Freie Universität Berlin

Diplomarbeit

Analyse von Defekten in einem großen Softwaresystem durch die Auswertung von Versionsverwaltungssystemen

Alexander Pepper

Betreuer & Gutachter: Prof. Dr. Lutz Prechelt
Zweitgutachter: Prof. Dr. Robert Tolksdorf

Matrikelnummer: 3977280 • pepper@inf.fu-berlin.de

Berlin, 7. November 2011

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.

Berlin, 7. November 2011

Alexander Pepper

Abstract

In dieser Diplomarbeit werden Verfahren des Wissenschaftsbereichs „Software Repository Mining" untersucht, verbessert sowie neu entwickelt, um Daten von Versionsverwaltungssystemen (kurz VCS) und Bugtrackern aufzubereiten und zu analysieren. Als Grundverfahren wird der SZZ-Algorithmus ([11] und [20]) bzw. dessen Implementierung (CVSAnalY mit der Erweiterung `HunkBlame`, [18]) verwendet.

Das Ziel dieser Diplomarbeit ist die Identifizierung und Analyse von Defekten. Dafür werden Bugreports mit ihren Reparaturen im VCS verknüpft sowie die ihnen zugrundeliegenden Bug-Commits identifiziert. Neben der Anwendung und Verbesserung der Verfahren werden diese hinterfragt und deren Ergebnisse untersucht. Bei der Überprüfung der Verfahren, insbesondere der Grundannahmen des SZZ-Algorithmus, zeigen sich gravierende Mängel, die weiteren Forschungsbedarf aufzeigen.

Als Forschungsobjekt dient das VCS des kommerziellen Infopark CMS der Infopark AG. Es enthält 45.000 Commits und seine Historie reicht über zehn Jahre zurück. Neben Vergleichen mit anderen Studien werden neue Ergebnisse präsentiert. Unter anderem zeigte sich überraschenderweise, dass Commits, die wenige Zeilen ändern, defektanfälliger sind als Commits, die viele Zeilen ändern. Außerdem konnte belegt werden, dass Dateien, die häufiger oder von vielen verschiedenen Entwicklern geändert werden, defektanfälliger sind als Dateien, die seltener geändert oder von weniger Entwicklern modifiziert werden.

English

In this diploma thesis, methods of "Software Repository Mining" are studied, improved and newly developed, in order to prepare and analyze data from version control systems (VCS) and bugtrackers. The SZZ algorithm ([11] and [20]), especially its actual implementation (CVSAnalY with the extension `HunkBlame`, [18]), is the basic method used for this research.

Identification and analysis of defects are the main goals of this thesis. Bugreports are linked to their bugfix commits, as well as to there underlying bug commit, in order to allow structured analysis. In addition to using and improving the known methods, the results are scrutinized. In the process the methods (the SZZ algorithm in particular) reveal significant shortcomings. There improvement remains subject to further research.

The VCS of the commercial Infopark CMS by the Infopark AG serves as the object of this study. The repository contains more than 45,000 commits and ten years of development history. In addition to comparing the results to other studies, new results are presented. Surprisingly, a commit is more susceptible to defects, the fewer lines of code are modified. Files that are modified more frequently or that are modified by several different developers, are also more susceptible to defects.

Vorwort und Danksagung

Diese Diplomarbeit entstand 2011 bei der Infopark AG, der ich für die Ermöglichung und Unterstützung dieser Arbeit danken möchte. Das Thema wurde in einer produktiven Diskussion mit Herrn Thomas Ritz (Infopark AG) entwickelt.

Ich möchte besonders Herrn Thomas Witt (Infopark AG) für seine Bereitschaft danken, mir den Quelltext von Infopark CMS als Grundlage dieser Arbeit zur Verfügung zu stellen. Außerdem geht mein Dank an Herrn Prof. Dr. Lutz Prechelt für die Betreuung dieser Diplomarbeit.

Des Weiteren danke ich den verschiedenen Korrekturlesern und Gesprächspartnern für ihre Anmerkungen: Alexander Gessler, Andreas Viebke, Anne Schulz, Cedric Sohrauer, David Császár, Kai-Uwe Humpert, Oliver Fritsch, Sebastian Prestel, Timo Renz, sowie Claudia Pepper und Reinhold Ludwig.

Für ihre vielseitige Unterstützung danke ich besonders meiner Ehefrau Julia Pepper.

Inhaltsverzeichnis

Abbildungsverzeichnis

Seite

Tabellenverzeichnis

Quelltextverzeichnis

Kapitel 1

Einleitung

Das Erkennen und Beheben von Defekten spielt in der Softwareentwicklung eine wichtige Rolle. So erhöht beispielsweise das Vermeiden oder frühzeitige Beseitigen von Defekten in einem Softwareprodukt die Kosteneffizienz des Herstellers. Verschiedene Studien zeigen, dass ein Defekt leichter und kostengünstiger behoben werden kann, je früher dieser in einer Software entdeckt wird ([5] und [13]). Außerdem ist es sinnvoll, Fehlerquellen zu identifizieren, um Defekte und deren Ursprung besser zu verstehen. Aus diesem Verständnis lassen sich Schlüsse für die zukünftige Reduktion von Defekten ziehen.

Wie werden Fehlerquellen identifiziert?

Um ein besseres Verständnis von Defekten zu erzielen, wertet der Wissenschaftsbereich „Software Repository Mining" Versionsverwaltungssysteme (engl. „Version Control System", kurz VCS) aus. In solch einem Repository werden zu jeder Code-Änderung der Entwickler und das Datum der Änderung festgehalten. Sowohl die Änderungen, die zu einem Defekt (*Bug*) geführt haben, als auch die Defekt-Reparatur (*Bugfix*) sind in dem VCS enthalten. Da es sich oft um große Datenmengen handelt, werden Ansätze des Data-Minings verwendet. Neben dem VCS kann ein Bugtracker als Datenquelle verwendet werden, der eine detailliertere Beschreibung des Defekts, den sogenannten *Bugreport*, enthält.

Die folgenden Annahmen werden als Grundlage verwendet, um Fehlerquellen identifizieren zu können:

1. Die Änderung eines Entwicklers (*Commit*) beinhaltet einen Defekt (*Bug-Commit*)[1].
2. Der Defekt wird zu einem späteren Zeitpunkt bemerkt und eine Defektmeldung (*Bugreport*) wird in einem Bugtracker angelegt.
3. Der Defekt wird behoben (*Bugfix-Commit*).

Da Bugfix-Commits im VCS noch nicht als solche ausgewiesen sind, müssen diese zunächst identifiziert werden. Dies geschieht unter Zuhilfenahme der folgenden Datenquellen:

- Commit-Message
- Bugreport-Kommentare

Mit diesen Datenquellen kann eine Verbindung zwischen Bugreport und Bugfix-Commit (*Bugfix-Link*) hergestellt werden. Anschließend wird der Bugfix-Commit dem ursprünglichen Bug-Commit zugeordnet.

1.1 Vorgehen

Aufbauend auf den oben genannten Annahmen existieren im Bereich des „Software

[1] Geänderte sowie neu erstellte Dateien gelten bei Bug-Commits als Änderung.

Repository Mining" bereits einige Erkenntnisse. Diese stammen hauptsächlich aus Analysen von Open-Source-Projekten. In der vorliegenden Diplomarbeit soll überprüft werden, ob sich diese Erkenntnisse auch auf eine kommerzielle Software übertragen lassen. Dabei werden vorhandene Tools aus dem Forschungsgebiet angewendet und verbessert, sowie die zugrunde liegenden Konzepte erweitert.

Fallstudie

Als Forschungsobjekt dient das VCS des kommerziellen Content Management Systems (CMS) der Infopark AG. 1997 begann die Entwicklung am CMS unter dem Namen Network Productivity System (NPS). 2004 wurde NPS in CMS Fiona umbenannt und 2011 in den Infopark Cloud Express als Komponente CMS eingegliedert. Im Folgenden wird die Software Infopark CMS oder NPS genannt. Besonders im deutschsprachigen Raum setzen verschiedene Business-Kunden Infopark CMS ein. Unter anderem werden die Webseiten der Freien Universität Berlin[2], der Max-Planck-Gesellschaft[3] und dem Satellitenbetreiber SES Astra[4] mit diesem CMS betrieben.

In der Entwicklungszeit des CMS wurden mehrere VCS eingesetzt, darunter Concurrent Version System (CVS), Subversion (SVN) und Git. Über 10 Jahre der Entwicklung lassen sich durch die Zusammenführung der verschiedenen VCS rekonstruieren. Sie enthalten mehr als 45.000 Commits.

1.2 Aufbau der Arbeit

Kapitel 2 dokumentiert den Auswahlprozess der Analyse-Software. Dabei werden die verfügbaren Tools aus dem Bereich „Software Repository Mining" vorgestellt und miteinander verglichen. Kapitel 3 beschreibt, wie die Repositorys[5] aus drei verschiedenen Zeitabschnitten der Infopark CMS-Entwicklung zu einem gemeinsamen Repository zusammengeführt werden. Dieses Repository wird daraufhin mittels eines in Kapitel 2 vorgestellten Tools in eine Datenbank überführt. Außerdem wird die Vorbereitung des Bugtrackers für die Analyse beschrieben. Kapitel 4 beschreibt die Identifizierung von Bugfixes und Bugfix-Links. In Kapitel 5 werden die Bug-Commits mit den identifizierten Bugfix-Commits verlinkt.

Die Annahmen dieser Diplomarbeit und der verwendeten Verfahren werden in Kapitel 6 überprüft. Es wird insbesondere untersucht, ob Bug-Commits automatisiert identifiziert werden können. Kapitel 7 enthält die Analysen und Ergebnisse, die anhand der gesammelten Daten möglich sind. Kapitel 8 fasst die Ergebnisse zusammen, zieht ein Fazit und bietet einen Ausblick darauf, welche weiteren Forschungsmöglichkeiten im Bereich des „Software Repository Mining" existieren.

1.3 Verwendete Statistik

In dieser Diplomarbeit werden mehrere Boxplot-Diagramme verwendet. Da verschiedene Darstellungsformen des Boxplots vorhanden sind, wird hier die in dieser Arbeit verwendete Art erklärt.

Abbildung 1.1 zeigt einen Boxplot mit 20 fiktiven Zahlen, die in Tabelle 1.1 aufgelistet sind. Ein Boxplot besteht aus drei vertikalen Linien: das erste Quartil, der Median und das dritte Quartil.

[2] siehe `http://www.infopark.de/fu-berlin` (Stand 01.11.2011)
[3] siehe `http://www.infopark.de/max-planck` (Stand: 01.11.2011)
[4] siehe `http://www.infopark.de/ses-astra` (Stand: 01.11.2011)
[5] Eine konkrete Ausprägung eines Versionsverwaltungssystem (VCS)

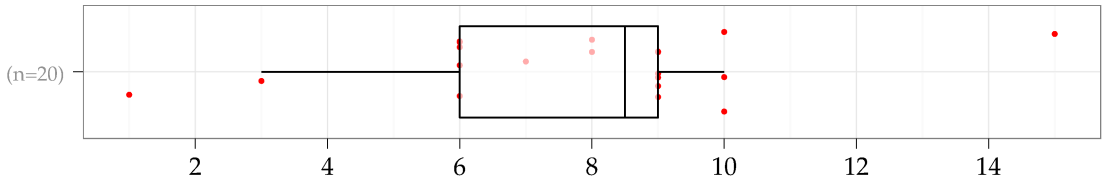

Abb. 1.1: Beispiel für die Verwendung eines Boxplot

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Wert	1	3	6	6	6	6	6	7	8	8	9	9	9	9	9	9	10	10	10	15

Tab. 1.1: Beispielwerte für einen Boxplot

Für die Berechnung der Werte werden diese aufsteigend sortiert. Der Wert, der in der Mitte der sortierten Werte liegt, ist der Median. Falls zwei Werte in der Mitte der sortierten Werte existieren (im aktuellen Beispiel Index 10 und 11), so wird der Mittelwert zwischen diesen beiden Werten als Median verwendet (im aktuellen Beispiel 8,5). Somit sind 50% der Werte größer oder gleich dem Median und 50% der Werte kleiner oder gleich dem Median. Das erste Quartil deckt die ersten 25% der Werte ab. Falls, wie beim Median, zwei Werte existieren, so wird wieder der Mittelwert zwischen den beiden Werten errechnet (im aktuellen Beispiel Index 5 und 6 mit dem Wert 6). Das dritte Quartil deckt die letzten 25% der Werte ab, im Beispiel zwischen Index 15 und 16 (Anzahl - (Anzahl/4)) und Wert 9. Die Differenz zwischen dritten und ersten Quartil wird Interquartilabstand (IQD) genannt. Die beiden langen Linien links und rechts von der Hauptbox werden Antennen genannt. Zur besseren Übersichtlichkeit werden die maximale Länge der Antennen auf 1,5 * IQD begrenzt. Im Beispiel ist der IQD bei (9 - 6 =) 3. Die Antennen sind somit maximal (1,5 * 3 =) 4,5 lang. Sie können jedoch kürzer sein, wenn die Werte in der Gruppe kleiner sind. Im Beispiel reicht die rechte Antenne nur bis 10, obwohl sie bis 13,5 reichen könnte. Es gibt jedoch keine Werte unterhalb von 13,5 die größer als 10 sind, weshalb die Antenne verkürzt ist. Alle Punkte, die außerhalb der Antennen liegen, werden Ausreißer genannt.

Wenn mehr als eine Gruppe miteinander verglichen werden, so werden die Gruppen links beschriftet. In diesem Beispiel gibt es nur eine Gruppe, weshalb sie nicht beschriftet worden ist. Die Boxplot-Diagramme enthalten jedoch immer eine Information, wie groß die zugrundeliegende Datenmenge ist (n=Zahl).

Außerdem wird neben dem Boxplot gleichzeitig ein sogenannter Jitterplot gezeigt. Dabei werden die Werte mit halbtransparenten roten Punkten auf der x-Achse dargestellt. Die y-Achse hat in diesem Fall keine Bedeutung und dient nur dazu, zu erkennen, wenn mehrere Punkte den gleichen Wert haben. Zusätzlich werden die Werte jeweils leicht verschoben, um doppelte Punkte besser sehen zu können. In dem Beispiel kommen die Werte 6, 8, 9 und 10 mehrfach vor.

Falls die Daten über einen großen Zahlenabschnitt verteilt sind und es „große" Ausreißer gibt (Werte, die um einige Größenordnungen größer sind), wird zusätzlich der gleiche Boxplot mit einer logarithmischen Skala dargestellt.

Kapitel 2

Auswahl des Analysetools

Um weitere Analysen und die Handhabung von Commits zu erleichtern, empfiehlt es sich, diese zunächst in eine Datenbank zu überführen. Hierzu werden verschiedene Tools daraufhin untersucht, inwiefern sie sich für den Anwendungsfall diese Diplomarbeit eignen. In Tabelle 2.1 sind alle evaluierten Tools aufgelistet. Sie lassen sich in die Kategorie „Tools von Wissenschaftlern" und „Tools von Softwareentwicklern" einteilen.

2.1 Tools von Wissenschaftlern

Apfel, HATARI und Kenyon

Das Tool Apfel stammt vom Wissenschaftler Thomas Zimmermann. Dieser promovierte an der Universität des Saarlandes unter Prof. Andreas Zeller und arbeitet seit dem bei Microsoft Research[1]. Während seiner Zeit als Doktorand hat er eine Vielzahl von Tools für die Analyse von Repositorys erarbeitet.

Unter anderem hat er 2004 zusammen mit Peter Weißgerber das Tool Apfel entwickelt, welches im Paper „Preprocessing CVS Data for Fine-Grained Analysis" [21] genauer beschrieben wurde. Dabei ging es vor allem darum, die Unzulänglichkeiten von CVS zu umgehen. Diese sind in SVN und Git bereits behoben.

Zusammen mit Jacek Śliwerski und Andreas Zeller stellte er 2005 das Eclipse-Plugin HATARI als Prototyp vor. Dieses warnt innerhalb von Eclipse vor „unsicheren" Code-Stellen, die bereits Defekte enthielten [19]. Im gleichen Jahr wurde das Tool Kenyon unter anderem von E. James Whitehead, Jr. und Sunghun Kim im Paper „Facilitating

Name	Paper	Url (Abruf: 30.06.2011)
Apfel	[21]	`http://www.st.cs.uni-saarland.de/softevo/apfel`
CVSAnalY	[17], [18]	`http://tools.libresoft.es/cvsanaly`
Evolizer	[10]	`http://evolizer.org`
FixCache	[12], [18]	`http://github.com/SoftwareIntrospectionLab/FixCache`
Git Statistics		`http://repo.or.cz/w/git-stats.git`
Gitanaly		`http://bitbucket.org/spaetz/gitanaly`
GitStats		`http://gitstats.sourceforge.net`
Git_mining_tools	[4]	`http://github.com/cabird/git_mining_tools`
Git_stats		`http://gist.github.com/393324`
HATARI	[19]	
Kenyon	[3]	`http://grase.soe.ucsc.edu/kenyon`
LookAtGit		`http://github.com/mpdehaan/lookatgit`
Metior		`http://koraktor.de/metior`

Tab. 2.1: Evaluierte Analysetools

[1] siehe `http://thomas-zimmermann.com/about` (Abruf: 30.06.2011)

Software Evolution Research with Kenyon" [3] als Framework für weitere Analysen vorgestellt. Alle drei Tools sind nicht mehr öffentlich zugänglich.

FixCache

2007 stellte Thomas Zimmermann zusammen mit E. James Whitehead, Jr., Sunghun Kim und Andreas Zeller das Tool FixCache im Paper „Predicting Faults from Cached History" [12] vor. Dieses Tool versucht, anhand von bisherigen Defekten und aktuellen Code-Änderungen 10% der fehleranfälligsten Dateien eines Projektes vorherzusagen. In der ersten Version basierte FixCache auf dem Tool Kenyon. In der zweiten Version wird CVSAnalY als Basis von FixCache verwendet. Diese Version wurde 2011 im Paper „An Empirical Analysis of the FixCache Algorithm" [18] unter anderem von Chris Lewis, Zhongpeng Lin und E. James Whitehead, Jr. vorgestellt.

CVSAnalY

CVSAnalY wiederum ist ein Tool, das ursprünglich 2004 von Gregorio Robles, Alvaro Navarro und Carlos Garcia Campos an der Universität Rey Juan Carlos in Madrid, Spanien entwickelt wurde. Im Paper „Remote analysis and measurement of libre software systems by means of the CVSAnalY tool" [17] von Gregorio Robles et al. wird die Vorgehensweise von CVSAnalY genauer beschrieben.

Die Grundfunktionalität von CVSAnalY ist die Überführung von CVS-Repositorys in eine SQL-Datenbank. Zusätzlich bietet es Erweiterungen an, die weitere Analysen vereinfachen. So können z.B. die geänderten Zeilen pro Commit oder Code-Komplexität pro Datei nach McCabe errechnet werden. Über die Zeit wurde CVSAnalY um Funktionen für SVN und Git erweitert. Für FixCache wurde es von Chris Lewis und Zhongpeng Lin an der University of California, Santa Cruz (UCSC) maßgeblich erweitert[2]. Sowohl FixCache in der zweiten Version als auch die verschiedenen Versionen von CVSAnalY sind quelloffen.

Git_mining_tools

Git_mining_tools ist eine Skript-Sammlung, die für das Paper „The Promises and Perils of Mining Git" [4] entwickelt wurde. Dabei werden, wie bei CVSAnalY, die Commits in eine Datenbank überführt. Darüber geht das Skript jedoch nicht hinaus.

Evolizer

Eine Alternative zu CVSAnalY ist das Tool Evolizer, welches an der Universität Zürich entwickelt wurde. Im 2009 von Harald Gall et al. verfassten Paper „Change Analysis with Evolizer and ChangeDistiller" [10], wird Evolizer ähnlich wie Kenyon als Framework für weitere Analysen beschrieben. Als Eclipse-Plug-in analysiert es sowohl CVS- als auch SVN- und Git-Repositorys und schreibt sie über eine Hibernate-Schnittstelle in eine Datenbank. Um die Daten weiter zu verarbeiten, soll die Datenbank nicht direkt, sondern über die Hibernate-Schnittstelle verwendet werden.

Evolizer enthält die Teilkomponente ChangeDistiller. Als einziges Tool in dieser Auflistung bietet ChangeDistiller die Funktionalität, den abstrakten Syntaxbaum von Java-Quelltext zu vergleichen. Damit ist es möglich, Codeänderungen zu erkennen, die nur auf einer Umformatierung oder Umbenennung von Variablennamen basieren und diese Änderungen für die Defekt-Ursprungssuche zu ignorieren. Jedoch kann ChangeDistiller nur zusammen mit Evolizer verwendet werden.

[2] siehe `http://github.com/SoftwareIntrospectionLab/cvsanaly` (Abruf: 30.06.2011)

2.2 Tools von Softwareentwicklern

Git Statistics

Git Statistics (auch Git-Stats genannt) wurde im Rahmen des Google Summer of Code[3] 2008 von Sverre Rabbelier entwickelt und soll die tägliche Arbeit in Open-Source-Projekten erleichtern[4]. In größeren Open-Source-Projekten gibt es oft verschiedene Code-Bereiche mit verschiedenen Ansprechpartnern. Git Statistics versucht zu ermitteln, wer der beste Ansprechpartner für einen erstellten Patch ist. Außerdem bietet das Command-Line-Tool Informationen, wie z.B. die Anzahl der Commits pro Entwickler oder geänderte Zeilen pro Datei. Es bietet auch Funktionen für die Analyse von Bugs innerhalb eines Repositorys. Beim Testen des Tools sind verschiedene Defekte in der Grundfunktionalität des Tools beobachtet worden, die zu Abbrüchen führen.

GitStats

GitStats ist ein Command Line Tool zum Erstellen von verschiedenen Statistiken über Git-Repositorys als HTML-Seite. Eine Vielzahl an Informationen, wie z.B. die Anzahl der Commits pro Stunde des Tages oder pro Wochentag, sowie detaillierte Informationen über einzelne Entwickler werden sowohl mit Zahlen als auch mit Diagrammen dargestellt. Für die beschriebenen Analysen funktioniert das Tool sehr schnell und zuverlässig. Für Analysen, die über die angebotenen Informationen hinausgehen, ist das Tool ungeeignet, da es die Ergebnisse nur als HTML-Seiten präsentiert.

Gitanaly, Git_stats und LookAtGit

Gitanaly, Git_stats und LookAtGit sind kleinere Skripte, die jeweils für eine bestimmte Aufgabe entwickelt wurden. Gitanaly analysiert nur Open-Office- bzw. Libre-Office-Repositorys. Git_stats[5] (bzw. git-stats) und LookAtGit[6] berechnen jeweils die Anzahl der Commits pro Entwickler. Für umfangreiche Analysen sind diese Skripte nicht geeignet.

Metior

Das jüngste Tool in dieser Auflistung ist Metior, da es erst seit März 2011 entwickelt wird. Metior bietet eine Ruby-API an, um Zugriffe auf Git-Repositorys zu vereinfachen. Damit ist es z.B. möglich, die Anzahl der geänderten Zeilen pro Commit zu berechnen. Über diese Basis-Funktionalität geht dieses Skript noch nicht hinaus.

2.3 Auswahl-Ergebnis

Nach eingehender Begutachtung der aufgezählten Tools bezüglich ihrer Tauglichkeit und Eignung für diese Arbeit wurde CVSAnalY ausgewählt.

Prinzipiell ausgeschlossen sind alle Tools, die nicht oder nicht mehr öffentlich zugänglich sind (Apfel, HATARI und Kenyon). Kleinere Tools (git_mining_tools, Gitanaly, git_stats und LookAtGit) sind für umfangreiche Analysen untauglich. Git Statistics ist unbrauchbar, da selbst die Grundfunktionen bereits fehleranfällig sind.

Metior hat viel Potenzial und ist eventuell für zukünftige Forschungen verwendbar.

[3] mehr Informationen unter `http://code.google.com/soc` (Abruf: 14.07.2011)
[4] siehe `http://sites.google.com/site/alturin2/gsoc2008` (Abruf: 30.06.2011)
[5] siehe `http://grosser.it/2009/11/14/what-are-your-git-stats` (Abruf: 30.06.2011)
[6] siehe `http://michaeldehaan.net/2010/01/01/git-statistics-simpler-faster` (Abruf: 30.06.2011)

Da es sich jedoch in einem frühen Entwicklungsstadium befindet, ist es für diese Arbeit derzeit noch ungeeignet. GitStats bietet bereits eine schnelle und umfangreiche Analyse eines Repositorys als HTML-Seite an. Es ist jedoch nicht möglich, diese Informationen weiter zu aggregieren und somit für umfangreiche Analysen ungeeignet.

Evolizer wäre als Tool geeignet, hat jedoch den Nachteil, dass per Hibernate-Schnittstelle mit der Datenbank kommuniziert werden muss. Besonders die Möglichkeit der Teilkomponente ChangeDistiller, den abstrakten Syntaxbaum zu verarbeiten, ist ein Alleinstellungsmerkmal. Diese Möglichkeit ist bisher jedoch auf Java beschränkt.

Das ausgewählte CVSAnalY ist quelloffen, ermöglicht umfangreiche Analysen und ist gut erweiterbar. Außerdem bietet es den Vorteil, dass bereits mehrere Erweiterungen für die Analyse von Repositorys zur Verfügung stehen. Besonders die Version der UCSC[7] bietet mit den Erweiterungen `Hunks` und `HunkBlame` eine Implementierung des Algorithmus von Śliwerski, Zimmermann und Zeller (SZZ-Algorithmus) an (siehe Kapitel 5). Neben den umfangreichen Funktionen ist es relativ leicht, mit den Entwicklern via E-Mail[8], Internet Relay Chat (IRC)[9] oder Issue-Tracker[10] in Kontakt zu treten. Auch deshalb ist dieses Tool gut für diese Arbeit geeignet.

[7] siehe `http://github.com/SoftwareIntrospectionLab/cvsanaly` (Abruf: 30.06.2011)
[8] siehe Mailingliste libresoft-tools-devel@lists.morfeo-project.org
[9] siehe Kanal `#libresoft` im freenode IRC Netzwerk
[10] siehe `http://github.com/SoftwareIntrospectionLab/cvsanaly/issues` (Abruf: 30.06.2011)

Kapitel 3

Datenbasis

In diesem Kapitel werden die Schritte dokumentiert, die zur Schaffung einer Datenbasis für weitere Analysen benötigt werden. Für jeden Zwischenschritt werden zuerst die zugrundeliegenden **Konzepte** vorgestellt, daraufhin die **Anwendung** dieser Konzepte im konkreten Fall erörtert und zuletzt die **Datenqualität** beleuchtet.

3.1 Vorbereitung des Repositorys

Konzepte

Durch den Wandel der Technik sind in den letzten Jahren und Jahrzehnten verschiedene VCS entstanden. Als eines der ersten Systeme wurde Revision Control System (RCS) Anfang der 80er Jahre entwickelt. Dieses konnte jeweils nur eine einzige Datei verwalten. CVS versteht sich als direkter Nachfolger von RCS und baut auf dessen Dateiformat auf. CVS erlaubt es, mehrere Dateien in einem Repository zu verwalten. Jede Datei hat jedoch eine individuelle Revisionsnummer, sodass die Zusammengehörigkeit von Änderungen zwischen verschiedenen Dateien untereinander nur sehr schwer wieder hergestellt werden kann. SVN gilt als indirekter Nachfolger von CVS und führte eine globale Revisionsnummer ein. Git, das jüngste Programm in dieser Auflistung, versteht sich als unabhängig von der gerade genannten Konkurrenz. Anders als CVS und SVN liegt bei dem verteilten Versionsverwaltungssystem immer eine lokale Kopie der vollständigen Historie vor.

Im Laufe eines langjährigen Software-Projektes werden oft verschiedene VCS eingesetzt. In einigen Fällen wird die bisherige Historie mit in das neue System übernommen[1]. In den meisten Fällen wird jedoch lediglich der letzte Zustand des alten VCS kopiert, um als Initialzustand des neuen VCS zu dienen. Dadurch ist die bisherige Historie des alten VCS in dem neuen VCS nicht mehr verfügbar.

Die meisten wissenschaftlichen Veröffentlichungen im Bereich „Software Repository Mining" ignorieren frühere VCS und analysieren nur das aktuelle Repository. Für die Analyse und Auswertung von Defekten ist genau diese Historie jedoch wichtig, um ihre Ursachen besser verstehen zu können.

Um im Nachhinein eine vollständige Historie von zwei verschiedenen VCS (z.B. CVS und Git) zu rekonstruieren, können folgende Schritte unternommen werden. Diese Schritte setzen voraus, dass das alte VCS noch zur Verfügung steht.

1. Altes VCS 1 (z.B. CVS) mit seiner Historie in ein neues VCS 1′ (z.B. Git) konvertieren.

2. Ersatz-Commit in VCS 1′ erstellen, der den Endzustand von VCS 1′ dem Initialzustand des neueren VCS 2 anpasst.

[1] Beispiel: Ruby on Rails

3. Das entstandene VCS 1′ mit dem neuen VCS 2 in ein gemeinsames VCS 2′ überführen.

4. Innerhalb vom gemeinsamen VCS 2′, den Initial-Commit des VCS 2 durch Ersatz-Commit in VCS 1′ ersetzen.

Im nächsten Abschnitt folgt eine ausführliche Beschreibung der Anwendung dieser Konzepte auf die VCS von Infopark CMS.

Für die Wahl des gemeinsamen VCS eignet sich Git, da eine Vielzahl an Konvertierungsmöglichkeiten aus anderen VCS existieren[2]. Außerdem werden alle Operationen lokal durchgeführt, sodass keine Kommunikation mit einem externen Server benötigt wird und dadurch der Vorgang beschleunigt werden kann. Zudem bietet Git mit dem Befehl `git replace` die Möglichkeit, Commits innerhalb der Historie zu ersetzten[3].

Anwendung

Infopark hat in seiner langjährigen Entwicklungszeit verschiedene VCS eingesetzt. Angefangen mit CVS im Jahre 2000, wechselte das Unternehmen Ende 2003 zu SVN und zuletzt Mitte 2008 zu dem dezentralen VCS Git. Bei jedem Wechsel wurde der letzte Stand des alten Systems kopiert und als Initialzustand des neuen Systems verwendet. Die Historie des alten Systems wurde dabei jedoch ignoriert. Glücklicherweise sind die Historien von CVS und SVN nicht verloren gegangen, sondern wurden später in zwei separate Git-Repositorys umgewandelt (Namen der Repositorys: `nps_cvs`, `nps_svn` und `nps_git`). Dabei wurden die Befehle `git cvsimport` und `git svn clone` verwendet (siehe Abbildung 3.1).

Die meisten VCS bieten eine Funktion, mit der ermittelt werden kann, welcher Entwickler eine Codezeile einer Datei geschrieben hat und zu welchem Zeitpunkt. In Git heißt diese Funktion `git blame`. Wenn die Historie nicht übernommen wurde, hat das

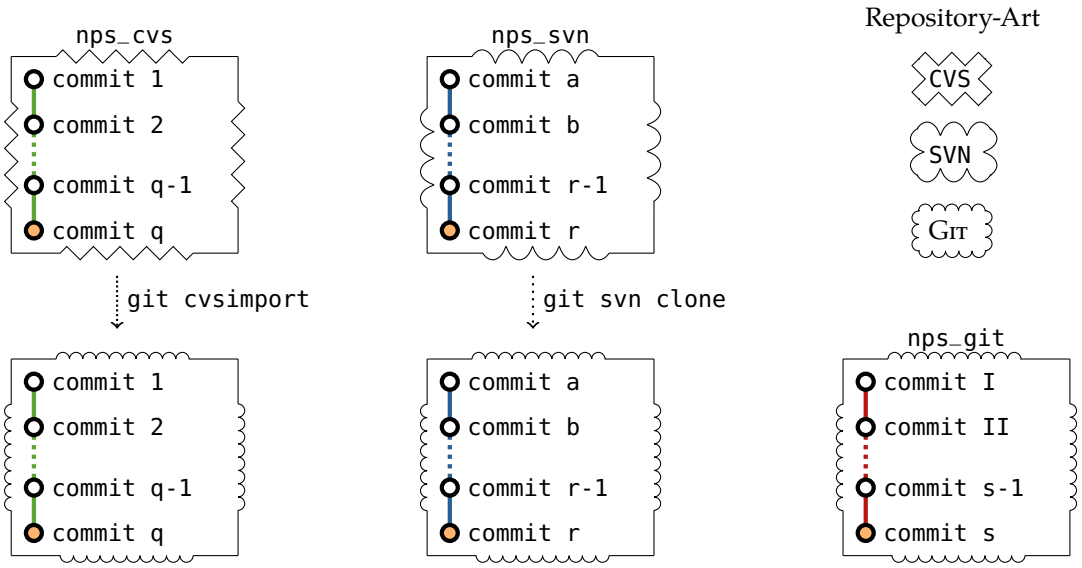

Abb. 3.1: Umwandlung von CVS und SVN in Git-Repositorys

[2] siehe `http://git.wiki.kernel.org/index.php/GitFaq#Importing_from_other_revision_control_systems` (Abruf: 28.06.2011)

[3] Die SHA-Werte der bisherigen Commits werden mit dem Befehl *nicht* geändert. Mehr Infos zu `git replace` siehe [7]

den Nachteil, dass `git blame` nur bis zum Importtag (Initialzustand des aktuellen VCS) reichende Daten liefern kann.

Um dies besser zu verdeutlichen, ist in Quelltext 3.1 der Aufruf von `git blame` auf die Datei `TEDatabaseSchema.m` im Repository `nps_git` aufgeführt. Jede Zeile der Datei wurde demnach am 2008-08-28 um 15:58:49 Uhr vom Entwickler `usera` erstellt. Dies ist genau das Datum des ersten Commits im Repository `nps_git`. Wenn die drei Repositorys `nps_cvs`, `nps_svn` und `nps_git` zu einem gemeinsamen Repository rekonstruiert werden (`nps_full`), sieht der gleiche Befehl aus wie in Quelltext 3.2.

Es ist zu erkennen, dass keine der Zeilen am 2008-08-28 geändert oder erstellt worden sind, sondern zu früheren Zeitpunkten. So ist z.B. die Zeile 4 am 2004-07-23, also vier Jahre vor der Umstellung auf Git, von `userb` anstatt von `usera` erstellt worden. Dies ist besonders für die zeitliche Zuordnung von Änderungen wichtig.

Zur Rekonstruktion des einheitlichen Repositorys `nps_full` aus `nps_cvs`, `nps_svn` und `nps_git` sind verschiedene Schritte notwendig. Die jeweils aufeinander folgenden Git-Repositorys müssen zu einem gemeinsamen Repository verknüpft werden. Beim Übergang zwischen den Einzel-Repositorys muss nachträglich ein verbindender Commit erstellt werden.

Hierfür wird zuerst die Git-Version des ältesten `nps_cvs` als Basis-Repository kopiert. Dort muss ein weiterer Commit zu dem letzten Commit aus `nps_cvs` hinzugefügt werden, um die Änderungen nachzustellen, die zum Initial-Zustand des Repositorys `nps_svn` geführt haben. Dazu wird der letzte Zustand von `nps_cvs` wiederhergestellt (`git checkout`). Daraufhin werden alle Dateien in dem Verzeichnis, bis auf das Verzeichnis `.git/`, gelöscht. Im nächsten Schritt wird in einem separaten Verzeichnis der Initialzustand von `nps_svn` wiederhergestellt (`git checkout`). Zuletzt werden alle Dateien (bis auf das Verzeichnis `.git/`) von `nps_svn` nach `nps_cvs` kopiert.

Git erkennt automatisch, welche Dateien neu erstellt, welche geändert und welche gelöscht wurden. In diesem Fall handelt es sich um diverse `.cvsignore` Dateien, die in SVN nicht mehr benötigt werden. Danach kann mit `git commit -m "commit a'"` ein neuer Commit `a'` erstellt werden. Der daraus resultierende Datei-Zustand entspricht genau dem Datei-Zustand vom ersten Commit des Repositorys `nps_svn` (siehe Abbildung 3.2). Um außerdem den Entwickler und das Datum des Commits anzupassen, kann Quelltext 3.3 verwendet werden. Der Dateizustand, sowie das Datum und der Entwickler der Commits `a` und `a'` unterscheiden sich nur noch darin,

```
> git blame TEDatabaseSchema.m
^758203a usera 2008-08-28 15:58:49  1) #include "TEDatabaseSchema.h"
^758203a usera 2008-08-28 15:58:49  2)
^758203a usera 2008-08-28 15:58:49  3) #pragma .h #include <CMCDatabaseSchema.h>
^758203a usera 2008-08-28 15:58:49  4) #include "Base/TPExportManager.h"
```

Quelltext 3.1: `git blame` *ohne* Historie

```
> git blame TEDatabaseSchema.m
1125c329 usera 2006-04-25 14:48:47  1) #include "TEDatabaseSchema.h"
892d6430 usera 2001-07-02 16:37:35  2)
f0cf9de0 userb 2007-06-22 08:42:36  3) #pragma .h #include <CMCDatabaseSchema.h>
5cc4c7ea userb 2004-07-23 14:02:18  4) #include "Base/TPExportManager.h"
```

Quelltext 3.2: `git blame` *mit* Historie

```
$ git commit --amend --date="Mon Dec 29 16:45:21 2003" --author="usera <usera@domain>"
```

Quelltext 3.3: Einen Commit nachträglich ändern

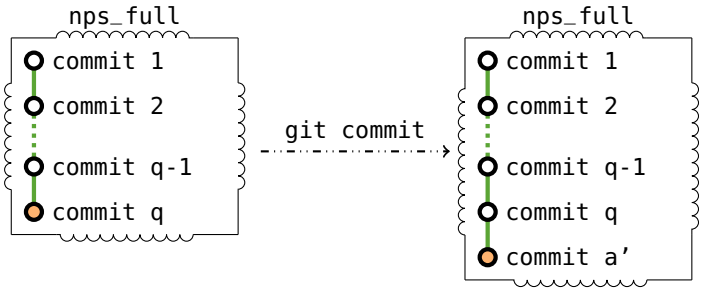

Abb. 3.2: Erweitern von nps_full um commit a'

dass a' mit der bisherigen Historie verbunden ist.

Um auch noch alle weiteren Commits aus nps_svn mit der Historie aus nps_cvs vertraut zu machen, muss nps_svn in nps_full importiert und der Befehl git replace angewendet werden (siehe Abbildung 3.4). Mit git replace wird der commit a durch den commit a' ersetzt. Somit ist der Elternteil von commit b jetzt commit a'. Für alle Commits, die ursprünglich in nps_svn waren, kann jetzt im Repository nps_full die komplette Historie ermittelt werden.

Um zusätzlich die Informationen aus nps_git zu erhalten, muss auch für dieses Repository ein Ersatz-Commit I' erstellt werden, nps_git importiert, und per git replace der Elternteil von Commit II durch I' ersetzt werden. Abbildung 3.3 zeigt das Endprodukt von nps_full nach allen Transformationen.

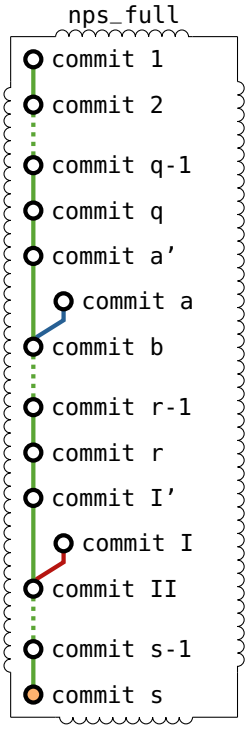

Abb. 3.3: Vollständige Version von nps_full

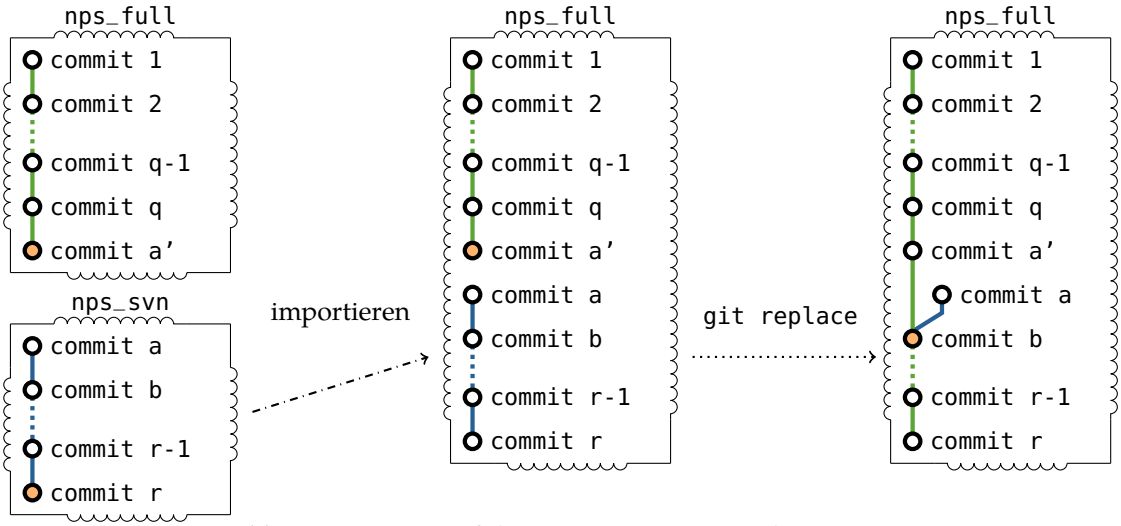

Abb. 3.4: Zusammenführung von nps_svn und nps_full

Mehrere Branches

Zum Endzustand von `nps_cvs` existierte nicht nur ein Branch, der in `nps_svn` weiter entwickelt wurde, sondern drei. Deshalb ist es notwendig, alle Historien-Stränge aus `nps_cvs` zu rekonstruieren. Die ersten beiden Branches sind Bugfix-Branches der damals aktuellen CMS-Versionen (Delphine (2002) und Eliza (2003)). Der dritte Branch ist die Weiterentwicklung für die Nachfolgeversion Fiona. Abbildung 3.5 zeigt schematisch den Endzustand von `nps_cvs`.

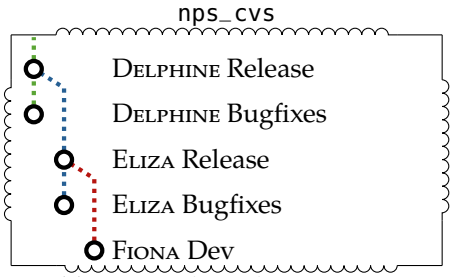

Abb. 3.5: Endstand von `nps_cvs`

Nach `nps_svn` wurden zuerst alle Änderungen von Delphine, daraufhin von Eliza und zuletzt von Fiona importiert. Abbildung 3.6 zeigt den Initialzustand von `nps_svn`.

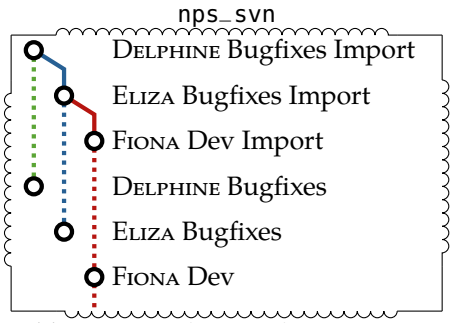

Abb. 3.6: Initialzustand von `nps_svn`

Um die drei Historien-Stränge zusammenzuführen, müssen echte Merges zwischen `nps_cvs` und `nps_svn` erstellt werden (mittels `git merge`, siehe Abbildung 3.7).

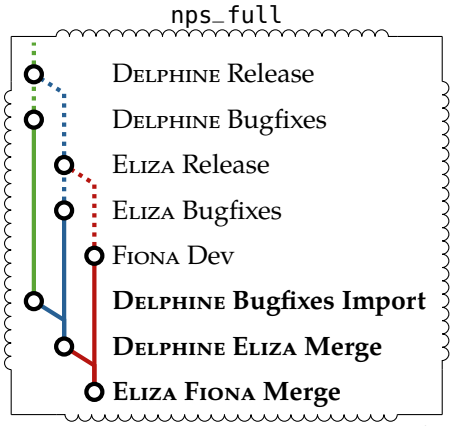

Abb. 3.7: Merge von Delphine, Eliza und Fiona

Daraufhin können wie oben beschrieben mit `git replace` die drei Historien-Stränge jeweils zusammengeführt werden. Abbildung 3.8 zeigt das Resultat dieser Operationen.

Somit konnte die komplette Historie der letzten 11 Jahre von NPS aus drei Repositorys zu einem rekonstruiert werden. Alle weiteren Analysen dieser Arbeit basieren auf `nps_full`, soweit nicht anders angegeben.

Datenqualität

Das Repository `nps_cvs` beinhaltet 25.653, `nps_svn` 14.694 und `nps_git` 5.261 Commits[4]. In der Summe sind das 45.608 Commits. Das zusammengeführte Repository `nps_full` beinhaltet 45.611 Commits, welches den vier zusätzlichen „Ersatz-Commits" und einem gelöschten leeren Initial-Commit aus `nps_svn` entspricht. Daraus folgt, dass durch die Transformation keine Daten verloren gegangen sind.

[4] gemessen mit GitX, siehe `http://github.com/brotherbard/gitx` (Abruf: 30.06.2011)

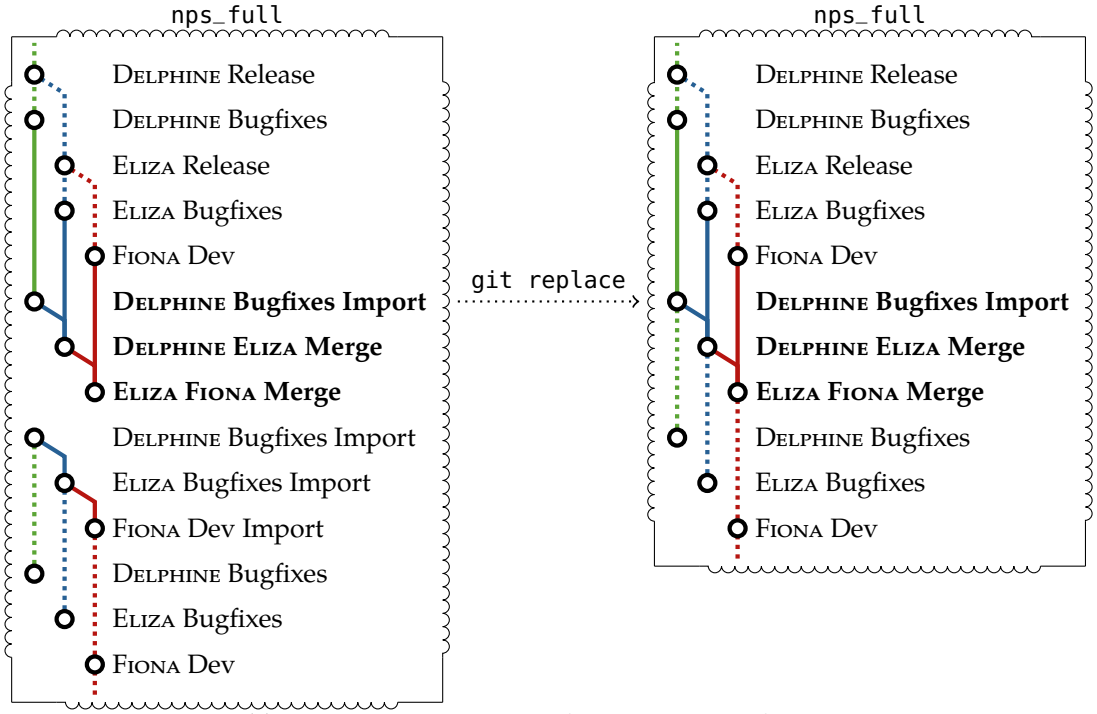

Abb. 3.8: `git replace` zwischen `nps_cvs` und `nps_svn`

3.2 Repository in eine Datenbank überführen

Konzepte

Für umfangreiche Analysen ist es sinnvoll, das Repository in eine für Analysen besser geeignete Struktur zu überführen. Die transformierte Struktur wird in der relationalen Datenbank MySQL gespeichert, mit der komplexe Abfragen möglich sind. Außerdem bieten verschiedene Analyse-Tools (insbesondere R) MySQL Unterstützung an. Um das Repository in MySQL zu überführen, wird das Tool CVSAnalY eingesetzt (siehe Abbildung 3.9, sowie Kapitel 2).

Die Auswahl der Branches ist ein wichtiger Aspekt, der in anderen wissenschaftlichen Arbeiten zu dem Thema kaum oder keine Erwähnung findet. Die meisten Analysen verwenden nur den aktuellen Entwickler- oder Stable-Branch. In der Praxis existieren jedoch meist mehrere Branches. Diese werden entweder für Feature-Entwicklung oder zur Pflege älterer Versionen verwendet. Deshalb ist es wichtig, auch die Pflege-Branches zu analysieren, da sie eine Vielzahl an Bugfixes enthalten.

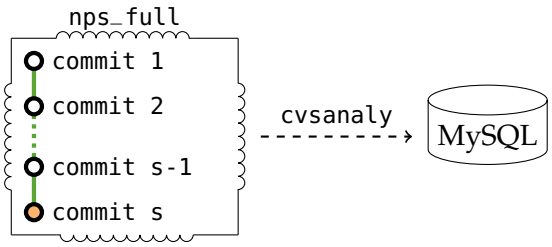

Abb. 3.9: Überführung vom `nps_full` in MySQL mit Hilfe von CVSAnalY

Anwendung

Infopark hat in den betrachteten 11 Jahren mehrere Versionen ihres CMS veröffentlicht. Für diese Diplomarbeit wurden als Branches alle Hauptversionen seit 2002 gewählt (Delphine, Eliza und Fiona in Version 6.0, 6.5, 6.6, 6.7 und 6.8), sowie die aktuelle und noch nicht veröffentlichte Weiterentwicklung Infopark CMS, an der seit Mitte

2009 gearbeitet wird. Vorherige Branches werden ignoriert. Dies ist dadurch begründet, dass das Bugtracking-System Bugzilla erst ab 2003 zum Einsatz kam. Eine Verlinkung zwischen Bugreports und Bugfixes *vor* 2003 ist somit nicht möglich, da keine Bugreports aus der Zeit existieren (siehe Kapitel 4).

Beim Anwenden auf das umfangreiche Repository `nps_full` traten mehrere Probleme auf, die zunächst gelöst werden mussten und im Folgenden beschrieben werden.

Dateipfad Problem

Während der Ausführung von CVSAnalY wurde in großem Umfang Arbeitsspeicher benötigt. So wurden bei Berechnungen teilweise über 20 Gigabyte an Arbeitsspeicher angefordert, welches den ausführenden Rechner überforderte. Nach längerem Debugging, Recherche und Kontakt mit den aktuellen Entwicklern stellte sich heraus, dass der Ursprung der großen Arbeitsspeicher-Anforderungen in den Dateinamen liegt.

Abbildung 3.11 zeigt das bisherige Datenbank-Modell, um Dateinamen in CVSAnalY abzuspeichern.

Dateinamen werden nicht vollständig gespeichert, sondern nur Referenzen auf das jeweilige Überverzeichnis. Beispielsweise wird die Datei `Apps/NPS/EC/install.sh` wie in Tabelle 3.1 abgespeichert.

Datei- und Verzeichnisnamen können sich im Laufe der Zeit durch Umbenennung ändern. Auch können Dateien verschoben werden. CVSAnalY deckt Umbenennungen und Verschiebungen ab, indem es immer die gleiche `file_id` für die gleiche Datei verwendet und die Informationen auf die Tabellen `actions` und `file_copies` verteilt. Um im Nachhinein für einen bestimmten Zeitpunkt (`commit_id`) und eine bestimmte `file_id` den aktuellen Pfad zu ermitteln, müssen somit die Tabellen `files`, `file_links`, `actions` und `file_copies` herangezogen werden. Dies wurde bisher über eine Adjazenzmatrix realisiert, deren Benutzung viel Arbeitsspeicher und Rechenzeit benötigt.

Um diesen Prozess zu beschleunigen, wurde im Rahmen dieser Diplomarbeit die Datenbanktabelle `actions` um das Feld `current_file_path` erweitert (siehe Abbildung 3.10).

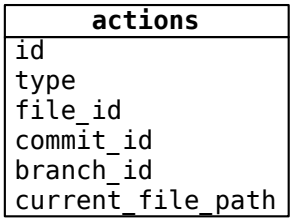

Abb. 3.10: Überarbeitete Tabelle `actions`

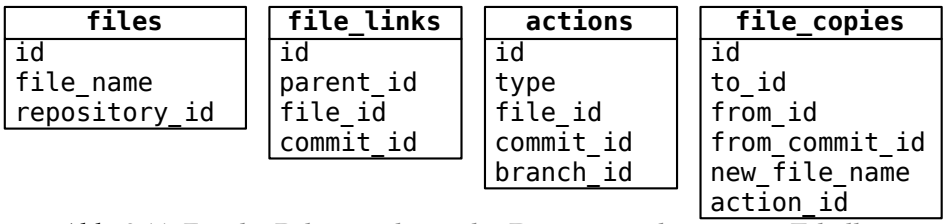

Abb. 3.11: Für die Rekonstruktion der Dateinamen benötigten Tabellen

files.file_name	files.id	file_links.parent_id
Apps	1	-1
NPS	3	1
EC	54	3
install.sh	72	54

Tab. 3.1: Tabellenauszug aus `files` für `Apps/NPS/EC/install.sh`

Beim Auslesen der Commits wird der vollständige aktuelle Pfad der Datei zusätzlich im Feld `current_file_path` gespeichert. Um im Nachhinein den aktuellen Pfad für eine `file_id` und `commit_id` zu ermitteln muss lediglich in der Tabelle `actions` nach dem Eintrag gesucht werden, dessen `file_id` und `commit_id` identisch ist. Quelltext 3.4 drückt dies als SQL-Abfrage aus.

```
SELECT current_file_path
FROM actions
WHERE file_id = '?' AND commit_id = '?'
```

Quelltext 3.4: SQL-Abfrage zur Ermittlung des Pfades einer Datei

Die Vereinfachung und Beschleunigung hat ihren Preis: Die Tabelle `actions` ist auf ca. die dreifache Größe angewachsen (von 7 MB auf 24 MB). Dafür konnte die Laufzeit in einem Test von von 2.069 Sekunden (ca. 34 Minuten) auf 36 Sekunden gesenkt werden. Außerdem wird nicht mehr diese große Menge an Arbeitsspeicher benötigt, da die Dateinamen nicht mehr vollständig im Arbeitsspeicher als Adjazenzmatrix vorgehalten werden müssen, sondern bei Bedarf schnell ermittelt werden können.

Die vorgenommenen Änderungen wurden bei den Maintainern eingereicht und sind in der aktuellen Version enthalten[5].

Weitere Probleme

Des Weiteren traten folgende Probleme auf, die durch die in den Fußnoten angegebenen Patches behoben werden:

Der konfigurierbare Branch, der für die Analysen verwendet werden soll, musste auf dem Original-Server vorhanden sein (z.B. `origin/master`)[6].

Die Tabellenspalte `date` ist nicht eindeutig. In Git ist es möglich, Änderungen (*Commits*) zu einem späteren Zeitpunkt von anderen Entwicklern anzuwenden und so zwischen dem Entwickler und dem Committer zu unterscheiden. `AuthorDate` beschreibt in Git, wann der Commit vom ursprünglichen Entwickler erstellt wurde. `CommitDate` beschreibt, wann der Commit angewandt wurde. Wenn der Entwickler und der Committer die gleiche Person ist, ist dieses Datum meistens identisch. Wenn ein Entwickler viele Commits über einen Zeitraum von einigen Tagen erstellt, diese danach „einreicht" und der Maintainer wenig später diese bestätigt, erhalten alle diese Commits als `CommitDate` den Zeitpunkt der Anwendung. Zeit-Analysen werden davon jedoch verzerrt, da z.B. 20 Commits alle das gleiche Datum haben, da sie zur gleichen Zeit angewandt wurden. Dies wird bereits bei den Tabellenspalten `committer_id` und `author_id` für den Namen aufgesplittet. Für das Datum war bisher nur die Tabellenspalte `date` verfügbar. Der erste Lösungsansatz bestand darin, in der Tabellenspalte `date` das `AuthorDate` zu speichern[7]. Beim zweiten Lösungsansatz wurden beide Informationen gespeichert. Dafür wurde die Tabellenspalte `date` in `commit_date` refaktorisiert und eine weitere Spalte `author_date` eingerichtet[8].

CVSAnalY bietet Extensions an. Welche Extensions verwendet werden sollen, kann in der Datei `\~{}/.cvsanaly2/config` unter `extensions` definiert werden. Einige

[5] siehe Diskussion `http://github.com/SoftwareIntrospectionLab/cvsanaly/issues/66`, sowie die Patches `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/89`, `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/108` und `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/111` (Abruf aller: 30.08.2011)

[6] siehe Patch `http://github.com/SoftwareIntrospectionLab/repositoryhandler/pull/4` (Abruf: 30.06.2011)

[7] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/82` (Abruf: 30.06.2011)

[8] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/94` (Abruf: 30.06.2011)

Extensions sind darauf angewiesen, dass andere Erweiterungen zuvor aktiv waren. Beispielsweise benötigt die Extension `HunkBlame` zuvor `Hunks`. Falls nicht weiter konfiguriert, werden die Extensions in alphabetischer Reihenfolge ausgeführt. Dies führt in dem Beispiel dazu, dass `HunkBlame` vor `Hunks` ausgeführt wird und dabei fehlschlägt. Für Abhilfe sorgt das Flag `hard_order=True` in der config-Datei.

Datenqualität

Nach der Überführung in MySQL enthält die Tabelle `scmlog` 32.548 Einträge. GitX zeigt mit der gleichen Auswahl an Branches 32.548 Commits an. Es sind also keine Daten bei der Überführung verloren gegangen.

3.3 Bugreport-Datenbank

Konzepte

Neben dem Repository wird die Bugreport-Datenbank für die weiteren Analysen benötigt. Für das Einlesen der Bugreport-Datenbank gibt es zwei Ansätze:

- Bugreports einzeln einlesen.
- Kopie der Datenbank verarbeiten.

Für den ersten Ansatz schlagen Fischer et al. [9] das Tool `wget` vor. Spezialisierte Lösungen wie `PyBugz`[9] können ebenfalls eingesetzt werden. Es ist jedoch einfacher, wenn eine Kopie der Bugreport-Datenbank zur Verfügung steht.

Anwendung

Bei Infopark wird seit Anfang 2003 das Open-Source-Tool Bugzilla[10] als Bugreport-Datenbank eingesetzt.

Da zunächst keine Kopie der Bugzilla-Datenbank vorlag, wurde PyBugz verwendet, um die Daten per Webservice abzuholen. PyBugz ist ein Python Kommandozeilen-Tool, welches über die XML-RPC Schnittstelle mit Bugzilla kommuniziert. Mit diesem Tool können unter anderem Informationen zu einem Bug erfragt oder nach einem bestimmten Bug gesucht werden. Um das Tool für diese Arbeit zu verwenden, mussten einige Änderungen am Quelltext durchgeführt werden[11].

Von Infopark wurde jedoch eine Kopie der Bugzilla-Datenbank zur Verfügung gestellt, was die Verarbeitung wesentlich erleichtert hat.

Die Datenbanktabellen `bugzilla` (siehe Abbildung 3.12) und `bugzilla_comments` (siehe Abbildung 3.13) wurden erstellt, um die Analysen zu vereinfachen.

bugzilla
id
title
date_reported
date_updated
reporter_email
assignee_email
status
resolution
duplicate_of
severity
priority
product
component
nr_attachments
nr_comments

Abb. 3.12: Schema der Tabelle `bugzilla`

bugzilla_comments
id
bugzilla_id
email
realname
date
comment

Abb. 3.13: Schema der Tabelle `bugzilla_comments`

[9] siehe `http://www.liquidx.net/pybugz` bzw. `http://github.com/williamh/pybugz` (Abruf: 30.06.2011)
[10] mehr Informationen unter `http://www.bugzilla.org` (Abruf: 30.06.2011)
[11] siehe Patches `http://github.com/apepper/pybugz` (Abruf: 30.06.2011)

Diese Tabellen können entweder per PyBugz befüllt oder als View für die eigentliche Bugzilla-Datenbank verwendet werden. Quelltext 3.5 erstellt Views für Bugzilla-Bugreports und -Kommentare.

Datenqualität

Die Tabelle `bugs` in der Datenbank `bugzilla` enthält 9.444 Einträge. Genau so viele Einträge sind in der View `bugzilla` in der Datenbank `cvsanaly` vorhanden. Die Tabelle `longdescs` in der Datenbank `bugzilla`, in der Kommentare zu einem Bugreport gespeichert sind, enthält 46.302 Einträge. Die View `bugzilla_comments` in der Datenbank `cvsanaly` beinhaltet ebenfalls genau so viele Einträge.

Somit sind keine Daten bei der Transformation verloren gegangen.

```
CREATE VIEW bugzilla AS
SELECT b.bug_id AS id, b.short_desc AS title, b.creation_ts AS date_reported,
  b.delta_ts AS date_updated, pf1.login_name AS reporter_email,
  pf2.login_name AS assignee_email, b.bug_status AS status, b.resolution AS resolution,
  d.dupe_of AS duplicate_of, b.bug_severity AS severity, b.priority AS priority,
  pd.name AS product, co.name AS component, b.version AS version,
  COUNT(a.attach_id) AS nr_attachments, COUNT(l.comment_id) AS nr_comments
FROM bugzilla.bugs b
JOIN bugzilla.profiles pf1 ON b.reporter = pf1.userid
JOIN bugzilla.profiles pf2 ON b.assigned_to = pf2.userid
JOIN bugzilla.products pd ON b.product_id = pd.id
JOIN bugzilla.components co ON b.component_id = co.id
LEFT JOIN bugzilla.attachments a ON b.bug_id = a.bug_id
LEFT JOIN bugzilla.longdescs l ON b.bug_id = l.bug_id
LEFT JOIN bugzilla.duplicates d ON b.bug_id = d.dupe
GROUP BY b.bug_id
ORDER BY b.bug_id;

CREATE VIEW bugzilla_comments AS
SELECT l.comment_id AS id, l.bug_id AS bugzilla_id, p.login_name AS email,
  p.realname AS realname, l.bug_when AS date, l.thetext AS comment
FROM bugzilla.longdescs l
JOIN bugzilla.profiles p ON l.who = p.userid;
```

Quelltext 3.5: Erstellen der View `bugzilla` und `bugzilla_comments`

Kapitel 4

Bugfixes und Bugfix-Links

Bei einem Bugfix handelt es sich um Änderungen im Quelltext, die einen Defekt beseitigen sollen. Diese Änderungen werden in einem oder mehreren Commits im VCS gespeichert. Neben dem direkten Bugfix existiert oft eine Bugreport-Datenbank für die Dokumentation der Defekte. Hier ist ein üblicher Ablauf, wie Defekte gemeldet und bearbeitet werden:

1. Ein Defekt (*Bug*) wird entdeckt und als *Bugreport* in einer Bugreport-Datenbank eingetragen und beschrieben.

2. Der Bugreport wird optional von Anderen kommentiert.

3. Der Defekt wird durch einen *Bugfix* repariert und committet (*Bugfix-Commit*).

4. Der dazugehörige Bugreport wird als gelöst markiert und geschlossen.

Für die Analysen dieser Diplomarbeit ist es essentiell, unter allen Code-Änderungen diejenigen zu identifizieren, bei denen es sich um Bugfixes handelt. Es ist auch wichtig, Bugfix-Links zu identifizieren, um beispielsweise Aussagen über Zeiträume treffen zu können (z.B. durchschnittliche Anzahl der Tage zwischen Bugreport und Bugfix).

Für das Aufspüren von Bugfixes werden in allen Arbeiten zu dem Thema die Commit-Messages des Repositorys als Quelle verwendet. Zumeist wird per Regular Expression nach Ziffernfolgen oder Wörtern wie „bug", „bugfix", „patch" oder ähnlichem gesucht (z.B. [14] oder [18]). Andere Arbeiten validieren die gefunden Bugfix-Links zusätzlich über die Bugreport-Datenbank (u.a. [2], [9], [12], [19] und [20]). Dabei werden teilweise unterschiedliche Metriken verwendet. Neben den bereits bekannten Verfahren für die Identifizierung von Bugfix-Links, macht sich diese Ausarbeitung zusätzliche Informationsquellen zunutze.

Ähnlich wie von Śliwerski et al. [20] beschrieben, wird in dieser Arbeit ein zweistufiges Verfahren für die Identifizierung von Bugfix-Links verwendet:

1. Bugfix-Links identifizieren.

2. Bugfix-Links validieren.

Wie im letzten Kapitel, werden für jeden Zwischenschritt zuerst die zugrundeliegenden **Konzepte** vorgestellt, daraufhin die **Anwendung** im konkreten Fall erörtert und zuletzt die **Datenqualität** beleuchtet.

4.1 Bugfixes identifizieren und verlinken

Konzepte

Zur Identifizierung von Bugfix-Links werden zwei Datenquellen verwendet:

- Commit-Message

- Bugreport-Datenbank-Kommentar

Commit-Message

Zuerst werden alle Commit-Messages auf enthaltene Ziffernfolgen untersucht. Es wird angenommen, dass diese Ziffernfolgen Bugreport-Datenbank-IDs sind. Um sicherzugehen, dass es sich um eine valide Bugreport-Datenbank-ID handelt, wird überprüft, ob ein Bugreport mit dieser ID existiert. Alle weiteren Einschränkungen oder Validierungen werden erst im nächsten Schritt angewendet.

Bugreport-Datenbank-Kommentar

Neben der Commit-Message gibt es weitere Quellen zur Identifizierung von Bugfix-Links, sofern eine bestimmte „Report-Kultur" vorhanden ist. In einer solchen „Report-Kultur" wird vom Entwickler in einem Kommentar zum Bugreport beschrieben, welcher Commit den Bugfix enthält. Dies kann ein Verweis auf eine SVN-Revisionsnummer oder ein Git-SHA-Wert sein. Um diese zu finden, werden alle Kommentare nach Verweisen auf einen Commit durchsucht. Bei einem gefundenen Verweis wird ebenfalls überprüft, ob der entsprechende Commit im Repository vorhanden ist.

Anwendung

Um die ermittelten Verweise zu persistieren, wird eine neue Datenbanktabelle `bugzilla_links` erstellt (siehe Abbildung 4.1 und Quelltext 4.1). Neben der `commit_id` und der `bugzilla_id` wird auch die Quelle des Eintrags (`source`) gespeichert. Die Quelle ist entweder `"Bugzilla SVN Comment"`, `"Bugzilla Git Comment"` oder `"Commit Message"`. Das Feld `false_positive` wird in Abschnitt 4.2 genauer beschrieben. Außerdem wird durch einen `UNIQUE`-Constraint sichergestellt, dass Links zwischen Bugzilla-ID und Commit-ID nicht mehrfach mit der gleichen Source gespeichert werden.

```
CREATE TABLE bugzilla_links (
  id int(11) NOT NULL AUTO_INCREMENT,
  commit_id int(11) DEFAULT NULL,
  bugzilla_id int(11) DEFAULT NULL,
  source varchar(255) DEFAULT NULL,
  false_positive tinyint(1) DEFAULT NULL,
  PRIMARY KEY (id),
  UNIQUE KEY reject_duplicates (commit_id,
    bugzilla_id, source)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Quelltext 4.1: Erstellen der Tabelle `bugzilla_links`

bugzilla_links
id
commit_id
bugzilla_id
source
false_positive

Abb. 4.1: Schema der Tabelle `bugzilla_links`

Commit-Message

Nach Bachmann et al. [1] ist es sinnvoll, alle Ziffernfolgen, die in Commit-Messages enthalten sind, daraufhin zu überprüfen, ob es sich dabei um Bugreport-IDs handelt. In Anlehnung an dieses Verfahren wird wie folgt vorgegangen:

1. Nach Ziffernfolgen in Commit-Message suchen.
2. In `bugzilla` überprüfen, ob Ziffernfolgen valide Bugreport-IDs ergeben.

Für den ersten Schritt wird die SQL-Abfrage aus Quelltext 4.2 verwendet.

```
SELECT id, message, author_date
FROM scmlog
WHERE message REGEXP
 '[^-@][[:<:]][1-9][0-9]{1,4}[[:>:]]'
AND author_date > '2003-03-20'
```

Quelltext 4.2: Ermittlung von Bugzilla-IDs in Commit-Messages

Alle Zahlen, die mit einem „@" anfangen werden ignoriert, da es sich dabei um SVN-Revisionsnummern handelt. Auch Zahlen

die mit „-" beginnen, werden ignoriert, da es sich hierbei um Datumsangaben handelt (z.B. „2008-10-27"). Zusätzlich wird das Datum des Commits eingeschränkt, da Bugzilla erst seit März 2003 eingesetzt wird und somit alle früheren Commits keine Referenzen enthalten können.

Quelltext 4.3 überprüft für den zweiten Schritt, ob eine Zahl einer Bugzilla-ID entspricht. Falls die ID nicht vorhanden ist, ergibt die Abfrage ein leeres Ergebnis.

```
SELECT * FROM bugzilla WHERE id='?'
```

Quelltext 4.3: Überprüfen der Bugzilla-ID

Bugreport-Datenbank-Kommentar

In den meisten Fällen wird bei Infopark ein korrigierter Defekt im Bugzilla vermerkt und in den Kommentaren des Bugreports explizit auf den Bugfix im Repository hingewiesen. Da verschiedene VCS eingesetzt wurden, haben sich bei Infopark zwei verschiedene Referenzarten für SVN und Git etabliert.

Bei SVN handelt es sich um ein zentrales VCS. Dementsprechend sind die Revisionsnummern aufsteigend und es gibt genau eine aktuelle Revision. Beispielsweise liegt Revision 125 zeitlich vor der Revision 148. Die Referenzart in den Bugzilla-Kommentaren für SVN-Commits ist der Buchstabe `r` gefolgt von der Revisionsnummer, z.B. `r148`. Quelltext 4.4 identifiziert alle Kommentare, die SVN-Revisionen enthalten.

```
SELECT * FROM bugzilla_comments
WHERE comment
 REGEXP '[[:<:]](R|r)[1-9][0-9]{1,}[[:>:]]'
```

Quelltext 4.4: Ermittlung von SVN-Revisionen in Kommentaren

Im Bugzilla werden neben Bugreports zu Infoparks CMS auch Informationen für andere Tools gesammelt, sodass sowohl bei der Suche nach SVN-Revisionen als auch bei der Suche nach Git-SHA-Werte jeweils überprüft werden muss, ob dieser Commit in `nps_full` vorhanden ist.

Bei der Umwandlung der SVN-Version von `nps_svn` in Git durch den Befehl `git svn clone` wurde die SVN-Revisionsnummer mit in der Commit-Message gespeichert. In Quelltext 4.5 ist eine Beispiel-SQL-Abfrage aufgeführt, um die `commit_id` für die Revision 148 zu ermitteln.

```
SELECT id AS commit_id
FROM scmlog
WHERE message LIKE '%@148 %'
```

Quelltext 4.5: Ermittlung der commit_id für SVN-Revisionen

Git hingegen ist ein dezentrales VCS. Daraus folgt, das es zu einem Zeitpunkt verschiede Zustände geben kann. Deshalb gibt es keine globalen Revisionsnummern wie bei SVN, sondern SHA-Werte, die einen Commit eindeutig identifizieren. Der einzelne Commit verweist auf seine Eltern, wodurch sich die gesamte Historie ermitteln lässt. Solch ein SHA-Wert hat 40 Hexadezimal-Stellen und sieht beispielsweise so aus: f6c214535ce7cfedf68017292156839625-29fc67. Quelltext 4.6 ermittelt alle Kommentare, die Git-SHA-Werte enthalten.

```
SELECT * FROM bugzilla_comments
WHERE comment
 REGEXP '[[:<:]][a-f0-9]{40}[[:>:]]'
```

Quelltext 4.6: Ermittlung von Git-SHA-Werten in Kommentaren

Für Git-SHA-Werte ist die Überprüfung sehr einfach, da die Revision in der Tabelle `scmlog` in der Spalte `rev` gespeichert wird. Quelltext 4.7 zeigt eine Beispiel-SQL-Abfrage.

```
SELECT id AS commit_id
FROM scmlog
WHERE rev=
 'f6c214535ce7cfedf68017292156839625 29fc67'
```

Quelltext 4.7: Ermittlung der commit_id für Git-SHA-Werte

Datenqualität

Durch die verschiedenen Ansätze konnten insgesamt 7.052 Links zwischen Bugfix-Commits und Bugreports identifiziert werden. Tabelle 4.1 listet die genaue Anzahl auf.

Maßnahme	# Bugfix-Links
Commit-Message	4.036
SVN-Werte	2.851
Git-Werte	165
Summe	7.052

Tab. 4.1: Anzahl der Bugfix-Links

4.2 Bugfixes verifizieren

Konzepte

Die im letzten Unterkapitel vorgestellten Maßnahmen zur Identifizierung von Bugfixes müssen überprüft werden, da diese auf Heuristiken basieren und somit Falsch-Positive enthalten können. Um sicherzustellen, dass es sich um einen gültigen Bugfix-Link handelt, können unterschiedliche Verfahren eingesetzt werden:

- Doppelte Referenzen erkennen
- Bugreport-Felder überprüfen
- Kommentar-Position berücksichtigen
- Zeitabschnitt überprüfen
- Commit-Messages überprüfen.

Doppelte Referenzen erkennen

Da durch die Bugzilla-Kommentare mehrere Quellen der Identifizierung vorhanden sind, kann der gleiche Bugfix-Link auch zweimal identifiziert werden. Beispielsweise schreibt der Entwickler in der Commit-Message „Bugfix BZ-#12984" und als Kommentar im Bugzilla zu diesem Bugreport: „gefixt in r148". Dann wird dieser Bugfix-Link sowohl über die Ziffernfolge in der Commit-Message identifiziert als auch über die SVN-Revisionsnummer. Außerdem wird per SQL-Constraint sichergestellt, dass ein Bugfix-Link zwischen einer Commit-ID und einem Bugreport pro Quelle (Commit-Message, SVN und Git) nur einmal eingetragen werden kann (siehe Quelltext 4.1). Insbesondere kann dieselbe SVN-Revisionsnummer in den Kommentaren zum gleichen Bugreport mehrfach vorkommen. Deshalb kann angenommen werden, dass es sich um korrekte Bugfix-Links handelt.

Bugreport-Felder überprüfen

Im Bugzilla von Infopark werden mehrere Projekte verwaltet. Somit kann es sein, dass SVN-Revisionsnummern oder Git-SHA-Werte von anderen Projekten gefunden werden. Im ersten Schritt „Bugfixes identifizieren und verlinken" wurde bereits überprüft, ob es sich um eine gültige SVN-Revision oder einen Git-SHA-Wert handelt. Doch gerade bei SVN können Revisionsnummern von einem anderen Projekten leicht als eigene Revisionsnummer missverstanden werden, da es sich um eine aufsteigende Zahl handelt. Um dies zu überprüfen, kann in der Tabelle `bugzilla` nachgeschaut werden, ob der Wert in `product` dem Projektnamen entspricht. Für Git-SHA-Werte ist dies nicht nötig, da zwar in der Theorie in zwei Repositorys der gleiche SHA-Wert existieren könnte, jedoch in der Praxis dieser Fall nicht auftritt[1].

Außerdem kann überprüft werden, ob es sich überhaupt um geschlossene Bugreports handelt. Eigentlich sollte, wenn ein Defekt behoben wurde, der Bugreport als gelöst (engl. „fixed") markiert werden. In wie vielen Fällen der Bugreport als gelöst

[1] siehe [6] Kapitel 6.1

markiert ist, wird im nächsten Abschnitt „Anwendung" erörtert.

Kommentarposition berücksichtigen

Oft wird in der Defektbeschreibung bereits eine Revision mit angegeben, in der das fehlerhafte Verhalten beobachtet wurde. Deshalb sind SVN-Revisionsnummern oder Git-SHA-Werte im Erst-Kommentar einen Bugreports sehr oft Falsch-Positive. Es kann sich jedoch auch um einen korrekten Link handeln, wenn z.B. zuerst der Defekt behoben wird und dann vollständigkeitshalber ein Bugreport erstellt wird, in dem gleich im Erst-Kommentar der Bugfix beschrieben wird. Über das genaue Verhältnis zwischen Falsch-Positiven und korrekten Bugfix-Links gibt der Abschnitt „Anwendung" Auskunft.

Zeitabstand überprüfen

Wenn ein Commit einen Bugreport referenziert, muss dieser Bugreport zum Zeitpunkt des Commits vorhanden sein. Im Beispiel in Abbildung 4.2 trifft dies auf `Commit B` und `C`, jedoch nicht auf `Commit A` zu. Deshalb kann davon ausgegangen werden, dass Ziffernfolgen, die in Commit-Messages auf „zukünftige" Bugreports verweisen Falsch-Positive sind.

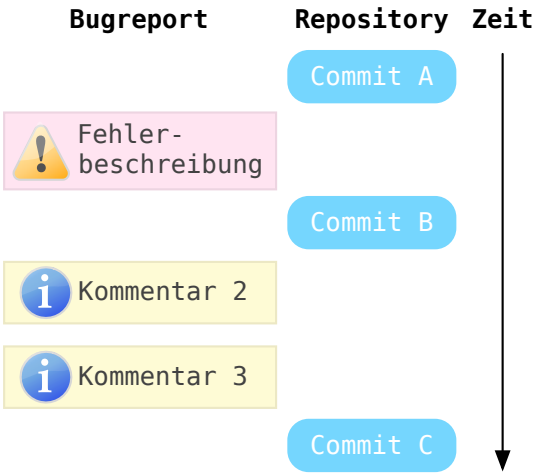

Abb. 4.2: Beispiel für zeitlichen Ablauf

Wenn in einem Bugreport-Kommentar ein Commit referenziert wird, muss dieser ebenfalls vor dem Kommentar existieren. Wenn Kommentar 2 aus Abbildung 4.2 z.B. auf `Commit B` verweist, kann es sich um einen korrekten Bugfix-Link handeln. Falls der gleiche Kommentar auf den „zukünftigen" `Commit C` verweist, so ist dies fehlerhaft, da der Commit zu diesem Zeitpunkt noch nicht existiert. Deshalb handelt es sich um einen Falsch-Positiven. `Commit A` als Referenz in Kommentar 2 wiederum ist ebenfalls falsch-positiv, da der Commit mit hoher Wahrscheinlichkeit den Defekt nicht behoben hat, sondern ein Verweis auf die Version ist, in der das Problem beobachtet wurde. Im Weiteren werden diese Fälle **negativer Zeitabstand** genannt. Quelltext 4.8 führt als Pseudocode an, wie Falsch-Positive per negativem Zeitabstand überprüft werden können.

```
IF Commit author_date >
   Bugreport date_reported
   THEN falsch-positiv
IF (Quelle SVN oder Git) AND
   (Commit author_date >
    Bugreport-Kommentar Date)
   THEN falsch-positiv
```

Quelltext 4.8: Pseudocode für negativen Zeitabstand

Wenn die Reihenfolge zwischen Commit und Bugreport oder Kommentar korrekt ist, dann kann der Zeitabstand trotzdem zur Identifizierung von Falsch-Positiven genutzt werden. Wenn z.B. in einer Commit-Message die Ziffernfolge 15 erwähnt wird und ein Bugreport mit dieser ID existiert, kann es sein, dass Monate, wenn nicht gar Jahre zwischen Bugreport und Commit liegen. In Abbildung 4.2 kann `Commit C` z.B. 4 Jahre nach dem Erstellen des Bugreports erstellt worden sein. Das es sich hierbei um einen korrekten Link handelt, kann nicht ausgeschlossen werden. Dennoch ist die Wahrscheinlichkeit recht groß, dass es sich

bei großen Zeitabständen um Falsch-Positive handelt, etwa durch eine Commit-Message der Art „Jetzt 15 mal so schnell". Im Weiteren werden diese Fälle **übergroßer Zeitabstand** genannt.

Wie groß dieser Zeitabstand sein kann, wird im Abschnitt „Anwendung" erörtert.

Manuelle Überprüfung

Bugfix-Links, die nur über Ziffernfolgen innerhalb einer Commit-Message gefunden wurden, müssen unbedingt manuell überprüft werden, da besonders kleine Zahlen mehrdeutig sein können. Bugzilla-IDs beginnen bei 1 und werden hochgezählt. Somit kann z.B. die Ziffernfolge 20 ein Verweis auf die Bugzilla-ID sein, oder einen anderen Kontext haben (z.B. „20 statt 15 Sekunden bis zum Timeout"). Auch Versions- und Portnummern (80, 8080, 3000, 4000 usw.) sowie HTTP-Statuscodes (200, 404 usw.) können zu Falsch-Positiven führen, da entsprechende Bugreport-IDs vorhanden sein können. Wie viele der Ziffernfolgen fehlerhaft sind, wird im nächsten Abschnitt untersucht.

Anwendung

Doppelte Referenzen erkennen

Wie bereits im konzeptionellen Teil dargelegt, können Bugfix-Links, die über zwei verschiedene Verfahren identifiziert wurden, als verifiziert betrachtet werden. Mehrfache Verknüpfungen zwischen einem Bugfix-Commit und Bugreport werden bei den Bugfix-Link-Unikaten nur einmal gezählt. Von den 7.052 gefundenen Bugfix-Links sind 5.005 Bugfix-Link-Unikate (siehe Quelltext 4.9).

```
SELECT COUNT(*)
FROM (SELECT *, COUNT(*) AS amount
  FROM bugzilla_links
  GROUP BY bugzilla_id, commit_id
) AS subquery
```

Quelltext 4.9: Anzahl der Bugfix-Link-Unikate

Die 7.052 - 5.005 = 2.047 doppelten Einträge (Duplikate) werden verwendet, um die entsprechenden 2.047 Bugfix-Link-Unikate zu verifizieren (entspricht ca. 41% der Gesamtmenge). Demnach sind 2.958 Bugfix-Links nur durch ein Verfahren identifiziert (*Einzel-Links*) und müssen weiter überprüft werden. Die SQL-Abfrage aus Quelltext 4.9 muss um ein `WHERE amount > 1` für die Duplikate bzw. `WHERE amount = 1` für die Einzel-Links erweitert werden, um die entsprechende Anzahl zu ermitteln. Tabelle 4.2 fasst die Ergebnisse zusammen. Im Folgenden sind die Bugfix-Link-Unikate als Gesamtzahl angegeben.

	# in `bugzilla`
Bugfix-Link-Unikate	5.005
Duplikate	2.047
Einzel-Links	2.958

Tab. 4.2: Anzahl der Bugfix-Links mit Duplikaten

Bugreport-Felder überprüfen

Von den gefundenen Bugfix-Links sind nur 34 Bugreports (0,68% der Gesamtzahl) aus einer anderen Produktkategorie als „Fiona" (siehe Tabelle 4.3).

Produkt	Anzahl	Anzahl pro Gesamtzahl
Fiona	4971	99,32%
Rails Connector	21	0,42%
infopark.de	6	0,12%
Playland	4	0,08%
IT-Sys-Admin	3	0,06%

Tab. 4.3: Aufteilung der Produktkategorie der Bugreports mit Bugfix-Link

Nach einer manuellen Überprüfung (siehe Tabelle 4.5) zeigt sich, dass die Duplikate wie angenommen keine Falsch-Positiven enthalten. Von den Einzel-Links sind 10 (etwas mehr als ein Drittel) der Bugfix-Links fehlerhaft. Von diesem Drittel wurden keine per Git-SHA-Wert identifiziert, da die Wahrscheinlichkeit einer zufällige Überlappung von Git-SHA-Werte zwischen

verschiedenen Projekten extrem gering ist[2]. Dass die Zahl der verifizierten Links auch bei anderer Produktangabe übereinstimmt, kann damit erklärt werden, dass die meisten anderen Produkte im Bugzilla bei Infopark etwas mit Infoparks CMS zu tun haben (z.B. Demo-Content oder die Firmenwebsite, die mit Fiona betrieben wird).

Bugzilla ermöglicht es, für jeden Bugreport zu vermerken, ob es eine Lösung gibt (engl. „Resolution"). Falls die Lösung nicht „fixed" (dt. repariert) ist, ist davon auszugehen, dass (noch) keine Lösung gefunden wurde. Insgesamt gibt es 471 verlinkte Bugreports, deren Lösung nicht „fixed" ist (siehe Tabelle 4.4).

Tabelle 4.6 listest die Ergebnisse der manuellen Überprüfung auf. Insbesondere bei SVN-Einzel-Links und Bugfix-Links, die über eine Ziffernfolge in der Commit-Message hergestellt wurden, waren Anteile von über 60% Falsch-Positive zu finden. Tabelle 4.7 zeigt die Falsch-Positiven pro Lösungskategorie. Keine Kategorie sticht besonders heraus, da alle Werte nur maximal 10% vom Median 59% abweichen.

Lösung	Anzahl	Anzahl pro Gesamtzahl
Fixed	4534	90,59%
Won't fix	162	3,24%
Works for me	108	2,16%
Duplicate	94	1,88%
<leer>	60	1,2%
Invalid	47	0,94%

Tab. 4.4: Aufteilung der Lösungskategorie der Bugreports mit Bugfix-Link

	# Anderes Produkt	Anderes Produkt pro Gesamtzahl	# Falsch-positiv	Falsch-positiv pro anderes Produkt	Falsch-positiv pro Gesamtzahl
Duplikate	7	0,14%	0	0%	0%
Einzel-Links					
Git	1	0,02%	0	0%	0%
SVN	12	0,24%	5	41,67%	0,1%
Message	14	0,28%	5	35,71%	0,1%
Summe	27	0,54%	10	37,04%	0,2%

Tab. 4.5: Verifizieren durch anderes Produkt

	# Andere Lösung	Andere Lösung pro Gesamtzahl	# Falsch-positiv	Falsch-positiv pro andere Lösung	Falsch-positiv pro Gesamtzahl
Duplikate	103	2,06%	0	0%	0%
Einzel-Links					
Git	4	0,08%	0	0%	0%
SVN	167	3,34%	105	62,87%	2,1%
Message	197	3,94%	105	53,3%	2,1%
Summe	368	7,35%	210	57,07%	4,2%

Tab. 4.6: Verifizieren durch andere Lösung

Lösung	Anzahl	Falsch-positiv	Falsch-positiv pro Anzahl
Won't fix	103	51	50%
Works for me	90	53	59%
Duplicate	82	52	63%
<leer>	51	25	49%
Invalid	42	29	69%

Tab. 4.7: Falsch-Positive pro Lösung von Einzel-Links

[2] siehe [6] Kapitel 6.1

Falls automatisiert Falsch-Positive ermittelt werden sollen, bietet sich folgende Heuristik an: Ein Bugfix-Link ist falsch-positiv, wenn die Produktkategorie des Bugreports sich vom untersuchten Produkt unterscheidet und der Bugfix-Commit nur über eine SVN-Revisionsnummer oder eine Ziffernfolge in der Commit-Message identifiziert wurde. Dies ist möglich, da nur wenige Bugfix-Links betroffen sind (0,5% der Gesamtzahl). Bei der Lösungskategorie können ebenfalls alle Einzel-Links per SVN oder Commit-Message als Falsch-Positive markiert werden, wenn die Kategorie nicht „fixed" entspricht. Dies ist möglich, da mehr Falsch-Positive als Positive enthalten sind (57,07%).

Kommentarposition berücksichtigen

Bei der Überprüfung, in welchem Kommentar die SVN-Revisionsnummer oder der Git-SHA-Wert erwähnt wird, zeigt sich deutlich, dass ein Großteil der Bugfix-Links als falsch-positiv zu markieren sind (siehe Tabelle 4.8). Nur die durch zwei Verfahren identifizierten Links (Duplikate) waren gültig, sowie ein einzelner Git-SHA-Wert und einige SVN-Revisionen (unter 4%). Somit ist dieses Verfahren gut geeignet, um Falsch-Positive automatisiert zu identifizieren.

Zeitabstand überprüfen

681 Bugfix-Links haben einen **negativen Zeitabstand** zwischen Commit und Bugreport-Erstellung (siehe Tabelle 4.9). Das heißt, dass in diesen Fällen zuerst der Commit existierte und erst später der Bugreport gemeldet wurde. Wenn eine Latenz von 24 Stunden eingeräumt wird, kann die Rate der detektierten Falsch-Positiven drastisch erhöht werden.

	# Erst-Kommentar	Erst-Kommentar pro Gesamtzahl	# Falsch-positiv	Falsch-positiv pro Erst-Kommentar	Falsch-positiv pro Gesamtzahl
Duplikate	2	0,04%	0	0%	0%
Einzel-Links					
Git	3	0,06%	2	66,67%	0,04%
SVN	177	3,54%	166	93,79%	3,32%
Summe	180	3,6%	168	93,33%	3,36%

Tab. 4.8: Verifizieren durch Erst-Kommentar

	# Neg. Zeit-abstand	Neg. Zeit-abstand pro Gesamtzahl	# Falsch-positiv	Falsch-positiv pro neg. Zeitabstand	Falsch-positiv pro Gesamtzahl
Duplikate	176	3,52%	1	0,57%	0,02%
Einzel-Links					
Git	6	0,12%	3	50%	0,06%
SVN	310	6,19%	208	67,1%	4,16%
Message	189	3,78%	43	22,75%	0,86%
Summe	505	10,09%	254	50,3%	5,07%
Einzel-Links mit Zeitabstand > 24 Stunden					
Git	6	0,12%	3	50%	0,06%
SVN	186	3,72%	165	88,71%	3,3%
Message	43	0,86%	42	97,67%	0,84%
Summe	229	4,58%	207	90,39%	4,14%

Tab. 4.9: Verifizieren durch negativen Zeitabstand zwischen Bugreport und Commit

So sind nur ca. 23% aller Bugfix-Links, die nur über die Commit-Message identifiziert wurden und bei denen der Commit *vor* dem Bugreport existierte Falsch-Positive. Mit einer 24-Stunden-Latenz liegt die gleiche Rate bei 98% bei fast gleicher Abdeckung der Falsch-Positiven (nur ein falsch-positiver Bugfix-Link, der innerhalb des 24 Stunden Fensters liegt). Auch bei der SVN-Revisionsnummer lässt sich so die Rate der detektierten Falsch-Positiven verbessern.

Selbst ein Duplikat wurde durch diese Maßnahme als falsch-positiv entdeckt. In dem konkreten Fall wurde eine SVN-Revisionsnummer in einem Bugreport-Kommentar als Defektursache referenziert. Ein anderer Kollege notierte in einem späteren Kommentar den passenden Git-SHA-Wert für das Repository `nps_svn`. Dies ist das einzige Duplikat, welches durch eine SVN-Revisionsnummer und einen Git-SHA-Wert identifiziert wurde. Alle anderen Duplikate wurden durch eine Commit-Message und wahlweise eine SVN- oder Git-Referenz ermittelt. Es ist anzunehmen, dass die Zeiteinstellung des Bugzilla-Servers und des jeweiligen Computers, auf dem die Änderungen (Commit) durchgeführt wurden, nicht synchronisiert waren und es deshalb zu diesen Zeitverschiebungen kam.

Tabelle 4.10 führt die Ergebnisse für negative Zeitabstände zwischen Bugreport-Kommentaren und Commits auf. Auch hier zeigt sich die Nützlichkeit eines 24-Stunden-Zeitfensters. Alle falsch-positiven Bugfix-Links, die durch dieses Verfahren gefunden wurden, haben einen negativen Zeitabstand von über 24 Stunden.

Insgesamt sind die Verfahren mit negativem Zeitabstand und einem Zeitfenster von 24 Stunden sehr gut geeignet, um Falsch-Positive automatisiert zu identifizieren.

Übergroße Zeitabstände zwischen Commit und erwähnendem Kommentar können einen Hinweis darauf geben, ob es sich um Falsch-Positive handelt. Abbildung 4.3 zeigt jeweils einen Boxplot für die Duplikate und die Einzel-Links, sowie den gleichen Boxplot mit einer logarithmischen Skala. Da die meisten Zeitabstände null Tage sind, musste vor dem Logarithmus eine +1 addiert werden. Pro Boxplot wird angezeigt, wie viele Tage zwischen einem referenzierten Commit und dem erwähnenden Kommentar liegen. Wie sich zeigt, sind in der Duplikatengruppe die ersten drei Quartile sowie der Median null Tage. Der Mittelwert beträgt 2,47 Tagen und der Maximalwert 618 Tagen. Bei der Einzel-Link-Gruppe liegt der Median ebenfalls bei null Tagen. Doch das dritte Quartil ist bei 5 Tagen und der Mittelwert bei 25,87 Tagen (Maximum 618 Tage). Nach Absprache mit den Entwicklern wurde eine Drei-Tagesgrenze als sinnvoll erachtet.

	# Neg. Zeit-abstand	Neg. Zeit-abstand pro Gesamtzahl	# Falsch-positiv	Falsch-positiv pro neg. Zeitabstand	Falsch-positiv pro Gesamtzahl
Duplikate	12	0,24%	0	0%	0%
Einzel-Links					
Git	6	0,12%	0	0%	0%
SVN	9	0,18%	6	66,67%	0,12%
Summe	15	0,3%	6	40%	0,12%
Einzel-Links mit Zeitabstand > 24 Stunden					
SVN	6	0,12%	6	100%	0,12%

Tab. 4.10: Verifizieren durch negativen Zeitabstand zwischen Kommentar und Commit

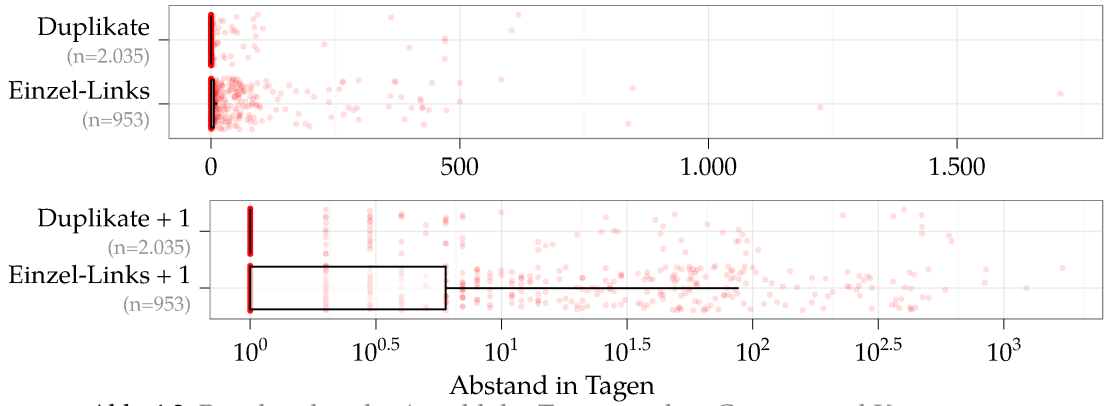

Abb. 4.3: Boxplot über die Anzahl der Tage zwischen Commit und Kommentar

Ob dieser Wert auf andere Projekte übertragbar ist, hängt sehr von dem verwendeten Prozess ab.

Tabelle 4.11 zeigt die Auflistung von allen Commits, die mindestens 24 Stunden (1 Tag) oder 3 Tage nach dem Original-Commit kommentiert wurden. Dabei ist auch das Duplikat als fehlerhaft aufgefallen, das bereits beim negativen Zeitabstand zwischen Bugreport und Commit identifiziert wurde. Der Kommentar war über 600 Tage älter als der Commit. Als automatisiertes Verfahren mit 3 Tagen Zeitabstand ist dieses Verfahren zumindest bei Infopark CMS einsetzbar.

Für Bugfix-Links, die nur über die Commit-Message identifiziert wurden, ist es schwierig, einen Datumsvergleich vorzunehmen. Natürlich kann das Erstellungsdatum des Commits mit dem Erstellungsdatum des Bugreports verglichen werden, doch zeigt es sich, dass eine Vielzahl der Commits lange nach dem ersten Erstellen des Bugreport erstellt wurden. Abbildung 4.4 zeigt die Verteilung des Abstands zwischen Bugreport-Erstellung und Commit in Tagen als Boxplot sowohl für Einzel-Links als auch für Duplikate. Hier wird ebenfalls ein zweiter Boxplot mit logarithmischer Skala angezeigt, bei dem alle Werte vorher um Eins erhöht wurden. In dem Plot werden nur Bugfix-Links verwendet, die unter anderem durch die Commit-Message identifiziert wurden.

	# Pos. Zeit-abstand	Pos. Zeit-abstand pro Gesamtzahl	# Falsch-positiv	Falsch-positiv pro pos. Zeitabstand	Falsch-positiv pro Gesamtzahl
Zeitabstand > 1 Tag (24h)					
Duplikate	72	1,44%	1	1,39%	0,02%
Einzel-Links					
Git	32	0,64%	3	9,38%	0,06%
SVN	290	5,79%	190	65,52%	3,8%
Summe	322	6,43%	193	59,94%	3,86%
Zeitabstand > 3 Tage (72h)					
Duplikate	52	1,04%	1	1,92%	0,02%
Einzel-Links					
Git	28	0,56%	3	10,71%	0,06%
SVN	236	4,72%	172	72,88%	3,44%
Summe	264	5,27%	175	66,29%	3,5%

Tab. 4.11: Verifizieren durch übergroßen Zeitabstand zwischen Commit und Kommentar

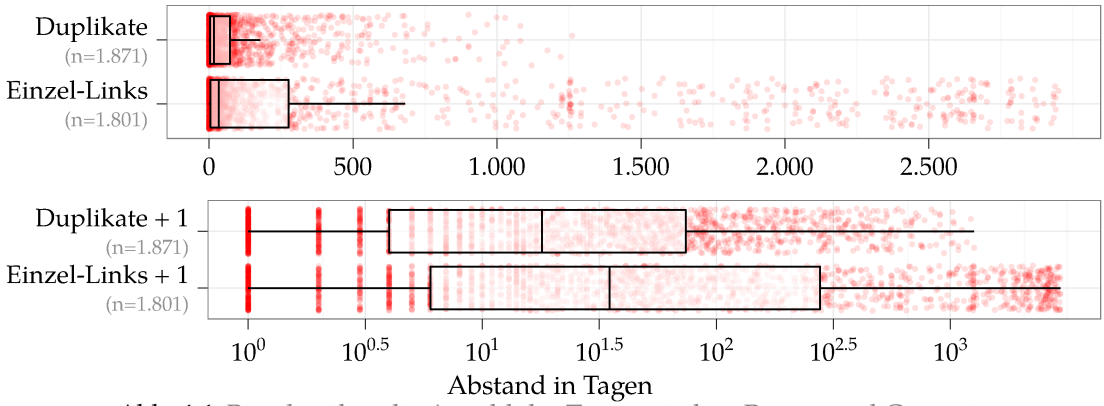

Abb. 4.4: Boxplot über die Anzahl der Tage zwischen Report und Commit

Dabei zeigt sich deutlich, dass die Unikate eine wesentlich breitere Verteilung als die Duplikate haben. Eine Möglichkeit ist es, den maximalen Zeitabstand der Duplikate (1.262 Tage) zur Unterteilung für die Einzel-Links zu verwenden. Tabelle 4.12 zeigt die Ergebnisse der manuellen Überprüfung aller Bugfix-Links, bei denen mehr als 1.262 Tage zwischen Bugreport und Commit vergangen sind. Mit fast 98% Falsch-Positiven ist diese Methode sehr gut für Automatisierung geeignet. Die fünf validen Bugfix-Links hatten einen Abstand von bis zu 2.185 Tagen (knapp 6 Jahre).

Manuelle Überprüfung

Die manuelle Überprüfung von Einzel-Links, die nur über die Commit-Message identifiziert wurden, ergab, dass von den 1.990 Einzel-Links 457 Falsch-Positive sind (siehe Tabelle 4.13). Somit sind in etwa ein Fünftel dieser Einzel-Links fehlerhaft. Wenn die Bugzilla-ID mit in Betracht gezogen wird, zeigt sich, dass mehr als die Hälfte der Falsch-Positiven eine Bugzilla-ID von unter oder gleich 500 haben. Mit einer hohen Falsch-Positive-Rate von über 2/3 ist dieses Kriterium mit der Kennzahl 500 gut für Automatisierung geeignet.

	# Pos. Zeit-abstand	Pos. Zeit-abstand pro Gesamtzahl	# Falsch-positiv	Falsch-positiv pro pos. Zeitabstand	Falsch-positiv pro Gesamtzahl
Einzel-Links mit Zeitabstand > 1262 Tage					
Commit-Message	240	4,8%	235	97,92%	4,7%

Tab. 4.12: Verifizieren durch übergroßen Zeitabstand zwischen Bugreport und Commit

Bugzilla-IDs	# Links	Links pro Gesamtzahl	# falsch-positive	falsch-positive pro Link	falsch-positive pro Gesamtzahl
<= 100	155	3,1%	148	95,48%	2,96%
<= 500	414	8,27%	297	71,74%	5,93%
<= 1000	654	13,07%	326	49,85%	6,51%
> 1000	1336	26,69%	131	9,81%	2,62%
Alle	1990	39,76%	457	22,96%	9,13%

Tab. 4.13: Verifizieren durch manuelle Überprüfung aller Commit-Message Einzel-Links

Datenqualität

Im vorausgegangenen Abschnitt wurden verschiedene Verfahren evaluiert, um Bugfix-Links zu verifizieren. Tabelle 4.14 zeigt die Ergebnisse nach allen (manuellen) Überprüfungen. Die 4.294 validierten Bugfix-Links verweisen auf 4.070 unterschiedliche Commits. Das von Sadowski, Lewis et al. [18] verwendete Verfahren, welches als Erweiterung *BugFixMessage* von CVSAnalY verfügbar ist und anhand von Regular Expressions auf Commit-Messages arbeitet, hat für den gleichen Zeitraum 4.279 Commits als Bugfixes identifiziert (siehe Tabelle 4.15).

Durch die beschriebenen Verfahren wurden weniger Bugfix-Links identifiziert, doch der Informationsgehalt und die Korrektheit ist wesentlich höher. Außerdem bringt die Verlinkung zwischen Bugreport und Bugfix mehr Analysemöglichkeiten.

Um die Datenqualität der Bugfix-Links besser beurteilen zu können, haben Bachmann et al. [2] vorgeschlagen, eine Reihe von Qualitätsmaßen zu verwenden. In dem eben genannten Paper sind alle Maße ausführlich dokumentiert.

Tabelle 4.16 listet die Qualitätsmaße für Infoparks CMS und stellt die Ergebnisse nach der Methode von Bachmann et al. [1] mit der hier zusätzlich verwendeten Methode der Identifizierung über SVN-Revisionen und Git-SHA-Werten gegenüber. Insgesamt konnte dadurch gegenüber der Methode von Bachmann et al. [1] eine Erhöhung der Datenqualität zwischen ca. 12 und 17% erreicht werden.

Bachmann et al. [2] untersuchten die Qualitätsmaße der Open-Source-Projekte Apache HTTPD, Eclipse, GNOME, NetBeans, OpenOffice und eine unbenannte kommerzielle Bankenanwendung. Im Vergleich mit diesen Ergebnissen zeigt sich, dass Infopark eine höhere reparierte Bugreport-Rate hat als Produkte, die in dem Paper untersucht wurden. Die Duplikat-Rate unter den Bugreports ist mit 8,86% geringer als der niedrigste Open-Source-Projekt-Wert (12,39%), jedoch höher als das verglichene kommerzielle Produkt (1,38%).

	# Links	Links pro Gesamtzahl	# falsch-positive	falsch-positive pro Link	falsch-positive pro Gesamtzahl
Duplikate	2047	40,9%	1	0,05%	0,02%
Einzel-Links					
Git	56	1,12%	3	5,36%	0,06%
SVN	912	18,22%	250	27,41%	5%
Message	1990	39,76%	457	22,96%	9,13%
Summe	2958	59,1%	710	24%	14,19%

Tab. 4.14: Endergebnis aller Verifizierungen

	# Commits	Commits pro Gesamtzahl
per CVSAnalY-Erweiterung ermittelt	4279	13,1%
per Bugfix-Links ermittelt	4070	12,5%
nur per CVSAnalY-Erweiterung ermittelt	1402	4,3%
nur per Bugfix-Links ermittelt	1193	3,7%
mit beiden Verfahren ermittelt	2877	8,8%

Tab. 4.15: Ermittelte Bugfix-Commits nach Verfahren

Dieses Ergebnis bestätigt die Annahme von Bachmann et al. [2], dass Closed-Source-Projekte weniger Bugreport-Duplikate als Open-Source-Projekte enthalten.

Bei der Works-for-me- und Invalid-Rate liegen alle untersuchten Projekte unter dem von Infopark, außer Apache HTTPD mit einem Wert von 34,46%. Bei der Validierung von Bugfix-Links wurde jedoch beobachtet, das es auch einige korrekte Bugfixes für Bugreports der Lösung „Works-for-me" und „Invalid" existieren (siehe Tabelle 4.7), womit nicht klar ist, wie aussagekräftig dieser Wert ist.

Die Leere Commit-Message-Rate liegt bei 7,28%. Nur Eclipse mit ca. 20% und die kommerzielle Bankenanwendung mit ca. 10% liegen im Vergleich höher. Die anderen Open-Source Projekte liegen unter 2%. Dazu ist zu erwähnen, dass mehr als 75% der leeren Commits beim automatisierten Build-Prozess eine Package-Nummer hochzählen. Ohne diesen PackageCount liegt die Leere Commit-Message-Rate bei 1,85%.

Mit ca. 13% Bugreport-Link-Rate liegt Infopark CMS genau auf dem Mittelwert 13% der untersuchten Projekte, jedoch über dem Median von ca. 9%. Nur Eclipse mit ca. 34,37% und NetBeans 12,92% haben eine höherer Bugreport-Link-Rate.

Sowohl die Bugfix-Link-Rate für alle Bugreports als auch für reparierte Bugreports liegt mit ca. 36% bzw. 67% weit über den Maximalwerten der anderen Produkte, die im Maximum bei ca. 29% bzw. 55% liegen. Dies ist der Disziplin der Entwickler zu verdanken, die die meisten Bugfixes für Bugreports explizit vermerkt haben.

Insgesamt konnten durch die verschiedenen Maßnahmen ca. 2/3 aller Bugfixes identifiziert werden. Dies entspricht nicht einer vollständigen Abdeckung, kann jedoch als gute Studiengrundlage verwendet werden.

Qualitätsmaß	Infopark CMS
Reparierte Bugreport-Rate #fixed bug reports / #bug reports	60,45%
Bugreport-Duplikat-Rate #duplicate bug reports / #bug reports	8,86%
Works-for-me & invalide Bugreport-Rate #works-for-me & invalid bug reports / #bug reports	23,82%
Leere Commit-Message-Rate #empty commit messages / #commit messages	7,28%

Qualitätsmaß	Infopark CMS	
	nach Bachmann [1]	*+ SVN und Git*
Bugreport-Link-Rate #commit messages with bug reportlinks / #commit messages (w/o empty)	10,68%	12,53%
Bugfix-Link-Rate (alle Reports) #linked bug reports / #bug reports	31,90%	35,94%
Bugfix-Link-Rate (nur reparierte Reports) #linked bug reports / #fixed bug reports	59,26%	66,78%

Tab. 4.16: Qualitätsmaße nach Bachmann [2]

Kapitel 5

Bug-Commits identifizieren

Die Verlinkung zwischen Bugfix und dem eigentlichen Bug ist der letzte Schritt zum Identifizieren eines Bug-Commits. Ein Bugfix-Commit beschreibt, welche Dateien und Zeilen geändert werden, um einen Defekt zu beseitigen. Somit ist bereits im Bugfix-Commit der Ort des Defektes (Datei und Zeile) identifizierbar.

Nach dem aktuellen Forschungsstand ist der SZZ-Algorithmus einer der wenigen Versuche, das Identifizieren von Bug-Commits zu automatisieren. Der SZZ-Algorithmus wurde 2005 im Paper [20] vorgestellt und ist nach seinen Autoren Śliwerski, Zimmermann und Zeller benannt. Zusätzlich wurde der Algorithmus in [11] verbessert und dient als Grundlage für die weiterführenden Forschungen von [8], [12] und [18].

Beim SZZ-Algorithmus wird pro geänderter Zeile eines Bugfix-Commits identifiziert, wann und von wem diese Zeile zuletzt geändert wurde.

Quelltext 5.1 beschreibt einen Beispiel-Bugfix-Patch, der die Ruby-Methode `say_hello` repariert. Die Funktion soll je nach gegebener Stunde des Tages „Good day" oder „Good night" zurück geben. Dabei wird ein Tag in zwei gleich große Hälften aufgeteilt. Beim Programmieren hat sich der Defekt `<= 12` eingeschlichen, der dafür sorgt, dass die erste Tageshälfte 13 und die zweite Tageshälfte nur 11 Stunden hat. Mit der Änderung zu `<` sind die beiden Tageshälften wieder gleich groß.

```
diff --git a/file_name.rb b/file_name.rb
index b9ac9ab..f6631c3 100644
--- a/file_name.rb
+++ b/file_name.rb
@@ -1,3 +1,3 @@
 def say_hello(hour)
-  "Good "+ (hour <= 12 ? "day":"night")
+  "Good "+ (hour < 12 ? "day":"night")
 end
```

Quelltext 5.1: Diff Patch eines Bugfixes

Die meisten Versionsverwaltungssysteme bieten folgende Funktion: Für jede Zeile einer Datei und Revision wird angegeben, wann diese Zeile zuletzt geändert wurde. CVS und SVN bieten das Kommando `annotate`, Git das Kommando `blame`. Quelltext 5.2 zeigt `git blame` für das oben genannte Beispiel. Demnach hat `User W` im Commit `fd2d09ec` die fehlerbehaftete Zeile zuletzt geändert. Nach dem SZZ-Algorithmus handelt es sich bei diesem Commit um einen „fix-inducing change" (dt. „Änderung, die eine spätere Defekt-Behebung auslöst"). Im Weiteren wird der Begriff *Bug-Commit* als Synonym für „fix-inducing change" verwendet.

```
$ git blame HEAD~1 -- file_name.rb
^0ca0768 User V   2011-09-20 15:37:24 +0200 1) def say_hello(hour)
fd2d09ec User W   2011-09-20 15:37:47 +0200 2)   "Good "+ (hour <= 12 ? "day":"night")
^0ca0768 User V   2011-09-20 15:37:24 +0200 3) end
```

Quelltext 5.2: Beispiel `git blame`

CVSAnalY in der Version der UCSC[1] bietet mit den Erweiterungen `Hunks` (dt. „großes Stück", „Brocken" oder „Abschnitt") und `HunkBlame` eine Implementierung des SZZ-Algorithmus an und wurde bereits in [18] eingesetzt. Im Folgenden wird der technische Aufbau dieser Implementierung beschrieben und auf Infopark CMS angewendet.

5.1 Technischer Aufbau

Abbildung 5.1 zeigt einen Überblick über die CVSAnalY-Erweiterungen, die in diesem Kapitel beschrieben und verwendet werden. Die Erweiterungen sind über oder neben den Pfeilen (z.B. `CommitsLOC`) und die Datenbanktabellen in dem Symbol für Datenbank (z.B. `commits_lines`) notiert. Fast alle Erweiterungen benötigen Lesezugriff auf die Änderungen pro Commit, weshalb diese von der Erweiterung `Patches` gespeichert werden. Aus diesen Änderungen lässt sich extrahieren, welche Zeilen genau geändert wurden (Erweiterung `Hunks`) und wer diese Zeilen vor ihrer Änderung zu verantworten hat (Erweiterung `HunkBlames`). Die beiden Erweiterungen `CommitsLOC` und `PatchLOC` werden zur Qualitätssicherung verwendet.

5.2 Patches

Konzepte

Die Erweiterung `Patches` liest für jeden Commit den vollständigen Diff[2] ein, teilt die Diffs in einzelne Dateien auf und speichert jeden Datei-Diff. Für das Einlesen des vollständigen Diffs eines Commits wird der Befehl `git show --find-copies` verwendet. Anschließend wird der vollständige Diff vom PatchParser aus dem Bazaar Projekt[3] verarbeitet und in Dateien aufgeteilt. Somit ist für jede Datei ein Patch vorhanden, die in einem Commit geändert wurde. Dieser Patch wird von der Erweiterung `Patches` in der Datenbanktabelle `patches` gespeichert.

Wie Abbildung 5.1 bereits andeutet, ist eine hohe Datenqualität der Tabelle `patches` für andere Erweiterungen Grundvoraussetzung.

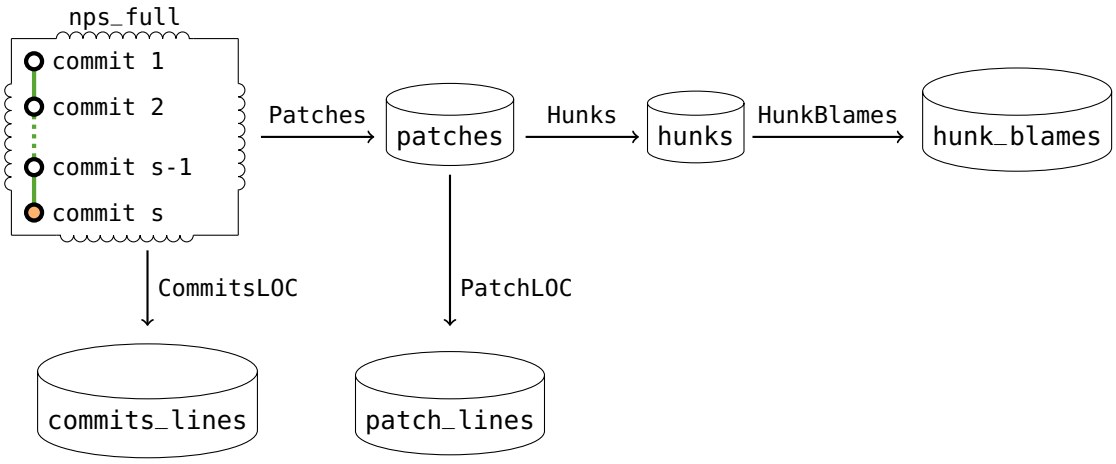

Abb. 5.1: Überblick der angewandten Erweiterungen von CVSAnalY

[1] siehe `http://github.com/SoftwareIntrospectionLab/cvsanaly` (Abruf: 01.09.2011)
[2] Darstellungsform von Unterschieden zwischen zwei Text-Dateien
[3] weiteres zum Versionsverwaltungssystem Bazaar siehe `http://bazaar-vcs.org` (Abruf: 23.09.2011)

Anwendung

Messinstrumente
Die Erweiterung `CommitsLOC`[4] berechnet, wie viele Zeilen pro Commit hinzugefügt und entfernt wurden. Dafür wird auf VCS-Ebene der Output von `git log --shortstat` geparst und anschließend in der Datenbanktabelle `commits_lines` gespeichert.

Um die Datenqualität der Erweiterung `Patches` besser beurteilen zu können, wurde im Rahmen dieser Diplomarbeit die Erweiterung `PatchLOC` entwickelt[5]. Für jeden Patch aus der Datenbank `patches` werden die Zeilen gezählt, die mit + bzw. - beginnen. Zeilen die mit `---` oder `+++` beginnen, werden nicht mitgezählt, da es sich bei diesen Zeilen um den Dateinamen handelt. Daraus ergibt sich die Anzahl der hinzugefügten und gelöschten Zeilen, die in der Datenbanktabelle `patch_lines` gespeichert werden.

Quelltext 5.3 vergleicht die Ergebnisse der Erweiterung `PatchLOC` und `CommitsLOC` miteinander und listet alle Commits auf, die nicht übereinstimmen. Beim ersten Vergleich zwischen den beiden Erweiterungen gab es eine Vielzahl nicht übereinstimmender Commits. Bei der Suche nach den Ursachen wurden verschiedene Bugs entdeckt und behoben. Die in den Fußnoten jeweils angegebenen Patches beheben die genannten Probleme.

Dateinamen-Erkennung
Dateinamen mit Leerzeichen bereiteten der Erweiterung `Patches` Schwierigkeiten, diese in der Datenbank wiederzufinden[6]. Dateinamen mit unterschiedlicher Groß- und Kleinschreibung führten ebenfalls zu nicht auffindbaren Dateinamen[7]. Außerdem kam es vor, dass Dateien, die ursprünglich auf einem anderen Branch erstellt und bearbeitet wurden, nicht wiedergefunden wurden[8].

Zeichensatz und Zeilenumbrüche
Während der Entwicklung von Infopark CMS wurden die Quelltext-Dateien teilweise in unterschiedlichen Zeichensätzen (engl. „character set" oder auch „charset" genannt) oder Zeilenumbrüchen gespeichert. Git gibt seinen generierten Text (z.B. Entwicklername oder Datum) als UTF-8 aus und verwendet „Line Feed" als Zeilenumbruch.

```
SELECT cl.commit_id,
  cl.added, p.added AS patch_added,
  cl.removed, p.removed AS patch_removed
FROM (
  SELECT commit_id,
    SUM(added) AS added,
    SUM(removed) AS removed
  FROM patch_lines
  GROUP BY commit_id
  ) AS p
RIGHT JOIN commits_lines cl ON p.commit_id = cl.commit_id
WHERE IFNULL(p.added,0) != cl.added OR IFNULL(p.removed,0) != cl.removed
```

Quelltext 5.3: Datenqualität Patches mit PatchLOC und CommitsLOC

[4] LOC - Lines of Code
[5] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/123` sowie `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/124` (Abruf: 01.09.2011)
[6] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/119` (Abruf: 01.09.2011)
[7] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/125` (Abruf: 15.09.2011)
[8] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/129` (Abruf: 15.09.2011)

Der Inhalt des Patches wird jedoch direkt weitergegeben, auch mit dem jeweils verwendeten Zeichensatz und Zeilenumbruch.

Um sicherzustellen, dass Umlaute oder andere Sonderzeichen den weiteren Programmverlauf von CVSAnalY nicht stören, muss der Zeichensatz umgewandelt werden[9]. Außerdem muss bei einer Binärdatei, die Git fälschlicherweise als Text-Datei identifiziert hat, sichergestellt werden, dass ungültige UTF-8 Zeichen entfernt werden[10]. Zusätzlich ist für die Verarbeitung der Patches wichtig, nur „Line Feeds" als Zeilenumbrüche zu verarbeiten[11].

Whitespaces

Wenn eine Zeile nur durch sogenannte Whitespaces (dt. Leerraum, dazu gehören Leerzeichen und Tabulatoren) geändert wurde, speichert Git diese Änderung. Für das Laufzeitverhalten der Software spielt diese Änderung keine Rolle (mit Ausnahme von „Off-Side Rule"-Programmiersprachen, wie z.B. Python). Diese Änderungen verschleiern jedoch, wer ursprünglich die Zeile verfasst hat. Eyolfson et al. [8] schlagen vor, in diesem Fall die Git-Option `-w` bzw. `--ignore-all-space` zu verwenden. Dafür wird der Aufruf von Git in der Erweiterung `Patches` um den Parameter `--ignore-all-space` erweitert[12].

Andere Fehlerquellen

Es zeigte sich, dass die Handhabung von Merge-Commits verbessert werden musste[13].

Außerdem wurde bei der Defektsuche offensichtlich, dass die Erweiterung `CommitsLOC` ihre eigenen Defekte hat. Zum einen wurden umbenannte Dateien als neue Datei gezählt[14]. Zum anderen stellte sich heraus, dass `git log --shortstat` selber Bugs enthält und sich teilweise verzählt. So werden leere Zeilen, die innerhalb eines Hunks sind, *nicht* von `git log --shortstat` mitgezählt. Dies ist ein generelles Problem von allen Git-Statistik-Funktionen (`--stat`, `--shortstat` und `--numstat`)[15].

Nach längerer Korrespondenz mit den Git-Entwicklern per Git-Mailingliste[16] stellte sich heraus, dass Git intern und extern zwei unterschiedliche Patches verwendet. Wenn eine Datei geändert wird, ist es möglich, diese Änderungen mit verschiedenen Patches abzubilden. Dem Tool `diff` kann z.B. der Parameter `--context <zahl>` übergeben werden um festzulegen, wie viele Zeilen vor und nach einer Änderung angezeigt werden sollen. Im Standardfall sind es drei Zeilen Kontext. Git verwendete intern jedoch null Zeilen Kontext, was zu einer anderen Anzahl hinzugefügter und entfernter Zeilen führen kann[17]. Version 1.7.7 löst das Problem, indem der intern verwendete Kontext konfiguriert werden kann[18].

[9] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/115` (Abruf: 15.09.2011)
[10] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/125` sowie `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/133` (Abruf: 15.09.2011)
[11] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/135` (Abruf: 15.09.2011)
[12] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/143` sowie `http://github.com/SoftwareIntrospectionLab/repositoryhandler/pull/19`(Abruf: 22.10.2011)
[13] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/116` (Abruf: 01.09.2011)
[14] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/122` sowie `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/136` (Abruf: 01.09.2011)
[15] siehe Mailingliste `http://www.spinics.net/lists/git/msg163494.html` (Abruf: 01.09.2011)
[16] siehe Mailingliste `http://www.spinics.net/lists/git/threads.html#165859` (Abruf: 30.09.2011)
[17] siehe Mailingliste `http://www.spinics.net/lists/git/msg165936.html` (Abruf: 23.09.2011)
[18] siehe `http://raw.github.com/gitster/git/master/Documentation/RelNotes/1.7.7.txt` (Abruf: 06.10.2011)

Weiter bestehende Einschränkungen
Falls ein Dateiname Umlaute in einem bestimmten Zeichensatz enthält und der Dateiinhalt Umlaute in einem anderen Zeichensatz enthält, wird der Patch nicht gespeichert. Im konkreten Fall handelt es sich um zwei Commits, die jeweils eine Zeile ändern.

In Git ist es möglich, Dateien zu löschen und durch symbolische Links zu ersetzen. Git merkt sich diese Änderung als zwei Patches für einen gegebenen Commit und eine gegebene Datei. Die Erweiterung `Patches` ist jedoch darauf ausgelegt, pro Commit und Datei nur einen Patch zu speichern. Im konkreten Fall handelt es sich um einen Commit, der um genau eine Zeile abweicht.

Datenqualität

Nach dem Beheben der oben genannten Probleme sind insgesamt 263.033 Patches für 31.854 Commits in der Datenbanktabelle `patches` gespeichert. Die SQL-Abfrage aus Quelltext 5.3 zeigt genau drei nicht übereinstimmende Commits auf. Diese lassen sich auf die Punkte aus „Weiter bestehende Einschränkungen" zurückführen. Davon dürfte die Datenqualität nicht erheblich beeinträchtigt sein, da es sich um 3 von 21.995.065 geänderten Zeilen handelt.

5.3 Hunks

Konzepte

Aus jedem Patch der Datenbanktabelle `patches` extrahiert die Erweiterung `Hunks`, welche Zeilen-Abschnitte hinzugefügt und entfernt wurden.

Quelltext 5.4 zeigt ein Beispiel für einen Patch, der zwei Hunks enthält. Es werden sowohl vor als auch nach jeder Änderung bis zu drei Zeilen Kontext angezeigt. Der erste Hunk entfernt zwei Zeilen der Datei `Blame.py` in Zeile 43 und fügt drei neue Zeilen hinzu. Die Erweiterung `Hunks` merkt sich für diesen Hunk, dass die alten Zeilen in der Datei zwischen Zeile 43 und 44 waren.

```
--- a/pycvsanaly2/extensions/Blame.py
+++ b/pycvsanaly2/extensions/Blame.py
@@ -42,8 +42,9 @@ class BlameJob(Job):
         def line(self, line):
-            self.authors.setdefault(line.author, 0)
-            self.authors[line.author] += 1
+            lauthor = to_utf8(line.author).decode("utf-8")
+            self.authors.setdefault(lauthor, 0)
+            self.authors[lauthor] += 1

         def end_file(self):
             pass
@@ -150,7 +151,7 @@ class Blame(Extension):
             import MySQLdb

             try:
-                cursor.execute("""CREATE TABLE blame ("
+                cursor.execute("""CREATE TABLE blame (
                                    id integer primary key not null,
                                    file_id integer,
                                    commit_id integer,
```

Quelltext 5.4: Beispiel für einen Unified Diff Patch

Die geänderten Zeilen sind in der neuen Datei zwischen Zeile 43 und 45. Somit verschieben sich alle folgenden Zeilen um eine Zeile. Der zweite Hunk ändert die Zeile 153. `Hunks` merkt sich eine Änderung der alten Zeile 153 und der neuen Zeile 154.

Damit lässt sich genau rekonstruieren, welche Zeilen einer Datei in einem Commit geändert wurden und welcher Zeilennummer dies in der vorherigen Revision entspricht. Es werden nur Patches verarbeitet, die Text-Dateien enthalten. Binäre Dateien werden ignoriert.

Anwendung und Datenqualität

Ein paar Defekte mussten beseitigt werden, um die Datenqualität nicht einzuschränken. Neben einem Cut-and-paste-Defekt[19] wurden gelöschte Dateien in der Erweiterung `Hunks` nicht berücksichtigt[20]. Die in den Fußnoten benannten Patches beheben die Probleme.

Messinstrumente

Die Ergebnisse der Erweiterung `Hunks` lassen sich mit der Erweiterung `CommitsLOC` und `PatchLOC` überprüfen. Quelltext 5.5 vergleicht, inwieweit `Hunks` von `CommitsLOC` abweicht. In der konkreten Anwendung sind es genau die gleichen drei Einträge, die auch schon bei der Erweiterung `Patches` abgewichen sind. Mit Quelltext 5.6 lässt sich zusätzlich nicht nur auf Commit-, sondern auch auf Datei-Ebene überprüfen, inwieweit `PatchLOC` und `Hunks` übereinstimmen.

Dabei zeigt sich, dass `Hunks` und `PatchLOC` vollständig übereinstimmen und keine Abweichungen existieren. Somit ist jeder Patch, der in `patches` gespeichert ist, auch von der Erweiterung `Hunks` korrekt verarbeitet worden.

```
SELECT cl.commit_id,
  cl.added, h.added AS calc_added,
  cl.removed, h.removed AS calc_removed
FROM (
  SELECT commit_id,
    SUM(new_end_line - new_start_line + 1) AS added,
    SUM(old_end_line - old_start_line + 1) AS removed
  FROM hunks
  GROUP BY commit_id
  ) AS h
RIGHT JOIN commits_lines cl ON h.commit_id = cl.commit_id
WHERE IFNULL(h.added,0) != cl.added OR IFNULL(h.removed,0) != cl.removed
```

Quelltext 5.5: Datenqualität Hunks mit CommitsLOC

```
SELECT pl.commit_id, pl.file_id,
  pl.added, h.added AS calc_added,
  pl.removed, h.removed AS calc_removed
FROM (
  SELECT h.commit_id, h.file_id,
    SUM(new_end_line - new_start_line + 1) AS added,
    SUM(old_end_line - old_start_line + 1) AS removed
  FROM hunks h
  GROUP BY commit_id, file_id
  ) AS h
RIGHT JOIN patch_lines pl ON h.commit_id = pl.commit_id AND h.file_id = pl.file_id
WHERE IFNULL(h.added,0) != pl.added OR IFNULL(h.removed,0) != pl.removed
```

Quelltext 5.6: Datenqualität Hunks mit PatchLOC

[19] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/113` (Abruf: 01.09.2011)
[20] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/114` (Abruf: 01.09.2011)

Insgesamt sind 637.301 Hunks für 31.426 Commits in der Datenbanktabelle `hunks` gespeichert.

5.4 HunkBlame

Konzepte

Nachdem die Erweiterung `Hunks` die geänderten Zeilenabschnitte jeder Datei und Revision identifiziert hat, kann `HunkBlame` deren Ursprung ermitteln.

Jede Änderung in einer Datei eines Commits, der als Bugfix markiert ist, wird verarbeitet. Dabei wird von `HunkBlame` zunächst ermittelt, wann diese Datei zuletzt *vor* dem besagten Commit geändert wurde. Außerdem muss `HunkBlame` den vorherigen Dateinamen ermitteln, da es sich um eine umbenannte oder kopierte Datei handeln kann. Anschließend wird auf der ermittelten Revision und dem ermittelten Dateinamen anhand von `git blame` ausgewertet, in welchem Commit die beanstandeten Zeilen zuletzt geändert wurden. Diese Information verwaltet `HunkBlame` in der Datenbanktabelle `hunk_blames`.

Tabelle 5.1 zeigt ein Beispiel einer Datei, die viermal geändert wurde. Wenn nun Commit D als Bugfix-Commit identifiziert wurde, ermittelt `HunkBlame` im ersten Schritt Commit C als die Revision, in der die Datei zuletzt vor Commit D geändert wurde. Im zweiten Schritt wird `git blame` auf die Datei mit dem Zustand von Commit C angewendet. Die erste beanstandete Zeile wurde im Commit A, die zweite im Commit B und die dritte im Commit C zuletzt geändert. Nach dem SZZ-Algorithmus handelt es sich bei Commit A, B und C um „fix-inducing changes", die somit die Ursache für den Bugfix sind. `HunkBlame` speichert diese Information in der Datenbank.

Anwendung

Bei der Anwendung der Erweiterung `HunkBlame` zeigte sich, dass die Ermittlung der vorherigen Revision und des vorherigen Dateinamens nicht trivial sind. Als erster Versuch wurden vom Entwicklerteam die Revision und der Dateiname über die Datenbank ermittelt. Dabei wurde für eine gegebene Datei und Revision in der Datenbank angefragt, welcher zeitlich der letzte Commit war, bei dem die Datei geändert wurde. Dies führt dann zu falschen Ergebnissen, wenn die Historie nicht chronologisch ist. In Git ist es mit dem Befehl `git rebase -i` möglich, die Reihenfolge der Commits beliebig zu ändern und somit auch einen Commit, der zeitlich früher erstellt wurde, nach einem Commit, der später erstellt wurde, zu platzieren.

Im Rahmen dieser Diplomarbeit sind verschiedene Verbesserungen zum Bestimmen der vorherigen Revision und des vorherigen Dateinamens umgesetzt und erprobt worden. Wie bereits im Abschnitt 3.2 (Dateipfad-Problem) ausgeführt, wurde die Bestimmung von Dateinamen überarbeitet.

Außerdem wurde eine Funktion erstellt, die die Ausgabe von `git log` auswertet und damit auf Git-Ebene bereits die vorherige Revision für eine Datei ermittelt[21]. In einer späteren Iteration wurde zusätzlich

Zeile	Commit A	Commit B	Commit C	Commit D
1	`def hello():`	`def hello():`	`def hello():`	`def hello_world():`
2	`  foo`	`  bar`	`  bar`	`  baz`
3	`  21`	`  21`	`  42`	`  23`

Tab. 5.1: Änderung einer Datei über vier Commits

[21] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/89` (Abruf: 30.09.2011)

der vorherige Dateiname ermittelt[22]. Außerdem hatte das Hilfsprogramm Guilty[23], welches von `HunkBlame` zum Parsen von `git blame` verwendet wird, Probleme, Dateinamen mit Leerzeichen zu erkennen[24].

Whitespaces
Wie in der Erweiterung `Patches` (siehe Abschnitt 5.2), ist es sinnvoll, Zeilen, in denen nur Whitespaces geändert werden, als Quelle zu ignorieren. `git blame` erlaubt ebenfalls die Option `-w` um Whitespaces zu ignorieren[25]. Auch das Hilfsprogramm Guilty musste um die Option `-w` erweitert werden[26].

Weiter bestehende Einschränkungen
In Git ist es mit `git submodule` möglich, auf bestimmte Versionen von anderen Repositorys zu verweisen[27]. Wenn eine andere (z.B. neuere) Version dieses anderen Repositorys verwendet werden soll, zeigt Git dies als „normalen" Diff an. Diese Änderung kann jedoch *nicht* mit `git blame` nachvollzogen werden und ist somit für `HunkBlame` nicht verarbeitbar.

Datenqualität

Mit Quelltext 5.7 lässt sich bestimmen, welche Hunks, die als Bugfix-Commit markiert wurden, *nicht* von `HunkBlame` verarbeitet wurden. Nach der Anwendung der oben genannten Patches gibt es nur acht Hunks, die abweichen. Alle acht abweichenden Hunks lassen sich auf Submodule zurückführen.

Insgesamt wurden 88.226 Blames für 3.576 Commits in der Datenbanktabelle `hunk_blames` gespeichert.

5.5 HunkBlame Erweiterung

Bisher wurde in `HunkBlame` für *jede* geänderte Zeile der Ursprung ermittelt. Dies gilt auch für:

- Leere Zeilen
- Zeilen, die nur Quelltext-Kommentare enthalten

Im Rahmen dieser Diplomarbeit wurde `HunkBlame` erweitert, um die eben genannten Fälle zu erkennen und *nicht* zu verarbeiten. Dafür wird mit Hilfe des Syntax-Highlighters Pygments[28] für jede Zeile einer

```
SELECT rev, current_file_path
FROM hunks h
JOIN scmlog s ON h.commit_id = s.id
JOIN actions a ON a.commit_id = h.commit_id AND a.file_id = h.file_id
WHERE h.id NOT IN (SELECT hunk_id AS id FROM hunk_blames hb)
  AND h.old_start_line IS NOT NULL
  AND h.old_end_line IS NOT NULL
  AND h.file_id IS NOT NULL
  AND h.commit_id IS NOT NULL
  AND s.is_bug_fix = TRUE
GROUP BY h.commit_id, h.file_id
```

Quelltext 5.7: Datenqualität HunkBlame

[22] siehe Patches `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/137` (Abruf: 04.10.2011), `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/140` (Abruf: 04.10.2011) und `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/141` (Abruf: 04.10.2011)
[23] siehe `http://github.com/SoftwareIntrospectionLab/guilty` (Abruf: 04.10.2011)
[24] siehe Patch `http://github.com/SoftwareIntrospectionLab/guilty/pull/1` (Abruf: 04.10.2011)
[25] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/143` (Abruf: 22.10.2011)
[26] siehe Patch `http://github.com/SoftwareIntrospectionLab/guilty/pull/2` (Abruf: 22.10.2011)
[27] mehr zu `git submodule` siehe `http://book.git-scm.com/5_submodules.html` (Abruf: 04.10.2011)
[28] mehr Informationen unter `http://pygments.org` (Abruf: 30.09.2011)

Quelltext-Datei ermittelt, ob es sich um eine leere Zeile, einen Kommentar oder ausführbaren Code handelt. Nur wenn es sich bei der Zeile um ausführbaren Code handelt, wird sie weiter verarbeitet[29].

Pygments kann für eine Vielzahl von Programmiersprachen den Code farbig hervorheben und wird unter anderem von GitHub eingesetzt[30]. Auch das Syntax-Highlighting dieser Diplomarbeit ist mit Pygments erstellt worden (siehe z.B. Quelltext 5.7).

Pygments ermittelt zuerst, welcher lexikalische Scanner (kurz Lexer oder auch Tokenizer) für den Quelltext verwendet werden soll. Dieser Lexer ist je nach Programmiersprache unterschiedlich. Pygments bietet eine Vielzahl an Lexern für verschiedene Sprachen an, z.B. für Java, Objective-C, Ruby, Tcl, HTML, CSS und JavaScript. Um den zu verwendenden Lexer zu ermitteln, wird zuerst überprüft, ob anhand des Dateinamens ein Lexer zugeordnet werden kann. So wird z.B. der Datei `webservice_controller.rb` der Ruby-Lexer zugewiesen, da der Dateiname mit `.rb` endet. Falls der Lexer nicht anhand des Dateinamens bestimmt werden kann, probiert Pygments anhand des Dateiinhalts einen passenden Lexer zu erraten. Falls dies immer noch nicht funktioniert, wird der Standard-Lexer `TextLexer` verwendet.

Anschließend wird der Code vom Lexer in Tokens zerlegt. So ein Token kann z.B. `Token.Comment.Single` oder `Token.Comment.Multiline` für ein und mehrzeilige Kommentare oder `Token.Text` für Text sein. Andere Tokens wie `Token.Number` oder `Token.Operator` sind ebenfalls möglich.

Anschließend werden diese Tokens von Pygments verarbeitet und je nach „Style" farblich dargestellt.

Für die Unterstützung von `HunkBlame` werden nur der Lexer und die Tokens benötigt. Nachdem die Tokens erstellt wurden, werden sie Zeilen zugeordnet. Um nun je Zeile zu ermitteln, ob es sich um ausführbaren Code handelt, wird sichergestellt, dass es sich nicht um eine leere Zeile handelt (`Token.Text` und leerem Zeileninhalt). Ebenso wird überprüft, dass die Zeile nicht mit einem Kommentar beginnt (`Token.Comment.*`).

Insgesamt wurden mit den beschriebenen Erweiterungen von `HunkBlame` 81.748 Blames für 3.539 Commits in der Datenbanktabelle `hunk_blames` gespeichert.

[29] siehe Patch `http://github.com/SoftwareIntrospectionLab/cvsanaly/pull/145` (Abruf: 02.11.2011)
[30] siehe `http://pygments.org/projects` (Abruf: 4.10.2011)

Kapitel 6

SZZ-Algorithmus bewerten und überprüfen

Im Folgenden wird untersucht, ob die Annahmen des SZZ-Algorithmus und dieser Diplomarbeit sich bestätigen.

6.1 Terminologie

Zunächst müssen noch einige Begriffe erklärt werden. Es handelt sich um ein *Versagen* (engl. „failure") einer Software, wenn ihr Verhalten nicht korrekt ist. Dieses Versagen tritt aufgrund eines oder mehrerer *Defekte* (engl. „defect" oder „fault") in der Software auf. Der Defekt ist durch einen *Fehler* (engl. „error") des Entwicklers entstanden. Solch ein Fehler kann entweder ein Versäumnis (engl. „omission") oder „Falschtun" (engl. „commission") des Entwicklers sein (siehe [15] Folie 4 und Abbildung 6.1). Die Unterscheidung zwischen Versagen, Defekt und Fehler wird in der Praxis selten vorgenommen, weshalb meistens die Begriffe Fehler, Defekt oder das englische „Bug" als Synonym oder Sammelbegriff verwendet werden.

Die Korrektheit einer Software kann durch verschiedene Quellen definiert werden (siehe [15] Folie 10):

- Formale Spezifikation
- Explizite Benutzeranforderung
- Implizite Benutzererwartung

Ein Beispiel für eine *formale Spezifikation* ist ein Wert, der nach einer bestimmten mathematischen Formel errechnet werden soll. Wenn dieser Wert falsch berechnet wird, handelt es sich um ein nicht korrektes Verhalten. Bei vielen Softwareprojekten ist eine formale Spezifikation nicht vorhanden, so auch bei Infopark CMS. Stattdessen sind meistens *explizite Benutzeranforderungen* gegeben. Diese existieren jedoch nur als Karten an einem Storyboard und können nachträglich nicht mehr der Implementierung zugeordnet werden. Eine Benutzeranforderung für ein CMS kann die Auswahl einer Sprache für die Benutzeroberfläche sein. Wenn beim Auswählen nichts passiert, handelt es sich um nicht korrektes Verhalten. Außerdem gibt es *implizite Erwartungen* des Benutzers an eine Software. So soll das Veröffentlichen einer Seite *nicht* dazu führen, dass andere Seiten gelöscht werden. Auch hier handelt es sich um nicht korrektes Verhalten.

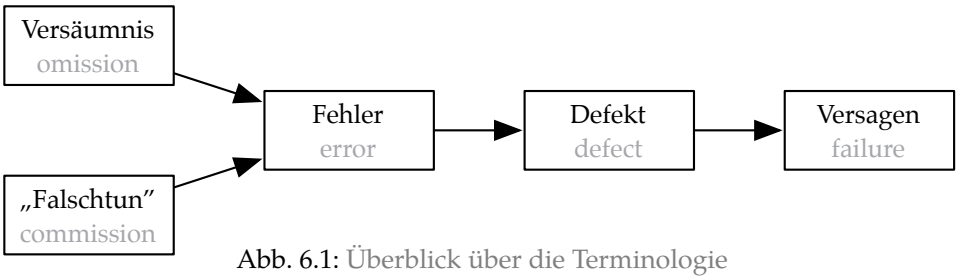

Abb. 6.1: Überblick über die Terminologie

Um nicht korrektes Verhalten im Nachhinein zu dokumentieren und zu reparieren, wird meistens ein Bugtracker eingesetzt. Ein Bugtracker ermöglicht es, nicht korrektes Verhalten als Bugreport zu melden und z.B. zu priorisieren. Da im Besonderen implizite Benutzererwartungen subjektiv sein können, sind die Übergänge zwischen Bug und neuer Funktionalität (engl. „feature") fließend. Die in der Softwareentwicklung verbreitete Redewendung spiegelt dieses Spannungsfeld wider:

„It's not a bug, it's a feature."[1]

Defekte im engeren Sinne setzen voraus, dass bereits zum Zeitpunkt der Codeerstellung die Erwartung an den Code bekannt ist. Wenn diese Erwartung zu einem späteren Zeitpunkt von der Software nicht erfüllt wird, handelt es sich um einen Defekt im engeren Sinne. Oft sind diese Erwartungen nicht zum Zeitpunkt der Codeerstellung bekannt und müssen deshalb als Anforderungsänderung (engl. „Change Request") oder als neue Funktionalität (engl. „feature") verstanden werden. Im Weiteren wird der Begriff „echter Defekt" synonym für Defekt in engeren Sinne verwendet.

In den nächsten drei Abschnitten sollen die zugrundeliegenden Annahmen anhand der vorhandenen Daten überprüft werden.

6.2 Bugreports

> Handelt es sich bei den verlinkten Bugreports um Meldungen defektbehafteten Codes oder handelt es sich um Feature-Requests?

Der bei Infopark eingesetzte Bugtracker Bugzilla bietet bereits mit dem Feld `severity` (dt. Schweregrad) die Möglichkeit, einen Bugreport als „Bug" oder als „Feature" zu klassifizieren. Tabelle 6.1 listet die Verteilung der Einstufung der verlinkten Bugreports auf. Mehr als 80% der verlinkten Bugreports sind als Bug mit dem Schweregrad Normal oder höher eingestuft. Dabei ist zu beachten, dass der Standardwert für den Schweregrad bei einem neuen Bugreport „3/Bug: Normal" ist.

Um besser beurteilen zu können, ob die von den Autoren des Bugreports vorgenommene Einteilung richtig ist, wird eine Stichprobe manuell überprüft. Für die Stichprobe werden 60 zufällig ausgewählte verlinkte Bugreports (etwa 2% der gesamten Datenmenge) inspiziert und daraufhin überprüft, ob es sich um echte Defekte handelt.

Tabelle 6.2 listet die Ergebnisse der Stichprobe auf. Demnach sind insgesamt ca. 75% der inspizierten Bugreports echte Defekte. Das heißt, dass ca. 25% aller als Bug klassifizierten Bugreports *keine* echten Defekte, sondern Feature-Requests oder Ähnliches sind.

Schweregrad	Anzahl	Anteil an Summe
1/Bug: Kritisch	238	8%
2/Bug: Hoch	883	30%
3/Bug: Normal	1275	43%
4/Bug: Gering	392	13%
Wunsch/Erweiterung	153	5%
Summe	2941	100%

Tab. 6.1: Verteilung des Schweregrades der verlinkten Bugreports

[1] Die Herkunft der Aussage ist unbekannt. Siehe `http://bits.blogs.nytimes.com/2008/03/25/did-bill-gates-really-say-that/` (Abruf: 13.10.2011), insbesondere die Kommentare zum Artikel.

Schweregrad	Anzahl	Anteil an Stichprobe	echte Defekte	echte Defekte pro Anzahl
1/Bug: Kritisch	7	12%	7	100%
2/Bug: Hoch	14	23%	12	86%
3/Bug: Normal	28	47%	18	64%
4/Bug: Gering	8	13%	6	75%
Summe der Bugs	57	95%	43	75%
Wunsch/Erweiterung	3	5%	1	33%

Tab. 6.2: Verteilung der Stichprobe über 60 verlinkte Bugreports

Alle als Bug eingestuften Bugreports enthalten, mit Ausnahme der kritischen Bugreports, Feature-Requests. Besonders der Schweregrad „3/Bug: Normal" enthält (mit 18 von 28 Bugreports) im Verhältnis die wenigsten echten Defekte der als Bug eingestuften Bugreports. Dies lässt sich auf die Zuweisung als Standardwert zurückführen. Neben Feature-Requests sind im „3/Bug: Normal"-Schweregrad auch To-Dos enthalten, wie z.B. die Aufgabe, eine externe Programmbibliothek zu aktualisieren. Von den drei Bugreports, die als Wunsch/Erweiterung eingestuft wurden, beschreibt einer einen Defekt im engeren Sinne.

Insgesamt handelt es sich bei ca. einem Viertel aller Bugreports *nicht* um Defekte im engeren Sinne. Hier besteht weiterer Forschungsbedarf, um die Unterscheidung zwischen Bug und Feature-Request zu verbessern.

Es gibt einige Möglichkeiten, die Datenqualität des Bugtrackers in Zukunft zu verbessern. Zum einen sollte die Kategorisierung von dem Ersteller des Bugreports besser vorgenommen werden. Ob es sich um einen Bug handelt, sollte separat von dem Schweregrad eingestuft werden. Zuletzt sollte der Standardwert für neue Bugreports nicht mehr „3/Bug: Normal", sondern leer sein.

Solange keine besseren Methoden vorhanden sind, um Bugreports zu klassifizieren, kann eine Einteilung anhand des Schweregrades vorgenommen werden. Dabei werden nur die „wichtigen Bugreports" als Bugreports gewertet. „Wichtige Bugreports" sind alle verlinkten Bugreports, die als „2/Bug: Hoch" oder „1/Bug: Kritisch" eingestuft sind und die *nicht* das Wort „To-do" im Titel enthalten. Damit soll sichergestellt werden, dass offensichtliche Feature-Requests nicht als Defekt gewertet werden. In den verwendeten Daten existieren 1.059 wichtige Bugreports.

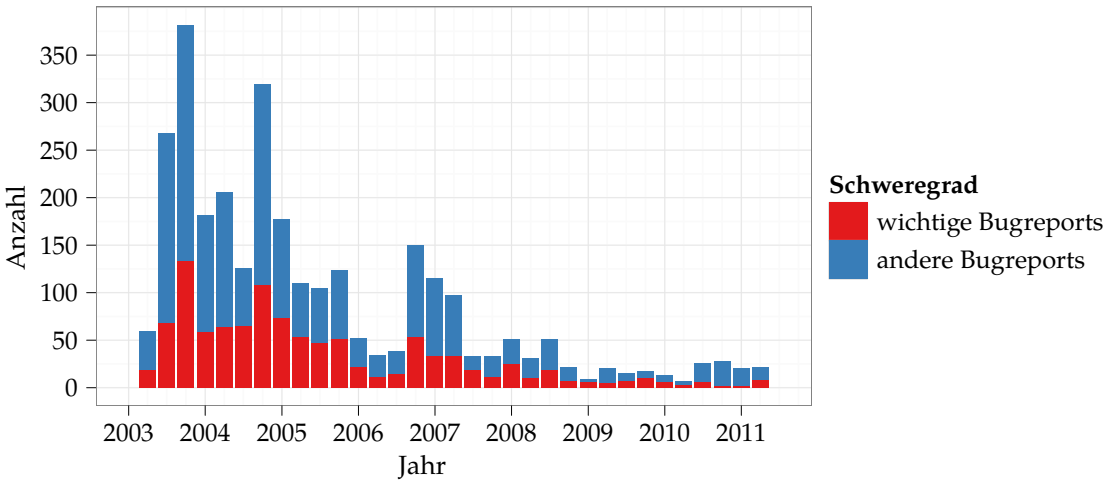

Abb. 6.2: Anzahl der verlinkten Bugreports pro Quartal, aufgeteilt in zwei Einstufungen

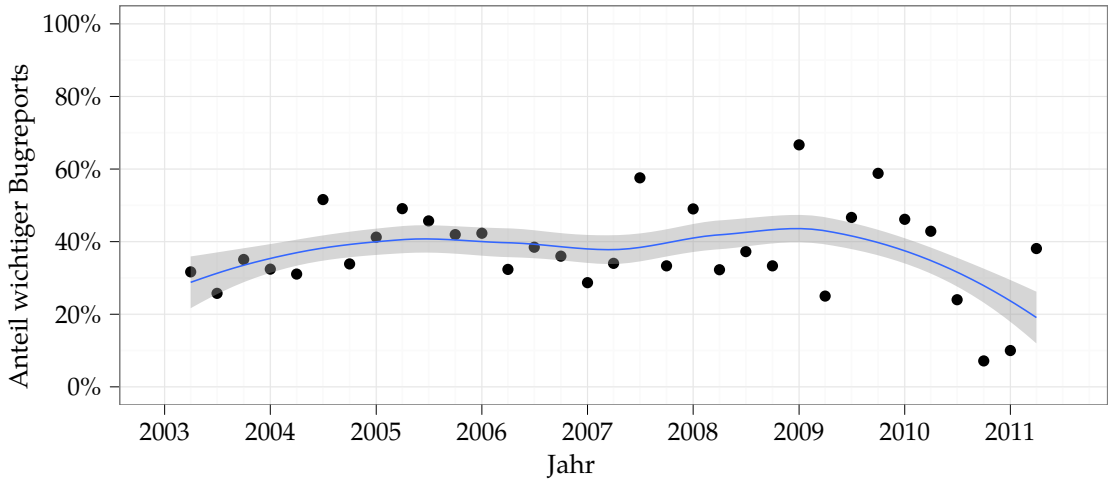

Abb. 6.3: Anteil der wichtigen Bugreports pro Gesamtzahl der verlinkten Bugreports pro Quartal

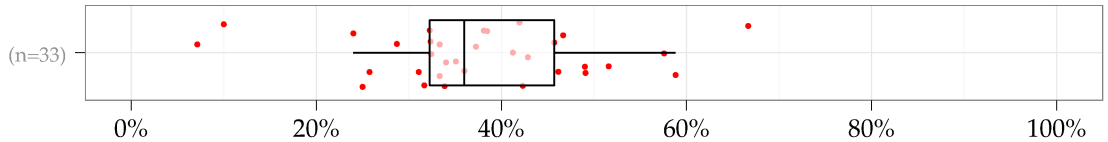

Abb. 6.4: Boxplot über den Anteil der wichtigen Bugreports im Verhältnis zu anderen Bugreports pro Quartal

Abbildung 6.2 und Abbildung 6.3 zeigen den zeitlichen Verlauf der verlinkten Bugreports für wichtige und andere Bugreports. Abbildung 6.4 stellt die Verteilung des Verhältnisses zwischen wichtigen und anderen Bugreports dar.

Wie Abbildung 6.3 und Abbildung 6.4 zeigen, ist in jedem Quartal der Anteil der wichtigen Bugreports bei mindestens 7%, im Median sogar bei 36%. Somit können die wichtigen Bugreports als repräsentative Datenmenge verwendet werden. Im Weiteren wird zur Unterscheidung entweder von allen verlinkten Bugreports oder von wichtigen Bugreports gesprochen.

6.3 Bugfix-Commits

> Behebt ein verlinkter Bugfix-Commit wirklich den beschriebenen Defekt des Bugreports?

Abbildung 6.5 und Abbildung 6.6 stellen vergleichend dar, ob Bugfix-Commits mehr oder weniger Zeilen bzw. Dateien als Nicht-Bugfix-Commits ändern. Die Bugfix-Commits werden in wichtige und andere Bugfix-Commits aufgeteilt (wie in Abschnitt 6.2 beschrieben). Insgesamt ist festzuhalten, dass im Vergleich Bugfix-Commits *mehr* Zeilen und Dateien pro Commit ändern als die Gruppe der Nicht-Bugfix-Commits. Sowohl der Median als auch das erste und dritte Quartil sind in der Bugfix-Commit-Gruppe höher als die der Nicht-Bugfix-Commit-Gruppe.

Der Vergleich zwischen wichtigen Bugfix-Commits und anderen Bugfix-Commits zeigt nur kleine Unterschiede. Bei den geänderten Zeilen pro Commit ist das erste Quartil (5 Zeilen) gleich. Der Median der wichtigen Bugfix-Commit-Gruppe (15 Zeilen) liegt um eine Zeile unter dem Median der anderen Bugfix-Commit-Gruppe (16 Zeilen). Das dritte Quartil ist jedoch in der Wichtige-Bugfix-Commits-Gruppe höher (57 Zeilen) als in der anderen Bugfix-Commit-Gruppe (53 Zeilen). Bei den geänderten Dateien pro Commit sind das erste Quartil (2 Dateien) und der Median (4 Dateien) in beiden Gruppen gleich.

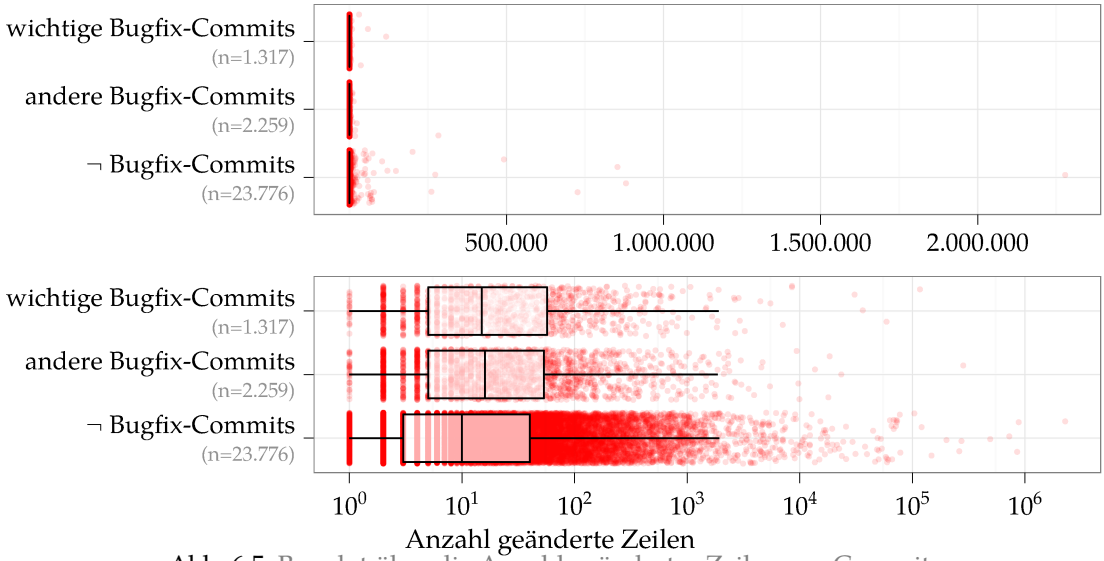

Abb. 6.5: Boxplot über die Anzahl geänderter Zeilen pro Commit

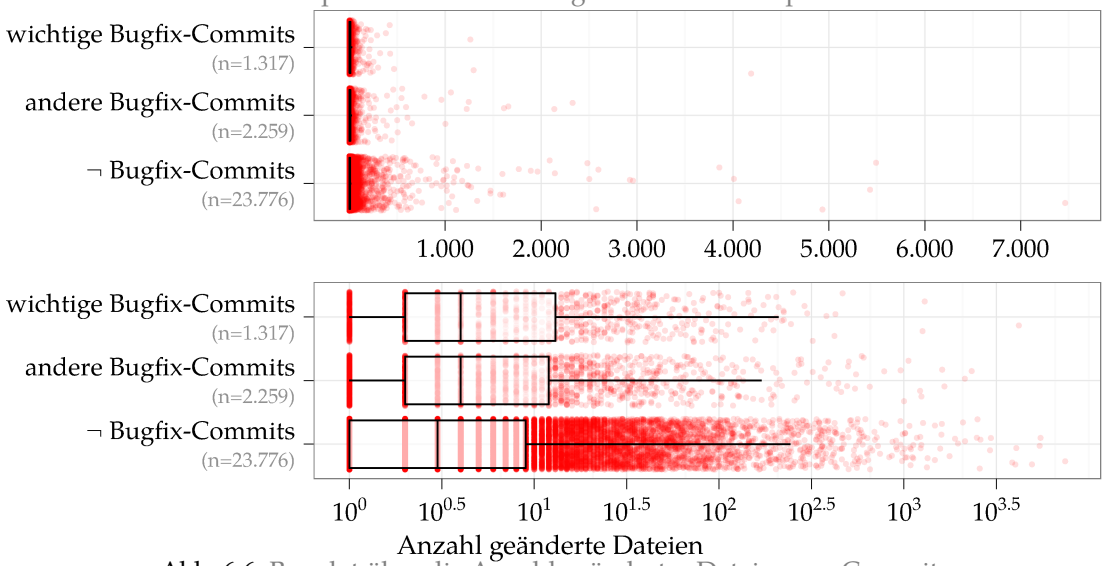

Abb. 6.6: Boxplot über die Anzahl geänderter Dateien pro Commit

Das dritte Quartil der wichtigen Bugfix-Commit-Gruppe (13 Dateien) ist jedoch um eine Datei größer als das der anderen Bugfix-Commit-Gruppe (12 Dateien). Dabei handelt es sich lediglich um kleinere Abweichungen, weshalb davon ausgegangen werden kann, dass die beiden Gruppen in Bezug auf geänderte Zeilen und Dateien sehr ähnlich sind.

Ausnahmen

Im folgenden Abschnitt werden verschiedene Fälle beschrieben, bei denen es sich *nicht* um Reparaturen des verlinkten Bugreports handelt. Pro Ausnahme wird beschrieben, inwiefern sie die Datenqualität beeinflusst und ob es Möglichkeiten gibt, diese Ausnahmen zu identifizieren.

Automatisierte Tests
Einige verlinkte Bugfix-Commits enthalten keine Reparaturen, sondern einen automatisierten Test, der den Defekt überprüfbar macht. Der automatisierte Test kann in einem separaten Commit oder in Kombination mit der eigentlichen Reparatur enthalten sein.

Um Dateien, die automatisierte Test enthalten, *nicht* zu überprüfen, können diese anhand von Dateinamen-Heuristiken identifiziert werden. Wenn z.B. der Text „test" im Dateipfad vorkommt, handelt es sich sehr wahrscheinlich um einen Test. Außerdem können Dateipfade, die den Text „spec/" enthalten, als Test angenommen werden, da z.B. die Test-Umgebung RSpec (für Ruby bzw. Ruby On Rails) diese verwendet. Dabei ist zu beachten, dass nicht der Text „inspector" gematchet wird, weshalb der Slash am Ende mit angegeben ist. Mit dieser Heuristik lassen sich 2.076 von 30.489 (ca. 7%) Dateien als automatisierte Tests innerhalb von verlinkten Bugfix-Commits identifizieren.

Externe Bibliotheken
Wie bereits in Abschnitt 6.2 beschrieben, können Bugreports ToDos enthalten, wie z.B. die Aufgabe eine externe Bibliothek zu aktualisieren. Dabei handelt es sich selten um Reparaturen. Bei Infopark werden alle externen Bibliotheken, die im Repository vorhanden sind, im Verzeichnis `External/` oder `External-osdep/` gespeichert. Um Bugfix-Commits von externen Bibliotheken zu bereinigen, kann überprüft werden, ob der Dateipfad mit „External" anfängt. Im konkreten Fall handelt es sich um 45 verlinkte Bugfix-Commits, denen 18 wichtige Bugreports zugeordnet sind.

Vorläufige Reparaturen
Außerdem können verlinkte Bugfix-Commits nur einen „Quick-Fix" enthalten. Diese Quick-Fix-Commits können für die Analyse trotzdem nützlich sein, da die geänderten Zeilen offensichtlich mit dem Defekt in Verbindung stehen.

Verteilung verlinkter Bugfix-Commits

Abbildung 6.7 zeigt die Anzahl der verlinkten Bugfix-Commits pro Bugreport sowohl für wichtige als auch für andere Bugreports. Selbst das dritte Quartil ist in beiden Gruppen bei 2 verlinkten Bugreports. In Abschnitt 4.2 wurden bereits Maßnahmen erörtert und durchgeführt, um sicherzustellen, dass die verlinkten Bugfix-Commits auf den richtigen Bugreport verweisen. Deshalb kann davon ausgegangen werden, dass die Mehrzahl der Bugreports den passenden Bugfix-Commit verlinken.

6.4 Bug-Commits

Abbildung 6.8 zeigt die Anzahl der Bug-Commits pro Bugfix-Commit, die der SZZ-Algorithmus identifiziert hat. Dabei wurden die Daten um automatisierte Tests und externe Bibliotheken bereinigt (siehe Abschnitt 6.3).

> Handelt es sich bei jedem der gefundenen Bug-Commits wirklich um den Ursprung des Defekts oder zumindest um defektbehafteten Code?

Um diese Frage zu beantworten, werden im nächsten Abschnitt Ausnahmen vorgestellt, bei denen der SZZ-Algorithmus falsche oder keine Ergebnisse liefert. Im darauf folgenden Abschnitt werden verschiedene Stichproben auf ihre Korrektheit hin untersucht.

Es zeigt sich, dass die beiden Gruppen „wichtige" und „andere" Bugreports sich kaum unterscheiden. Der Median aller Werte liegt bei 2 Bug-Commits pro verlinktem Bugfix-Commit. Das dritte Quartil liegt bei 6 Bug-Commits.

Abb. 6.7: Boxplot über die Anzahl Bugfix-Commits pro Bugreport

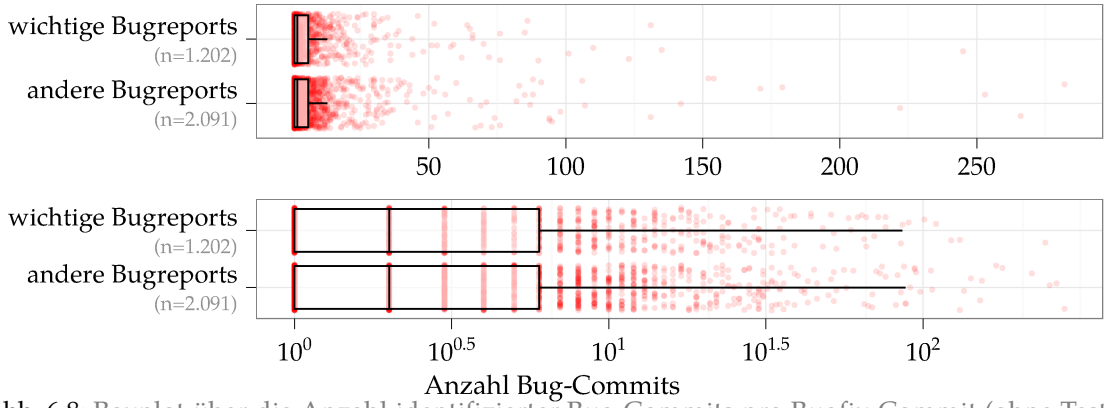

Abb. 6.8: Boxplot über die Anzahl identifizierter Bug-Commits pro Bugfix-Commit (ohne Tests und externe Bibliotheken)

6.4.1 Ausnahmen

Eine der zugrundeliegenden Annahmen des SZZ-Algorithmus ist es, dass jede Zeile, die in einem Bugfix-Commit geändert wird, vorher defektbehaftet war. Im Folgenden werden verschiedene Fälle aufgeführt, in denen der SZZ-Algorithmus keine oder fehlerhafte Ergebnisse liefert. Außerdem wird erörtert, welche Lösungsmöglichkeiten existieren.

Versäumnisse

Wie bereits in Abschnitt 6.1 ausgeführt, können Defekte aufgrund eines „Falschtuns" oder eines Versäumnisses entstehen. Versäumnisse können jedoch vom SZZ-Algorithmus *nicht* oder nur schwer identifiziert werden.

Quelltext 6.1 zeigt einen Bugfix-Commit, bei dem in einer Konfigurationsdatei eine benötigte Bibliothek deklariert wird. Zu einem früheren Zeitpunkt wurde vergessen, diese einzubinden, weshalb es zu einem Defekt kam. Der SZZ-Algorithmus kann in diesem Fall *nicht* nachvollziehen, wem das Versäumnis unterlaufen ist.

Selbst wenn es vorher nur eine einzige Änderung der Datei gab (deren Erstellung), ist immer noch nicht sicher, ob zu diesem Zeitpunkt die Bibliothek bereits benötigt wurde. Um dieses Versäumnis richtig zuordnen zu können, muss festgestellt werden, wann die Library zuerst benötigt wurde. Hierfür besteht Forschungsbedarf.

```
diff a/Apps/PM/webapps/PM-EX/WEB-INF/web.xml b/Apps/PM/webapps/PM-EX/WEB-INF/web.xml
--- a/Apps/PM/webapps/PM-EX/WEB-INF/web.xml
+++ b/Apps/PM/webapps/PM-EX/WEB-INF/web.xml
@@ -57,6 +57,13 @@
         <url-pattern>/debug/*</url-pattern>
     </servlet-mapping>

+    <jsp-config>
+      <taglib>
+        <taglib-uri>http://java.sun.com/portlet</taglib-uri>
+        <taglib-location>/META-INF/portlet.tld</taglib-location>
+      </taglib>
+    </jsp-config>
+
 </web-app>
```

Quelltext 6.1: Beispiel Bugfix-Diff für ein Versäumnis

Whitespaces

Werden in einer Zeile nur Whitespaces geändert, so werden diese Änderungen beim Patch angezeigt. Der SZZ-Algorithmus analysiert die geänderten Zeilen ungeachtet ihres Inhalts. Somit wird auch eine geänderte Zeile analysiert, in der nur Whitespaces geändert werden.

Eine Möglichkeit, das Problem zu lösen, besteht darin, bereits beim Erstellen der Patches darauf zu achten, dass Whitespaces ignoriert werden.

Außerdem kommt es vor, dass ein Commit, der in einer Zeile nur Leerraum ändert, fälschlicherweise als Bug-Commit identifiziert wird. Wie bereits in Abschnitt 5.2 und 5.4 beschrieben, wurde der Aufruf von Git um den Parameter `-w` bzw. `--ignore-all-space` erweitert. Diese Änderungen lösen das Problem.

Quelltext-Kommentare

Wie bereits im vorausgegangenen Abschnitt ausgeführt, analysiert der SZZ-Algorithmus jede geänderte Zeile unabhängig von ihrem Inhalt. Zeilen, die nur aus Quelltext-Kommentaren bestehen, haben in den meisten Fällen keinen Einfluss auf das Laufzeitverhalten der Anwendung. Die Erweiterung aus Abschnitt 5.5 behebt das Problem. Zu diesem Zweck wird jede Zeile vom Programm Pygmentize verarbeitet und daraufhin überprüft, ob es sich um eine Kommentarzeile handelt.

Refactorings

Weitere Änderungen, die der SZZ-Algorithmus fälschlicherweise analysiert, sind Refactorings, d.h. insbesondere semantisch irrelevante Änderungen.

Umbenennung von Variablen
Wenn etwa eine Variable konsistent umbenannt wird, so hat dies keine Auswirkungen auf das Laufzeitverhalten der Anwendung. In Quelltext 6.2 wird die Variable `tip` in `hint` umbenannt. Der englische Begriff „tip" (dt. Trinkgeld) ist zwar die falsche Übersetzung des deutschen „Tipp", jedoch hat die Benennung keine Auswirkung auf das Laufzeitverhalten der Software.

Zeilenumbrüche
Quelltext 6.3 zeigt einen vermeintlichen Bug-Commit, der die Änderung aus dem Bugfix-Commit Quelltext 6.4 verursacht haben soll. Im Bugfix-Commit Quelltext 6.4 wird der Wert von `objectId` zu `id` geändert, um einen Defekt zu reparieren. Die Änderung in Quelltext 6.3 sorgt nur dafür, dass das Programm-Statement nicht mehr über zwei Zeilen verteilt ist. Ansonsten sind die Zeilen vom Inhalt her identisch und

```
diff a/Apps/Packaging/Installer/setup.sh b/Apps/Packaging/Installer/setup.sh
--- a/Apps/Packaging/Installer/setup.sh
+++ b/Apps/Packaging/Installer/setup.sh
@@ -224,12 +224,12 @@ checkLib()
         1)
             test -n "$libVersion" && {
-                tip="Install $libName version $libVersion or higher."
+                hint="Install $libName version $libVersion or higher."
                 true
             } || {
-                tip="Make sure that you have installed the required libraries."
+                hint="Make sure that you have installed the required libraries."
             }
-            error "Could not find library $fileName: $output" "$tip"
+            error "Could not find library $fileName: $output" "$hint"
             ;;
```

Quelltext 6.2: Beispiel Bugfix-Diff für Variablen Refactoring

```
diff a/Apps/CM/src/Base/CMNpsobjProcessor.m b/Apps/CM/src/Base/CMNpsobjProcessor.m
--- a/Apps/CM/src/Base/CMNpsobjProcessor.m
+++ b/Apps/CM/src/Base/CMNpsobjProcessor.m
@@ -168,25 +416,107 @@
  } else {
-   [[[self chain] dynamicLinksProtocol]
-       addObject: [result objectForKey: @"objectId"]];
+   [[[self chain] dynamicLinksProtocol] addObject: [result objectForKey: @"objectId"]];
  }
```

Quelltext 6.3: Beispiel eines vermeintlichen Bug-Commits

```
diff a/Apps/CM/src/Base/CMNpsobjProcessor.m b/Apps/CM/src/Base/CMNpsobjProcessor.m
--- a/Apps/CM/src/Base/CMNpsobjProcessor.m
+++ b/Apps/CM/src/Base/CMNpsobjProcessor.m
@@ -416,7 +416,7 @@
  } else {
-   [[[self chain] dynamicLinksProtocol] addObject: [result objectForKey: @"objectId"]];
+   [[[self chain] dynamicLinksProtocol] addObject: [result objectForKey: @"id"]];
  }
```

Quelltext 6.4: Beispiel für einen Bugfix-Commit

haben keine Auswirkung auf das Laufzeitverhalten der Software. Jedoch findet der SZZ-Algorithmus diesen Commit als vermeintlichen Bug-Commit.

Lösungsansatz
Eine Möglichkeit, diese „einfachen" Refactorings zu erkennen, ist, den abstrakten Syntaxbaum zwischen zwei Commits zu vergleichen. Das Tool ChangeDistiller der Universität Zürich bietet diese Funktion für Java-Programme an ([10]).

Der Einsatz abstrakter Syntaxbäume ist jedoch sehr aufwendig und benötigt für alle zu analysierenden Sprachen ein passendes Tool. Im Falle von Infopark CMS würden Tools, unter anderem für die Programmiersprachen Java, Objective-C, Ruby, Tcl, HTML, CSS und JavaScript, benötigt werden. Das Tool ChangeDistiller kann nur in Kombination mit Evolizer, nicht jedoch mit CVSAnalY verwendet werden. Deshalb sind in dieser Diplomarbeit keine abstrakten Syntaxbäume zum Einsatz gekommen.

Andere Refactorings
In Quelltext 6.5 ist ein Beispiel-Diff aufgeführt, in dem eine Initialisierung des Wertes `OREILLY_LIB_INSTALLED` in die Oberklasse (`Makefiles.2/Makefile.required`) verschoben wird. Im gleichen Commit wird die gleiche Änderung neben der Datei `../cm/htmlgui/Makefile` an 18 `Makefile`-Dateien durchgeführt.

```
diff a/Makefiles.2/Makefile.required b/Makefiles.2/Makefile.required
--- a/Makefiles.2/Makefile.required
+++ b/Makefiles.2/Makefile.required
@@ -29,6 +29,11 @@ VISUAL_EDITOR=VisualEditor-5.2.0-161
 OREILLY_LIB=oreillyServlet-2002-11-05
+OREILLY_LIB_INSTALLED=$(APPMODULES_INSTALLED)/$(OREILLY_LIB)
diff a/.../htmlgui/Makefile b/.../htmlgui/Makefile
--- a/Apps/CMHTMLGUI/Java/com/infopark/cm/htmlgui/Makefile
+++ b/Apps/CMHTMLGUI/Java/com/infopark/cm/htmlgui/Makefile
@@ -27,14 +27,13 @@ CLASSPATH:=$(CLASSPATH):$(TOMCAT_INSTALLED)/javalib

-OREILLY_DIR=$(APPMODULES_INSTALLED)/$(OREILLY_LIB)
-CLASSPATH:=$(CLASSPATH):$(OREILLY_DIR)/javalib
+CLASSPATH:=$(CLASSPATH):$(OREILLY_LIB_INSTALLED)/javalib
```

Quelltext 6.5: Beispiel für Refactoring

Alle diese Änderungen werden vom SZZ-Algorithmus mit analysiert und als ursprünglicher Defekt mit angegeben.

Interface-Änderung

Einige Defekte lassen sich nur durch zusätzliche Funktionalität lösen. So wird z.B. in Quelltext 6.6 die Oberklasse `GuiSubpage` im Konstruktor um die Variable `isMultipart` erweitert. Alle Unterklassen von `GuiSubpage` müssen nun beim Aufruf des Konstruktors von `GuiSubpage` angepasst werden, so auch die Datei `AttributeEditPage.java`. Neben `AttributeEditPage.java` müssen in diesem Commit fünf weitere Unterklassen von `GuiSubpage` angepasst werden. Der Aufruf des Konstruktors der Oberklasse war vor dieser Änderung in den sechs Unterklassen *nicht* fehlerhaft. Der SZZ-Algorithmus identifiziert diese Zeilen dennoch als defektbehaftet.

Unterstützender Code

Es bereitet dem SZZ-Algorithmus Schwierigkeiten, wenn ein Bugfix-Commits sogenannten „unterstützenden Code" enthält. Um einen defektbehafteten Abschnitt zu reparieren, wird oft zusätzlich unterstützender Code geschrieben. Dieser Code wird gegebenenfalls an anderen (nicht defektbehafteten) Stellen verwendet. Wenn z.B. für die Behebung eines Defekts eine Variable aufwendiger initialisiert werden muss, kann z.B. eine Methode dafür entwickelt werden, die an der defekten Stelle, jedoch auch an anderen Stellen eingesetzt wird.

Der SZZ-Algorithmus verarbeitet jedoch jede Zeile, die geändert wurde, und somit auch die Stellen, die vorher korrekt waren und jetzt nur mit einer anderen Methode laufen.

```
diff a/.../browse/GuiSubpage.java b/.../browse/GuiSubpage.java
--- a/Apps/CMHTMLGUI/Java/com/infopark/cm/htmlgui/browse/GuiSubpage.java
+++ b/Apps/CMHTMLGUI/Java/com/infopark/cm/htmlgui/browse/GuiSubpage.java
@@ -80,19 +81,21 @@ public abstract class GuiSubpage extends Subpage
      * @param name The name of the new page.
      * @param useForm When set to <code>true</code> the page will have a form.
+     * @param isMultipart whether the form contains file upload fields
      * @param withOkAndCancel Gives the page an OK and CANCEL button if it has a
      * form.
      * @param formTableColumnCount For layout reasons you must specify the
      * number of columns of the form table here.
      */
-    protected GuiSubpage(String name, boolean useForm, boolean withOkAndCancel,
-            int formTableColumnCount)
+    protected GuiSubpage(String name, boolean useForm, boolean isMultipart,
+            boolean withOkAndCancel, int formTableColumnCount)
     {
         super(name);
diff --git a/.../browse/AttributeEditPage.java b/.../browse/AttributeEditPage.java
--- a/Apps/CMHTMLGUI/Java/com/infopark/cm/htmlgui/browse/AttributeEditPage.java
+++ b/Apps/CMHTMLGUI/Java/com/infopark/cm/htmlgui/browse/AttributeEditPage.java
@@ -28,7 +28,7 @@ public abstract class AttributeEditPage extends EditPage
     protected AttributeEditPage(String name)
     {
-        super(name);
+        super(name, false);
     }
```

Quelltext 6.6: Beispiel-Diff für ein Interface Change

6.4.2 Stichproben

Um die Datenqualität des SZZ-Algorithmus zu beurteilen, werden zwei verschiedene Stichproben untersucht. Pro Stichprobe wird manuell ermittelt, ob es sich erstens um eine Zeile handelt, die einen Defekt beinhaltet und ob zweitens der vom SZZ-Algorithmus gefundene Bug-Commit diesen Defekt zu verantworten hat. Für die Stichproben werden nur Dateien verwendet, die zu einem verlinkten Bugfix-Commit für wichtige Bugreports gehören und bei denen es sich nicht um einen automatisierten Test oder eine externe Bibliothek handelt. Diese Auswahl beinhaltet 4.814 geänderte Dateien in 1.250 Bugfix-Commits.

Erste Stichprobe: Zeilen pro Datei
Für die erste Stichprobe werden die geänderten Dateien nach der Anzahl der geänderten Zeilen *pro Datei* in zwei Gruppen eingeteilt. Abbildung 6.9 zeigt die Verteilung der Anzahl der geänderten Zeilen pro geänderter Datei. Als Aufteilungsgrenze wird der Median von 3 Zeilen pro Datei verwendet:

- Alle Dateien, in denen weniger als vier Zeilen geändert wurden (n=2.561).
- Alle Dateien, in denen mehr als drei Zeilen geändert wurden (n=2.253).

Pro Gruppe werden 25 geänderte Dateien (ca. 1% der Gruppe) zufällig ausgewählt und manuell untersucht.

Zweite Stichprobe: Zeilen pro Commit
Für die zweite Stichprobe werden die geänderten Dateien nach der Anzahl der geänderten Zeilen *pro Commit* in zwei Gruppen aufgeteilt.

Abbildung 6.10 zeigt die Verteilung der Anzahl der geänderten Zeilen pro Commit. Als Aufteilungsgrenze wird der Median von 8 Zeilen pro Commit verwendet:

- Alle Dateien, die zu einem Commit gehören, in dem weniger als neun Zeilen geändert wurden (n=831, Anzahl Commits = 642).
- Alle Dateien, die zu einem Commit gehören, in dem mehr als acht Zeilen geändert wurden (n=3.983, Anzahl Commits = 608).

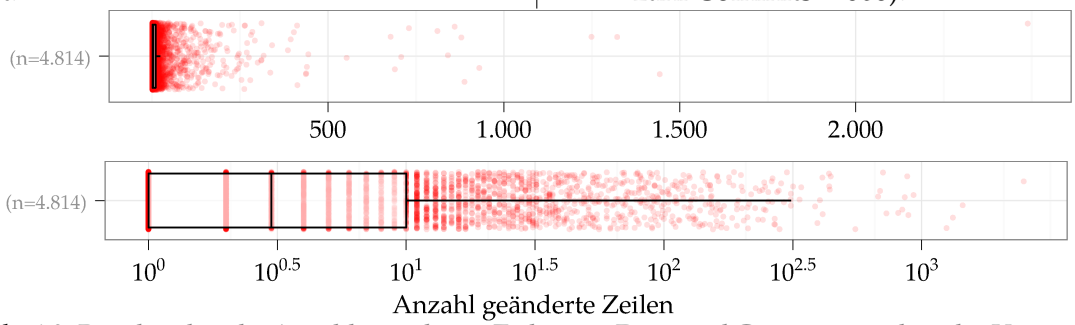

Abb. 6.9: Boxplot über die Anzahl geänderter Zeilen pro Datei und Commit, von dem der Ursprung berechnet wurde.

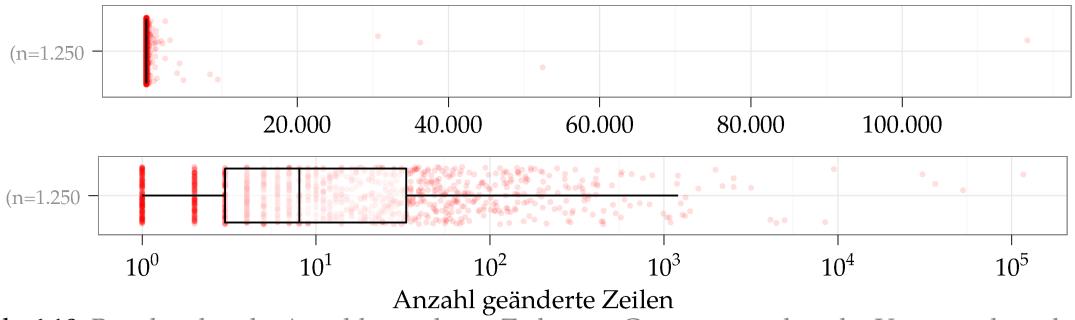

Abb. 6.10: Boxplot über die Anzahl geänderter Zeilen pro Commit, von dem der Ursprung berechnet wurde.

Hier werden ebenfalls pro Gruppe zufällig 25 geänderte Dateien ausgewählt und manuell untersucht. Die zweite Gruppe ist wesentlich größer, da Commits, die viele Zeilen ändern, tendenziell mehr Dateien ändern als Commits, die nur wenige Zeilen ändern.

Ergebnisse
Tabelle 6.3 listet die Ergebnisse der beiden Stichproben auf. In der Stichprobe, in der die Dateien nach ihrer Anzahl eingeteilt wurden, sind lediglich 5 von 50 Bug-Commits (10%) richtige identifiziert. In der Stichprobe, die nach Zeilen pro Commit aufgeteilt wurden, sind 9 von 50 Bug-Commits (18%) richtig identifiziert. In beiden Stichproben ist ein klares Gefälle zwischen der Anzahl der geänderten Zeilen festzustellen. In der ersten Stichprobe sind in der Gruppe mit weniger geänderten Zeilen 4, in der anderen Gruppe lediglich ein richtiger Bug-Commit gefunden worden. In der zweiten Stichprobe sind in der Gruppe mit weniger geänderten Zeilen 8, in der anderen Gruppe ein richtiger Bug-Commit gefunden worden.

Anhand dieser Daten lässt sich folgende These aufstellen: Je mehr Zeilen in einem Bugfix-Commit geändert werden, desto schwieriger ist es, die passenden Bug-Commits zu identifizieren.

In allen Datensätzen, die während der Stichproben untersucht wurden, sind lediglich 14% echte Bugfix-Commits. Hier besteht noch großer Forschungsbedarf.

Da eine Fehlerrate von 86% für seriöse Analyse *nicht* tragbar ist, wird die folgende Einteilung vorgenommen, um die Fehlerrate zu senken: Der SZZ-Algorithmus wird nur auf Bugfix-Commits angewendet, bei denen die Anzahl der geänderten Zeilen unter 4 liegt.

Dieses Kriterium schränkt die Datenmenge von 4.814 auf 441 geänderte Dateien und von 1.250 auf 395 Commits ein. In den untersuchten Stichproben sind dann 9 von 18 echte Bug-Commits (50%). Somit konnte, durch die Einschränkung der Datenbasis, die Fehlerrate auf 50% gesenkt werden. Im Folgenden wird diese Datenbasis als „verkleinerte Datenbasis" bezeichnet.

Abbildung 6.11, 6.12 und 6.13 zeigen den Verlauf bzw. die Verteilung der Bugfix-Commits aus der verkleinerten Datenbasis im Vergleich zur Gesamtmenge aller Bugfix-Commits. Es handelt sich um eine wesentlich kleinere Datenmenge, die im Median ca. 10% aller Bugfix-Commits enthält. Wie Abbildung 6.11 und 6.13 zeigen, ist die verkleinerte Datenbasis meistens pro Quartal vorhanden.

	Zeilen pro Datei		Zeilen pro Commit		Alle
	<=3	>3	<=8	>8	
Richtiger Bug-Commit	4	1	8	1	14
Falscher Bug-Commit	2	0	1	2	5
Keine Defektzeile	19	24	16	22	81
Summe	25	25	25	25	100

Tab. 6.3: Ergebnisse der verschiedenen Stichproben von Hunk Blame

Abb. 6.11: Boxplot über den Anteil der Bugfix-Commits aus der verkleinerten Datenbasis pro Gesamtzahl aller Bugfix-Commit

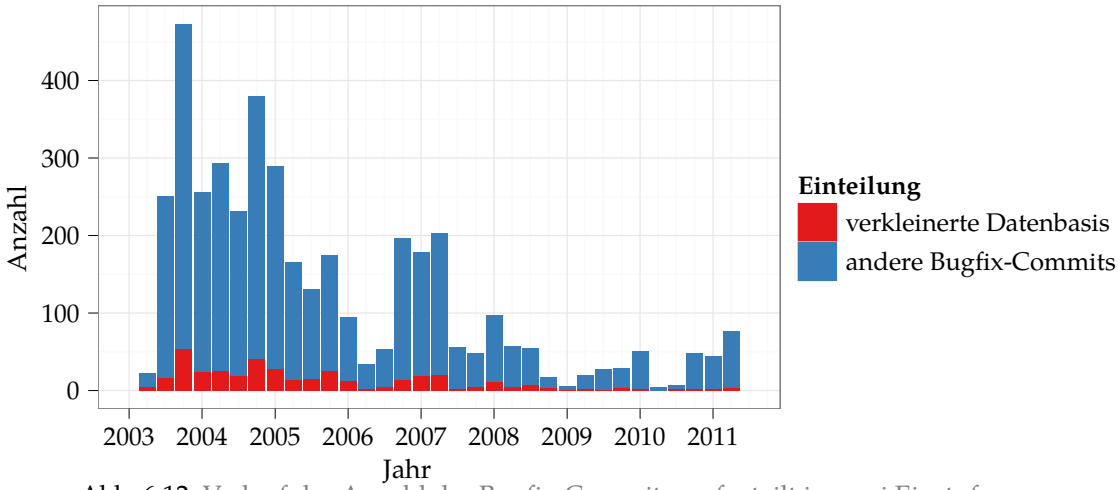

Abb. 6.12: Verlauf der Anzahl der Bugfix-Commits, aufgeteilt in zwei Einstufungen

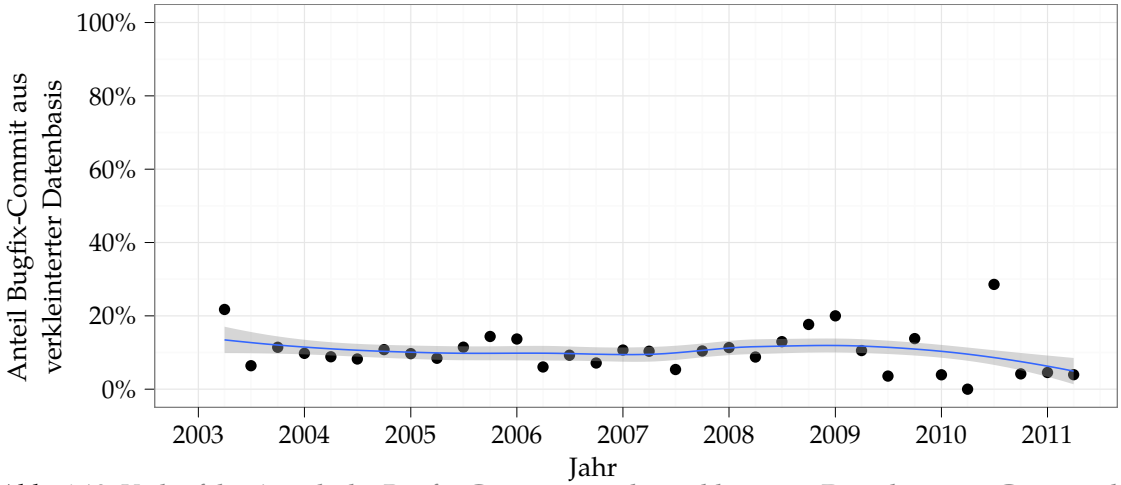

Abb. 6.13: Verlauf des Anteils der Bugfix-Commits aus der verkleinerten Datenbasis pro Gesamtzahl aller Bugfix-Commit

6.4.3 Fazit

Der SZZ-Algorithmus ist kaum in der Lage, den Ursprung von Defekten zu identifizieren, da die meisten geänderten Zeilen nicht den eigentliche Defekt beinhalten. Auch bei Zeilen, die einen Defekt beinhalten, hat der SZZ-Algorithmus Schwierigkeiten, den Ursprung zu identifizieren, da er schon durch minimale Änderungen an einer Zeile (z.B. Einfügen eines Zeilenumbruchs) zu falschen Ergebnissen gelangt.

Die Ergebnisse lassen viel Raum für zukünftige Forschungen:

- Wann handelt es sich bei einer geänderten Zeile um einen wirklichen Defekt?
- Wie können Defekte identifiziert werden, die auf Versäumnisse zurückzuführen sind?
- Wie kann das Ergebnis des SZZ-Algorithmus bei einer defekten Zeile noch verbessert werden?

Kapitel 7

Datenanalyse

In den vorherigen Kapiteln wurde beschrieben, wie Bugfix-Commits, Verlinkungen zu Bugreports und Bug-Commits identifiziert werde können. In diesem Kapitel werden diese Daten analysiert.

In Abschnitt 7.1 wird untersucht, ob Unterschiede zwischen verschiedenen Entwicklern sowohl bezüglich der Anzahl der Commits, wie auch der Bugfix-Commits und der Bug-Commits existieren. In Abschnitt 7.2 wird untersucht, ob an bestimmten Stunden des Tages oder Tagen in der Woche besonders viele Defekte comittet werden. Ob andere Eigenschaften eines Commits (Anzahl der geänderten Dateien oder Zeilen) Auswirkungen auf deren Defektanfälligkeit haben, wird in Abschnitt 7.3 untersucht. Ob ähnliche Zusammenhänge bei Änderungen einer Datei existieren (Anzahl der Änderungen, Anzahl der Entwickler pro Datei) wird in Abschnitt 7.4 inspiziert.

Ferner wird in Abschnitt 7.5 untersucht, ob bestimmte Komponenten absolut und relativ mehr Defekte beinhalten. Zuletzt werden in Abschnitt 7.6 verschiedene Statistikdaten über den Zeitverlauf von Bugreports und Bugfix-Commits sowie Bug-Commits zusammengestellt und mit anderen Projekten verglichen.

7.1 Entwickler

Mit den gesammelten Daten lassen sich mehrere Aussagen über Entwickler treffen. Wer hat die meisten Änderungen durchgeführt? Wer hat die meisten Defekte repariert? Wer hat die meisten Defekte verursacht? Für diese Analysen wurden die Namen der Entwickler pseudonymisiert.

Abbildung 7.1 zeigt den Verlauf verschiedener Maße pro Quartal von Infopark CMS im Zeitraum 2002 bis April 2011. Die erste Grafik zeigt die Anzahl der Commits pro Entwickler. In der zweiten Grafik wird die Anzahl der geänderten Dateien pro Entwickler dargestellt. Die Anzahl der geänderten Zeilen pro Entwickler wird in der dritten Grafik aufgeführt. Ferner sind in jeder der Grafiken die Veröffentlichungszeitpunkte der Infopark CMS Versionen mit angegeben.

Es zeigt sich, dass einige Entwickler über den ganzen Zeitraum aktiv sind (z.B. Ingo und Karl) und andere Entwickler nur in einem bestimmten Abschnitt tätig sind (z.B. Georg und Victor). Dies ist vor allem darauf zurückzuführen, dass in den 9 Jahren unterschiedliche Infopark-Mitarbeiter an dem Produkt gearbeitet haben. Außerdem ist zu beobachten, dass die Anzahl der geänderten Zeilen über den Zeitraum weniger, die Anzahl der geänderten Dateien jedoch mehr wird. In den Quartalen, in denen eine neue Version veröffentlich wurde, wird meistens weniger geändert als in den ein oder zwei Quartalen davor.

Tabelle 7.1 listet die ersten 10 der 42 Entwickler auf, die die meisten Commits erstellt haben. Der Entwickler Karl, der die meisten Commits erstellt hat, verantwortet mehr als 20% der Commits. Die ersten drei Entwickler verantworten zusammen mehr als 50% der Commits.

Entwickler	Anzahl Commits	Anteil an der Summe
Karl	7320	22,49%
Ingo	5482	16,84%
Elton	3744	11,5%
Heinz	2714	8,34%
Bob	2576	7,91%
unbekannt	2096	6,44%
Georg	1884	5,79%
Victor	1136	3,49%
Rob	1107	3,4%
Udo	797	2,45%
andere	3692	11,34%
Summe	32548	100%

Tab. 7.1: Die 10 Entwickler, von denen die meisten Commits sind.

Tabelle 7.2 listet die ersten 10 der 23 Entwickler, die Bugfix-Commits verfasst haben. Der erste Entwickler, Ingo, verantwortet mehr als 20% der Bugfix-Commits.

Entwickler	Anzahl Bugfix-Commits	Anteil an der Summe
Ingo	869	21,35%
Elton	724	17,79%
Karl	595	14,62%
Rob	489	12,01%
Bob	400	9,83%
Heinz	375	9,21%
Quentin	217	5,33%
Will	140	3,44%
Georg	72	1,77%
unbekannt	50	1,23%
andere	139	3,42%
Summe	4070	100%

Tab. 7.2: Die 10 Entwickler, von denen die meisten Bugfix-Commits sind.

Wie bereits bei der Anzahl der Commits sind die ersten drei Entwickler für mindestens 50% der Bugfix-Commits verantwortlich. In den Top 3 sind die gleichen Entwickler wie bei der Anzahl der Commits, nur dass die Reihenfolge unterschiedlich ist. Wie bereits in Unterkapitel 6.4.2 ausgeführt, ist die zugrunde liegende Datenmenge über Bug-Commits sehr ungenau und eingeschränkt. In Stichproben zeigten sich lediglich nur 50% der Bug-Commits als tatsächliche Defekte. Deshalb sollten diese Daten *nicht* überinterpretiert werden. Dennoch listet Tabelle 7.3 die ersten 10 der 17 Entwickler auf, die nach der verkleinerten Datenmenge Bug-Commits verfasst haben.

Entwickler	Anzahl Bug-Commits	Anteil an der Summe
Ingo	116	28,02%
Elton	87	21,01%
Bob	40	9,66%
Karl	39	9,42%
Heinz	36	8,7%
Will	27	6,52%
Rob	23	5,56%
Georg	20	4,83%
Quentin	10	2,42%
Otto	4	0,97%
andere	12	2,9%
Summe	414	100%

Tab. 7.3: Die 10 Entwickler, von denen die meisten Bug-Commits sind. [a]

[a] Bitte die Hinweise aus Unterkapitel 6.4.2 beachten

Trotz allem ist auch hier zu beobachten, dass die ersten drei Entwickler mehr als 50% der Gesamtzahl an Bug-Commits verfasst haben. Neben Ingo und Elton, die auch bei der Anzahl der Commits und der Anzahl der Bugfix-Commits in den Top 3 waren, befindet sich Bob auf Platz drei. Bob ist in den anderen beiden Platzierungen jeweils auf Platz 5 gewesen. Der dritte Platz unterscheidet sich nur um einen Bug-Commit von Platz 4 (Karl), sodass bei der ungenauen Datenlage wenig Aussagekraft besteht.

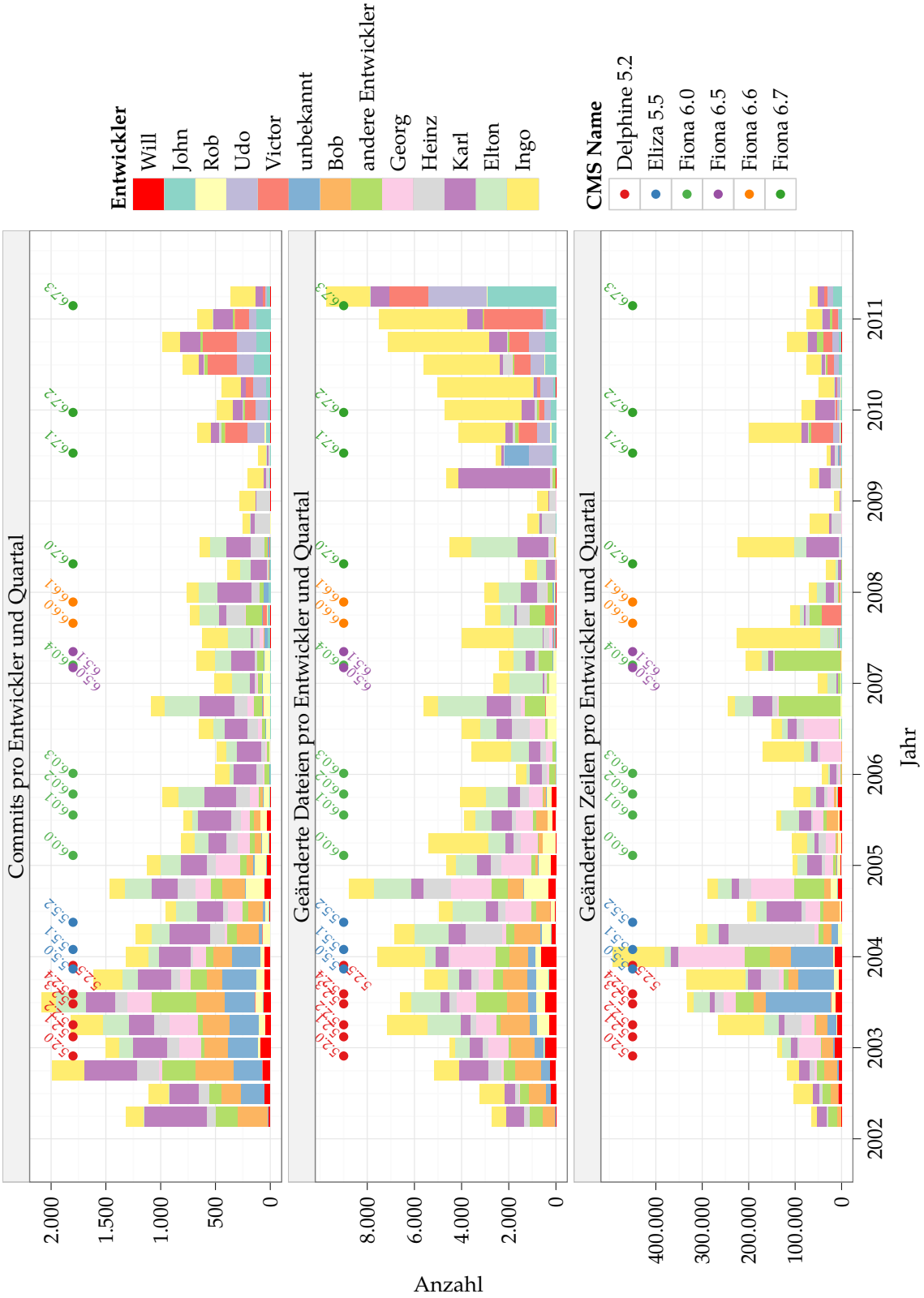

Abb. 7.1: Anzahl der Commits, der geänderten Dateien und geänderter Zeilen pro Entwickler und Quartal. Daten um externe Bibliotheken bereinigt.

Tabelle 7.4 listet den jeweiligen Anteil an Bug-Commits pro Commits des jeweiligen Entwicklers auf. Wenn ein Entwickler 100 Commits verfasst hat und 5 davon Bug-Commits sind, so liegt der Bug-Commit-Anteil bei 5%.

Entwickler	Bug-Commit Anteil
Will	4,16%
Elton	2,32%
Ingo	2,12%
Quentin	2,11%
Rob	2,08%
Bob	1,55%
Heinz	1,33%
Georg	1,06%
Otto	0,88%
Karl	0,53%
andere	0,20%
alle Commits	1,27%

Tab. 7.4: Anteil Bug-Commits der 10 Entwickler, von denen die meisten Bug-Commits sind.

Will führt die Liste mit 27 Bug-Commits von 649 Commits (4,16%) an und liegt damit sowohl deutlich über dem Mittelwert (1,27%) als auch mit Abstand vor Platz 2 und 3. Platz 2 und 3 sind von Elton und Ingo belegt, die bisher bei jeder Aufstellung in den Top 3 waren. Karl, der sonst auch immer in den Top 3 war, hat mit 0,53% eine besonders niedrige Rate. Nach Rücksprache stellte sich heraus, dass Karl lange Zeit für die Qualitätssicherung (Erstellung von automatisierten Tests) zuständig war. Wie in Abschnitt 6.3 beschrieben, wurden automatisierte Tests aus der Menge der potenziellen Bug-Commits herausgenommen. Grund dafür ist, dass kaum unterschieden werden kann, ob die Änderung im Bugfix-Commit durchgeführt wurde, um den defekten Test zu reparieren oder um den Defekt im Test sichtbar zu machen.

7.2 Zeitpunkt

Im Folgenden wird untersucht, ob ein Zeitpunkt (Stunde des Tages und Tag der Woche) mit der Defektanfälligkeit eines Commits korreliert.

Stunde des Tages

> Korreliert die Stunde des Tages mit der Defektanfälligkeit von Commits?

Eyolfson et al. haben diese Frage im Paper „Do Time of Day and Developer Experience Affect Commit Bugginess?" [8] untersucht. Dafür haben sie ebenfalls den SZZ-Algorithmus verwendet, gehen jedoch auf Bedenken hinsichtlich der Datenqualität nicht ein. Unter anderem zeigen die Autoren, dass Commits am häufigsten Bug-Commits sind, wenn sie zwischen Mitternacht und 4 Uhr („Late-Night Commits") erstellt wurden.

Die Hauptarbeitszeit bei Infopark ist zwischen 10 und 18 Uhr. Wie Abbildung 7.2 zeigt, lässt sich das auch an der Anzahl der Commits pro Stunde des Tages ablesen. Auch lässt sich die Mittagspause, die meistens zwischen 12 und 13 Uhr abgehalten wird, in der Anzahl der Commits pro Stunde beobachten. Abbildung 7.2 zeigt ebenfalls, wie hoch zu welcher Stunde des Tages der Anteil an Bugs und Bugfixes ist. „Late-Night Commits", wie sie Eyolfson et al. [8] ausgemacht haben, sind bei Infopark nicht zu beobachten. In der Zeit zwischen Mitternacht und 4 Uhr ist kein einziger Bug-Commit gefunden worden. Das liegt daran, dass zwischen 0 und 7 Uhr bei Infopark nur sehr selten gearbeitet wird. Zwischen 19 und 20 Uhr werden im Verhältnis die meisten Commits eingecheckt, die sich später als defektbehaftet erweisen.

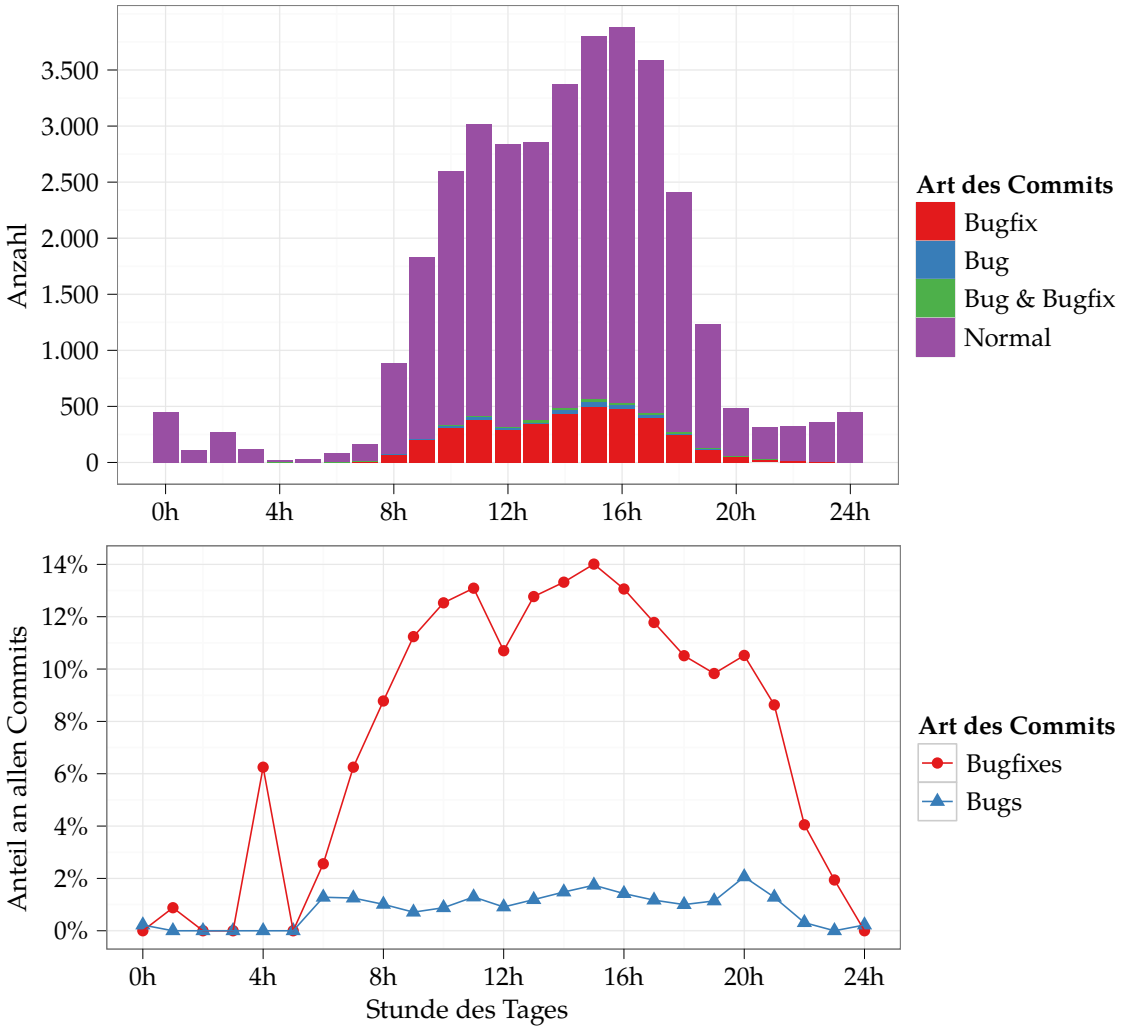

Abb. 7.2: Anzahl der Commits pro Stunde des Tages der letzten 11 Jahre in absoluten und relativen Zahlen. Anm.: Die Datenmenge der Bug-Commits ist sehr klein (siehe Unterkapitel 6.4.2).

Die meisten Bugfix-Commits werden sowohl in absoluten Zahlen (532 Bugfix-Commits) als auch anteilig (14%) zwischen 14 und 15 Uhr eingecheckt. In der gleichen Stunde werden ebenfalls die meisten Bug-Commits in absoluten Zahlen (66 Bug-Commits) eingecheckt.

Tag der Woche

Ferner haben Eyolfson et al. [8] untersucht, ob der Tag der Woche mit der Defektanfälligkeit von Commits korreliert. Die untersuchten Projekte hatten unterschiedliche Wochentage, an denen es am meisten Bug-Commits gab.

Abbildung 7.3 zeigt die Anzahl der Commits pro Wochentag. Anhand der Abbildung wird offensichtlich, dass am Wochenende nur sehr selten gearbeitet wird. Am Dienstag werden die meisten Commits verfasst, bis zum Freitag nimmt die Zahl langsam ab. Dieses Verhalten beobachten Eyolfson et at. [8] auch beim Linux Repository. In ihren Ergebnissen stellen sie ebenfalls fest, dass die Werte der einzelnen Tage, bezüglich der Bug-Commit-Rate, nah beieinander liegen. 2005 hatten Śliwerski et al. [20] beobachtet, dass Freitag der Tag mit den meisten „change inducing fixes" ist. Im Falle von Infopark CMS ist Freitag der Wochentag, an dem absolut wie relativ die

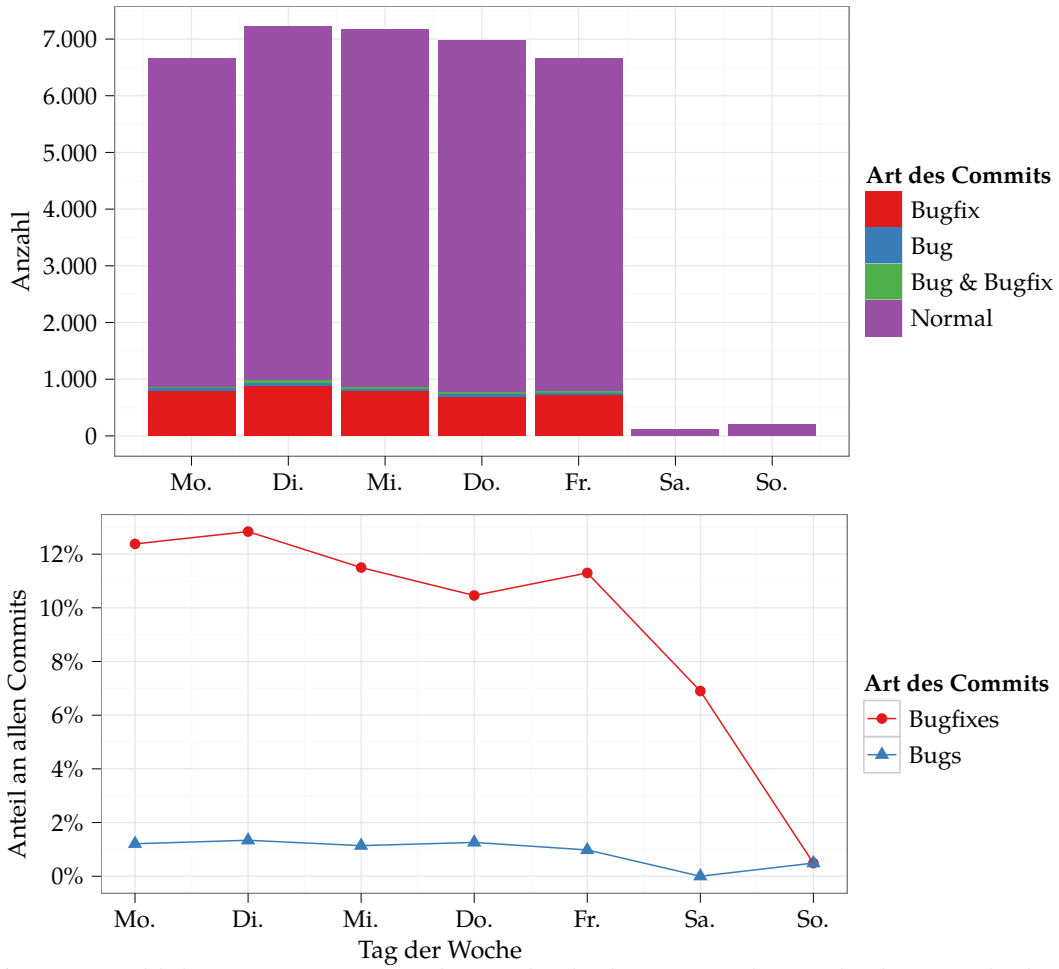

Abb. 7.3: Anzahl der Commits pro Tag der Woche der letzten 11 Jahre in absoluten und relativen Zahlen. Anm.: Die Datenmenge der Bug-Commits ist sehr klein (siehe Unterkapitel 6.4.2).

wenigsten Bug-Commits eingecheckt werden. Der Tag mit den meisten Bug-Commits in absoluten wie relativen Zahlen ist ebenfalls Dienstag. An dem Tag werden ebenfalls die meisten Bugfix-Commits eingecheckt. Somit decken sich die Ergebnisse mit Eyolfson et at. [8], die feststellen, dass es keinen projektübergreifenden Tag der Woche gibt, an dem die meisten Bug-Commits erstellt werden.

7.3 Commit Metriken

Haben Eigenschaften des Commits Auswirkungen auf ihre Defektanfälligkeit?

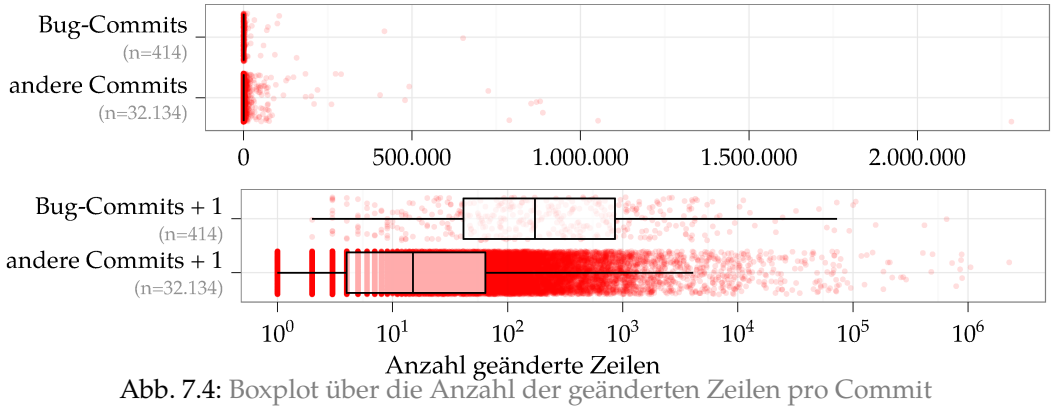

Abb. 7.4: Boxplot über die Anzahl der geänderten Zeilen pro Commit

Geänderte Zeilen pro Commit

Absolute Zahlen
Abbildung 7.4 vergleicht Bug-Commits mit allen anderen Commits bezüglich der Anzahl der geänderten Zeilen. Wie die Abbildung verdeutlicht, werden in Bug-Commits in den verschiedenen Durchschnittswerten (das erste und dritte Quartil, sowie der Median) mehr Zeilen geändert als in den anderen Commits. Selbst der Mittelwert ist in der Bug-Commit-Gruppe (5.346) größer als in der anderen Gruppe (615), obwohl der Maximalwert in der anderen Gruppe mit 2.278.863 wesentlich höher als der Maximalwert 651.502 aus der Bug-Commit-Gruppe ist.

Relative Zahlen
Zusätzlich stellt sich die Frage, wie viele Zeilen prozentual defektbehaftet sind. Um dieser Frage nachzugehen, wird für jeden Commit berechnet, wie viele Zeilen geändert wurden und wie viele Zeilen davon defektbehaftet waren. Auch hier lässt sich die Defekt-Rate berechnen (Anzahl der defektbehafteten Zeilen durch die Anzahl der geänderten Zeilen). Wie bei den Dateien, werden auch hier zwei Gruppen gebildet:

- Commits, die mehr als 15 Zeilen ändern („große Commits", n=15.715)
- Commits, die weniger als 16 Zeilen ändern („kleine Commits", n=15.222)

Im Median wurden 15 Zeilen pro Commit geändert. Wie bereits bei den Dateien pro Commits kann hier beobachtet werden, dass „kleine Commits" eine *größere* Defekt-Rate (0,10%) haben als „große Commits" (0,01%).

Geänderte Dateien pro Commit

absolute Zahlen
Abbildung 7.5 vergleicht Bug-Commits mit allen anderen Commits bezüglich der Anzahl geänderter Dateien. Hier ist ebenfalls, wie bereits bei der Anzahl der Zeilen, zu beobachten, dass die Bug-Commits in den verschiedenen Durchschnittswerten (das erste und dritte Quartil, sowie der Median) wesentlich mehr Dateien ändern als in der Gruppe der anderen Commits.

relative Zahlen
Doch stellt sich die Frage, ob umfangreiche Commits im Datei-Mittelwert mehr oder weniger Defekte enthalten. Um diese Frage zu beantworten wird für jeden Commit berechnet, wie viele Dateien geändert wurden und wie viele von diesen Dateien sich später als defektbehaftet herausstellen. Pro Commit kann dann die Defekt-Rate berechnet werden (Anzahl der defektbehafteten Dateien geteilt durch die Anzahl der geänderten Dateien).

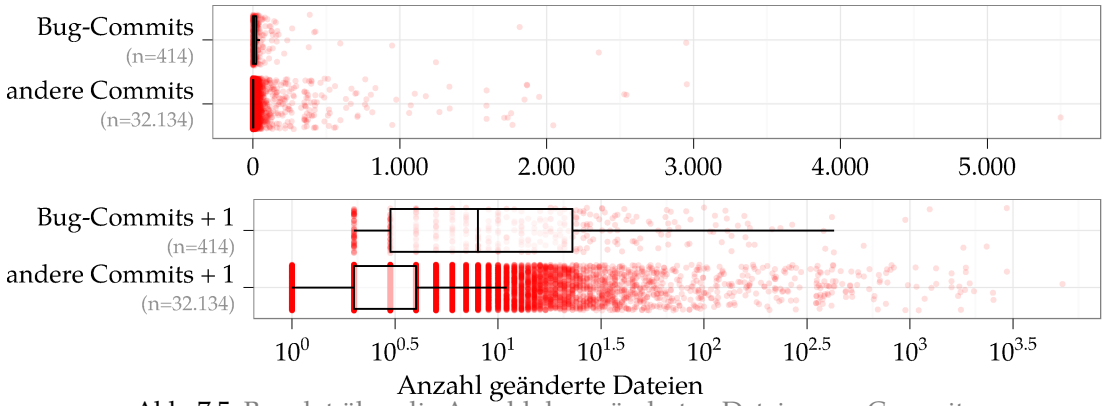

Abb. 7.5: Boxplot über die Anzahl der geänderten Dateien pro Commit

Die Commits werden außerdem in zwei Gruppen eingeteilt:

- Commits, die mehr als 6 Dateien ändern („große Commits", n=27.183)
- Commits, die weniger als 7 Dateien ändern („kleine Commits", n=3.754))

6 Dateien wurden als Grenze verwendet, da es sich bei dem Wert um den Mittelwert der Anzahl der geänderten Dateien pro Commit handelt. Der Median ist in diesem Fall ungeeignet, da dieser bei einer geänderten Datei pro Commit liegt. Mit dieser Gruppierung stellt sich heraus, dass „kleine Commits" mit 0,34% eine *höhere* Defekt-Rate als „größere Commits" (0,17%) aufweisen.

7.4 Datei-Metriken

Haben Verlaufseigenschaften einer Datei Auswirkungen auf ihre Defektanfälligkeit?

Änderungen pro Datei

Ist eine Datei, die häufiger geändert wird defektanfälliger als Dateien, die seltener geändert werden?

Absolute Zahlen
Abbildung 7.6 zeigt, wie oft eine Datei geändert wurde, aufgeteilt in Dateien die mindestens einen Defekt hatten und andere Dateien. Wie in der Abbildung deutlich zu erkennen ist, werden Dateien, die defektbehaftet sind, wesentlich häufiger geändert als Dateien, die keine Defekte enthalten.

Relative Zahlen
Defekte können mit den verwendeten Verfahren nur erkannt werden, wenn sie später repariert werden. Deshalb ist es nicht verwunderlich, dass Dateien defektanfälliger sind, die oft geändert werden. Doch bezieht sich dieser Wert auch auf ein relatives Maß? Um die Frage zu beantworten wird pro Datei ermittelt, wie oft sie geändert und wie oft die Datei als defektbehaftet markiert wurde. Mit der Anzahl der Änderungen und der Anzahl der defektbehafteten Änderungen kann die Defekt-Rate berechnet werden. Dateien, die nie geändert wurden, werden nicht berücksichtigt, da nur durch eine Änderung der Defekt für diese Analysen sichtbar wird. Die Dateien werden in zwei Gruppen eingeteilt:

- Dateien, die mehr als dreimal geändert wurden („oft geänderte Dateien", n=9.720)
- Dateien, die weniger als viermal geändert wurden („andere Dateien", n=10.730)

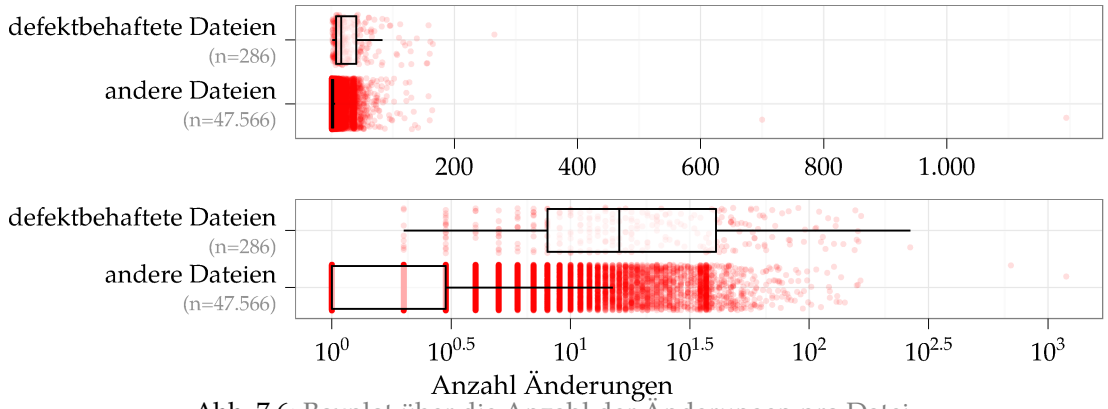

Abb. 7.6: Boxplot über die Anzahl der Änderungen pro Datei

Der Median der Änderungsanzahl einer Datei, die mindestens einmal geändert wurde, liegt bei drei Änderungen. Deshalb wurde diese Grenze als Einteilungsgröße verwendet.

Es lässt sich beobachten, dass die Gruppe der oft geänderten Dateien eine wesentlich höhere Defekt-Rate (0,31%) als die Gruppe der anderen Dateien (0,10%) hat.

Entwickler pro Datei

> Hat die Anzahl der Entwickler Auswirkungen auf die Defekt-Rate einer Datei?

Absolute Zahlen
Abbildung 7.7 zeigt die Anzahl der unterschiedlichen Entwickler pro geänderter Datei. Die Abbildung zeigt, dass defektbehaftete Dateien (Dateien, die mindestens einen Defekt hatten) deutlich häufiger durch eine größere Anzahl an Entwicklern geändert wurden.

Relative Zahlen
Um die Frage zu beantworten, wird für jede Datei ermittelt, wie oft sie geändert wurde, von wie vielen Entwicklern sie geändert wurde und wie oft diese Änderungen defektbehaftet waren. Auch hier werden nur Dateien berücksichtigt, die mindestens einmal geändert wurden. Die Dateien werden in zwei Gruppen aufgeteilt:

- Dateien, die von weniger als vier Entwicklern geändert wurden (n=14.019).
- Dateien, die von mehr als drei Entwicklern geändert wurden (n=6.431).

Der Median der Anzahl der Entwickler pro Datei liegt bei drei Entwicklern.

Wie bereits bei der Anzahl der Änderungen ist zu beobachten, dass die Defekt-Rate in der Gruppe mit vielen unterschiedlichen Entwicklern wesentlich höher ist (0,35%) als in der anderen Gruppe (0,12%).

Korrelation mit Änderungsanzahl
Abbildung 7.8 vergleicht die Anzahl der Entwickler mit der Anzahl der Änderungen einer Datei. Insbesondere mit einer logarithmischen Skala wird deutlich, dass eine Korrelation zwischen diesen beiden Größen existiert. Deshalb verwundert es nicht, dass Metriken, die auf der Anzahl der Änderungen und der Anzahl der Entwickler basieren, sehr ähnliche Ergebnisse liefern.

Zusammenfassung

Insgesamt lässt sich festhalten, dass Dateien, die oft geändert werden, defektanfälliger sind als Dateien, die seltener geändert werden. Zusätzlich konnte festgestellt werden, dass Dateien, die von mehreren Entwicklern geändert werden, defektanfälliger sind als Dateien, die nur von einigen Entwicklern geändert werden. Zwischen Anzahl der Änderungen und Anzahl der Entwickler besteht eine Korrelation. Deshalb müssen weitere Forschungen zeigen, welches der beiden Maße besser geeignet ist, um defektbehaftete Commits zu identifizieren oder vorherzusagen.

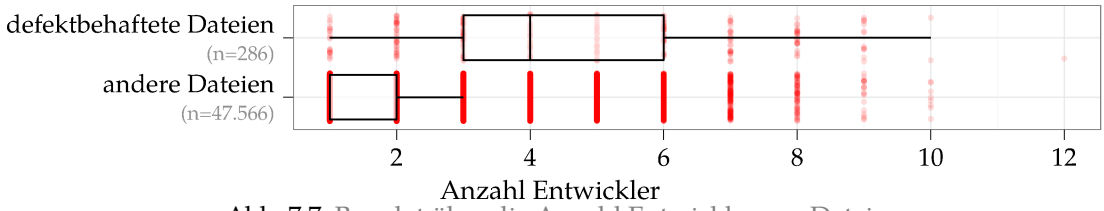

Abb. 7.7: Boxplot über die Anzahl Entwickler pro Datei

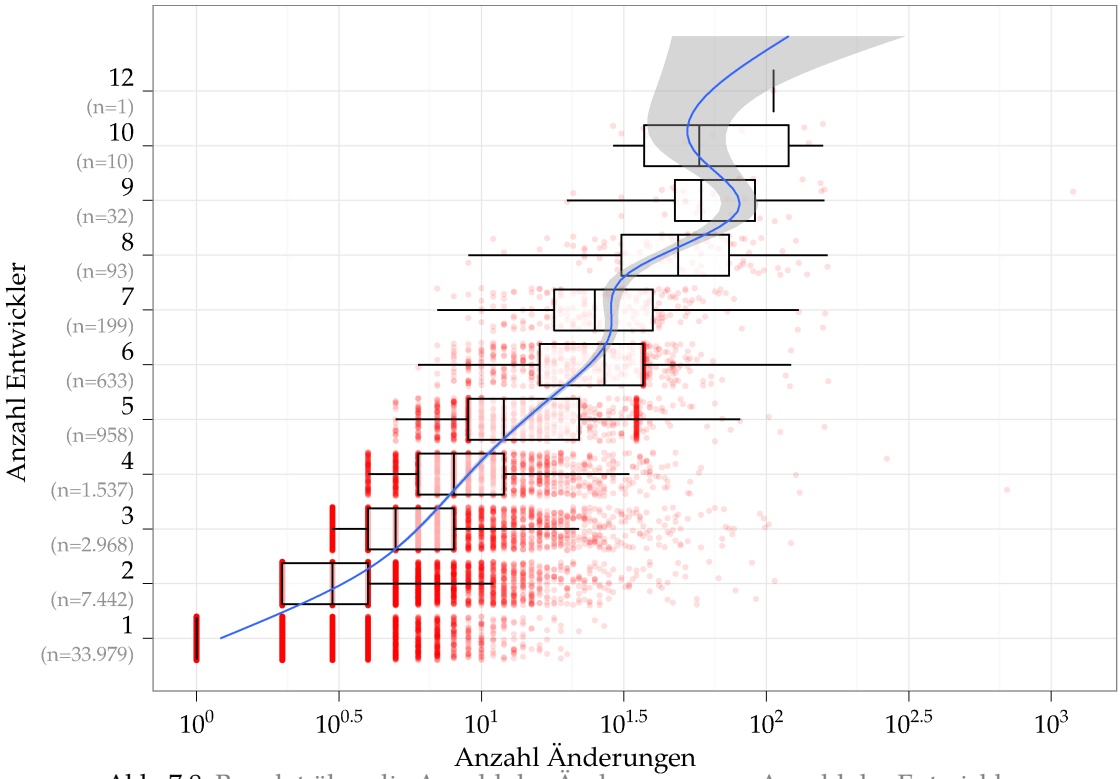

Abb. 7.8: Boxplot über die Anzahl der Änderungen pro Anzahl der Entwickler

7.5 Örtliche Zuordnung

Gibt es bestimmte Komponenten, die besonders defektanfällig sind?

Da die Datenmenge der Bug-Commits begrenzt ist, wird in dieser Diplomarbeit ein Verzeichnis mit einer Komponente gleichgesetzt. Abbildung 7.9 zeigt den Verlauf geänderter Dateien und Zeilen pro Hauptverzeichnis. Der Großteil der Änderungen wurde im Verzeichnis `Apps` durchgeführt. Abbildung 7.10 zeigt den Verlauf der Unterverzeichnisse von `Apps`.

Um zu überprüfen, ob bestimmte Komponenten defektanfälliger sind, werden in Tabelle 7.5 die Unterverzeichnisse von `Apps` aufgelistet, die mindestens einen Defekt und mehr als 1.000 Dateiänderungen beinhalten. Auffällig ist, dass das Verzeichnis `Apps/CM` wesentlich mehr Dateien ändert als die anderen Verzeichnisse, jedoch im Vergleich zu `Apps/CMHTMLGUI` wesentlich weniger Defekte beinhaltet (sowohl absolut als auch relativ). Das Verzeichnis `Apps/CMHTMLGUI` beinhaltet die graphische Benutzeroberfläche des Redaktionssystem. Beim Verzeichnis `Apps/CM` handelt es sich um das Backend von Infopark CMS. Durch Rücksprache mit den Entwicklern stellte sich heraus, dass die graphische Benutzeroberfläche um einiges komplexer ist als das Backend. Deshalb wurde tendenziell ein Bugreport für einen Defekt erstellt, der von einem Entwickler in der graphischen Benutzeroberfläche entdeckt wurde. Die im Backend gefundenen Defekte wurden jedoch meistens direkt repariert, ohne einen Bugreport zu erstellen. Diese Diplomarbeit identifiziert Defekte nur anhand von gemeldeten Bugreports, weshalb diese Reparaturen nicht gefunden werden. Außerdem werden Defekte in der graphischen Benutzeroberfläche wesentlich besser gefunden, da sie einer großen Menge an Nutzern „ausgeliefert" ist.

Um Komponenten miteinander zu vergleichen, die aus dem gleichen Teil-Programm stammen, werden in Tabelle 7.6 drei

Java-Pakete der graphischen Benutzeroberfläche untersucht. Es ist festzustellen, dass die Pakte `node` und `inspector` sowohl absolut als auch relativ mehr Defekte als das Paket `component` enthalten.

Somit zeigt sich, dass verschiedene (Teil) Komponenten unterschiedlich anfällig für Defekte sind. Inwieweit sich die Verzeichnisse unterscheiden, muss jedoch mit einer besseren Datenbasis analysiert werden.

Verzeichnis	Anzahl Dateien	Anzahl Änderungen	Anzahl Defekte	Defekt-Rate pro Datei	Defekt-Rate pro Änderung
`Apps/CM`	15700	47531	51	0,32%	0,11%
`Apps/CMHTMLGUI`	4996	31093	171	3,42%	0,55%
`Apps/NPS`	4891	14354	17	0,35%	0,12%
`Apps/PM`	3909	13288	5	0,13%	0,04%
`Apps/GUI`	923	6968	1	0,11%	0,01%
`Apps/Packaging`	789	2298	15	1,9%	0,65%
`Apps/TP`	395	2259	12	3,04%	0,53%
`Apps/PE`	108	1486	1	0,93%	0,07%
`Apps/SES`	290	1107	4	1,38%	0,36%
Summe	32001	120384	277	0,87%	0,23%

Tab. 7.5: Auflistung der Anzahl der Defekte und Änderungen pro Datei in Unterverzeichnissen von `Apps`, die mindestens 1.000 geänderte Dateien und einen Defekt beinhalten.

Java-Paket	Anzahl Dateien	Anzahl Änderungen	Anzahl Defekte	Defekt-Rate pro Datei	Defekt-Rate pro Änderung
`component`	179	3531	10	5,59%	0,28%
`node`	73	2193	15	20,55%	0,68%
`inspector`	85	2093	13	15,29%	0,62%
Summe	337	7817	38	11,28%	0,49%

Tab. 7.6: Auflistung der Anzahl der Defekte und Änderungen pro Datei für drei Java-Pakete aus der Komponente `CMHTMLGUI`.

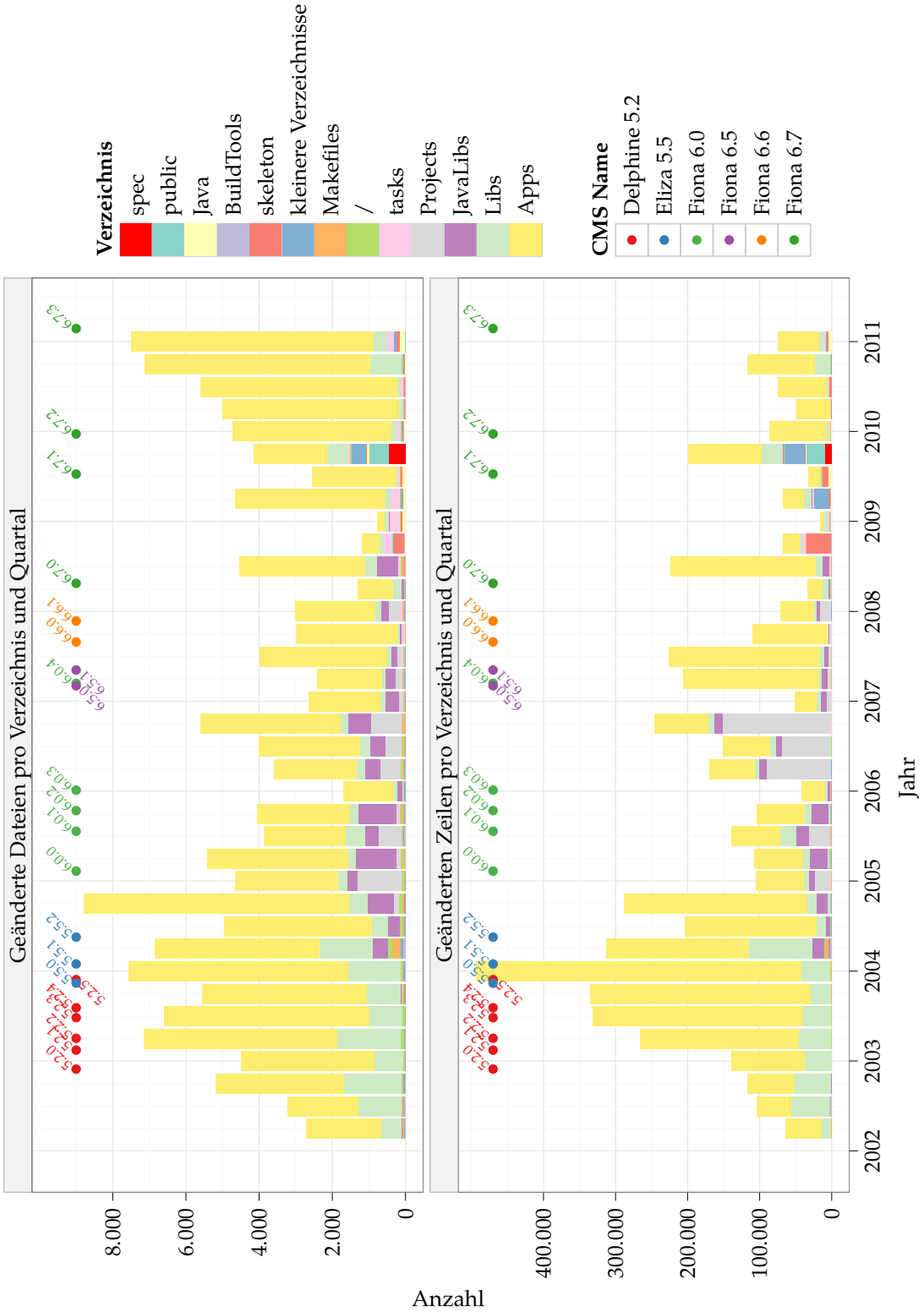

Abb. 7.9: Anzahl der geänderten Dateien und geänderter Zeilen pro Verzeichnis und Quartal. Daten um externe Bibliotheken bereinigt.

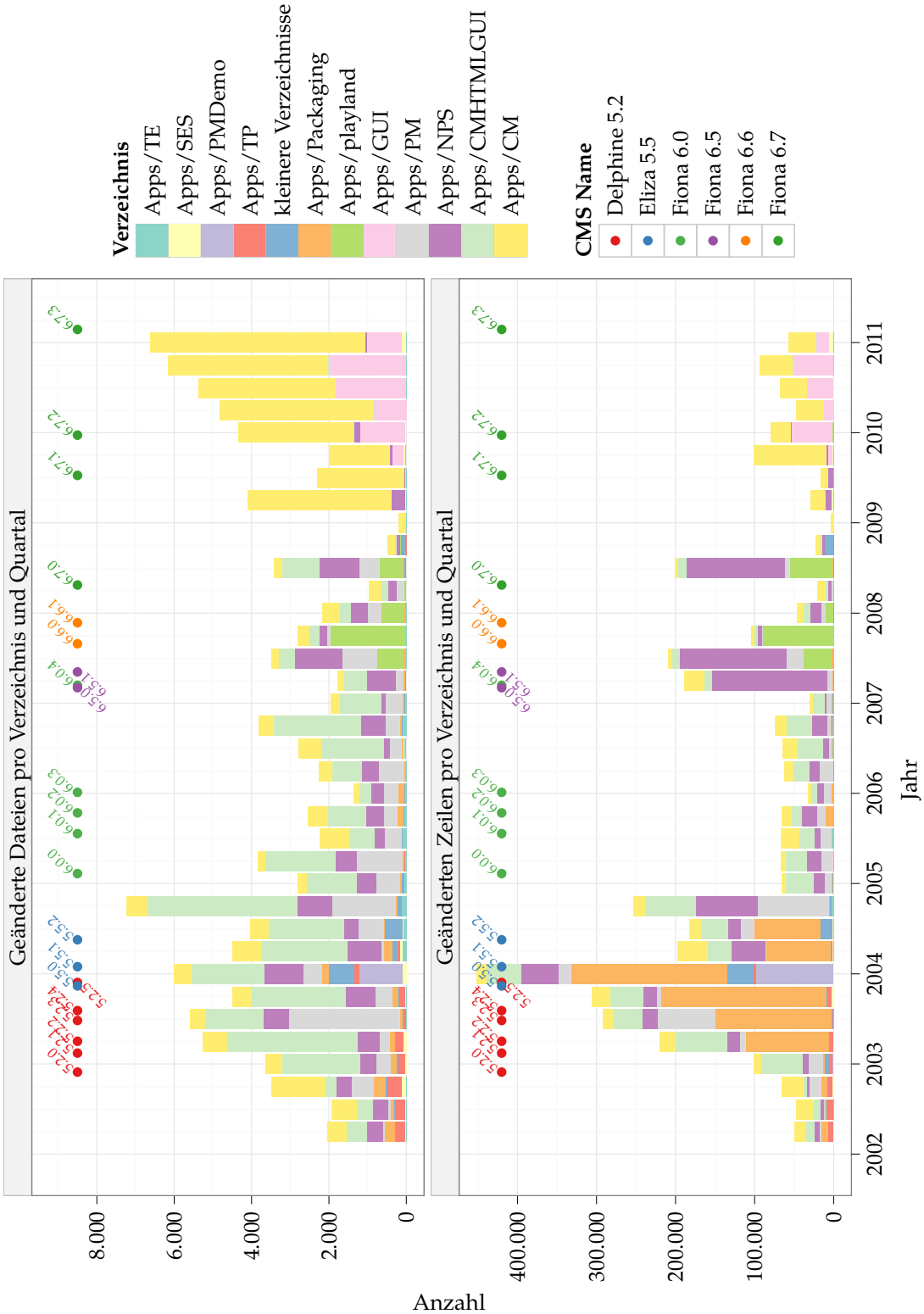

Abb. 7.10: Anzahl der geänderten Dateien und geänderter Zeilen pro `Apps`-Verzeichnis und Quartal. Daten um externe Bibliotheken bereinigt.

7.6 Zeitliche Abstände ...

...zwischen Bugreport und Bugfix-Commits

Abbildung 7.11 zeigt den zeitlichen Abstand in Tagen zwischen der Erstellung eines Bugreports und dem jeweils letzten verlinkten Bugfix-Commit. Die Bugreports sind in die beiden Gruppen „wichtige Bugreports" und „andere Bugreports" eingeteilt (siehe Abschnitt 6.2). In der Gruppe der „wichtigen Bugreports" liegt der Median bei 5 Tagen. In der Gruppe der „anderen Bugreports" liegt der Median bei 15 Tagen.

Pro Bugreport können mehrere verlinkte Bugfix-Commits existieren (siehe Abbildung 6.7 aus Abschnitt 6.4). In Abbildung 7.12 werden die „wichtigen Bugreports" weiter untersucht. Es wird verglichen, wie lange es dauert, bis der erste und der letzte Bugfix-Commit erstellt wurden. Wenn ein Bugreport nur mit einem Bugfix-Commit verlinkt ist, so erscheint er in beiden Gruppen. Die ersten Bugfix-Commits werden im Median nach 3, die letzten nach 5 Tagen erstellt. Der Mittelwert liegt in der Gruppe der ersten Bugfix-Commits bei 40, in der letzten bei 51 Tagen.

Wie die Abbildung 7.13 verdeutlicht, werden jedes Jahr unterschiedliche Mediane erreicht. So dauerte es 2007 im Median nur 4 Tage, bis ein Bugfix-Commit erstellt wurde. 2010 dauerte es 30 Tage. Dies ist damit in Zusammenhang zu bringen, dass 2010 sehr stark an der neuen Version von Infopark CMS gearbeitet wurde. Für das Jahr 2011 liegen nur Daten bis April vor, weshalb sie für diese Abbildung aus der Datenmenge genommen wurden.

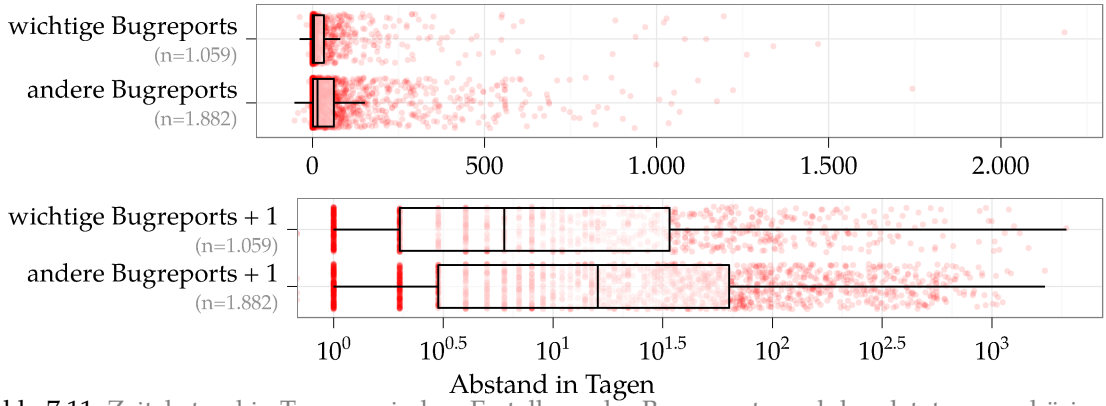

Abb. 7.11: Zeitabstand in Tagen zwischen Erstellung des Bugreports und dem letzten zugehörigen Bugfix-Commit

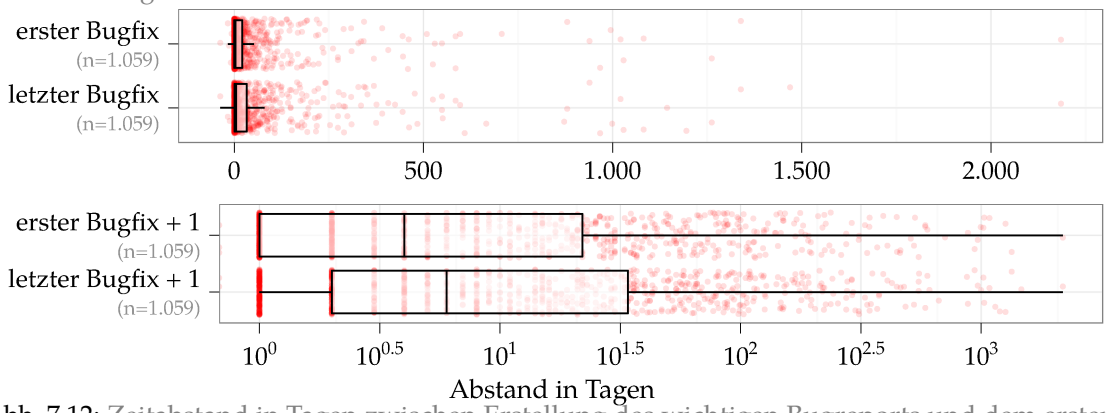

Abb. 7.12: Zeitabstand in Tagen zwischen Erstellung des wichtigen Bugreports und dem ersten, sowie letzten zugehörigen Bugfix-Commit

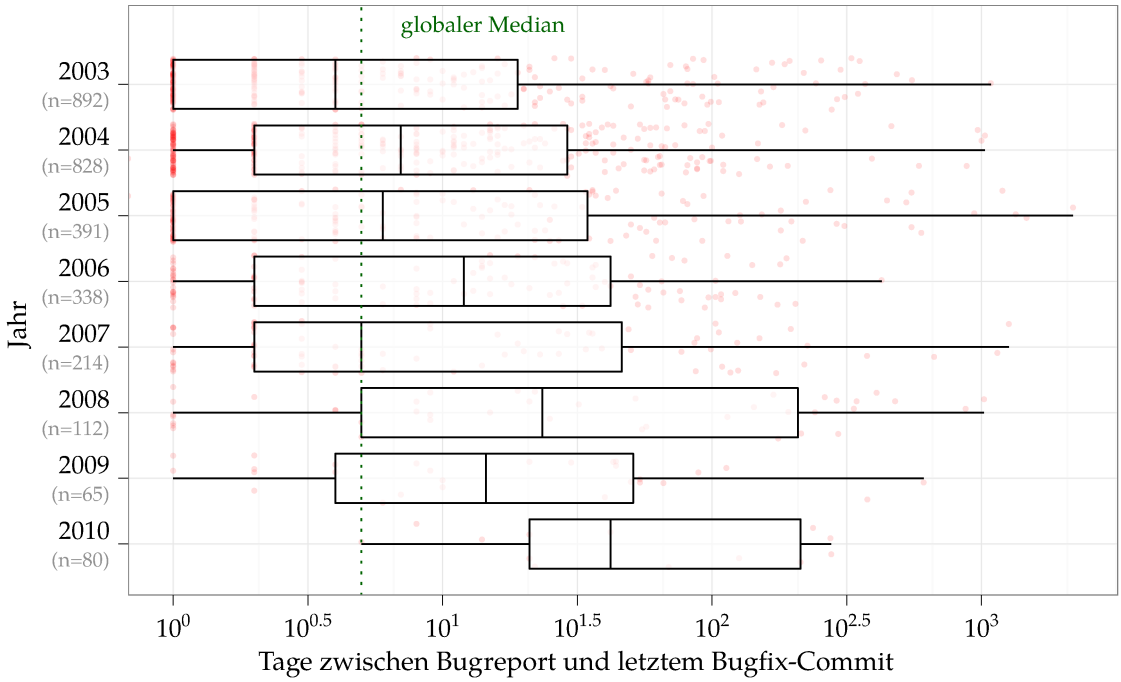

Abb. 7.13: Zeitabstand in Tagen zwischen Erstellung des wichtigen Bugreports und dem letzten zugehörigen Bugfix-Commit pro Jahr. Jeder Wert wurde um 1 erhöht, um den Logarithmus berechnen zu können.

...zwischen Bug- und Bugfix-Commits

Abbildung 7.14 zeigt mit einem Boxplot, wie viele Tage zwischen einem Bug-Commit und einem Bugfix-Commit vergehen. Abbildung 7.15 zeigt den gleichen Boxplot pro Jahr.

Der Median in dieser Gruppe liegt bei 95 Tagen. Der Mittelwert liegt bei 274 Tagen. Wie bereits in Abschnitt 6.4 ausgeführt, ist die Datenmenge der Bug-Commits sehr klein und hat eine Fehlerquote von ca. 50%. Deshalb ist davon auszugehen, dass noch wesentlich mehr Bug-Commits existieren.

Eyolfson et al. [8] haben ebenfalls die „Lebenszeit" eines Defektes für die Projekte Linux-Kernel und PostgreSQL gemessen. Mit ihrem Verfahren messen die Autoren für den Linux-Kernel im Mittelwert 1,38 Jahre. Für PostgreSQL messen sie 3,07 Jahre. Wenn die Messergebnisse von Infopark CMS sich mit einer größeren und valideren Datenmenge bestätigen, könnte dies bedeuten, dass kommerzielle Produkte Defekte wesentlich schneller beheben. Ebenso kann es sein, dass Defekte insbesondere in Linux durch seine weite Verbreitung und vielen Nutzer früher gefunden und gemeldet werden. Eric Raymond formuliert in seinem Buch [16] das Linus-Gesetz:

> „Given enough eyeballs, all bugs are shallow."

Weitere Forschungen müssen zeigen, ob Defekte in kommerziellen Produkten tatsächlich eine kürzere „Lebenszeit" haben als in Open-Source-Projekten.

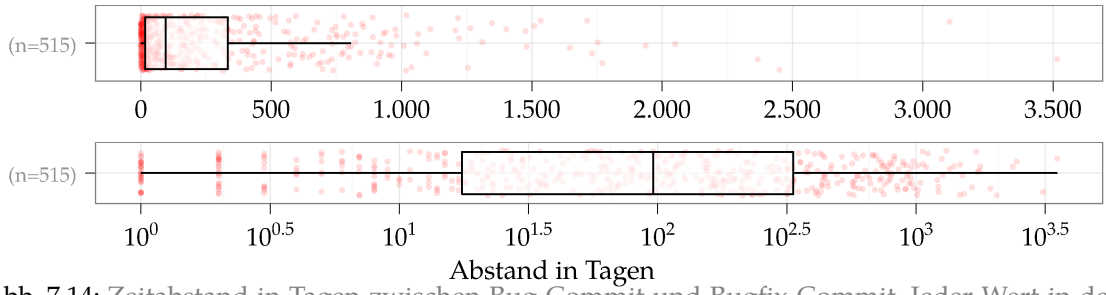

Abb. 7.14: Zeitabstand in Tagen zwischen Bug-Commit und Bugfix-Commit. Jeder Wert in der logarithmischen Darstellung wurde um 1 erhöht, um den Logarithmus berechnen zu können.

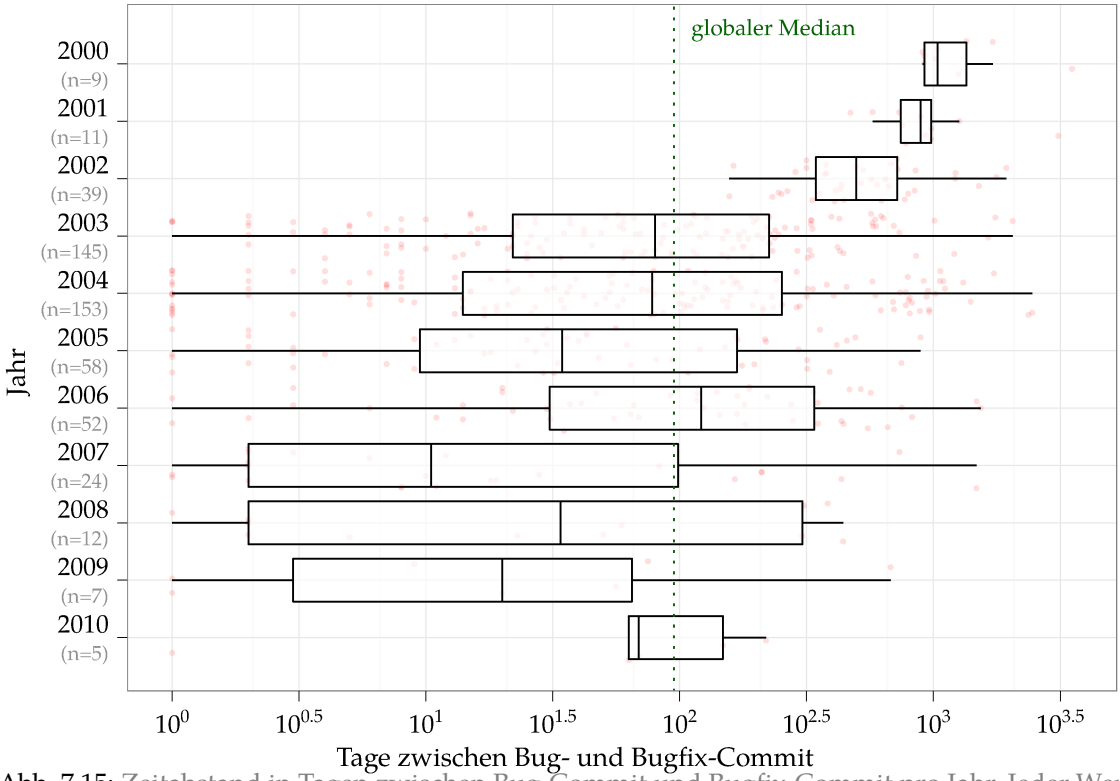

Abb. 7.15: Zeitabstand in Tagen zwischen Bug-Commit und Bugfix-Commit pro Jahr. Jeder Wert wurde um 1 erhöht, um den Logarithmus berechnen zu können.

Kapitel 8

Fazit und Ausblick

8.1 Zusammenfassung

In dieser Diplomarbeit wurden Verfahren untersucht, entwickelt und verbessert, um Daten von Versionsverwaltungssystemen (kurz VCS) und Bugtrackern aufzubereiten und zu analysieren. Ziel war es, Bugreports mit ihren Reparaturen im VCS zu verlinken, sowie deren zugrundeliegende Bug-Commits zu identifizieren.

Für dieses Ziel wurden verschiedene Tools auf ihre Tauglichkeit hin überprüft (siehe Kapitel 2). Zur Verarbeitung des VCS wurde das Tool CVSAnalY in der Version der University of California, Santa Cruz (kurz UCSC) ausgewählt. Neben einer stabilen und schnellen Analyse von Software-Repositorys bietet das Tool eine Implementierung des SZZ-Algorithmus an. Der SZZ-Algorithmus ist einer der wenigen Versuche, den Ursprung eines Defektes automatisiert zu lokalisieren. Bei diesem Verfahren wird angenommen, dass Zeilen, die in einem Bugfix-Commit geändert wurden, vorher defekt waren. Der letzte Entwickler, der die reparierte Zeile zuvor geändert hat, muss somit der Verursacher des Defekts gewesen sein.

Die Verfahren wurden an der kommerziellen Software Infopark CMS der Infopark AG getestet. Infopark CMS wird seit mehr als 14 Jahren entwickelt und bei einer Vielzahl von Kunden produktiv eingesetzt. In den Jahren der Entwicklung von Infopark CMS wurde zwischen drei verschiedenen VCS-Technologien gewechselt (CVS zu SVN und SVN zu Git). Kapitel 3 beschreibt unter anderem, wie die Repositorys aus den verschiedenen Jahren zusammengeführt wurden, sodass über 10 Jahre Historie und 45.000 Commits verfügbar sind.

Diese Arbeit untersucht insbesondere, ob der SZZ-Algorithmus geeignet ist, um Bug-Commits zu identifizieren. Dafür werden zuerst die Commits identifiziert, die Reparaturen (Bugfixes) enthalten. Neben Verfahren aus der aktuellen Forschung werden in Kapitel 4 zusätzlich neue Verfahren vorgestellt, um die Datenqualität der Bugfix-Commits zu steigern. Dafür wird insbesondere der Bugtracker als weitere Quelle zum Identifizieren und Validieren von Bugfix-Commits verwendet. Auch wird die Datenqualität der Bugfix-Links[1] mit anderen Projekten verglichen. Mit den in dieser Arbeit vorgestellten Verfahren ist es möglich, die Bugfix-Link-Rate[2] von 59% auf 67% zu steigern.

Nach Abschluss der Identifizierung der Bugfix-Commits wurde in Kapitel 5 beschrieben, wie die aktuelle Implementierung des SZZ-Algorithmus in CVSAnalY aufgebaut ist. Neben der Anwendung von CVSAnalY wurden in dieser Arbeit Messinstrumente entwickelt, um die Datenqualität besser beurteilen zu können. Ferner wurde die Erweiterung `HunkBlame` um die Funktion erweitert, Quelltext-Kommentare bei der Analyse zu ignorieren. Außerdem wurden mehr als 20 Korrekturen an CVSAnalY vorgenommen, die Defekte oder

[1] die Verlinkung zwischen Bugreport und Bugfix-Commit

[2] Anzahl der verlinken Bugfix-Commits durch die Anzahl der reparierten Bugreports

Inkonsistenzen im Tool beseitigen. Alle Korrekturen sind bei den Entwicklern der UCSC eingereicht worden und die meisten sind bereits in der aktuellen Version von CVSAnalY enthalten. Die zahlreichen Verbesserungen des Tools im Rahmen dieser Arbeit führte zum Erhalt von Commit-Rechten für die Repositorys der UCSC.

Nach der Anwendung der bekannten Verfahren auf das zusammengeführte Repository von Infopark CMS, wurde im Kapitel 6 die Validität der Ergebnisse stichprobenartig überprüft. Dabei stellte sich heraus, dass bereits einige Grundannahmen sowohl dieser Arbeit als auch des SZZ-Algorithmus *nicht* zutreffen.

Zum einen ist die Unterscheidung zwischen Bug und Feature schwierig. Sowohl Bugfix-Commits als auch Bugreports können neue Features bzw. Feature-Requests enthalten, obwohl sie als Bug ausgewiesen sind. Bei den Stichproben stellte sich heraus, dass bereits 25% aller Bugreports eigentlich Feature-Requests sind. Um eine arbeitsfähige Datenbasis zu erhalten, wurden im weiteren nur Bugreports verwendet, die mit dem Schweregrad „Hoch" oder „Kritisch" eingestuft sind.

Zum anderen zeigt der SZZ-Algorithmus gravierende Mängel. Neben konzeptionellen Problemen (z.B. Versäumnisse) gibt es eine Vielzahl an Situationen, in denen er fehlschlägt. Im Kapitel 6 werden verschiedene Ausnahmen genannt und mit konkreten Beispielen belegt. In den Stichproben werden außerdem geänderte Zeilen von Bugfix-Commits daraufhin untersucht, ob sie erstens überhaupt defektbehaftet sind und zweitens mit dem SZZ-Algorithmus der richtige Ursprungscommit entdeckt wird. Nach den Stichproben sind nur 19 der 100 untersuchten Zeilen defektbehaftet. Die anderen 81 Zeilen waren vorher *nicht* defektbehaftet, sondern wurden etwa aufgrund von Refactorings geändert. Von den 19 defektbehafteten Zeilen identifiziert der SZZ-Algorithmus in nur 14 Fällen den richtigen Ursprungscommit. Die Fehlerrate des SZZ-Algorithmus ließ sich drastisch senken, indem nur Bugfix-Commits als Quelle verwendet wurden, die weniger als 4 Zeilen ändern. Mit dieser verkleinerten Datenbasis lag der SZZ-Algorithmus in den Stichproben in 50% richtig.

Zuletzt werden in Kapitel 7 verschiedene Analysen der Daten durchgeführt. Unter anderem wurde untersucht, ob bestimmte Entwickler mehr Defekte verursachen als andere Entwickler und ob bestimmte Uhrzeiten oder Wochentage eine Korrelation mit Defekten aufweisen. Selbst mit der eingeschränkten Datenmenge ließ sich zeigen, dass bestimmte Entwickler mehr Defekte produzieren als andere Entwickler. Doch die Ursachen hierfür können auch darin liegen, dass komplexe Stellen nur von besonders qualifizierten Entwicklern modifiziert werden. Außerdem zeigte sich, dass ein Entwickler lange Zeit in der Qualitätssicherung tätig war und deshalb hauptsächlich mit dem Verfassen von automatisierten Tests beschäftigt war. Aus den Daten ist ebenfalls hervorgegangen, dass zwischen 19 und 20 Uhr die meisten defektbehafteten Commits eingecheckt wurden. Obwohl Dienstag der Tag der Woche ist, an dem die meisten Bug-Commits eingecheckt wurden, war die Bug-Commit-Rate an anderen Wochentagen nicht signifikant niedriger.

Mit der kleinen Datenmenge zeigte sich überraschenderweise, dass Commits, die *wenige* Zeilen ändern, defektanfälliger sind als Commits, die viele Zeilen ändern. Das Gleiche wurde auch bei der Anzahl der Dateien beobachtet. Ob es sich hierbei um Effekte aus der verkleinerten Datenmenge handelt oder um eine substanzielle Erkenntnis, müssen weitere Forschungen zeigen. Außerdem konnte belegt werden, dass Dateien, die viel oder von vielen verschiedenen Entwicklern geändert werden, defektanfälliger sind als Dateien, die seltener

geändert oder von weniger Entwicklern modifiziert werden. Hierbei kann es sich jedoch ebenfalls um Verfahrensfehler handeln, da Defekte mit dem Verfahren nur entdeckt werden können, wenn eine Reparatur vorliegt. Zusätzlich wurde gezeigt, dass eine Korrelation zwischen Anzahl der Entwickler pro Datei und Anzahl der Änderungen existiert. Des Weiteren wurde untersucht, ob es Komponenten gibt, die defektanfälliger sind. Diese Annahme wurde bestätigt.

Darüber hinaus wurden im Kapitel 7 einige statistische Daten über den Entwicklungsverlauf von Infopark CMS zusammengetragen und erörtert.

8.2 Forschungsbedarf

Im Rahmen dieser Diplomarbeit sind verschiedene Bereiche identifiziert worden, die weitere Forschung erfordern.

Die Frage, ob es sich bei einem Änderungswunsch um eine Defekt-Korrektur oder eine neue Funktionalität handelt, ist für die Kategorisierung von Bugreports und Bugfix-Commits wichtig. Dieselbe Frage stellt sich in leicht abgewandelter Form: Handelt es sich bei einer in einem Bugfix-Commit geänderten Zeile um eine vorher defektbehaftete oder um eine intakte Zeile Quellcode? Eine Zeile in einem Bugfix-Commit kann aufgrund von Refactorings oder „Supporting Code" geändert worden sein und war vorher nicht defektbehaftet.

Eine konkrete Möglichkeit, den SZZ-Algorithmus zu verbessern, ist, anstatt Zeilen abstrakte Syntaxbäume miteinander zu vergleichen. Dies würde einige Refactorings unsichtbar machen und verhindern, dass der SZZ-Algorithmus diese als Quelle des Defektes identifiziert.

Zusätzlich stellt sich die konzeptionelle Frage, wie Defekte, die auf Versäumnissen beruhen, einem Ursprung zugeordnet werden können. Wenn z.B. eine bestimmte Bibliothek geladen werden muss, so stellt sich die Frage, seit wann dies der Fall ist. Der SZZ-Algorithmus hat bisher keine Möglichkeit, dies zu identifizieren.

Zu guter Letzt stellt sich die spezielle Frage, wie bei automatisierten Tests entschieden werden kann, ob es sich um einen Defekt oder um eine Erweiterung des Tests handelt, um den gemeldeten Defekt besser sichtbar zu machen.

Anhang A

Abkürzungsverzeichnis

CMS
Content Management System.

CVS
Concurrent Version System.

IQD
Interquartilabstand.

IRC
Internet Relay Chat.

NPS
Network Productivity System.

RCS
Revision Control System.

SVN
Subversion.

SZZ-Algorithmus
Algorithmus von Śliwerski, Zimmermann und Zeller.

UCSC
University of California, Santa Cruz.

VCS
Versionsverwaltungssystem (engl. „Version Control System”).

Anhang B

Literaturverzeichnis

[1] Bachmann, Adrian ; Bernstein, Abraham: Data Retrieval, Processing and Linking for Software Process Data Analysis / University of Zurich, Department of Informatics. Zürich, Schweiz, 2009. – Forschungsbericht

[2] Bachmann, Adrian ; Bernstein, Abraham: Software Process Data Quality and Characteristics: A Historical View on Open and Closed Source Projects. In: *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*. New York, NY, USA : ACM, 2009 (IWPSE-Evol '09). – ISBN 978–1–60558–678–6, S. 119–128

[3] Bevan, Jennifer ; Whitehead, Jr., E. James ; Kim, Sunghun ; Godfrey, Michael: Facilitating Software Evolution Research with Kenyon. In: *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA : ACM, 2005 (ESEC/FSE-13). – ISBN 1–59593–014–0, S. 177–186

[4] Bird, Christian ; Rigby, Peter C. ; Barr, Earl T. ; Hamilton, David J. ; German, Daniel M. ; Devanbu, Prem: The Promises and Perils of Mining Git. In: *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*. Washington, DC, USA : IEEE Computer Society, 2009 (MSR '09). – ISBN 978–1–4244–3493–0, S. 1–10

[5] Boehm, Barry ; Basili, Victor R.: Software Defect Reduction Top 10 List. In: *Computer* Bd. 34. Los Alamitos, CA, USA : IEEE Computer Society Press, 2001. – ISSN 0018–9162, S. 135–137

[6] Chacon, Scott: *Pro Git*. Berkely, CA, USA : Apress, 2009. – ISBN 978–1–43021–833–3

[7] Chacon, Scott: *Replace Kicker*. Version: März 2010. `http://progit.org/2010/03/17/replace.html`, Abruf: 29.06.2011

[8] Eyolfson, Jon ; Tan, Lin ; Lam, Patrick: Do Time of Day and Developer Experience Affect Commit Bugginess? In: *Proceeding of the 8th Working Conference on Mining Software Repositories*. New York, NY, USA : ACM, 2011 (MSR '11). – ISBN 978–1–4503–0574–7, S. 153–162

[9] Fischer, Michael ; Pinzger, Martin ; Gall, Harald: Populating a Release History Database from Version Control and Bug Tracking Systems. In: *Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA : IEEE Computer Society, 2003 (ICSM '03). – ISBN 0–7695–1905–9, S. 23–33

[10] Gall, Harald C. ; Fluri, Beat ; Pinzger, Martin: Change Analysis with Evolizer and ChangeDistiller. In: *IEEE Software* Bd. 26. Los Alamitos, CA, USA : IEEE Computer Society Press, January 2009. – ISSN 0740–7459, S. 26–33

[11] Kim, Sunghun ; Zimmermann, Thomas ; Pan, Kai ; Whitehead, Jr., E. James: Automatic Identification of Bug-Introducing Changes. In: *Proceedings of the 21st IEEE International Conference on Automated Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2006 (ASE'06). – ISBN 0–7695–2579–2, S. 81–90

[12] Kim, Sunghun ; Zimmermann, Thomas ; Whitehead, Jr., E. James ; Zeller, Andreas: Predicting Faults from Cached History. In: *Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2007 (ICSE '07). – ISBN 0–7695–2828–7, S. 489–498

[13] McConnell, Steve: Upstream Decisions, Downstream Costs. In: *Windows Tech Journal*. Eugene, OR, USA : Oakley Publishing Co., November 1997. – ISSN 1061–3501

[14] Mockus, Audris ; Votta, Lawrence G.: Identifying Reasons for Software Changes Using Historic Databases. In: *Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA : IEEE Computer Society, 2000 (ICSM '00). – ISBN 0–7695–0753–0, S. 120–131

[15] Prechelt, Lutz: *Folien zum Kurs „Spezielle Themen der Softwaretechnik": Vorlesung „Errors and Defects"*. Version: Wintersemester 2011/2012. `http://www.inf.fu-berlin.de/inst/ag-se/teaching/V-SWT2-2011/31_defects.pdf`, Abruf: 11.10.2011

[16] Raymond, Eric S.: *The Cathedral and the Bazaar*. Sebastopol, CA, USA : O'Reilly & Associates, Inc., 2001. – ISBN 0–596–01008–8

[17] Robles, Gregorio ; Koch, Stefan ; González-Barahona, Jesús M.: Remote Analysis and Measurement of Libre Software Systems by Means of the CVSAnalY Tool. In: *Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems*. Washington, DC, USA : IEEE Computer Society, 2004 (RAMSS), S. 51–55

[18] Sadowski, Caitlin ; Lewis, Chris ; Lin, Zhongpeng ; Zhu, Xiaoyan ; Whitehead, Jr., E. James: An Empirical Analysis of the FixCache Algorithm. In: *Proceeding of the 8th Working Conference on Mining Software Repositories*. New York, NY, USA : ACM, 2011 (MSR '11), S. 219–222

[19] Śliwerski, Jacek ; Zimmermann, Thomas ; Zeller, Andreas: HATARI: Raising Risk Awareness. In: *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA : ACM, 2005 (ESEC/FSE-13). – ISBN 1–59593–014–0, S. 107–110

[20] Śliwerski, Jacek ; Zimmermann, Thomas ; Zeller, Andreas: When Do Changes Induce Fixes? In: *Proceedings of the 2005 International Workshop on Mining Software Repositories*. New York, NY, USA : ACM, 2005 (MSR '05). – ISBN 1–59593–123–6, S. 1–5

[21] Zimmermann, Thomas ; Weissgerber, Peter: Preprocessing CVS Data for Fine-Grained Analysis. In: *Proceedings of the 1st International Workshop on Mining Software Repositories*, 2004 (MSR '04), S. 2–6