methoden. Die Idee der PatchProcessoren wird von dem Patch-Flow Crawler übernommen und für die ReviewTicketCrawlJobs eingesetzt. Hierfür wird eine neue Klasse ReviewTicketProcessor eingeführt, der die Bearbeitung eines Reviewtickets übernimmt. In der Methode `getReviewTicketProcessors()`, werden diese Prozessoren für die Reviewtickets der CrawlEngine übergeben. Diese übergibt allen ReviewTicketCrawlJobs die Prozessoren. Außerdem wird analog zu dem ScmAdapter eine Klasse *ReviewToolAdapter* implementiert. Der *ReviewToolAdapter* ist ein Interface und hat die Aufgabe für die ReviewTicketCrawlJobs, die ReviewTickets zu laden.

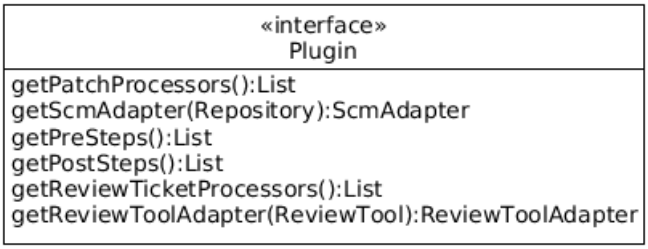


Abbildung 4.6: Klassendiagramm des erweiterten Plugin

### 4.3.3 Zusammenspiel von CrawlEngine und Plugin

Der Patch-Flow Crawler gibt durch die CrawlEngine einen festen Ablauf vor. In diesem muss die Komponente zum Erfassung von ReviewTickets eingebaut werden. Dabei ist es wichtig, die Bearbeitung der Reviewtickets nach der Bearbeitung der Patches einzufügen, da die Reviewtickets auf die Patches referenzieren und diese bekannt sein müssen. Um sich an die Architektur des Patch-Flow Crawlers anzupassen, wurde außerdem das Plugin um die zwei neuen Methoden erweitert. In Abbildung 4.7 wird das Zusammenspiel von CrawlEngine und Plugin beschrieben. Die CrawlEngine kennt das Interface *Plugin*. Um die Bearbeitung für ein bestimmtes Projekt zu realisieren, werden mithilfe eines konkreten Plugins die entsprechenden Objekte für die jeweils eingesetzten Systeme übergeben. *ProjektAPlugin* liefert demnach alle Objekte zur Bearbeitung für die Systeme, welche in ProjektA, und *ProjektBPlugin* für die Systeme in ProjektB eingesetzt werden.

## 4.4 ReviewToolAdapter

Der *ReviewToolAdapter* hat die Aufgabe alle *ReviewTickets* eines *ReviewTicketRepository* zu erstellen. Das Klassendiagramm in Abbildung 4.8 zeigt den Aufbau des Adapters mit seinen Abhängigkeiten. Der Adapter ist selbst ein Interface und erfordert einer Methode `fetch(ReviewTicketRepository, Date, Date)`. Darunter befindet sich die Klasse *AbstractReviewTicketAdapter*. In dieser wird eine Hilfemethode `getReviewTickets(ReviewTicketRepository, Date, Date)` bereitgestellt, die von der fetch-Methode genutzt werden kann. Diese sorgt für die korrekte Sortierung der *ReviewTickets*. Die fetch-Methode selbst liefert einen Iterator mit ReviewTickets zurück. Wie dieser Iterator aufgebaut ist und welchen Zweck dieser hat, wird in Abschnitt 4.8 genauer erklärt.

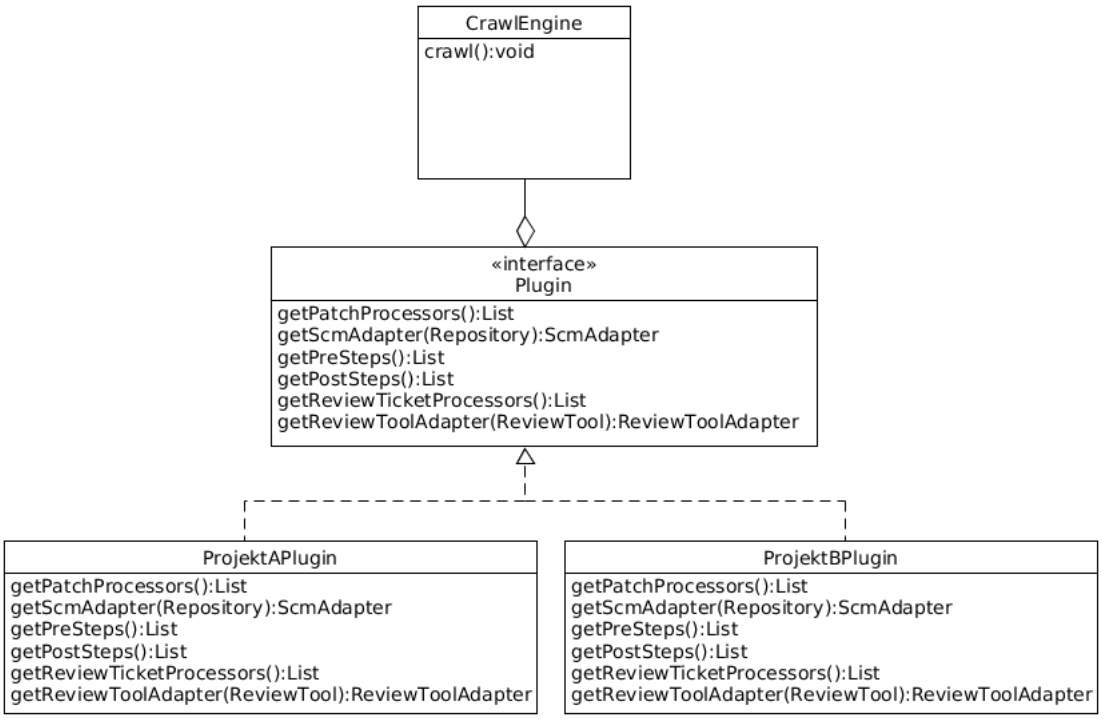


Abbildung 4.7: Klassendiagramm der Erweiterung des Patch-Flow Crawlers

### 4.4.1 ReviewToolFacade

Um mit einem konkreten Repository zu kommunizieren, benutzt der Adapter ein Interface *ReviewToolFacade*. Abbildung 4.9 zeigt das Interface einer ReviewToolFacade. Diese beinhaltet die Methoden zum Laden aller Reviewtickets für einen bestimmten Zeitraum und Methoden zum Laden aller Activities für jeweils ein Reviewticket. Für die Implementierung der einzelnen Repositories müssen daher entsprechende Facaden und konkrete *ReviewTicketAdapter* entwickelt werden. In Abbildung 4.8 werden beispielhaft jeweils ein Adapter und Facade für GitHub und GitLab dargestellt.

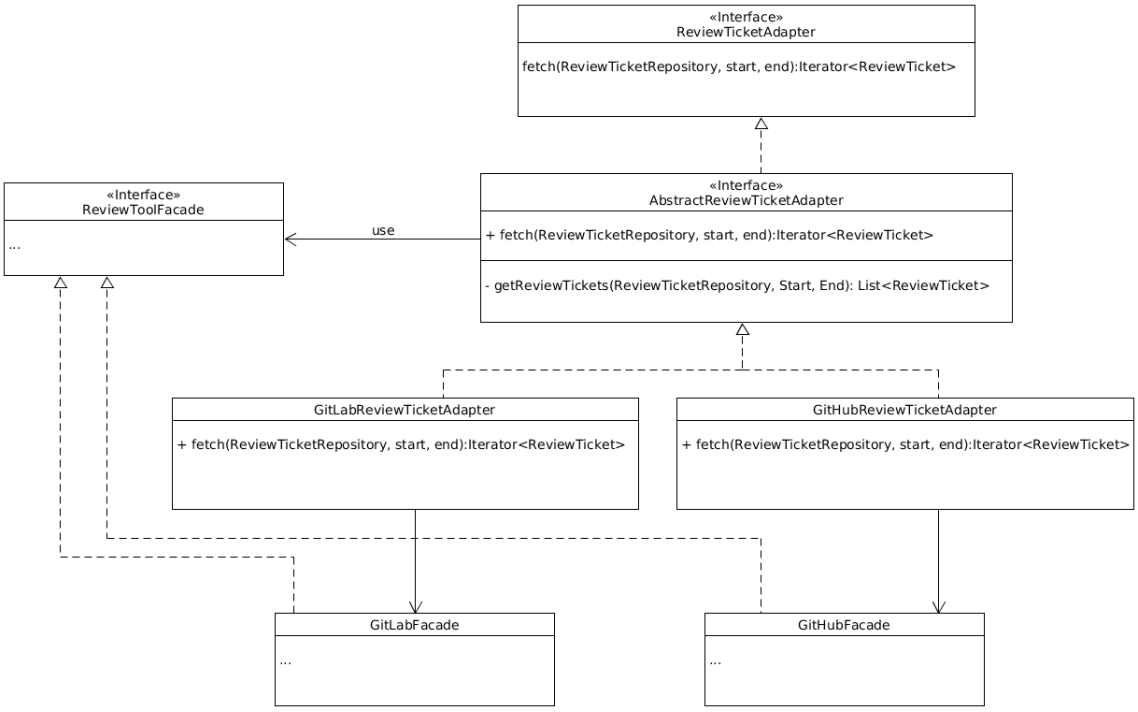


Abbildung 4.8: Klassendiagramm des ReviewTicketAdapter

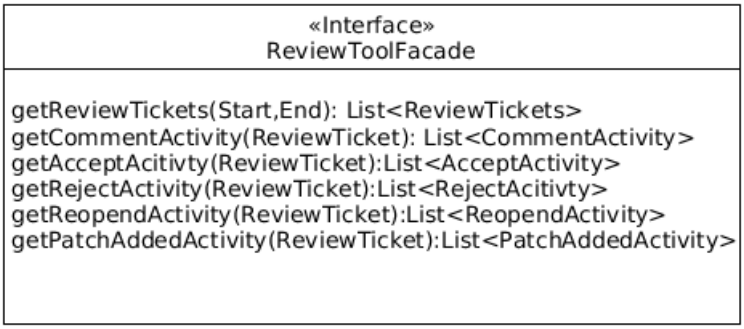


Abbildung 4.9: Klassendiagramm des ReviewTicketAdapter

### 4.4.2 Iterator

Die fetch-Methode liefert einen Iterator mit Reviewtickets zurück. Die übergebenen Reviewtickets sind jedoch noch nicht vollständig zusam-

mengebaut. Erst wenn der Iterator ein Reviewticket übergeben muss, werden in diesem alle Activities für das jeweilige Reviewticket geladen. Aus diesem Grund wird beim Erstellen des Iterators in dem Adapter auch die ReviewToolFacade übergeben.
Hintergrund dieser Designentscheidung war, dass nicht gleichzeitig zu viele Anfragen an ein Review-Werkzeug gesendet werden. Für jedes Reviewticket müssen möglicherweise mehrere Abfragen versendet werden, um alle Activities zu erhalten. Bei einer sehr großen Anzahl von Reviewtickets würde dieser Prozess sehr lange dauern und müsste beim Fehlerfall komplett neu gestartet werden. Mit der Möglichkeit nur bei Bedarf weitere Abfragen an das Review-Werkzeug zu senden, kann die Software sequenziell ein Reviewticket zusammenbauen, bearbeiten und sichern bevor es weitere Anfragen an das Review-Werkzeug tätigt.
Abbildung 4.10 zeigt das Zusammenstellen der Reviewtickets in dem ReviewToolAdapter und ReviewticketIterator.
Zuerst fordert der ReviewToolAdapter alle Reviewtickets von der ReviewToolFacade an. Diese werden sortiert und dem ReviewticketIterator übergeben. Sobald der Iterator ein Reviewticket übergeben muss, werden mithilfe der ReviewToolFacade die einzelnen Activities für jeweils ein Reviewticket angefordert und diese abhängig des Erstelldatums sortiert.

## 4.5 CrawlRun

Bei mehrmaliger Anwendung oder im Falle eines Abbruch der Software muss bestimmt werden, welches Reviewticket als letztes bearbeitet wurde. Hierfür wird ein Konzept des Patch-Flow Crawlers übernommen.

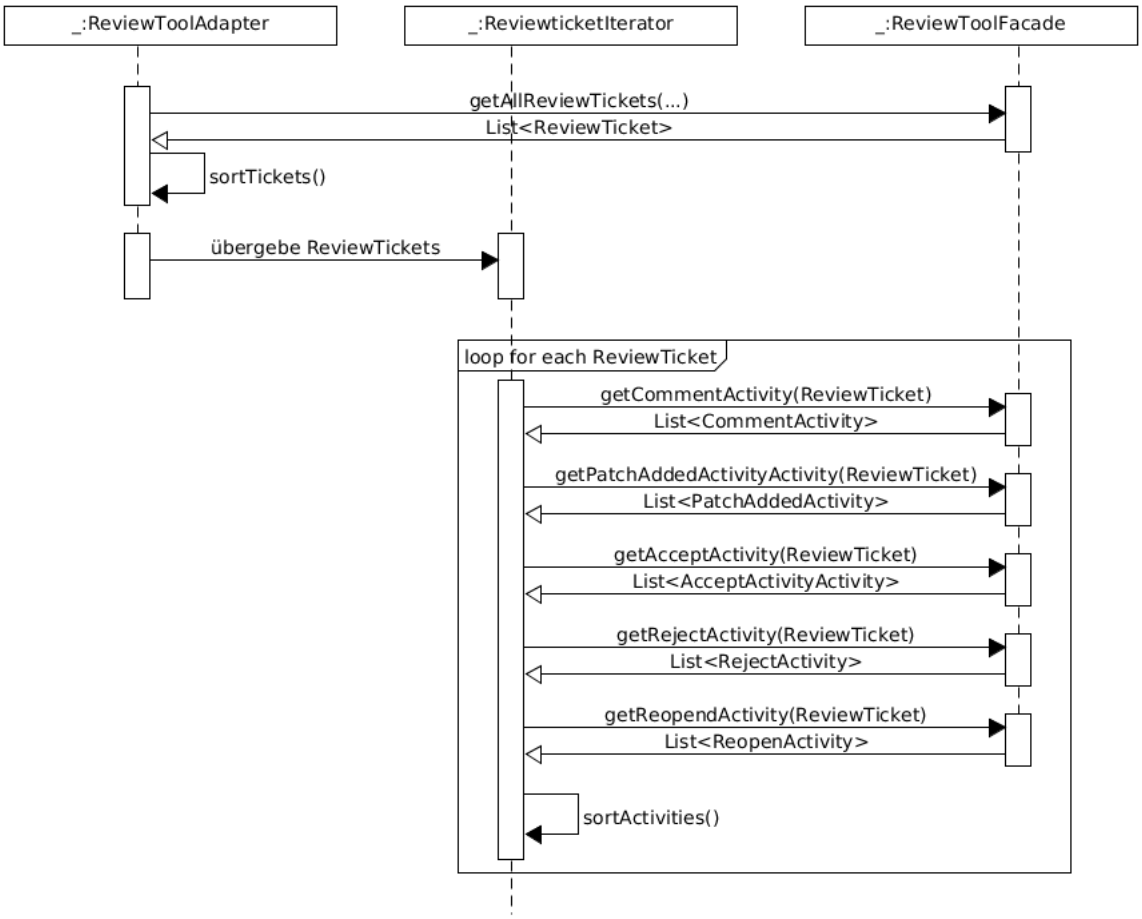


Abbildung 4.10: Sequenzdiagramm von der Zusammenstellung der Reviewtickets

### 4.5.1 Konzept der CrawlRuns

In Fällen eines Abbruchs merkt sich der Patch-Flow Crawler, welchen Patch er als letztes bearbeitet hat. Dafür ist das Objekt *CrawlRun* zuständig. Dieser besteht aus einem Timestamp und einem Status. Der *CrawlRun* ist mit dem *PatchRepository* und dem *Patch* verbunden. (siehe Abbildung 4.11)
Vor Beginn jeder Anwendung wird ein neues Objekt *CrawlRun* erstellt

und der Zustand `INITIALIZED` vergeben. Beim erfolgreichen Bearbeiten der PatchRepositories wird dieser Status auf `CRAWLED` geändert. Zusätzlich wird nach jedem erfolgreichem Bearbeiten eines Patches diesem den aktuellen *CrawlRun* zugeordnet und der *Patch* dem Objekt *PatchRepository* als `youngestPatch` verknüpft. Mit diesen Informationen ist die Anwendung in der Lage selbständig zu bestimmen, welche Patches bereits bearbeitet wurden.

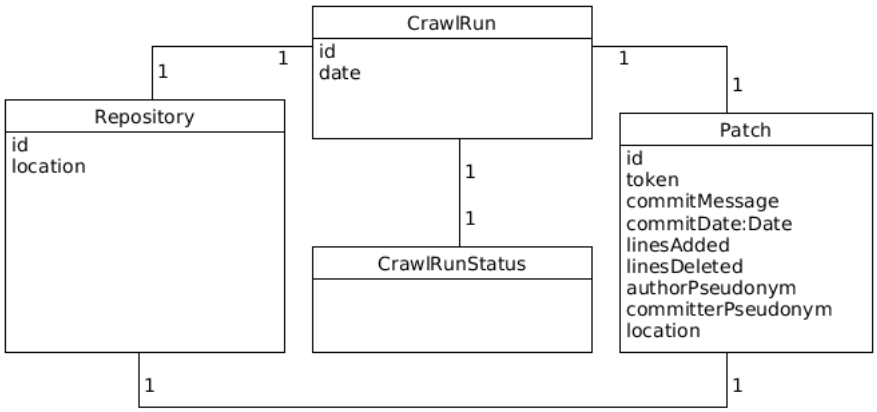


Abbildung 4.11: Vereinfachtes Datenmodell mit CrawlRun

### 4.5.2 CrawlRun für die Reviewtickets

Wie bereits erläutert existiert ein zusätzliches Objekt mit dem Namen *CrawlRun*. Dieser beinhaltet einen Timestamp und kann zwei verschiedene Zustände einnehmen, nämlich `INITIALIZED` und `CRAWLED`. Die Idee des CrawlRun wird analog zur Vorgehensweise der Patches genutzt. Nach einem erfolgreichen oder erfolglosen Beenden der Anwendung muss die Software selbstständig ermitteln, welches Reviewticket sie als letztes er-

folgreich bearbeitet hat. Die daraus gewonnene Erkenntnis führt beim nächsten Durchlauf zu einer geeigneten Startzeit.
Zum Programmstart wird ein *CrawlRun* mit dem Status `INITIALIZED` erstellt, bevor dieser die Reviewtickets bearbeitet. Danach wird der nächste Startzeitpunkt ermittelt, ab wann die nächsten Reviewtickets berücksichtig werden müssen. Wie dies geschieht wird in Abschnitt 4.6 näher erklärt. Anhand dieser Informationen wird eine Liste bestehend aus *ReviewTickets* geladen und einzeln bearbeitet. Das zu bearbeitete Reviewticket bekommt den aktuellen CrawlRun angehängt und wird dann mithilfe der *ReviewTicketProcessoren* bearbeitet. Tritt in dieser Phase ein Fehler auf, bricht der komplette CrawlRun ab. War die Bearbeitung erfolgreich, wird das bearbeitete *ReviewTicket* dem *ReviewTicketRepository* als *youngestReviewTicket* angehängt. Für den nächsten CrawlRun ist damit ersichtlich, dass dieses Ticket bereits erfasst wurde. Nachdem alle Reviewtickets erfolgreich bearbeitet wurden, wird dem *ReviewTicketRepository* abschließend der aktuelle CrawlRun übergeben, als *lastCompleteCrawlRun* gespeichert und der Status des CrawlRuns zu `CRAWLED` geändert.

## 4.6 Ermittlung der Start und Endzeit

In der CrawlEngine kann eine Start- und Endzeit angegeben werden. In diesem Zeitraum werden alle Patches die in dieser Zeit erstellt wurden bearbeitet. Da die CrawlEngine hinsichtlich der Reviewtickets erweitert wurde, ist es notwendig, diese Möglichkeit auch für Reviewtickets bereitzustellen.
Die Start- und Endzeit in der fetch-Methode des Interfaces *ReviewTicketAdapter* (siehe Abbildung 4.8) sorgen dafür, dass nur in diesem Zeitraum alle Reviewtickets geladen werden. Beide Zeiten können manuell angegeben werden bzw. werden abhängig des *CrawlRun*-Objekt ermittelt. Anders als bei den Patches muss bei den Activities mehr als nur die

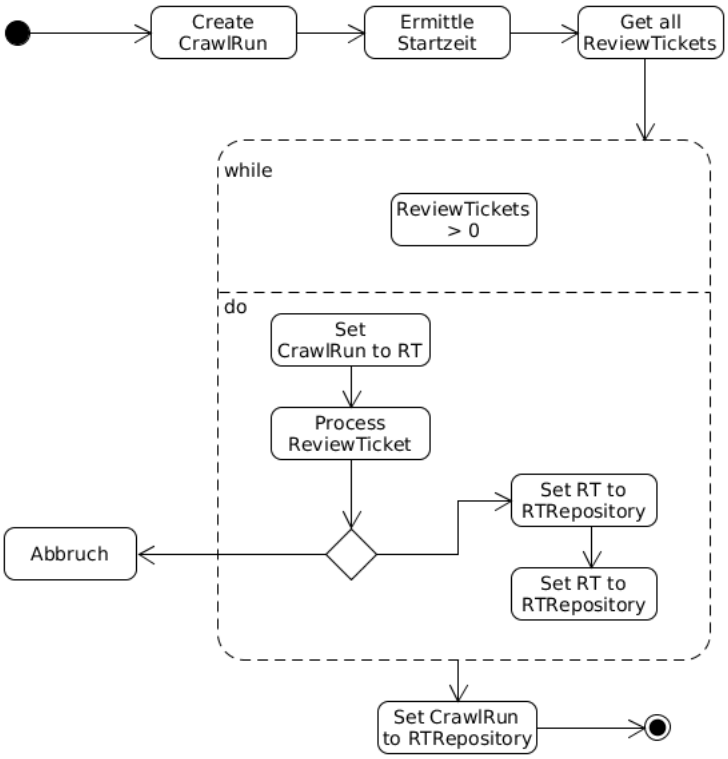


Abbildung 4.12: Ablauf der Bearbeitung eines CrawlRuns als Aktivitätsdiagramm

Startzeit berücksichtigt werden. Um konsistente Daten zu erhalten, wird jedes Reviewticket überprüft, wann es zum letzten Mal verändert wurde. Abbildung 4.13 illustriert die Problematik bei Reviewtickets.
In einem ersten CrawlRun mit der Startzeit t0 bis t1 wurden die Tickets 1-3 erfasst und in der Datenbank gespeichert. In diesem Durchlauf ist Ticket 1 als gemerged und Ticket 3 als geschlossen erfasst worden. Ticket 2 war zu diesem Zeitpunkt noch in Bearbeitung. In einem zweiten Crawlrun wurde t1 als neue Startzeit ermittelt und speichert Ticket 4 in die Datenbank. rcaWährend des ersten und zweiten CrawlRun wurde Ticket 2 beispielsweise geschlossen. Diese Informationen ist zwischen den beiden CrawlRuns nicht erfasst. Hier überprüft der Adapter auch alle vergange-

nen Reviewtickets ob diese nach der Zeit t0 verändert wurde und hängt die neue Änderung als Activity an dem gespeicherten Reviewticket hinzu.

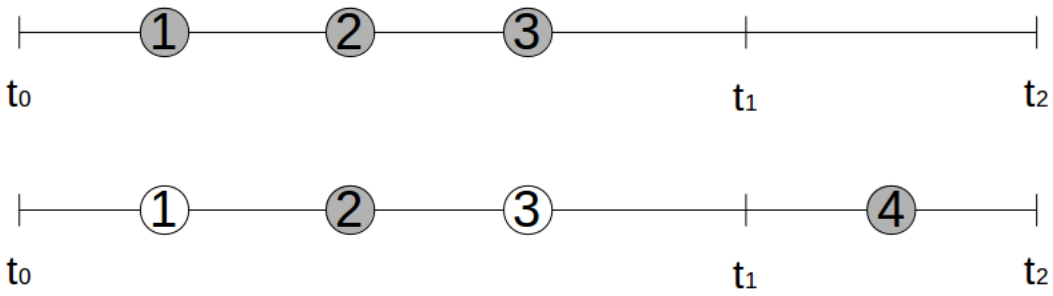


Abbildung 4.13: Zeitleiste mit Reviewtickets, die bearbeitet werden.

### 4.6.1 Startzeit ermitteln

Bevor die Reviewtickets mithilfe der Facade abgefragt werden, ermittelt die Anwendung anhand der früheren CrawlRuns selbständig die Startzeit. In dem *ReviewTicketRepository* wird das letzte erfolgreiche *ReviewTicket* als `youngestReviewTicket` und der letzte erfolgreiche *CrawlRun* als `lastCompleteCrawlRun` gespeichert. Sind beide in dem aktuellen *ReviewTicketRepository* nicht angegeben, wurde noch kein CrawlRun durchgeführt und die Startzeit ist das kleinste Datum, dass die Java-Date Bibliothek anbieten kann. Ist ein *ReviewTicket* im Repository angegeben, wird mithilfe des angegeben *CrawlRun* die neue Startzeit erstellt. Sollte im Repository das Feld `lastCompleteCrawlRun` angegeben sein, muss zusätzlich überprüft werden, ob der *CrawlRun* älter ist als der *CrawlRun* des *ReviewTickets*. Falls das der Fall ist, wird die Zeit des letzten

CrawlRuns übernommen, falls nicht, der Zeitpunkt des Tickets.

## 4.7 Exception Handling

Für das Exception Handling stehen zwei Exceptions bereit, nämlich die *TemporaryReviewTicketAdapterException* und die *ReviewTicketAdapter-Exception*. Beide leiten sich von der *RuntimeException* ab und brechen den Programmablauf sofort ab. Erstere wird geworfen, wenn es sich um eine zeitlich bedingte Ausnahme handelt, die sich durch ein wiederholtes Ausführen der Anwendung von selbst erledigt. Die andere stellt eine Ausnahme dar, die ohne dem Eingreifen eines Entwicklers nicht behoben werden kann.

### 4.7.1 Problematik vieler PullRequests

PullRequests sind Reviewtickets für die Plattform GitHub. Bevor ein Entwickler (sog. Contributer) seinen Beitrag in das Projekt integrieren kann, muss er einen Antrag, einen PullRequest erstellen. In Abschnitt 4.6 wird erklärt, warum nach einem zweiten Start der Anwendung nicht das letzte bearbeitete Reviewticket sondern ein Abgleich aller Reviews erfolgen muss. Bei einem Review wird untersucht, wann dieses zum letzten mal verändert wurde. GitHub bietet hierfür das attribute last-modified an. Dadurch und in Verbindung mit dem Rate Limit (GitHub) von 5000 Requests pro Stunde, ergab sich eine Problemsituation. Bei einem Projekt mit einer hohen Anzahl an PullRequests bricht die Software wegen dem Rate Limit ab und musste erneut angestoßen werden. Bei einem erneuten Start bearbeitete er die gleiche Menge erneut, weil alle Reviews zu einem späteren Zeitpunkt mit einem Label versehen wurden. Das Hinzufügen eines Labels an ein Review verursacht eine Aktualisierung des Attributes last-modiified. In solchen speziellen Fällen muss der Entwickler eingreifen

und die Software kann nicht selbstständig darauf reagieren.

## 4.8 GitHub und GitLab Client

Um direkt mit den Reviewticket-Repositories zu kommunizieren, benötigen die Facaden Zugang zu einem Client. Für diese Arbeit wurden schwerpunktmäßig ein GitHub- und GitLab-Client entwickelt. Die beiden Systeme GitHub und GitLab bieten eine REST-API an, mit deren Hilfe auf die Daten der beiden Systeme zugegriffen werden kann. Die Spezifikation dieser APIs sind unter Documentation (b) für GitLab und Documentation (a) für GitHub zu finden.

### 4.8.1 Library für Clients

Für beide Systeme werden Java-Clients der jeweiligen APIs angeboten. GitHub zählt drei Clients auf dessen Homepage auf (Libraries (a)). Für GitLab wurden zwei Java-Clients auf GitHub gefunden (Libraries (b)). Alle Clients würden die Erfassung der Reviewtickets unterstützen und basieren auf den aktuellen Versionen der APIs. Allerdings können keine Reviewtickets abhängig von der Zeit abgefragt werden, sodass entweder die Client geforked und umgeschrieben oder eine zeitliche Selektierung in der Facade implementiert werden müsste. In dieser Überlegung wurde beschlossen beide Clients selbst zu entwickeln.

### 4.8.2 Architektur der Clients

Im Folgenden werden für beide Clients die Architektur anhand von GitHub erklärt. Bei Unterscheidungen wird separat auf diese eingegangen. Für den Anwender wird ein *GHService* bereitgestellt der aus den thematisch unterteilten Bereichen der API Services bereitstellt.

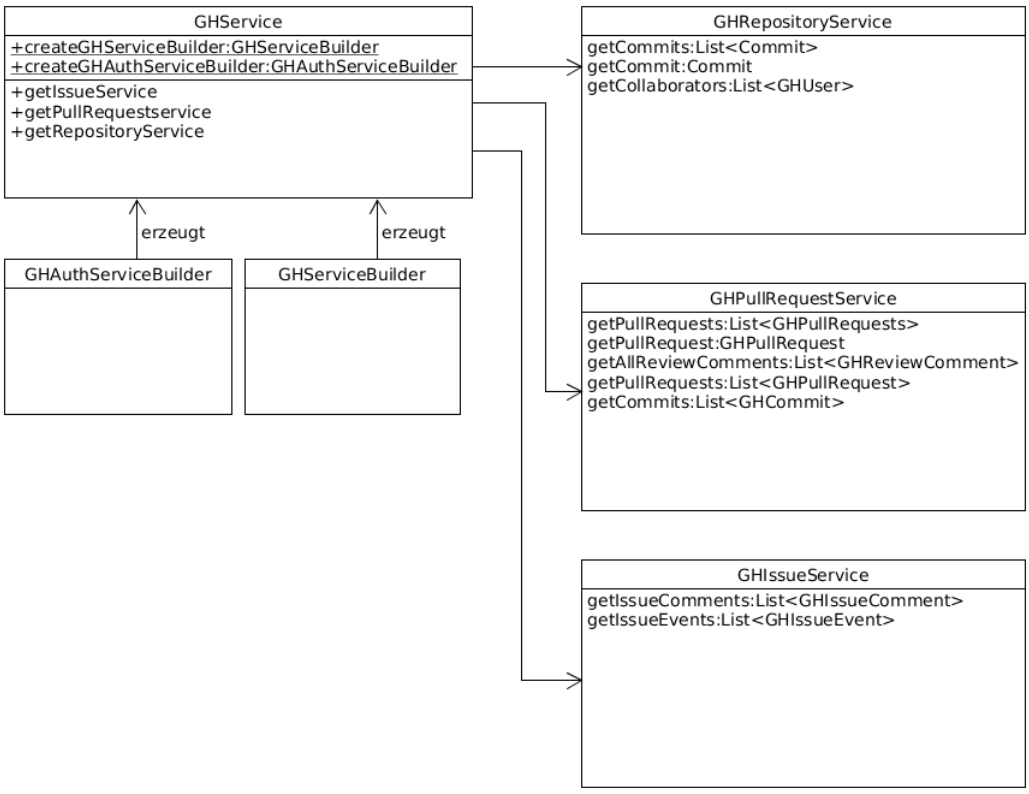


Abbildung 4.14: Klassendiagramm: Aufbau von GHService, Builder und Services

### 4.8.3 Requests

Die Ausführung eines Requests wird im GitHub- als auch im GitLab-Client mit dem Command-Pattern umgesetzt und das Zusammensetzen der Parameter in ein Request-Objekt gekapselt. Wie in Abbildung 4.15 zu sehen, wird zunächst ein abstrakter Service mit eine execute()-Methode bereitgestellt. Dieser Methode muss ein Objekt vom Typ *GHRequest* übergeben werden. Innerhalb dieser execute- Methode wird der *GHRequest* ausgeführt und ein allgemeines Exception Handling angeboten. Wird der *GHPullRequestService* betrachtet, so müssen für alle HTTP-Requests eine eigene Klasse vom Typ *GHRequest* implementiert werden.

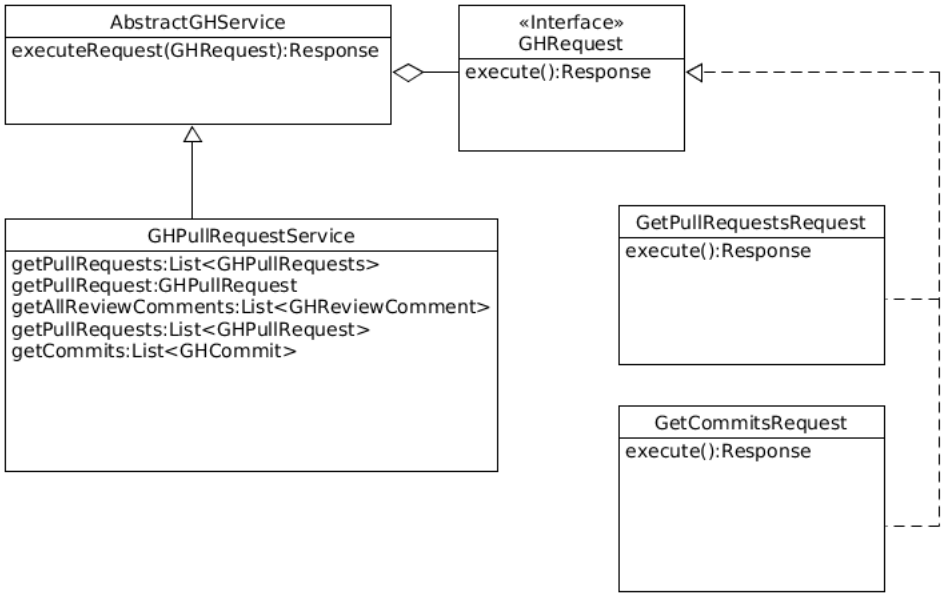


Abbildung 4.15: Klassendiagramm von dem Aufbau des GitHub-Clients

# 5 Implementierung

Nachdem im letzten Kapitel die Architektur und das Design des erweiterten Patch-Flow Crawlers betrachtet wurde, wird in diesem die Implementierung besprochen. Hierbei werden besondere Aspekte der Implementierung präsentiert. Zuerst wird erklärt, wie die Daten-Objekte, dann wie die Umsetzung der Review Tool Clients umgesetzt wurden. Abschließend wird die Rolle der Fassaden hervorgehoben.

## 5.1 Value-Objects

Bei der Implementierung der Datenobjekte in Abbildung 4.2, aber auch für die Objekte der GitHub und GitLab-Clients, wurde das Pattern Value-Objects angewendet. Damit wird eine Designentscheidung des Patch-Flow Crawlers fortgeführt. Dieser hatte bereits dessen Daten-Objekte in dieser Art implementiert. Ziel des Pattern ist, Seiteneffekte durch Unveränderbarkeit der Zustände eines Objektes zu verringern. (Riehle (2006)). Die Umsetzung dieser Unveränderbarkeit wird dadurch sichergestellt, dass alle Attribute als private markiert werden und Setter immer ein neues Objekt erzeugen. Die Anwendung des Value-Object wird anhand des *ReviewTicket* demonstriert (Siehe Listing 5.1.1). In Zeile 3 ist die Sichtbarkeit der Attribute sowie die Konstruktoren (Zeile 11-18) als private gesetzt. Den Zugang zu den Attributen erhält man ausschließlich durch die Setter und Getter (siehe Zeile 22- 34). In der Methode `setID` wird eine neue ID gesetzt und dabei eine neue Instanz von

dem *ReviewTicket* erstellt. Um die Erzeugung des veränderten Objekts zu vereinfachen, wird die alte Instanz des *ReviewTicket* einem Copy-Konstruktor (siehe Zeile 15-18) übergeben. Dieser erstellt eine neue Instanz von sich selbst und übernimmt alle Werte aus dem übergebenen Objekt. Das Ergebnis ist ein aktualisiertes *ReviewTicket*, welches durch den Setter zurückgegeben wird.

```
1  public class ReviewTicket{
2
3      private long id;
4
5      ...
6
7      private ReviewTicket(){
8          activityList = new LinkedList<>();
9      }
10
11     private ReviewTicket(ReviewTicket _reviewTicket){
12         id= _reviewTicket.id;
13         ...
14     }
15
16     public ReviewTicket setId(long _id) {
17         ReviewTicket reviewTicket = new ReviewTicket(this);
18         reviewTicket.id = _id;
19         return reviewTicket;
20     }
21
22     public long getId() {
23         return id;
24     }
25
26     ...
27 }
```

Listing 5.1.1: ReviewTicket als Value-Object umgesetzt.

## 5.2 Implementierung der Clients

In Abschnitt 4.8 wurde die Architektur der GitHub- und GitLabs-Client vorgestellt. In diesem Abschnitt geht es um die konkrete Umsetzung dieser Architektur.

#### Retrofit

Für die nachfolgenden Clients wird Retrofit in der Version 2 verwendet. Retrofit ist eine Java-Bibliothek zur Entwicklung von REST-Clients. In der Anwendung generiert die Bibliothek aus Methoden mithilfe von Annotations REST-Aufrufe. In Listing 5.2.1 erzeugt Retrofit aus der Methode `getIssueComments` *GHIssueClientApi* einen REST-Aufruf. Durch die Annotation `@GET` in Zeile 2 kann Retrofit aus dem Methodenaufruf, dem Pfad und die übergebenen Parameter einen HTTP-GET-Befehl mit einer URL zu einer Ressource umwandeln. Nachdem das Interface definiert ist, kann ein Service mithilfe Retrofit erstellt werden (siehe Abbildung 5.2.2). Hierzu wird ein Builder erzeugt, der im einfachsten Fall nur eine Base-URL benötigt. Dieser erstellt in Zeile 5 einen vollständigen REST-Client, der als einfache Klasse im Programmcode verwendet werden kann. Bei dem Aufruf der Methode `getIssueComments` mit den Parametern *owner*, *repo* und *34*, wird ein GET-Request mit der URL `https://api.github.com/repos/owner/repo/issues/34/comments` aufgerufen und der Response als JSON in eine Liste aus *GHIssueComments* umgewandelt. Diese werden in einem Call-Objekt zurückgegeben.

### 5.2.1 Serviceklassen

In Abschnitt 4.8.2 wird erklärt, dass für den Anwender eine Service-Klasse zur Verfügung steht. Der *GHService* verhindert den Aufruf auf dessen Konstruktoren und kann nur mit einem Builder initialisiert werden. Es stehen zwei Builder zur Verfügung. Der eine ist der *GHServiceBuilder*

```java
public interface GHIssueClientApi {
    @GET("/repos/{owner}/{repo}/issues/{number}/comments")
    Call<List<GHIssueComment>> getIssueComments(
            @Path("owner") String repositoryOwner,
            @Path("repo") String repository,
            @Path("number") String pullRequestnumber);
}
```

Listing 5.2.1: Auszug des GHIssueClientApi

```java
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .build();

GHIssueClientApi service = retrofit.create(GHIssueClientApi.class);
```

Listing 5.2.2: Erzeugung eines Service durch Retrofit

und definiert den Standard-Builder. Der zweite ist der *GHAuthenticationServiceBuilder* und verlangt neben der baseUrl, Nutzername und Passwort für die Authentifizierung an der API. Ein Aufruf der API würde folgendermaßen aussehen. Zunächst muss mithilfe des richtigen Builders der Service erstellt werden. In Listing 5.2.3 wird ein AuthenticationBuilder aufgerufen, der Host, User und Password übergeben bekommt. Dieser erstellt den Service. Über diesen kann auf den *GHPullRequestService* zugegriffen und eine seiner Methoden aufgerufen werden.

```java
      GHService service = GHService.createGHServiceBasicAuthorizationBuilder()
               .toHost("Host")
               .withUsername("User")
               .withPassword("Password")
               .build();
       pullRequestService = service.getPullRequestService();
       pullRequestService.getPullRequest("RepositoryOwner", "Repository", "1");
```

Listing 5.2.3: Erzeugung und Aufruf eines Services

### 5.2.2 Requests

Wie bereits in Abschnitt 4.8.3 erwähnt, wird die Ausführung eines Requests in einem Command-Pattern gekapselt. Damit diese Command-Objekte deren Requests ausführen können, muss ihnen einmal ein entsprechender Request-Client und die notwendigen Parameter über den Konstruktur übergeben werden. Listing 5.2.4 zeigt exemplarisch einen konkreten Request. Über den Konstruktor wird der entsprechende Retrofit-Client mit den Parametern des Requests übergeben, sodass dieser in der execute-Methode ausgeführt werden kann. In Listing 5.2.5 wird die Initialisierung und Ausführung des Requests in einem Service gezeigt. In Zeile 5-6 wird der Request mit seinen Parametern erstellt und in der darauf folgenden Zeile mithilfe der allgemeinen executeRequest-Methode ausgeführt. Der Response, als Ergebnis dieser Methode wird dann noch weiter entpackt und zu einer Liste aus GHPullRequests gecasted.

### 5.2.3 Authentifizierung

**Github** bietet drei Möglichkeiten für die Authentifizierung an. Für den GitHub-Client wird nur eine dieser drei Varianten implementiert und durch den *GHAuthenticationServiceBuilder* konfiguriert. Hierzu muss man der Retrofit-Bibliothek einen HTTP-Client übergeben, der Benutzername und Password verschlüsselt in dem Header platziert.

**GitLab** bietet auch drei Varianten der Authentifizierung an wobei auch nur eine implementiert wird. Für den GitLab-Client wird ein privater Token erstellt, der dann bei jedem Request mitgeliefert wird. Hierfür wird der Retrofit-Konfiguration ein HTTP-Client mit dem entsprechenden Header-Eintrag für den privaten Token mitgeliefert.

```java
public class GetPullRequestsRequest implements GHRequest {

    private GHPullRequestClientApi pullRequestClientApi;
    private String repositoryOwner;
    private String repository;

    public GetPullRequestsRequest(GHPullRequestClientApi _pullRequestClientApi,
                String _repositoryOwner,
                String _repository){
        pullRequestClientApi = _pullRequestClientApi;
        repositoryOwner =  _repositoryOwner;
        repository = _repository;
    }


    @Override
    public Response execute() throws GHException {
        try {
            return pullRequestClientApi.getAllPullRequests(repositoryOwner, repository)
                .execute();
        } catch (IOException e) {
            throw new GHException(e);
        }
    }
}
```

Listing 5.2.4: Ausführung eines Request-Objekt

```java
public class GHPullRequestService extends AbstractGHService {

    public List<GHPullRequest> getPullRequests(String _repositoryOwner, String _repository)
        throws GHException {
        GetPullRequestsRequest request =
        new GetPullRequestsRequest(pullRequestClient,_repositoryOwner,_repository);
        return (List<GHPullRequest>) executeRequest(request).body();
    }

}
```

Listing 5.2.5: Aufruf eines Request-Objekts

### 5.2.4 Exception Handling

Bei der Ausführung eines Request-Objekts wird ein Response-Objekt von Retrofit zurückgegeben. Dieses Objekt muss auf den erwarteten Typ gecasted werden. Falls der Response allerdings einen Fehlercode zurückliefert, ist das nicht möglich und eine *ClassCastException* wird geworfen. Aus diesem Grund muss der Response-Code geprüft und bei einem Fehlercode eine entsprechende Exception geworfen werden. Damit nicht in

allen Methoden dieses Exception Handling durchgeführt werden muss, wird in der Klasse *AbstractGHService* eine Methode bereitgestellt. Durch die Methode `executeRequest` ist es möglich, ein einheitliches Exception Handling zu implementieren. Jeder übergebene Request wird ausgeführt und bei erfolgreicher Durchführung, der Response zurückgegeben. Im Fehlerfall wird der mitgelieferte Fehlercode ausgelesen und die jeweilige Exception geworfen.

```
1         Response response = _request.execute();
2         if(response.isSuccessful()){
3             return response;
4         }
5
6         int code = response.code();
7         switch (code){
8             case 400:
9                 throw new GHClientException(response.message());
10            case 401:
11                throw new GHAuthenticationException(response.message());
12            case 403:
13                throw new GHRateLimitException(response.message());
14            case 404:
15                throw new GHClientException(response.message());
16            case 500:
17                throw new GHServerException(response.message());
18            default:
19                throw new GHException("Unknown response code! Something goes wrong");
20        }
21
```

Listing 5.2.6: Zentrale Fehlerbehandlung des GitHub-Clients

## 5.3 Fassaden

Wie bereits erwähnt, besteht die Software aus zwei logischen Teilen. Der eine umfasst die Programmlogik und wird von der CrawlEngine, PatchCrawlJob und ReviewCrawlJobs übernommen. Der andere ist auswechselbar und wird mithilfe der Plugins der Anwendung übergeben.

Speziell für die Durchführung der ReviewCrawlJobs spielen die Facaden eine wichtige Rolle. Bei diesen ist es entscheidend, dass korrekte Reviewtickets erstellt werden, die dann weiter verarbeitet werden können. Hierfür werden beispielsweise für die GitHubFacade vier Transformatoren-Klassen implementiert. Diese haben die Aufgabe die Objekte aus dem GitHub-Client in Objekte des Review Ticket Crawler zu portieren. Falls wichtige Attribute fehlen, wird eine *IllegalStateException* geworfen und das Programm beendet.

# 6 Evaluation

Um die Tauglichkeit der Anwendung zu demonstrieren, werden im folgenden Kapitel Open-Source-Projekte untersucht und ausgewertet. Um möglichst verschiedene Projekte auszuwählen, werden Kriterien aufgestellt, aus denen Projekte gesucht werden. Abschließend wird die Erweiterung anhand der gewonnenen Erfahrungen reflektiert und die positiven als auch die negativen Aspekte der Implementierung diskutiert.

## 6.1 Vorgehen

Ziel der Untersuchung von Open-Source-Projekte durch die Erweiterung des Patch-Flow Crawlers ist die Tauglichkeit der Anwendung für zukünftige Forschungen zu beweisen. Hierfür wurden im Vorfeld vier typische Fragestellung über Review-Prozesse vorgestellt. Diese mussten wegen eines fehlenden Inner-Source-Projekt umgestellt werden (siehe Abschnitt 3.1). Um einen Einsatz der Software zu simulieren, wird die Auswahl auf möglichst verschiedene Arten von Projekten angewendet. Zuerst werden Kriterien formuliert, zu denen passende Projekte gesucht werden.
Für die Auswertung werden einfache Datenbankabfragen erstellt die aus den Daten der Anwendung gewonnen wurden. Im Anhang der Tabelle 7.1 werden alle untersuchten Projekte aufgelistet mit dessen Link zu GitHub und wann diese untersucht wurden.

## 6.2 Auswahl der Kriterien

Damit die Tauglichkeit der Anwendung demonstriert werden kann, ist es entscheidend, möglichst unterschiedliche Projekte auszuwählen.
Das erste Kriterium ist der Projekttyp. In der Arbeit Nakakoji et al. (2002) werden drei Projekttypen herausgearbeitet, die sich auf GitHub befinden. Die Typen werden *exploration-based-*, *service-based-* und *utility-based-Projekte* genannt. Mit Exploration-based-Projekte werden diejenige Projekte bezeichnet, die für andere Anwendungen bestimmte Neuerungen, wie beispielsweise Vorgehensweisen oder Algorithmen, vorstellen. Service-based Projekte stellen einen Service dar. Darunter fallen beispielsweise Projekte wie der Webserver Apache oder die Datenbank PostgreSQL. Utility-based Projekte sind meistens Teil eines größeren Themenkomplexes, wie beispielsweise eines Service-based Projekt. Für diese Anwendungen werden zusätzliche Funktionalität benötigt und ein Teil der Community treibt diese in eigenständigen Projekten weiter.
Weitere Kriterien beschäftigen sich mit Merkmalen eines Projekts. Darunter fallen die Anzahl der Patches, PullRequests und Contributers. PullRequests sind Reviews für GitHub. Als Contributer werden Projektbeteiligte bezeichnet. Interessant für die Auswahl dieser Kriterien ist, ob die Anwendung kleine als auch große Mengen dieser Kriterien der Projekte gut bearbeiten kann. Zusammengefasst werden 24 Projekte untersucht und die Ergebnisse nachfolgend präsentiert.

## 6.3 Ergebnisse

In Tabelle 6.1 ergeben sich von allen 24 untersuchten Projekten folgende Werte: Ingesamt wurden knapp 30 % aller Patches die gereviewed wurden, abgelehnt. Die durchschnittliche Bearbeitungszeit eines Reviews betrug 560,3 Stunden. Insgesamt wurden 65,1 % aller Patches gereviewed und

| | |
|---|---|
| Abgelehnte Patches | 29,47 % |
| Bearbeitungszeit | 560,3 Stunden |
| Patches mit Reviews | 65,1 % |
| Anzahl Personen | 3,26 |

Tabelle 6.1: Zusammenfassung der Durchschnittlichen Werte aller Projekte

| Typ | abgelehnte Patches (%) | Bearbeitungzeit (Std) | Patches gere-viewed (%) | Personen |
|---|---|---|---|---|
| exploration-based | 36,5 | 627 | 71 | 3,6 |
| service-based | 23,5 | 288 | 57,1 | 3,2 |
| utility-based | 28,3 | 765,8 | 66,8 | 2,9 |

Tabelle 6.2: Aufteilung der Ergebnisse anhand der Projekttypen

es waren durchschnittlich 3,25 Personen daran beteiligt.

### 6.3.1 Projekttypen

Werden die Werte der einzelnen Kriterien betrachtet, ergeben sich deutliche Unterschiede. Zuerst werden die Projekte in drei Kategorien unterteilt, *exploration-based-*, *service-based-* und *utility-based-Projekte*.
Bei der Anzahl der Personen ergibt sich, dass weiterhin durchschnittlich drei Personen an einem Review beteiligt sind. Bei dem Anteil, wie viel Patches insgesamt gereviewed werden, ergibt sich nur eine kleine Differenz. Einen deutlichen Unterschied ergibt sich bei der Bearbeitungszeit eines Reviews. In den service-based Projekten werden die Reviews schneller bearbeitet als in den anderen beiden Kategorien. Hinsichtlich der abgelehnten Patches werden bei den exploration-based Projekten mehr abgelehnt als bei den anderen. Die Differenz ist hier jedoch sehr gering.

### 6.3.2 Merkmale der Projekte

Um zu erfahren, ob die Ergebnisse abhängig von den Projektmerkmalen, wie Anzahl der Patches, PullRequests oder Contributers sind, werden diese einzeln sortiert und ausgewertet. Die Auswertung wird in einem Liniendiagramm dargestellt.
Auf die horizontale Koordinate werden die sortierten Projekte gelegt. Die Sortierung ist abhängig von dem untersuchten Merkmal. Beispielsweise wird die Anzahl der Patches von geringer zu hoher Anzahl, beginnend von links auf die horizontale Achse platziert. Analog wird mit der Anzahl der PullRequests und Contributers verfahren. Die vertikale Koordinate zeigt die jeweiligen Ergebnisse der Projekte.

#### Anzahl der beteiligten Personen

Die durchschnittliche Anzahl der Beteiligten an einem Review, steigt leicht in allen drei Fällen, bei höherer Anzahl der Patches (siehe Abbildung 6.1), PullRequests (siehe Abbildung 6.3) oder Contributers (siehe Abbiludng 6.2).

#### Anteil der geprüften Patches

Der durchschnittliche Anteil der gereviewten Patches erfährt nur einen leichten Anstieg bei einer höheren Anzahl an PullRequests (siehe Abbildung 6.4).Die Schwankungen sind allerdings stark. Projekte wie *restangular*, *mermaid*, *vscode* und *babel* haben im Vergleich zu den anderen Projekten nur einen sehr niedrigen Wert.
Bei einer höheren Anzahl an Patches und Contributers können keine Zusammenhänge gefunden werden (siehe Abbildung 7.1 und 7.2 im Anhang).

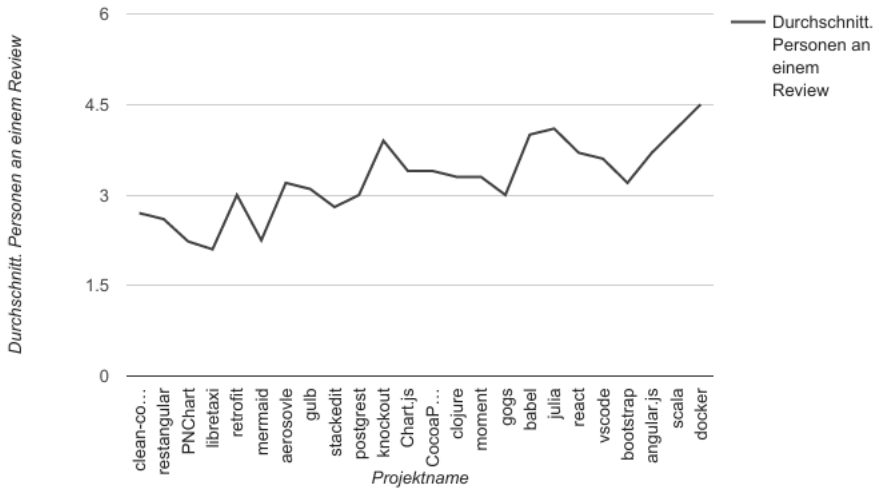


Abbildung 6.1: Durchschnittliche Anzahl der Personen an einem Review pro Projekt bei steigender Anzahl der Patchtes.

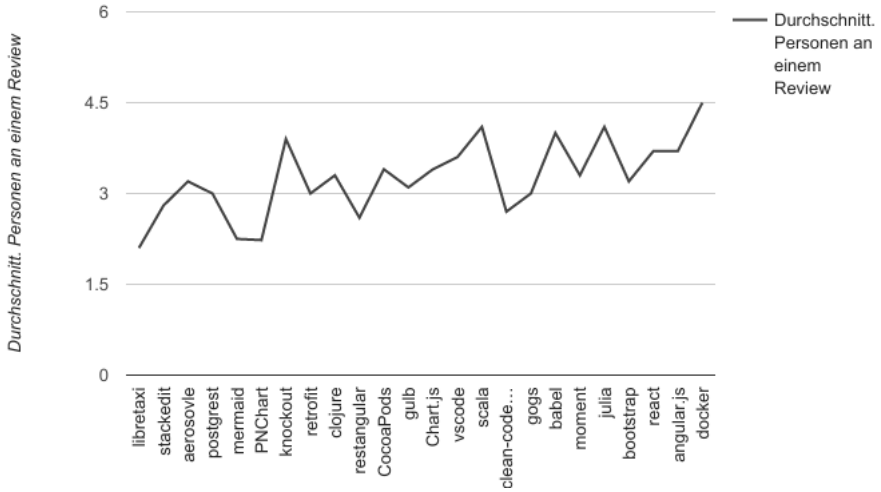


Abbildung 6.2: Durchschnittliche Anzahl der Personen an einem Review pro Projekt bei steigender Anzahl der Contributers.

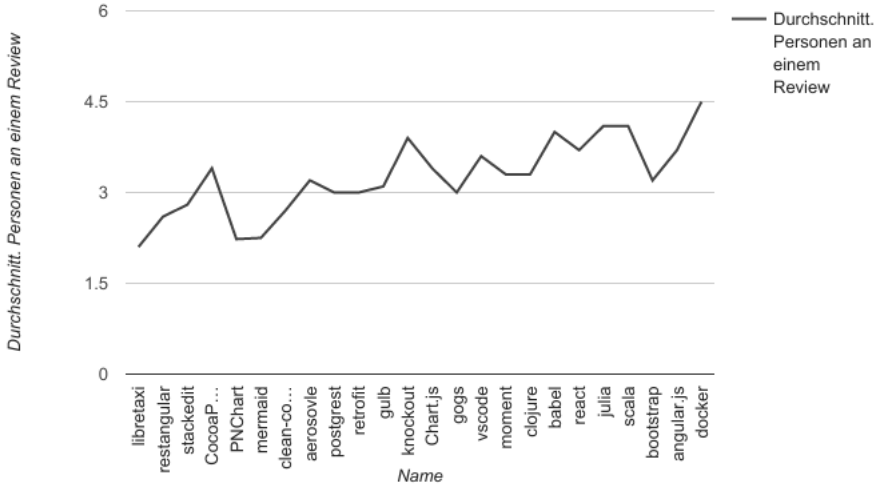


Abbildung 6.3: Durchschnittliche Anzahl der Personen an einem Review pro Projekt bei steigender Anzahl der PullRequests.

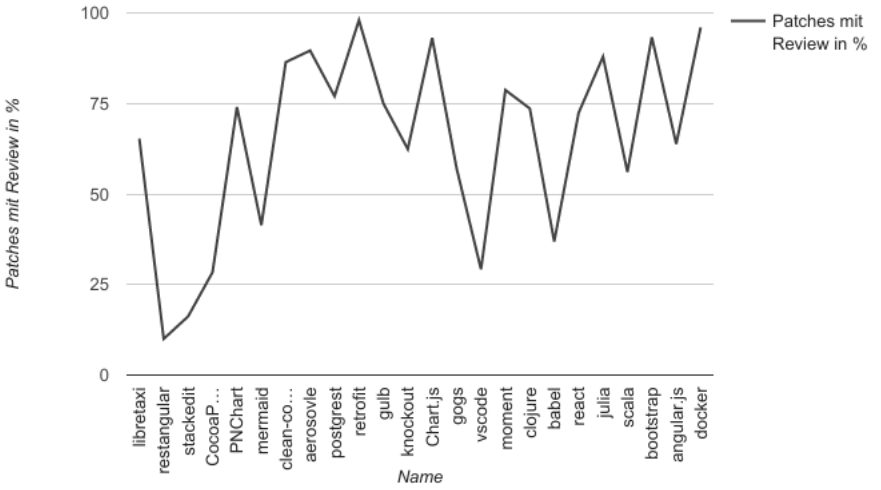


Abbildung 6.4: Durchschnittlich Anzahl der Patches die gereviewed wurden pro Projekt bei steigender Anzahl der PullRequests.

### Bearbeitungszeit eines Reviews

Die Abbildung 7.3, 7.4 und 7.5 im Anhang zeigen Liniendiagramme, die gut verdeutlichen, dass es keinen Zusammenhang zwischen einer höheren Anzahl an Patches, Contributers und PullRequests eines Projektes und der durchschnittlichen Bearbeitungszeit eines Reviews gibt.

### Abgelehnte Patches

Hinsichtlich der durchschnittlich abgelehnten Patches können keine Zusammenhänge zu der Anzahl an Patches und PullRequests festgestellt werden (siehe Abbildung 7.6 und 7.7 im Anhang). Lediglich die Anzahl der Personen beeinflusst leicht den durchschnittlichen Anteil der abgelehnten Patches (siehe Abbildung 6.5) Hier steigt der Wert mit einer höheren Anzahl an Projektbeteiligten.

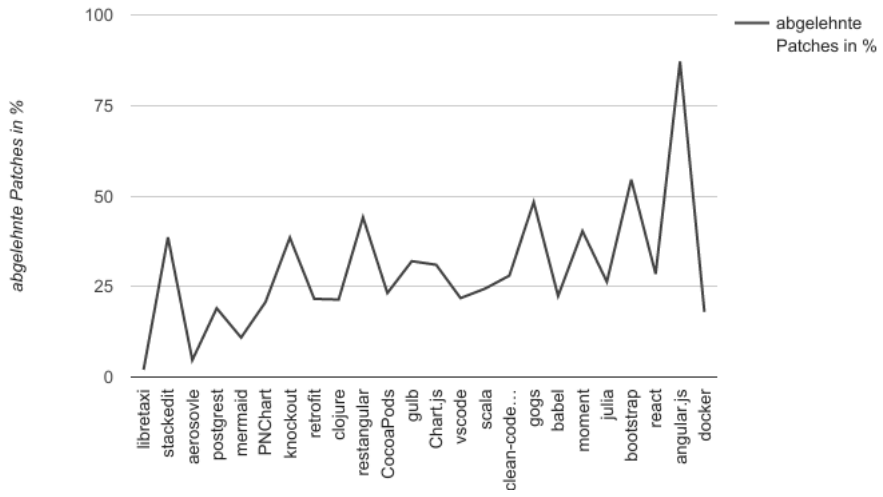


Abbildung 6.5: Durchschnittliche Anzahl der abgelehnte Patches pro Projekt bei steigender Anzahl der Contributers.

## 6.4 Fazit

Durch den Testlauf der GitHub-Projekte konnte grundsätzlich die Tauglichkeit der neuen Komponente bewiesen werden. In Kapitel 3.2 wurden Anforderungen an die Software gestellt, die einzeln überprüft werden.
Die nichtfunktionalen Anforderungen in Abschnitt 3.2, NF1 und NF2, sind beide erfüllt worden. Die Software konnte problemlos an den PatchFlow Crawler angeknüpft werden und musste schwerpunktmäßig an dem Plugin, der CrawlEngine und dem Datenbankmodell erweitert werden. Die Erweiterungen werden in Abschnitt 4 genauer erklärt. Außerdem wurde durch eine geeignete Architektur sichergestellt, dass die neue Komponente für verschiedene Systeme erweiterbar ist. Zum einen ist das Plugin-System übernommen worden. Zum anderen wurden Facaden zur Bearbeitung der verschiedenen Reviewtools eingeführt.
Die funktionalen Anforderungen in Abschnitt 3.2 konnten durch das Datenbankmodell vollständig abgedeckt werden. Die erste Anforderung **F1** wird dadurch festgestellt, dass ein *Patch* mit einer *PatchAddedActivity* verbunden ist. Werden keine *PatchAddedAcitvity* für einen Patch gefunden, wurde dieser nicht gereviewed. In dem zugehörigen Reviewticket der *PatchAddedActivity*, kann ermittelt werden, ob dieses noch offen, abgelehnt oder angenommen wurde. Mit der Erkenntnis kann eine Abfrage zur Beantwortung von **F2** realisiert werden. Jede Activity hat einen Ersteller und ist einer Person zugeordnet. Dadurch können alle Aktivitäten eines Reviews abgefragt und die Anforderung **F3** erfüllt werden. Die Dauer eines Reviews definiert sich auch über dessen Activities. In Abschnitt 4.1.2 sind alle Aktivitäten aufgelistet. Durch die Aktivität *RejectActivity* und *AcceptActivity* kann die Dauer eines Reviews bestimmt werden, mit dem die Abfrage **F4** gewährleistet wird.

## 6.5 Ausblick

Für zukünftige Forschungen ist die Software einsatzbereit und kann Daten der Patches und Reviews erfassen. Als Nebenprodukt sind zwei Clients für die GitHub und GitLab API entstanden. Diese können bisher nur Daten erfassen, die für den Zweck dieser Arbeit benötigt wurden. Die Clients sind jedoch gut erweiterbar und können zukünftig von dem Crawler gelöst werden, um eigenständig weiterentwickelt zu werden. Das beschriebene Problem in Abschnitt 4.7.1 ist ungelöst. Für die bisherige Analyse der GitHub-Projekte ist das Problem einmal aufgetreten. Bei Inner-Source-Projekten würde es wegen dem fehlenden Rate Limit nicht auftreten. Es muss also abgeschätzt werden, ob es sich rentiert, den Crawler speziell für diese Ausnahme anzupassen.

# Literaturverzeichnis

Bitergia. Bitergia reports - dig deeper with bitergia reports, 2015. URL https://bitergia.com/products/reports/. aufgerufen am 30.01.2017.

Bitgeria. Bitgergia dashboards - built for software development analysis, a. URL https://bitergia.com/products/dashboards/. aufgerufen am 30.01.2017.

Bitgeria. Bitgergia grimoirelab - free, open source tools for software development analytics, b. URL https://bitergia.com/opensource/. aufgerufen am 30.01.2017.

Maximilian Capraro and Dirk Riehle. Inner source definition, benefits, and challenges. page 36, 2016.

GitHub Documentation. Github documentation, a. URL https://developer.github.com/v3/. aufgerufen am 25.02.2017.

GitLab Documentation. Gitlab documentation, b. URL https://docs.gitlab.com/ee/api/README.html. aufgerufen am 25.02.2017.

Arie van Deursen Georgios Gousios, Martin Pinzger. An exploratory study of the pull-based software development model. 2014a.

Arie van Deursen Georgios Gousios, Martin Pinzger. Small patches get in! 2014b.

GitHub. Github api - rate limit. URL https://developer.github.com/v3/rate_limit/. aufgerufen am 23.03.2017.

Georgios Gousios. The GHTorrent dataset and tool suite. 2013.

Yujuan Jiang, Bram Adams, and Daniel M. German. My patch make it? and how fast? 2013.

GitHub Libraries. Github api, a. URL https://developer.github.com/libraries/. aufgerufen am 25.02.2017.

GitLab Libraries. Gitlab api, b. URL https://about.gitlab.com/applications/#api-clients. aufgerufen am 25.02.2017.

Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. 2002.

Reid Holmes und Michael W. Godfrey Olga Baysal, Oleksii Kononenko. The influence of non-technical factors on code review. 2013.

Dirk Riehle. Value object. 2006.

Dirk Riehle, Maximilian Capraro, and Detlef Kips. Inner source in platform-based product engineering. 2016.

# 7 Anhang

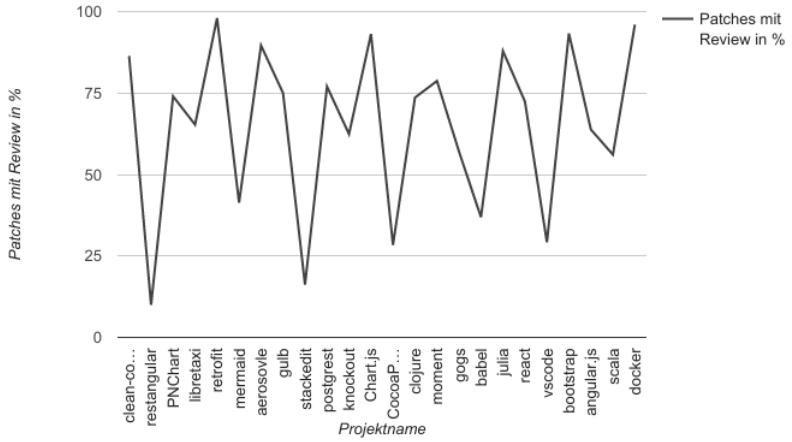


Abbildung 7.1: Durchschnittlich Anzahl der Patches die gereviewed wurden pro Projekt bei steigender Anzahl der Patches.

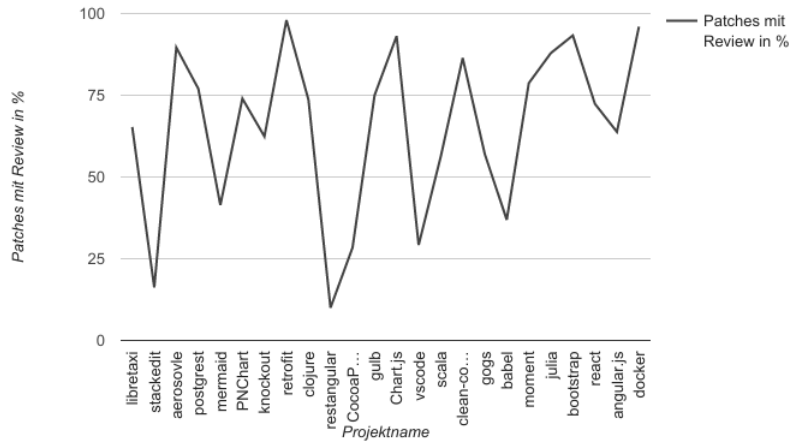


Abbildung 7.2: Durchschnittlich Anzahl der Patches die gereviewed wurden pro Projekt bei steigender Anzahl der Contributers.

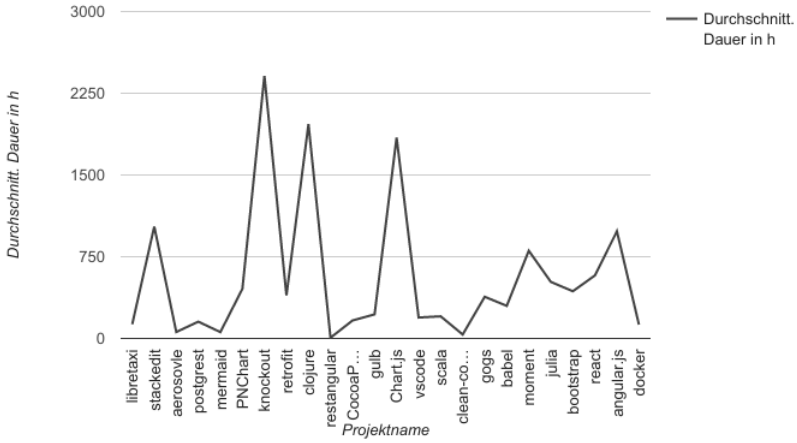


Abbildung 7.3: Durchschnittliche Bearbeitungszeit eines Reviews pro Projekt bei steigender Anzahl der Contributers.

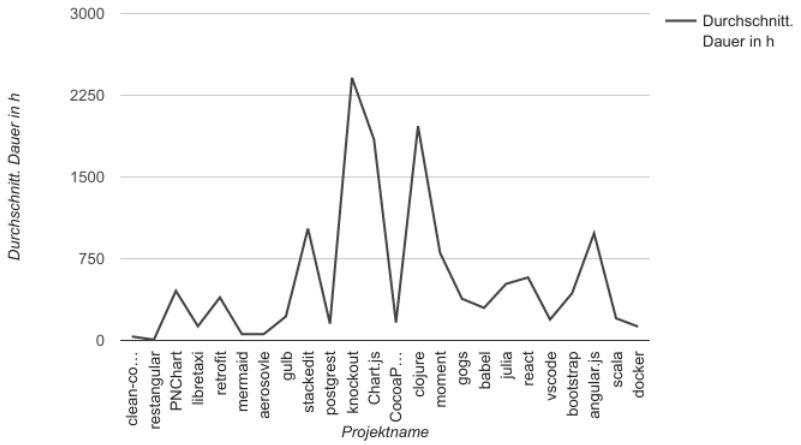


Abbildung 7.4: Durchschnittliche Bearbeitungszeit eines Reviews pro Projekt bei steigender Anzahl der Patches.

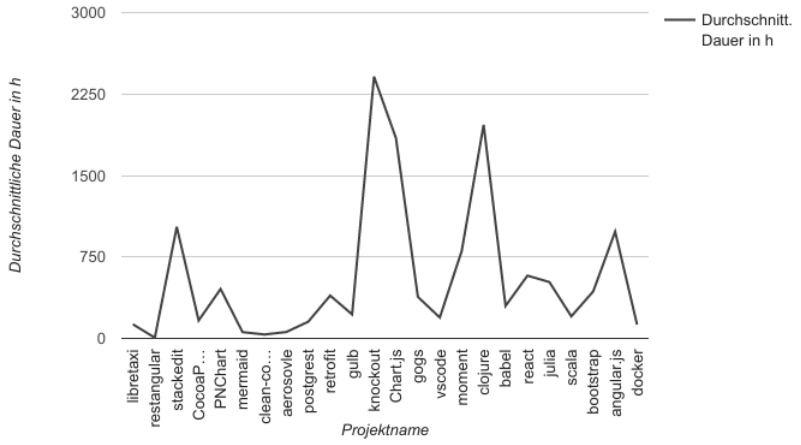


Abbildung 7.5: Durchschnittliche Bearbeitungszeit eines Reviews pro Projekt bei steigender Anzahl der PullRequests.

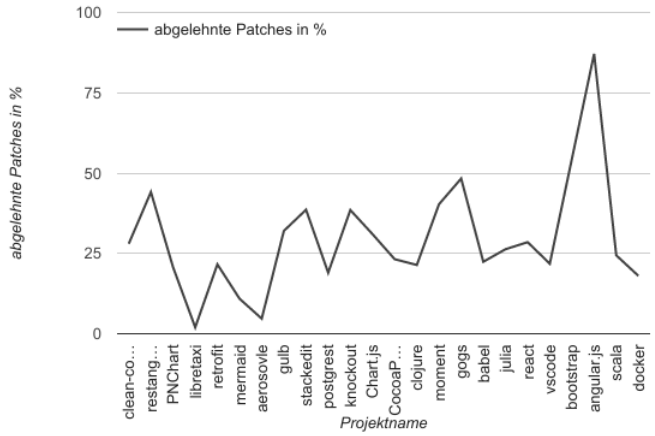


Abbildung 7.6: Durchschnittliche Anzahl der abgelehnte Patches pro Projekt bei steigender Anzahl der Patches.

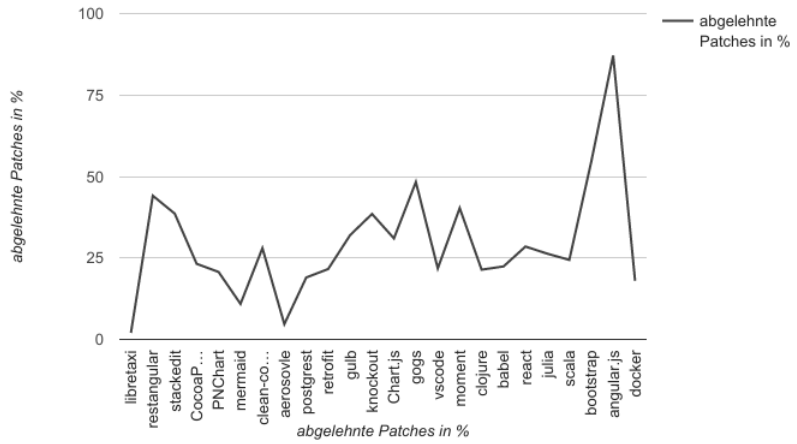


Abbildung 7.7: Durchschnittliche Anzahl der abgelehnte Patches pro Projekt bei steigender Anzahl der PullRequests.

| Name | URL | Erfassung am: |
|---|---|---|
| angular.js | https://github.com/angular/angular.js | 1. Februar 2017 |
| react | https://github.com/facebook/react | 4. Februar 2017 |
| bootstrap | https://github.com/twbs/bootstrap | 2. Februar 2017 |
| babel | https://github.com/babel/babel | 20. Februar 2017 |
| scala | https://github.com/scala/scala | 20. Februar 2017 |
| clean-code-javascript | https://github.com/ryanmcdermott/clean-code-javascript | 1. Februar 2017 |
| julia | https://github.com/JuliaLang/julia | 22. Februar 2017 |
| clojure | https://github.com/clojure/clojure | 21. Februar 2017 |
| docker | https://github.com/docker/docker | 5. Februar 2017 |
| gogs | https://github.com/gogits/gogs | 6. Februar 2017 |
| gulb | https://github.com/gulbjs/gulb | 6. Februar 2017 |
| vscode | https://github.com/Microsoft/vscode | 24. Februar 2017 |
| moment | https://github.com/moment/moment | 5. Februar 2017 |
| Chart.js | https://github.com/chartsjs/Chart.js | 2. Februar 2017 |
| mermaid | https://github.com/knsv/mermaid | 9. Februar 2017 |
| retrofit | https://github.com/square/retrofit | 10. Februar 2017 |
| libretaxi | https://github.com/ro31337/libretaxi | 15. Februar 2017 |
| aerosolve | https://github.com/airbnb/aerosolve | 15. Februar 2017 |
| postgrest | https://github.com/begriffs/postgrest | 14. Februar 2017 |
| PNChart | https://github.com/kevinzhow/PNChart | 25. Februar 2017 |
| restangular | https://github.com/mgonto/restangular | 22. Februar 2017 |
| knockout | https://github.com/knockout/knockout | 23. Februar 2017 |
| CocoaPods | https://github.com/CocoaPods/CocoaPods | 24. Februar 2017 |
| stackedit | https://github.com/benweet/stackedit | 20. Februar 2017 |

Tabelle 7.1: Auflistung aller untersuchten Projekte mit dessen URL und wann diese Untersucht wurden