Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Analyse, Entwurf und prototypische Implementierung einer Wissensdatenbank zur Teilautomatisierung des Code-Review-Prozesses im Saros-Projekt

Dennis Ventzke
Matrikelnummer: 4470800
dennis.ventzke@fu-berlin.de

Betreuer: Franz Zieris, M.Sc.
Eingereicht bei: Prof. Dr. Lutz Prechelt

Berlin, 22. September 2016

Zusammenfassung

Seit 2006 wird im Rahmen des Saros-Projekts an einem IDE-Plugin gearbeitet, dass die Paar-Programmierung verbessern und vor allem erleichtern soll. In dieser Zeit wurde nicht nur das Produkt an sich, sondern auch der Entwicklungsprozess verbessert. Insbesondere im Code-Review-Prozess wurden einige Teile bereits vollautomatisiert. Dennoch teilten einige langjährige Saros-Entwickler die Vermutung, dass gewisse Fehler immer wieder begangen werden und daher immer wieder thematisiert werden müssen, was ermüdend sein kann und weitere Probleme mit sich bringt.

In dieser Bachelorarbeit stelle ich zunächst fest, dass diese Vermutung der Wirklichkeit entspricht, erarbeite anschließend die Anforderungen an ein System, das die Probleme in Form einer Teilautomatisierung lösen kann, worauf abschließend der Entwurf und die prototypische Implementierung solch einer Wissensdatenbank folgt.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

22. September 2016

Dennis Ventzke

Inhaltsverzeichnis

1 Einleitung

Dieses Kapitel gibt zunächst einen Einblick in die Problematik, der sich diese Bachelorarbeit widmet. Anschließend wird kurz vorgestellt, was im Rahmen dieser Arbeit erreicht werden soll und wie sie strukturiert ist.

1.1 Problemstellung

Die Arbeitsgruppe „Software Engineering“ der Freien Universität Berlin arbeitet seit 2006 an einem Open-Source-Projekt namens „Saros“ mit dem Ziel, Paar-Programmierung zu vereinfachen. In dieser Zeit wurde vielen Studenten die Möglichkeit gegeben, in Form einer Bachelor- oder Masterarbeit daran mitzuwirken. Allein auf der Saros-Webseite sind derzeit 89 Mitwirkende aufgeführt.[1]
In der Entwicklung des Projekts gilt ein strikter Code-Review-Prozess, der besagt, dass kein Code in die Code-Basis aufgenommen wird, der nicht gemäß dieses Prozesses begutachtet wurde. Hilfreiche Gutachten, die insbesondere bei den vielen neuen Projektmitgliedern wichtig sind, zu schreiben ist sehr aufwändig, da der komplette Code, sowie dessen beabsichtigte und tatsächliche Funktion verstanden werden muss, um Fehler erkennen zu können. Für diese Fehler muss anschließend noch eine Erklärung und bestenfalls auch ein oder mehrere Lösungsvorschläge gegeben werden.
Um den hohen Aufwand für den Gutachter zu reduzieren, wurden bereits viele Maßnahmen ergriffen. Es existieren Tests, die automatisch bei jeder Änderung des Codes ausgeführt werden, zudem liefert Sonarqube[2], ein Open-Source-Werkzeug zur automatischen Codeanalyse, Rückmeldung, gegen welche Regeln und Konventionen der geschriebene Code verstößt. Außerdem ist Archnemesis im Einsatz, ein von Arsenij E. Solovjev im Rahmen seiner Masterarbeit[Sol14] entwickeltes Werkzeug, welches Verstöße gegen aufgestellte Architekturregeln überwacht. Das hat zur Folge, dass bestimmte Arten von Fehlern schon durch den Einsatz dieser automatischen Werkzeuge aufgedeckt werden, bevor der Gutachter sich dem Code widmet.
Stefan Rossbach und Franz Zieris, beide langjährige Entwickler im Saros-Projekt, ist jedoch aufgefallen, dass sich Teile der Gutachten dennoch wiederholen, da bestimmte Ratschläge immer wieder erteilt werden müssen. Für den Gutachter bedeutet das, dass er bei jedem Bedarf eines solchen Ratschlags (1) eine neue Erklärung dafür schreiben, (2) nach bereits verfassten Erklärungen von ihm selbst oder anderer Gutachter suchen oder (3) die Erklärung aus einem selbst verwalteten Archiv kopieren muss. In allen drei Fällen würde eine Automatisierung den Aufwand für die Gutachter noch weiter reduzieren.

[1] http://www.saros-project.org/history
[2] http://www.sonarqube.org/

1.2 Ziele der Arbeit

Ziel dieser Arbeit ist es zunächst zu klären, ob sich Teile der Gutachten im Saros-Projekt tatsächlich wiederholen, es also im Review-Prozess wiederholt mehrere Vorkommnisse von Bedarfen für ein bestimmtes „Topic“[3] gibt, da dies bisher nur auf der Wahrnehmung von Stefan Rossbach und Franz Zieris beruht.
Wenn dies der Fall ist, soll eine geeignete Art der Automatisierung gewählt und implementiert werden. Dabei ist zu beachten, dass diese Automatisierung bei erfolgreicher Anwendung im Saros-Projekt auch für andere Open-Source-Projekte erweitert werden soll, die ebenfalls einen Code-Review-Prozess betreiben, jedoch eventuell andere Review-Werkzeuge dafür benutzen. Diese Ausdehnung auf andere Projekte soll jedoch nicht im Rahmen dieser Arbeit geschehen, sie soll lediglich bei wichtigen Entscheidungen eine Rolle spielen.

1.3 Aufbau der Arbeit

Im folgenden Kapitel werden die wichtigsten Begriffe definiert und alle Grundlagen erklärt, die für das Verständnis der restlichen Arbeit erforderlich sind. Das dritte Kapitel befasst sich mit der Frage, ob im Saros-Projekt ein Bedarf für eine Automatisierung im Rahmen des Code-Review-Prozesses besteht und wie dieser Bedarf konkret beschaffen ist. Anschließend werden in Kapitel vier, unter anderem mit den gewonnenen Erkenntnissen aus Kapitel drei, die Anforderungen erarbeitet, die sich an die zu entwickelnde Software ergeben. Die danach folgenden Kapitel beinhalten den Entwurf sowie Besonderheiten und Schwierigkeiten der Implementierung. Abschließend wird ein Fazit gezogen, indem das im Rahmen dieser Arbeit tatsächlich Erreichte mit dem Vorgenommenen verglichen wird.

[3] Mit „Topic“ sind hier und im weiteren Verlauf der Arbeit sowohl allgemeine Prinzipien, als auch Lösungen für Probleme gemeint, die durchaus sehr projektspezifisch sein können. Topics zeichnen sich dadurch aus, dass sie auf viele Fälle Anwendung finden, während Fehler in ihrer konkreten Form in der Regel einzigartig sind. Ein Fehler kann somit den konkreten Bedarf eines solchen Topics darstellen.

2 Grundlagen

Da in dieser Arbeit der Code-Review-Prozess des Saros-Projekts verbessert werden soll, werden in diesem Kapitel Code-Reviews allgemein erklärt, sowie deren konkrete Umsetzung im Rahmen des Saros-Projekts beleuchtet.

2.1 Code-Review

Den Ursprung der Code-Reviews stellen die von Fagan[Fag76] 1976 definierten „Code-Inspektionen“ dar. Dabei wird ein sehr strikter, formaler Prozess beschrieben, bei dem geschriebener Code von mehreren Personen in mehreren Phasen Zeile für Zeile untersucht wird. Dieser Vorgang kostet jedoch sehr viel Zeit, weshalb sich insbesondere in Open-Source-Projekte neue Code-Review-Prozesse entwickeln, die Review-Werkzeuge benutzen[Rig12], um den Aufwand für alle Beteiligten zu reduzieren, und sich damit von den klassischen Code-Inspektionen entfernen. Alberto Bacchelli und Christian Bird[BB13] haben daher 2013 den Begriff der „modernen Code-Reviews“ als Gutachten definiert, die (1) informell geschehen, (2) von Review-Werkzeugen unterstützt und (3) derzeit regelmäßig angewendet werden. Diese Definition lässt dabei Freiraum in Bezug auf die konkrete Art der Umsetzung. Bacchelli und Bird haben verschiedene Entwicklungsteams von Microsoft untersucht und dabei festgestellt, was deren Motivation ist, trotz der Abkehr von den Code-Inspektionen nach Fagan dennoch an der Idee festzuhalten, Code-Reviews durchzuführen, wenn auch im modernen Sinne:

1. **Defekte finden:**
   Ein offensichtlicher Vorteil ist, dass der Gutachter Defekte finden kann, die durch begangene Fehler des Autors entstanden sind. Falls diesem Fehler ein Irrtum zugrunde liegt, der Autor ihn also nicht versehentlich begangen hat, wird hier sogleich Wissen vermittelt, worauf im vierten Punkt dieser Auflistung näher eingegangen wird.

2. **Code verbessern:**
   Neben der Beseitigung von Defekten spielt auch die Verbesserung des Codes eine große Rolle. Dabei sind alle Eigenschaften des Codes gemeint, die sich nicht auf die korrekte Erfüllung der funktionalen Anforderungen beziehen, wie beispielsweise Lesbarkeit, angemessenes Kommentieren und die Einhaltung von teamspezifischen Konventionen.

3. **Alternative Lösungen aufzeigen:**
   Gutachter können außerdem neue Ansätze vorschlagen, die es dem Autor ermöglichen, bessere Lösungen zu implementieren.

4. **Wissen vermitteln:**
   Ein weiterer wichtiger und für diese Arbeit sehr relevanter Punkt ist

der Austausch von Wissen zwischen den Entwicklern. Dies funktioniert dabei in beide Richtungen. Zum einen kann der Gutachter sein Wissen und seine Erfahrungen durch die Kommentare mit dem Autor teilen, zum anderen muss sich der Gutachter selbst für die Überprüfung in den Code einarbeiten. Dadurch lernt er sowohl, wie das konkrete Feature implementiert oder der konkrete Fehler beseitigt wurde, also das Ziel des Patches, als auch die Methoden und Werkzeuge, die der Autor dafür verwendet hat.

5. **Transparenz fördern:**
   Dass sich Entwickler nicht ausschließlich mit ihrem eigenen Code befassen können führt auch dazu, dass sie sich in einem höheren Maße über die Arbeit der Teammitglieder bewusst sind. Dadurch kann es ihnen leichter fallen zu erkennen, wann sie beispielsweise von Teammitgliedern bereitgestellte Funktionen für ihren eigenen Code nutzen können oder aber, wann diese Teammitglieder Code einreichen wollen, der bestehende Funktionen einschränkt.

2.2 Saros

Bei der Paar-Programmierung sitzen zwei Entwickler an einem Rechner und arbeiten gemeinsam an einem Algorithmus, einer Programmier-Aufgabe oder einem Design[CW00]. Saros wurde mit dem Ziel ins Leben gerufen, dass beide Entwickler ihre eigenen Rechner benutzen können, was beispielsweise den Vorteil hat, dass sie in einer vertrauten Umgebung arbeiten können und sich bei bestehender Audioverbindung nicht einmal am gleichen Ort befinden müssen.[4]
Mittlerweile bietet Saros als IDE-Plugin unter anderem Teams mit beliebiger Größe die Möglichkeit, verteilt in Echtzeit Dateien zu bearbeiten und jederzeit verfolgen zu können, wer welche Änderungen vorgenommen hat.[5]

2.3 Gerrit

Gerrit[6] ist ein Code-Review-System, das vom Saros-Projekt als Plattform benutzt wird, um Reviews durchzuführen. Es zeichnet sich dadurch aus, dass jede Änderung, die an der Codebasis vorgenommen werden soll, zunächst lediglich als „Change“ in Gerrit gespeichert wird. Dieser kann dann über die bereitgestellte Weboberfläche diskutiert und bewertet werden. Sollte es Bedarf zur Nachbesserung geben, kann der Autor eine weitere Iteration dieses „Changes“ hochladen. Diese Iterationen werden als „Patch Sets“ bezeichnet. Bei der Durchsicht des Codes hat der Gutachter dann die Möglichkeit, Kommentare auf Ebene des „Patch Sets“, der einzelnen Dateien, die verändert

[4] http://www.saros-project.org/history
[5] http://www.saros-project.org/features
[6] https://www.gerritcodereview.com/

wurden, oder sogar auf eine konkrete Codezeile bezogen abzugeben. Diese Zeilen- und Dateikommentare werden dabei zunächst als „Draft Comments“, also Entwürfe gespeichert und sind erst dann für andere sichtbar, wenn der Gutachter die Durchsicht als abgeschlossen kennzeichnet. Bei diesem Abschluss muss er dem „Patch Set“ eine Bewertung auf einer Skala von -2 bis +2 geben und hat die Möglichkeit, einen optionalen zusammenfassenden Kommentar auf Ebene des „Patch Sets“ abzugeben.

2.4 Entwicklung im Saros-Projekt

Im Saros-Projekt gilt die strikte Regel, dass kein Code in den Quellcode aufgenommen wird, der nicht den vorgegebenen Code-Review-Prozess durchlaufen ist[7]. Dies wird zum einen damit begründet, dass dadurch Defekte schon frühzeitig aufgedeckt werden und die Kosten zur Beseitigung somit wesentlich geringer ausfallen, als wenn der Defekt erst später gefunden worden wäre, da in der Zwischenzeit beispielsweise neuer Code geschrieben wurde, der dann ebenfalls angepasst werden muss. Zum anderen wird auf den oben bereits beschriebenen Effekt des Wissenstransfers verwiesen, der sowohl dem Gutachter, als auch dem Autor dabei hilft, jetzt und in Zukunft besseren Code zu schreiben.
Dabei sind nicht nur die Code-Reviews an sich, sondern auch die Vorbereitung des Codes klar geregelt, wobei jeder Schritt, der vom Code-Autor befolgt werden muss, die Code-Review an sich konkret unterstützt:

1. **Lokalen Commit erstellen.**
   Hierbei soll darauf geachtet werden, dass die Größe der Änderung möglichst klein gehalten und auf mehrere Änderungsanfragen aufgeteilt wird. Je schneller es geht eine Code-Review durchzuführen, desto eher werden Gutachter dazu bereit sein. Außerdem soll besonderen Wert darauf gelegt werden, dass die die Änderung beschreibende Nachricht so prägnant wie möglich formuliert ist.

   Dieser Schritt dient demnach dazu, den Prozess so einfach wie möglich für den Gutachter zu gestalten.

2. **Review-Anfrage in Gerrit erstellen.**
   Dadurch wird wie in 2.3 beschrieben ein „Change“ in Gerrit angelegt.

3. **Auf Rückmeldung von Jenkins warten und eventuelle Fehler beheben.**
   Jenkins[8] ist ein Werkzeug zur kontinuierlichen Integration, welches den Quellcode überwacht und bei jeder Änderung alle betroffenen Teile neu kompiliert und die entsprechenden automatisierten Tests ausführt.

[7] http://www.saros-project.org/review
[8] https://jenkins.io/

4. **Andere Entwickler dazu einladen, die eigenen Änderungen zu begutachten (währenddessen zwei eigene Code-Reviews durchführen).**
   Da für jede erstellte Review-Anfrage zwei eigene Reviews durchgeführt werden sollen, soll dieser Schritt sicherstellen, dass sich immer genügend Gutachter finden lassen.

   Wie diese Reviews konkret aussehen, ist in 2.5 näher beschrieben.

5. **Die von den Gutachtern angesprochenen Probleme verstehen und mögliche Lösungen diskutieren.**
   Dies soll den Wissensaustausch sicherstellen und gleichzeitig Mehraufwand durch Missverständnisse vermeiden.

6. **Die im fünften Schritt ausgearbeiteten Lösungen implementieren und zu Schritt drei gehen.**
   Hier ist zu beachten, dass die Änderung auch durch diesen Schritt nicht größer wird als unbedingt nötig. Eventuell sollten die in Schritt 5 erarbeiteten Lösungen also in einem neuen Change implementiert werden.

7. **Code zur Codebasis hinzufügen, sofern es genügend positive und keine negativen Bewertungen gab.**
   Dabei beziehen sich „positive“ und „negative“ Bewertungen auf die in 2.3 beschriebene Skala.

   Dadurch, dass zum Zeitpunkt des Hinzufügens des Codes zur Codebasis keine negativen Bewertungen erlaubt sind soll sichergestellt werden, dass kritische Rückmeldungen nicht ignoriert werden können und somit auf die Bedenken des Gutachters eingegangen wird, der eine negative Bewertung gegeben hat.

2.5 Code-Review im Saros-Projekt

Die Struktur der Reviews im Saros-Projekt orientiert sich an dem in 2.3 beschriebenen allgemeinen Ablauf einer Gerrit-Review[9]. Inhaltlich werden dabei folgende Dinge vorgegeben:

- **Fehler finden:**
  Um Fehler zu finden, sollen verschiedene Perspektiven vom Gutachter eingenommen werden, wenn er den Code betrachtet. So sollen sowohl die einzelnen veränderten Codezeilen und deren umgebenden Methoden untersucht werden, als auch der „Change“ als Ganzes betrachtet werden. Insbesondere wenn neue Strukturelemente wie Methoden,

[9] http://www.saros-project.org/gerrit-review

Schnittstellen oder Klassen hinzukommen, soll der gesamte Code inklusive der unveränderten Codezeilen dahingehend betrachtet werden, ob diese neu hinzugefügten Elemente nicht schon in einer ähnlichen Form existieren, bzw. die gleiche Funktionalität mit bestehenden Komponenten erreicht werden kann.

- **Probleme präsentieren:**
  Wenn der Gutachter einen Fehler gefunden hat, soll er zunächst das dahinterstehende Problem erklären und dem Autor verdeutlichen, weshalb dieser hier einen Fehler begangen hat. Wäre dem Autor das schon bewusst, hätte er den Fehler vermutlich nicht begangen.

  Anschließend soll der Gutachter vorschlagen, wie das Problem gelöst werden kann und Anforderungen an eine solche Lösung stellen. Wenn möglich, soll er abschließend per URL auf weiterführende Hintergrundinformationen verweisen. Dies gilt sowohl für die Frage weshalb hier ein Problem vorliegt, als auch für die Frage wie dieses identifizierte Problem gelöst werden kann.

- **Umgang mit wiederkehrenden Problemen:**
  Bestimmte Fehler treten an vielen verschiedenen Stellen auf. Fällt dem Gutachter so etwas auf, so soll er das dahinterstehende Problem (im Sinne des in 1.2 definierten „Topics“) anhand eines konkreten Fehlers erläutern und mitteilen, dass es sich hierbei um ein wiederkehrendes Problem handelt. Damit ist es dem Autor überlassen, alle Stellen zu finden, an dem solch ein Fehler begangen wurde.

- **Umgang mit Mangel an Verständnis:**
  Wenn der Gutachter bestimmte Stellen des Codes nicht nachvollziehen kann, soll er einfach nachfragen. Somit lernt er dazu und kann anschließend seinen eigenen Blickwinkel beisteuern.

- **Einstieg in einen bestehenden Review-Prozess:**
  Auch wenn ein Gutachter nicht der erste ist, der einen „Change“ bzw. ein „Patch Set“ untersucht, so soll er dem Autor auch dann seine Meinung mitteilen, wenn er nichts Neues beizutragen hat. Schließlich ist es für den Autor durchaus interessant, ob fünf Gutachter einen Punkt für wichtig erachten oder das die Meinung eines Einzelnen ist.

3 Bedarfsanalyse des Saros-Projekts

In diesem Kapitel wird untersucht, ob der in Kapitel eins vermutete Bedarf an einer Automatisierung von Teilen des Review-Prozesses tatsächlich im Saros-Projekt vorliegt und wie die ebenfalls in Kapitel eins definierten „Topics“ konkret aussehen. Dies soll zunächst dabei helfen zu klären, ob die Entwicklung und Implementierung einer Automatisierung gerechtfertigt ist. Zudem soll das Verständnis der konkreten Bedürfnisse bei der Anforderungsanalyse helfen. Letztlich soll dabei ein Grundbestand an Topics samt zugehörigen Lösungen entstehen, der Teil der Automatisierung werden soll.

3.1 Ausarbeitung des Vorgehens

Um zu zeigen, dass sich im Review-Prozess des Saros-Projekts Teile der Gutachten wiederholen, weil immer wieder Fehler begangen werden, die einem bestimmten „Topic“ zuzuordnen sind, muss zunächst entschieden werden, wie diese Topics und ihre zugehörigen Fehler gesucht werden sollen. Mögliche Quellen sind beispielsweise

1. die bisherigen Review-Kommentare in Gerrit,
2. die Webseite des Saros-Projekts, auf der bereits viele Hinweise und Lösungen gesammelt wurden,
3. die Dokumentation des bestehenden Codes an sich, sowie
4. die direkte Kommunikation mit den Saros-Entwicklern.

Da es sowohl darum geht konkrete Beispiele aufzuzeigen, in denen Topics immer wieder im Rahmen des Review-Prozesses thematisiert wurden, als auch darum einen Grundbestand an Topics aufzubauen, bietet sich als Quelle Variante eins an, also eine Analyse der bestehenden Kommentare dieses Review-Prozesses. Dies soll grundsätzlich wie folgt ablaufen:

1. **Finden und Zusammentragen von möglichen Topics**
   Zunächst sollen Kommentare gefunden werden, in denen ein Problem, das mehrfach auftreten könnte, ausführlich erklärt wird, da diese Erklärung ein Kandidat für den Grundbestand an Topics ist, der aufgebaut werden soll.
2. **Untersuchung eines Topic-Kandidaten**
   Einer dieser im vorherigen Schritt gefundenen Kandidaten soll dahingehend untersucht werden, ob tatsächlich mehrere Fehler begangen wurden, die diesem Topic zuzuordnen sind, um den zu Beginn dieses Kapitels erwähnten Bedarfs nachzuweisen.

Beide Schritte sollen zudem dazu dienen, den bestehenden Bedarf besser nachvollziehen zu können, um eine erfolgreiche Anforderungsanalyse zu ermöglichen.

Nun ist zu klären, wie diese Kommentare, die ausführliche Kommentare zu möglichen Topics enthalten, gefunden werden sollen. Problematisch ist hierbei zum einen die mit 53.925 Exemplaren hohe Anzahl der Kommentare, die zur Verfügung stehen. Zum anderen fehlen mir die Erfahrungswerte eines Saros-Entwicklers.
Um die Anzahl der Kommentare auf ein überschaubares Maß zu reduzieren, kommen mehrere Filterkriterien in Betracht:

1. **Länge der Kommentare:**
   Um die Wahrscheinlichkeit zu erhöhen, dass ein Kommentar eine Erklärung eines Problems enthält, kann nach der Länge gefiltert werden. Je länger ein Kommentar ist, desto wahrscheinlicher erscheint es, dass ein Problem vorlag und auf dieses ausführlich eingegangen wurde.

2. **Diskussionslänge (Anzahl Kommentare pro "Change"/"Patch Set"/Datei/Zeile):**
   Ausführlich diskutierte Probleme könnten ebenfalls dadurch gefunden werden, dass eine bestimmte Diskussionslänge vorausgesetzt wird, also mehrere Kommentare, die sich beispielsweise auf eine konkrete Codezeile beziehen.

3. **Ähnlichkeitsmaß der Kommentare:**
   Um umfangreiche Problembeschreibungen zu finden, könnte nach dem Ähnlichkeitsmaß zwischen Kommentaren gefiltert werden. Dadurch könnten beispielsweise Problembeschreibungen gefunden werden, die der Gutachter aus einem anderen Kommentar kopiert hat.

4. **URLs in Kommentaren:**
   URLs in Kommentaren können ebenfalls ein Indikator dafür sein, dass hier auf eine externe, umfangreichere Erklärung verwiesen wurde.

5. **Einbeziehung der Saros-Entwickler:**
   Um meine fehlenden Erfahrungswerte zu kompensieren, bietet es sich an, die bestehenden Saros-Entwickler einzubeziehen. Ein Mittel, um alle Entwickler gleichzeitig zu erreichen, bietet die bestehende Mailing-Liste.

Da es am einfachsten ist nach der Zeilenlänge von Kommentaren zu filtern, habe ich mich dazu entschieden, zunächst diesen Filter anzuwenden, die resultierende Kommentarmenge einzeln nach Kandidaten durchzugehen, die mir erscheinen, als seien sie Beschreibungen zu Problemen, die häufig auftreten könnten und diese Liste der möglichen Topics mit der Bitte um Rückmeldung an die Mailing-Liste zu schicken.

3.2 Erstellung der Kandidatenliste

Das Filtern nach einer Mindestlänge von 300 Zeichen hat 1938 Kommentare ergeben, von denen ich in etwa die ersten 250 einzeln durchgesehen habe. Viele Kommentare konnte ich überspringen, da es sich um automatisch erstellte Fehlerberichte von Jenkins bzw. Sonarqube handelte. Oftmals handelte es sich zudem lediglich um längere Diskussionen, die jedoch keine nennenswerte Erklärung eines Problems enthielten. Am Ende dieser ersten Durchsicht hatte ich Kommentare mit umfassenden Erklärungen zu 11 möglichen Topics identifiziert, die sowohl projektspezifische Probleme, wie die Bestimmung des Speicherorts von Dateien innerhalb des Saros-Projekts, sowie grundsätzliche Probleme wie Namenskonventionen oder die Lesbarkeit von Funktionen enthalten.
Diese habe ich in einer Liste[10] zusammengefasst und wie geplant an die Mailing-Liste verschickt, um Rückmeldungen von den Saros-Entwicklern zu erhalten, ob sie meine Einschätzung teilen und eventuell von ihren Erfah-

[10]

Gerrit-Change-ID	Comment Info	Problem
1572	Franz Zieris, May 28, 2014 - 3rd overall comment in file	WhenToRebase
2019	Franz Zieris, Nov 15, 2014 - 11th overall comment in change	PerformanceNotABinaryProperty
2196	Franz Zieris, May 27, 2015 - 10th overall comment in change	CreatingEclipseInstancesForTesting
2891	Stefan Rossbach, Sep 23, 2015 - 19th overall comment in change Franz Zieris, Sep 23, 2015 - 20th overall comment in change	MergingVsCherryPicking
2898	Franz Zieris, Sep 28, 2015 - 9th overall comment in change	IntroducingNonSarosFilesIntoRepo
2777	Franz Zieris, Oct 13, 2015 - 32nd overall comment in change	JavaClassNamingConventions
2983	Franz Zieris, Feb 28, 2016 - 109th overall comment in change	BoyScoutRule
2945	Denis Washington, Nov 11, 2015 - 5th overall comment in change Denis Washington, Nov 11, 2015 - 7th overall comment in change	FunctionReadibility
3027	Stefan Rossbach, Feb 24, 2015 - 30th overall comment in change Matthias Bohnstedt, Feb 24, 2015 - 31st overall comment in change	DesignFlaws
2749	several inline comments	NamingConventions
1220	Franz Zieris, Nov 29, 2013 - 6th overall comment in file Stefan Rossbach, Nov 30, 2013 - 8th overall comment in file	CommentWhyNotWhat

rungen berichten können, ob diese Topics tatsächlich thematisiert wurden oder nicht.

3.3 Untersuchung eines konkreten Problems

In der Zwischenzeit habe ich einen dieser Topic-Kandidaten stellvertretend dahingehend untersucht, ob er mehrmals in Kommentaren thematisiert wurde und um damit ein konkretes Beispiel zu nennen, bei dem eine Automatisierung den Aufwand für die Gutachter hätte reduzieren können. Dafür habe ich eine Erklärung der Funktion von Kommentaren gewählt, die besagt, dass in Kommentaren nicht beschrieben werden soll, was der Code konkret macht, da dies aus dem Code selbst hervorgeht und auch nicht, warum der Code das tut, da dies technische Gründe hat. Vielmehr sollte beschrieben werden, warum der Autor wollte, dass der Code dies tut, also die Frage nach der Intention des Autors, da dies meist nicht am Code selbst erkennbar ist. Diese Erklärung hat sich angeboten, da nach den Schlüsselwörtern „why“ und „what“ gesucht werden konnte.
Bei der Durchsicht der Kommentare, die diese beiden Schlüsselwörter enthalten, sind mir 12 Vorkommnisse aufgefallen, in denen das Topic zumindest erwähnt wurde, wie folgende verkürzte Auflistung der Kommentartexte zeigt (vgl. Abbildung 1 im Anhang):

Nr.	Datum	Kommentartext
1	05.08.2012	„document why, not what“ - „comment why, not what“ (2x)
2	19.09.2012	„comment why, not what“ (2x)
3	04.10.2012	„comment why, not what“
4	17.11.2012	„Always describe what you did (also why if this is necessary)“
5	26.11.2012	„[...] If necessary state the thoughts behind your changes (”why”, not ”what”).“
6	14.03.2013	„document WHY not WHAT“ (2x)
7	16.04.2013	„[...] Given enough time I can figure out what you changed but regardless of how much time I spend I will not come up with an answer to ”why”. So why is a complete make over necessary? Which specific issues are you addressing? Why is this the best change?“
8	13.09.2013	„I think the whole section could use some comments.“ (Christoph Krüger) „Normally you should only comment why and not what. If you must comment what the code is doing indicates that it is to hard to understand and should be refactored. [...]“ (Stefan Rossbach)
9	23.09.2013	„Comment WHY not what !“ (Stefan Rossbach) „What about Why and What?“ (Richard Möhn)
10	29.11.2013	„[...] But a word on comments (and I quote Stefan and probably many others): Comment WHY, not WHAT. Rationale: The WHAT is, in most cases, directly understandable from the code itself. It tells you what it does. The WHY is the intention of doing so. Not every piece of code is suited to explain this. This is when you should write (or request) a comment.“
11	26.07.2014	„Explain why and not what (again, do not repeat the code)“
12	10.06.2015	„ [...] Your comments only explain what you do but not WHY you do it in this way. Nobody will see the optimization. A normal developer will ’optimize’ this back because it is not obvious why the code is such complicated for any reason unless he/she is reading the history of this file. [...]„

Dies zeigt zunächst, dass mindestens 12 Fehler aufgetreten sind, die diesem Topic zuzuordnen sind. Folgende Problematiken lassen sich anhand dieses Beispiels anschaulich zeigen:

1. **Mehraufwand durch erneutes Verfassen der Erklärungen:**
   Das Topic wurde in den Kommentaren 7, 8, 10 und 12 mehrfach ausführlich beschrieben. Die gleiche Aufgabe viermal zu lösen stellt dabei einen unnötigen Mehraufwand dar.

2. **Erwähnen statt Erklären eines Topics:**
   Die restlichen acht Kommentare erklären das Topic entweder nur sehr verkürzt oder erwähnen es sogar lediglich. Dies ist für den Adressaten jedoch nur dann hilfreich, wenn er grundsätzlich mit der Problematik vertraut ist und nur übersehen hat, dass diese hier vorliegt. In diesem Fall bedeutet dies, dass er grundsätzlich weiß, welche Funktion Kommentare erfüllen, er den Sinn der Phrase „Kommentiere warum, nicht was!“ also versteht, ihm aber nur nicht klar war, dass sein Code-Kommentar diesen Anforderungen nicht genügt. Dass jedoch nicht allen Entwicklern diese Nennung des Topics ausreicht zeigt Kommentar 9, in dem nachgefragt wird, was genau mit dieser Phrase gemeint ist. In diesem Fall muss der Autor entweder nachfragen und auf eine ausführlichere Antwort hoffen, was beiden Seiten Zeit kostet, oder aber selbst recherchieren, was es mit diesem Topic auf sich hat und wie es zu lösen ist, was ebenfalls Zeit kostet.

Dieses Beispiel zeigt demnach, dass im Saros-Projekt ein konkreter Bedarf an einer Automatisierung im Bereich des Review-Prozesses besteht.

3.4 Analyse des bestehenden Bedarfs

Fraglich ist jedoch, wie diese Automatisierung aussehen sollte. Optimal erscheint hier zunächst eine Vollautomatisierung, da diese lediglich in der Konfiguration menschlichen Aufwand erfordert und somit sehr effizient läuft. Einige wiederholte Kommentare ließen sich damit sicherlich verhindern. Vorstellbar wäre beispielsweise eine Prüfung auf ausschließlich US-englische Wörter in den Code-Kommentaren mithilfe eines Wörterbuchs. Die statische Code-Analyse, die von Sonarqube bereitgestellt wird, sowie die Überprüfung der Einhaltung von grundlegenden Architektur-Regeln durch Archnemesis stellen dabei ebenfalls Beispiele für eine Vollautomatisierung dar. Für das oben untersuchte Topic der „Funktion von Code-Kommentaren“ erscheint eine Vollautomatisierung jedoch sehr schwierig. Wie soll automatisiert festgestellt werden, ob ein Code-Kommentar die korrekte Intention seines Autors enthält? Dies zeigt den großen Nachteil der Vollautomatisierung: Sie ist nur auf eine begrenzte Art von Fällen anwendbar. Vorliegend soll jedoch eine Lösung gefunden werden, die die Gutachter bei der Erklärung von allen vorstellbaren Topics unterstützt. Daher erscheint eine Teilautomatisierung, beispielsweise in Form einer Wissensdatenbank, die alle bereits verfassten Erklärungen enthält und von den Gutachtern leicht referenziert werden kann, als eine geeignete Lösung. Diese Teilautomatisierung besitzt

zwar nicht die Effizienz einer Vollautomatisierung, da die Benutzung noch immer einen Aufwand für den Gutachter darstellt, auch wenn dieser Aufwand eventuell stark reduziert ist, jedoch ist sie auf alle Fälle anwendbar und kann damit für die Topics genutzt werden, für die noch keine Vollautomatisierung im Einsatz ist. Weiterhin würde eine Teilautomatisierung auch die Probleme des oben untersuchten Topics der „Funktion von Code-Kommentaren“ lösen:

1. **Mehraufwand durch erneutes Verfassen der Erklärungen:**
   Hier müsste nur beim ersten Auftreten eines Fehlers eine neue Erklärung für dieses Topic geschrieben werden. Alle Erklärungen für darauffolgende Fehler könnten die bestehende Erklärung nutzen, wodurch sich der Aufwand für die Gutachter, die diese darauffolgenden Fehler entdecken, deutlich reduzieren würde.

2. **Erwähnen statt Erklären eines Topics:**
   Da in diesen Fällen der Aufwand für die Gutachter bereits minimal war, lässt sich dieser Aufwand nicht weiter reduzieren. Dagegen wird es jedoch möglich, bei minimalem Mehraufwand für den Gutachter, in Form einer Referenzierung einer bestehenden Erklärung, dem Autor eine umfassende Erklärung des Topics zu präsentieren.

3.5 Fazit

Der Bedarf für eine Teilautomatisierung im Bereich der Code-Reviews, beispielsweise durch eine Wissensdatenbank, ist somit im Saros-Projekt konkret gegeben und deren Entwicklung und Implementierung daher gerechtfertigt. Somit wurde das erste Ziel der Bedarfsanalyse erfüllt. Weiterhin hatte ich nach der in diesem Kapitel beschriebenen Analyse das Gefühl den Bedarf im Saros-Projekt gut genug verstanden zu haben, um die im nächsten Kapitel folgende Anforderungsanalyse durchzuführen, wodurch auch das zweite Ziel der Bedarfsanalyse erfüllt ist.
Aus Zeitgründen nicht erfüllt wurde das dritte Ziel, nämlich das Sammeln eines Grundbestands von Topics. Dies muss dementsprechend noch nachgeholt werden, beispielsweise durch erneute Durchsicht der Gerrit-Kommentare nach der Anwendung der oben genannten Filter, sowie der Auswertung der bestehenden Rückmeldungen zu der in 3.2 verschickten Kandidatenliste und das Einholen von weiterem Feedback von Saros-Entwicklern. Besonders geeignet für diesen Teil der Aufgabe scheint zudem die Webseite zu sein, da die hier gesammelten Topics bereits aufgearbeitet und mit Lösungen versehen sind, sodass sie ohne großen Aufwand übertragen werden können.
Rückblickend kann ich zudem sagen, dass zwar das Filtern nach der Kommentarlänge an sich sehr schnell ging, das anschließende Durchsuchen von vielen langen Kommentaren, die zum großen Teil keine gesuchten Informationen enthielten, jedoch sehr viel Zeit gekostet hat. Dementsprechend wäre

es vermutlich ratsam gewesen, eine der genannten Alternativen, beispielsweise die Suche nach URLs in Kommentaren zu verwenden und somit mehr Zeit in die Erstellung des Filters zu investieren, um durch eine höhere Trefferquote weniger Zeit bei der Durchsicht zu verlieren.

4 Anforderungsanalyse

In diesem Kapitel werden die Anforderungen an die zu entwickelnde Anwendung erarbeitet. Dies geschieht durch das Erstellen von Anwendungsfällen, die zugleich die ersten funktionalen Anforderungen darstellen. Abschließend werden noch weitere Anforderungen thematisiert.

4.1 Erstellung von Anwendungsfällen

Um festzustellen, welche Anforderungen an eine Lösung bestehen, die den im vorherigen Kapitel beschriebenen Bedarf an einer Teilautomatisierung des Code-Review-Prozesses im Saros-Projekt deckt, habe ich zunächst versucht, in Form von Anwendungsfällen zu beschreiben, wie der Code-Review-Prozess künftig aussehen soll.

4.1.1 Anwendungsfälle des bestehenden Code-Review-Prozesses

Die ersten von mir geschriebenen Anwendungsfälle waren jedoch viel zu ausführlich und gingen auf zu viele Details ein, was in einer hohen Anzahl von Anwendungsfällen resultierte, die dadurch nur schwer zu überblicken waren. Nach mehreren Gesprächen mit Franz Zieris, der mich an seinen Erfahrungen als Gutachter im Saros-Projekt hat teilhaben lassen, und mehreren Iterationen an Anwendungsfällen bin ich dazu übergegangen, mit einem großen Anwendungsfall zu starten, der den gesamten Code-Review-Prozess aus Sicht eines Gutachters beschreibt, und anschließend detailliertere Anwendungsfälle für die Schritte zu schreiben, die durch eine Teilautomatisierung verändert werden würden. Dadurch soll klar werden, welche Bereiche des bisherigen Review-Prozesses verändert werden sollen und welche unberührt bleiben. Dies hat folgenden großen Anwendungsfall ergeben:

- **Titel:**
  Ein Gutachten erstellen
- **Identifikator:**
  A1
- **Primärer Akteur:**
  Gutachter
- **Geltungsbereich:**
  Gerrit
- **Ebene:**
  Benutzerziel
- **Interessengruppen und deren Interessen:**

  - Gutachter:
    * Möchte die Codequalität des Saros-Projekts erhalten und Autoren helfen, indem er ihren Code begutachtet.
    * Möchte sein Wissen und seine Erfahrungen teilen.
    * Möchte nicht mehr Zeit als nötig für das Gutachten verwenden.
  - Autor:
    * Möchte hilfreiche und konstruktive Rückmeldungen zu seinem Code erhalten, damit er aus seinen Fehlern lernen und diese berichtigen kann
    * Möchte, dass sein Code so schnell wie möglich begutachtet wird, damit er weiter daran arbeiten kann

- **Vorbedingung:**
  Autor hat einen „Change“ bzw. ein neues „Patch Set“ in Gerrit hochgeladen

- **Auslöser:**
  Gutachter bemerkt offenen Change in Gerrit und entscheidet sich dazu, diesen zu begutachten

- **Haupterfolgsszenario:**

  1. Gutachter öffnet Change in Gerrit, um ihn zu begutachten.
  2. Gutachter inspiziert Code und/oder führt den Code aus.
  3. Gutachter findet einen Fehler.
  4. Gutachter schreibt einen Kommentar, der die ausführliche Beschreibung des Fehlers und eine entsprechende Lösung enthält. (siehe Anwendungsfall A1a)
  5. Gutachter wiederholt Schritte 2-4, bis es keine unkommentierten Fehler mehr gibt.
  6. Gutachter schließt das Gutachten ab.

Für eine Teilautomatisierung interessant ist dabei lediglich Schritt 4, das Verfassen eines Kommentars in Gerrit während einer Code-Review, weshalb ich diesen Schritt ebenfalls in Form eines eigenen Anwendungsfalls dargestellt habe:

- **Titel:**
  Einen Kommentar in einer Gerrit-Review schreiben

- **Identifikator:**
  A1a

- **Primärer Akteur:**
  Gutachter
- **Geltungsbereich:**
  Gerrit, sowie neues System mit Bestand an Erklärungen
- **Ebene:**
  Unterfunktion
- **Interessengruppen und deren Interessen:**
  siehe Anwendungsfall A1
- **Vorbedingung:**
  Gutachter befindet sich in einer Code-Review in Gerrit.
- **Auslöser:**
  Gutachter findet einen Fehler in dem Code, den er begutachtet.
- **Haupterfolgsszenario:**
  1. Gutachter wählt Erklärung für gefundenen Fehler aus Bestand an Erklärungen als Vorlage für den Kommentar. (optional – siehe Anwendungsfall A1a1)
  2. Gutachter erklärt den gefundenen Fehler bzw. passt die in Schritt 1 gewählte Vorlage gegebenenfalls an.
  3. Gutachter reicht die in Schritt 2 geschriebene Erklärung für eine Aufnahme in den Bestand an Erklärungen ein. (optional – siehe Anwendungsfall A1a2)
  4. Gutachter speichert Kommentar.

Anwendungsfall A1a ist im Gegensatz zu A1 im derzeitigen Zustand des Code-Review-Systems nicht durchführbar, was bedeutet, dass hier in Form der Schritte 1 und 3 erstmals neue funktionale Anforderungen an die zu entwickelnde Teilautomatisierung gestellt werden. Wichtig ist hier anzumerken, dass beide neuen Schritte optional sind, sodass die Gutachter auch weiterhin so arbeiten können wie bisher und die neue Teilautomatisierung nicht nutzen müssen.

4.1.2 Neue Anwendungsfälle für Code-Reviews in Gerrit

Schritt 1 des Anwendungsfalls A1a erforderte das Schreiben eines neuen Anwendungsfalls, der zugleich die erste funktionale Anforderung an die Teilautomatisierung konkretisiert:

- **Titel:**
  Erklärung eines Problems als Vorlage aus Bestand an Erklärungen bekommen

- **Identifikator:**
  A1a1
- **Primärer Akteur:**
  Gutachter
- **Geltungsbereich:**
  Gerrit, sowie neues System mit Bestand an Erklärungen
- **Ebene:**
  Unterfunktion
- **Interessengruppen und deren Interessen:**
  siehe Anwendungsfall A1
- **Vorbedingung:**
  Gutachter schreibt einen Kommentar als Teil einer Code-Review in Gerrit.
- **Auslöser:**
  Gutachter klickt einen Button in der Benutzeroberfläche von Gerrit an, um eine bestehende Erklärung als Vorlage auszuwählen.
- **Haupterfolgsszenario:**
  1. Gutachter sucht nach dem passenden Topic im Grundbestand von Erklärungen.
  2. Gutachter wählt korrektes Topic aus.
  3. System fügt Erklärung des gewählten Topics in Kommentarfeld ein und speichert einen Verweis auf diesen Kommentar für dieses Topic.

Auch Schritt 3 des Anwendungsfalls A1a erforderte das Schreiben eines eigenen Anwendungsfalls:

- **Titel:**
  Einreichung einer Erklärung zur Aufnahme in den Bestand von Erklärungen
- **Identifikator:**
  A1a2
- **Primärer Akteur:**
  Gutachter
- **Geltungsbereich:**
  Gerrit, sowie neues System mit Bestand an Erklärungen

- **Ebene:**
  Unterfunktion
- **Interessengruppen und deren Interessen:**
  siehe Anwendungsfall A1
- **Vorbedingung:**
  Gutachter hat einen Kommentar als Teil einer Code-Review in Gerrit verfasst.
- **Auslöser:**
  Gutachter klickt einen Button in der Benutzeroberfläche von Gerrit an, um seinen geschriebenen Kommentar in den Bestand von Erklärungen aufzunehmen.
- **Haupterfolgsszenario:**
  1. Gutachter sucht nach dem passenden Topic im Bestand von Erklärungen.
  2. Gutachter wählt korrektes Topic aus.
  3. Gutachter reicht seinen geschriebenen Kommentar zur Durchsicht von einem Redakteur mit dem Ziel ein, die in dem Kommentar enthaltene Erklärung für das ausgewählte Topic in den Bestand von Erklärungen aufzunehmen.

Anwendungsfall A1a2 erwähnt erstmals die neue Rolle des Redakteurs. Diese Rolle habe ich in erster Linie eingeführt, um Versionskonflikte zu lösen, ohne die Rolle des Gutachters damit zu belasten. Problematisch war nämlich, wie damit umgegangen werden sollte, wenn ein Gutachter eine neue Version einer Erklärung für ein Topic speichern wollte. Sollte immer automatisch die neuste Version genommen werden? Dann könnte ein flüchtig geschriebener Eintrag eine aufwändig geschriebene Version ersetzen. Sollte der Gutachter bei der versuchten Einreichung der neuen Version beide zusammenfügen müssen? Das würde ihn zusätzlich belasten und damit dem Ziel widersprechen, seinen Aufwand zu reduzieren. Außerdem müsste er in beiden Fällen die Formulierungen anpassen, da der Kommentartext auf den Autor des begutachteten Codes und seinen begangenen Fehler zugeschnitten sein sollte, während die Erklärung im Bestand der Erklärungen losgelöst vom konkreten Fehler auf das dahinterstehende Topic eingehen sollte. Daher habe ich den Bedarf für die neue Rolle des Redakteurs ausgemacht.

4.2 Funktionale Anforderungen an das neue System

Um den in Anwendungsfällen A1a, A1a1 und A1a2 genannten „Bestand an Erklärungen“ nutzen zu können, muss dieser verwaltet und aktuell gehalten

werden können. Dies ergibt folgende funktionale Anforderung an das neue System:

- **Bestand an Erklärungen muss verwaltet werden können (FA1):**
  Es muss Möglichkeiten für die Rolle des Redakteurs geben, Erklärungen zu erstellen, aktualisieren und zu löschen. Außerdem muss er in der Lage sein, eingereichte Entwürfe mit bestehenden Erklärungen zusammenzuführen.

Um mögliche Fehler, die von Code-Autoren begangen und dann im Code-Review-Prozess von Gutachtern aufgedeckt und erklärt werden müssen, schon im Vorfeld zu vermeiden, ergibt sich zudem folgende funktionale Anforderung:

- **Einträge in der Wissensdatenbank müssen eingesehen werden können (FA2):**
  Alle Projektmitglieder müssen außerhalb des Code-Review-Prozesses lesend auf den Bestand der Erklärungen zugreifen können, um sich über die Topics informieren zu können.

4.3 Qualitätsanforderungen

Weiterhin ergeben sich folgende Anforderungen an die Qualitätsmerkmale nach ISO / IEC 9126.

4.3.1 Funktionalität

Zusätzlich zur Einführung der durch die Anwendungsfälle A1a1, A1a2, sowie den funktionalen Anforderungen FA1 und FA2 beschriebenen neuen Funktion soll die bestehende Funktionalität komplett erhalten bleiben. Dies impliziert, dass der bestehende Review-Prozess in seiner derzeitigen Form ohne Nutzen dieser neuen Funktion auch weiterhin durchführbar sein muss.

4.3.2 Zuverlässigkeit

Die Zuverlässigkeit der Gerrit-Interaktion darf durch die vorgenommenen Änderungen nicht negativ beeinflusst werden. Dies bedeutet, dass ein Ausfall oder eine sonstige Fehlfunktion des neuen Systems bzw. der neuen Funktionen den bisherigen Review-Prozess nicht stören darf.

4.3.3 Benutzbarkeit

- **Bedienbarkeit:**
  Für den Fall, dass die neue Funktion genutzt wird, soll sich die Benutzbarkeit für den Gutachter in der Hinsicht verbessern, dass der

benötigte Aufwand zur Durchführung des Code-Review-Prozesses vermindert wird. Sollte die neue Funktion nicht genutzt werden, so sollte sich der Aufwand für den Gutachter zumindest nicht erhöhen, es also kein Mehraufwand im Vergleich zum Review-Prozess in seiner bisherigen Form entstehen.

- **Verständlichkeit und Erlernbarkeit:**
  Weiterhin sollen dem Gutachter genügend Hinweise gegeben werden, wie diese neue Funktionen zu verwenden sind. Da die bereits beschriebenen Anforderungen zur Funktionalität und Zuverlässigkeit jedoch sicherstellen, dass beim Auftreten von Problemen in diesem Kontext weiterhin auf die derzeitigen, den Anwendern bereits bekannten, Funktionen zurückgegriffen werden kann, ist dieser Punkt nicht allzu kritisch zu bewerten. Da die Funktion des Einsehens von Einträgen auch außerhalb des Review-Prozesses insbesondere für neue Projektmitglieder gedacht ist, damit diese sich zu Beginn ihrer Arbeit im Projekt häufig begangene Fehler anschauen können, um diese selbst zu vermeiden und somit in erster Linie Zeit zu sparen, muss diese Funktion jedoch auf solch eine Art und Weise bereitgestellt werden, dass es den Anwender wenig Zeit kostet, sie zu verstehen und zu benutzen.

4.3.4 Effizienz

Die Effizienz des Review-Prozesses kann sich unter Umständen verschlechtern, da zwar bei Verwendung der neuen Funktion in Form von möglicherweise umfangreicheren Erklärungen mehr geleistet wird, gleichzeitig jedoch eine neue Anwendung betrieben werden muss und somit der Bedarf an technischen Ressourcen steigt. In diesem Fall muss sichergestellt werden, dass dieser Vorteil der eventuellen Mehrleistung und das oben beschriebene Zeitersparnis bei der Bedienung diesen gesteigerten Ressourcenbedarf mehr als rechtfertigen.
Sollte die neue Funktion nicht genutzt werden, entsteht kein Vorteil jeglicher Art, während die neue Anwendung trotzdem betrieben werden muss. Hier darf der Abfall der Effizienz für den Anwender nicht merklich sein.

4.3.5 Wartbarkeit

Um eine Wartbarkeit des Codes gewährleisten zu können, sollten die verschiedenen vorgegebenen[11] Coding-Richtlinien für Saros eingehalten werden, um vor allem Saros-Entwicklern etwaige nötige Änderungen so weit wie möglich zu erleichtern.

[11] http://www.saros-project.org/coderules

4.3.6 Übertragbarkeit

Da das System bei erfolgreichem Einsatz auch auf andere Open-Source-Projekte ausgeweitet werden soll, die ebenfalls einen Code-Review-Prozess betreiben, jedoch eventuell ein anderes Code-Review-Werkzeug benutzen, sollte der Gerrit-spezifische Teil so gering wie möglich gehalten werden, damit bei dieser Übertragung möglichst wenig Code angepasst werden muss.

5 Entwurf

In diesem Kapitel wird eine erste Version des Systems entworfen, das die im vorherigen Kapitel ausgearbeiteten Anforderungen erfüllen soll. Dabei wird zunächst die grobe Struktur vorgestellt und anschließend auf die einzelnen Bestandteile eingegangen.

5.1 Allgemeiner Entwurf

Im vorherigen Kapitel wurde in Form der Anwendungsfälle A1a1, A1a2, sowie der funktionalen Anforderungen FA1 und FA2 herausgearbeitet, dass das zu entwickelnde System folgende Funktionen bereitstellen muss:

- Gutachter im Review-Prozess des Saros-Projekts sollen innerhalb von Gerrit, dem verwendeten Review-Werkzeug, auf einen Bestand von Erklärungen zugreifen können.
- Zudem soll dieser Bestand außerhalb des Review-Prozesses verwaltet und eingesehen werden können.

Als Speicherort des Bestands von Erklärungen bietet sich eine Datenbank an. Da das System zudem leicht auf Open-Source-Projekte, die ebenfalls einen Code-Review-Prozess betreiben, jedoch eventuell ein anderes Review-Werkzeug als Gerrit dafür verwenden, erweiterbar sein soll, empfiehlt sich eine Trennung in einen Gerrit-spezifischen Teil und einen allgemeinen Teil des Systems, wobei der Gerrit-spezifische Teil so klein wie möglich gehalten werden sollte.
Um ermöglichen zu können, dass die Gutachter Gerrit nicht verlassen müssen, um auf den Bestand der Erklärungen zuzugreifen, scheint das Schreiben eines Gerrit-Plugins alternativlos zu sein.
Für die restlichen Aufgaben scheint eine Webanwendung, die alle Datenbankzugriffe übernimmt und über eine REST-API mit dem Gerrit-Plugin kommuniziert, die beste Wahl zu sein, da die Anpassung an andere Review-Werkzeuge nur die Verwendung dieser REST-API erfordert.

5.2 Datenbankschema

Es wird eine Datenbank benötigt, in der Beschreibungen zu allen Topics gespeichert werden können. Dies umfasst allgemeine Hinweise und Verhaltensrichtlinien, die der Fehlervermeidung bzw. Qualitätssicherung dienen, sowie konkrete Probleme und deren mögliche Lösungen. Das Datenbankschema sollte daher eine Tabelle enthalten, in der all diese Einträge gespeichert werden können. Wichtig ist hierbei der im vorherigen Kapitel ausgearbeitete Punkt, dass neue Versionen von Einträgen nicht sofort übernommen, sondern zunächst von einem Redakteur überprüft und mit der bisherigen

Version zusammengefügt werden sollten. Diese neuen, zunächst ungeprüften, Einreichungen könnten ebenfalls in der Tabelle der Einträge gespeichert werden, da die Einreichungen den fertigen Einträgen auf den ersten Blick sehr ähneln. Vorstellbar wäre beispielsweise eine Spalte, die angibt, ob es sich bei diesem Eintrag um eine fertige Erklärung, oder lediglich um eine Einreichung handelt. Dagegen spricht jedoch, dass es durchaus Attribute geben kann, die nur für eine Art der Einträge anwendbar sind. So möchte man beispielsweise für eine fertige Erklärung wissen, welcher Redakteur diese zusammengetragen und überprüft hat. Dieses Feld wäre für Einreichungen immer leer, da es hier noch keinen Redakteur geben kann. Ein weiterer Punkt ist, dass beide Arten von Einträgen vermutlich sehr unterschiedlich behandelt werden. Um die Datenbank übersichtlich zu halten, habe ich mich daher dazu entschieden, diese Einreichungen in einer eigenen Tabelle zu speichern.
Dabei soll die erste Tabelle der fertigen Einträge folgende Spalten enthalten:

- **ID:**
  Dies bezeichnet den technischen Index des Datenbankeintrags.
- **Wiki-Index:**
  Hiermit ist die Identifikation des Topics in Form eines „WikiWords“, beispielsweise „MustReleaseColors“, gemeint.
- **Version:**
  Dies bezeichnet die Version des Topics. Die Kombination aus Wiki-Index und Version sollte dabei einzigartig sein. Durch das Speichern der Versionsnummer soll ermöglicht werden, dass auch veraltete Einträge in der Datenbank erhalten bleiben können.
- **Beschreibung:**
  Hier soll der eigentliche Inhalt des Topics gespeichert werden. Eventuell sollte dieses Feld künftig in mehrere Felder, beispielsweise „Problembeschreibung“, „mögliche Lösungsansätze“ und „Beispiele“ aufgeteilt werden. Um die erste Version des Systems jedoch möglichst simpel zu halten und damit eine Implementierung im Rahmen dieser Arbeit zu ermöglichen, sollen diese Felder zunächst als „Beschreibung“ zusammengefasst werden.
- **Autor:**
  Der Autor der Beschreibung. Dies kann beispielsweise ein Gutachter aus dem Review-Prozess des Saros-Projekts sein, der die Beschreibung für dieses Topic über die Gerrit-Oberfläche eingereicht hat.
- **Verantwortlicher Redakteur:**
  Hiermit ist der Redakteur gemeint, der eine Einreichung überprüft und zusammengeschrieben hat, woraus dieser fertige Eintrag entstanden ist.

- **Kennzeichnung, ob neueste Version des Topics:**
  Dieses abgeleitete Attribut soll einen einfachen Zugriff auf alle aktuellen Versionen der Topics ermöglichen. Es ist für einen Tabelleneintrag „wahr“, wenn kein Tabelleneintrag mit dem gleichen Wiki-Index und einer höheren Version existiert.
- **Datum der Erstellung**
- **Datum der letzten Änderung**

Folgende Spalten sollen in der Tabelle der Einreichungen enthalten sein:

- **ID**
- **ID des fertigen Eintrags, auf den sich bezogen wurde:**
  Hiermit ist die Information verfügbar, auf welches Topic (Wiki-Index) sich die Einreichung bezieht und welches die zum Zeitpunkt der Erstellung der Einreichung aktuelle Version des fertigen Eintrags war.
- **Version**
- **Beschreibung**
- **Autor**
- **Kennzeichnung, ob die Bearbeitung durch einen Redakteur aussteht:**
  Dieses Attribut gibt an, ob die Einreichung bereits bearbeitet wurde oder noch offen ist. Es ist „wahr“, wenn diese Einreichung bereits von einem Redakteur abgelehnt oder in einen fertigen Eintrag zusammengeführt wurde.
- **Datum der Erstellung**
- **Datum der letzten Änderung**

5.3 Webanwendung

Die Kernfunktion der Webanwendung besteht in der Verwaltung der Datenbank. Sie soll zum einen über eine Weboberfläche erreichbar sein, damit sich Saros-Projektmitglieder auch außerhalb des Review-Prozesses über die in der Datenbank gespeicherten Topics informieren und Redakteure Einträge verwalten, sowie ausstehende Einreichungen bearbeiten können. Zum anderen soll sie eine REST-API anbieten, über die insbesondere das Gerrit-Plugin bestehende Einträge abfragen und neue Einträge einreichen kann.

5.4 Gerrit-Plugin

Die Aufgabe des Gerrit-Plugins ist dreiteilig:

1. Bereitstellen der Möglichkeit für den Anwender beim Schreiben eines Kommentars zu signalisieren, dass er auf die Beschreibung eines bestimmten Topics aus der Datenbank zugreifen möchte bzw. dass er diesen Kommentar für ein bestimmtes Topic einreichen möchte.

2. Transformation der Anfrage aus Schritt 1 in eine REST-Anfrage, stellen dieser REST-Anfrage an die Webanwendung und Auswertung der Antwort.

3. Entsprechende Aktualisierung der Gerrit-Benutzeroberfläche, beispielsweise durch das Einfügen der angeforderten Beschreibung in das Gerrit-Kommentarfeld bzw. der Rückmeldung, ob die Einreichung erfolgreich war.

Ich habe zwei mögliche Ansätze gesehen, um diese Aufgaben zu lösen:

1. **Serverseitiges Nutzen des Gerrit-Ereignis-Flusses:**
   Gerrit stellt im Rahmen seiner Plugin-API den sogenannten „Ereignis-Fluss“ bereit, mithilfe dessen man sich über das Eintreten bestimmter Ereignisse, unter anderem der Erstellung bzw. Änderung eines Kommentars, informieren lassen kann.

   Man könnte hier bestimmte Muster vereinbaren, beispielsweise „WikiWord:GET“ zum Anfordern der Beschreibung des Topics mit dem Wiki-Index „WikiWord“ bzw. „WikiWord:POST“ zum Einreichen einer Beschreibung für solch ein Topic. Anschließend können dann alle neuen bzw. geänderten Kommentare auf diese Muster hin untersucht, die Anfragen entsprechend verarbeitet und die Gerrit-Kommentare über die Gerrit-REST-API aktualisiert werden.

   Problematisch ist bei dieser Variante die Aktualisierung der Gerrit-Benutzeroberfläche, da der Anwender erst einmal nicht bemerkt, dass die Kommentare im Hintergrund geändert wurden. Zudem besteht serverseitig keine Möglichkeit, diesen einen Kommentar nachladen zu lassen, weshalb die gesamte Seite neu geladen werden müsste, was zwar den dritten Teil der Aufgabe des Gerrit-Plugins erfüllt, für den Anwender aber nicht sehr komfortabel ist.

2. **Clientseitiges Überwachen der DOM-Elemente:**
   Gerrit stellt außerdem eine umfassende Javascript-API bereit, mit der die einzelnen DOM-Elemente der HTML-Seite, die dem Anwender in

Form der Gerrit-Benutzeroberfläche angezeigt wird, überwacht werden können. Dadurch ist es möglich, auf jede Benutzeraktion zu reagieren, unter anderem auch auf das Speichern eines Kommentars, wodurch der gleiche Ablauf wie beim serverseitigem Nutzen des Gerrit-Ereignis-Flusses realisierbar ist. Zusätzlich kann nach dem Aktualisieren des Gerrit-Kommentar-Objekts über die Gerrit-REST-API auch die Gerrit-Benutzeroberfläche clientseitig nach Belieben manipuliert werden, sodass auch ohne Neuladen der Seite die aktualisierte Form des Kommentars angezeigt werden kann.

Diese Variante erfordert mehr Aufwand, da die Erstellung bzw. Änderung eines Kommentars anhand von DOM-Elementen selbst ausgemacht werden muss, allerdings bietet sie sehr viel mehr Freiraum, wenn die Funktionalität erweitert werden soll, da man zum einen nicht auf die von Gerrit bereitgestellten Ereignisse angewiesen ist und zum anderen zusätzlich zur Verwendung der Gerrit-REST-API die direkte Manipulation der Benutzeroberfläche möglich ist.

Aufgrund des breiteren Spektrums an Möglichkeiten, sowie der Aussicht, die Seite nicht nach jeder Anpassung eines Kommentars neu laden zu müssen, habe ich mich dazu entschieden, die zweite Variante zu implementieren.
Rückblickend muss ich an dieser Stelle jedoch sagen, dass die erste Variante vermutlich die bessere Wahl gewesen wäre, da viele Probleme, die während der Implementierung aufgetreten sind und im nächsten Kapitel genauer beleuchtet werden, bei Verwendung der ersten Variante nicht aufgetreten wären. Der Aufwand, den allein das Überwachen der DOM-Elemente verursacht hat, war auch abgesehen von diesen Problemen deutlich höher als erwartet und schließlich konnte ich nicht einmal das Neuladen der Seite bei Veränderung eines Kommentars durch das Plugin vermeiden, was einer der Hauptgründe war, weshalb ich mich für diese Variante entschieden hatte.

6 Implementierung

In diesem Kapitel werden die größten Probleme thematisiert, die während der prototypischen Implementierung[12] des im vorherigen Kapitel ausgearbeiteten Entwurfs aufgetreten sind.
Während die Implementierung der Datenbank in Form einer MySQL-Instanz und der Webanwendung in Java relativ problemlos verlief, traten beim Erstellen des Javascript-Gerrit-Plugins gleich zwei große Probleme auf.

6.1 Problem im Aufbau des Javascript-Gerrit-Plugins

Das erste dieser Probleme ist eng mit dem Aufbau des Javascript-Gerrit-Plugins verknüpft. Das Ziel des Plugins sollte sein, dass sobald der Anwender einen Kommentar speichert, dieser Kommentar nach den vereinbarten Mustern „WikiWord:GET“ und „WikiWord:POST“ durchsucht wird und diese dem Entwurf aus dem vorherigen Kapitel entsprechend verarbeitet werden. Diese Verarbeitung soll durch eine Funktion geschehen, die jedes mal aufgerufen wird, wenn der Anwender den „Speichern“-Button eines Kommentarfeldes anklickt. Problematisch war hierbei zunächst, dass die „Speichern“-Buttons generisch in dem Moment erzeugt werden, in dem der Anwender ein neues Kommentarfeld öffnet, was bedeutet, dass in diesem Moment auch generisch die „Verarbeitungs-Funktion“ für das Klick-Ereignis dieses Buttons registriert werden muss. Das wurde erreicht, indem alle Stellen des HTML-Dokuments überwacht werden, in denen diese Kommentarfelder entstehen können. Sobald sich dort etwas ändert wird überprüft, ob ein neues Kommentarfeld erzeugt wurde und gegebenenfalls die „Verarbeitungs-Funktion“ für das Klick-Ereignis registriert. Dies ergibt folgenden Ablauf:

1. Anwender ruft beliebige Seite der Gerrit-Weboberfläche auf.
2. Javascript-Plugin prüft per regulärem Ausdruck, ob es auf dieser Seite aktiv werden muss.
3. Wenn ja, werden alle Stellen überwacht, an denen Kommentarfelder generisch erzeugt werden können.
4. Anwender klickt eine Stelle im Code an.
5. Gerrit erstellt automatisch ein Kommentarfeld.
6. Funktion aus Schritt 3 wird aktiv und registriert die „Verarbeitungs-Funktion“ für das Klick-Ereignis
7. Anwender schreibt Kommentar, der ein oben beschriebenes vereinbartes Muster enthält.

[12] https://github.com/saros-project/sia

8. Anwender klickt den „Speichern“-Button.
9. Gerrit speichert Kommentar in der vom Anwender eingegebenen Form in der Gerrit-Datenbank und aktualisiert das HTML-Dokument und damit die Benutzeroberfläche.
10. „Verarbeitungs-Funktion“ wird aufgerufen.

In diesem 10. Schritt soll nun der Kommentar nach den vereinbarten Mustern durchsucht werden, eine entsprechende Kommunikation mit der in dieser Arbeit entwickelten Webanwendung erfolgen und der Kommentar entsprechend angepasst werden. Die Anpassung des Kommentar-Objekts in der Gerrit-Datenbank ist dabei über die angebotene REST-API unproblematisch, anders sieht es bei der Aktualisierung der Benutzeroberfläche aus. Prinzipiell könnte hier einfach der zu diesem Zeitpunkt nicht mehr aktuelle Kommentartext angepasst werden, jedoch benutzt Gerrit eine nicht näher dokumentierte Markdown-ähnliche Formatierung zum Anzeigen seiner Kommentare – die Kommentare werden also anders angezeigt, als sie in der Gerrit-Datenbank gespeichert werden. Da ich keine Möglichkeit gesehen habe, den veränderten Kommentar auf diese Art und Weise zu formatieren ohne die Seite komplett neu zu laden, verursacht nun jede Anpassung eines Kommentars durch das Javascript-Plugin ein Neuladen der Seite. Normale Kommentare, die keine vereinbarten Muster enthalten und bei denen das Javascript-Plugin keine Änderung vornehmen muss, sind davon jedoch nicht betroffen.

6.2 Same-Origin-Policy

Ein anderes Problem ist im Rahmen der Kommunikation zwischen dem Javascript-Plugin und der Webanwendung aufgetreten. Aufgrund der Same-Origin-Policy[13] dürfen in Javascript nur Aufrufe an solche Webseiten erfolgen, die das gleiche Protokoll, den gleichen Host und den gleichen Port verwenden wie die Seite, auf der das Skript eingebettet ist. Selbst wenn die Webanwendung auf dem gleichen Server gehostet wird wie Gerrit, so unterscheidet sich zumindest der Port, weshalb keine direkte Kommunikation möglich ist. Um dieses Problem zu lösen, habe ich das Gerrit-Plugin noch um ein serverseitiges Java-Modul erweitert, dessen Funktionen vom Javascript-Modul aus aufgerufen werden können und das als serverseitiges Modul problemlos mit der Webanwendung kommunizieren kann.

[13] https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

7 Fazit

In diesem Kapitel wird der aktuell erreichte Stand beleuchtet und mit dem verglichen, was im Rahmen dieser Arbeit erreicht werden sollte.

7.1 Bedarfsanalyse des Saros-Projekts

Zunächst sollte in dieser Arbeit

1. untersucht werden, ob im Saros-Projekt ein tatsächlicher Bedarf an einer (Teil-)Automatisierung im Rahmen des Review-Prozesses besteht,
2. dieser in Punkt 1 festgestellte Bedarf so genau verstanden werden, dass daraus Anforderungen an die (Teil-)Automatisierung erarbeitet werden können, sowie
3. ein Grundbestand an Topics gesammelt werden, die (teil-)automatisiert werden sollen.

Im Rahmen der in Kapitel 3 dargestellten Bestandsanalyse konnte nachgewiesen werden, dass im Saros-Projekt ein konkreter Bedarf für eine Teilautomatisierung des Code-Review-Prozesses besteht. Wie genau ich diesen festgestellten Bedarf verstanden habe, lässt sich objektiv schwer messen. Ich habe mir jedoch zugetraut, nach der Untersuchung der bisherigen Gerrit-Kommentare und den Gesprächen mit Franz Zieris ausreichend genaue Anforderungen zu formulieren, auch wenn diese Anforderungen nach dem Testen der prototypischen Implementierung eventuell noch in Zusammenarbeit mit den anderen Saros-Entwicklern angepasst werden müssen.
Ein Ziel, das aus Zeitgründen nicht erreicht wurde ist das Sammeln eines Grundbestands an Topics. Dies möchte ich in den kommenden Wochen noch nachholen, damit die entwickelte Wissensdatenbank bereits zur Einführung vollumfänglich genutzt werden kann und deren Bestand an Einträgen nicht erst natürlich wachsen muss.

7.2 Stand der Implementierung

Die erste Version der Implementierung[14] erfüllt die in Kapitel vier definierten funktionalen Anforderungen mit folgenden Einschränkungen:

- **Anwendungsfall A1a1:**
  Da bisher noch keine Suche implementiert ist, muss der Gutachter wissen, wie der Wiki-Index des gewünschten Topics lautet und diesen in der Form „WikiIndex:GET“ an den Beginn des Kommentars schreiben. Sollte der Eintrag nicht gefunden werden können, wird eine entsprechende Fehlermeldung im Kommentar gespeichert.

[14] https://github.com/saros-project/sia

Weiterhin werden in der Datenbank noch keine Verweise gespeichert, in welchem Kommentaren auf die Topics zugegriffen wurde.

- **Anwendungsfall A1a2:**
  Auch hier gilt, dass der Wiki-Index des gewünschten Topics bekannt sein und per „WikiIndex:POST“ an den Beginn des Kommentars geschrieben werden muss, um diesen einzureichen. Existiert noch kein Topic mit diesem Wiki-Index, wird ein neues Topic angelegt. Der Eintrag muss jedoch trotzdem noch von einem Redakteur überprüft werden.

- **Anforderung FA1:**
  Das Ändern eines fertigen Eintrags ist nur indirekt über das Zusammenführen einer Einreichung möglich. Diese Einreichung kann jedoch nicht nur über das Gerrit-Plugin, sondern auch direkt über die Weboberfläche der Webanwendung geschehen. Zudem können bislang keine Einträge gelöscht werden.

- **Sonstige Einschränkungen:**
  Derzeit werden nur Inline-Kommentare unterstützt, also keine abschließenden Review-Kommentare, die sich auf ein gesamtes „Patch Set“ beziehen.

  Außerdem ist das Gerrit-Plugin derzeit nur aktiv, wenn eine einzelne Version einer Datei angezeigt wird, nicht wenn zwei Versionen miteinander verglichen werden.

Bezogen auf die Qualitätsmerkmale nach ISO / IEC 9126 sieht der Stand wie folgt aus:

- **Funktionalität:**
  Die bisherige Funktionalität sollte nicht eingeschränkt werden. Mit Sicherheit lässt sich dies allerdings erst sagen, wenn das System im Saros-Projekt getestet wurde.

- **Zuverlässigkeit:**
  Gleiches gilt für die Zuverlässigkeit: Ein Ausfall des neuen Systems oder einzelner Komponenten sollten den Ablauf des bisherigen Code-Review-Prozesses nicht behindern. Auch dies muss jedoch noch getestet werden.

- **Benutzbarkeit:**
  Der Aufwand wird für den Gutachter nur dann merklich reduziert werden können, wenn ein angemessen großer Bestand an Topics in der Datenbank enthalten ist, was derzeit noch nicht der Fall ist. Zudem werden derzeit noch keinerlei Hinweise dazu gegeben, wie das neue

System zu benutzen ist. Dies muss in den kommenden Versionen noch erfolgen.

- **Effizienz:**
  Beim Testen des neuen Systems im Saros-Projekt muss insbesondere darauf geachtet werden, ob es zu erhöhten Wartezeiten bei der Benutzung der Gerrit-Benutzeroberfläche kommt, da das Javascript-Modul des neuen Gerrit-Plugins bei jedem Seitenaufruf prüft, ob es aktiv werden muss. Davon abgesehen lassen sich vor den Tests noch keine Aussagen zur Effizienz treffen.

- **Wartbarkeit:**
  Um die gewünschten funktionalen Anforderungen im Rahmen dieser Arbeit zumindest im Kern erfüllen zu können, habe ich die Coding-Richtlinien des Saros-Projekts bisher noch nicht beachtet. Darüber hinaus fehlen sämtliche Maßnahmen zur Qualitätssicherung, wie beispielsweise automatische Tests. Auch dies möchte ich in den kommenden Wochen noch nachholen.

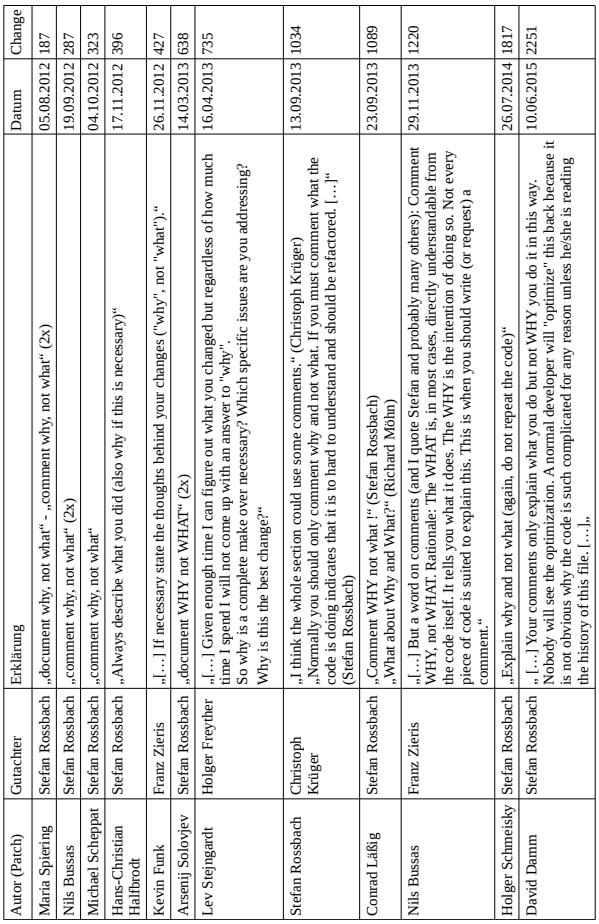
- **Übertragbarkeit:**
  Die Anforderungen zur Übertragbarkeit sind erfüllt, da das Gerrit-Plugin lediglich die Gerrit-spezifischen Funktionen enthält und somit so klein wie möglich gehalten wurde.

Trotz der Funktionsfähigkeit der Implementierung ist noch offen, wie nützlich sie ist und ob sie den Wissenstransfer im Saros-Projekt tatsächlich verbessern kann, da sie dort bisher noch nicht getestet wurde. Auch dies soll in den nächsten Wochen geschehen.

7.3 Bewertung

Der aktuelle Stand der Implementierung bietet meiner Ansicht nach eine gute Grundlage, die nun mit den Saros-Entwicklern zusammen getestet werden kann, um anschließend die Anforderungen gegebenenfalls anpassen und dann komplett erfüllen zu können.
Unzufrieden bin ich rückblickend mit der Entscheidung, ein clientseitiges Javascript-Gerrit-Plugin zu implementieren, da ich die Nachteile im Vorfeld unterschätzt habe und Teile der erwarteten Vorteile am Ende wie im Kapitel sechs beschrieben nicht nutzbar waren.

8 Anhang



Autor (Patch)	Gutachter	Erklärung	Datum	Change
Maria Spiering	Stefan Rossbach	„document why, not what“ - „comment why, not what“ (2x)	05.08.2012	187
Nils Bussas	Stefan Rossbach	„comment why, not what“ (2x)	19.09.2012	287
Michael Scheppat	Stefan Rossbach	„comment why, not what“	04.10.2012	323
Hans-Christian Halfbrodt	Stefan Rossbach	„Always describe what you did (also why if this is necessary)“	17.11.2012	396
Kevin Funk	Franz Zieris	„[…] If necessary state the thoughts behind your changes ("why", not "what").“	26.11.2012	427
Arsenij Solovjev	Stefan Rossbach	„document WHY not WHAT“ (2x)	14.03.2013	638
Lev Steingardt	Holger Freyther	„[…] Given enough time I can figure out what you changed but regardless of how much time I spend I will not come up with an answer to "why". So why is a complete make over necessary? Which specific issues are you addressing? Why is this the best change?“	16.04.2013	735
Stefan Rossbach	Christoph Krüger	„I think the whole section could use some comments.“ (Christoph Krüger) „Normally you should only comment why and not what. If you must comment what the code is doing indicates that it is to hard to understand and should be refactored. […]“ (Stefan Rossbach)	13.09.2013	1034
Conrad Läßig	Stefan Rossbach	„Comment WHY not what !“ (Stefan Rossbach) „What about Why and What?“ (Richard Möhn)	23.09.2013	1089
Nils Bussas	Franz Zieris	„[…] But a word on comments (and I quote Stefan and probably many others): Comment WHY, not WHAT. Rationale: The WHAT is, in most cases, directly understandable from the code itself. It tells you what it does. The WHY is the intention of doing so. Not every piece of code is suited to explain this. This is when you should write (or reques) a comment.“	29.11.2013	1220
Holger Schmeisky	Stefan Rossbach	„Explain why and not what (again, do not repeat the code)“	26.07.2014	1817
David Damm	Stefan Rossbach	„ […] Your comments only explain what you do but not WHY you do it in this way. Nobody will see the optimization. A normal developer will "optimize" this back because it is not obvious why the code is such complicated for any reason unless he/she is reading the history of this file. […],,	10.06.2015	2251

Abbildung 1: Vollständige Liste - Comment ’why’, not ’what’

Literatur

[BB13] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pages 712–721. IEEE Press, 2013.

[CW00] Alistair Cockburn and Laurie Williams. The costs and benefits of pair programming. *Extreme programming examined*, pages 223–247, 2000.

[Fag76] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 15(3):182–211, September 1976.

[Rig12] Peter C Rigby. Open source peer review–lessons and recommendations for closed source. 2012.

[Sol14] Arsenij E. Solovjev. Operationalizing the Architecture of an Agile Software Project. Master's thesis, Freie Universität Berlin, Inst. für Informatik, 2014.