Institut für Softwaretechnologie
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3147

Priorisierung von Testfall-Vorschlägen

Ralf Ebert

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. Jochen Ludewig
Betreuer:	Dipl.-Ing. Rainer Schmidberger
begonnen am:	14. Februar 2011
beendet am:	16. August 2011
CR-Klassifikation:	D.2.5

Inhalt

Der kombinierte Black-Box- und Glass-Box-Test bietet Möglichkeiten, neue Testfälle zu finden, indem er unüberdeckte Codeblöcke aufzeigt. Jeder unüberdeckte Codeblock entspricht einem neuen potenziellen Testfall. Diese Liste von Testfallempfehlungen kann zur Verbesserung einer existierenden Systemtestsuite verwendet werden. Eine Hürde auf dem Weg zur Verbesserung einer existierenden Systemtestsuite ist die notwendige Wirtschaftlichkeit. Die Kosten der wahrscheinlich verhinderten Fehler müssen die Kosten der Testsuiteverbesserung übersteigen.

Um einem Tester die wirtschaftliche Abarbeitung der Testfallempfehlungen zu ermöglichen, sollten diese priorisiert werden. So soll erreicht werden, dass der Tester seine Zeit für die Erstellung von Testfällen mit hoher Wahrscheinlichkeit, einen Fehler zu finden, einsetzt.

In dieser Diplomarbeit wird ein Modell für die Priorisierung von Testfallempfehlungen vorgestellt. Das Modell basiert auf Heuristiken, die die Fehlerfindwahrscheinlichkeit und mögliche Fehlerschwere von Testfallempfehlungen bewerten. Die Heuristiken nehmen ihre Informationen dazu aus verschiedenen Quellen, unter anderem aus dem Quellcode und der Versionsgeschichte des Programms. Alle Heuristiken stammen dabei aus der bestehenden Literatur zu den Themen „risikobasierter Test“ und „Fehlerprognose“. Das Priorisierungsmodell ist allerdings nicht starr auf eine Menge von Heuristiken ausgelegt, sondern kann durch zusätzliche Heuristiken erweitert werden.

Das Modell wurde als CodeCover-Erweiterung implementiert und fügt der Eclipse-Integration von CodeCover eine weitere Sicht hinzu. Mit Hilfe dieser CodeCover-Erweiterung können nach einem Testdurchlauf mit CodeCover Testfallempfehlungen generiert und mit Hilfe mehrerer Heuristiken priorisiert werden. Die Implementierung ist dabei als Framework für die Erprobung weiterer Heuristiken zur Priorisierung von Testfallempfehlungen geeignet.

Die Arbeit schließt mit einer Erprobung der CodeCover-Erweiterung bei einem Industriepartner.

Abstract

The combined black-box and glass-box test offers possibilities of finding new test cases by showing uncovered code blocks. Every uncovered code block corresponds to a potential new test case. This list of test case recommendations can be used to improve an existing suite of system test cases. An obstacle on the way to the improvement of an existing system test suite is the required cost effectiveness. The cost of the probably avoided errors in production must outweigh the cost of test suite improvement.

To facilitate the cost-effective working off the test case recommendations they should be priorized. Thus the tester can focus his time on the creation of test cases with a high probability of finding an error.

In this diploma thesis a model for the priorization of test cases is presented. The model is based on heuristics that rate the error finding probability and the potential covered error severity of a test case recommendation. These heuristics rely on several sources such as the source code and the version archive. The heuristics are extracted from the existing literature about risk-based testing and error prediction. The priorization model is however not limited to a certain set of heuristics but it can be extended to include further heuristics.

The model has been implemented as an extension for the tool CodeCover. It supplements the Eclipse integration with another view. Using this view one can generate a list of test case recommendations after a test execution and priorize them using multiple heuristics. The implementation is fit to be used as a framework for the trial of further heuristics for the priorization of test cases.

This work concludes with an experiment on the performance of the implementation at a company.

Danksagungen

Ich möchte allen danken, die mir bei dieser Diplomarbeit geholfen haben, insbesondere Rainer Schmidberger für die hervorragende Betreuung. Weiterhin möchte ich mich beim Industriepartner – vor allem bei Volker Werner – für die Unterstützung bei der Erprobung bedanken. Abschließend möchte ich mich bei Günter für die investierte Zeit, das Korrekturlesen und die Anmerkungen herzlich bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Aufgabenstellung

Vollständiges Zitat der Aufgabenstellung:

> **Hintergrund**
>
> Das Open-Source Glass-Box-Test-Werkzeug CodeCover unterstützt einen kombinierten Black- Box-/Glass-Box-Testansatz, als dessen Resultat der Tester konkrete Vorschläge für neue Testfälle erhält. Der besondere Vorteil dieser Vorschläge ist der Bezug zu bestehenden Black-Box- Testfällen, die der Tester als Grundlage für die neuen Testfälle nutzen kann. In der aktuellen CodeCover-Version werden diese Testfall-Vorschläge vollständig in eine Tabelle geschrieben, die z.B. mit Excel geöffnet und ausgewertet werden kann. Eine wichtige Funktion bilden dabei Priorisierungen oder Filterungen, da die Anzahl der Vorschläge für Systeme der industriellen Praxis sonst unhandlich groß wird. Derzeit werden hierfür die Sortier- und Filterfunktionen des Tabellensystems genutzt, eine Integration in CodeCover gibt es nicht. Es gibt auch keine systematische Hilfestellung für den Benutzer, also keinerlei Hinweise, in welchen Fällen welche Filter oder Priorisierungen verwendet oder kombiniert werden sollen.
>
> **Aufgabenstellung**
>
> Um eine solche systematische Filterung und Priorisierung der Testfall-Vorschläge und eine Integration dieser Funktionen in CodeCover geht es im ersten Abschnitt dieser Arbeit. Im zweiten Abschnitt soll die CodeCover-Erweiterung an einem Beispiel aus der industriellen Praxis erprobt werden.
>
> Zu Beginn der Arbeit sollen die Priorisierungen des risikobasierten Testens in einer umfangreichen Literaturrecherche untersucht werden. Auch gibt es im Bereich der Testsuite-Reduktion (im Zusammenhang mit Regressionstest) einige interessante Ansätze zur Testfall-Priorisierung. Die so ermittelten Priorisierungen sollen auf ihre prinzipielle Anwendbarkeit und Praxistauglichkeit geprüft werden und als ein wichtiges Zwischenresultat tabellarisch zusammengefasst werden. Auch in dieser Phase können ein oder mehrere Industriepartner im Rahmen von Interviews zur Bewertung der Praxistauglichkeit mit einbezogen werden.
>
> Zur Integration der Priorisierungsfunktionen in CodeCover ist zunächst eine kurze Spezifikation zu erstellen. Die Integration soll anschließend implementiert werden. Notwendig ist eine gute Benutzerführung; der Benutzer soll systematisch zu einer für sein System und seine Testziele geeigneten Priorisierung geführt werden.

In einem zweiten Abschnitt der Arbeit sollen die CodeCover-Erweiterung für ein System der industriellen Praxis angewendet werden. Die Praxistauglichkeit der aus der Literatur ermittelten Priorisierungen sowie die Bedienbarkeit der CodeCover-Erweiterung sollen am konkreten Beispiel überprüft werden. Der konkrete Nutzen für den Tester soll bewertet werden.

Folgende Teilaufgaben sind während der Diplomarbeit durchzuführen:

- Erstellen eines Projektplans mit expliziter Angabe von Meilensteinen [DLLS05]
- Literaturrecherche zum Thema risikobasierter Test und Testsuite-Reduktion
- Einarbeitung in CodeCover und den kombinierten GBT/BBT [Sch10]
- Zusammenstellung der Priorisierungen der Literatur
- Erarbeiten konkreter Lösungsvorschläge zur Umsetzung in CodeCover, Erstellen der Spezifikation
- Zwischenvortrag
- Implementierung der CodeCover-Erweiterung
- Erprobung an einem Beispiel aus der industriellen Praxis, Aufwands- und Nutzenbewertung
- Präsentation der Ergebnisse der Diplomarbeit in einem Endvortrag.
- Berichtserstellung

CodeCover steht unter der OpenSource-Lizenz EPL (Eclipse Public Licence). Die zu implementierende Erweiterung soll ebenso unter EPL gestellt werden.

Vertrauliche Informationen der Praxiserprobung müssen ggf. in einem zum Bericht getrennten Dokument verfasst werden. Die Ergebnisse werden im Bericht dann auf die Zusammenfassung reduziert.

1.2 Motivation

Der Test einer Software ist eine Qualitätssicherungsmaßnahme, die nahezu überall eingesetzt wird, weil ihr enorme Wichtigkeit zugeschrieben wird. Unter Test verstehen wir in dieser Arbeit die – auch mehrfache – Ausführung eines Programms auf einem Rechner mit dem Ziel, Fehler zu finden [LL07].

Der Test besteht dabei aus der sequenziellen Ausführung vorher definierter Testfälle, die aus Vorbedingung, Aktion und Nachbedingung bestehen. Ein Testfall hat dabei eine Wahrscheinlichkeit zwischen null und hundert Prozent einen Fehler zu finden. Testfälle, die sicher einen Fehler anzeigen, und solche, die nie einen Fehler finden werden, brauchen nicht ausgeführt werden. Von den verbleibenden Testfällen sind insbesondere diejenigen interessant, die bei

möglichst wenig Durchführungsaufwand eine möglichst große Wahrscheinlichkeit aufweisen, einen Fehler zu finden.

Die begrenzten Ressourcen an Zeit und Personal müssen möglichst effizient eingesetzt werden, um möglichst viele Eingaben an das Programm abzudecken. Hierzu sind Testfälle notwendig, die durch hohe Fehlerfindwahrscheinlichkeit und geringen Durchführungsaufwand wirtschaftlich sind.

Der kombinierte Black-Box- und Glass-Box-Test bietet Möglichkeiten, neue Testfälle zu finden. Die Testfallempfehlungen des Werkzeugs CodeCover sollen priorisiert werden, um dem Tester Hilfestellung bei der Entwicklung neuer Testfälle zu geben. Die Möglichkeiten des Glass-Box-Tests, die Qualität eines Testfalls in Bezug auf seine Codeüberdeckung zu bewerten werden dabei genutzt. Leider sind absolut objektive Bewertungen von Testfällen so nicht möglich, so dass hier auf Heuristiken zurückgegriffen werden muss.

Diese Arbeit bewegt sich im Bereich der heuristischen Priorisierung von Testfällen. Aus einer großen Liste möglicher Testfälle sollen aufgrund verschiedener Kriterien diejenigen ausgewählt werden, die die höchste Wahrscheinlichkeit haben, einen Fehler zu finden. Dazu wird ein erweiterbares Modell zur Priorisierung von Testfallempfehlungen aufgestellt, implementiert und erprobt. Der Tester hat damit die Möglichkeit, aus den besten Kandidaten tatsächliche Testfälle zu erstellen und so wirtschaftlich die Qualität des Tests und damit die Qualität des Endproduktes zu verbessern.

Diese Arbeit ist im Softwarelebenslauf nach dem Systemtest und der Inbetriebnahme angesiedelt. Die Verbesserung einer Testsuite nach Inbetriebnahme lohnt sich, wenn das Produkt lange in Betrieb sein wird und wenn die Fehlerfolgekosten hoch sind, da die Erweiterung der Testsuite wirtschaftlich bleiben muss. Zudem sind eine dokumentierte Testsuite und ein Testprozess notwendig.

Gliederung der Arbeit

Die Arbeit besteht aus vier Blöcken: Literaturrecherche, Entwurf eines Priorisierungsmodells, Implementierung des Modells und Erprobung des Modells.

Während der Literaturrecherche wurden die Grundlagen des risikobasierten Tests, die vorhandenen Möglichkeiten zur Priorisierung von Testfallempfehlungen zusammengestellt und in Form von Thesen notiert. Auf Basis dieser Thesen wurde ein Modell entworfen, das die Priorisierung der von CodeCover generierten Testfallempfehlungen ermöglicht. Nach einer knappen Beschreibung des Entwurfs und der Implementierung des Modells folgt die Dokumentation der Erprobung des Modells beim Industriepartner.

2 Literaturrecherche

Die folgenden Themenbereiche in der Literatur scheinen für das Thema interessant:

- **Risikobasierter Test:** Der risikobasierte Test zielt auf einen wirtschaftlichen Test ab, der durch Fokussierung auf die größten Risiken erreicht werden soll.
- **Fehlerprognose:** Fehlerprognose ist die Vorhersage der Position und Häufigkeit von Fehlern. Es existieren viele Ansätze und Verfahren, um die Fehlerzahl in einem Programm(teil) vorherzusagen.
- **Testsuite-Reduktion:** Testsuite-Reduktion bezeichnet Verfahren, die darauf abzielen, eine Menge an Testfällen zu reduzieren, ohne deren Potenzial, einen Fehler zu finden, wesentlich einzuschränken.
- **CodeCover:** CodeCover ist das zentrale Werkzeug dieser Arbeit.
- **Sonstiges:** Literatur, die nicht in die anderen Kategorien passt.

Im Folgenden werden 14 Veröffentlichungen aus diesen Gebieten zusammengefasst.

2.1 Risikobasierter Test

Grundlagen

In [Amloo] beschreibt Amland die Grundlagen, Grundbegriffe und Konzepte des risikobasierten Tests. Er zeigt wie diese angewendet werden, gibt eine Vorgehensweise für den risikobasierten Test an und schildert die Erfahrungen, die bei der Erprobung dieses Wissens gemacht wurden.

Der risikobasierte Test ist eine Vorgehensweise, die bei der Auswahl der zu testenden Teile und deren Testtiefe hilft, mit dem Ziel, eine gute Qualität innerhalb möglichst kurzer Zeit zu erreichen.

Die Grundbegriffe sind:

- **Programmeinheit**: Eine Programmeinheit ist ein Teil des Quellcodes eines Programms. Im Kontext der folgenden Definitionen kann eine Programmeinheit eine Methode, eine Klasse, eine Funktionalität, ein GUI-Dialog, ein Modul oder ein ganzes Programm sein.

- **Problem**: Ein Problem ist ein sicher eintretendes Ereignis mit negativer Auswirkung auf das Projektziel. Im Gegensatz zum Risiko ist das Eintreten nicht von Wahrscheinlichkeiten abhängig, sondern sicher.
- **Risiko**: Ein Risiko ist ein mögliches Problem, das eintreten kann, aber nicht muss, aber im Falle seines Eintretens negative Auswirkungen auf das Projektziel hat. Ein Risiko kann durch seine **Eintrittswahrscheinlichkeit** und die im Schadensfall entstehenden **Risikokosten** beschrieben werden. Das Produkt aus Eintrittswahrscheinlichkeit und Risikokosten ergibt den **Risikowert**.
- **Risikoidentifikation**: Die Phase des Tests, in der die Arten der möglichen Risiken zusammengetragen werden. Solche Arten können kaufmännische Risiken, technische Risiken oder indirekte Risiken beispielsweise durch schlechte Usability sein. Zudem wird hier die Risikostrategie festgelegt.
- **Risikoabschätzung**: Bei der Risikoabschätzung werden die Eintrittswahrscheinlichkeit und die Risikokosten für das Versagen jeder einzelnen Programmeinheit abgeschätzt.
- **Risikostrategie**: Die Risikostrategie definiert die Kriterien, nach denen der Test optimiert werden soll. Beispiele solcher Kriterien sind „Keine Klasse soll einen Risikowert von über 1000 Euro haben“ oder „Der Risikowert der Druckfunktion soll unter dem Risikowert der Exportfunktion liegen“. Die Risikostrategie ist damit Interpretationsmittel für das Ergebnis der Risikoabschätzung, das sonst ohne Bedeutung wäre. Sind alle in der Risikostrategie angegebenen Kriterien erfüllt, gilt der Test als abgeschlossen, da alle seine Ziele erreicht sind und das Risiko im Zielbereich liegt.

Das Vorgehen ist dabei ein Zyklus aus den Schritten Risikoidentifikation, Risikoabschätzung und Risikominderung.

Risikominderung beschreibt hier den eigentlichen Test und das Beheben der Fehler, wodurch ein Risiko gefunden und anschließend eliminiert wird. (D.h. die Eintrittswahrscheinlichkeit des Problems soll auf Null gesenkt werden, wodurch es kein Risiko mehr ist.) Die Testfälle werden nach der Risikoabschätzung entsprechend des Risikowertes sortiert und bei der Testdurchführung in dieser Reihenfolge abgearbeitet, vom höchsten Risikowert zum niedrigsten. Dabei wird laufend geprüft, ob die Kriterien der Risikostrategie erfüllt sind. Ist dies sicher der Fall, wird der Test beendet. Zudem müssen laufend Zeit- und Ressourcenaufwand kontrolliert werden, damit der risikobasierte Test nicht zum priorisierten Test verkommt. Mit Hilfe der gesammelten Testaufwandsdaten lassen sich die für den weiteren Test benötigten Ressourcen errechnen und mit den Risiken vergleichen, so dass ein optimales Testende für maximale Wirtschaftlichkeit gewählt werden kann.

Ist nur begrenzt Zeit für die Testdurchführung vorhanden wird automatisch sichergestellt, dass für die größten Risiken die meisten Testressourcen aufgewendet wurden.

Der Artikel beschreibt die Anwendung des risikobasierten Tests auf einer Banksoftware im Umfang von 730.000 SLOC[1]. Bei der Risikobewertung zeigte sich, dass es sinnvoll ist, die

[1]Statement lines of code

Risiken gründlich zu bewerten. So wurden verschiedene Programmeinheiten mit hohem Risikowert identifiziert, die aber teils sehr hohe Risikokosten bei moderater Eintrittswahrscheinlichkeit hatten und teils Programmeinheiten, die trotz geringen Risikokosten wegen der hohen Eintrittswahrscheinlichkeit einen hohen Risikowert hatten. Als Nebenergebnis war allen beteiligten Testern bekannt, auf was sie beim Test der jeweiligen Programmeinheiten besonders Wert legen mussten (Probleme bei der Testdurchführung, zu erwartende Fehlerarten).

Es standen nicht genug Ressourcen für eine detaillierte Risikoabschätzung aller Komponenten zur Verfügung. Daher wurde eine Top-20-Liste der wichtigsten Funktionen aus Kundensicht zusammengestellt. Der weitere Test konzentrierte sich dann auf die Programmeinheiten, aus denen diese Funktionen bestanden.

In dem beschriebenen Testprojekt wurden Metriken für verschiedene Zwecke eingesetzt. Zum einen für die Identifikation von Bereichen mit besonders hohem Risiko, zum anderen für die Messung des Testfortschritts. Dabei wurde gemessen, wie viel Zeit einzelne Testfälle in Anspruch genommen haben und wie viel Zeit der Test einer Programmeinheit benötigt hat. Zudem wurden für die Teststeuerung die gefundenen Fehler aufgezeichnet, um die Effizienz des Tests einschätzen zu können, beispielsweise in gefundenen Fehlern oder gesenkten Gesamtrisikowert pro Mannstunde. Diese Metriken können frühzeitig wesentliche Informationen über den Testfortschritt geben und das Testendekriterium möglicherweise beeinflussen. Es wäre durchaus denkbar, dass ein Team von Testern immer mehr Fehler pro Zeiteinheit findet, denn es kennt das System immer besser.

Um das Risiko der Programmeinheiten übersichtlich darzustellen, schlägt Amland ein Diagramm mit den Achsen Eintrittswahrscheinlichkeit und Risikokosten vor. So können die Betrachter u.a. durch farbliche Unterstützung schnell einen Überblick über die Risikosituation des Systems gewinnen und beispielsweise schnell sehen, ob das System mit vielen leichten, oder mit wenigen schweren Problemen kämpft.

Der Autor warnt davor, den risikobasierten Test als Allheilmittel zu betrachten und gibt zu bedenken, dass der risikobasierte Test nur funktioniert, wenn schon ein funktionierender Testprozess etabliert ist und wenn er von der Unternehmensorganisation unterstützt wird. Ein Tester alleine kann keine Entscheidungen über Risiken für den Kunden treffen.

Amland gibt einen guten Überblick über den risikobasierten Test und setzt seinen Fokus auf die prinzipielle Durchführung und die organisatorischen Rahmenbedingungen und Vorteile unter der Voraussetzung, dass ein Test in vielen Fällen nicht perfekt oder gut, sondern nur „gut genug“ sein muss. Diese Diplomarbeit wird sich in großen Teilen um Metriken für die Risikoabschätzung kümmern.

Anwendung

Der Artikel [Bac99] von James Bach ist in Form eines Erfahrungsberichtes verfasst und fokussiert sich auf die Erfahrungen des Autors im Bereich der Risikoidentifikation. Dabei unterscheidet der Autor zwei Analyseansätze.

- Outside-in
- Inside-out

Der Outside-In-Ansatz verwendet als Basis vordefinierte Listen von möglichen Risiken. Diese enthalten allgemeine Qualitäts- und Risikokriterien und entstehen aus Erfahrung des Testers. Bei diesem Ansatz geht der Tester diese Listen durch und versucht festzustellen, ob diese beim aktuellen Prüfling in Frage kommen. Wo im Produkt nach Risiken gesucht wird, lässt Bach offen. Die Risiken können also mit Hilfe der Spezifikation, aber auch im Quellcode gesucht werden. Dieser Ansatz ist laut Bach allgemein gut und einfach verwendbar, geht allerdings nicht unbedingt auf spezielle Merkmale des Prüflings ein, da nur nach den Risiken in den vordefinierten Listen gesucht wird.

Bei der Inside-out-Vorgehensweise werden die Risiken im fertigen Programm, genauer im Quelltext und im Entwurf, gesucht. Die Vorgehensweise ist also mit Glass-Box-Ansätzen vergleichbar. Der Tester geht das Programm Teil für Teil durch und sucht nach möglicherweise auftretenden Problemen. Dabei stellt er sich (oder einem Entwickler, der das Produkt gut kennt) folgende Fragen:

- Welche Probleme könnten hier auftreten? Welche Schwächen hat diese Komponente?
- Welche Eingaben oder Ereignisse könnten hier einen Fehler auslösen?
- Welche Komponenten würden von einem Fehler hier beeinflusst? Welche Konsequenzen hätte ein Fehler hier?
- Wie wahrscheinlich ist es, dass diese Programmeinheit einen Fehler enthält? [FO00]

Der zweite Teil des Artikels handelt von der Organisation des risikobasierten Tests. Der Autor stellt verschiedene Möglichkeiten vor, Risiken und Komponenten zu verwalten. Im Schlusswort weist der Autor eindringlich darauf hin, dass das Arbeitsmaterial am Beginn des risikobasierten Tests aus Gerüchten besteht. Der risikobasierte Test ist eine Heuristik und liefert damit weder genaue noch vollständige Ergebnisse. Wegen des Risikos, dass ein risikobasierter Test in Frühstadien des Projekts nur eine schlechte Fehlererkennung leistet, empfiehlt der Autor, statt ausschließlichem risikobasiertem Test auch andere Testverfahren und möglichst breit gefächerte Datenquellen zu verwenden.

2.2 Fehlerprognose

Fenton und Ohlsson haben 1997 in [FO00] einige nahe liegende oder verbreitete Hypothesen zur Fehlerverteilung in Programmen empirisch überprüft und sind dabei zu einigen teilweise überraschenden Ergebnissen gelangt. Tabelle 2.1 zeigt die meisten Hypothesen und die Ergebnisse deren empirischer Überprüfung. Die meisten Hypothesen hielten der empirischen Überprüfung nicht stand. Die Überprüfung wurde bei Ericsson Telecom AB auf bestehenden Daten durchgeführt. Es wurden keine besonderen Experimentbedingungen aufgesetzt. Die Codebasis für die Untersuchung bestand aus 140 bzw. 246 Modulen mit je zwischen 1000 und 6000 LOC. Die Fehlerdaten wurden aus vier Testphasen gewonnen: Funktionstest, Systemtest,

Nummer	Hypothese	Belege gefunden?
1a	Eine kleine Zahl an Modulen enthält die meisten Fehler, die vor dem Release gefunden wurden	Ja - 20/60-Regel
1b	Wenn Hypothese 1a stimmt, liegt dies daran, dass diese Module den meisten Code enthalten	Nein
2a	Eine kleine Zahl von Modulen enthält die meisten im Betrieb gefundenen Fehler	Ja - 20/80-Regel
2b	Wenn Hypothese 2a stimmt, liegt dies daran, dass diese Module den meisten Code enthalten	Nein, Belege für's Gegenteil
3	Höhere Fehlerhäufigkeit im Funktionstest bedeutet höhere Fehlerhäufigkeit im Systemtest	Schwacher Beleg
4	Höhere Fehlerhäufigkeit im Test vor dem Release bedeutet höhere Fehlerhäufigkeit im Betrieb	Nein - starke Ablehnung
5a	Kleinere Module sind weniger fehleranfällig als große	Nein
5b	Größenmetriken wie LOC eignen sich zur Vorhersage der Fehlerzahl eines Moduls vor dem Release	Schwacher Beleg
5c	Größenmetriken wie LOC eignen sich zur Vorhersage der Fehlerzahl eines Moduls nach dem Release	Nein
5d	Größenmetriken wie LOC eignen sich zur Vorhersage der Fehlerdichte eines Moduls vor dem Release	Nein
5e	Größenmetriken wie LOC eignen sich zur Vorhersage der Fehlerdichte eines Moduls nach dem Release	Nein
6	Komplexitätsmetriken sind besser zur Vorhersage geeignet, als einfache Größenmetriken	Nein, Schwacher Beleg für SigFF-Metriken [a]
8	In ähnlichen Bedingungen hergestellte Softwaresysteme haben überwiegend ähnliche Fehlerdichten bei ähnlichen Test- und Einsatzphasen	Ja

Tabelle 2.1: Unvollständige Liste der Hypothesen aus [FO00]

[a] [OA96]

Fehler in den ersten 26 Betriebswochen und Fehler in den ersten 52 Betriebswochen. Der Testprozess beim Unternehmen war über viele Jahre etabliert und wohl erprobt.

Bell, Ostrand und Weyuker haben 2006 in [BOW06] ein sprachgesteuertes Telefonsystem untersucht und dabei ein Modell zur Vorhersage der Fehlerdichte in einer Codedatei auf Basis von Codemetriken entwickelt. Das Modell verwendet unter anderem die Heuristiken LOC, Alter, Änderungsstatus und Programmiersprache. Die verwendeten Heuristiken wurden gewichtet, um den tatsächlichen Fehlerdaten zu entsprechen. Der dabei entstehende Vorhersager war dann in der Lage, mit den 20% der als am fehleranfälligsten identifizierten Dateien 75% der tatsächlichen Fehler abzudecken. Allerdings geben die Autoren zu bedenken, dass diese Ergebnisse keinesfalls verallgemeinerbar sind und dass das Vorhersagemodell

für jedes Projekt erst kalibriert werden muss, was viel Zeit in Anspruch nimmt und eine umfangreiche Basis an verknüpfbaren Fehler- und Versionsdaten voraussetzt.

Neuhaus, Zimmermann, Holler und Zeller beschreiben in [NZZ07] die von ihnen durchgeführte Analyse des Mozilla-Browser-Quellcodes [Moz11] auf der Suche nach einem Zusammenhang von Quellcodeeigenschaften und Sicherheitslücken. Weil eine Sicherheitslücke auch „nur“ ein Fehler, wenn auch ein sehr spezieller, ist, können die Ergebnisse dieser Arbeit zu einem gewissen Teil verallgemeinert werden. Im Gegensatz zu anderen Studien ([BOW06], [OWB05], [FO00], [Kim03]), in denen eine Fehlerverteilung von ca. 80% der Fehler in 20% der Module empirisch festgestellt wurde, wurden Sicherheitslücken nur in 4% der Module von Mozilla gefunden. Die Autoren durchsuchten daraufhin diese Module nach Gemeinsamkeiten und stellten fest, dass Module, die bestimmte Headerdateien [Wik10] importieren, zu 90%-100% Sicherheitslücken enthielten. Ähnliche Ergebnisse ergeben sich für Aufrufe bestimmter Funktionen, was allerdings mit den importierten Headerdateien korreliert, da nur vorher importierte Funktionen aufgerufen werden können. Die Autoren haben diese Erkenntnisse verwendet, um einen Vorhersager zu implementieren, der 45% aller verwundbaren Komponenten als solche markiert hat. Von den markierten Komponenten waren 70% tatsächlich verwundbar, d.h. sie enthielten ausnutzbare Sicherheitslücken.

In [NBZ06] beschreiben Nagappan, Ball und Zeller eine empirische Studie, in der in fünf verschiedenartigen Softwareprojekten von Microsoft ein Zusammenhang zwischen Codemetriken und Fehlerdichte von Modulen gesucht und bei manchen Metriken und manchen Projekten gefunden wurde.

Sie verfolgten das Ziel, der Qualitätssicherung eine gezieltere Fehlersuche zu ermöglichen. Dazu werden in dem Artikel zwei verschiedene Ansätze zur Vorhersage der Fehlerdichte von Modulen beschrieben und kombiniert angewendet. Die Ansätze basieren auf den bereits entdeckten Fehlern und dem aktuellen sowie früheren Quellcode der Module, also auf deren Versionsgeschichte.

Beim kombinierten Ansatz wurden die bereits entdeckten Fehler der einzelnen Module aus den vorhandenen Fehlerdaten ermittelt und für diese Module wurden verschiedene Codemetriken[2] erhoben. Anschließend wurden diese Daten einer Korrelationsanalyse unterzogen.

Die Ergebnisse dieser Analyse waren für jedes der fünf Projekte sehr verschieden. Während in zwei der Projekte die Codemetriken mit den gefundenen Fehlern stark korrelierten, waren in den drei verbleibenden Projekten nur wenig Übereinstimmungen zwischen Codekomplexität und gefundener Fehlerzahl vorhanden. Die Autoren konnten keine Gruppe von Metriken finden, die in allen Projekten mit den tatsächlichen Fehlern übereinstimmende Ergebnisse geliefert haben. Allerdings ist ihnen gelungen, mit vorher an einem Projekt kalibrierten Metriken die Fehlerdichte in gleichartigen Projekten akkurat vorherzusagen. Die Autoren begründen die Unterschiede zwischen den Projekten unter anderem mit verschiedenen Entwicklungsprozessen, die teilweise die später verwendeten Metriken schon während der Entwicklung erheben und darauf reagieren.

[2] LOC, Klassen pro Modul, Funktionen pro Modul, globale Variablen pro Modul, Zeilen pro Funktion, Parameter pro Funktion, etc.

Aus den Ergebnissen ihrer Studie folgern die Autoren, dass es keine Menge von Codemetriken gibt, die für jedes beliebige Projekt die zu erwartende Fehlerdichte eines Moduls mit nützlicher Genauigkeit vorhersagen kann. Zwar ist es möglich, für ein bestimmtes Projekt Metriken zu finden und zu kalibrieren, um die Fehlerdichte eines Moduls vorherzusagen, dies ist aber sehr aufwendig und setzt statistisches Expertenwissen voraus.

2010 haben Sliversky, Zimmermann und Zeller in [SZZ05] bei der Untersuchung des Eclipse-Quellcodes, seiner Versions- und Fehlergeschichte festgestellt, dass freitags die meisten fehlereinführenden Commits („fix-inducing change“) gemacht wurden. Es ist allerdings davon auszugehen, dass der Artikel im Wesentlichen zur Demonstration der Datenkombination von Versions- und Fehlergeschichte dient.

Hovemeyer und Pugh haben 2004 in [HP04] einen Weg, Programmfehler auf Codeebene zu finden, vorgestellt. In ihrem Artikel „Finding Bugs is Easy“ beschreiben sie das von ihnen entwickelte Werkzeug FindBugs.

Die Autoren haben einige Methoden zur statischen Analyse auf der Basis sog. „bug patterns“ entwickelt. Ein „bug pattern“ oder Fehlermuster ist dabei eine Codefolge, die wahrscheinlich einen Fehler enthält. Nach Meinung der Autoren sind Codedurchsichten zwar ein sehr effektives Mittel, um Fehler zu finden, allerdings haben diese den Nachteil, zeitintensiv zu sein. Zudem seien menschliche Prüfer anfällig, im Code zu sehen, was der Code tun soll, anstatt was er wirklich tut. Automatisierte Prüfungen haben diesen Nachteil nicht. Ein weiterer Vorteil dieser automatisierten statischen Analyse auf Codeebene ist, dass die Ergebnisse dieser Prüfung nicht von der Qualität und Menge der Testfälle abhängt.

Allerdings sind diese Prüfungen keineswegs vollständig. Die Korrektheit eines Programmes zu beweisen ist in den meisten Fällen kaum möglich. Unvollständige Verifikationen hingegen können zu gewissen Teilen die Aufgaben einer Korrektheitsprüfung ausführen. Sie erzeugen dabei zwar auch Fehldiagnosen, sind aber relativ einfach machbar.

Die implementierten Fehlermustersucher („bug pattern detectors“) verwenden verschiedene Strategien, um Fehler zu finden. Dabei werden die folgenden Aspekte des Quellcodes analysiert. Alle Untersuchungen verwenden den compilierten Bytecode als Basis:

Klassen- und Vererbungsstruktur Hier wird nur die Struktur innerhalb der Klassen, sowie deren Vererbung betrachtet. Die eigentlichen Anweisungen werden ignoriert.

Einfacher Codedurchlauf Analysiert die Methoden der Klassen um einen Zustandsautomaten zu erstellen. Es wird nicht die komplette Kontrollflussinformation genutzt, sondern an einigen Stellen werden Heuristiken eingesetzt.

Kontrollfluss Es wird ein genauer Kontrollflussgraph erstellt.

Datenfluss Der Datenfluss wird analysiert um beispielsweise `NullPointerExceptions` erkennen zu können.

Mit Hilfe dieser Analyseansätze können Fehlermuster gefunden werden, die in die folgenden Kategorien fallen:

- Single-Thread-Korrektheit

- Multithread- / Synchronisierungskorrektheit
- Geschwindigkeitsproblem
- Sicherheitslücken und Verwundbarkeiten

Einige Beispiele für Fehlermuster, nach denen FindBugs sucht:

Cloneable Not Implemented Correctly Eine Javaklasse, die `clone()` überschreibt, muss `super.clone()` aufrufen.

Equal Objects Must Have Equal Hash-codes Eine Javaklasse, die `equals(Object)` überschreibt muss auch `hashCode()` überschreiben. Eine Javaklasse, die `hashCode()` überschreibt, muss auch `equals(Object)` überschreiben.

Uninitialized Read in Constructor Es ergibt üblicherweise keinen Sinn, im Konstruktor den Wert eines Felds zu lesen, das nicht initialisiert wurde.

Null Pointer Dereference Warnt, wenn der Kontrollfluss es ermöglicht, eine Methode oder ein Feld einer Variable aufzurufen, deren Wert `null` ist.

Im Jahr 2004 haben die Autoren die erste Version von FindBugs evaluiert, in dem sie FindBugs den Code von sechs verschiedenartigen Java-Projekten analysieren ließen. FindBugs erreichte dabei eine Korrektheitsquote von erkannten Fehlern je nach Projekt zwischen 54% und 85%. Dementsprechend lag die Quote der fälschlicherweise angezeigten Fehlern (false positive) zwischen 13% und 45%. Dabei geben die Autoren zu bedenken, dass die false-positive-Quote mit der Dauer der Benutzung des Programmes steigt, denn sind alle von FindBugs angezeigten tatsächlichen Fehler behoben, bleiben nur die fälschlicherweise angezeigten übrig.

FindBugs liegt unter anderem als standalone-Variante und als Eclipse-Plugin vor. In der Zwischenzeit wurde FindBugs um viele weitere Fehlermuster erweitert.

Rutar et al. haben 2004 in [RAF04] die fünf automatischen Java-Fehlersuchwerkzeuge FindBugs, JLint, PMD, Bandera und ESC/Java verglichen, in dem sie die Analyseergebnisse von vier quelloffenen, verbreiteten Java-Programmen verglichen haben. Dabei haben sie festgestellt, dass die erprobten Werkzeuge überwiegend disjunkte Warnungen generierten, so dass es grundsätzlich wünschenswert wäre, ihre Ergebnisse mittels eines Meta-Werkzeuges auszuwerten.

Die Werkzeuge verwenden unterschiedliche Methoden zur Analyse des Quellcodes. Syntaxprüfungen werden von FindBugs, JLint und PMD durchgeführt. Der Datenfluss wird von FindBugs und JLint analysiert. Bandera basiert ausschließlich und als einziges Werkzeug auf Modellprüfung, was es für normale, nicht extensiv annotierte, Java-Programme nicht anwendbar macht. ESC/Java verwendet formale Verifikation auf Basis von Bedingungen, die der Programmierer in Form von Annotationen einfügen muss.

Die Dauer der Analyse reicht von einigen Sekunden bei JLint über einige Minuten bei FindBugs und PMD bis zu einigen Stunde bei ESC/Java.

FindBugs sticht unter den erprobten Werkzeugen durch eine niedrige Zahl generierter Warnungen und eine geringe false-positive-Quote heraus, was es im Vergleich zu den anderen Werkzeugen einfach zu bedienen macht. Dazu deckt FindBugs im Vergleich zu seinen Konkurrenten die meisten Fehlerkategorien ab.

Die Autoren schlagen ein Meta-Werkzeug und eine Metrik vor, die die Ergebnisse mehrerer Fehlersuchwerkzeuge kombinieren und werten können. Die Implementierung eines solchen Werkzeuges ist mit dieser Diplomarbeit teilweise erfolgt.

2.3 CodeCover

In [Scho8] werden die Grundlagen des Glass-Box-Tests beschrieben und die Entstehung des Werkzeugs CodeCover, sowie dessen Funktionen, erklärt. Der Glass-Box-Test ist im Gegensatz zum Black-Box-Test nicht von einer Spezifikation abhängig, um Testfälle zu finden. Allerdings können die aus der Spezifikation erstellten Testfälle verwendet werden, um mit Hilfe von CodeCover weitere Testfälle zu finden.

Die Nutzen des Glass-Box-Tests werden wie folgt aufgezählt:

Codeüberdeckung als Metrik zur Testgüte Die während des Tests erhobenen Überdeckungswerte eignen sich als objektive Metrik zur Messung der Testgüte

Testsuite-Erweiterung Mit den ermittelten Überdeckungswerten ist es möglich, aus unüberdeckten Codestellen auf fehlende Testfälle zu schließen.

Testsuite-Reduktion Testfällen mit gleicher oder sehr ähnlicher Überdeckung wird eine geringere Chance auf das Auffinden von Fehlern eingeräumt als Testfällen mit deutlich unterschiedlicher Überdeckung. Diese können damit aus der Testsuite genommen oder geringer priorisiert werden.

Grundlage für selektiven Regressionstest Weiß man, welche Testfälle ein bestimmtes Stück Programmcode abdecken, reicht es, beim Regressionstest diese Testfälle auszuführen. [Scho9]

Für einige dieser Zwecke muss man wissen, welcher Code von einzelnen Testfällen ausgeführt wird. Daher unterstützt CodeCover den testfallselektiven Glass-Box-Test, mit dessen Hilfe die Überdeckung einzelner Testfälle gemessen werden kann.

[Sch10] geht näher auf die Kombination von Black-Box- und Glass-Box-Test ein. Im Artikel wird beschrieben, wie mit CodeCover eine Liste von Testfallempfehlungen unter Anwendung eines solchen kombinierten Ansatzes generiert werden kann.

Dabei kommt neben dem Werkzeug CodeCover das Werkzeug Justus[3] zum Einsatz. Justus ist ein Werkzeug für den Black-Box-Test. Zusätzlich bietet Justus eine Schnittstelle für die

[3]http://justus.tigris.org

Verwendung mit CodeCover, mit der die Übermittlung der Testfallinformationen an CodeCover realisiert wird. Während der Testdurchführung mit Hilfe von Justus wird CodeCover automatisch über Beginn und Ende eines Testfalles informiert, so dass eine testfallselektive Auswertung der Ergebnisse möglich wird.

Der Denominator eines Codeelementes ist ein Codestück, durch das jeder Kontrollfluss in dieses Codeelement zuvor führt. Die Liste von Testfallempfehlungen wird generiert, indem nach Codeblöcken gesucht wird, die selbst nicht ausgeführt werden, deren direkter Denominator aber ausgeführt wird. Diese Codeblöcke werden also tangiert, aber nicht ausgeführt. Das Prädikat eines Codeblocks ist der Ausdruck, der den Kontrollfluss in diesen Codeblock kontrolliert, beispielsweise der Ausdruck `number == 3` einer `If`-Anweisung.

Diese tangierten, aber nicht ausgeführten, Codeblöcke sind für die Herleitung neuer Testfälle von großer Bedeutung, da sie ein Indiz für möglicherweise anders zu wählende Eingabedaten sind, die dann den Codeblock ausführen. Solche Empfehlungen können nutzlos sein, wenn die Erfüllung des Prädikats nicht möglich ist, der Tester das Prädikat nicht versteht oder das Prädikat zur defensiven Programmierung gehört, also nie wahr werden kann.

Der Autor berichtet, dass für Programme aus der industriellen Praxis mit mehreren Tausend Testfallempfehlungen zu rechnen ist. Prinzipiell ist jede Empfehlung, die mit dem oben beschriebenen Verfahren gefunden wurde, geeignet, einen Fehler zu finden. Allerdings ist es weder zumutbar noch wirtschaftlich, eine so lange Liste durchzuarbeiten und aus den einzelnen Einträgen neue Testfälle zu erstellen. Die Empfehlungen müssen daher priorisiert werden. Der Autor schlägt die folgenden Kriterien vor:

Priorität der tangierenden Testfälle Ein tangierender Testfall ist ein Testfall, dessen Kontrollfluss den unüberdeckten Codeblock, der Ziel der Empfehlung ist, tangiert. Die Variante eines wichtigen Testfalls ist vermutlich auch wichtig.

Aufwand zur Ausführung Unter der Annahme, zwei Testfallempfehlungen sind „gleich gut“, würde man die mit dem geringeren Ausführungsaufwand bevorzugen.

Bevorzugung spezifischer Testfälle Je weniger Testfälle einen Codeblock tangieren, desto geeigneter sind daraus entstandene Testfälle. [4]

Gewichtung der Code-Elemente Unwirksame `if-` oder `else`-Blöcke haben im Allgemeinen mehr Gewicht als unvollständige Termüberdeckung oder Schleifenwiederholungen.

Black-Box-Fehlerprognose Expertenwissen und Fehlerstatistik geben Anhaltspunkte, wo bevorzugt nach besseren Empfehlungen zu suchen ist.

Fehlerdichte des Codes In bekannt fehleranfälligen Programmeinheiten sollte eher nach weiteren Fehlern gesucht werden.

Der Autor weist darauf hin, dass die Priorisierung dem Tester lediglich eine methodische Hilfestellung bei der Abarbeitung gibt. Eine Sicherheit, dass eine der gering priorisierten Testfallempfehlungen keinen Fehler aufdecken kann, gibt es nicht.

[4] Dies ist ein Erfahrungswert der bei der Erprobung des Ansatzes zur Testsuiteverbesserung gewonnen wurde.

2.4 Testsuite-Reduktion

Die Techniken der Testsuite-Reduktion nehmen als Eingabe die Codeüberdeckung einzelner Testfälle und bewerten diese dann entsprechend ihrer Redundanz. Dieser Ansatz ist bei der hier vorliegenden Problemstellung nicht anwendbar, da die Testfälle ja erst empfohlen werden und daher noch keine Überdeckung vorhanden ist.

In [RHRH02] werden mehrere Studien zur Testsuite-Reduktion betrachtet, verglichen und es werden deren Ergebnisse kritisch betrachtet. Die Autoren kommen dabei zu dem Schluss, dass Testsuite-Reduktion das Fehlererkennungspotenzial einer Testsuite schwer beeinträchtigen kann. Das Risiko, dass eine Testsuite bei der Anwendung von Reduktionstechniken unerwartet viel an Fehlererkennungspotenzial einbüßt seien zu groß.

Weil das Ziel der Arbeit die Findung guter neuer Testfälle, und nicht die Löschung schlechter, ist, scheinen die Techniken der Testsuite-Reduktion nicht hilfreich zu sein.

2.5 Sonstiges

In [LL07], Kap. 13 sagen die Autoren, dass Reviews ein gutes Mittel zur Steigerung der Qualität der geprüften Dokumente sind.

In [FAI97] wird eine Studie beschrieben, in der gezeigt wurde, dass gestresste Softwareentwickler mehr Fehler machen.

2.6 Schlussfolgerungen aus der Literaturrecherche

Die oben zusammengefassten Artikel sind die Basis für die im Folgenden aufgestellten Thesen, die die Grundlage der weiteren Kapitel der Diplomarbeit darstellen. Diese Thesen sind Aussagen aus der Literatur. Sie sind nicht widerspruchsfrei. Dies ist allerdings für das später vorgestellte Priorisierungsmodell nicht zwingend notwendig, da die Auswahl der tatsächlich verwendeten Thesen dem Benutzer überlassen wird. Mehr weiter unten.

Zur Gruppierung der Thesen mehr in Kapitel 4.

Code

1. **Komplexitätsmetriken 1**: Komplexitätsmetriken, insbesondere Metriken für objektorientierte Software, korrelieren positiv mit der Fehlerdichte. [NBZ06] [HPH+09]
2. **Komplexitätsmetriken 2**: Für jedes Projekt gibt es eine Menge von Komplexitätsmetriken, die mit den gefundenen Fehlern korreliert. [NBZ06]
3. **Dateigröße**: Größere Dateien haben eine höhere Fehlerdichte. [OWB05] [BOW06]

4. **Kopplung**: Kopplungsmetriken sind zur Fehlerprognose geeignet. [HPH+09]
5. **Code smell**: Sog. „Code smell metrics“ sind zur Fehlerprognose geeignet. [HPH+09] [OWB05] [RAF04] [HP04]
6. **Abhängigkeiten**: Importe[5] bestimmter Komponenten korrelieren mit der Fehlerwahrscheinlichkeit. [SZZ06] [NZZ07]

Versionsgeschichte

7. **Änderungshäufigkeit**: Häufiger geänderte Dateien haben eine höhere Fehlerdichte. [OWB05] [BOW06]
8. **Fehleranfälligkeitspriorität**: Die Erhöhung der Überdeckung lohnt besonders in Modulen mit hoher Fehlerdichte. [Sch10]
9. **Dateialter**: Ältere Dateien haben eine geringere Fehlerdichte. [OWB05] [BOW06]
10. **Übertragbarkeit der Fehlerdichte**: Die Anzahl der Fehler im vergangenen Release ist zur Fehlerprognose geeignet. War eine Programmeinheit in der Vergangenheit fehleranfällig, besteht Grund zur Annahme, dass sie weiterhin fehleranfällig ist. [OWB05] [NBZ06] [BOW06]

Test

11. **Prioritätsvererbung**: Die Variante eines wichtigen Testfalls ergibt eher einen wichtigen Testfall, als die Variante eines unwichtigen Testfalls. [Sch10]
12. **Spezifitätspriorität**: Spezifische Testfälle ergeben geeignetere Testfallempfehlungen. [Sch10]
13. **Technische Prädikate**: Technisch formulierte Prädikate eignen sich weniger für Testfallempfehlungen. [Sch10]
14. **Unit-Test**: Durch Unit-Test qualitätsgesicherte Programmeinheiten haben eine höhere Qualität als Programmeinheiten, die keinem Unit-Test unterzogen wurden. [HPH+09]
15. **If-Priorität**: Die Überdeckung eines `If`-Blocks sollte der Term- oder Schleifenwiederholungsüberdeckung vorgezogen werden. [Sch10]

[5] In Java die `import`-Anweisung, C die `include`-Anweisung etc.

Fehler

16. **Pareto-Verteilung**: In ca. 20% der Dateien befinden sich ca. 80% der Fehler. [OWB05]
17. **Schadenspotenzial**: Das Schadenspotenzial der Fehler ist nicht gleichmäßig über den Quellcode verteilt. [HPH+09]
18. **Getestetes Programm**: Die Fehlerverteilung eines Programms nach Systemtest und Produktionsfreigabe ist anders als die des ungetesteten Programms. [FO00] [NBZ06]

Fehlerprognose

19. **Übertragbarkeit der Vorhersagemodelle**: Vorhersagemodelle sind nur dann präsize, wenn sie aus dem gleichen oder einem ähnlichen Projekt gewonnen wurden. Auf andere Projekte übertragen erreichen diese nicht zwingend die gleiche Genauigkeit. [NBZ06]
20. **Verschiedenheit der Projekte**: Es gibt keine Menge von Metriken, die für alle Projekte als Fehlervorhersagemodell passt. [NBZ06]

Qualität

21. **Wesentliche Risiken**: Eine Software kann hinreichend gut sein, wenn die wesentlichen Risiken ausgeschlossen sind. [Aml00]
22. **Überdeckungsrelevanz**: Codeüberdeckung und Testgüte korrelieren. [HFGO94]
23. **Stressfaktoren**: Der Stress des Entwicklers hat Einfluss auf die Programmqualität. [FAI97]
24. **Review**: Durch Reviews können Fehler in den geprüften Dokumenten gefunden werden, was deren Qualität steigert. [LL07] [HPH+09]

3 CodeCover

CodeCover ist ein quelloffenes Glass-Box-Test-Werkzeug, das im Rahmen eines Studienprojektes 2007 an der Universität Stuttgart entwickelt wurde. Es unterstützt den Glass-Box-Test in Java- und Cobol-Programmen durch die Messung von u.a. Anweisungs-, Zweig- und Termüberdeckung. Darüberhinaus werden Threadsynchronisierungsüberdeckungen erfasst und es ist MC/DC-Überdeckungsmessung möglich. CodeCover läuft auf verschiedenen Plattformen, besitzt eine Eclipse-Integration und wurde unter der Eclipse Public Licence (EPL) veröffentlicht [Cod11] [Sch08].

Wie die meisten Glass-Box-Test-Werkzeuge verwendet CodeCover Code-Instrumentierung, um die Überdeckungsmessung zu ermöglichen. Die Instrumentierung kann bei CodeCover entweder über ein Batch-Programm, oder direkt in der Entwicklungsumgebung Eclipse erfolgen, dann allerdings mit eingeschränktem Funktionsumfang. Wird das instrumentierte Programm ausgeführt, hat es zusätzlich eine JMX-Schnittstelle [Ora11].

Eine Besonderheit von CodeCover ist die Fähigkeit, die Überdeckungsmessung während der Programmausführung zu beliebigen Zeitpunkten zurückzusetzen, um so die Überdeckung einzelner Testfälle messen zu können. CodeCover erkennt Beginn und Ende eines Testfalls also nicht durch Programmstart und -ende, sondern der Tester gibt dies explizit an. Die Anwendung muss hierzu nicht neu gestartet werden. So wird die Überdeckung der einzelnen Testfälle einer Systemtestsuite gemessen, ohne das zu testende Programm neu zu starten.

Realisiert wird das über die oben erwähnte JMX-Schnittstelle, die dem Programm bei der Instrumentierung hinzugefügt wird. Diese Schnittstelle bietet zwei Methoden an, mit denen dem instrumentierten Programm während dessen Ausführung Beginn und Ende eines Testfalls mitgeteilt werden kann. Ebenso kann der Download des Ergebnisses der Überdeckungsmessung gestartet werden. Im CodeCover-Eclipse-Plugin steht hierfür eine Sicht zur Verfügung.

Die Eclipse-Integration besteht aus einer Perspektive mit mehreren Sichten, Assistenten, Im- und Exportfunktionalitäten etc.

Abbildung 3.1 zeigt die Überdeckungsmarkierungen von CodeCover zur Überdeckung einer Funktion. Das Prädikat des ersten `If`-Statement ist nie wahr, daher wird der darunterstehende `If`-Block nie ausgeführt, ist also rot markiert. Da der `If`-Block nicht ausgeführt wird, sind nicht alle Zweige der `If`-Anweisung ausgeführt, weswegen das Statement selbst gelb eingefärbt wird. Dieser `If`-Block wird tangiert, aber nicht ausgeführt. Ein nicht ausgeführter Block, dessen Bedingungsstatement ausgewertet wurde, wird als tangierter Block bezeichnet.

Es werden beide Zweige der in der Schleife stehenden `If`-Anweisung ausgeführt. Das `If`-Statement ist daher grün markiert. Die dort im `If`-Block verschachtelte `If`-Anweisung wird

ausgeführt, allerdings nicht vollständig. Der implizite `Else`-Block würde nur ausgeführt werden, wenn `number != 4`, was aber nicht der Fall ist.

Weitere Dokumentation zu CodeCover unter [Cod11].

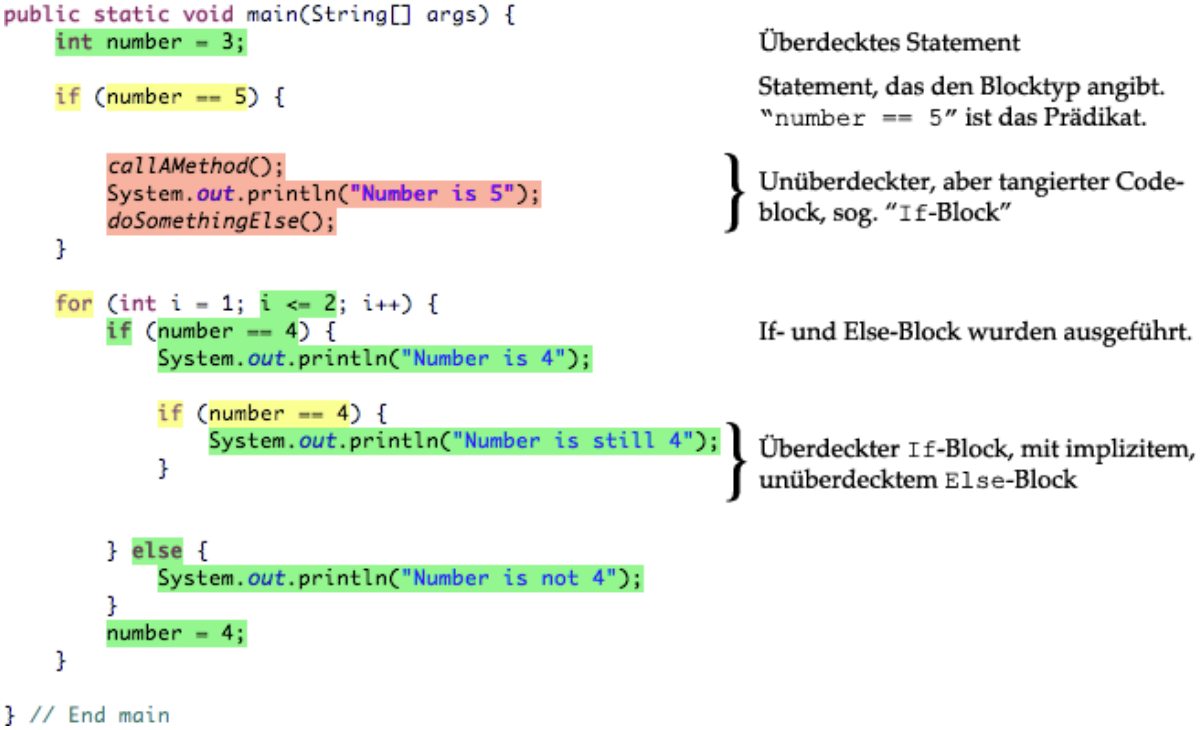

Abbildung 3.1: Beispiel der CodeCover-Ausgabe der Überdeckung einer Funktion

[Sch10] beschreibt ausführlich ein neues Zusatzmodul für CodeCover. Dieses Zusatzmodul erlaubt die Analyse der gesammelten Daten zur Generierung von Testfallempfehlungen. Ergebnis der Analyse ist eine Liste von tangierten Codeblöcken. Jeder dieser Codeblöcke entspricht einer Testfallempfehlung. Diese Liste ist allerdings bei Programmen aus der industriellen Praxis mehrere tausend Einträge lang und unsortiert. Die erheblichen Unterschiede in Umsetzungsaufwand einer Empfehlung in einen Testfall und erwarteter Fehlerfindwahrscheinlichkeit machen eine Priorisierung der Empfehlungen notwendig.

4 Priorisierungsmodell

Teilaufgabe der Diplomarbeit ist die Konzeption und Umsetzung einer Erweiterung für CodeCover für die Anzeige und Priorisierung von Testfallempfehlungen. In diesem Kapitel wird ein allgemein verwendbares Modell zur Priorisierung von Testfallempfehlungen vorgestellt, das die Grundlage einer solchen CodeCover-Erweiterung darstellt. Mit diesem Modell können Quellcodeblöcke nach ihrer Wahrscheinlichkeit, einen Fehler zu enthalten, sortiert werden.

Eine Implementierung des Modells kann eine Liste von Testfallempfehlungen, wie die von CodeCover generierte, priorisieren. Die Empfehlungen werden nach zwei groben Kriterien geprüft und sortiert. Diese entsprechen den Begriffen „Risikokosten“ und „Eintrittswahrscheinlichkeit“ des risikobasierten Tests.

Fehlerschwere Nicht alle Fehler wiegen gleich schwer. Fehler, die dem Programmablauf bzw. dem Benutzer mehr schaden, sollten bevorzugt vor denen behandelt werden, die den Programmablauf bzw. den Benutzer nicht stören. Codeblöcke, die potenziell schwerwiegendere Fehler enthalten, sollten bevorzugt durch einen Testfall abgedeckt werden.

Fehlerwahrscheinlichkeit Nicht in jedem unüberdeckten Codeblock befindet sich mit gleicher Wahrscheinlichkeit ein Fehler. Die Codeblöcke, die eine höhere Wahrscheinlichkeit aufweisen, einen Fehler zu enthalten, sollten bevorzugt durch einen Testfall abgedeckt werden.

Weder Fehlerschwere noch Fehlerwahrscheinlichkeit sind objektiv exakt ermittelbar. Da also keine Verfahren zum Errechnen der tatsächlichen Werte für Fehlerschwere und Fehlerwahrscheinlichkeit in einem Codeblock zur Verfügung stehen, ist ein Modell zur Fehlerprognose grundsätzlich auf Heuristiken angewiesen. Ein Priorisierungsmodell mit guten Heuristiken kann aber als wesentliches Ergebnis die zehn besten Ergebnisse ausgeben. Als Vergleich bietet sich ein Suchergebnis einer Web-Suchmaschine an. Ein solches Suchergebnis enthält oft weit über 200.000 Webseiten. Relevant sind aber nur die ersten zehn Ergebnisse. Auch diese Priorisierung wird mit Hilfe von Heuristiken vorgenommen und genügt den Anforderungen.

In diesem Kapitel werden die Grundbegriffe, Bewertungsebenen, Heuristiken und Annahmen, auf denen das Priorisierungsmodell basiert, vorgestellt. Anschließend wird die Wertung der Ergebnisse der einzelnen Heuristiken besprochen und wie diese untereinander gewichtet und verrechnet werden. Danach wird beschrieben wie die Priorisierung in einzelnen Schritten, abläuft.

Das Priorisierungsmodell ist nicht starr auf eine Menge von Heuristiken ausgelegt, sondern kann durch zusätzliche Heuristiken erweitert werden, wenn diese geeignet erscheinen.

4.1 Grundbegriffe des Modells

In diesem Abschnitt werden die zum Verständnis des Modells notwendigen Begriffe erklärt.

- Ein **Fehlerindikator** ist eine Information, die zur Fehlerprognose verwendet werden kann. Beispiele sind „LOC einer Datei“ oder „Modul durch Unit-Tests getestet“.
- Eine **Fehlerdatenquelle** ist eine Informationskategorie, die Hinweise auf mögliche Fehler gibt. Eine Fehlerdatenquelle kann dabei ein Teil des Produktes, eine Person oder Dokumentation sein. Fehlerindikatoren sind einzelne Informationsaspekte und ergeben zusammen eine Fehlerdatenquelle. Ein Beispiel ist die Fehlerdatenbank „Quellcode“, zu der mehrere Codemetriken gehören.
- Die Ausgabe der Analyse eines mit CodeCover ausgeführten Tests besteht aus **Testfallempfehlung**en. Eine Testfallempfehlung enthält ein unüberdecktes Stück Code, beispielsweise einen `If`-Block. Zudem ist das Prädikat bekannt, das den Kontrollfluss in den Block kontrolliert.
- Die **Fehlerfindwahrscheinlichkeit** einer Testfallempfehlung ist die vom Modell einer Testfallempfehlung zugeschriebene Wahrscheinlichkeit, einen Fehler zu finden. Die Fehlerfindwahrscheinlichkeit wird in Punkten auf einer offenen Skala angegeben und ist relativ zu der Punktbewertung anderer Empfehlungen zu verstehen.
- Die **Bewertung** oder **Wertung** bezeichnet die von einer Heuristik an eine Datei, Zeile oder an einen Block vergebene Fehlerfindwahrscheinlichkeit in Punkten. Die **Gewichtung** bezeichnet die Multiplikation einer Bewertung mit einem Faktor, den entweder der Benutzer vorgegeben hat oder der im Modell vorgegeben ist.

4.2 Bewertungsebenen

Heuristiken zur Fehlerwahrscheinlichkeit und Fehlerschwere geben ihre Bewertungen für verschiedene Ebenen ab. Manche Heuristiken geben die Fehlerwahrscheinlichkeit für eine Datei an, andere für Codeblöcke, manche sogar für Bereiche von Codezeilen oder einzelne Zeilen.

Um die Wahrscheinlichkeit eines Fehlers in einem unüberdeckten Codeblock zu bewerten, muss das Modell daher die Heuristiken für Codeblock, Datei und Zeilen kombinieren. Bei der Untersuchung eines Codeblocks werden deshalb die Werte für diesen Codeblock, für die Datei, in der er sich befindet, und für die Zeilen im Codeblock zusammengezählt. Die Summe der einzelnen Heuristiken ergibt so die Bewertung für den Codeblock.

4.3 Überblick über das Modell

Abbildung 4.1 gibt einen Überblick über die Funktionsweise des Priorisierungsmodells und zeigt, wie Informationen über eine Empfehlung gewonnen werden.

Eine CodeCover-Testfallempfehlung besteht hauptsächlich aus einer Referenz auf einen unüberdeckten Codeblock. Zusätzlich ist noch bekannt, welche Testfälle den Codeblock tangiert haben und welches Prädikat den Kontrollfluss zum Codeblock kontrolliert.

Der Codeblock befindet sich in einer Datei und besteht aus einzelnen Zeilen in dieser Datei. Datei und Zeile werden durch einzelne Fehlerdatenquellen bewertet. Diese Bewertungen fließen in die Bewertung des Codeblocks ein, dessen Bewertung die Bewertung der Testfallempfehlung ausmacht.

Die untersten fünf Boxen bezeichnen Fehlerdatenquellen. Die Inhalte dieser Boxen sind Fehlerindikatoren. Weitere Fehlerindikatoren der in der Grafik nicht explizit aufgeführten Fehlerdatenquelle „CodeCover-Überdeckung“ sind die Inhalte der Boxen „Prädikat“ und „Black-Box-Testfall“. Die im Modell verwendeten Fehlerdatenquellen und -indikatoren werden im folgenden Kapitel näher vorgestellt.

4.4 Fehlerdatenquellen

Das Priorisierungsmodell stützt sich auf sechs Fehlerdatenquellen, die wiederum einzelne Fehlerindikatoren beinhalten. Die verwendeten Fehlerdatenquellen sind:

- Quellcode (Thesen 1-6)
- Versionsgeschichte (Thesen 7-10)
- Stressfaktoren (These 23)
- Expertenwissen (Thesen 8, 10, 12, 16, 17)
- Verwendete Qualitätssicherungsmaßnahmen (Thesen 14, 24)
- CodeCover-Überdeckung (Thesen 11-13, 15, 22)

In den folgenden Abschnitten werden diese Fehlerdatenquellen vorgestellt. Es werden ihre Relevanz und ihre praktische Anwendbarkeit, also das Beschaffen und Auswerten der Daten, diskutiert. Dabei wird auf die in der Literaturrecherche aufgestellten Thesen (siehe 2.6) referenziert.

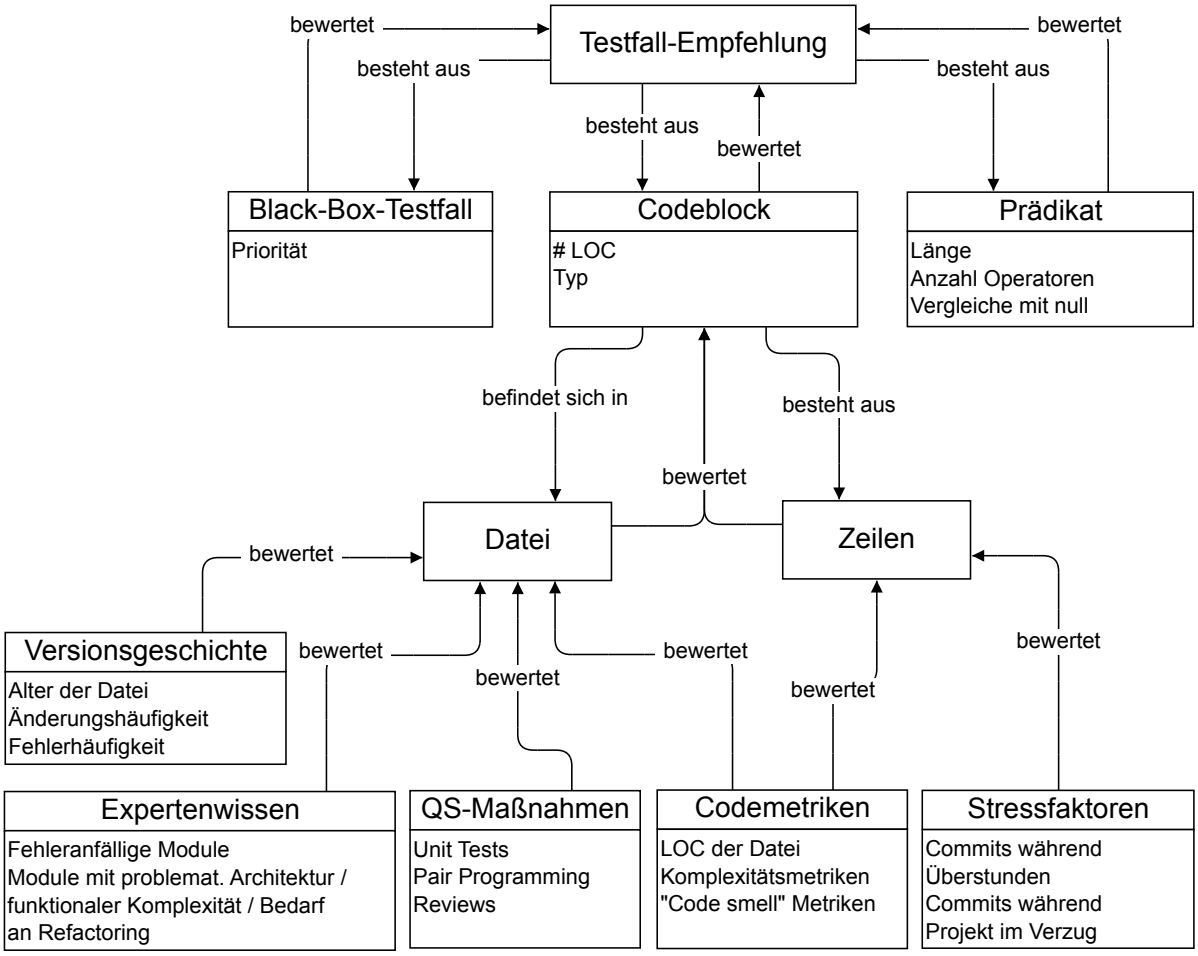

Abbildung 4.1: Überblick über die Konzepte des Modells und ihre Beziehungen

4.4.1 Quellcode

Im Quellcode eines Programms befinden sich die bei der Implementierung gemachten Fehler. Fehler, die in früheren Phasen der Entwicklung der Software gemacht wurden, manifestieren sich hier. Es liegt also nahe, den Quellcode des zu testenden Programms nach Fehlern zu untersuchen. Zwei Möglichkeiten hierfür sind die Auswertung von Codemetriken und die Suche nach Fehlermustern.

Wahrscheinlich sind im Zielprojekt nicht alle Metriken zur Fehlerprognose geeignet (Thesen 19 und 20). Manche werden nicht oder nur schwach mit der tatsächlichen Fehlerdichte korrelieren, manche hingegen sehr gut [NBZ06]. Daraus folgt, dass die hier angegebenen Metriken nicht für jedes Projekt gleich gute Ergebnisse liefern. Es bleibt dem Tester überlassen, für sein Projekt die richtigen Metriken zu wählen.

Werkzeuge zur statischen Codeanalyse suchen mittels Verfahren wie Datenflussanalyse, Überprüfung von Zusicherungen, Kontrollflussanalyse oder Vergleich mit typischen Fehlermustern nach möglichen Fehlern. Sie sind eine einfache Möglichkeit, eine gute Fehlerprognose zu erhalten. Das Werkzeug FindBugs scheint dafür wegen der vergleichsweise niedrigen Zahl an Warnungen und der sehr geringen false-positive-Quote gut geeignet zu sein [RAF04]. Zudem wird vom Entwickler kein Aufwand durch das Schreiben von Annotationen oder Zusicherungen verlangt, was für die universelle und einfache Anwendbarkeit von FindBugs sorgt. Für den Einsatz in diesem Priorisierungsmodell sind andere Werkzeuge zur statischen Codeanalyse zur Fehlermustersuche auch denkbar. Beispiele solcher Werkzeuge finden sich in [RAF04].

Folgende Fehlerindikatoren gehören zur Fehlerdatenquelle „Quellcode“:

LOC einer Datei Die Lines Of Code einer Datei ist eine sehr einfach zu erhebende Metrik, die häufig zur Fehlerprognose herangezogen wird ([BOW06], [OWB05], [Neu05], These 3).

LOC einer Methode Die LOC einer Methode ist laut [Aml00] zur Fehlerprognose geeignet.

Anzahl der Unteraufrufe in einer Methode Laut [Neu05] gut zur Fehlerprognose geeignet.

Komplexitätsmetriken Es gibt viele Komplexitätsmetriken, die zur Fehlerprognose verwendet werden können. Beispiele sind die zyklomatische Komplexität nach McCabe oder die Metriken für objektorientierte Software von Chidamber und Kemerer, sowie die Kopplungsmetriken von Martin ([HPH+09], Thesen 1, 4, 6).

FindBugs-Warnungen pro Datei Die Ausgabe von FindBugs besteht aus Warnungen und Fehlern. Eine Warnung ist ein verdächtiges Codemuster, ein Fehler ist eine Warnung mit deutlich höherer Trefferwahrscheinlichkeit. Die Ausgabe von FindBugs wird daher im Modell auf zwei verschiedene Arten ausgewertet. Bei diesem Fehlerindikator werden die Warnungen einer Datei gezählt und auf die Fehlerwahrscheinlichkeit der Datei angerechnet (vgl. 4.2).

FindBugs-fehlermarkierte Zeilen Die Fehlermarkierungen auf Zeilenebene von FindBugs werden auf die entsprechenden Zeilen angerechnet.

4.4.2 Versionsgeschichte

Die Versionsgeschichte einer Datei bezeichnet die Summe aller Änderungen an dieser Datei, die im Versionskontrollsystem festgehalten sind. Die Versionsgeschichte beginnt mit dem ersten Checkin der Datei und setzt sich über die daran vorgenommenen Änderungen fort. Eine Änderung enthält den Zustand vor und nach der Änderung, sowie Autor und Zeitpunkt der Änderung. Damit lassen sich aus der Änderung die geänderten Dateien, Methoden und Zeilen sowie Zeitpunkt und Autor der Änderung ermitteln. Das Alter lässt sich aus dem Datum des ersten Checkins ableiten.

Daraus ergeben sich die Fehlerindikatoren:

Alter der Datei Je länger eine Datei im System ist, desto geringer ist die Wahrscheinlichkeit, dass sie einen Fehler enthält ([BOW06] [OWB05] [Bac99], These 9). Denn je länger die Datei im System war, desto mehr Zeit hatte ein Fehler, entdeckt zu werden oder einen Fehlerzustand bei der Ausführung zu verursachen.

Änderungen im letzten Release / Jahr Die Anzahl der Änderungen an einer Datei ist ein Hinweis auf mögliche Fehler, da mit einer gewissen Wahrscheinlichkeit Änderungen und Problembehebungen neue Fehler in das System einführen ([SZZ05]). Wurde eine Datei im letzten Entwicklungszeitraum (häufig) bearbeitet, ist dies daher ein Indiz für Fehler ([BOW06] [Aml00] [Bac99] [Neu05], Thesen 7 und 10). Mehr Änderungen sprechen für eine höhere Fehlerwahrscheinlichkeit.

Eine mit Versionsinformationen kombinierbare Fehlerdatenquelle ist eine Fehlerdatenbank, bzw. die gesammelten Daten des Fehlerverwaltungssystems. Aus ihnen lässt sich ablesen, welche Stellen der Software Fehler enthielten. Enthalten die einzelnen Fehlereinträge Referenzen zu Commits im Versionskontrollsystem (oder andersherum), lassen sich die Datenbestände von Versionskontrollsystem und Fehlerdatenbank kombinieren [SZZ05].

Die Kombination der Daten von Versionskontrollsystem und Fehlerdatenbank setzt entweder eine automatische, technisch erzwungene, Lösung oder viel Disziplin auf Seiten der Entwickler voraus. Werden die Fehler nicht vollständig mit den Änderungen im Versionskontrollsystem verknüpft, ergeben sich große Ungenauigkeiten in der Auswertung. Sollten diese Daten allerdings verfügbar sein, können zusätzlich die folgenden Fehlerindikatoren verwendet werden:

Im letzten Release fehleranfällige Dateien Es lässt sich ermitteln, in welchen Dateien im letzten Release Fehler gefunden wurden. Diese Dateien haben eine erhöhte Wahrscheinlichkeit, wieder Fehler zu enthalten ([Mye01], Thesen 8 und 10).

Dateien mit fehlerreicher Vergangenheit Ebenso kann über den Zeitraum des letzten Releases hinaus nach Fehlern in der Vergangenheit einer Datei gesucht werden. (These 10)

4.4.3 Stressfaktoren

Die Arbeitsbedingungen, insbesondere der auf den Entwicklern lastende Zeitdruck, wirken sich auf die Programmqualität aus (These 23). Ein beim Industriepartner befragter Entwickler bestätigt diese These. Steht ein Entwickler unter starkem Zeitdruck oder liegt sein Projekt weit hinter dem Zeitplan, so vermutet er, dass die Arbeit eine höhere Fehlerrate aufweist. Aus einem Projektbericht (z.B. ein Burn Down Chart im Scrum-Prozess) oder aus einer Arbeitszeitdokumentation lassen sich Zeiten hoher Arbeitsbelastung ablesen.

Fehlerindikatoren aus der Fehlerdatenquelle Stressfaktoren:

Änderungen, während Überstunden gemacht wurden Überstunden erhöhen die Arbeitsbelastung des Entwicklers, was zu niedrigerer Konzentration führt und Fehler fördert.

Änderungen, während das Projekt im Verzug war Ist ein Projekt in Verzug, wird i.d.R. schneller und häufig schlampiger gearbeitet, um die verlorene Zeit wieder zu gewinnen. Dies führt laut befragtem Entwickler zu mehr Fehlern.

4.4.4 Expertenwissen

Da sämtliche bisher beschriebene Verfahren eines gewissen Analyseaufwandes bedürfen, liegt es nahe, auch einen Entwickler oder Tester, der das Produkt gut kennt, zu befragen. Arbeitet ein Entwickler länger an einem Produkt, wird er das Produkt gut kennen. Er wird wahrscheinlich Erfahrungswerte haben, welche Stellen des Produktes weniger gut als andere sind.

Diese Fehlerdatenquellen gehören zur Fehlerdatenquelle Expertenwissen:

Module oder Dateien mit schlechter Entwurfs-Qualität Ein Entwickler, der das Produkt gut kennt, kann möglicherweise Dateien oder Module mit schlechter Entwurfsqualität nennen. In diesen Modulen ist die Fehlerdichte evtl. höher.

Module oder Dateien mit hoher funktionaler Komplexität Schwierige Berechnungen, solche, die hohe Genauigkeit erfordern oder für den Kunden besonders wichtig sind, kann ein Entwickler identifizieren. Dort ist die Fehlerwahrscheinlichkeit höher als in simpleren Dateien und Modulen.

Die 20% der Module mit der vermuteten höchsten Fehleranfälligkeit Nach These 16 entspricht die Fehlerverteilung im Programmcode ungefähr einer 20/80-Pareto-Verteilung. Es liegt also nahe, einen Entwickler zu bitten, das fünftel der Module auszusuchen, in denen er die höchste Fehlerdichte vermutet.

Module oder Dateien, die der Entwickler gerne neu schreiben würde Ein Entwickler kann möglicherweise Codestellen angeben, die er gerne neu schreiben würde. In diesen sollte besonders nach Fehlern gesucht werden.

4.4.5 Verwendete Qualitätssicherungsmaßnahmen

Während des Entwurfs und der Codierung durchgeführte Qualitätssicherungsmaßnahmen sollen die Anzahl der Fehler im Quellcode senken. Codebereiche, die solchen Qualitätssicherungsmaßnahmen unterzogen wurden, haben demnach eine geringere Fehlerdichte als vergleichbare Module, bei denen solche Maßnahmen nicht durchgeführt wurden.

Fehlerindikatoren aus der Fehlerdatenquelle verwendete Qualitätssicherungsmaßnahmen:

Überdeckung durch Unit-Tests Es ist anzunehmen, dass ein Modul, das mit Unit-Tests geprüft wurde, weniger Fehler enthält als ein ähnliches Modul, bei dem auf Unit-Tests verzichtet wurde. Die prozentuale Überdeckung eines Moduls kann dann als Indikator verwendet werden.

Modul unter Einsatz von Pair-Programming entwickelt Pair Programming dient zur Steigerung der Qualität [LL07]. Ein unter Pair Programming entwickeltes Modul hat demnach eine geringere Fehlerdichte.

Modul wurde Reviews oder Durchsichten unterzogen Ein durch Reviews oder Durchsichten geprüftes Stück Code hat eine geringere Fehlerdichte als ein vergleichbares Stück Code, das nicht so geprüft wurde.

4.4.6 CodeCover-Überdeckung

Aus einer Testdurchführung mit CodeCover lässt sich eine Liste von unüberdeckten Codestellen nach [Sch10] erstellen. Aus dieser Liste selbst lassen sich Priorisierungskriterien ableiten. In diesem Sinne sind die hier angegebenen Indikatoren nicht alle Indikatoren für die Fehlerwahrscheinlichkeit, da sich einige mit der Fehlerschwere und der Erhöhung der Überdeckung befassen. Aus Konsistenzgründen wurde der Begriff jedoch beibehalten.

Fehlerindikatoren aus der Fehlerdatenquelle CodeCover-Überdeckung:

Typ des unüberdeckten Codeblocks Nach [Sch10] sollten `If`-Blöcke relativ hoch gegenüber unwirksamen Bedingungstermen oder Schleifenwiederholungen (abhängig vom Testziel) gewichtet werden (These 15).

Anzahl der unüberdeckten Codezeilen Ist es möglich, mit einem einzigen Testfall relativ viele weitere Zeilen zu überdecken, sollte dieser höher gewichtet werden.

Anzahl der Operatoren im Prädikat Ist das Prädikat überaus komplex, spricht dies für hohe funktionale Komplexität im Code, die möglicherweise Fehler nach sich zieht.

Priorität des zu Grunde liegenden Black-Box-Testfalls Da jede unüberdeckte Codestelle bei der Durchführung eines Black-Box-Testfalls gefunden wurde, bietet es sich an, Codestellen, die durch hoch priorisierte Testfälle tangiert wurden, höher zu bewerten (These 11).

Anzahl der tangierenden Testfälle Wird eine Codestelle von sehr vielen Testfällen passiert, handelt es sich vermutlich um häufig ausgeführten Code, in dem ein Fehler vermutlich eher auffällt, als in Codestellen, die nur von sehr wenigen (oder nur einem einzigen) Testfällen überdeckt werden. Daher sollten Codestellen, die nur von wenigen Testfällen überdeckt werden, höher gewichtet werden (These 12).

4.5 Bewertung der Fehlerwahrscheinlichkeit

Das vorgestellte Modell verwendet die oben beschriebenen sechs Fehlerdatenquellen, um eine Bewertung der Fehlerwahrscheinlichkeit eines unüberdeckten Codeblocks zu ermöglichen. Das Modell lässt allerdings die genaue Verrechnung der Bewertungen der einzelnen Fehlerindikatoren offen. Gründe hierfür sind die Probleme, die das Aufstellen einer vordefinierten Berechnungsvorschrift erschweren. Eine Implementierung des Modells muss die folgenden

Probleme durch vordefinierte Gewichtungen lösen, oder dem Benutzer die Gewichtung überlassen:

Die Bewertungen der Fehlerindikatoren liegen auf verschiedenen Skalen Während manche Indikatoren Werte auf einer nach oben offenen Rationalskala liefern, liefern manche Indikatoren Werte auf einer Ordinalskala. Die Metriken LOC und die meisten anderen Komplexitätsmetriken liefern Werte auf einer Rationalskala. Die Priorität eines überdeckten Testfalls liegt auf einer Ordinalskala. Ob eine Codezeile durch einen Commit beeinflusst wurde, während das Projekt im Verzug war, wird nur durch einen Wahrheitswert angeben.

Abbildung auf Rationalskala Um die Werte der Ordinalskalen auf eine Rationalskala zum Verrechnen abzubilden, müssen den Werten dieser Skalen Zahlenwerte zugewiesen werden.

Skalierung der Zahlenwerte Die Wahrheitswerte true und false können auf die Zahlenwerte 1 und 0 abgebildet werden, aber auch auf die Zahlenwerte 10 und 5. Es muss eine Skalierung und Verschiebung, also eine Gewichtung der Werte, vorgenommen werden können. Die Resultate der einzelnen Fehlerindikatoren sollten dabei so skaliert werden, dass kein Indikator gegenüber den anderen unbedeutend ist.

Verschiedenheit der Zielprojekte Eine Gewichtung der Indikatoren mag für ein Programm gute Ergebnisse liefern. Wie in [NBZ06] festgestellt unterscheiden sich Programme aber so sehr, dass die Gewichtungen, die in einem Programm gute Ergebnisse liefern, in einem anderen schlechte Ergebnisse liefern können. In einem Programm haben möglicherweise besonders lange Dateien die größte Fehlerdichte, während in einem anderen die korrekte Ausführung von `Catch`-Blöcken besonders wichtig ist.

4.6 Ablauf der Priorisierung

Der prinzipielle Ablauf der Priorisierung mit Hilfe des vorgestellten Modells besteht aus drei Schritten.

1. **Erstellung der unpriorisierten Liste von Testfallempfehlungen**: Die CodeCover-Messergebnisse werden vom Benutzer angegeben und anschließend automatisch ausgewertet. Hierbei entsteht die Liste von Testfallempfehlungen. Die einzelnen Empfehlungen enthalten den unüberdeckten Codeblock (angegeben als Datei + Offset vom Dateianfang). Bei `If-` und `Switch`-Anweisungen sowie Schleifen ist das Prädikat angegeben. Bei `Catch`-Blöcken die Art der Exception.
2. **Bewertung der Fehlerdatenquellen und Fehlerindikatoren**: Für jede Testfallempfehlung wird jeder Fehlerindikator nach seiner Wertung „befragt“. Dabei werden entweder automatische Auswertungen durchgeführt oder vom Benutzer definierte Dateien mit diesen Informationen werden eingelesen.

3. **Die Liste wird sortiert**: Die unüberdeckten Codeblöcke enthalten nun Bewertungen der verschiedenen Fehlerindikatoren. Diese werden anhand der vom Benutzer eingestellten Gewichtungen bewertet (siehe 4.5) und nach der Summe der Werte der Fehlerdatenquellen sortiert. Diese Liste stellt das Endergebnis der Priorisierung dar. Der Benutzer sollte nun die Testfallempfehlungen von hoher zu niedriger Wertung hin bearbeiten.

4.7 Bewertung des Modells

Möchte ein Tester ohne konkrete, aus der Auswertung eines Glass-Box-Tests gewonnene, Testfallempfehlungen eine Testsuite verbessern, stehen ihm dafür als Anhaltspunkte der gesamte Quellcode, das Wissen der Entwickler sowie Spezifikationsdokumente zur Verfügung. Er muss also aus diesen teilweise unkonkreten Informationen konkrete Testfälle gewinnen. Dabei läuft er Gefahr, dass zum einen redundanten Testfälle entstehen, zum anderen ist es nicht wirtschaftlich, eine schon bestehende Testsuite durch das Hinzufügen weiterer zufälliger Testfälle zu erweitern. Besser wäre es, die Testfälle dort anzusiedeln, wo beispielsweise bisher nicht durch Tests überdeckter Code steht, oder wo Fehlerwahrscheinlichkeit und Fehlerkosten besonders hoch sind.

Die von CodeCover generierte Empfehlungsliste ist ein erster Schritt auf diesem Weg. Mit dieser Liste hat der Tester Anhaltspunkte für Code, der bisher nicht von der Testsuite erfasst worden ist. Allerdings ist die Empfehlungsliste in der Regel unüberblickbar lang [Sch10]. Der Entwickler müsste die Liste der Reihe nach abarbeiten oder sich zufällig einige Empfehlungen heraussuchen. Auch dies ist nicht wirtschaftlich, da der Zeitaufwand enorm ist und es keine Anhaltspunkte für die Qualität der neuen Testfälle gibt.

Eine nach mehreren Heuristiken priorisierte Liste von Testfällen erlaubt dem Tester jedoch, seine Testfallauswahl auf die Testfälle zu konzentrieren, die eine höhere Wahrscheinlichkeit haben, einen Fehler aufzudecken oder einen gravierenderen Fehler zu finden. Unter der Annahme, dass das getestete Programm eine Fehlerdichte von fünf Fehlern pro 1000 LOC hat, bedeuten 200 weitere überdeckte Codezeilen durchschnittlich einen gefundenen Fehler. Geht man davon aus, dass die vorgeschlagenen Codeblöcke tatsächlich die fehleranfälligeren sind und eine dreifache Fehlerdichte gegenüber dem Rest des Programmes haben, reicht die Überdeckung von ca. 70 Zeilen aus, um einen weiteren Fehler zu finden.

Ob sich die Entwicklung weiterer Testfälle lohnt, ist von den wirtschaftlichen Rahmenbedingungen abhängig. Bei einem Programm, das nur noch drei Monate im Einsatz ist und bei dem nur geringe Fehlerfolgekosten entstehen, wird es sich selten rentieren, neue Testfälle zu suchen. Ist ein Programm hingegen noch einige Jahre in Dienst und der Hersteller haftet für durch Programmfehler entstehende Schäden, kann es sehr wohl wirtschaftlich sein, Aufwand in die Suche von Fehlern zu investieren. Ein gutes Mittel dazu ist die Vervollständigung der vorhandenen Testsuite.

Mit dem Modell ist es jedoch nicht möglich, den genauen Ort eines bestimmten Fehlers mit Gewissheit anzugeben. Genausowenig garantiert das Modell, dass ein Testfall, der aus einer sehr hoch bewerteten Empfehlung entstanden ist, einen Fehler aufdeckt. Es könnte

im entsprechenden Codeblock schlicht kein Fehler sein. Auch kann das Modell nicht für jeden potenziell vorhandenen Fehler eine erhöhte Wahrscheinlichkeit erkennen, wenn keine Heuristik dafür vorliegt. Den Anspruch der ultimativen Sicherheit kann weder das Modell noch sonstwer erfüllen.

Die im Modell enthaltenen Fehlerdatenquellen und Fehlerindikatoren sind eine erste Auswahl der in der Literatur vorhandenen Heuristiken zur Fehlersuche. Das Modell kann und sollte durch weitere Heuristiken erweitert werden, um weitere Informationen über die Fehleranfälligkeit einzubringen.

5 Umsetzung

In den folgenden Abschnitten wird die Umsetzung des beschriebenen Modells in eine Eclipse-Sicht und einen Priorisierungsalgorithmus beschrieben. Dazu gehören die Ziele der Umsetzung sowie die Grundzüge der Architektur der Umsetzung, insbesondere mit Fokus auf die Verwendung als Framework zur Erprobung weiterer Heuristiken.

5.1 Ziele

Ziele der Umsetzung sind die folgenden. Diese wurden alle erreicht:

- Sprachunterstützung für Java.
- Der Benutzer soll mit wenig Aufwand in einer neuen Sicht eine priorisierte Liste von Testfallempfehlungen aus einem in die CodeCover-Ansicht in Eclipse importierten Test-Session-Container[1] erhalten können.
- Der Benutzer soll die Gewichtung der Fehlerdatenquellen einstellen können.
- Der Benutzer soll nach Paketen filtern können.
- Der Benutzer soll nach dem Typ des unüberdeckten Blocks filtern können.
- So viele Fehlerindikatoren wie möglich sollen automatisch ausgewertet werden.
- Der Benutzer soll sich die Details der Bewertung einer Empfehlung anschauen können. Die Punktevergabe soll nachvollziehbar angezeigt werden.
- Der Benutzer soll einzelne Klassen schnell ausschließen können.
- Dem Benutzer soll Hilfestellung zur Erstellung von Fehlerinformationsdateien gegeben werden.
- Die generierte Empfehlungsliste soll exportiert werden können.
- Fehlerindikatoren sollen konfigurierbar sein, d.h. einfach einstellbare Parameter haben.
- Die gemachten Einstellungen sollen nach Neustart von Eclipse erhalten bleiben.

[1] Ein Test-Session-Container ist eine Datei im XML-Format, die von CodeCover während der Instrumentierung angelegt wird. Sie enthält den gesamten Quellcode des Projektes und die Überdeckungsmessergebnisse der Testdurchführung

- Die Implementierung des Modells soll erweiterbar sein, d.h. es soll mit geringem Aufwand möglich sein, weitere Fehlerdatenquellen und Fehlerindikatoren hinzuzufügen, so dass die implementierte Erweiterung als Framework für die Erprobung weiterer Priorisierungsverfahren verwendet werden kann. Die automatische Auswertung von Fehlerindikatoren soll auch nachträglich hinzugefügt werden können.

5.2 Umgesetzte Modellelemente

Die Implementierung des Modells setzt alle wesentlichen Elemente des Fehlerprognosemodells im letzten Kapitel um. Die Generierung der Basisempfehlungen, Fehlerdatenquellen, Fehlerindikatoren, sowie deren Gewichtung und Auswertung sind enthalten.

Die folgenden Fehlerindikatoren sind im Modell vorhanden und können automatisch ausgewertet werden:

- Code: LOC einer Datei
- Code: FindBugs-Warnungen pro Datei
- Code: FindBugs-fehlermarkierte Zeilen
- Versionsgeschichte: Alter der Datei
- CodeCover-Überdeckung: Typ des Codeblocks
- CodeCover-Überdeckung: Anzahl der unüberdeckten Codezeilen
- CodeCover-Überdeckung: Priorität des zu Grunde liegenden Black-Box-Testfalls
- CodeCover-Überdeckung: Anzahl der tangierenden Testfälle
- CodeCover-Überdeckung: Länge des Prädikats

Die folgenden Fehlerindikatoren sind in der Implementierung vorhanden, benötigen aber eine manuelle Eingabe der Daten:

- Versionsgeschichte: Änderungshäufigkeit seit letztem Release
- Versionsgeschichte: Gefundene Fehler im letzten Release
- Versionsgeschichte: In der Vergangenheit fehleranfällige Dateien
- Stressfaktoren: Commits während Überstunden
- Stressfaktoren: Commits während Projekt im Verzug
- Expertenwissen: Module oder Dateien mit schlechter Entwurfsqualität
- Expertenwissen: Module oder Dateien mit hoher funktionaler Komplexität
- Expertenwissen: 20% der Module mit der vermuteten höchsten Fehleranfälligkeit
- Expertenwissen: Module oder Dateien, die der Entwickler gerne neu schreiben würde

- QS-Maßnahmen: Modul durch Unit-Tests getestet
- QS-Maßnahmen: Modul unter Einsatz von Pair-Programming entwickelt
- QS-Maßnahmen: Modul wurde Reviews oder Durchsichten unterzogen

Die Implementierung unterstützt die Gewichtung von Fehlerdatenquellen, jedoch nicht die Gewichtung einzelner Fehlerindikatoren.

5.3 Entwurf und Implementierung

5.3.1 Datenstruktur

Die wesentliche Datenstruktur der CodeCover-Erweiterung ist die Datenstruktur der Fehlerdatenquellen (`ErrorDataSource`) und Fehlerindikatoren (`ErrorIndicator`). Deren Hierarchie ist in Abbildung 5.1 abgebildet. Der `RecommendationGenerator` ist die Basisklasse des Priorisierungsalgorithmus. Er verwaltet die Fehlerdatenquellen, die wiederum aus Fehlerindikatoren bestehen. Diese haben optional Parameter und optional genau einen `DataCollector`. Ein `DataCollector` ist ein Interface, das von Algorithmen zur automatischen Erstellung von Priorisierungsinformationen implementiert werden muss.

Der rechte Teil des Diagramms wird in Abschnitt 5.3.3 erläutert.

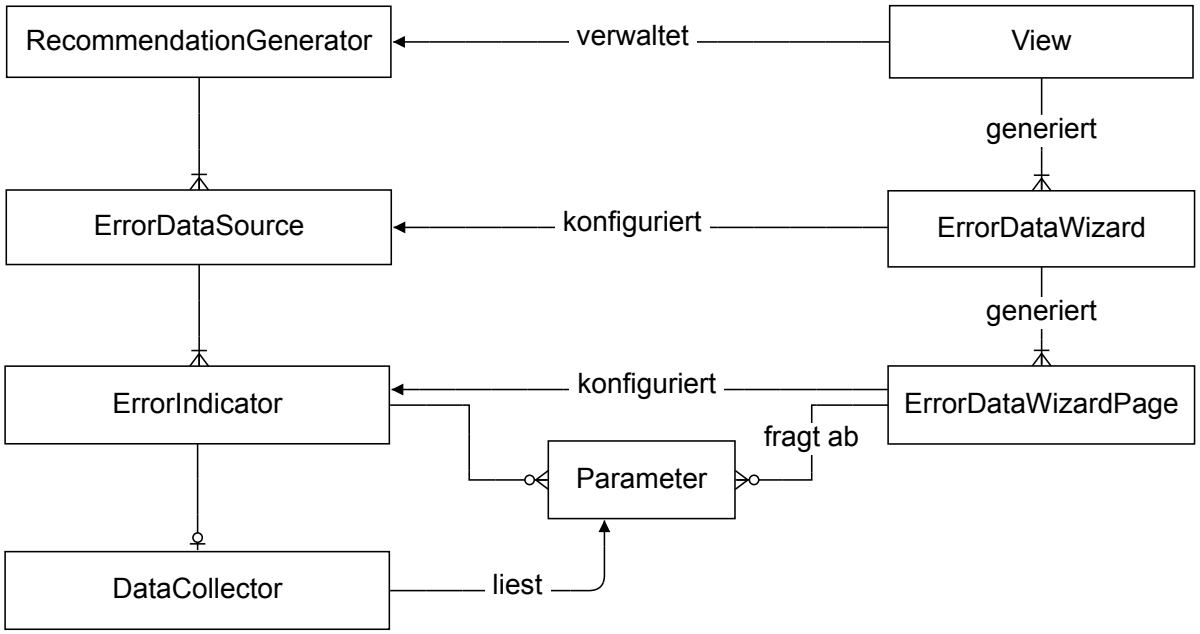

Abbildung 5.1: Datenstruktur der CodeCover-Erweiterung

5.3.2 Ablauf der Priorisierung

Abbildung 5.2 zeigt den Ablauf der vom Benutzer angestoßenen Priorisierung. Nachdem der Benutzer die Priorisierung angestoßen hat, wird zunächst die unsortierte Empfehlungsliste generiert.

Anschließend werden rekursiv mit ihren `invoke`-Methoden alle Fehlerdatenquellen und damit alle Fehlerindikatoren aufgefordert ihre Informationen auszuwerten. Dabei wird für jede Fehlerdatenquelle ein Thread gestartet, was die Dauer des Gesamtvorgangs erheblich senkt[2].

Wird ein Fehlerindikator nicht automatisch ausgewertet (hat keinen `DataCollector`), liest er nach dem Aufruf seiner `invoke`-Methode die eingestellte Fehlerinformationsdatei aus, anstatt den Aufruf an den `DataCollector` weiterzureichen.

Im zweiten Schritt iteriert der `RecommendationGenerator` über die vorhin generierten Rohempfehlungen und befragt jede Fehlerdatenquelle nach ihrer Punktzahl für diese Empfehlung (`getValueFor()`). Die Fehlerdatenquellen geben diesen Aufruf wiederum rekursiv an ihre Fehlerindikatoren weiter, die die tatsächlichen Fehlerpunktzahlen gespeichert haben.

Der letzte Schritt ist die Sortierung. Nun werden die Fehlerpunktzahlen der Fehlerdatenquellen nach den Vorgaben des Benutzers gewichtet (Abbildung 5.5) und sortiert ausgegeben.

5.3.3 Erweiterbarkeit

Die implementierte CodeCover-Erweiterung kann als Framework zum Ausprobieren weiterer Priorisierungsverfahren verwendet werden. Bei Entwurf und Implementierung wurde Wert darauf gelegt, dass neue Fehlerdatenquellen und Fehlerindikatoren mit wenig Programmieraufwand hinzugefügt werden können. Um künftigen Bedürfnissen gerecht zu werden, können Fehlerindikatoren mit Parametern versehen werden, die deren Bewertung bei automatischer Auswertung verändern. Beispielsweise könnte dem Fehlerindikator „LOC der Datei“ ein Parameter für die kritische Zeilenzahl hinzugefügt werden, ab der die doppelten Fehlerpunkte vergeben werden.

Die Fehlerdatenquellen, Fehlerindikatoren und Parameter werden in der Klasse `RecommendationGenerator` einmal angegeben. Sollte ein Indikator automatisch ausgewertet werden können, muss noch eine Implementierung des Interfaces `DataCollector` angegeben werden. Für jede Fehlerdatenquelle wird aus diesen Informationen ein Einstellungsassistent generiert, der für jeden Indikator den Modus (aus, automatisch, manuell, siehe Abbildung 5.3) sowie die definierten Parameter abfragt. Die andernfalls notwendige zeitraubende Implementierung von Benutzeroberflächen für die Konfiguration entfällt.

[2] Beispielsweise sind Auswertungen des Versionskontrollsystems von der Netzwerkgeschwindigkeit abhängig, während die Auswertung von FindBugs von der lokalen Rechenleistung abhängt.

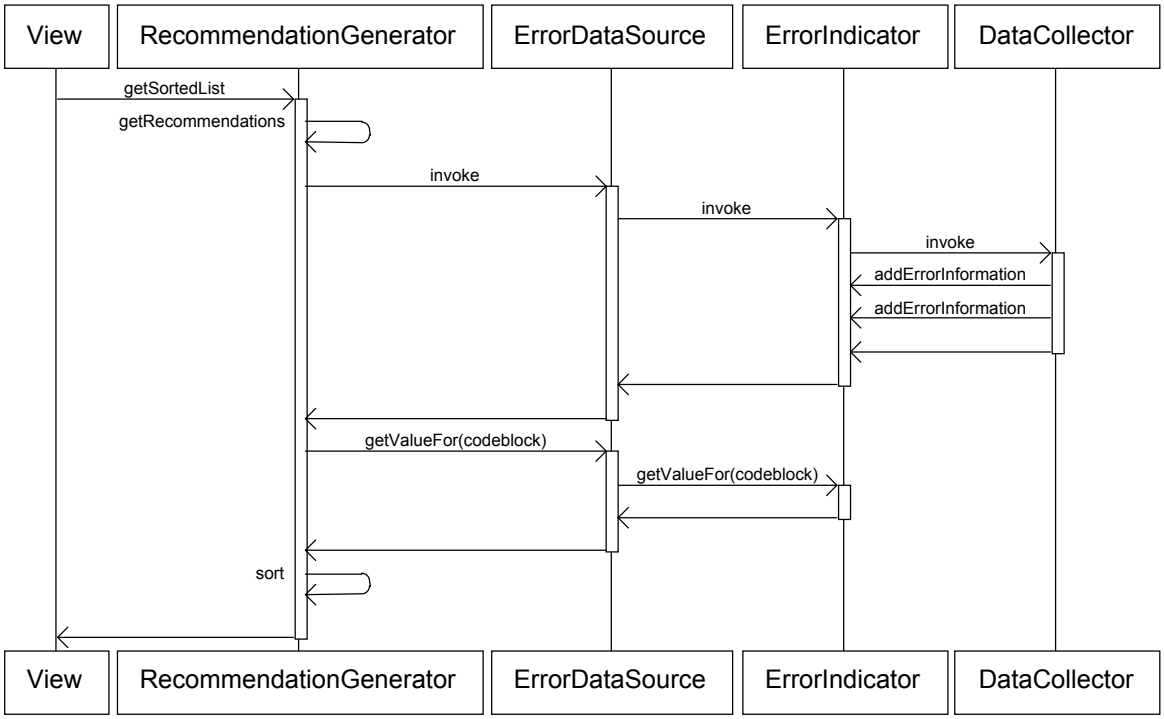

Abbildung 5.2: Ablauf der Priorisierung

Dies erspart dem Programmierer wesentlichen Aufwand beim Einbinden weiterer Fehlerdatenquellen und Fehlerindikatoren und unterstützt so das effiziente Bewerten neuer Priorisierungsverfahren.

Abbildung 5.1 zeigt den Zusammenhang zwischen den Elementen der Datenstruktur und den Elementen der Konfigurationsassistenten. Ein Assistent (Wizard) konfiguriert eine Fehlerdatenquelle. Pro Dialogseite des Assistenten wird ein Fehlerindikator konfiguriert. Auf einer Seite werden alle Parameter eines Fehlerindikators abgefragt.

5.3.4 Eclipse als notwendiger Übersetzer

CodeCover speichert den Quellcode des Prüflings sowie die Überdeckungsmessergebnisse in einer XML-Datei ab, dem sogenannten Test-Session-Container. Dieser enthält zwar den reinen Quellcode des Prüflings, allerdings leider keine vollständigen Informationen über den Pfad einer Quelldatei innerhalb ihres Projektes. Es ist aber notwendig zu wissen, in welcher Datei auf dem Dateisystem sich eine Codestelle befindet, um automatische Auswertungen wie die FindBugs-Analyse oder die Analyse der Versionsgeschichte durchführen zu können, da diese Auswertungen auf Basis des Test-Session-Containers alleine nicht möglich sind. Auch für

die Auswertung manueller Fehlerinformationsdateien muss der absolute Pfad einer Datei bekannt sein, da nur so eine Datei eindeutig referenziert werden kann.

Da es mit CodeCover-Bordmitteln nicht möglich ist, diese Information zu erhalten, wurden die semantischen Möglichkeiten von Eclipse verwendet, um die Abbildung vom Code bzw. einer Klasse darin auf die enthaltende Datei zu ermöglichen. Das Eclipse-Java-Modell-Framework ist in der Lage anzugeben, in welcher Datei eine Klasse enthalten ist. Auf diese Art ist es auch möglich, mithilfe des Subversion-Plugins Subclipse [Sub11a] an die Versionsinformationen der Software im Versionsverwaltungssystem Subversion (SVN) [Sub11b] zu gelangen.

5.3.5 Speicherung von Zwischenergebnissen

Während des Auswertungs- und Priorisierungsprozesses muss eine Kette von zusammengehörenden Informationen des zugrundeliegenden Code-Datenmodells aufgebaut werden. Dieses umfasst die Abbildung von unüberdeckten Zeilen über CodeCover-Code-Hierarchielevels zu Eclipse-Java-Elementen, Datei-Deskriptoren und kompilierbaren Einheiten. Auch muss bekannt sein, Welche Testfälle den Inhalt einer Datei (teilweise) abdecken. Beim Auswerten der Versionsgeschichte muss bekannt sein, welche „Remote-Resource“ zu einer lokalen Datei gehört. CodeCover gibt Codestellen als Offset vom Beginn der Datei an, FindBugs hingegen rechnet in Zeilennummern, so dass eine Abbildung von Zeichen-Offset in einer Datei auf Zeilennummern nötig ist.

Alle diese Informationen werden bei Bedarf ermittelt und anschließend zur späteren Verwendung zwischengespeichert. Ohne diese Optimierungen wäre die Auswertung nicht in akzeptabler Zeit möglich. Realisiert wird dieser Zwischenspeicher (Cache) durch Java-`HashMaps`.

5.3.6 Zukünftige Verbesserungen

Aufgrund des experimentellen Charakters der Implementierung ist diese zwar technisch solide, aber nicht in allen Details abgeschlossen. Folgende Punkte könnten noch erledigt werden:

- Ansprechendere Icons in der Symbolleiste der RecommendationsView
- Sortierung der Empfehlungen nach anderen Kriterien (LOC, Typ des Codeblocks, ...). Die Tabelle sollte nach allen Spalten sortiert werden können.
- Der Benutzer sollte die Gewichtung nicht nur auf Ebene der Fehlerdatenquellen, sondern auch auf Ebene der Fehlerindikatoren vornehmen können. Hierfür eignen sich die schon vorhandenen Parameter.
- Übersichtlichere Darstellung der ausgewählten Fehlerdatenquellen und Fehlerindikatoren mit Parametern und Gewichtung.

- Nicht überdeckte implizite `Else`-Blöcke werden in der Ausgabe als `If`-Blöcke mit einer Länge von null Zeilen angegeben. Dies ist zwar technisch korrekt, aber für den Benutzer nicht intuitiv. Der Codeblock-Typ sollte in diesem Fall als `Impl. else` o.ä. angegeben werden.
- Das Eclipse-Plugin „Metrics“ könnte in CodeCover integriert werden. Seine Möglichkeiten zur Berechnung verschiedener Metriken könnten zur automatischen Auswertung einiger Fehlerindikatoren interessant sein.
- Statt der „Länge des Prädikates“ sollte im gleichnamigen Fehlerindikator die Komplexität des Prädikates betrachtet werden.

5.4 Screenshots

Dieser Abschnitt enthält einige Screenshots der implementierten Erweiterung, die im folgenden kurz besprochen werden.

Abbildungen 5.4 zeigt die ganze neue Eclipse-Sicht. Sie besteht aus der Tabelle, die die Empfehlungen (priorisiert) anzeigt. Die Spalten enthalten die Methode, die den Codeblock enthält, das Prädikat bzw. Statement, das den Kontrollfluss in den Block kontrolliert, den Typ des Blocks, die Summe der Werte der Fehlerdatenquellen und die Werte der einzelnen Fehlerdatenquellen.

Die Konfiguration erfolgt über die Werkzeugleistenknöpfe in der rechten oberen Ecke. Dort werden die Assistenten zur Konfiguration der Fehlerdatenquellen aufgerufen (siehe Abbildung 5.3), die Gewichtung eingestellt (siehe Abbildung 5.5), Filter eingestellt (siehe Abbildung 5.6) und die Priorisierung wie in Abschnitt 5.3.2 geschildert.

In der Tabelle werden dem Benutzer die wesentlichen Informationen zu einer Testfallempfehlung angezeigt: Methode, in der der unüberdeckte Codeblock ist, das Statement bzw. Prädikat des Codeblocks, sein Typ sowie die Bewertungen der Fehlerdatenquellen. Weitere Details zu einer Empfehlung können über das Kontextmenü aufgerufen werden. Siehe dazu Abbildung 5.7.

5.5 Fazit zur Umsetzung

CodeCover wurde durch eine weitere Eclipse-Sicht zur Priorisierung von Testfallempfehlungen erweitert. Die Erweiterung stellt ein Framework dar, mit dem Methoden zur Priorisierung von Testfallempfehlungen evaluiert und mit anderen Methoden verglichen werden können. Dies ermöglicht die schnelle und aufwandsarme weitere Forschung auf diesem Gebiet.

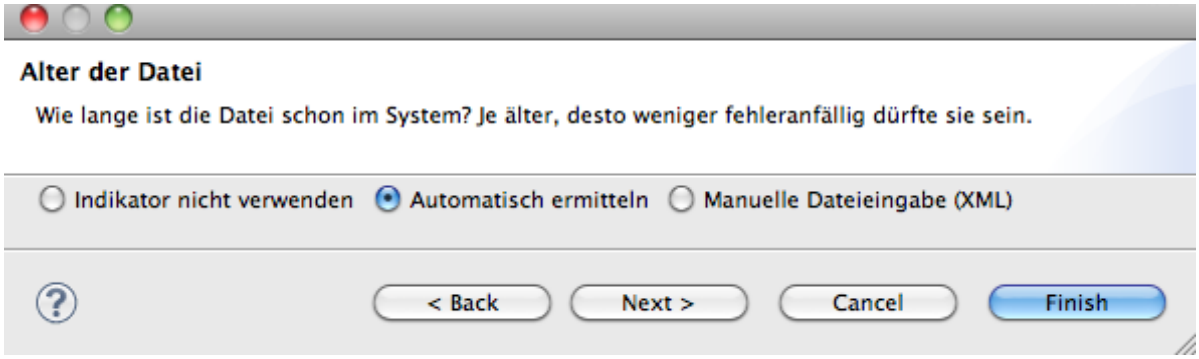

Abbildung 5.3: Konfiguration eines Fehlerindikators

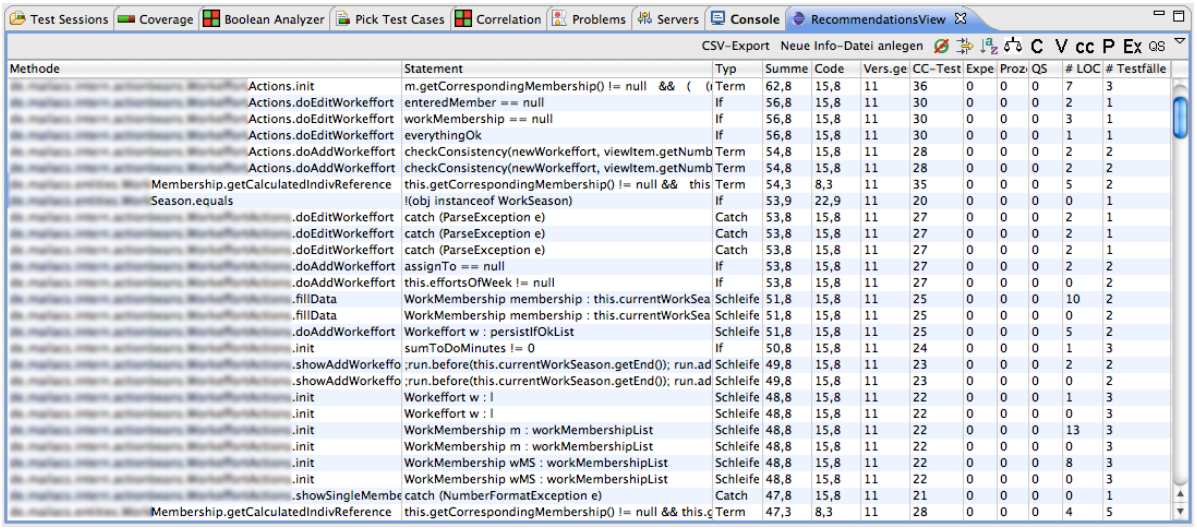

Abbildung 5.4: Sicht zur Anzeige und Priorisierung der Testfallempfehlungen

Es ist ein Grundstock an Fehlerindikatoren implementiert, der in Zukunft auf zweierlei Art erweitert werden kann:

- Es können weitere Fehlerindikatoren zur automatischen Ermittlung von Priorisierungsinformationen aufgerüstet werden. Dadurch sinkt der Aufwand zur Durchführung der Priorisierung und es entfällt die manuelle Pflege von Fehlerinformationsdateien im XML-Format.
- Es können weitere Fehlerindikatoren und Fehlerdatenquellen gesucht und der CodeCover-Erweiterung hinzugefügt werden.

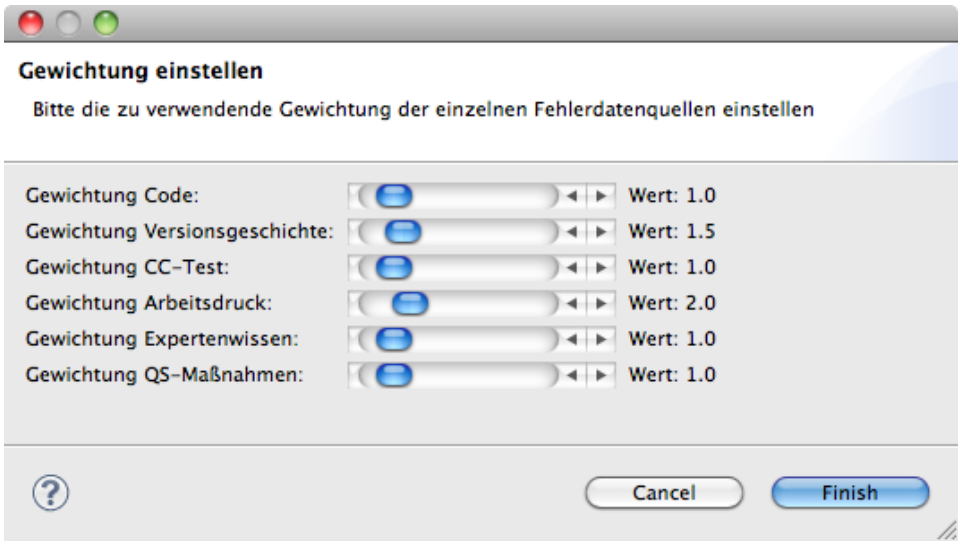

Abbildung 5.5: Dialog zur Gewichtung der Fehlerdatenquellen

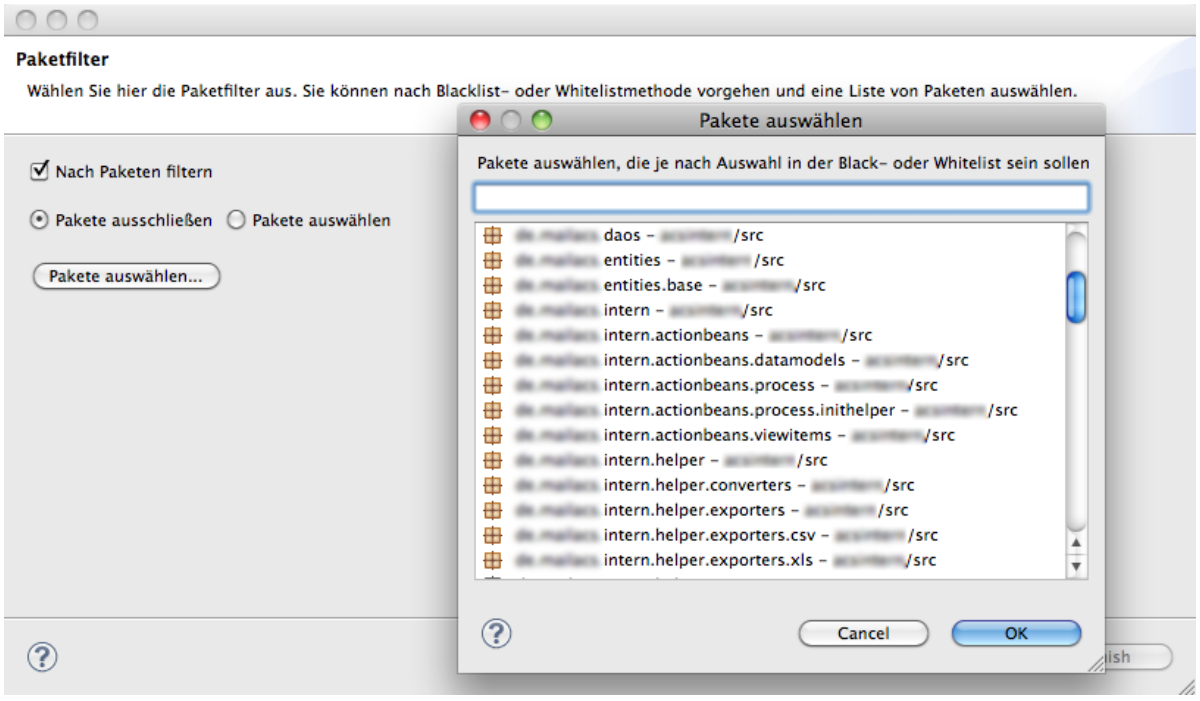

Abbildung 5.6: Dialog zum Filtern von Paketen

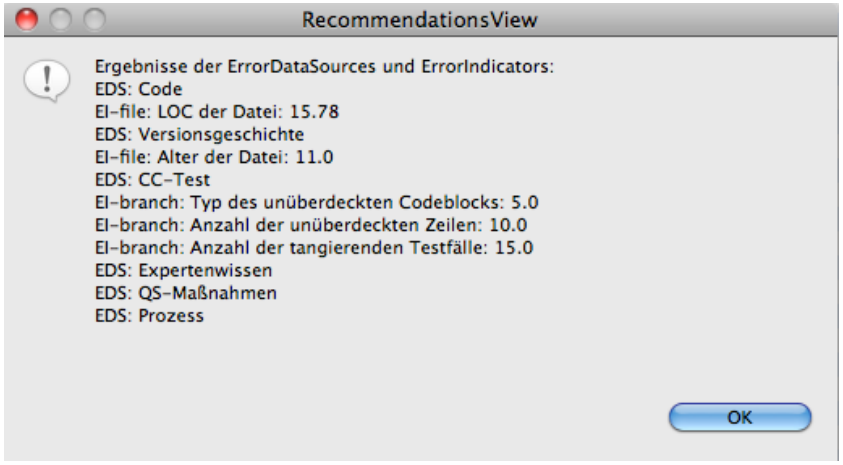

Abbildung 5.7: Detailansicht der Bewertung einer Testfallempfehlung

6 Erprobung

Die Erprobung der neu implementierten CodeCover-Erweiterung fand bei einem Industriepartner statt. Zielsetzung war, festzustellen, ob das verwendete Modell seinen Zweck erfüllen und bei einem der beim Industriepartner entwickelten Programme neue Testfälle empfehlen kann, die besser sind als eine zufällige Auswahl von Testfällen aus der Liste der CodeCover-Empfehlungen.

6.1 Testumgebung

Beim Industriepartner gehört Testautomatisierung seit einigen Jahren zum Testprozess, so dass für die meisten Programme soweit möglich automatisierte Testsuiten vorliegen. Als Werkzeug für die Oberflächentestautomatisierung kommt vor allem IBM Rational Functional Tester (RFT) zum Einsatz.

RFT ist ein Werkzeug für die Testautomatisierung und den Regressionstest insbesondere von Programmen mit graphischer Benutzerschnittstelle. Testskripte (eigentlich Java-Programme) können dabei mit Hilfe eines Aktionsrekorders aufgenommen oder direkt programmiert werden. Das bedeutet, dass alle vom Benutzer durchgeführten Klicks und Eingaben an ein Programm aufgezeichnet werden. RFT generiert daraus Java-Code und Oberflächenobjektbeschreibungen, die sich auf die direkten Eigenschaften eines Objektes sowie seine Position in der Oberflächenhierarchie stützen. Mit Hilfe des Java-Code wird mit diesen Objekten interagiert. So können Benutzeraktionen wie z.B. Klicks aufgeführt werden, in dem der Aufruf `button_ok().click()` ausgeführt wird.

RFT ist gut erweiter- und anpassbar. Unter anderem bietet RFT benutzerdefinierbare Callback-Methoden an, die bei bestimmten Ereignissen, u.a. Testskriptbeginn und -ende ausgeführt werden.

Weitere Informationen zu IBM Functional Tester finden sich in [Rat11].

Die Testfälle selbst werden mit IBM Rational ClearQuest TestManager (CQ) verwaltet. Dort sind Metadaten wie Name, Autor, etc. gespeichert. RFT-Skripte referenzieren mit Hilfe einer Testfall-ID auf die Testfälle in ClearQuest. Es existiert die Möglichkeit, diese Verbindung und damit den Testfallnamen programmatisch abzufragen.

6.2 Geplanter Ablauf

Wesentliches Ergebnis der Diplomarbeit ist eine Methode zur Priorisierung von Testfallempfehlungen von CodeCover, also von unüberdeckten Codeblöcken. Bei der Erprobung an einer Software aus der Praxis soll festgestellt werden, ob das vorgeschlagene Priorisierungsmodell bzw. seine Implementierung geeignet sind, eine Menge von Testfällen zur Implementierung vorzuschlagen, die einer zufälligen Auswahl von Testfällen überlegen ist.

Der Industriepartner gewinnt dabei Informationen über fehlende Testfälle im gewählten Prüfling.

Die Dauer der Erprobung war im Projektplan auf zwei Wochen limitiert.

Da beim Industriepartner mehrere möglicherweise geeignete Clientprogramme vorhanden sind, muss zunächst eines ausgewählt werden, das die folgenden Kriterien erfüllt:

- Programmiersprache Java
- Mit CodeCover instrumentier- und auswertbar
- Entwicklung abgeschlossen
- Für den Kunden bestimmtes Programm im Produktiveinsatz
- Vollständige automatisierte Testsuite
- Noch längere Zeit im Einsatz

Diese Kriterien sollen sicher stellen, dass die Untersuchung machbar ist und für beide Parteien nützliche Ergebnisse liefert.

Zunächst soll der Prüfling von CodeCover instrumentiert und anschließend von der automatischen Testsuite getestet werden. Aus dem sich hieraus ergebenden Test-Session-Container werden die Testfallempfehlungen generiert und sortiert. Sollten die Gewichtungen der Fehlerdatenquellen ein verzerrtes Ergebnisbild ergeben, werden diese angepasst.

Zur Überprüfung, ob das Modell seinen Anforderungen entspricht, sollen einem Experten für den Prüfling zwei Mengen von Testfallempfehlungen vorgelegt werden: Zum ersten eine zufällige Auswahl von 20 Testfallempfehlungen aus den unsortierten Empfehlungen und zum zweiten die 20 höchstbewertesten Ergebnisse der Priorisierung. Die zweite Methode zur Überprüfung besteht darin, dem Experten 20 Testfallempfehlungen zu zeigen, von denen zehn zufällig ausgewählt wurden und zehn vom Priorisierungsmodell als gut bewertet wurden.

Vom Industriepartner benötigte Ressourcen sind zum einen ein Rechnerarbeitsplatz mit Zugang zum Quellcode des Prüflings sowie Lesezugang zum Versionskontrollsystem. Zur Durchführung des Tests sind dazu RFT und CQ mit Zugang zu den Testfällen notwendig. Zum anderen sind einige Stunden eines Experten für den Prüfling nötig, um ggf. Unterstützung bei der Instrumentierung zu leisten, sowie um Fragen zum Prüfling zu beantworten, insbesondere für die Fehlerdatenquellen Expertenwissen, Stressfaktoren und QS-Maßnahmen.

6.2.1 Integration von Functional Tester mit CodeCover

Um das von CodeCover instrumentierte Programm über Beginn und Ende eines Testfalles zu informieren, muss dem instrumentierten Programm bei Beginn und Ende einer Testfalls je eine JMX-Nachricht gesendet werden. Functional Tester bietet für diesen Zweck benutzerdefinierbare Callback-Methoden an, die zu Beginn und am Ende eines Testskriptes ausgeführt werden. Diese Methoden wurden soweit vorhanden erweitert, dass sie die benötigte JMX-Nachricht an den instrumentierten Prüfling schicken, wenn ein Testfall beginnt oder endet. Weitere Modifikationen der Testumgebung waren nicht notwendig.

6.2.2 Risiken

Im Vorfeld der Erprobung wurden mehrere Risiken identifiziert. Einige davon haben im Eintrittsfall Auswirkungen auf die Durchführungsdauer der Erprobung, andere auf die Qualität des Ergebnisses, manche auf die grundsätzliche Durchführbarkeit der Untersuchung beim Industriepartner. Einige dieser Risiken sind eingetreten.

- Technische Schwierigkeiten:
  - Probleme bei der Einrichtung der Entwicklungsumgebung des Clients
  - Probleme bei der Instrumentierung des Clients
  - Probleme bei der Ausführung des Clients (fehlende Testdaten, Testsystem nicht verfügbar)
  - Probleme mit der automatisierten Testsuite
- Organisatorische Schwierigkeiten:
  - Kein Mitarbeiter für die Einrichtung der Entwicklungsumgebung verfügbar
  - Kein Mitarbeiter für die Befragung für die Fehlerdatenquellen Prozess, QS und Expertenwissen verfügbar
  - Kein Mitarbeiter für den Vergleich der Testfallempfehlungen verfügbar

6.3 Tatsächlicher Ablauf

6.3.1 Auswahl des zu bearbeitenden Programms

Beim Industriepartner existieren mehrere Clientprogramme, die prinzipiell für die Erprobung geeignet sind. Alle diese Clients sind Java-Programme, die über eine SOAP-Schnittstelle mit den Servern Daten austauschen und in einer lokalen Entwicklungsumgebung gestartet werden können.

Die neueste Generation dieser Clients hatte vor ca. einem halben Jahr ihre Markteinführung, ist also noch vergleichsweise jung. Diese Clients sind mit SWT entwickelt worden, setzen also auf einer Eclipse-Plattform auf und sind als Plugins für diese implementiert. Für zwei dieser Clients existiert eine automatisierte Testsuite, so dass diese als erstes Ziel ins Auge gefasst wurden. Nach der langwierigen Installation und Einrichtung der Entwicklungsumgebung stellte sich dann heraus, dass der Ablauf der Functional Tester Skripte bei lokaler Ausführung unmöglich ist. Die Autoren der RFT-Integration vermochten das Problem nicht zu lösen.

Die ältere Generation von Clients hat auch eine Testsuite, die allerdings wegen der Außerbetriebnahme der Clients nicht mehr gewartet wird. Die Clients selbst sind noch im Produktiveinsatz, der allerdings bald ausläuft, weswegen der Code nicht mehr gewartet wird und keine Experten dafür vorhanden sind. Der Prüfling ist ein solcher und wurde wegen der Verfügbarkeit einer (halbwegs) brauchbaren Testsuite verwendet.

6.3.2 Prüfling

Das zur Erprobung verwendete Programm ist ein Java-Programm mit Swing-Oberfläche. Die fachliche Bedeutung des Programms ist für die Erprobung irrelevant und wird daher nicht weiter ausgeführt.

Das Programm (im Folgenden „Prüfling“) kommuniziert über eine SOAP-Schnittstelle mit den Servern beim Industriepartner. Die so erhaltenen Daten können angezeigt, gefiltert und auf verschiedene Art manipuliert werden.

Entwicklungsseitig ist der Prüfling recht einfach gehalten. Es handelt sich um ein normales Java-Projekt, das ohne eine Vielzahl an Build-Skripten ausführbar ist, wodurch die Instrumentierung und Ausführung vereinfacht wird.

6.3.3 Einrichtung der Entwicklungsumgebung

Der Prüfling ist ein einfaches Java-Programm, das im Versionskontrollsystem als Eclipse-Projekt vorliegt. Zur Einrichtung musste es nur aus dem Versionskontrollsystem ausgecheckt und als Eclipse-Projekt importiert werden.

6.3.4 Instrumentierung

Die Instrumentierung direkt in Eclipse mithilfe des CodeCover-Plugins funktionierte nicht. Das Übersetzen der instrumentierten Klassen brach in mehreren Überdeckungsvarianten mit Fehlermeldung ab. Die Batch-Instrumentierung funktionierte hingegen und der Prüfling ließ sich mit Instrumentierung und JMX-Schnittstelle für den Empfang von Testfallinformationen starten.

6.3.5 Testumgebung und Testsuite

Wegen Arbeiten am Testsystems war dieses im Zeitraum der Erprobung nicht verfügbar, so dass serverseitig auf das Entwicklungssystem ausgewichen werden musste. Dort war die Testausführung zwar möglich, allerdings mit Einschränkungen. Es waren nicht die von den Testskripten erwarteten Testdaten verfügbar und Fehler im System verhinderten den normalen Ablauf der Testskripte. Diese mussten teilweise einzeln angepasst werden, damit sie auch mit den nicht idealen Bedingungen abliefen. Dadurch trat eine gewisse Abweichung vom Normalablauf auf, die das Überdeckungsergebnis möglicherweise abgefälscht hat. Zudem war die Anpassung der einzelnen Testskripte mühsam und war nur dank reichlich Erfahrung mit RFT und der Testumgebung beim Industriepartner möglich.

6.3.6 Auswertung der Ergebnisse

Es konnte durch die Ausführung der vorhandenen und funktionierenden Testskripte eine Anweisungsüberdeckung von 38%, eine Zweigüberdeckung von 23%, eine Schleifenüberdeckung von 13% sowie eine Termüberdeckung von 22% erreicht werden. Diese Werte schwankten zwischen Model-, View- und Controller-Paketen nur wenig. Einige Funktionalitätsgruppen konnten mangels Testdaten bzw. völliger Nichtfunktion der Testskripte gar nicht verwendet bzw. getestet werden.

Wegen des Alters des Prüflings standen keine nutzbaren Daten zu Änderungen in letzter Zeit zur Verfügung. Da die Entwicklung vor vielen Jahren extern stattfand, sind keine Daten über während der Implementierung möglicherweise verwendete Qualitätssicherungsmaßnahmen verfügbar. Expertenwissen über den Quellcode des Prüflings stand ebenfalls nicht zur Verfügung, bzw. eine Befragung war durch Zeitdruck der Entwickler nicht realisierbar. Es konnte auch kein Entwickler zur Qualität der Testfallempfehlungen befragt werden.

Damit blieben als verfügbare Fehlerdatenquellen der Code selbst, die Versionsgeschichte und die Informationen aus der CodeCover-Überdeckung übrig. Ausgewertete Fehlerindikatoren waren:

- Code: LOC der Datei
- Code: FindBugs-Resultate
- Versionsgeschichte: Alter der Datei
- CodeCover-Überdeckung: Typ des Codeblocks
- CodeCover-Überdeckung: Anzahl der Codezeilen
- CodeCover-Überdeckung: Anzahl der tangierenden Testfälle
- CodeCover-Überdeckung: Länge des Prädikats

Diese relativ schmale Datenbasis ist leider nicht geeignet um das angewandte Verfahren ausführlich bewerten zu können.

Die Liste der generierten Empfehlungen enthält ca. 1700 Einträge. Bei 640 Einträgen enthält das Prädikat einen Vergleich mit `null`. Lässt man den Oberflächen-Code weg, bleiben ca. 800 Empfehlungen übrig, von denen bei 350 das Prädikat einen Vergleich mit `null` enthält.

Ca. 24% der Empfehlungen bezogen sich auf unüberdeckte `If`-Blöcke, 34% auf unüberdeckte implizite `Else`-Blöcke , 28% auf Schleifenüberdeckung, 8% auf Termüberdeckung und je ca. 3% auf Catch- und Switch-Blöcke.

Die ersten 50 Empfehlungen (2,3%) haben summiert 2,96% der Punkte erhalten. Die beste Empfehlung bekam 76,3 Punkte, die schlechteste 16,9. Der Durchschnitt betrug 43,5 Punkte, der Median 42,7. 80% der Empfehlungen bewegten sich zwischen 30 Punkten und 58 Punkten. Abbildungen 6.1 und 6.2 zeigen die Verteilung der Gesamtpunkte. Es ist ersichtlich, dass die Fehlerdatenquelle Versionsgeschichte gleichmäßig zwischen 8 und 15 Punkten schwankt. Sie hat keine wesentliche Auswirkung auf die Summe.

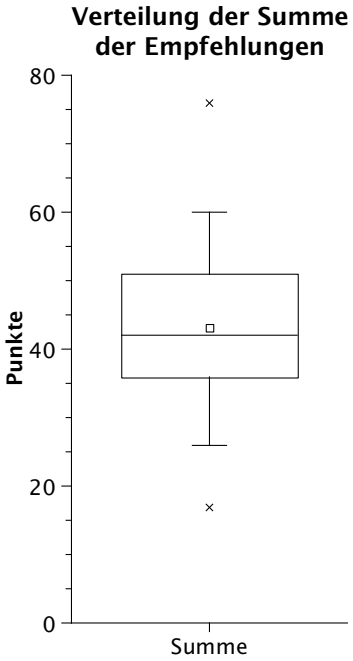

Abbildung 6.1: Boxplot der Verteilung Punkte der einzelnen Datenquellen sowie deren Summe

Die Abbildungen 6.3 und 6.4 zeigen die Punkteverteilungen der einzelnen Fehlerdatenquellen, wiederum ohne und inkl. GUI-Code. Es fällt auf, dass die Versionsgeschichte, im Gegensatz zu CodeCover und Code relativ undifferenzierte Ergebnisse geliefert hat.

Eine Bewertung der Ergebnisse durch einen Programmierer oder Tester beim Industriepartner war nicht möglich, weil aus Zeitgründen kein Mitarbeiter zur Verfügung stand.

Prädikat	LOC	#TF	C	V	CC	Summe
`cache.get(FBZWConstants.CACHE_MONTHLY_TIMEMGMT_SUM) != null`	4	1	31,3	11	34	76,3
`removedSumDate.size() > 0`	1	1	31,3	11	30	72,3
`removedDetailsData.size() > 0`	1	1	31,3	11	30	72,3
`removedEventsData.size() > 0`	1	1	31,3	11	30	72,3
`FBZWConstants.CACHE_TIMEMGMT_EVENTS.equals(key)`	0	1	31,3	11	30	72,3
`isUpdate`	0	1	31,3	11	30	72,3
`(driverIdsListLeft.containsAll(driverIdsListRight) || driverIdsListRight.containsAll(driverIdsListLeft)) && driverIdsListLeft.size() == driverIdsListRight.size()`	0	1	21	9	42	72
`activityComboBox.getItemAt(i).toString().equals( res.getDisplayLabels( LabelConstants.FILTER_DRVTIME_LIST + ."+ FBZWConstants.ONLY_MESSAGES))"`	0	1	20	13	38	71
`((TimemgmtSumRequestDO) requestInput).getMonth() != null`	1	1	21,3	11	34	66,3
`map.containsKey(dataKey) && map.get(dataKey) != driverMap.get(dataKey)`	2	1	21,3	11	34	66,3
`res.getDisplayLabels(LabelConstants.DRIVINGTIME_FILTER_VEHICLELIST_ALL) .equalsIgnoreCase(vehicleSelection)`	11	1	20	13	33	66
`ResourceManager.getUserProperties().getLastSelectedTab() == ((JTabbedPane)getParent()).getSelectedIndex()`	1	1	20	13	31	64
`ResourceManager.getUserProperties().getLastSelectedTab() == ((JTabbedPane)getParent()).getSelectedIndex()`	2	1	21	9	33	63
`beginActivityList != null`	0	1	20	13	30	63
`filterPane == null`	0	1	20	13	30	63
`vehicle[i].getLicenceNumber() != null && (!vehicle[i].getLicenceNumber().equals(""))`	1	1	20	13	30	63
`fromDateChooser.getSpinner().isEnabled() && toDateChooser.getSpinner().isEnabled()`	3	1	20	13	30	63
`timemgmtDetailsRequestDO.getTimeRange(). equalsIgnoreCase(FBZWConstants.PERIOD)`	5	1	20	13	30	63
`filterDO.getBeginActivity().equalsIgnoreCase(FBZWConstants.PERIOD)`	10	1	20	13	30	63
`driverPath != null && driverPath.getPath().length <= 2`	5	1	20	13	30	63
`update && selectedDriverIds != null && (timemgmtSumFilterDO != null) && timemgmtSumFilterDO.equals(getFilterSettingDO())`	5	1	20	12	31	63
`ResourceManager.getUserProperties().getLastSelectedTab() == ((JTabbedPane)getParent()).getSelectedIndex()`	1	1	20	12	31	63
`!key.toString().equalsIgnoreCase( FBZWConstants.CACHE_LATEST_TIMEMGMT_DETAILS)`	1	1	21,3	11	30	62,3
`requestInput instanceof TimemgmtDetailsRequestDO`	39	1	21,3	11	30	62,3
`map == null`	0	1	21,3	11	30	62,3
`driverMap != null`	0	1	21,3	11	30	62,3
`_next != null`	1	1	21,3	11	30	62,3
`driverMap != null`	0	1	21,3	11	30	62,3
`!_removed.isEmpty()`	1	1	21,3	11	30	62,3
`!dataList.isEmpty()`	1	1	21,3	11	30	62,3
`requestInput instanceof TimemgmtSumRequestDO`	0	1	21,3	11	30	62,3

Tabelle 6.1: Die 31 höchstgewichteten Empfehlungen mit LOC, Anzahl tangierender Testfälle, Punkte der Fehlerdatenquellen Code, Versionsgeschichte und CodeCover-Überdeckung sowie der Summe der Punkte. Alle empfohlenen Code-Blöcke sind `If`-Blöcke. Die Blöcke mit null Codezeilen sind implizite `Else`-Blöcke.

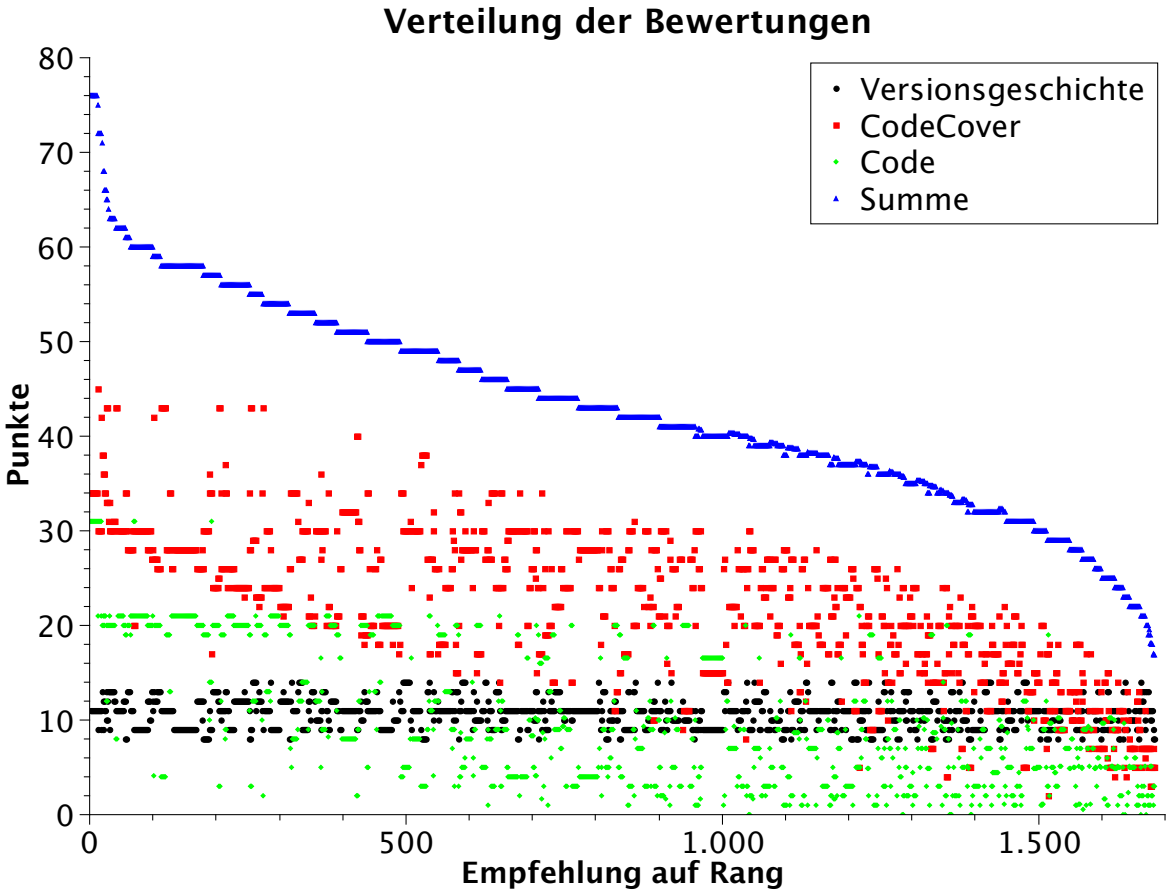

Abbildung 6.2: X/Y-Plot der Verteilung der Gesamtpunkte

Tabelle 6.1 zeigt die 31 bestbewerteten Testfallempfehlungen. Dies entspricht den besten 0,88%. Ausgelassen wurden sehr ähnliche Empfehlungen und solche mit sehr langen Prädikaten. `If`-Blöcke mit null Codezeilen stellen implizite `else`-Blöcke dar, was bedeutet, dass das Prädikat der entsprechenden `If`-Anweisung stets wahr war. Es fällt auf, dass alle Codeblöcke nur von einem Testfall tangiert wurden, was wahrscheinlich den Eigenschaften der Testsuite zuzuschreiben ist.

Ca. 15 der besten 31 Empfehlungen scheinen Prädikate zu enthalten, die auf technische Details prüfen, oder zur defensiven Programmierung gehören. Es dürfte einem Tester, der den Quellcode des Programms nicht kennt, nicht einfach möglich sein, aus den Prädikaten neue Testfälle zu entwickeln, da die Prädikate in den meisten Fällen nicht „sprechend“ sind, also keinen direkten Rückschluss auf ggf. zu ändernde Eingabedaten zulassen.

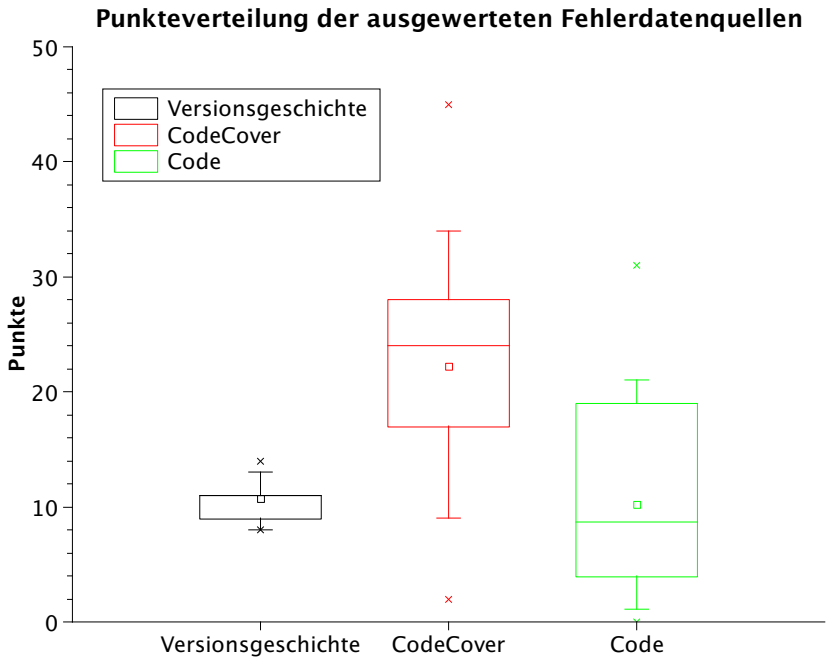

Abbildung 6.3: Verteilung der Bewertungen der Fehlerdatenquellen

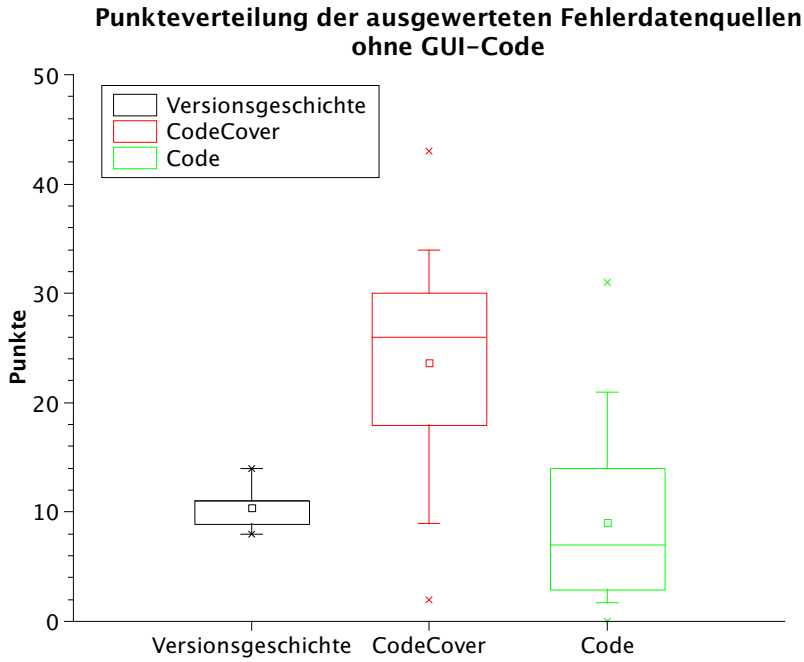

Abbildung 6.4: Verteilung der Bewertungen der Fehlerdatenquellen ohne GUI-Code

6.4 Schlussfolgerungen aus der Erprobung

Die Erprobung konnte nicht wie gewünscht durchgeführt werden. Folgende Probleme haben eine umfassendere Erprobung verhindert:

- Aktuelle Dokumente und Entwickler, die den Quellcode des Zeitwirtschaftsclients kennen, standen nicht zur Verfügung. Daher konnten die meisten Fehlerindikatoren des Modells nicht eingesetzt werden. Die verbleibenden sieben Fehlerindikatoren konnten zwar ausgewertet werden, allerdings kann auf dieser spärlichen Datenmenge das Modell nicht angemessen bewertet werden.
- Die Ergebnisse der Priorisierung konnten keinem Entwickler zur Bewertung vorgelegt werden. So bleibt die Bewertung der Testfallempfehlungen dem Autor der Diplomarbeit überlassen, der den Quellcode nicht kennt und die Qualität der Empfehlungen damit nicht ausreichend beurteilen kann.

Jedoch lassen sich ohne detaillierte Kenntnisse der Anwendung die folgenden allgemeinen Beobachtungen aus den obigen Diagrammen sowie der Liste an Empfehlungen feststellen:

- Die unpriorisierte Liste ist mit 1700 Einträgen zu lang, um sie einzeln durchzuarbeiten. Eine automatisierte Priorisierung ist erforderlich.
- Die Empfehlungen, deren Prädikat den String „`null`“ enthält, scheinen technische Prädikate zu sein. Diese sind größtenteils nicht „sprechend“ und beziehen sich nicht direkt auf durch Testfälle manipulierbare Eingabedaten.
- Beim getesteten Programm liegen die Punktzahlen des Fehlerindikators „Alter der Datei“, der den einzigen Fehlerindikator der Fehlerdatenquelle „Versionsgeschichte“ ausgemacht hat, sehr nah zusammen. Die Werte erscheinen wie zufällige Werte und beeinflussen das Ergebnis nicht. Siehe Abbildungen 6.2 und 6.3.

7 Zusammenfassung

Im Rahmen der Diplomarbeit wurde zunächst in der Literatur nach Möglichkeiten zur Priorisierung von Testfallempfehlungen gesucht. Neben Veröffentlichungen zu CodeCover wurden Veröffentlichungen der Themen risikobasierter Test und Fehlerprognose studiert und geeignete Vertreter zusammengefasst. Aus diesen Veröffentlichungen wurden Thesen extrahiert, die die Grundlage für das im Hauptteil der Arbeit erstellte Modell zur Priorisierung von Testfallempfehlungen bilden.

Zum Modell gehören die Beschreibung einiger Heuristiken zur Fehlerprognose auf Basis des Quellcodes, der Versionsgeschichte, des Wissens der Entwickler, der eingesetzten Qualitätssicherungsmaßnahmen, der während der Entwicklung herrschenden Stressfaktoren sowie den Eigenschaften der Testfallempfehlung selbst. Weiterhin gibt das Modell auch an, wie die Resultate dieser Heuristiken zu einer Gesamtempfehlung verrechnet werden können und beschreibt den Ablauf der Priorisierung. Durch die Anwendung des Priorisierungsmodells ist es einem Tester möglich, sich aus einer großen Menge von Testfallempfehlungen diejenigen auszusuchen, die die heuristisch besten sind, was die Wirtschaftlichkeit seiner Arbeit erhöht.

Danach wurde das Modell als CodeCover-Erweiterung implementiert. Der Eclipse-Integration von CodeCover wurde dabei eine weitere Sicht hinzufügt. Mit Hilfe dieser Sicht können nach einem Testdurchlauf mit CodeCover Testfallempfehlungen generiert und mit Hilfe mehrerer Heuristiken priorisiert werden. Die Implementierung ist dabei als Framework für die Erprobung weiterer Heuristiken zur Testfallpriorisierung ausgelegt. Durch die Framework-Funktionen entfällt die zeitraubende Implementierung von Oberflächen zur Konfiguration von Fehlerindikatoren. Weiter Fehlerdatenquellen, Fehlerindikatoren und automatische Auswerter können mit wenig Zeitaufwand eingebaut werden. Einige Heuristiken wurden mit automatischer Auswertung implementiert, für einige muss der Tester selbst die Daten liefern.

Es ist dem Tester nun möglich, sich nach der Testdurchführung mit CodeCover eine Liste von Empfehlungen anzeigen zu lassen. Diese kann er mit Hilfe der konfigurierbaren Heuristiken dann sortieren lassen. Der Tester bekommt damit eine Auswahl von heuristisch guten Testfallempfehlungen angezeigt. Mit diesen kann er eine Testsuite mit relativ wenig Aufwand verbessern. Diese Wirtschaftlichkeit könnte es ermöglichen, eine Testsuite trotz engem Zeitplan zu verbessern.

Die Erprobung beim Industriepartner konnte eingeschränkt durchgeführt werden. Es kam dabei nur eine kleine Auswahl der Heuristiken im Modell zum Einsatz. Die Bewertung der höchstpriorisierten Testfallempfehlungen war nicht möglich. Dennoch hat die Erprobung

gezeigt, dass eine Priorisierung mit Hilfe dieses Verfahrens möglich ist und differenzierte Ergebnisse liefert.

Literaturverzeichnis

[Amloo] S. Amland. Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study. *Journal of Systems and Software*, 53(3):287 – 295, 2000. doi:DOI:10.1016/S0164-1212(00)00019-4. URL http://www.sciencedirect.com/science/article/pii/S0164121200000194. (Zitiert auf den Seiten 15, 27, 35 und 36)

[Bac99] J. Bach. Heuristic Risk-Based Testing. 1999. (Zitiert auf den Seiten 17 und 36)

[BOW06] R. M. Bell, T. J. Ostrand, E. J. Weyuker. Looking for bugs in all the right places. pp. 61–72, 2006. doi:http://doi.acm.org/10.1145/1146238.1146246. URL http://doi.acm.org/10.1145/1146238.1146246. (Zitiert auf den Seiten 19, 20, 25, 26, 35 und 36)

[Cod11] CodeCover. *CodeCover-Homepage*. 2011. URL http://codecover.org/. [Online; Stand 28. Juni 2011]. (Zitiert auf den Seiten 29 und 30)

[DLLS05] M. Deininger, H. Lichter, J. Ludewig, K. Schneider. *Studien-Arbeiten – ein Leitfaden zur Vorbereitung, Durchführung und Betreuung von Studien-, Diplom- und Doktorarbeiten am Beispiel Informatik*. 2005. (Zitiert auf Seite 12)

[FAI97] T. Furuyama, Y. Arai, K. Iio. *Analysis of fault generation caused by stress during software development*, volume 38. 1997. doi:DOI:10.1016/S0164-1212(97)00064-2. URL http://www.sciencedirect.com/science/article/pii/S0164121297000642. Achieving Quality in Software. (Zitiert auf den Seiten 25 und 27)

[FOoo] N. Fenton, N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *Software Engineering, IEEE Transactions on*, 26(8):797 –814, 2000. doi:10.1109/32.879815. (Zitiert auf den Seiten 9, 18, 19, 20 und 27)

[HFGO94] M. Hutchins, H. Foster, T. Goradia, T. Ostrand. *Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria*. ICSE '94. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994. URL http://portal.acm.org/citation.cfm?id=257734.257766. (Zitiert auf Seite 27)

[HP04] D. Hovemeyer, W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39:92–106, 2004. doi:http://doi.acm.org/10.1145/1052883.1052895. URL http://doi.acm.org/10.1145/1052883.1052895. (Zitiert auf den Seiten 21 und 26)

[HPH+09] T. Holschuh, M. Pauser, K. Herzig, T. Zimmermann, R. Premraj, A. Zeller. Predicting defects in SAP Java code: An experience report. pp. 172 –181, 2009. doi:10.1109/ICSE-COMPANION.2009.5070975. (Zitiert auf den Seiten 25, 26, 27 und 35)

[Kim03] Y. W. Kim. Efficient use of code coverage in large-scale software development. pp. 145–155, 2003. URL http://portal.acm.org/citation.cfm?id=961322.961347. (Zitiert auf Seite 20)

[LL07] J. Ludewig, H. Lichter. *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, 2007. (Zitiert auf den Seiten 12, 25, 27 und 38)

[Moz11] Mozilla. *Website der Mozilla Corporation*. 2011. URL http://www.mozilla.com/. [Online; Stand 28. Juni 2011]. (Zitiert auf Seite 20)

[Mye01] G. Myers. *Methodisches Testen von Programmen*. Oldenbourg, 2001. URL http://books.google.com/books?id=srqkL75liPkC. (Zitiert auf Seite 36)

[NBZ06] N. Nagappan, T. Ball, A. Zeller. Mining metrics to predict component failures. pp. 452–461, 2006. doi:http://doi.acm.org/10.1145/1134285.1134349. URL http://doi.acm.org/10.1145/1134285.1134349. (Zitiert auf den Seiten 20, 25, 26, 27, 34 und 39)

[Neu05] D. Neun. Codemetriken zur Bewertung und Prognose der Fehlerhäufigkeit. 2005. URL http://elib.uni-stuttgart.de/opus/volltexte/2005/2399. (Zitiert auf den Seiten 35 und 36)

[NZZ07] S. Neuhaus, T. Zimmermann, A. Zeller. Predicting Vulnerable Software Components. 2007. (Zitiert auf den Seiten 20 und 26)

[OA96] N. Ohlsson, H. Alberg. Predicting Fault-Prone Software Modules in Telephone Switches. *IEEE Transactions on Software Engineering*, 22:886–894, 1996. doi:http://doi.ieeecomputersociety.org/10.1109/32.553637. (Zitiert auf Seite 19)

[Ora11] Oracle. *JMX Technology Home Page*. 2011. URL http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html. [Online; Stand 29. Juni 2011]. (Zitiert auf Seite 29)

[OWB05] T. J. Ostrand, E. J. Weyuker, R. M. Bell. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering*, 31:340–355, 2005. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2005.49. (Zitiert auf den Seiten 20, 25, 26, 27, 35 und 36)

[RAF04] N. Rutar, C. B. Almazan, J. S. Foster. A Comparison of Bug Finding Tools for Java. pp. 245–256, 2004. doi:10.1109/ISSRE.2004.1. URL http://portal.acm.org/citation.cfm?id=1032654.1033833. (Zitiert auf den Seiten 22, 26 und 35)

[Rat11] I. Rational. *Functional Tester Homepage*. 2011. URL http://www-01.ibm.com/software/awdtools/tester/functional/. [Online; Stand 5. August 2011]. (Zitiert auf Seite 53)

[RHRH02] G. Rothermel, M. J. Harrold, J. von Ronne, C. Hong. *Empirical studies of test-suite reduction*, volume 12. John Wiley & Sons, Ltd., 2002. doi:10.1002/stvr.256. URL http://dx.doi.org/10.1002/stvr.256. (Zitiert auf Seite 25)

[Sch08] R. Schmidberger. *Glassboxtest zur Testsuite-Optimierung*. 2008. (Zitiert auf den Seiten 23 und 29)

[Sch09] S. Schumm. *Praxistaugliche Unterstützung beim selektiven Regressionstest*. 2009. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2923&engl=0. (Zitiert auf Seite 23)

[Sch10] R. Schmidberger. *Ein kombinierter Black-Box- und Glass-Box-Test*. 2010. (Zitiert auf den Seiten 12, 23, 26, 30, 38 und 40)

[Sub11a] Subclipse. *Subclipse-Projekt-Homepage*. 2011. URL http://subclipse.tigris.org/. [Online; Stand 28. Juni 2011]. (Zitiert auf Seite 48)

[Sub11b] Subversion. *Subversion-Projekt-Homepage*. 2011. URL http://subversion.apache.org/. [Online; Stand 28. Juni 2011]. (Zitiert auf Seite 48)

[SZZ05] J. Sliwerski, T. Zimmermann, A. Zeller. Don't Program on Fridays! How to Locate Fix-Inducing Changes. 2005. (Zitiert auf den Seiten 21 und 36)

[SZZ06] A. Schröter, T. Zimmermann, A. Zeller. Predicting component failures at design time. pp. 18–27, 2006. doi:http://doi.acm.org/10.1145/1159733.1159739. URL http://doi.acm.org/10.1145/1159733.1159739. (Zitiert auf Seite 26)

[Wik10] Wikipedia. *Header-Datei — Wikipedia, Die freie Enzyklopädie*. 2010. URL http://de.wikipedia.org/w/index.php?title=Header-Datei&oldid=82784591. [Online; Stand 28. Juni 2011]. (Zitiert auf Seite 20)

Alle URLs wurden zuletzt am 15.08.2011 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Ralf Ebert)