Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Entwicklung einer Heuristik für den Testbedarf von Open Source Softwareprojekten auf einer Social Coding Site

Masterarbeit

im Studiengang Informatik

von

André Schnabel

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Jun.-Prof. Dr. Robert Jäschke
Betreuer: Raphael Pham

Hannover, 06.05.2013

Zusammenfassung

GitHub und andere Social Coding Sites beherbergen aufgrund ihrer Beliebtheit viele Projekte. Beteiligung an fremden Projekten kann durch Beitragen von Tests geschehen. Die Auswahl des bedürftigsten Projekts hierfür ist aufwendig. Hierbei soll die in dieser Arbeit entwickelte Heuristik behilflich sein.

Eine Analyse der technischen und sozialen Aspekte des Testens auf GitHub wurde als Vorbereitung durchgeführt. Eine an Java-Projekten vorgenommene Messung zeigt, dass nur wenige der erfassten Projekte überhaupt Tests enthalten. Nur eine Minderheit der an den Projekten beteiligten Entwickler steuern Tests bei.

Mithilfe von Überlegungen zum Testbedarf und einer Literaturrecherche von Fehlerzahlmodellen wurden mögliche Einflussfaktoren für die Heuristik bestimmt.

Zwei Umfragen mit GitHub-Nutzern wurden durchgeführt und daraus ein Benchmark abgeleitet. Das Benchmark erlaubt die Bewertung von Vorhersagemodellen für die intuitive Einschätzung des betriebenen Testaufwands und der verbleibenden Fehlerzahl.

Zur Unterstützung der Arbeit wurde ein Werkzeug zur Messung, Analyse und Visualisierung auf GitHub implementiert.

Gestützt auf die gesammelten Ergebnisse wurden jeweils ein grobes Maß für den Testaufwand und die Fehlerzahl vorgestellt. Die Kombination dieser Metriken erlaubt die Berechnung eines einzigen Werts als Schätzwert für den relativen Testbedarf eines Open Source Softwareprojekts auf einer Social Coding Site.

Abstract

Due to their popularity GitHub and other Social Coding Sites host many projects. Participation in external projects can happen through contribution of tests. Selecting the project with the biggest need for tests is a tedious task. The heuristic developed in this thesis is supposed to help selecting a project for testing.

An analysis of technical and social testing aspects on GitHub was done in preparation. A measurement of Java projects shows, that only a few of the examined projects contain tests. Only a minority of the contributors to those projects are working on tests.

Aided by thoughts on testing need and a literature review of defect models possible factors of influence have been determined.

Two surveys of GitHub users were conducted. A benchmark for predicting the intuitive estimation of test effort and bug count was derived from them.

To support this thesis a tool for measurement, analysis and visualization on GitHub was implemented.

Based upon the collected results metrics for test effort and bug count were presented. Combining those metrics allows the computation of a single value as estimate for the relative testing need of an open source software project on a social coding site.

Abkürzungsverzeichnis

Anzahl
API Application Programming Interface (Programmierschnittstelle)
BLOB Binary Large Object
BDD Behavior Driven Development
CD Compact Disc
CI Continouous Integration (Kontinuierliche Integration)
cloc Count Lines Of Code (Perl-Skript für LOC)
CPD Copy Paste Detector (Erkennung duplizierten Codes)
CSV Comma Separated Values*
DVCS Distributed Version Control System (Verteiltes Versionsverwaltungssystem)*
gdw. genau dann, wenn
Git Verteiltes Versionsverwaltungssystem* (Keine Abkürzung)
GPL GNU General Public License (Lizens für freie Software)
GQM Goal Question Metric
GUI Graphical User Interface
HTML Hypertext Markup Language
IEEE Institute of Electrical and Electronics Engineers
ISO International Organization for Standardization
IEC International Electrotechnical Commission
JRE Java Runtime Environment (Laufzeitumgebung von Java)
JSON JavaScript Object Notation
JVM Java Virtual Machine (Teil des JRE)
KSLOC 10^3 SLOC
LOC Lines Of code* (Anzahl Codezeilen)
MSR Mining Software Repositories
NCSS Non Commented Source Statements (hier synonym zu LOC verwendet)
NP Nichtdeterministische Polynomialzeit (Komplexitätsklasse)
OSS Open Source Software
PDF Portable Document Format (Dateiformat für Dokumente)
PNG Portable Network Graphics (Format für Rastergrafik)
SCS Social Coding Site*
SW Software
SLOC Single Lines Of Code (Synonym für LOC)
SWQ Softwarequalität
SRGM Software Reliability Growth Model
TDD Test Driven Development*
TLOC Test Lines Of Code* (Anzahl Testcodezeilen)
URI Uniform Resource Identifier (Ressourcen-Identifikator)
URL Uniform Resource Locator (URI mit zusätzlicher Lokalisierung)
VCS Version Control System (Versionsverwaltungssystem)
XML eXtensible Markup Language* (Textuelle Auszeichnungssprache)

*mit Definition oder Erklärung im Einführungskapitel 2.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Das Fachgebiet Software Engineering an der Leibniz Universität Hannover möchte einen Web-Service zum Crowdsourcing von Testbemühungen entwickeln. Der Web-Service soll ein komfortables und schnelles Beitragen von Tests für bestehende Projekte auf GitHub[1] erlauben. Weiterhin soll der Service die zum Testen vorgeschlagene Projekte in einer dem Testbedarf[2] entsprechenden Reihenfolge anzeigen. Dadurch soll die Effektivität der Testbemühungen maximiert werden. Die Bezeichnung für diesen Service lautet *TestHub* in Anlehnung an die Social Coding Site GitHub.

Die Tests sollen mithilfe von *TestHub* im Browser mit möglichst geringem Aufwand erstellt werden können. Die Hürde für freiwillig arbeitende Testersteller soll dadurch auf ein Minimum reduziert werden. Als Anreiz ist ein Belohnungsmechanismus mit Punkten für Tests vorgesehen. Der Projektinhaber kann diese Punkte nach Überprüfung der Tests bestätigen. Dadurch soll sichergestellt werden, dass die Tests augenscheinlich korrekt sind und einen zusätzlichen Nutzen für die Qualitätssicherung im betrachteten Projekt bringen.

Die Erstellung von *TestHub* steht vor dem Hintergrund von Forschungen zur Testkultur auf Social Coding Sites in [Pham et al. '13] sowie [Pham et al. '13a].

Wichtige Bestandteile von *TestHub* sind:

- Website zum Ranking der Projekte. Dies wird in [Stoliar '13] behandelt.
- Heuristik zur automatisierten Ermittlung des Testbedarfs. Thema dieser Arbeit.
- Infrastruktur zum Beitragen und Verifizieren von neuen Testfällen in einem Web-Service.

1.2 Problemstellung

Das Hauptziel dieser Arbeit ist die wissenschaftlich begründete Erstellung einer Heuristik für den Testbedarf von Open Source Softwareprojekten auf einer allgemeinen Social Coding Site. Dies beinhaltet eine Literaturrecherche zu Testbedarf und Testverfahren. Weiterhin soll das Testverhalten von öffentlichen Projekten auf GitHub analysiert werden. Eine konkrete Umsetzung der Heuristik soll für die Social Coding Site

[1] `https://github.com/`

[2] Anzahl der noch im Programm vorhandenen Fehler. Eine ausführliche Definition folgt in Kapitel 2.

GitHub erfolgen. Diese Umsetzung soll auch für die Evaluation der Heuristik an einer großen Anzahl von Projekten verwendet werden. Zusätzlich wird die Erstellung eines Analyse- und Anzeigetools gefordert.

1.2.1 Anforderungen an die Heuristik

Die Heuristik bekommt als *Eingabe* eine Menge von Projekten auf einer Social Coding Site. Als *Ausgabe* soll die Heuristik eine strikte Ordnung auf der gegebenen Projektmenge liefern. Diese Ordnung soll die Projekte absteigend bezüglich ihres geschätzten Testbedarfs sortieren. Dadurch wäre ein Testbedarf-Ranking der Projekte für *TestHub* gegeben.

Die Heuristik soll möglichst unabhängig von einer konkreten technologischen Umsetzung beschrieben werden. Das beinhaltet Abstraktion der in den Projekten verwendeten Technologien (zum Beispiel der Programmiersprache). Analog sollte die Heuristik keine Besonderheiten von GitHub beinhalten, um eine Übertragung auf andere Social Coding Sites wie beispielsweise Bitbucket[3] zu ermöglichen.

Weiterhin muss sich zeigen lassen, dass die Heuristik genau den Testbedarf misst und ihre Bestandteile und deren mathematische Beziehung zueinander aus Messungen und Literaturergebnissen hergeleitet worden sind.[4]

1.2.2 Anforderungen an die Umsetzung

Die Umsetzung der Heuristik muss einige Eigenschaften besitzen, um für den praktischen Einsatz in *TestHub* tauglich zu sein.

Es muss eine obere Schranke für die durchschnittliche Laufzeit[5] der Heuristik bezogen auf alle Projekte geben.

Wenige fehlerhafte Einschätzungen können akzeptiert werden, da nicht etwa eine exakte Metrik sondern eine Heuristik entwickelt werden soll. Auf der anderen Seite kann jeder Fehler im Ranking Nutzer von *TestHub* frustrieren. Deshalb wäre es wünschenswert, wenn die Restfehlerwahrscheinlichkeit so klein wie möglich gehalten wird.

Weiterhin wird in der Exploration gezeigt, dass die Daten auf GitHub sehr inhomogen sind und eine hohe Varianz aufweisen. Daher muss die Umsetzung der Heuristik auch mit ungewöhnlichen und fehlerhaften Eingaben umgehen können (Robustheit). Dies impliziert die Notwendigkeit einer umfangreichen Test Suite aus sehr unterschiedlichen Projekten zur Überprüfung der Umsetzung der Heuristik.

[3] `https://bitbucket.org/`

[4] Vergleiche hierzu den Goal Question Metric Ansatz von [Basili et al. '94]. Die Heuristik soll vom Konzept des Testbedarfs ausgehend "top-down" entwickelt werden. Nicht jedoch von willkürlich gewählten einfach zu messenden Größen. Ein Versuch der Anwendung des GQM-Ansatzes auf die Problemstellung dieser Arbeit wird in Kapitel 6 dokumentiert.

[5] Das ist also eine "weiche" Echtzeitanforderung.

1.3 Gliederung

Im 2. Kapitel werden zunächst einige später benötigte Begriffe aus den Bereichen Softwarequalität, Softwaretechnik und Verteilte Softwareentwicklung definiert. Danach werden im 3. Kapitel einige verwandte Arbeiten zu den Themen Fehlervorhersage, Mining Software Repositories und Testen vorgestellt. Darauf werden in Kapitel 4 die für die Heuristik potenziell nutzbaren Datenquellen aufgelistet. Im 5. Kapitel wird eine Analyse der Ausgangssituation des Testens auf GitHub vorgenommen. In Kapitel 6 wird der Versuch unternommen, den GQM-Ansatz zur Bestimmung von Indikatoren für den Testbedarf zu benutzen. Danach wird in Kapitel 7 das Konzept des Testbedarfs näher betrachtet. In Kapitel 8 werden frühe Vorüberlegungen zu möglichen Metriken mit Bezug auf den Testbedarf dokumentiert. Dann hat das 9. Kapitel Modelle zur Vorhersage von Fehlern und die Schätzung der Fehlerzahl in Softwareprojekten zum Thema. Im 10. Kapitel geht es um Überlegungen zur statischen Abschätzung der Testabdeckung. Anschließend folgt im 11. Kapitel die Beschreibung des im Rahmen dieser Arbeit entwickelten Mess-, Analyse-, Visualisierungs- und Beurteilungswerkzeugs *JProjectInspector*. Darauf werden im 12. Kapitel die zwei auf GitHub durchgeführten Umfragen zusammen mit der Auswertung ihrer Ergebnisse dokumentiert. Das 13. Kapitel ist *das zentrale Kapitel* dieser Arbeit. Dort werden die Resultate aus Kapitel 7 und 12 mithilfe des in Kapitel 11 beschriebenen Werkzeugs zu einer Testbedarfsheuristik zusammengeführt. Abgeschlossen wird diese Arbeit im 14. Kapitel durch eine kritische Würdigung, einen Ausblick auf mögliche anknüpfende Arbeiten und eine rückblickende Zusammenfassung.

2 Grundkonzepte

Es folgen Definitionen von zentralen Begriffen der nachfolgenden Kapitel.

2.1 Softwarequalität

Definition 2.1.1 (Testen)**.** Testen ist das Ausführen eines Programms mit dem Ziel Fehler zu finden (Nach [Myers '04]).

Definition 2.1.2 (Fehler)**.** Ein Fehler ist eine Abweichung von Spezifikation (Soll) und Programmverhalten (Ist).

Definition 2.1.3 (Korrektheit)**.** Korrektheit ist das Ausmaß, in dem SW ihre Spezifikation erfüllt (Nach [Schneider '07]).

Definition 2.1.4 (Testebenen)**.** Tests lassen sich nach der Granularität der getesteten SW-Einheit in folgende Testlevels einteilen:

- **Modultests:** Testet Modul in Isolation.
- **Integrationstest:** Testet Zusammenspiel von zwei oder mehreren Modulen.
- **Systemtest:** Testet System in Gesamtheit. Kann Abnahmetest sein.

In dieser Arbeit spielen nur automatisierte Tests eine Rolle. Dafür werden oft Frameworks für Modultests verwendet.

Definition 2.1.5 (Testkosten)**.** Die Testkosten sind definiert als die Anzahl der in Testbemühungen investierten Personenstunden.

Die Testkosten werden nachfolgend auch Fehlerbehebungskosten genannt.

Definition 2.1.6 (Testeffektivität)**.** Die Testeffektivität ist die Anzahl der durch Tests gefundenen Fehler.

Definition 2.1.7 (Testeffizienz)**.** Testeffizienz $=_{\text{Def.}} \frac{\text{Testeffektivität}}{\text{Testkosten}}$.

Definition 2.1.8 (Testbedarf)**.** Testbedarf $=_{\text{Def.}} \frac{\text{Fehlerzahl}}{\text{Testaufwand}}$.

Definition 2.1.9 (Testbarkeit)**.** Ausmaß, in dem ein System das Erstellen von Testbedingungen erleichtert, sowie die Durchführung der Tests, ob die Bedingungen erfüllt sind (nach [Schneider '07]).

Definition 2.1.10 (Testgetriebene Entwicklung)**.** Als testgetriebene Entwicklung wird eine Methode der agilen Softwareentwicklung bezeichnet. Es wird für jede neue Funktionalität zuerst der Testcode und erst danach der zugehörige Produktivcode geschrieben. Kennzeichnend ist eine Feedback-Schleife: Testfälle für neue Features müssen zuerst scheitern (roter Balken[1]). Produktivcode wird nur geschrieben, bis er dem Test genügt (grüner Balken). Dann wird der Testfall verfeinert, wodurch er wieder scheitert. Dieser Zyklus wird abgebrochen, wenn im Testfall alle an die getestete Code-Einheit gestellten Anforderungen enthalten sind.

2.2 Softwaretechnik

Definition 2.2.1 (Refaktorisierung)**.** Refaktorisierung bezeichnet Veränderung der Codestruktur ohne Beeinflussung des Programmverhaltens. Ziel ist die Verbesserung von Merkmalen der Softwarequalität (insbesondere Wartbarkeit und Flexibilität).

Da Anforderungen sich im Laufe eines Projekts ändern, unterliegt längere Zeit unangetasteter Code einem Verfallsprozess. Dem kann Refaktorisierung entgegenwirken. Refaktorisierung erfordert hohe Testabdeckung. Veränderungen können bisher korrekten Code fehlerhaft machen (Regressionen). Bei geringer Testabdeckung könnten diese Fehler übersehen werden.

Definition 2.2.2 (Software-Metrik)**.** Eine Software-Metrik ist eine Abbildung von einem Aspekt einer SW-Einheit auf einen Zahlenwert.

Oftmals sind Ergebnisse einer Metrik in bestimmten Intervallen (Kategorien) mit einer Bewertung zu interpretieren. Veränderung dieser Maßzahlen können zur Fortschrittsmessung oder zur Prozessbewertung (handlungsleitend) genutzt werden. Dies spielt bei GQM zur Kontrolle (control) und Verbesserung (improve) von Qualitätsaspekten eine Rolle.

Definition 2.2.3 (Goal Question Metrik)**.** Der GQM-Ansatz ist ein Paradigma zur Bestimmung von Metriken für Ziele. Entwickelt in [Basili et al. ’94]. Fordert nicht zu messen, was einfach zu messen ist, sondern was wichtig für Ziele ist. Ausgehend von Zielen werden Fragestellungen entwickelt und dafür passende Metriken gewählt.

Definition 2.2.4 (Heuristik)**.** Eine Heuristik ist definiert als Regel, die eine Vorgehensweise vorgibt. Sie ist begründet und bewährt. Ihr Erfolg ist aber nicht garantiert (Sinngemäß nach [Schneider ’07]).

Definition 2.2.5 (Testabdeckung)**.** Die Testabdeckung ist der Anteil des Produktivcodes, welcher von automatisierten Tests ausgeführt und somit geprüft wird.

Testabdeckung wird auch Coverage genannt.

[1] Integrierte Entwicklungsumgebungen visualisieren das Ergebnis der Ausführung eines Modultests über einen gefärbten Balken. Er ist *rot*, gdw. der ausgeführte Test gescheitert ist. Ansonsten ist er *grün*.

Definition 2.2.6 (Kohäsion)**.** Die Kohäsion bezeichnet das Ausmaß für Zusammengehörigkeit der Methoden eines Moduls.

Definition 2.2.7 (Kopplung)**.** Kopplung ist das Ausmaß der Abhängigkeit eines Moduls von anderen Modulen.

Definition 2.2.8 (Lines Of Code (LOC))**.** LOC ist definiert als die Anzahl der tatsächlichen Codezeilen in einem Modul. Kommentare oder Whitespace (Leerraum) werden nicht mitgezählt.

Man beachte, dass hier Kommentare bei der Zählung der Codezeilen ignoriert werden. In [Hoffmann '08] wird diese Art der Zählung NCSS (Non Commented Source Statements) genannt. Hingegen werden dort bei LOC auch die Kommentarzeilen mitgezählt.

Definition 2.2.9 (Zyklomatische Komplexität[2])**.** Die zyklomatische Zahl $v(G)$ eines Graphen G mit n Knoten, e Kanten und p Zusammenhangskomponenten ist definiert als $v(G) = e - n + 2p$.

Auf Code angewandt wird für G der Kontrollflussgraph einer Methode eingesetzt. Die Zahl $v(G)$ ist dann ein Maß für den Grad der Verzweigung des Kontrollflusses einer Methode. In der Literatur wird sie häufig auch nach ihrem Entwickler als McCabe-Metrik bezeichnet.

Definition 2.2.10 (Web Scraping)**.** Mit Web Scraping bezeichnet man das Extrahieren von Informationen aus einer Webseite.

Scraping kann bei XHTML-Dokumenten mithilfe von XML-Parsern in Kombination mit XPath-Ausdrücken geschehen. Bei den deutlich häufigeren HTML-Dokumenten werden reguläre Ausdrücke benutzt.

Definition 2.2.11 (Regulärer Ausdruck)**.** Ein regulärer Ausdruck ist eine Notation einer regulären Sprache in Form einer Zeichenkette.

2.3 Verteilte Entwicklung

Definition 2.3.1 (Versionsverwaltungssystem (VCS))**.** Ein Versionsverwaltungssystem archiviert den vollständigen Zustand der Artefakte eines Projekts zu bestimmten Zeitpunkten. Diese Zustände werden Revisionen genannt. Revisionen können mit Nachrichten annotiert werden.

Die Gesamtheit der Daten einer Versionsverwaltung zu einem Projekt werden als Repository dieses Projekts bezeichnet. Da im weiteren Verlauf dieser Arbeit alle betrachteten Projekte mit VCS verwaltet werden, wird der Begriff Repository synonym zu Projekt verwendet.

[2] Entnommen aus [McCabe '76].

Definition 2.3.2 (Verteiltes Versionsverwaltungssystem (DVCS))**.** Ein Versionsverwaltungssystem ohne ein zentrales Repository wird verteilt genannt.

In einem DVCS besitzt jeder Beitragende ein eigenes gleichwertiges Repository. Die Repositories können beliebig miteinander abgeglichen werden (push/pull). Fremde Repositories können geklont werden. Dadurch entsteht eine identische lokale Kopie.

Bekannte Vertreter von DVCS sind aktuell Bazaar, BitKeeper, Git und Mercurial.

Definition 2.3.3 (Git)**.** Git ist ein verteiltes Versionsverwaltungssystem, welches besonders auf hohe Effizienz und Sicherheit ausgelegt ist.

Definition 2.3.4 (Social Coding Site)**.** Eine Social Coding Site ist ein Web-Dienst, der anbietet Software-Projekte zu beherbergen. Das beinhaltet Versionsverwaltung, Issue-Tracker, Wikis und Diskussionsforen.

SCSs vereinfachen die Koordination und Kommunikation der Mitentwickler an den gehosteten Projekten.

Definition 2.3.5 (Commit)**.** Änderungen an aktueller Revision zu einer neuen Revision zusammenfassen. Meist mit Nachricht annotiert.

Definition 2.3.6 (Merge)**.** Integration zweier unabhängiger Repositories.

Definition 2.3.7 (Branch)**.** Entwicklungsstrang.

Definition 2.3.8 (Fork)**.** Abspaltung eines Repositories.

Definition 2.3.9 (Revision)**.** Schnappschuss des Zustands aller Artefakte eines Projekts zu bestimmten Zeitpunkt.

Definition 2.3.10 (Push/Pull)**.** Verschieben von Änderungen zwischen Repositories in einem DVCS.

Definition 2.3.11 (Pull-Request)**.** Verbund von Änderungen, welche vorgeschlagen werden in ein Repository zu übernehmen.

Definition 2.3.12 (Clone)**.** Erstellung einer exakten Kopie eines entfernten Repositories.

Definition 2.3.13 (GitHub)**.** Sehr populäre[3] Social Coding Site, welche als Versionsverwaltungssystem Git benutzt.

[3] Momentan sind es 3,4 Millionen Nutzer und 6,2 Millionen Repositories laut `https://github.com/about/press` (Stand: 19.04.2013).

3 Verwandte Arbeiten

Ein Teil der Problemstellung dieser Arbeit ist eine Literaturrecherche zum Testbedarf. Da Testbedarf hier in Fehlerzahl und Testaufwand zerlegt wird, wurden auch Publikationen zu diesen Themen recherchiert.

3.1 Modelle für Fehler

Die *Fehlervorhersage* und *Abschätzung der Fehlerzahl* sind seit Jahrzehnten beliebte Ziele der Forschung in der Softwaretechnik. Daher gibt es hierzu sehr viel Literatur.

Die gefundenen Quellen werden nun kurz aufgelistet. In Kapitel 9 werden sie ausführlicher betrachtet.

- Das Fehlerzahl-Maß B von Halstead's Software Scienes wird in [Halstead '77], [Ottenstein et al. '76] und [Mancoridis '09] erläutert.
- Kritische Betrachtungen der SW-Sciences wurden in [Fitzsimmons und Love '78], [Hamer und Frewin '82] und [Stetter '86] dokumentiert.
- Eine Kritik an der Codezeilen-Metrik (LOC) gibt es in [Rosenberg '97].
- Stoppregeln sind Thema von [Chao und Yang '93], [Dalal und Mallows '88].
- Um Einführung von Fehlern und Testeffektivität geht es in [Nikora und Munson '98].
- Weitere Modelle für die Fehlerzahl und Fehlerzahldichte werden in [Li et al. '98], [Gaffney '84] und [Lipow '82] vorgestellt.
- Erkennung von fehlerhaften Modulen mithilfe der Historie wird in [Hata '12] erläutert.
- Fehlervorhersage feingranulare Historien [Hata et al. '12].
- Suchbasierte Fehelrzahldaten [Afzal et al. '09].
- Fehlerort und Fehlerzahl ist Thema von [Ostrand et al. '05].
- Der Ansatz der Vorhersage der Fehlerzahl mithilfe eines generalisierten stochastischen Petri-Netz Modell wird in [Mizuno und Kusumoto '97] verfolgt.
- Eine systematisches Review der Fehlervorhersagemodelle wurde von [Hall et al. '11] durchgeführt.
- Kopplungsbasierte Fehlervorhersage ist Thema von [Ahsan und Wotawa '11].

- Vorhersage neuer Fehler auf Basis bisher gefundener Fehler wird in [Joshi et al. '07] beschrieben.
- Der relative Code-Churn wird [Nagappan und Ball '05] als Indikator für Fehlerdichte diskutiert.
- Auf die Notwendigkeit Metriken von naiven Modellen auf kausale erklärende Modelle zu erheben wird in [Fenton und Neil '00] hingewiesen.
- Sehr einfache Zuverlässigkeitsmetriken für OSS werden von [Tiwari und Pandey '12] vorgeschlagen.
- Querschnittliche Belange als Fehlerquelle sind Thema von [Eaddy und Zimmermann '08].
- Ein Vergleich von Software-Vorhersagetechniken wird von [Shepperd und Kadoda '01] mithilfe von Simulation vorgenommen.
- Ein stochastischer Ansatz zur Ermittlung der Fehlerzahl mithilfe von Martingale lässt sich in [Yip und Xi '99] finden.

3.2 Mining Software Repositories

Der zu entwickelnden Heuristik steht für jedes Projekt als Datenquelle ein Repository zur Verfügung. Daher ist auch Literatur aus dem Bereich *Mining Software Repositories* (MSR) von Relevanz. Hier war von besonderem Interesse [Bird et al. '09]. Darin wird vor den Nachteilen der Fehlinterpretation von Daten aus verteilten Versionsverwaltungssystemen gewarnt. Denn verteilte Versionierungssysteme wie Git erlauben das umsortieren, löschen oder modifizieren von Commits beim Verschieben zwischen Repositories. Dies liefert eine Bedrohung der Gültigkeit von Folgerungen, welche von diesen Daten abgeleitet werden.

3.3 Testen

3.3.1 Testabdeckung

Bei der Einschätzung der Effektivität des Testens spielt die Testabdeckung eine wichtige Rolle. Wir werden später aber sehen, dass diese dynamisch nicht automatisiert messbar ist. Ein Lösungsansatz zur *statische Annäherung der Testabdeckung* mithilfe von Code-Splicing wird von [Alves und Visser '09] vorgestellt. (siehe dazu Kapitel 10)

3.3.2 Testkultur

Auf den sozialen Aspekt von Social Coding Sites wurde bereits in [Thung et al. '13] näher eingegangen. Speziell um die Testkultur auf Social Coding Sites ging es in [Kochhar et al. '13], [Pham et al. '13] sowie [Pham et al. '13a]. (mehr in Kapitel 5).

3.3.3 Testbedarf

Konkret zum Thema “Testbedarf von Projekten auf Social Coding Sites” ließ sich zum Zeitpunkt der Erstellung dieser Arbeit keine Literatur finden.

3.4 Werkzeug

Ähnlichkeit zu dem im Rahmen dieser Arbeit entwickelten Werkzeug *JProjectInspector* besitzen die Tools *rEvolution*[1], *msr-tools*[2] sowie `GlueTheos`[3]. Von diesen Programmen war jedoch keines hinreichend geeignet, da sie nur auf Daten des Repositories arbeiten. Die zusätzlichen Daten einer Social Coding Site werden von ihnen nicht beachtet.

[1] `https://github.com/mauricioaniche/rEvolution`
[2] `https://github.com/kirnosenko/msr-tools`
[3] Siehe [Robles et al. ’04].

4 Datenquellen

In diesem Kapitel folgt nun eine Aufzählung der für die Heuristik potenziell zur Verfügung stehenden Datenquellen. Dabei ist zu beachten, dass konkrete Projekte im Regelfall nur eine Teilmenge dieser Daten tatsächlich bereitstellen. Im Extremfall kann ein Projekt auf einer SCS auch leer sein. Dann bietet es keine Daten an.

4.1 Repository

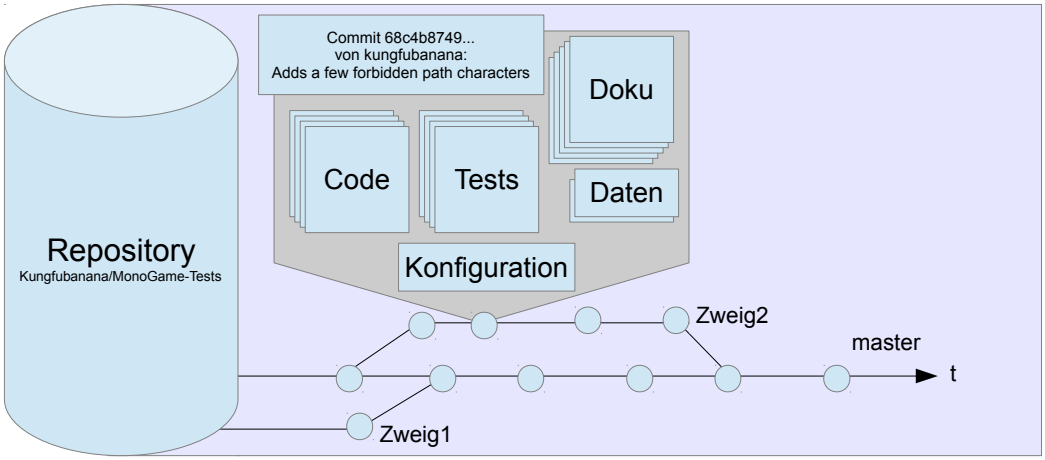


Abbildung 4.1: Inhalte eines Git-Repositories

Die Artefakte eines Projekts auf einer Social Coding Site werden über ein Versionsverwaltungssystem verwaltet. Aktuell sind besonders verteilte Versionsverwaltungssysteme populär. Das Repository einer verteilten Versionsverwaltung beinhaltet den aktuellen Stand des Projekts mit gesamtem Verzeichnisbaum sowie dessen Historie in Form von Revisionen. Eine Revision ist ein Schnappschuss des gesamten Zustands. Das beinhaltet Produktivcode (auch Domänencode genannt), evtl. Testcode und Dokumentation sowie weitere Artefakte.

Dabei muss jedoch darauf geachtet werden, dass die Historie modifiziert werden kann. Mehr zu der Problematik der Fehlinterpretation von Daten eines verteilten Versionsverwaltungssystem lässt sich in [Bird et al. ’09] nachlesen.

4.2 Tracker

Auf einer Social Coding Site gibt es zusätzlich zu Repositories einer Versionsverwaltung noch Tracker[1] mit Diskussionsmöglichkeit. Diese verwalten die Issues und Pull-Requests. Issues sind Bugsreports (Fehlerberichte) und Anfragen nach neuer Funktionalität. Die Pull-Requests sind Änderungsvorschläge, welche einem Review-Prozess unterzogen werden können. Falls der Verwalter eines Projekts einen Änderungsvorschlag akzeptiert, kann er diesen in das Projekt einpflegen (mergen).

4.2.1 Pull-Requests

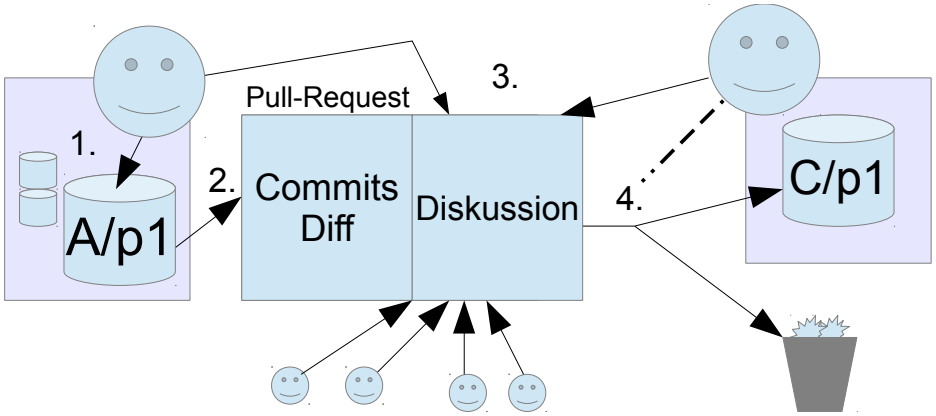


Abbildung 4.2: Pull-Requests auf einer Social Coding Site

In Abbildung 4.2 wird ein möglicher Ablauf eines Pull-Requests dargestellt. Nutzer A hat das Projekt p1 von Nutzer C “geforkt”. Das heißt Nutzer A hat eine Kopie des aktuellen Zustands vom Projekt p1 von C für seinen Zugang erzeugt. Der Fork heißt folglich A/p1.

Er hat auf seiner Kopie Änderungen vorgenommen (Schritt 1). Diese Änderungen fasst er zu einem Pull-Request zusammen (Schritt 2). Der Pull-Request wird im ursprünglichen Projekt C/p1 eingereicht. Es findet eine Diskussion der Änderungen beim Tracker der Pull-Requests von C/p1 auf GitHub statt (Schritt 3). Bei dieser Diskussion können sich auch externe Personen beteiligen. A kann versuchen in der Diskussion seine Änderungen zu erklären und zu verteidigen. C kann als Verwalter vom ursprünglichen Projekt C/p1 die Änderungen einarbeiten (mergen) oder verwerfen (Schritt 4).

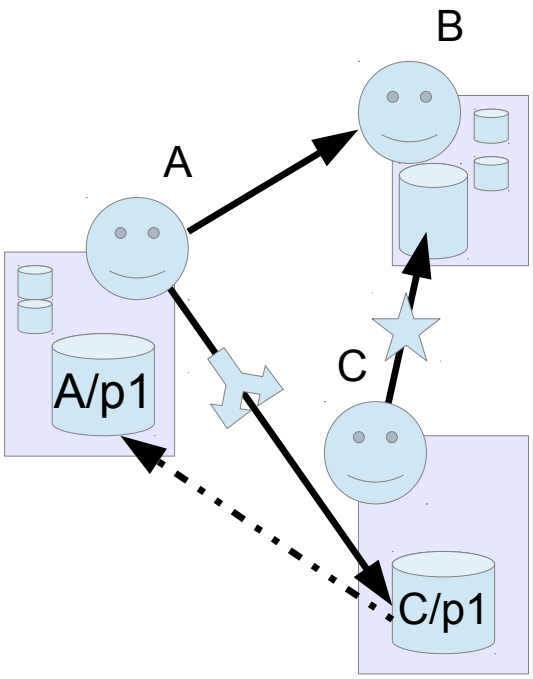


Abbildung 4.3: Assoziationen auf einer Social Coding Site

4.3 Assoziationen

Hinzu kommen auf einer SCS noch soziale Assoziationen. So kann ein Nutzer anderen Nutzern folgen. Ein Nutzer kann aber auch Projekte als interessant kennzeichnen, indem er sie mit einem Stern versieht.

In Abbildung 4.3 ist zu sehen wie Nutzer A den Nutzer B folgt und ein Fork von Projekt C/p1 von Nutzer C vornimmt. Dadurch enthält A eine eigene unabhängige Kopie p1 die nun A/p1 lautet. Weiterhin “sternt” in der Abbildung Nutzer C ein Projekt von Nutzer B.

4.4 Wikis

Zur gemeinschaftlichen Dokumentation bieten SCSs Wikis an.

4.5 Betrachtete Projekte

Proprietäre Industrieprojekte (*closed source*) lassen sich für die Herleitung der Heuristik in dieser Arbeit nicht verwenden. Zwar benutzen auch einige dieser Projekte GitHub.

[1] Tracker dienen der Verwaltung von Aufgaben und Vorschlägen.

Jedoch nur mit nicht öffentlichen Repositories. Es kann also nachfolgend immer nur auf Open Source Softwareprojekte zurückgegriffen werden.

Starke Struktur und Systematik aus kommerziellen Projekten fehlt allgemein bei Open Source Projekten.

In einigen Fällen stellen Firmen auch interne Projekte in öffentlichen Repositories frei zur Verfügung. Viele kommerziellen Softwareprodukte setzen auf freien Bibliotheken auf. Daher sind tendenziell auch Industrieprojekte durch Analyse von Open Source zu erreichen.

Von allen öffentlich verfügbaren Open Source Softwareprojekten werden hier bei der Umsetzung aber nur eine bestimmte Teilmenge betrachtet. Nämlich die auf der Social Coding Site GitHub gehosteten Projekte.

4.6 Testerkennung

4.6.1 Motivation

Vorkommen von Testsammlungen erkennen ist relevant zur Ermittlung von Metriken mit Bezug auf Testcode. Das sind unter anderem die Anzahl der Codezeilen von Tests sowie Anzahl der Beitragenden zu Tests. Weiterhin ist es auch Voraussetzung für die Approximation der Testabdeckung auf statischem Wege. Alle diese Metriken benötigen einen Klassifikator für Code, welcher zuverlässig in Test- oder Produktivcode einordnen kann.

4.6.2 Ansätze

Zur Testerkennung ließen sich in der Literatur keine relevante Arbeiten finden.

Erkennungsmerkmale für Tests sind Konfigurationsdateien für Test-Runner. Sprachen wie C# und Java benutzen zur Kennzeichnung von Testmethoden für JUnit/NUnit spezielle Annotationen. Bei JUnit sind Namenskonventionen üblich: Die Testklasse enthält den Namen der zu testenden Klasse mit Suffix “Test” und Testmethoden werden wie die getestete Methode mit Präfix “test” genannt. Bezeichner nach dieser Konvention werden von integrierten Entwicklungsumgebungen wie beispielsweise Eclipse[2] und IntelliJ IDEA[3] als Vorlagen benutzt.

Zusätzlich können Tests in verschiedenen Projekten in verschiedenen Repositories sein. Dadurch werden sie sehr schwer für eine automatische Testerkennung zu finden.

JUnit[4] ist eins der ersten populären Frameworks für Modultests. Es übernahm Ideen aus SUnit für Smalltalk. Es hat eine ganze Reihe von Unit-Testing-Frameworks unter

[2] http://www.eclipse.org
[3] http://www.jetbrains.com/idea/
[4] http://junit.org/

dem Oberbegriff “xUnit” inspiriert. Obwohl für Programme in Java JUnit eine besonders hohe Popularität besitzt, ist die Verteilung der Test-Frameworks bei anderen beliebten Programmiersprachen auf GitHub von einer sehr hohen Varianz gekennzeichnet.

4.6.3 Frameworks

Es gibt eine unüberschaubare Mengen von Frameworks. Oft mehrere für eine einzige Sprache. Hier eine kleine Auswahl für die häufigsten Sprachen auf GitHub:

- JavaScript: JSUnit, QUnit, Jasmine (BDD).
- Ruby: Test::Unit, RSpec (BDD), Shoulda, Bacon.
- Python: unittest, Doctest, Nose.
- Java: JUnit, TestNG, JWalk, JTest.

Neben den eigenständigen Frameworks für Tests gibt es auch Frameworks, die speziell für ein Projekt maßgeschneidert wurden. Beispielsweise JetBrains testet ihre Entwicklungsumgebung IntelliJ[5] neben JUnit auch mit einem eigenen Framework[6]. Manche Frameworks kommen auch aus dem Umfeld eines bestimmen Unternehmens. Ein prominentes Beispiel hierfür ist googletest[7] von Google.

4.6.4 DSLs für Tests

Die Tests für sehr spezifische Frameworks könnte man höchstens über Schlüsselwörter wie “assert” erkennen. Aber selbst dies wird bei Frameworks, deren Tests in einer internen oder externen Domänenspezifischen-Sprache geschrieben sind[8] nicht nützen. Dort können nämlich Tests sehr verschiedene Formen annehmen. Besonders bei der verhaltensgetriebenen Entwicklung[9] sind prosaische Formulierungen wie “x should be 10” üblich. So etwas wäre theoretisch auch mit einem Algorithmus erkennbar. Nur müsste der in hunderte Sonderfälle verzweigen.

Zusätzlich können Tests für kleine SW-Einheiten (wie Funktionen) auch innerhalb interner[10] oder externer Dokumentation notiert werden. Ein Beispiel hierfür ist das Modul *doctest*[11]. Es durchsucht die Kommentare von Funktionen[12] nach Text, der die Struktur einer beispielhaften Benutzung der Funktion hat. Dann kann *doctest* überprüfen, ob das Codebeispiel wirklich die angegebenen Rückgaben liefert. Hier wird also ein

[5] `http://www.jetbrains.com/idea/`
[6] `https://github.com/JetBrains/intellij-community/tree/master/java/testFramework`
[7] `https://code.google.com/p/googletest/`
[8] Ein Beispiel für eine interne DSL wäre das F#-Framework FsUnit (`https://github.com/dmohl/FsUnit`).
[9] Behavior Driven Development (BDD)
[10] Gemeint sind hier Kommentare innerhalb des erläuterten Codes.
[11] http://docs.python.org/2/library/doctest.html
[12] In Python “docstrings” genannt.

Modultest als Beispiel in der Dokumentation ausgedrückt. Zusätzlich lässt sich aus diesen Kommentaren mit dem Werkzeug *pydoc* externe Dokumentation generieren. Letzendlich wird dadurch die Überprüfung der Konsistenz von Dokumentation und Code automatisiert. Eine Erkennung von Tests in dieser Notation ist machbar (doctest zeigt dies). Die Besonderheit steht aber im Widerspruch zur geforderten Allgemeinheit der zu entwickelnden Heuristik.

Bei Verhaltensgetriebener Entwicklung (BDD) werden die Anforderungen in prosaischer Form aufgeschrieben. Dann werden über reguläre Ausdrücke für die Tests relevante Informationen aus diesen Texten extrahiert. Es gibt bestimmte Konvention mit speziellen Verben. BDD soll Tests näher an die Domäne bringen, ähnlich wie DSLs für Tests.

Eine verbreitete Konvention für Test-Guidelines in externer Dokumentation innerhalb eines Projekts konnte nicht gefunden werden. Eine Extraktion derartiger Informationen aus unstrukturierter Prosa ist momentan Zukunftsvision.

4.7 Scraping vs. Web-API

Scraping hat kein Rate-Limit[13] und erfordert keine Authentifizierung für die Erhöhung desselben. GitHub-Web-API ist technisch einfacher, da sie ihre Antworten in leicht einzulesenden JSON[14] zurück liefert und es bereits für viele Programmiersprachen Bibliotheken zur einfachen Verwendung gibt.

4.7.1 Anfälligkeit von Scraping

Scraping hingegen erfordert schreiben eines regulären Ausdrucks[15], der den zu extrahierenden Inhalt in einer Gruppe “einfängt” (sogenannte capture groups). GitHub-API verändert sich über die Zeit weniger als HTML-Webseiten. Daher sind an ein Scraping-Modul höhere Anforderungen bezüglich der Robustheit gestellt. Die Wahrscheinlichkeit, dass es innerhalb eines Zeitraums aufgrund des geänderten Datenformats kaputt geht, ist deutlich höher als bei einer Web-API.

Dies macht es für den für die Wartung des Scraping-Moduls zuständigen Entwickler notwendig, dass die Regressionstests für diese Module regelmäßig ausgeführt werden, um frühzeitig geändertes Seitenlayout zu erkennen. Auf Änderungen des Aufbaus der Webseite muss gegebenfalls durch Anpassung der regulären Ausdrücke reagiert werden.

Falls es bei Fehlern der Messung von Online-Metriken mit der dieser CD beiliegenden Version kommt, ist das Benutzten der aktuellsten Version von GitHub[16] zu empfehlen.

[13] Ein Rate-Limit begrenzt die Anzahl der Aufrufe einer Schnittstelle in einem gewissen Zeitraum. Ist dieses Limit erreicht, werden weitere Anfragen nicht mehr beantwortet.

[14] JavaScript Object Notation

[15] Siehe Definition 2.2.11

[16] `https://github.com/0x17/JProjectInspector`

5 Testen auf GitHub

Zu Beginn dieser Arbeit soll eine Analyse des Testverhaltens von frei zugänglichen Projekten auf GitHub durchgeführt werden. Dies soll einen Überblick über die Umgebung der zu entwickelnden Heuristik geben.

Es sollen eine Reihe von Fragen bezüglich des Testens auf GitHub beantwortet werden: Welche Projekte testen überhaupt? Welche Technologien werden verwendet? Wer testet? Einige dieser Fragen lassen sich leicht beantworten, wohingegen andere aus Gründen des Aufwands nur heuristisch genähert werden können. Zuerst aber sollen verwandte Resultate aus der Literatur betrachtet werden.

5.1 Literaturergebnisse

Zur Netzstruktur auf GitHub steht einiges in [Thung et al. '13]. Dort wurden einflussreiche Entwickler und Projekte eines Unternetzes von GitHub mithilfe des PageRank-Algorithmus ermittelt.

Die Sprachverteilung auf GitHub ist öffentlich einsehbar[1]. Die beliebtesten Sprachen derzeit sind JavaScript mit 21%, Ruby mit 12%, Python mit 8% und Java ebenfalls mit 8%.

In der Literatur ließen sich drei Artikel zum Thema "Testen auf GitHub" finden. Dies sind [Kochhar et al. '13], [Pham et al. '13] und [Pham et al. '13a].

In [Kochhar et al. '13] wurde die **Anzahl der Testfälle**, die **Projektgröße** in SLOC[2] und die **Anzahl der Entwickler** von 50.000 Projekte auf GitHub-Projekte gemessen.

Etwa 42.66% (21.328) der betrachteten Projekte enthalten mindestens einen Test. Von denen mit Testfällen haben etwa 90% weniger als 100 Testfälle und lediglich 2.57% mehr als 500 Testfälle.

Bei den ausgewählten Projekten liegen die ohne Testfälle im Mittel bei 1.548 LOC mit einem Median von 244. Die Projekte mit Testfällen haben im Schnitt 59.893 LOC bei einem Median von 701. Es gibt eine positive Beziehung zwischen der Anzahl der Codezeilen und der Anzahl der Tests. Es wurde auch festgestellt, dass mit der Projektgröße in Codezeilen die Anzahl der Tests pro Codeziele sinkt, es hier also eine negative Korrelation gibt.

Zu diesen 50.000 Projekten haben 2.967.146 Entwickler beigetragen. Leider wurde

[1] `https://github.com/languages`

[2] Es wurde hier das Werkzeug SLOCCount für die Messung der Codezeilen verwendet.

nicht in den Daten von Git gemessen, welcher Teil dieser Entwickler *tatsächlich zu einem Testfall etwas beigetragen hat*. Es wurde lediglich geschaut, wer überhaupt an einem Projekt mit Testfällen beteiligt war. Das waren 2.864.179 Entwickler also 96.6%. Lediglich 102.967 Beitragende also 3.4% haben die Projekte ohne Tests.

Es wurde festgestellt, dass die Anzahl der Tests zwar mit der Entwicklerzahl steigt, jedoch die Anzahl Tests pro Entwickler mit dem Ansteigen der Entwicklerzahl sinkt.

Zu diesen Ergebnissen bleibt anzumerken, dass zur Erkennung von Testfällen nur eine ungenaue Heuristik verwendet worden ist. Nämlich das Vorkommen der Zeichenkette “test” im Dateinamen. Die Autoren von [Kochhar et al. ’13] haben diese Schwäche selbst erkannt, rechtfertigen sie jedoch mit Gründen der Skalierbarkeit. Ausgefallenere Erkennungstechniken würde das Erfassen von 50.000 Projekten erschweren.

Als Ausblick wird in [Kochhar et al. ’13] noch auf die Beziehung von Fehlerzahl, verwendeter Programmiersprache und der Existenz von Tests im Projekt hingewiesen.

In [Pham et al. ’13] wurden Nutzer auf GitHub über den Einfluss der verstärkten sozialen Transparenz durch GitHub auf das Testverhalten befragt. Weiterhin werden Strategien zur positiven Beeinflussung des Testverhaltens gezeigt.

In [Pham et al. ’13a] wird ein Vorgehen zur Senkung des Aufwands von Projektverwaltern zum Erweitern der Test-Suites vorgestellt, welches auf die Ausnutzung der drive-by-commits aufbaut. Drive-by-commits sind kleine Änderungen, die von dem Projekt fremden aber fähigen Nutzern mit geringem Aufwand beigetragen werden.

Aufgrund des Fokus auf den sozialen Mechanismen hinter dem Testen auf GitHub sind [Pham et al. ’13] und [Pham et al. ’13a] für unsere Betrachtung nur von peripherem Interesse. Ob Tests durch drive-by-commits oder konventionelle Commits beigetragen werden, kann aber einen leichten Einfluss auf den Testbedarf haben. Man könnte vermuten, dass lange am Projekt beteiligte Entwickler mit den Anforderungen des Projekts vertrauter sind als fremde Ersteller von drive-by-commits. Weiterhin können messbare Indikatoren für die soziale Situation (Einstellung zum Testen, d.h. Testkultur) für eine Heuristik des Testbedarfs eine Rolle spielen.

5.2 Eigene Messungen

Es wurde eine eigene Messung an einer Stichprobe von $N = 316$ Java-Projekten vorgenommen.

5.2.1 Projektwahl

Die Auswahl der Projekte fand durch Abgreifen aus der GitHub-Timeline statt. Diese Timeline listet aktuell stattfindende Ereignisse auf GitHub auf. Das könnte beispielsweise ein Commit sein. Es ist anzumerken, dass die Auswahl so jedes derzeit aktive Projekt auf GitHub enthalten könnte. Jedoch sind die Chancen zu Gunsten von relativ aktiveren Projekten erhöht, da diese häufiger Ereignisse auslösen.

5.2.2 Ergebnisse

Von den 316 Projekten enthielten nur 74 Testfälle (23.4%). Testfälle wurden anhand von Schlüsselwörtern[3] in Dateien mit Endung ".java" erkannt. Für diese Heuristik lassen sich falsch positive Ergebnisse konstruieren. Jedoch liegt vermutlich die Genauigkeit über der Lösung aus [Kochhar et al. '13] durch Erkennung der Zeichenkette "test" im Dateinamen.

Von den 74 Projekten mit Testfällen verwendeten 86% JUnit, 9% TestNG und 5% Mockito als Test-Framework. Diese Unterteilung fand durch Suche nach spezifischen Schlüsselwörtern[4] für die Frameworks statt.

Insgesamt haben zu den 74 Projekten eine Menge von 2594 Entwicklern beigetragen. Also im Mittel $\mu = 8.21$ Beitragende pro Projekt. Die Anzahl der Beitragenden wurde von GitHub per Scraping bestimmt.

Von den 2594 Beitragenden haben 292 Entwickler (11.3%) zu Tests beigetragen. Dies wurde mit dem `git-log`-Befehl auf den Testdateien des Git-Repositories der Projekte ausgelesen.

Zusammenfassend scheint also in dieser Stichprobe der Anteil der Projekte mit Testfällen mit 23.4% deutlich unter den 42.66% zu liegen, welche von [Kochhar et al. '13] gemessen worden sind. Dies kann einerseits daran liegen, dass [Kochhar et al. '13] Projekte beliebiger Programmiersprache betrachtet, wohingegen hier nur Java-Projekte betrachtet werden. Also es um die Testkultur von Gemeinschaften anderer Programmiersprachen womöglich besser gestellt ist. Andererseits ist natürlich ein Stichprobenumfang von 316 deutlich kleiner als die 50.000 Projekte, welche in [Kochhar et al. '13] betrachtet worden sind. Zusätzlich könnte das unterschiedliche Verfahren zur Erkennung von Testfällen zu den verschiedenen Ergebnissen mit beigetragen haben. Falls häufig verwendete Frameworks für Tests von den gewählten Schlüsselwörtern nicht erfasst werden, kann die Erkennung von Tests mithilfe des Teilstrings "test" im Dateinamen womöglich doch die bessere Lösung sein.

[3] Die verwendeten Schlüsselwörter waren: "cucumber", "@Test", "org.junit", "assertEqual".

[4] "org.testng" für TestNG, "org.mockito" für Mockito, "@Exemplars" und "@UseCase" für SureAssert sowie "@Test", "org.junit" und "assertEqual" für JUnit.

6 Goal Question Metric

Dieses Kapitel dokumentiert einen Versuch den GQM-Ansatz für die Bestimmung von Metriken zur Untersuchung des Testbedarfs zu verwenden. Der GQM-Ansatz stammt von [Basili et al. ’94]. Dabei wurde das Vorgehen aus [Schneider ’07] befolgt.

6.1 Zielbaum

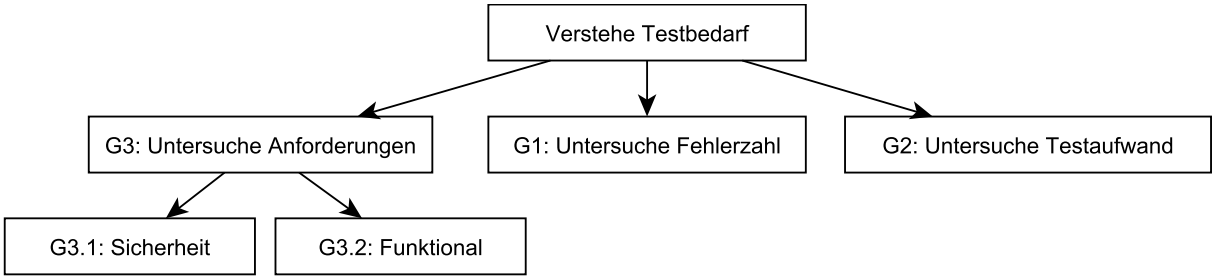


Abbildung 6.1: Zielbaum.

Es wurde mit der Erhebung und Verfeinerung der Ziele begonnen. Dies wird im Zielbaum in Abbildung 6.1 dargestellt.

6.2 Facettenbeschreibung

Ziel	Zweck	Qualitätsaspekt	Betrachtungsgegenstand	Perspektive
G1	Untersuche	Korrektheit (Fehlerzahl)	Projekt auf SCS	Projektverwalter

Tabelle 6.1: Zielfacetten

Das erste Ziel G1 wurde in der Facettenbeschreibung in Tabelle 6.1 in Aspekte zerlegt.

6.3 Abstraction Sheet

Ein Abstraction Sheet zur Untersuchung von Ziel G1 ist in Abbildung 6.2 dargestellt. Die rechten beiden Quadranten konnten hierbei laut [Schneider ’07] aufgrund des Zwecks der Untersuchung (characterize) ausgelassen werden. Beim Abstraction Sheet

Abstraction Sheet — GQM-Goal G1

Zweck	**Q-Aspekt**	**Betrachtungsgegenstand**	**Perspektive & Umgebung**
Untersuche	Korrektheit	OSS Projekt auf SCS	Proj.verwalter

1. Qualitätsfaktoren: Fehlerzahl, Testaufwand und Testqualität.	4. Einflussfaktoren: Leer, da Zweck „Untersuche“.
2. Ausgangshypothesen: Nicht anwendbar, da Betrachtungsgegenstand zu allgemein und abstrakt ist!	3. Einflusshypothesen: Leer, da Zweck „Untersuche“.

Abbildung 6.2: Abstraction Sheet zur Untersuchung des Testbedarfs.

zeigt sich, dass der GQM-Ansatz bei fortschreitender Konkretisierung mit der Allgemeinheit der Aufgabenstellung in Konflikt gerät. Das Problem ist im unteren linken Quadranten sichtbar.

6.4 Anwendbarkeit von GQM

GQM wurde ursprünglich für die Auswertung von Zielen *einer* Organisation entwickelt. Nach [Basili et al. ’94] wurde die Verwendung von GQM aber auf einen größeren Kontext ausgedehnt. Dieser Kontext umfasst aber nicht beliebige Projekte auf einer Social Coding Site. Dies zeigte sich bereits beim Ausfüllen des Abstraction Sheets.

Das Schätzen des derzeitigen Zustands von Qualitätsfaktoren bei einem Projekt ist machbar. Bei mehreren Projekten innerhalb einer Organisation geht es bei Gemeinsamekeiten auch. Bei Millionen von sehr heterogenen SCS-Projekten ist dies jedoch nicht mehr möglich. Sie verwenden verschiedene Technologien und Entwicklungsmethoden.

Die Anforderung der Allgemeinheit an die Heuristik verhindert also einen kompletten Durchlauf des GQM-Ansatzes. Trotz dieses Rückschlags muss der Grundgedanke von GQM weiter verfolgt werden. Die Heuristik soll vom Konzept des Testbedarfs ausgehend entwickelt werden. Dieses Konzept ist Thema des nächsten Kapitels.

7 Testbedarf

Dieses Kapitel soll die in Kapitel 2 gegebene Definition 2.1.8 des Testbedarfs begründen.

7.1 Gesamtkosten

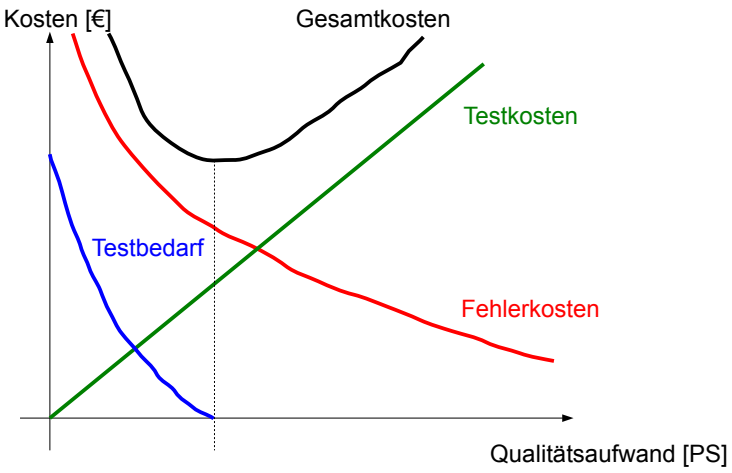


Abbildung 7.1: Kosten eines Projekts in Abhängigkeit vom Qualitätsaufwand in Personenstunden. Nach [Schneider '07].

Die Gesamtkosten der Entwicklung von Software ergeben sich als Summe der Fehlerkosten und der Fehlerbehebungskosten (Testkosten). Die Kurve der Gesamtkosten ist in Abbildung 7.1 dargestellt. Die Gesamtkosten sinken bei steigenden Testkosten und sinkenden Fehlerkosten bis zu einem bestimmen Punkt. Nach diesem Punkt ist eine Art Sättigung des Testens eingetreten und die Fehlerkosten sinken schwächer als die Testkosten ansteigen. Dann ist weiteres Testen nicht mehr wirtschaftlich. Eine wichtige Aufgabe ist es, diesen Punkt zu erkennen und ihn als Testabbruchskriterium zu sehen. Es gibt eine Reihe Literatur, welche sich mit diesem Thema befasst[1].

Aus der Abbildung lässt sich auch ablesen, dass der Testbedarf positiv korreliert zu den Fehlerkosten sein muss. Die Korrelation des Testbedarfs mit den Testkosten ist negativ.

[1] Zum Beispiel [Chao und Yang '93] sowie [Dalal und Mallows '88].

Testaufwand und Fehlerzahl können als Stellvertreter für Testkosten beziehungsweise Fehlerkosten gesehen werden. Bei Annahme eines direkt linearen Zusammenhang mit dem Testbedarf ergibt sich Definition 2.1.8:

$$\text{Testbedarf} =_{\text{Def.}} \frac{\text{Fehlerzahl}}{\text{Testaufwand}}$$

7.2 Anforderungen

Die Anforderungen einer Software spielen für Fehler und damit auch den Testbedarf eine große Rolle. Leider sind die Anforderungen auf SCS nicht maschinenlesbar hinterlegt und daher nicht automatisiert ermittelbar.

7.3 Risiken

Verwandt zu den Anforderungen sind die ein Projekt betreffenden Risiken. In systematisch durchgeführten Projekten werden diese bereits früh im Rahmen einer Risikoanalyse bestimmt. Ob Projekte auf einer SCS dies ebenfalls praktizieren ist fraglich. Da es keine Konvention oder maschinenlesbares Format zur Dokumentation der Risiken gibt, könnten diese ebenfalls nicht algorithmisch extrahiert werden.

Risiken spielen jedoch für effektives Testen eine wichtige Rolle. So werden bei risikogetriebenem Testen zuerst die für die ermittelten Risikos relevanten Testfälle eingeführt. Dabei wird die Reihenfolge der Erstellung der Tests an absteigender “risk-exposure”[2] der Risiken ausgerichtet.

[2] Ergibt sich als Produkt von Auftretenswahrscheinlichkeit eines Risikos multipliziert mit dem erwarteten Schaden durch Eintritt.

8 Ausgangshypothesen

Das hier sind alles Ausgangshypothesen die zu Beginn der Arbeit aufgestellt worden sind. Die Umfrage und die auf den dazugehörigen Projekten betriebenen Messungen dieser Größen sollen diese Hypothesen überprüfen, d.h. bestätigen, korrigieren oder widerlegen.

8.1 Projektmetriken (DVCS, Social Coding Site)

Projekt-Popularität (#Contributors) (PP) Je höher die Projektbeliebtheit, desto größer die Anzahl der am Projekt beteiligten Entwickler. Diese benutzen diese Software, lesen ihren Code und tragen zu ihr im Rahmen von Pull-Requests und dazugehörigen Diskussionen bei. Erfahrungsgemäß steigt die Qualität von OSS-Projekten, welche von vielen Personen gereviewed wird. Dies spricht für gesenkten Testbedarf bei hoher Popularität. Ein Beispiel für ein beliebtes und qualitativ hochwertiges Projekt auf GitHub ist Bootstrap[1] mit 100 Beitragenden, etwa 50.000 Sternen und einer Historie von über 7.000 Pull-Requests bei derzeit nur 200 offenen Issues (wobei bereits über 7.000 Issues geschlossen wurden).

Andererseits verderben viele Köche bekanntlich den Brei. Viele beteiligte Entwickler führen zu Inkonsistenzen und damit auch zu mehr Fehlern. Dies zeigten auch [Meneely und Williams '12] dadurch, dass Änderungen von fremden Code stärker mit Fehlern korrelieren. Die möglicherweise vorkommenden Inkonsistenzen bei mehreren Entwicklern lassen sich nur durch strikt eingehaltene Richtlinien mildern. Der Overhead der Kommunikation steigt exponentiell an. Der Nutzen von Regressionstests steigt besonders, da die Tests meist die Absichten des ursprünglichen Autor eines Moduls konserviert. Nicht dokumentierte und nicht in Tests ausgedrückte Annahmen eines Entwicklers verursachen bei Projekten mit größeren Teams viel wahrscheinlicher Fehler. Dies spricht für erhöhten Testbedarf bei hoher Popularität.

Falls sich die entgegengesetzten Einflüsse auf den Testbedarf nicht zu einer Nullsumme ausspielen (d.h. im Endeffekt Popularität orthogonal zu Testbedarf), muss durch Messungen ermittelt werden, ob die Popularität den Testbedarf senkt oder erhöht.

Testpopularität (#Contributors (Test-Pull-Reqs.) / #Contributors) (TP) Anteil der Tests beisteuernden Entwickler von der Gesamtzahl der Code beitragenden Entwickler. Indiz für Testkultur bei >50%. Ist der Großteil der Beitragenden am Testen, so steigt

[1] `https://github.com/twitter/bootstrap`

nach [Pham et al. '13] die Motivation neuer Entwickler (speziell auch Entwickler von drive-by commits) ebenfalls Tests für bestehenden Code oder eigens beigetragenen Code zu erstellen. Dies spricht für einen gesunkenen Testbedarf bei hoher Testpopularität.

Defektmaß (#Bugs in Issue-Tracker) (DM) Nur Issues betrachten vom Typ "Fehler", keine Feature-Anforderungen. Nach Auffinden eines Fehlers muss zuerst der zuständige Testfall korrigiert/erweitert werden beziehungsweise ein neuer Testfall verfasst werden. Erst dann sollte der Fehler im System behoben werden, was durch den Test, der erst scheitert und nach der Korrektur erfolgreich verläuft, validiert wird. Folglich steigt der Testbedarf direkt mit der Anzahl der bekannten Fehler.

Selektivität (#Pull-Reqs.-#Merges) (SEL) Wenn ein Projektverwalter nur einen kleinen Bruchteil der Pull-Requests in das Projekt integriert, kann dies mehrere Gründe haben. Unter anderem nur ungewollte Änderungen oder nur Änderungen von Nutzern, denen der Verwalter nicht vertraut. Die Vermutung liegt aber nahe, dass mit einer hohen Selektivität auch ein ausführlicheres Review der Pull-Requests von Seiten des Verwalters voraus geht. Dies würde für geringeren Testbedarf durch steigende Selektivität sprechen.

Codefrequenz ((#Added-#Deleted) in last week) (CF) Die Codefrequenz ist ein Maß für den Umfang der aktuellen Code-Veränderungsaktivitäten am Projekt. Besonders Projekte mit hoher Codefrequenz können durch Tests profitieren, da (ausreichende) Fehlerfreiheit von vielen Änderungen in kurzer Zeit manuell nicht praktikabel überprüfbar ist. Es spricht also eine hohe Codefrequenz für hohen Testbedarf.

Projektalter (Age of git repo) (PA) Gerade bei jungen Projekten ist häufig die Architektur noch nicht verfestigt. Speziell in dieser Situation kann das Hinzufügen von Tests und die dadurch möglicherweise nötigen Umstrukturierungen (Stichwort: Testbarkeit und Schnittstellen) zu einer Verbesserung des Designs führen. Zusätzlich sind jüngere Projekte meist kleiner und daher besonders tief hängende Früchte für Tests im Rahmen von drive-by commits. Die inhärente Korrektheit ist jedoch vom Projektalter unabhängig. Der Testbedarf kann aber für jüngere Projekte höher angesetzt werden, da der Nutzen von Tests bei diesen Projekten besonders hoch ist. Erstens wegen der genannten Architekturverbesserung und zweitens aufgrund des Vorbild/Beispielcharakters für hinzukommende Entwickler. Zusätzlich sind die Fehlerkosten umso höher, je älter das Projekt ist. Da Fehler am günstigsten an einem frühen Zeitpunkt entfernt werden können. Die Fehlerkosten steigen mit dem Projektalter exponentiell an.

Churn Veränderungen führen zu neuen Fehlern. Selbst Veränderungen zur Behebung eines Fehlers können an einer anderen Stelle einen neuen Fehler erschaffen.

8.2 Software-/Produktmetriken

Lines of code (LOC) Anzahl der Codezeilen steigt mit Umfang des Codes. Der Leseaufwand (aber nicht unbedingt der Verständnisaufwand) erhöht sich mit steigendem LOC-Wert. Es wird oft je nach CMM-Level von bestimmter Fehlerzahl pro kLOC ausgegangen. Das impliziert eine linear proportionale Beziehung von LOC eines Systems und der Zahl der Fehler. Folglich steigt der Testbedarf mit den LOC.

Zyklomatische Komplexität (McCabe, CC) McCabe steigt mit Kanten (Verzweigungen, Schleifen) aber nicht mit Knoten (Anweisungsfolgen). Die logische Komplexität des Kontrollflusses einer Funktion erhöht sich speziell bei Verzweigungen und Schleifen. Höhere logische Komplexität erhöht die Fehlerwahrscheinlichkeit und somit den Testbedarf.

Halstaedt Siehe auch Kapitel 9. Die Halstead Fehlerzahl B ist als Stellvertreter für die tatsächliche Fehlerzahl in einem Projekt mit mehr oder weniger großem Erfolg verwendet worden. Für die Halstead-Metriken sind sprachspezifische Größen wie die Anzahl der Operatoren und Operanden zu ermitteln. Dies betrifft den Produktivcode.

Kopplung Bei stark gekoppelten Modulen erhöht sich die Wahrscheinlichkeit bei lokalen Änderungen ein Fehlverhalten in entfernten Modulen (Fernwirkung) einzuführen. Solche Fehler lassen sich oft nur durch Regressionstests erkennen. Bei Modulen mit wenig wechselseitigen Abhängigkeiten und hoher Kohäsion ist es dagegen leichter durch lokale Änderungen hervorgerufenes Fehlverhalten durch manuelles Testen der vom Modul bereitgestellten Funktionalität zu sehen. Der vermutete SWQ-Gewinn durch besseres Testen von SW mit hoher Kopplung (Abhängigkeitsgraph mit hohen Ein/Ausgangsgraden an den Knoten) ist also höher als bei geringer Kopplung. D.h. gestiegener Testbedarf bei hoher Kopplung.

8.3 Produkt-/Test-Metriken

Abdeckung Je kleiner die Abdeckung, desto größer ist der Anteil des Anwendungscode, welcher bei Tests überhaupt nicht geprüft wird. D.h. desto höher ist auch der Testbedarf. Je feiner die Granularität des Abdeckungs-Maß (Modul<Methode<Pfad), desto genauere Aussagen über den Testbedarf (vor allem relativ gesehen) sind möglich.

Testredundanz Die Testabdeckung lässt sich nach Berücksichtigung aller Pfade nicht weiter erhöhen. Dennoch können möglicherweise durch Hinzufügen weiterer Prüf-Fälle zusätzliche Fehler aufgedeckt werden. Beispielsweise kann durch Probieren zusätzlicher Eingabewerte beim Aufruf einer mathematischen Bibliotheksfunktion eine

Singularität[2] getroffen werden. Dann würde dieser zusätzliche Test einen neuen Fehler finden bei unveränderter Testabdeckung. Bei vergleichbarer Testabdeckung ist der Testbedarf von Projekten mit höherer Testredundanz also eventuell als besser zu bewerten.

Erhöhte Testredundanz kann bei nicht steigender Abdeckung aber auch eine Erhöhung des Testaufwands ohne Senkung der Fehlerkosten bedeuten. Da Pfadabdeckung die Bibliotheksfunktionen (oder gar tieferer Code-Schichten) nicht beachtet, ist dies aber nicht zwangsläufig der Fall. Allgemein ist geringe Abdeckung ein stärkeres Indiz für Testbedarf als geringe Testredundanz.

8.4 Testmetriken

Testqualität Automatisierte Tests und speziell Modultests sind meistens selbst Code[3]. Dieser Code kann ebenso wie System-Code durch seine Struktur seine eigenen Qualitätsmerkmale (Lesbarkeit, Wartbarkeit und so weiter) beeinflussen. Es würde sich anbieten eine spezielle Heuristik für die Qualität von Tests zu definieren. Dies kann auf Basis von Empfehlungen aus der Literatur geschehen. Ein Projekt mit Testcode niedriger Qualität hat erhöhten Testbedarf. Für ausreichende Begründung, muss gesenkte Testeffektivität bei niedrigen Werten einer Metrik für Testqualität gezeigt werden.

Test-LoC/LoC (TCR) Stellt Umfang von Testcode im Verhältnis zum eigentlichen Systemcode. Üblicherweise <100%. Es ist ein Maß für den relativen Aufwand. Klar ist, dass Projekte mit sehr kleinem Wert (zum Beispiel 5%) einen erhöhten Testbedarf haben. Besonders hohe Werte (beispielsweise 95%) lassen geringe Testeffizienz vermuten. Bei ansonsten vergleichbaren Ausprägungen in anderen Merkmalen sollte ein Projekt mit geringerem TCR einen höheren Testbedarfs-Wert erhalten.

Testfails (TE) #Gescheiterter-Testdurchläufe/#Commits. Ein Grund für das Testen ist das Auffindung von Fehlern. Besteht ein System nach jeder Veränderung immer alle Tests, so ist dies ein Indiz (Smell) für mangelnde Testqualität/quantität. Daher sollte der Testbedarf bei großer Zahl von fehlgeschlagenen Tests geringer sein: Hier besteht eher Fehlerbehebungsbedarf am System. Leider kann man nicht eindeutig für das Senken des Testbedarfs bei Projekten mit häufigem Status “gescheiterte Tests” argumentieren. Denn zusätzliche Tests könnten hier weitere Fehler aufdecken. Praktisch ist dieser Wert aufwendig zu bestimmen (außer bei Projekten mit Integration von Travis-CI). Daher bin ich mir noch unsicher, ob auf Basis von Messungen weitere Beurteilung des Einflusses von Testscheitern auf den Testbedarf zielführend ist.

[2] Zum Beispiel $\tan(x)$ für alle $x = k \cdot \pi$ mit $k \in \mathbb{Z}$.

[3] In einigen Ausnahmefällen auch Tabellen siehe FitNesse (`http://en.wikipedia.org/wiki/FitNesse`)

9 Fehlerzahlmodelle

Dieses Kapitel befasst sich mit bisherigen Forschungsergebnissen bezüglich von Modellen zur Vorhersage der Fehlerzahl.

Die Bestimmung der noch in einem Programm verbleibenden Fehler hat eine große Bedeutung für den Entwicklungsprozess von Software im Allgemeinen und für das Testen im Speziellen. Denn aus der geschätzten Zahl verbleibender Fehler lassen sich Stoppregeln ableiten. Hat man diese Zahl auch noch feiner für einzelne Module, ließe sich gar eine Teststrategie durch sie begründen. Da für die Softwaretechnik die Fehlerzahl einen so hohen Wert hat, gibt es bereits aus den 70er Jahren eine Reihe von Forschungsbestrebungen in diese Richtung.

Von den in der Literatur entwickelten Modelle für Teststrategien, Stoppregeln, Fehlerzahlschätzungen ist nur eine Teilmenge auf unsere Problemstellung übertragbar. Projekte auf einer Social Coding Site stellen andere Daten zur Verfügung als konventionelle Industrieprojekte. Artefakte aus Anforderungs- oder Entwurfsphasen sind meist nicht vorhanden und häufig gibt es diese Phasen nicht einmal. Dafür gibt es aber zusätzliche Informationen auf SCS, welche es bei herkömmlichen Projekten nicht gibt. Hier sticht besonders die Veränderungshistorie und die Konversationen der Beitragenden bei Pull-Requests hervor.

9.1 Motivation für Fehlervorhersagemodelle

Warum ist die Defektdichte ($\frac{\#\text{Bugs}}{\text{SLOC}} = \frac{N}{\text{SLOC}} =: \rho$) eines Projekts bestimmend für seinen Testbedarf?

Der **Testbedarf** wird geprägt durch das Zusammenspiel von bisherigen Testkosten (Q-Aufwand) und Kosten durch verbleibende Fehler. *Bisherige* **Testkosten** lassen sich grob durch Code Churn von Testdateien in einer Versionsverwaltung einschätzen. Hierbei sollten gelöschte Zeilen nicht beachtet werden. Die **Fehlerkosten** lassen sich mit der Fehlerzahl B nähern.[1]

Modelle zur Näherung von B gibt es eine Reihe in der Literatur mit unterschiedlicher Übertragbarkeit auf Social Coding Sites.

Eine positive Korrelation einer Testbedarfs-Heuristik mit der Defektdichte wäre insofern auch wünschenswert, da die Chance mit einem Test einen Fehler zu finden so maxi-

[1] Beides sehr ungenau. Aufwand pro hinzugefügter/modifizierter Zeile in Testcode kann stark unterschiedlich sein (ausgeklügelter Test? Trivialtest?). Kosten pro einzelnem Fehler können ebenfalls stark variieren aufgrund des unterschiedlichen Schaden. Stichwort: Bug severity.

miert werden würde. Denn laut [Myers '04]: Ziel des Testens ist es Fehler in einem Projekt zu finden. Das würde also die **Effektivität** des Testens steigern.

9.2 Modelle aus der Literatur

9.2.1 Fehlerzahl über Größe: SLOC

"Single lines of code" (SLOC) ist primitivstes Maß für Fehlerzahl bei Annahme von konstanter Fehlerdichte ρ. Gaffney [Gaffney '84] fand 21 Bugs pro KSLOC aber es gibt sehr viele unterschiedliche Werte. [Li et al. '98] schildern, dass NASA es mit ihren bis dahin besten Methoden auf 0.1 Bug pro KSLOC geschafft hat. Level der Programmiersprache hat laut Gaffney keinen Einfluss auf die Fehlerdichte. Vokabelgröße (vgl. SW-Science) ebenfalls geringe Bedeutung.

Laut [Rosenberg '97] ist genaue Definition von LOC für analytische Betrachtung weitgehend **irrelevant**! *Andere Metriken korrelieren stark mit SLOC*. Auf dieses Resultat sind auch [Herraiz und Hassan '10] gekommen. Weiterhin postuliert Rosenberg, dass SLOC zur Vorhersage der Qualität ungeeignet ist und eher als Covariate zur Einbeziehung der Größe in anderen Metriken sinnvoll ist. Laut [Lipow '82] korreliert Defektdichte positiv zu SLOC.

9.2.2 Halstead Software Science

[Halstead '77]: Halstaed's gelieferte Fehler (B) ist eine Abschätzung für die Anzahl der Fehler in der Implementierung. Sie berechnet sich wie folgt:

$$B = \frac{E^{\frac{2}{3}}}{3 \cdot 10^3}.$$

Wobei neuerdings auch $B = \frac{V}{3 \cdot 10^3}$ akzeptiert wird.

Die verwendeten Terme sind wie folgt definiert:

- Aufwand $E = D \times V$,
- Schwierigkeit $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$,
- Volumen $V = N \times \log_2 \eta$,
- Programmlänge $N = N_1 + N_2$,
- Vokabular $\eta = \eta_1 + \eta_2$,
- Anzahl eindeutiger Operatoren η_1,
- Anzahl eindeutiger Operanden η_2,
- Gesamtzahl Operatoren N_1,
- Gesamtzahl Operanden N_2.

Positive Beurteilungen

Die Validität dieser Metrik wurde experimentell sehr oft unabhängig und bei unterschiedlichen Programmen und Sprachen nachgewiesen. Siehe dazu [Ottenstein et al. '76]. In [Mancoridis '09] steht: "Bei Programmgrößen zwischen 300 und 12k Anweisungen fand Lipow [Lipow '82] nur 8% Abweichung bei vorhergesagter Fehlerzahl und tatsächlicher Fehlerzahl."

Es gibt in [Stetter '86] einen Verbesserungsvorschlag der eine einfachere Berechnungsvorschrift vorschlägt:

$$\frac{B}{P} = \frac{K}{3000} \cdot \mathrm{ld}\left(\frac{8 \cdot K \cdot P}{1 + 8 \cdot \mathrm{ld}(K \cdot P) - 9 \cdot \mathrm{ld}(\mathrm{ld}(K \cdot P))}\right).$$

Mit Anzahl Fehler B, SLOC P, durchschnittlicher Anzahl Operatorenund Operanden pro Codezeile K.

Diese Metrik von Halstead ist *relativ* einfach zu berechnen und **vollständig automatisierbar**. Beispielsweise das Werkzeug CodePro AnalytiX[2] kann sie bestimmen.

Negative Beurteilungen

Laut [Fitzsimmons und Love '78] ist Aussagekraft von SW-Sciences durch "unreine" Programme eingeschränkt. Unreinheiten sind inverse Operatoren, mehrdeutige Operanden, synonyme Operanden, gemeinsame Unterausdrücke, unnötige Ersetzungen, unfaktorisierte Ausdrücke. In Forschung analysierte Programme meist schon in Publikationsform gebracht, also rein. Es wird also die Übertragbarkeit von Halstead auf nicht konstruierte Beispiele angezweifelt.

Ein sehr kritisches Dokument zu SW-Sciences von [Hamer und Frewin '82], in welchem kritisiert wird, dass "Großteil der Beziehungen und Interpretationen der SW-Sciences sind weder Naturgesetze noch nützliche Approximationen für Ingenieure!"

Weitere Kritik an den SW-Scienes findet sich auch in [Schneider '07]. Erstens lassen sich die in ihr enthaltenen Metriken nicht auf objektorientierte Sprachen übertragen. Zweitens wird ihre Eignung für als zu optimierende handlungsleitende Größe angezweifelt. Als Beispiel verkleinert sich durch Wiederverwendung von Bezeichnern für Variablen die Anzahl der Operanden. Dies beeinflusst das Qualitätsziel der Lesbarkeit negativ. Halstead's Maß der geschätzten Fehlerzahl sei konkret nur als Richtwert für die insgesamt durch Tests zu findene Anzahl von Fehlern.

Leider sind Implementierungen dieser Metrik jedoch sprachabhängig. Denn Operatoren und Operanden müssen anhand der Syntax erkannt werden. Aufgrund der hohen Varianz der verwendeten Sprachen in Projekten auf Social Coding Sites, sind die Metriken von Halstead hier also nicht anwendbar.

[2] `https://developers.google.com/java-dev-tools/codepro/doc/`

9.2.3 Basierend auf logischer Kopplung

Es wurden in [Ahsan und Wotawa '11] acht Metriken für logische Kopplung zwischen Quelldateien vorgestellt und gezeigt, dass diese Metriken stark mit der Fehlerzahl korrelieren. Es wird gefolgert, dass die Metriken für Vorhersagemodelle genutzt werden könnten. Es wurden zwei Ansätze verfolgt:

1. Regressionsbasiertes Fehlervorhersagemodell: Auf Basis schrittweiser Regressionsanalyse. Zuerst Modell mit Metriken, welche größte Korrelation mit Fehlerzahl haben anfangen. Danach zusätzliche Metriken hinzufügen basierend auf partieller Korrelation mit Metriken, die schon im Modell sind. Nach Hinzufügen einer neuen Metrik in das Modell wird das Modell evaluiert und Metriken, welche nicht signifikant beitragen, werden entfernt. Dadurch kommt am Ende ein Modell heraus, welches die maximale verbleibende Varianz erklärt.
2. Klassifikationsbasiertes Fehlervorhersagemodell: Auf Basis von "machine learning algorithm" J48 (Decision-Tree). Verschiedene Evaluationsmaße: Precision, Recall & Accuracy. Definiert wie aus Information Retrieval bekannt. Trainingsinstanzen (Menge Metrikwerten zu einer Quelldatei) in drei Klassen: LOW, MEDIUM, HIGH. Kriterium für Klassen ist # post release bugs.

9.2.4 Negativ Binomiales Regressionsmodell

Beschrieben in [Ostrand et al. '05]. In welchen Dateien sind die meisten Fehler beim nächsten Release? (Für traditionelle kommerzielle Projekte. Anwendbarkeit auf GH eingeschränkt!)

Vorhersage basierend auf aktuellem Code und der gemeldeten Fehler und der Veränderungshistorie der Datei. Multivariate Analysis. Abwandlung linearer Regression. Teilweise Poisson-Verteilung.

9.2.5 Exponentielles SRGM (software reliability growth model)

SRGM ist bekanntestes Modell zur quantitativen Evaluation des Testprozess und liefert nützliche Informationen beispielsweise zur Anzahl der verbleibenden Fehler in der Software oder Mean Time Between Failures (MTBF). [Mizuno und Kusumoto '97]

SRGM gehen von Zusammenhang zwischen Anzahl gefundener Defekte und Testzeit aus. Basis für den Ansatz aus [Li et al. '98].

9.2.6 Basierend auf Coverage

Beschrieben in [Li et al. '98]. Testabdeckung hilft bei Vorhersage von Fehlerzahl. Die Anzahl verbleibende Fehler wichtigstes Maß für SWQ. Ab bestimmten Schwellenwert

(C_{Knee}) der Testabdeckung (abhängig von Abdeckungsmaß) lineare Beziehung zwischen Coverage und gefundener Defektzahl. Totale Defektzahl etwa wo Abdeckung 100% nähert (minus toter Code). Wäre mit Ansatz zu statische Testabdeckung mit Program-Slicing von Alves [Alves und Visser '09] automatisierbar.

9.2.7 Code Churn

Code Churn bezeichnet die Anzahl der hinzugefügten, veränderten und entfernten Zeilen in einem Repository in einem Zeitraum oder bei einem Commit.

Nagappan und Ball zeigen in [Nagappan und Ball '05], dass Maße, welche Code Churn in Beziehung zu anderen Variablen wie Komponentengröße oder zeitliche Ausdehnung des Churn setzen, ein guter Prädiktor für Defektdichte sind.

Hauptursache an diesem Zusammenhang sollen laut [Eaddy und Zimmermann '08] Veränderungen an "Cross Cutting Concerns" sein. Also die querschnittlichen Aspekte (vergleiche Aspektorientierte Programmierung), welche aufgrund ihrer querschnittlichen Natur nicht ordentlich modularisiert und zu hoher Kohäsion gebracht werden können. Diese Aspekte sollen zumindest eine Ursache für Fehler sein. Da gibt es auch in [Eaddy und Zimmermann '08] Metriken zu, die den "Diffusionsgrad" (scattering) von Aspekten zu quantifizieren versuchen. Ist sehr spannendes Thema, da jedem Programmierer die Problematik von zerstreutem Code zu einem Aspekt bekannt ist.

Laut [Meneely und Williams '12] konnte gezeigt werden, dass Code Churn korreliert ist mit Fehlern und Sicherheitslücken. Jedoch beachtet Churn nicht den menschlichen Faktor. Also wer die Veränderungen macht. Vielversprechend sind interaktive Churn-Metriken, als Code-Churn Variante. Mit `git blame` lässt sich der zu gegebener Revision letzte Entwickler, der eine gegebene Codezeile verändert hat, bestimmen. Dann kann man für eine gegebene Revision sagen, ob der Autor der Revision seinen eigenen Code (self churn) oder fremden Code (interactive churn) modifiziert hat. Diese Metrik ist statistisch korreliert mit Sicherheitslücken bei Veröffentlichung eines Release. Weiterhin ist sie nur schwach korreliert mit Code Churn und SLOC. Man könnte für Projekte den durchschnittlichen interaktiven Churn berechnen, indem man den interaktiven Churn für alle Revisionen mittelt. Das ist jedoch sehr aufwendig.

9.2.8 Bayesian-SPAM-Klassifier (Text-Mining)

H. Hata beschreibt in seiner Dissertation [Hata '12] folgendes: Die Messung von Quellcode und/oder Repository für Fehlervorhesage sehr aufwendig. Auswahl der Metriken schwer. Es gibt keine beste Teilmenge der Metriken für Fehlervorhersage. Komplexitätsmetriken sind nur nach Validation für Zielprojekt nützlich. Aufgrund der starken Korrelation könnte man, wenn Teilmenge der Metriken reicht, auch alle nehmen. Wie man Metriken nutzt ist wichtiger als welche man sich aussucht.

Lösungsansatz: Mizuno/Kikuno/Hata haben auf Spamfilter basierendes Vorhersagemodell vorgeschlagen.

SW-Modul wird wie E-Mail behandelt und klassifiziert in fehleranfällig (failure prone = FP) und nicht fehleranfällig (non failure prone = NFP). Ansatz ist Sprachunabhängig und unabhängig von Repository und daher besonders praktisch.

Es wurde bei Hata neben dem Bayes-Klassifikator dafür auch logistische Regression angewendet und Resultate verglichen.

9.2.9 Feingranulare Historie

Aus [Hata '12] und speziell auch in [Hata et al. '12]: Veränderungshistorie auf Methoden-Ebene und nicht nur Datei-Ebene. Tool für GitHub $\rightarrow$ feine Historie: Historage. Das ist sehr interessant, da gerade bei größeren Commits, welche viele Änderungen umfassen oder gar durch Zusammenfassung mehrerer lokaler Commits entstanden sind, kleine Änderungen und Bugfixes in der Masse untergehen. Analyse mit Deltas von feinerer Granularität können ganz intuitiv schon mehr oder genauere Muster aufzeigen.

9.2.10 OSS Zuverlässigkeitsmetriken

In [Tiwari und Pandey '12] geht es darum, dass Open Source Software (OSS) aufgrund von Befürchtungen bezüglich der Zuverlässigkeit sich in Indien nicht durchgesetzt hat. Daher braucht man Metriken zum einfachen Assessment von Zuverlässigkeit und SWQ im Allgemein von OSS.

Wichtig: OSS Entwicklungsmethode anders als konventionell: Hat keine formellen Dokumente für Anforderungen, Entwurf, Testen, et ceteri. Bietet aber öffentliche Statistiken für: #contribs, #commits, #users, bugreports, fixing time.

Qualität und Zuverlässigkeit von OSS muss mit diesen Parametern eingeschätzt werden. Zuverlässigkeitsmetriken wurden zu Metriken basiert auf Repositorymaßen von OSS abgeleitet. Es werden zwei einfach zu ermittelnde Maße vorgestellt:

1. $\frac{\#\text{contribs}}{\text{KSLOC}}$
2. $\frac{\#\text{commits}}{\text{KSLOC}}$

Die beiden Vorschläge werden durch Argumentation aus der Literatur begründet. Es findet in dem Artikel (aus 2012!) keine Evaluation statt. Aufgrund der Einfachheit der Ermittlung könnte man Berechnung der beiden Metriken zumindest in das Tool mit einbauen.

9.2.11 Relative Komplexitätsänderung

Aus [Nikora und Munson '98]: Fault "surrogate" zur Schätzung von Fehlerzahl und Lokation. Surrogate basiert auf Änderungen der relativen Komplexität. Fehlerzahl korreliert positiv damit.

Es wird gezeigt, dass die Änderungsrate der relativen Komplexität als guter Index für die Fehlerinjektionsrate dient. Hat man die Fehlerinjektionsrate ermittelt, so kann man die Anzahl verbleibender Fehler zu jedem beliebige Zeitpunkt in der Entwicklung abschätzen. Weiteres aus diesem sehr ausführliches Paper von 31 Seiten führt jetzt zu weit. Werde bei Gelegenheit komplett lesen.

9.2.12 Poisson-Annahme

In [Dalal und Mallows '88]. Gesamtzahl der Fehler in einem Modul (N) ist unbekannt und hat eine Poisson-Verteilung mit Mittel λ, welches zufällig (über die Module) verteilt ist nach einer bekannten Γ-Verteilung. Bei gegebenem Bug ist die Zeit zum Auffinden beim Testen zufällig mit gegebener Verteilnug G einer beliebigen Form. Die Lebenszeiten (einfügen bis erkennen und fixen) verschiedener Bugs sind unabhängig.

Nicht sicher vielversprechend da eher abstrakte Überlegungen mit vielen Unbekannten.

9.2.13 Genetische Programmierung

Fehlermodell aus [Afzal et al. '09] auf Basis von "genetric programming" liefert bessere oder gleichwertige Ergebnisse zu traditionellen Methoden. Es wird in [Afzal et al. '09] festgestellt, dass es bisher keinen Konsens über bestes Modell zur Fehlervorhersage gibt. Je nach Datensatz haben die Modelle unterschiedliche Stärken und Schwächen. Das Vorhersageproblem selbst ist NP-hart laut [Shepperd und Kadoda '01].

Benötigt Fehlerzahldatensätze zum Training, welche Fehlerzahlen als Zeitserie enthalten. Sie müssen in Training und Testdatensätze unterteilt werden. Üblicherweise erste 2/3 zum Training und letzten 1/3 zur Evaluation.

Lösungen im Suchraum werden als symbolische Ausdrücke in Form von Parsbbäumen dargestellt, welche aus Funktionen und Terminalzeichen bestehen. Die Qualität einer Lösung ist gemessen mit einer Evaluationsfunktion. Üblicherweise wird die Differenz zwischen erhaltenem und erwartetem Ergebnis benutzt: $\sum_{i=1}^{n} |e_i - e_i^{'}|$ wobei e_i tatsächliche Fehlerzahl und $e_i^{'}$ die geschätzte Fehlerzahl. Parsebaumlänge kann durch Variationsoperatoren beeinflusst werden. Es gibt Selektionsmechanismen zur bestimmung der Individuen der Nachfolgegeneration. Hier wurde Cross-Over mit Branch-Swapping (Zweigvertauschung) gewählt und zufällig Knoten von zwei Elternbäumen gewählt. Zusätzliche wurden Mutationen in welchen ein zufälliger Knoten vom Elternbaum ersetzt wird mit einem neuen zufälligen Knoten welcher mit allen verfügbaren Terminalzeichen und Funktionen erzeugt wurde. Kleiner Anteil von Individuen wurde auch in die Nachfolgegeneration ohne Änderung kopiert. Selektionsmechanismus wählte zufällige Anzahl Individuen von der Population und wählte den mit besseren Fitness. Falls geliche Fitness dann den mit weniger Knoten.

9.2.14 Martingale

In [Yip und Xi '99] wird eine Martingale-Gleichung verwendet zur Einschätzung der Fehlerzahl beim Debugging. Es wird ein bekannter Proportionalitätsfaktor zwischen Fehlerrate eines neu entdeckten Fehler und eines vorgegeben Fehlers vorausgesetzt. Daher nicht übertragbar auf GitHub. In [Yip '95] werden auch eine bekannte Anzahl "seeded" Fehler in das System eingefügt. Also ebenfalls nicht übertragbar.

9.2.15 Gibbs-Sampler

Fehlerzahlschätzung über "recapture debugging model". Gibbs-Sampler und Metropolis-Algorithmus zur Inferenzprozedur. Es wäre ν die geschätzte #Bugs im Modell. Ansatz sieht mathematisch spannend aus aber beim kurzen Überfliegen würde ich sagen, dass die Annahmen zu Beginn sind für GitHub nicht zu halten ist.

9.2.16 Jacknife Estimator

Aus [Chao und Yang '93]: Homogenität von Fehlerraten spielt eine Rolle für Modell der Fehlerzahl. Maximum Likelihood Estimator für gleiche Fehlerraten geeignet. $\hat{N}_1^*$ für ungleiche Raten geeignet, da hohe und niedrige Frequenzen separat behandelt werden wobei N die Anzahl der Fehler ist. Vorgeschlagen ursprünglich von Burnham & Overton (1978, 1979). Bezug auf "recapture debugging".

9.2.17 Lokale und Globale Neuigkeitsgewichtung

Aus [Joshi et al. '07]: Anzahl Fehler welche in Monat i gemeldet wird ist am stärken von Anzahl Fehler gemeldet in Monat $i-1$ beeinflusst. Also $\#\text{Bugs}_i = \alpha_i + \#\text{Bugs}_{i-1}$ mit Stabilisierungsfaktor α_i. Fehlervorhersage (gemeldete Fehler) für Zukunft auf Basis bisheriger Bugtracker-Daten möglich. Im Rahmen von MSR-Challenge also relativ nah zur Anwendung auf GitHub.

9.2.18 Generalisiertes Stochastisches Petri-Netz

Ein GSPN-Modell [Mizuno und Kusumoto '97] zur Vorhersage der Fehlerzahl durch Nutzung von Daten aus bisherigen Projekten. SRGM-Modelle erlauben die Fehlerzahl zu nähern. Program-Slicing erlaubt Fehlerposition zu bestimmen. Relevanz für mich gering, da Daten aus früheren Projekten nötig. Sehr an konventionellen Entwicklungsprozess mit Anforderungen, Entwurf, Implementierung und Review angelehnt. Daher kaum auf Projekte einer SCS übertragbar.

9.2.19 Invarianten-Analyse

Daikon ist ein Programm, welches wahrscheinliche Invarianten in Software erkennt. Eine Invariante ist eine Bedingung, welche immer eingehalten wird in bestimmten Programmbereichen. Es wird hauptsächlich für das Debugging benutzt. Eine gewisse Nähe zur statischen Analyse ist erkennbar. Invarianten finden könnte auch zur Testfallgenerierung nützlich sein.

9.2.20 Systematic Literature Review

Laut [Hall et al. '11]: Viele Studien zur Fehlervorhersage (wichtig für Teststrategien) mangeln an Kontext und Methodeninformation zum vollen Verständnis. Von 208 Studien wurde in systematischer Literaturrecherche (gibt es Guideline zu: SLR) haben nur 36 die Kontext/Methodenkriterien erfüllt. Gute Modelle wurden in Kontext von großen Systemen erstellt und benutzen als unabhängige (ungebundene) Variablen eine Menge von Metriken. Modelle die KSLOC benutzen sind nicht schlechter. Gute Modelle nutzen einfache Modellierungstechniken wie Naive Bayes oder logistische Regression.

9.2.21 Anmerkungen zu Metriken und Defekten

In [Fenton und Neil '00] gibt es eine Reihe Anmerkungen zu Metriken und Defekten:

- Bestehende Modelle unfähig Fehler akkurat vorherzusagen mit Größe- und Komplexitätsmetriken alleine. Keine Erklärung des genauen Art des Einfluss von Fehlerentstehung und Erkennung auf Fehlerzahl.
- Größenbasierte Metriken sind schlecht zur Fehlervorhersage (obwohl sie mit Fehlerzahl korrelieren).
- Statische Komplexitätsmaße wie McCabe sind nicht signifikant besser aufgrund starker Korrelation mit Größenmetriken (ist uns ja schon bekannt).
- Die Verwendung von Defektzahlen als stellvertretendes Maß ("Surrogate") für Qualität ist grundsätzlich problematisch. Besonders die häufig durchgeführte Zählung von Fehlern vor Veröffentlichung[3] ist ein schlechter Indikator für Qualität.

In [Nagappan et al. '06] wird behauptet, dass fehleranfällige Software-Entitäten statistisch korreliert mit Komplexitätsmetriken sind. Es gebe aber keine einzige Menge von Komplexitätsmetriken, welcher als universelles bestes Vorhersagemodell geeignet wäre.

[3] Solche Aussagen haben leider keine Gültigkeit auf Social Coding Sites in denen ein Projekt höchstens einen Alpha- oder Beta-Status hat aber ansonsten in kontinuierlicher Entwicklung ist und nicht in die konventionellen Projektzustände eingeteilt werden kann.

9.2.22 Pareto-Verteilung der Fehler

Laut [Weyuker und Ostrand ’10] gibt es eine Pareto-Verteilung der Fehler in Software. Es befinden sich grob 80% der in nur 20% des Quellcodes.

10 Testabdeckung

10.1 Testeffektivität

Die Testeffektivität wurde als Anzahl der durch Tests gefundenen Fehler definiert. Sie ist positiv korreliert mit der Testabdeckung.

Jedoch gilt bezüglich der Fehlerverteilung im Programm laut [Weyuker und Ostrand '10] eine Pareto-Verteilung. Das heißt 80% der Defekte befinden sich in lediglich 20% des Quelltextes. Daher gibt es mit steigender Testabdeckung kein lineares Wachstum der Testeffektivität.

Die Steigung dieses Wachstum hängt nämlich von der Fehlerdichte in den zusätzlich getesteten Abschnitten des Produktionscodes ab. Diese Dichte ist maximal in den genannten ca. 20% des Quellcodes, welcher etwa 80% der Fehler enthält. Diese 20% stellen einen Häufungspunkt dar. Sind sie vollständig von korrekten Testfällen erfasst erfolgt eine Sättigung der Testeffektivität (Siehe auch Abbildung 7.1). Dies kann eine sinnvolle Stoppregel sein, da Testen danach nicht mehr wirtschaftlich ist. Eine weitere Indirektion zwischen Testabdeckung und Testeffektivität ist die Tatsache, dass zusätzliche Testfälle, welche die Anforderungen nicht korrekt oder vollständig erfassen, zu keiner gesteigerten Testeffektivität führen.

Die Testabdeckung wächst zwar monoton mit der Anzahl der Testcodezeilen. Jedoch ist dieses Wachstum nicht streng monoton und zudem wie bereits begründet *nicht linear*. In der Literatur ließ sich keine enge Beziehung zwischen Testcodezeilen und Abdeckung finden, weshalb die Testcodezeilen leider nicht als einfach zu bestimmender Proxy der Coverage benutzt werden können. Daher wird neben der Heuristik zur Erkennnung von Testfällen[1] auch eine statische sprachunabhängige Heuristik für die Testabdeckung benötigt.

10.2 Dynamisch vs. Statisch

Für *dynamische Testabdeckung* spricht: Verschiedene Varianten unterschiedlicher Granularität (Methode, Pfad, Anweisung) möglich. Lässt sich möglicherweise automatisieren bei Projekten, welche entsprechende Konfigurationsdateien beinhalten. Ideal ist dies bei Projekten, welche den Online-Continuous-Integration(CI)-Dienst TravisCI verwenden. Hier müssen Konfigurationen für das CI-System sowie Build- und Test-Systeme vorliegen, die ein automatisiertes Laden und Installieren der Abhängigkeiten

[1] Siehe Abschnitt 11.3.

sowie Bauen und Ausführen der System-Tests erlauben. Weiterhin könnte u.U. auch andere Projekte automatisiert kompilierbar und ausführbar sein, falls diese ein Buildsystem wie Maven oder Gradle benutzen, welche die nötigen Abhängigkeiten explizit (genaue Version) und mit ihrer Quelle (Weblink) angeben. In jedem Fall aber ist der zu erwartende technische Aufwand viele Projekte derart zu verarbeiten aufgrund der Heterogenität der eingesetzten Technologien ein hoher.

Automatisiertes “bauen” und Ausführen von Programm und deren Tests ist aber nur in Sonderfällen möglich. Ein solcher Sonderfall wäre Einbettung in bekanntes Continuous Integration System (CI-System). Beispielsweise Travis CI. Dies ist aber nur bei Minderheit der Projekte der Fall.

Für *statische Testabdeckung* spricht also, dass kein Kompilieren und Ausführen nötig. Das wäre sonst schwierig zu automatisieren und besitzt viele Abhängigkeiten. Performance von statischer Messung sollte in Größenordnung von Codezeilen liegen. Dynamische Messung hingegen kann beliebig lange benötigen. Beispielsweise schlechte Test-Suites, welche aufgrund von Eingabe/Ausgabe-Abhängigkeiten sehr lange Ausführungszeiten haben. Je nach Umsetzung könnte man auch mehrere Test-Frameworks einfacher berücksichtigen.

10.3 Umgesetzte statische Testabdeckung

Nachfolgend wird ein Algorithmus für die statische Annäherung der Testabdeckung von Projekten auf einer allgemeinen Social Coding Site entwickelt.

Ideal wäre eine vollständige Sprachunabhängigkeit. Dies würde aber leider aufgrund der hohen Varianz der Testing-Frameworks und deren Konventionen den Rahmen dieser Arbeit um ein vielfaches sprengen. Daher konzentrieren wir uns auf die vier populärsten Sprachen auf GitHub. Dies sind in Reihenfolge absteigender Popularität JavaScript, Ruby, Java und Python.

Es müssen für die Approximation der Methodenabdeckung im Produktivcode/Domänencode alle Deklarationen von Methoden erkannt werden. Aus den Deklarationen wird ein Index der Methoden aufgebaut. Dann müssen Testfälle im Code erkannt werden und die darin aufgerufenen Methoden des Domänencode notiert werden. Für jede der Sprachen wurden bereits für die Heuristik der Testfallerkennung Schlüsselwörter definiert, welche auf Testcode schließen lassen. Das Verhältnis von im Testcode aufgerufenen Methoden zur Gesamtzahl aller deklarierten Methoden im Produktivcode ist dann der approximierte Wert.

Zusammenfassend lautet der Algorithmus wie folgt:

Definition 10.3.1 (Coverage Algorithmus)**. Eingabe:** Projekt (owner/repo)
Ausgabe: Angenäherte Methodenabdeckung

1. Codedateien in Verzeichnisbaum in Testcode und Produktivcode einteilen.
2. Index aus im Produktivcode deklarierten Methoden aufbauen.

3. Anzahl der Aufrufe jeder Methode aus Index in Testcode zählen.
4. Methodenabdeckung $:= \frac{\text{\# aufgerufener Methoden}}{\text{\# Methoden gesamt}}$.

Deklarationen und Aufrufe von Funktionen werden über reguläre Ausdrücke erkannt. Dabei gibt es für jede der Sprachen JavaScript, Java, Ruby und Python ein spezielles Modul, welches aus dem Quelltext Deklarationen und Aufrufe extrahiert. Dafür wurde eine Schnittstelle `IFunctionIndexer` eingeführt.

Die Metrik wird in der Klasse `RoughFunctionCoverage` implementiert und greift auf Indexer für die verschiedenen Sprachen zu. Die Indizierer für Code der Sprachen befinden sich im Paket:

```
org.andreschnabel.jprojectinspector.tests.offline.metrics.
test.coverage.indexers.
```

11 Werkzeug zur Analyse und Visualisierung

Dieses Kapitel beschäftigt sich mit dem im Rahmen dieser Arbeit entwickelten Werkzeug *JProjectInspector*. Das Werkzeug ermöglicht die Erfassung, Analyse und Visualisierung von Metriken beliebiger Projekte auf GitHub. Zusätzlich enthält es eine Funktion zur Bewertung von Vorhersagemodellen für den geschätzten relativen Testaufwand und die geschätzte relative Fehlerzahl von Projekten eines Nutzers. Die hierfür benötigten vertraulichen Daten befinden sich auf der beiliegenden CD (siehe Abschnitt "Compact Disc" im Anhang). Sie stammen aus den beiden während dieser Arbeit mit Nutzern von GitHub durchgeführten Umfragen. Die Bezeichnung für dieses Werkzeug wurde gewählt, da es in der Programmiersprache Java verfasst ist und es Projekte durch Ermittlung von Messwerten "inspiziert".

Ein ähnliches Werkzeug ist GlueTheos, welches nach meinem Wissen nicht auf GitHub anwendbar ist. Sein Aufbau wird in [Robles et al. '04] beschrieben. Der Aufbau dieses Werkzeugs ist ähnlich. Nach bestem Wissen des Autors gab es noch kein ausreichend geeignetes Werkzeug für die Problemstellung, weshalb ein eigenes Werkzeug entwickelt worden ist.

11.1 Entwicklungsmethode

Das Werkzeug wurde schrittweise in kleinen Inkrementen entwickelt. Dadurch konnte gut auf geänderte Anforderungen im Verlauf der Arbeit reagiert werden. Zuerst wurden für das Explorationskapitel 5 Klassen zur Erkennung von Tests, Test-Frameworks für Java und dem Anteil der an Tests beteiligten Entwickler umgesetzt. Dann wurden für die Umfrage Klassen zum zufälligen Abgreifen von Nutzern aus der GitHub-Timeline erstellt. Als Vorarbeit zur Beurteilung von Vorhersagemodellen wurden Module zum Messen weiterer Metriken implementiert. Erst danach wurde für die bestehende Funktionalität eine grafische Oberfläche hinzugefügt.

Bei der Entwicklung wurde durchgehend eine hohe Testabdeckung angestrebt. Dadurch war es möglich ohne Risiko von Regressionen frühe und häufige Refaktorisierungen durchzuführen. Auf aussagekräftige Bezeichner und kurze Funktionen wurde geachtet. Der Quellcode wurde durchgehend kommentiert, was die Erzeugung einer HTML-Dokumentation mithilfe des Werkzeugs `javadoc` ermöglichte. Diese befindet sich neben *JProjectInspector* auf der beigefügten CD[1].

[1] Siehe auch "Compact Disc"-Abschnitt im Anhang.

Mit Test Driven Development entwickeln. Häufiges Refaktoring. Dokumentation nur wenn nötig. Ansonsten durch viele Tests (Beispielcode) und Wahl von guten Bezeichnern dokumentieren. Siehe auch UncleBob. Vorbildsreferenz für geringen Testbedarf.

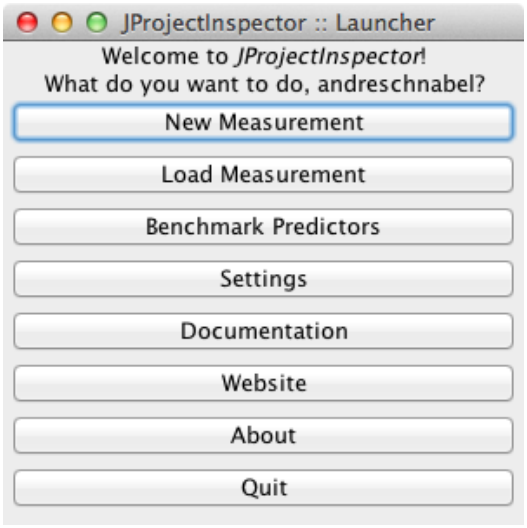


Abbildung 11.1: Auswahlfenster.

11.2 Bedienung

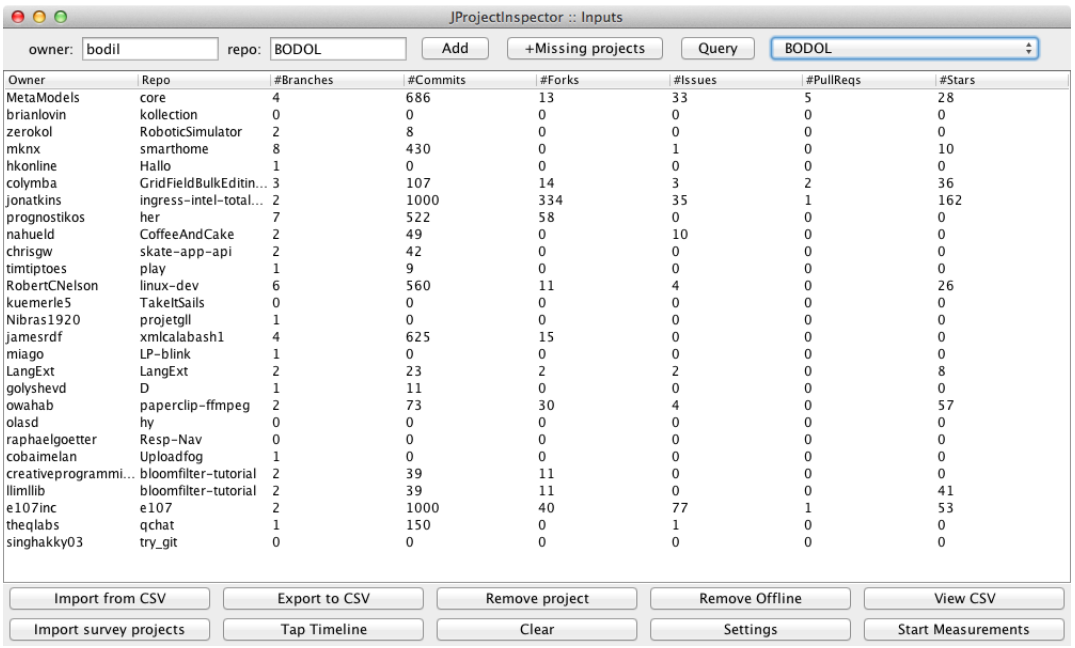
Das Werkzeug startet in ein Auswahlfenster (siehe Abbildung 11.1), welches den Nutzer aus einer Reihe verschiedener Tätigkeiten wählen lässt.

Es lässt sich im Auswahlfenster über jeweils eine Schaltfläche das Handbuch und das GitHub-Profil von *JProjectInspector* öffnen. Im Handbuch ist die Installation und Einrichtung des Werkzeugs beschrieben. Das Werkzeug benutzt eine Reihe von Programmen, deren Pfad im Einstellungsdialog mitgeteilt werden sollte. Die Programme sind bestehen aus dem Kommandozeilen-Client von Git, einem Perl-Interpreter und dem Skript *cloc*[2]. Dabei wird *cloc* zur Messung der Anzahl der Codezeilen benutzt. Für die Messung vieler Metriken muss eine Kopie des betrachteten Repository vorliegen. Das Verzeichnis, in welches dieses temporär “geklont” wird, lässt sich im Einstellungsdialog festlegen.

11.2.1 Projektauswahl

Die Schaltfläche “New Measurement” führt den Nutzer in ein Fenster, in welchem er Projekte für die Messung sammeln kann (Abbildung 11.2). Projekte lassen sich dort aus CSV-Dateien importieren, aus der Timeline abgreifen oder durch Eingabe

[2] Genaue Funktionsweise von *cloc* wird in [Danial ’13] beschrieben. Verwendet und auf der beiliegenden CD enthalten ist *cloc* in der Version 1.58.



Owner	Repo	#Branches	#Commits	#Forks	#Issues	#PullReqs	#Stars
MetaModels	core	4	686	13	33	5	28
brianlovin	kollection	0	0	0	0	0	0
zerokol	RoboticSimulator	2	8	0	0	0	0
mknx	smarthome	8	430	0	1	0	10
hkonline	Hallo	1	0	0	0	0	0
colymba	GridFieldBulkEditin...	3	107	14	3	2	36
jonatkins	ingress-intel-total...	2	1000	334	35	1	162
prognostikos	her	7	522	58	0	0	0
nahueld	CoffeeAndCake	2	49	0	10	0	0
chrisgw	skate-app-api	2	42	0	0	0	0
timtiptoes	play	1	9	0	0	0	0
RobertCNelson	linux-dev	6	560	11	4	0	26
kuemerle5	TakeItSails	0	0	0	0	0	0
Nibras1920	projetgll	1	0	0	0	0	0
jamesrdf	xmlcalabash1	4	625	15	0	0	0
miago	LP-blink	1	0	0	0	0	0
LangExt	LangExt	2	23	2	2	0	8
golyshevd	D	1	11	0	0	0	0
owahab	paperclip-ffmpeg	2	73	30	4	0	57
olasd	hy	0	0	0	0	0	0
raphaelgoetter	Resp-Nav	0	0	0	0	0	0
cobaimelan	Uploadfog	1	0	0	0	0	0
creativeprogrammi...	bloomfilter-tutorial	2	39	11	0	0	0
llimllib	bloomfilter-tutorial	2	39	11	0	0	41
e107inc	e107	2	1000	40	77	1	53
theqlabs	qchat	1	150	0	1	0	0
singhakky03	try_git	0	0	0	0	0	0

Abbildung 11.2: Projekte zur Messung sammeln.

von Nutzername und Repository-Name manuell eingeben. Dabei erlaubt die Schaltfläche “Query” die automatische Erfassung aller Projekte des im Textfeld eingegebenen GitHub-Nutzers. Zusätzlich bietet das Fenster zur Projektauswahl noch eine Reihe weiterer Optionen. Es können Projekte aus einer Umfrage importiert werden. Alle gesammelten Projekte können mit “Clear” verworfen werden. Nicht mehr verfügbare Projekte über “Remove Offline” und das Projekt der gewählten Zeile über “Remove Project” entfernt werden.

11.2.2 Metriken

Wurden Projekte gesammelt, kann durch Betätigung der Schaltfläche “Start Measurement” das Fenster zur Auswahl der zu erfassenden Metriken geöffnet werden (Abbildung 11.3). Dort werden auf der linken Seite die verfügbaren Metriken in einer Liste dargestellt. Diese Metriken wurden im Werkzeug bereits implementiert. Auf der rechten Seite befindet sich eine Liste der für die Messung ausgewählten Metriken. In der Mitte kann der Benutzer in einem Beschreibungstext mehr Informationen über die zuletzt in einer der Listen ausgewählte Metrik erhalten. Zwei Schaltflächen mit Pfeilen erlauben das Verschieben von Metriken zwischen den beiden Listen. In der Mitte befindet sich eine Schaltfläche, mit der sich die Messung einleiten lässt.

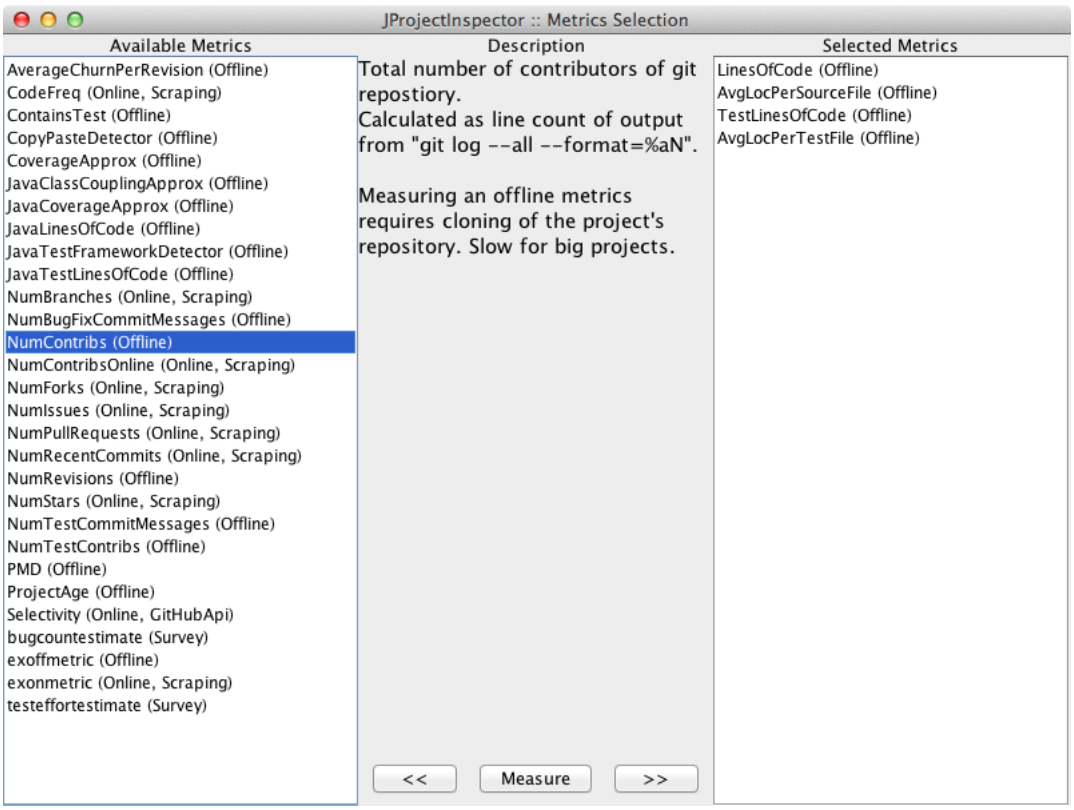


Abbildung 11.3: Fenster zur Auswahl von Metriken für die Messung.

11.3 Heuristik zur Testerkennung

Zur Bestimmung der Test-LOC wurden mithilfe einer Heuristik die Dateien im Verzeichnisbaum des Projekts bestimmt, welche Testfälle enthalten. Die Heuristik sucht je nach Dateiendung nach Schlüsselwörtern aus einer gegebenen Liste. Beispielsweise wird für Dateien mit der Endung “java” im Dateinamen nach den Schlüsselwörtern “cucumber”, “Test” , “org.junit” und “assertEqual” im Inhalt der Datei gesucht. Wird ein solches Schlüsselwort gefunden, wird diese Datei als Test kategorisiert. Zusätzlich werden alle Dateien mit Quellcode unterhalb eines Ordners, welcher den Teilstring “test” enthält, als Tests klassifiziert.

Mitgelieferte Metriken

Zum Abgabezeitpunkt dieser Arbeit werden folgende sprachunabhängige Metriken in *JProjectInspector* mitgeliefert:

- LinesOfCode: #Codezeilen im Produktivcode. Durch Anwendung von “cloc” auf komplettes Repository und Auslesen der gemessenen Codezeilen.
- TestLinesOfCode: #Codezeilen im Testcode. Testcode-Dateien werden über Heu-

ristik mit Schlüsselwörtern[3] erkannt. Dann werden die Codezeilen dieser Dateien mit “cloc” ermittelt und aufsummiert.

- CoverageApprox: Statische Approximation des Testbedarfs. Vorgehen wurde in Abschnitt 10.3 beschrieben.
- NumBranches: #Branches (Entwicklungszweige) auf GitHub. Wird von Webseite des Repositories auf GitHub mit Scraping extrahiert.
- NumForks: #Forks (Abspaltungen) auf GitHub. Wird ebenfalls per Scraping ermittelt.
- NumIssues: #Issues auf GitHub. Ebenfalls Scraping.
- NumStars: #Sterne auf GitHub. Ebenfalls Scraping.
- NumContributors: #Beitragenden. Über “git log” aus dem Git-Repository ausgelesen.
- NumRevisions: #Revisionen. Über “git rev-list” vom Git-Repository extrahiert.
- NumPullRequests: #Pull-Requests (Änderungsvorschläge) auf GitHub. Per Scraping ermittelt.
- AverageChurnPerRevision: Durchschnittlicher Churn (Anzahl Zeilen hinzugefügt sowie entfernt pro Revision) über alle Revisionen. Über “git show” vom Repository ermittelt.
- AvgLocPerFile: Durchschnittliche Anzahl Codezeilen pro Codedatei. Ergibt sich aus LinesOfCode dividiert durch die Anzahl der Codedateien, welche von “cloc” ermittelt wurde.
- AvgLocPerTestFile: Durchschnittliche Anzahl Testcodezeilen pro Datei mit Testcode. Aus TestLinesOfCode dividiert durch die Anzahl der Dateien, die von der Heuristik für die Testerkennung erkannt worden sind. Die Heuristik wurde in Abschnitt 11.3 beschrieben.
- NumTestCommitMessages: #Commit-Nachrichten welche Stichwort “test” enthalten. Mithilfe von “git log” gemessen.
- NumBugFixCommitMessages: #Commit-Nachrichten welche ein mit Fehlern und deren Korrektur assoziiertes Stichwort enthalten.

Dabei wurden die Messung der Codezeilen und des Churn durch die Literaturrecherche aus Kapitel 9 motiviert. Weitere Metriken wurden zur Überprüfung der Ausgangshypothesen aus Kapitel 8 implementiert. Das detaillierte Vorgehen bei der Messung lässt sich in den entsprechenden Klassen im Paket

`org.andreschnabel.jprojectinspector.metrics`

nachvollziehen.

[3] Siehe dazu Abschnitt 4.6.

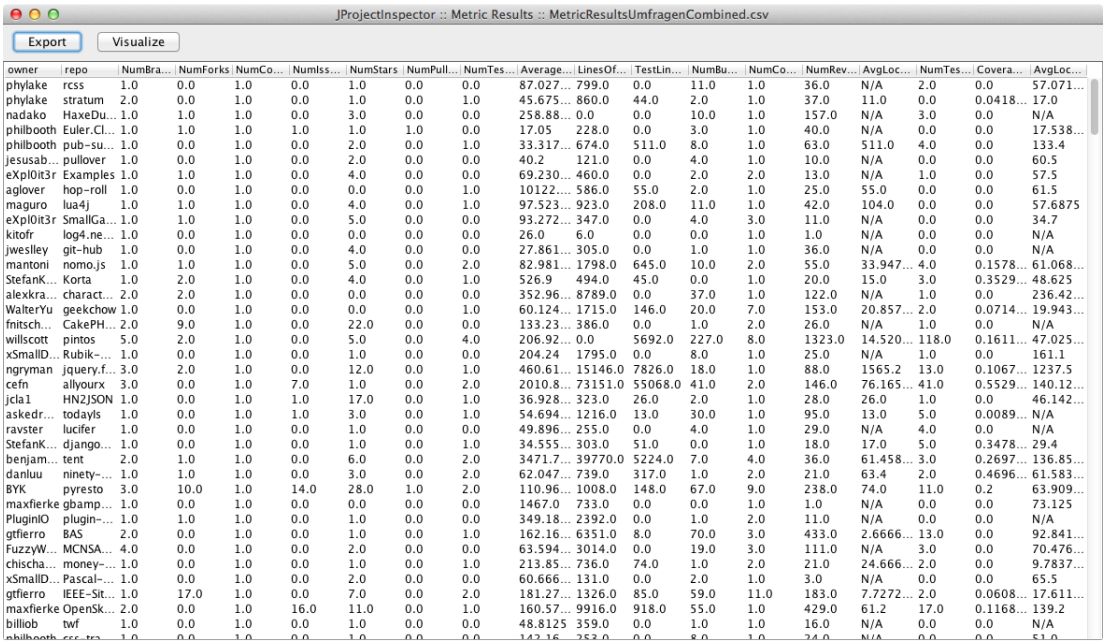
Die LOC-Werte wurden mithilfe des Perl-Skript *cloc*[4] ermittelt. Dieses Skript ermittelt die Anzahl der tatsächlichen Codezeilen, der Kommentarzeilen und der leeren Zeilen. Das genaue Vorgehen ist auf der Webseite von “cloc” beschrieben. Hier wird nur die Anzahl der Codezeilen weiter verwendet.

Zusätzliche sind noch einige Java-spezifische Metriken enthalten. Sie wurden zu einem sehr frühen Zeitpunkt der Arbeit umgesetzt. Besonders hervorzuheben sind:

- PMD: Anzahl Regelverstöße laut PMD[5],
- CPD: Anzahl duplizierter Codezeilen laut CPD (copy paste detector) von PMD.

Diese nur für Java-Projekten ermittelbare Metriken konnten leider für die Entwicklung der Heuristik nicht weiter verwendet werden, da in den Umfragen kein Nutzer nur in dieser Sprache geschriebene Projekte besaß. Weiterhin liegt das Thema dieser Arbeit nicht speziell auf Projekten, welche Java verwenden.

11.3.1 Messung



owner	repo	NumBra...	NumForks	NumCo...	NumIss...	NumStars	NumPull...	NumTes...	Average...	LinesOf...	TestLin...	NumBu...	NumCo...	NumRev...	AvgLoc...	NumTes...	Covera...	AvgLoc...
phylake	rcss	1.0	0.0	1.0	0.0	1.0	0.0	0.0	87.027...	799.0	0.0	11.0	1.0	36.0	N/A	2.0	0.0	57.071...
phylake	stratum	2.0	0.0	1.0	0.0	1.0	0.0	1.0	45.675...	860.0	44.0	2.0	1.0	37.0	11.0	0.0	0.0418...	17.0
nadako	HaxeDu...	1.0	1.0	1.0	0.0	3.0	0.0	0.0	258.88...	0.0	0.0	10.0	1.0	157.0	N/A	3.0	0.0	N/A
philbooth	Euler.Cl...	1.0	1.0	1.0	1.0	1.0	1.0	0.0	17.05	228.0	0.0	3.0	1.0	40.0	N/A	0.0	0.0	17.538...
philbooth	pub-su...	1.0	0.0	1.0	0.0	2.0	0.0	1.0	33.317...	674.0	511.0	8.0	1.0	63.0	511.0	4.0	0.0	133.4
jesusab...	pullover	1.0	0.0	1.0	0.0	2.0	0.0	0.0	40.2	121.0	0.0	4.0	1.0	10.0	N/A	0.0	0.0	60.5
eXpl0it3r	Examples	1.0	1.0	1.0	0.0	4.0	0.0	0.0	69.230...	460.0	0.0	2.0	2.0	13.0	N/A	1.0	0.0	57.5
aglover	hop-roll	1.0	0.0	1.0	0.0	0.0	0.0	1.0	10122....	586.0	55.0	2.0	1.0	25.0	55.0	0.0	0.0	61.5
maguro	lua4j	1.0	1.0	1.0	0.0	4.0	0.0	1.0	97.523...	923.0	208.0	11.0	1.0	42.0	104.0	0.0	0.0	57.6875
eXpl0it3r	SmallGa...	1.0	1.0	1.0	0.0	5.0	0.0	0.0	93.272...	347.0	0.0	4.0	3.0	11.0	N/A	0.0	0.0	34.7
kitofr	log4.ne...	1.0	0.0	1.0	0.0	0.0	0.0	0.0	26.0	6.0	0.0	0.0	1.0	1.0	N/A	0.0	0.0	N/A
jwesley	git-hub	1.0	0.0	1.0	0.0	4.0	0.0	0.0	27.861...	305.0	0.0	1.0	1.0	36.0	N/A	0.0	0.0	N/A
mantoni	nomo.js	1.0	1.0	1.0	0.0	5.0	0.0	2.0	82.981...	1798.0	645.0	10.0	2.0	55.0	33.947...	4.0	0.1578...	61.068...
StefanK...	Korta	1.0	2.0	1.0	0.0	4.0	0.0	1.0	526.9	494.0	45.0	0.0	1.0	20.0	15.0	3.0	0.3529...	48.625
alexkra...	charact...	2.0	2.0	1.0	0.0	0.0	0.0	0.0	352.96...	8789.0	0.0	37.0	1.0	122.0	N/A	1.0	0.0	236.42...
WalterYu	geekchow	1.0	0.0	1.0	0.0	0.0	0.0	1.0	60.124...	1715.0	146.0	20.0	7.0	153.0	20.857...	2.0	0.0714...	19.943...
fnitsch...	CakePH...	2.0	9.0	1.0	0.0	22.0	0.0	0.0	133.23...	386.0	0.0	1.0	2.0	26.0	N/A	1.0	0.0	N/A
willscott	pintos	5.0	2.0	1.0	0.0	5.0	0.0	4.0	206.92...	0.0	5692.0	227.0	8.0	1323.0	14.520...	118.0	0.1611...	47.025...
xSmallD...	Rubik-...	1.0	0.0	1.0	0.0	1.0	0.0	0.0	204.24	1795.0	0.0	8.0	1.0	25.0	N/A	1.0	0.0	161.1
ngryman	jquery.f...	3.0	2.0	1.0	0.0	12.0	0.0	1.0	460.61...	15146.0	7826.0	18.0	1.0	88.0	1565.2	13.0	0.1067...	1237.5
cefn	allyourx	3.0	0.0	1.0	7.0	1.0	0.0	2.0	2010.8...	73151.0	55068.0	41.0	2.0	146.0	76.165...	41.0	0.5529...	140.12...
jcla1	HN2JSON	1.0	0.0	1.0	1.0	17.0	0.0	1.0	36.928...	323.0	26.0	2.0	1.0	28.0	26.0	1.0	0.0	46.142...
askedr...	todayls	1.0	0.0	1.0	1.0	3.0	0.0	1.0	54.694...	1216.0	13.0	30.0	1.0	95.0	13.0	5.0	0.0089...	N/A
ravster	lucifer	1.0	0.0	1.0	0.0	1.0	0.0	0.0	49.896...	255.0	0.0	4.0	1.0	29.0	N/A	4.0	0.0	N/A
StefanK...	django...	1.0	0.0	1.0	0.0	1.0	0.0	1.0	34.555...	303.0	51.0	0.0	1.0	18.0	17.0	5.0	0.3478...	29.4
benjam...	tent	2.0	1.0	1.0	0.0	6.0	0.0	2.0	3471.7...	39770.0	5224.0	7.0	4.0	36.0	61.458...	3.0	0.2697...	136.85...
danluu	ninety-...	1.0	1.0	1.0	0.0	3.0	0.0	2.0	62.047...	739.0	317.0	1.0	2.0	21.0	63.4	2.0	0.4696...	61.583...
BYK	pyresto	3.0	10.0	1.0	14.0	28.0	1.0	2.0	110.96...	1008.0	148.0	67.0	9.0	238.0	74.0	11.0	0.2	63.909...
maxfierke	gbamp...	1.0	0.0	1.0	0.0	0.0	0.0	0.0	1467.0	733.0	0.0	0.0	1.0	1.0	N/A	0.0	0.0	73.125
PluginIO	plugin-...	1.0	1.0	1.0	0.0	1.0	0.0	0.0	349.18...	2392.0	0.0	1.0	2.0	11.0	N/A	0.0	0.0	N/A
gtfierro	BAS	2.0	0.0	1.0	0.0	1.0	0.0	1.0	162.16...	6351.0	8.0	70.0	3.0	433.0	2.6666...	13.0	0.0	92.841...
FuzzyW...	MCNSA...	4.0	0.0	1.0	0.0	2.0	0.0	0.0	63.594...	3014.0	0.0	19.0	3.0	111.0	N/A	3.0	0.0	70.476...
chischa...	money-...	1.0	0.0	1.0	0.0	1.0	0.0	1.0	213.85...	736.0	74.0	1.0	2.0	21.0	24.666...	2.0	0.0	9.7837...
xSmallD...	Pascal-...	1.0	0.0	1.0	0.0	2.0	0.0	0.0	60.666...	131.0	0.0	2.0	1.0	3.0	N/A	0.0	0.0	65.5
gtfierro	IEEE-Sit...	1.0	17.0	1.0	0.0	7.0	0.0	2.0	181.27...	1326.0	85.0	59.0	11.0	183.0	7.7272...	2.0	0.0608...	17.611...
maxfierke	OpenSk...	2.0	0.0	1.0	16.0	11.0	0.0	1.0	160.57...	9916.0	918.0	55.0	1.0	429.0	61.2	17.0	0.1168...	139.2
billiob	twf	1.0	0.0	1.0	0.0	1.0	0.0	0.0	48.8125	359.0	0.0	1.0	1.0	16.0	N/A	0.0	0.0	N/A
philbooth	css-tra...	1.0	0.0	1.0	0.0	1.0	0.0	0.0	142.16	253.0	0.0	8.0	1.0	24.0	N/A	0.0	0.0	51.0

Abbildung 11.4: Fenster zum Anzeigen der Ergebnisse der Messung.

Für die Messung werden die gewählten Projekte von GitHub in ein temporäres lokales Verzeichnis geklont. Falls Prozess “git clone” zulange in “Cloning into” verweilen, den Prozess töten und Meldung davon geben. Danach werde auf dem Repository die ausgewählten Metriken gemessen. Ist dies fertiggestellt, wird das Repository wieder vom

[4] [Danial ’13]
[5] http://pmd.sourceforge.net

lokalen Dateisystem gelöscht. Nun wird das nächste Projekt geklont und so weiter, bis für alle Projekte die Metriken gemessen worden sind.

11.3.2 Darstellung der Ergebnisse

Wird der Button mit der Aufschrift “Measure” geklickt, so öffnet sich ein Fenster mit einer Tabelle der Ergebnisse (Abbildung 11.4). Die Tabelle füllt sich langsam mit gültigen Messwerten. Im Hintergrund werden die gewählten Projekte aus ihren Repositories geklont und die ausgesuchten Metriken darauf gemessen. Die Ergebnisse lassen sich für weitere Verarbeitung (beispielsweise in einer Tabellenkalkulation wie Excel) in eine CSV-Datei exportieren. Zusätzlich kann durch drücken der Schaltfläche für Visualisierungen der Visualisierungsdialog geöffnet werden.

Wurden bereits früher Messungen vorgenommen, so können die dabei exportierten Ergebnisse auch wieder importiert und innerhalb eines Fenster in einer Tabelle angezeigt werden (ebenfalls 11.4). Dieses Fenster enthält dann auch eine Schaltfläche zum Öffnen eines Visualisierungsdialogs.

11.3.3 Visualisierungen

Im Visualisierungsdialog kann der Nutzer einen Diagrammtyp für die Visualisierung der Messergebnisse auswählen. Die verschiedenen Diagramme wurden unter Verwendung der Java-Bibliothek “JFreeChart”[6] realisiert. Visualisierungen lassen sich nach Wunsch in eine Vektorgrafik (im PDF-Format) oder in eine Rastergrafik (im JPG-Format) auf das Dateisystem exportieren. Dafür öffnet sich beim Betätigen der Schaltfläche “Export” ein Dialog zum Speichern.

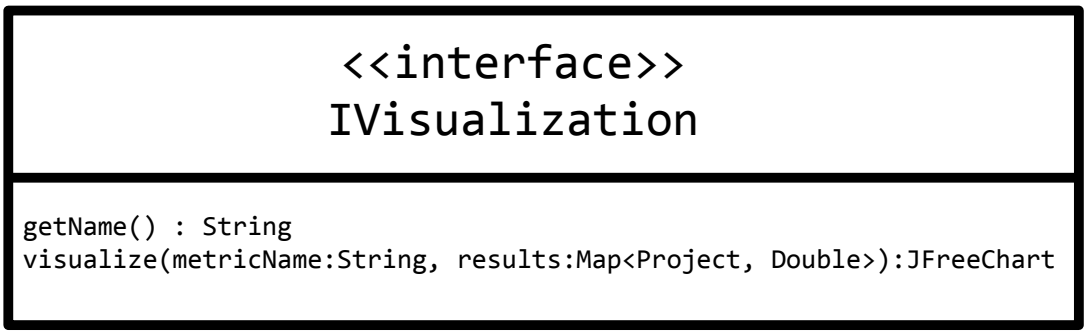


Abbildung 11.5: Schnittstelle für Visualisierungen.

Alle Visualisierungen implementieren die Schnittstelle `IVisualization` (Abbildung 11.5). Diese fordert für jede Visualisierung, dass sie analog zu Metriken einen Namen besitzen, um in der Oberfläche betitelt werden zu können. Als Hauptaufgabe müssen

[6] http://www.jfree.org/jfreechart/

Visualisierungsklassen eine Methode überschreiben, welche bei Eingabe einer Abbildung von Projekten auf ihr Messergebnis ein JFreeChart-Diagramm zurückliefern. Dieses Diagramm wird dann in einem Fenster dargestellt und kann nach Wunsch gespeichert werden.

Mitgelieferte Visualisierungen

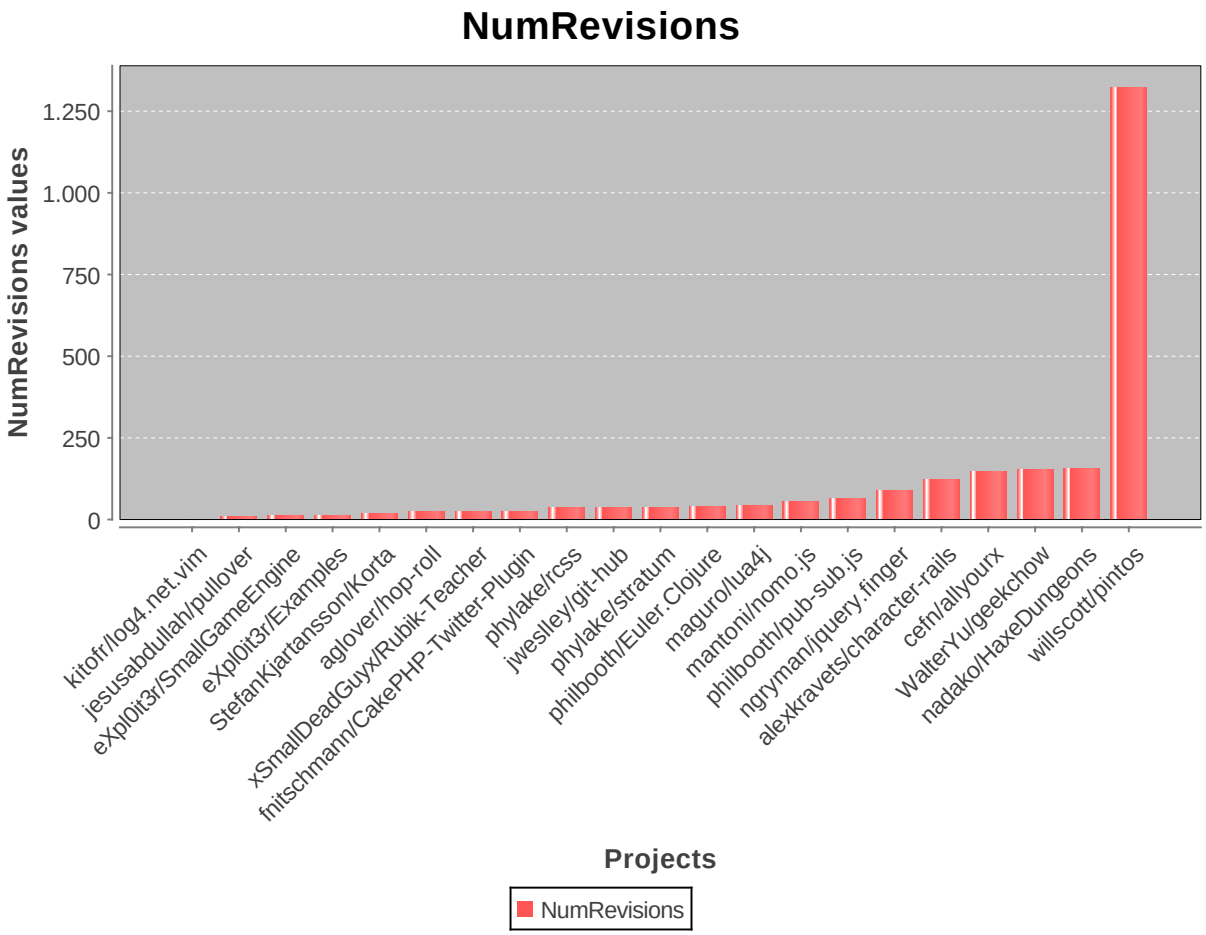


Abbildung 11.6: Visualisierung mit Balkendiagramm

Zum Abgabezeitpunkt dieser Arbeit werden folgende Visualisierungen für einzelne Metriken mitgeliefert:

- Balkendiagramm (Abbildung 11.6),
- Streudiagramm (Abbildung 11.7),
- Box-Whisker-Plot (Abbildung 11.8).

Zusätzlich lassen sich in einem Box-Whisker-Plot alle Metriken gemeinsam darstellen (Abbildung 11.9).

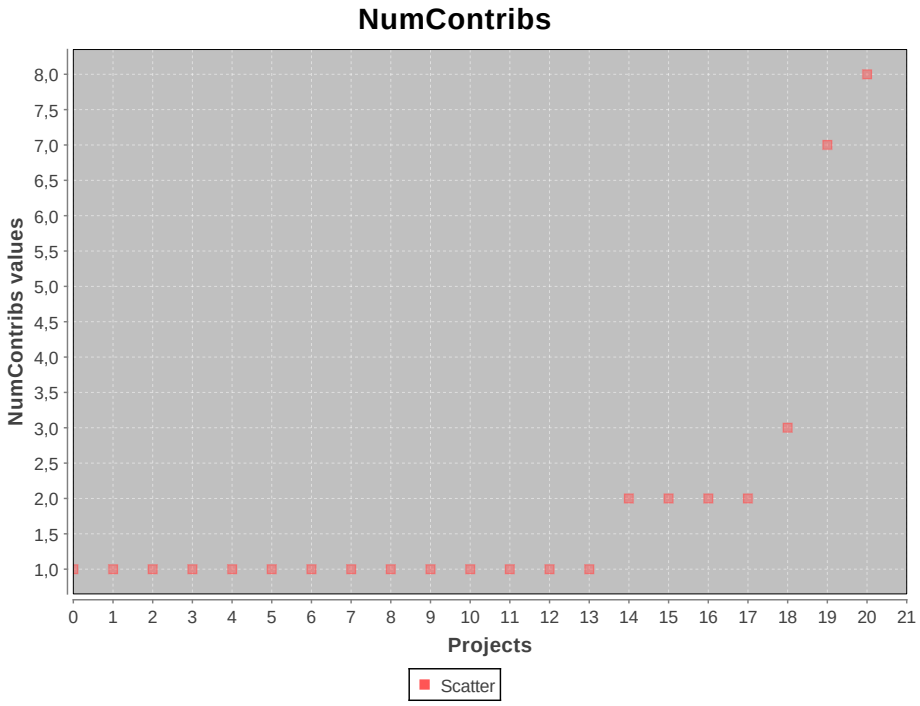


Abbildung 11.7: Visualisierung mit Streudiagramm

Im Balkendiagramm werden für eine ausgewählte Metrik jeweils für ein Projekt ein Balken dargestellt. Der Balken besitzt eine Höhe, die der Ausprägung der Metrik für das zugehörige Projekt entspricht. Dieses Projekt wird eingeblendet, wenn man mit dem Mauszeiger über diesen Balken geht.

Im Streudiagramm werden für eine ausgewählte Metrik für jedes Projekte kleine quadratische Punkte eingezeichnet. Ihre Lage auf der Ordinate entspricht der Ausprägung der Metrik für sie.

Im Box-Whisker-Plot wird für eine gewählte Metrik eine Box eingezeichnet. Dabei wird der Median der Ausprägungen der Projekte dieser Metrik als horizontal Strich eingezeichnet. Außreißer sind als “Schnurrhaare” (whiskers) ebenfalls als horizontalle Striche dargestellt.

Zusätzlich lassen sich in einem Box-Whisker-Plot alle Metriken kombiniert darstellen. Dann gibt es eine Box je Metrik.

11.3.4 Benchmark

Die dritte Option im Auswahlfenster öffnet das Benchmark-Fenster (Abbildung 11.10). Dieses Fenster dient der Beurteilung von Formeln für die Vorhersage des Projekts eines Nutzers mit minimalen, maximalen Testaufwand und Fehlerzahl.

Dort kann der Anwender von “JProjectInspector” die Einschätzungen aus der ersten,

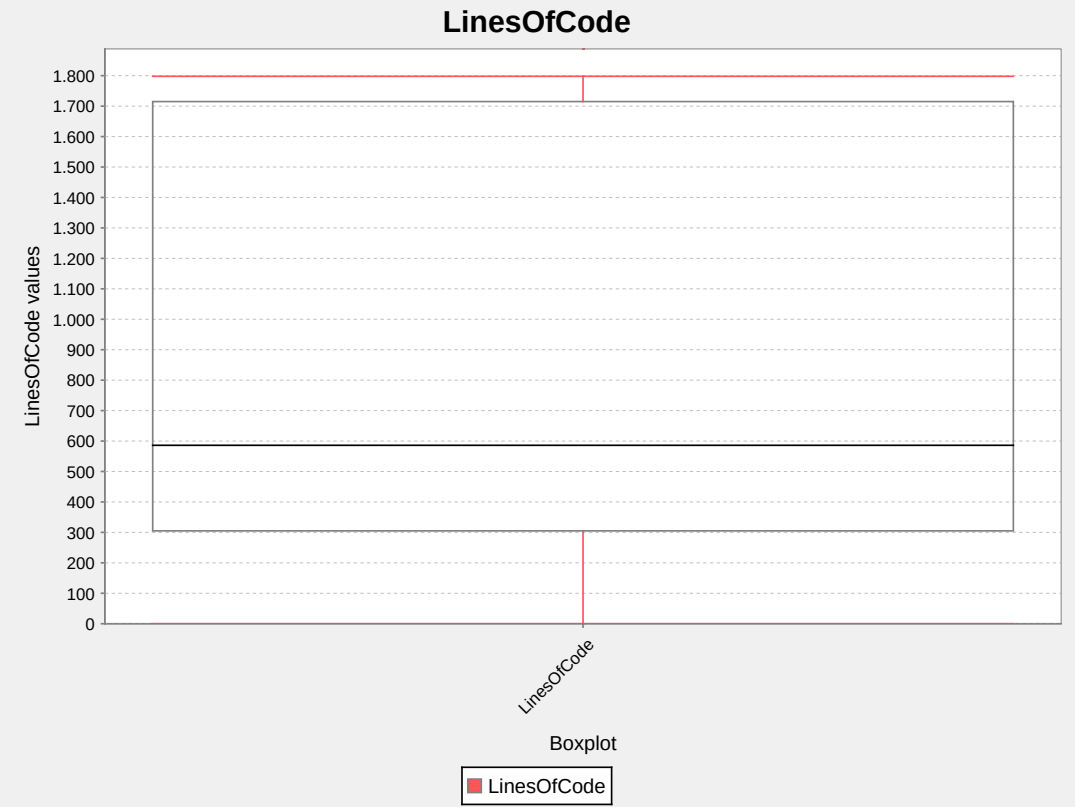


Abbildung 11.8: Visualisierung mit Boxplot

zweiten oder beiden Umfragen einlesen. Dazu sollte er Ergebnisse vorher getätigten Messungen von Metriken für die in den Antworten vorkommenden Projekte laden. Auf der dieser Arbeit beiligenden Compact Disc befinden sich dafür verwendbare Datensätze.[7]

Dann kann er im Textfeld “Prediction equation” verschiedenen gemessenen Metriken als Terme in einer Formel verknüpfen. Das Ziel ist, dass der Wert der Formel bei Projekten mit maximaler Einschätzung auch maximal ist und bei Projekten mit minimaler Einschätzung entsprechend minimal. Je nach eingestelltem Modus (Testaufwand oder Fehlerzahl) werden die Vorhersagen mit den Einschätzungen bezüglich des Testaufwands beziehungsweise der Fehlerzahl verglichen. Die Festlegung eines guten Vorhersagemodells für die intuitive relative Fehlerzahl (beziehungsweise Testaufwand) von Projekten eines GitHub-Nutzers wird dadurch zu einem Optimierungsproblem.

Dabei wurden alle Antworten, bei denen der Nutzer nicht für die Qualitätssicherung des Projekts zuständig ist, vorher ausgefiltert. Denn nur ein für die Qualitätssicherung zuständiger Verwalter, kann eine *begründete* Einschätzung von Testaufwand und Fehlerzahl abgeben.

[7] Das sind die Einschätzungen aus den Umfragen in `WeightedEstimatesUmfrage{1,2,Combined}.csv` sowie die zugehörigen Messergebnisse in `MetricResultsUmfrage{1,2,Combined}.csv`.

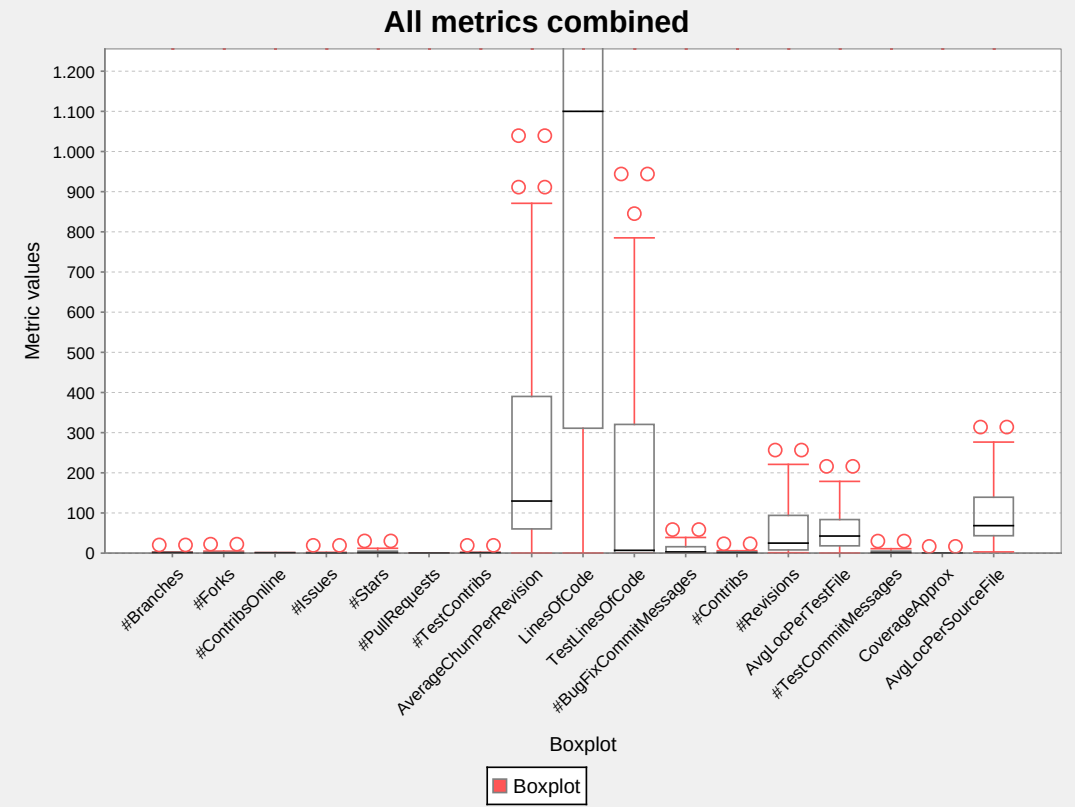


Abbildung 11.9: Visualisierung Boxplot Kombiniert

Weiterhin besitzen die Antworten eine Gewichtung. Sie ergibt sich aus dem Anteil der Buzzwords, die der Befragte im Fragebogen angekreuzt hat. Die Gewichtung fließt in die Anzahl korrekter Sortierungen gewichtet und gesamte Gewichtsumme mit ein. Dadurch gibt es neben der Bewertung, welche Nutzer neutral behandelt, auch eine Bewertung, welche Aussagen von Nutzern mit vermutlich hoher Testerfahrung stärker in die Bewertung eingehen lässt.

Das Benchmark-Fenster zeigt dann für die im dafür vorgesehenen Textfeld eingegebene Formel eine Bewertung an. Im unteren Bereich des Fensters befindet sich eine Tabelle. In der Tabelle enthält jeder Kandidat der Umfrage eine Zeile mit seinen Namen, geschätzte Ordnung zweier genannten Projekten und der Vorhersage der Reihenfolge der genannten Projekte durch die Formel. Falls eine falsche Ordnung vorhergesagt wurden, werden die Zellen der Vorhersage in der Tabelle rot markiert. Bei korrekter Vorhersage werden sie grün markiert.

Über die Schaltfläche “enumerate” können automatisch verschiedene Formeln mit gemeinsamer Struktur “ausprobiert” werden. Dazu muss die Formel Platzhalter enthalten. Diese Platzhalter haben die Form einer Variable aus einem Zeichen. Zum Beispiel “x”. Beim Drücken von “enumerate” werden dann in die Platzhalter Metriken eingesetzt. Es werden dabei alle Kombinationen ausprobiert, wobei keine Metrik doppelt vorkommt. Die Kombination mit der höchsten Bewertung wird dann in

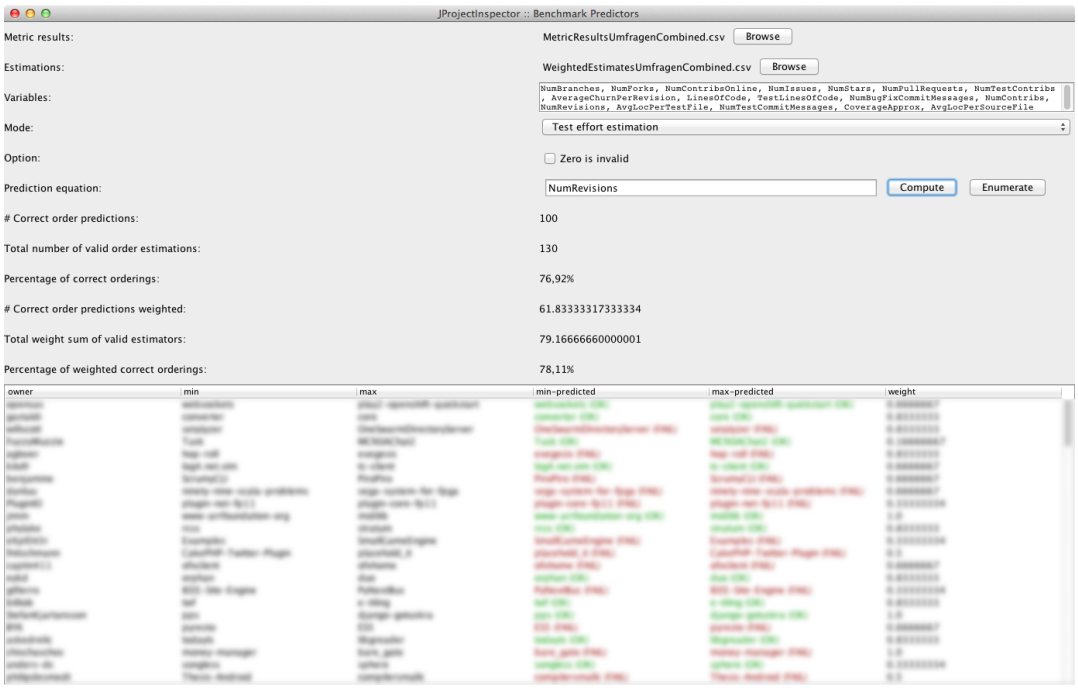


Abbildung 11.10: Fenster zur Bewertung von Vorhersageformeln

das Textfeld für die Formel gelegt. Beispielsweise für "$\frac{a}{b}$" würden durch "enumerate" die Formeln $\frac{\text{LinesOfCode}}{\text{TestLinesOfCode}}$, $\frac{\text{LinesOfCode}}{\text{NumContribs}}$ und so weiter ausprobiert. Dieses Vorgehen entspricht einer Brute-Force-Lösung des Optimierungsproblem durch vollständiges Absuchen eines Pfades mit gegebener Struktur. Die Struktur entstammt dem Aufbau der Formel mit den Platzhaltern.

11.4 Aufbau

Der Quellcode von *JProjectInspector* ist auf oberster Ebene in mehrere Packages aufgeilt: `evaluation`, `githubapi`, `gui`, `metrics`, `model`, `scrapers`, `tests` und `utilities`. Dabei befindet sich in `evaluation` Code für die Durchführung der Umfrage und der Evaluation. In `githubapi` befinden sich Hilfsklassen für die Erfassung von Daten über die Web-API von GitHub. In `gui` befinden sich alle Klassen für die grafische Oberfläche des Werkzeugs. In `metrics` liegen die Module zur Erfassung der verschiedenen Metriken. Scrapers enthält Hilfsklassen für die Extraktion von Daten aus GitHub mithilfe von Web-Scraping.

11.4.1 Tests

In `tests` befinden sich alle Tests. Wobei die Tests wiederum unterteilt wurden in Offline-Tests, Online-Tests und visuelle Tests. Die **Offline-Tests** sind alle in wenigen Millisekunden ausführbar und benötigen zur Ausführung höchsten lokale Daten vom Dateisystem. Die **Online-Tests** beziehen einige Daten von GitHub und benötigen daher deutlich mehr Zeit zur Ausführung. Sie wurden absichtlich in ein eigenes Paket gelegt, damit die unabhängigeren Offline-Tests häufiger ausgeführt werden können. Die Online-Tests werden weniger frequentiert ausgeführt.

Die Offline-Tests sind am stärksten isoliert. Die Online-Tests können bereits als Integrationstests angesehen werden, da sie das Zusammenspiel mit GitHub testen.

Am seltensten werden die **visuellen Tests** ausgeführt. Sie benutzen eine vom Autor entwickelte eigene Infrastruktur. Diese wurde auf das Testen von Oberflächencode ausgelegt. Die zu testenden Oberflächenkomponenten werden in ein speziell dafür vorgesehenes Fenster gelegt. Dann wird auf das Schließ-Ereignis dieses Fensters gewartet. Wenn dieses Ereignis eingetreten ist, wird dem Entwickler ein Dialog gezeigt, in dem er den vorangegangenen Test als erfolgreich oder gescheitert klassifiziert. Als Hilfestellung enthält jeder Test dazu eine kurze Beschreibung. Darin wird beschrieben, was zu sehen sein *sollte*.

Diese Tests sind semi-automatisch. Sie laufe zwar automatisch hintereinander ab, benötigen jedoch zur Beurteilung des Erfolgs eine manuelle Interaktion. Dies ist systematischer als komplett manuelle Begutachtung der verschiedenen grafischen Komponenten. Da alle Klassen der Oberflächen mit solchen visuellen Tests abgedeckt werden, können GUI-Fehler effektiver gefunden werden. Vorteil dieser Technik gegenüber besser automatisierten Lösungen ist, dass Tests bei einer Umgestaltung einer Oberfläche nicht verändert werden müssen. Denn das entscheidende Kriterium für den Erfolg eines Tests in der gewählten Lösung ist die Plausibilität der dargestellten Komponente für den ausführenden Entwickler. Würde man noch weitere Komponenten zur grafischen Oberfläche von *JProjectInspector* hinzufügen wäre jedoch ein Wechsel auf ein vollautomatisiertes Framework[8] für Oberflächentests sinnvoll. Manuelle Interaktionen bei Tests skalieren schlecht.

11.5 Erweiterbarkeit

Neben den mitgelieferten Metriken, welche sich im Paket `metrics` befinden, können durch Implementierung der Schnittstellen `IOfflineMetric` und `IOnlineMetric` weitere zusätzliche Metriken entwickelt werden (Abbildungen 11.11). Eine Klasse, welche eine der beiden Schnittstellen umsetzt, kann durch positionieren ihres kompilierten Bytecodes (.class-File) ins “plugin”-Verzeichnis in das Programm eingebunden werden. Durch Verwendung der Bibliothek “ASM”[9] werden Klassen in diesem Verzeichnis,

[8] Beispiele hierfür sind WindowTester Pro (`https://developers.google.com/java-dev-tools/wintester/html/`) und FEST (`http://code.google.com/p/fest/`).

[9] http://asm.ow2.org

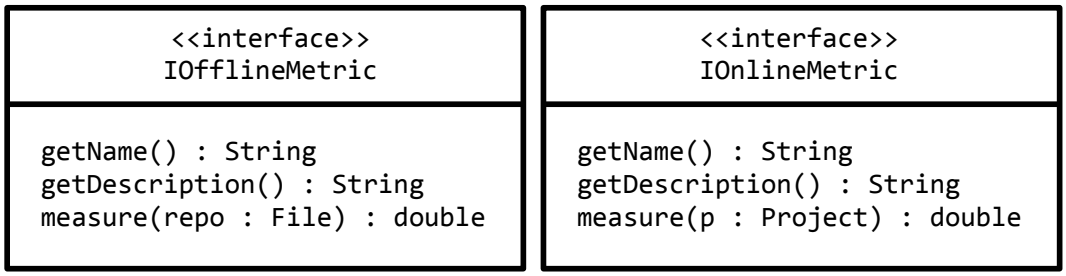


Abbildung 11.11: Schnittstellen für Offline- und Online-Metriken.

welche mindestens eine der entsprechenden Schnittstellen implementieren, automatisch erkannt. Sie werden danach von *JProjectInspector* über den Java-Classloader geladen und stehen dann für Messungen zur Verfügung.

Programmiersprache für Plugins ist Java. Die Plugins werden im Ordner `plugins` abgelegt. Das Werkzeug erkennt sie, falls sie eine Metrik-Schnittstelle implementieren. Die Erkennung wurde mithilfe des Bytecode-Manipulationsframework “ASM” umgesetzt.

Beide Schnittstellen erfordern, dass die neue Metrik-Klasse ihren Namen und einen beschreibenden Text durch Überschreiben von `getName` und `getDescription` als Zeichenkette liefert. Zusätzlich müssen Offline-Metriken für ein gegebenes Verzeichnis eines geklonten Repositories den Wert ihrer Metrik als Double (Fließkommazahl mit doppelter Genauigkeit) zurückliefern. Online-Metriken liefern ebenfalls eine Fließkommazahl zurück, bekommen jedoch als Eingabe die eindeutige Bezeichnung eines Repositories auf GitHub in Form eines (`owner`, `repo`)-Tupel. In beiden Fällen wird das Ergebnis der Messung durch Überschreiben der Methode `measure` für das Werkzeug zugänglich gemacht.

11.6 Anbindung an TestHub

Das Werkzeug *JProjectInspector* lässt sich für ein Ranking des Testbedarfs von externen Anwendungen nutzen. Insbesondere auch von TestHub. Dazu muss eine CSV-Datei mit den Spalten “owner,repo” angelegt werden. Dort sind als Zeilen die Tupel aus Verwalter und Repostiory der zu sortierenden Projekte eingetragen. Dann muss man *JProjectInspector* über den Befehl

```
java -jar JProjectInspector.jar Ranking
projectList=A.csv out=B.csv testingNeedEquation=C
```

aufrufen. Dabei ist

- *A.csv* die CSV-Datei mit der Liste der zu sortierenden Projekte.

- *B.csv* die Ausgabedatei, in der die Projekte mit Reihenfolge des absteigenden Testbedarfs geschrieben werden.
- *C* eine Formel, welche als Variablen die Namen von Metriken enthält, welche in JProjectInspector implementiert worden sind.

Nach dem Aufruf werden die in der Formel vorkommenden Metriken für die Projekte aus der Liste gemessen. Die Ergebnisse werden in die Formel eingesetzt und liefern so einen einzigen Fließkommawert für jedes Projekt. Dann werden die Projekte in absteigender Reihenfolge dieser Werte sortiert. In dieser Form werden sie in die Ausgabedatei geschrieben.

Ein beispielhafter Durchlauf lässt sich mit `exampleTestHubRun.bat/sh`[10] starten. In diesem Beispiel werden fünf kleine Projekte in absteigender Reihenfolge bezüglich des Testbedarfs sortiert. Der Testbedarf wurde dabei im Beispiel als über die Formel LinesOfCode definiert.

Jede beliebige externe Anwendung kann nun `JProjectInspector` benutzen, um ein Ranking von Projekten bezüglich einer Vernüpfung von Metriken zu ermitteln. Speziell kann auch die Formel für den Testbedarf verwendet werden. Zur Integration muss nur ein neuer Prozess für *JProjectInspector* mit den entsprechenden Argumenten erzeugt werden. Danach sollte die aufrufende Anwendung solange blockieren, bis der Prozess beendet ist. Dann lässt sich das Ranking aus der Ausgabe-CSV auslesen.

[10] `exampleTestHubRun.bat` ist die Batch-Datei für Windows. Systeme mit einem Unix-Betriebssystem können das Shell-Skript `exampleTestHubRun.sh` verwenden.

12 Umfragen

Zwei Umfragen sollen neben der Literaturrecherche eine empirische Basis für die Heuristik bilden.

12.1 Ziel

Zur Beurteilung eines Kandidaten für die Heuristik muss der Grad der Übereinstimmung mit den von GitHub-Nutzern intuitiv getroffenen Einschätzungen gemessen werden. Dies setzt voraus, dass diese intuitiven Einschätzungen aufgezeichnet worden sind. Das soll mithilfe von zwei Umfragen erreicht werden.

Zusätzlich soll die erste Umfrage klären, was GitHub-Nutzer unter Testbedarf verstehen und aus welchen Facetten er sich ihrer Meinung nach zusammensetzt.

12.2 Vorüberlegungen

Für die Umfragen wurden zwei verschiedene Varianten in Erwägung gezogen:

- Umfrage A, in welcher *außenstehende Freiwillige* fremde Projekte in Reihenfolge des absteigenden Testbedarfs sortieren sollen, oder
- Umfrage B, in welcher *Projekteigentümer* auf GitHub in einer E-Mail zu Aussagen über durch sie verwaltete aktive Projekte gebeten werden.

Es folgt eine kurze Abwägung der Vor- und Nachteile dieser beiden Umfragen zur Erfassung des intuitiven relativen Testbedarfs.

12.2.1 Umfrage A: Außenstehende Freiwillige

Eine Besonderheit von Umfrage A ist, dass die Probanden eine unvoreingenommene Sicht auf die zu beurteilenden Projekte haben. Ähnlich wie ein Code-Review im Idealfall nicht vom Autor des Codes getätigt werden sollte, könnte eine “Außensicht” auf das Projekt die Einschätzung des Testbedarfs objektiver machen.

Problematisch an diesem Experiment ist, dass das Auffinden von bereitwilligen Teilnehmern für diese Umfrage schwierig ist. Denn der Bearbeitungsaufwand ist aufgrund des notwendigen Einarbeiten in fremde Projekte hoch. Die erwartete Teilnehmerzahl wäre also vergleichsweise gering.

Auch liegen bei dieser Umfrage keine maschinenlesbaren Informationen über die Teilnehmer vor. Folglich wäre ein eventuell gewünschtes automatisiertes Filtern der Teilnehmer nach bestimmten Kriterien hier nicht möglich.

12.2.2 Umfrage B: Projekteigentümer

Vorteil dieses Experiment ist, dass alle Probanden sich bereits besonders gut mit den betrachteten Projekten auskennen. Es sind nämlich ihre eigenen Projekte. Als Autoren der Projekte sind ihnen die Anforderungen zudem bekannt.

Die Ermittlung der Probanden lässt sich automatisch durchführen. Ebenfalls kann man das Anschreiben der Probanden per E-Mail in diesem Experiment automatisieren. Dadurch lässt sich eine deutliche größere Anzahl von Teilnehmern realisieren. Dies erhöht die Repräsentativität der aus dem Experiment gewonnenen Aussagen. Das wiederum ist der Korrektheit der daraus abgeleiteten Heuristik zuträglich.

Weiterhin können die Fragebögen aufgrund des vorhandenen Vorwissen über die Projekte deutlich schneller bearbeitet werden. Dies erlaubt mehr Fragen als in Umfragevariante A zu stellen, ohne dabei den Gefragten zu überlasten. Zusätzlich lassen sich weitere Informationen über den Probanden auf GitHub gewinnen. Beispielsweise könnte man die Anzahl der bisherigen Commit-Nachrichten durch den Probanden mit dem Teilstring “test” ermitteln. Das wäre ein grober Stellvertreter für die Testerfahrung. Zusätzlich könnte man die aktuelle Aktivität und Beliebtheit (Followers, Stars) eines Probanden messen. Diese Informationen könnten für die Gewichtung der Aussagen eines Probanden hilfreich sein.

Zusätzlich kann eine emotionale Bindung an die von einem persönlich verwalteten Projekt den Anreiz zur Bearbeitung erhöhen. Möglicherweise hat der Entwickler sogar persönlich Interesse an einer automatischen Testbedarfs-Einschätzung seiner Projekte und daher auch ein Interesse an der Mithilfe bei der Verbesserung einer derartigen Heuristik.

Neben dem geringeren Einarbeitungsaufwand und der automatischen Erfassung potenzieller Teilnehmer erhöht die Individualisierung der Aufforderung zumindest die erwartete Zahl der teilnehmenden Probanden deutlich im Vergleich zu Umfrage A.

Ein Nachteil dieser Umfrage ist, dass eine emotionale Bindung an das Projekt die Beurteilung des Testbedarfs subjektiver machen könnte. Auch könnte die zeitliche Reihenfolge der vom Probanden genannten Projekte seine Aussagen beeinflussen. Denn Programmierer entwickeln sich im Laufe der Zeit durch Lernprozesse selbst weiter. Häufig haben sie eine kritische Beziehung zu Code, welchen sie vor langer Zeit geschrieben haben.

12.3 Auswahl der Teilnehmer

Die Teilnehmer sollen möglichst viel Wissen über ihre letzten Projekte besitzen, um eine genaue Aussage über den relativen Testbedarf treffen zu können. Daher werden als Probanden nur GitHub-Nutzer gewählt, welche Inhaber von mindestens drei Projekten[1], zu welchen sie jeweils innerhalb von weniger als etwa 1.5 Jahren (seit 2012) etwas beigetragen haben.

Die Ermittlung dieser Probanden geschah durch Abgreifen von den an Ereignissen in der GitHub-Timeline beteiligten Nutzern. Dadurch ist eine Zufälligkeit der Auswahl gewährleistet. Über reguläre Ausdrücke werden die Login-Namen sowie E-Mail und Klarname von Nutzern aus der Timeline extrahiert[2].

Von deren HTML-Profil auf GitHub werden Projekte, welche ab Jahr 2012 Aktivität vom Nutzer aufweisen, über weitere reguläre Ausdrücke entnommen. Nur Kandidaten, bei denen der vorherige Schritt mindestens drei Projekte liefert, werden übernommen. Der zur Ermittlung der Probanden benutzte Code lässt sich im Paket

```
org.andreschnabel.jprojectinspector.evaluation
```

von *JProjectInspector* finden.

Aus Datenschutzgründen können die in den Umfragen verwendeten Listen der Probanden in dieser Publikation nicht wiedergegeben werden.

Es waren 500 verschiedene GitHub-Nutzer für die erste Umfrage. Bei der zweiten Umfrage waren es 2000 verschiedene Nutzer. Dabei haben beide Listen keinen gemeinsamen Nutzer. Es wurde also kein Entwickler doppelt gefragt. Insgesamt verwalteten die 25000 befragten Entwickler eine Anzahl von ca. 40.000 Projekten. Durchschnittlich hatte ein Nutzer also etwa 16 Projekte.

12.4 Vorgehen

Es wurden mit einem speziell dafür angefertigten Programm 500 E-Mails an Nutzer aus der GitHub-Timeline versandt. Diese Nutzer mussten mindestens ein Projekt seit 2012 verwaltet haben, um sich zu qualifizieren. Die E-Mail enthält einen Link auf die Umfrage. Durch den Versuchsaufbau werden in die Umfrage nur bestimmte Nutzer einbezogen. Die Nutzer besitzen folgende Eigenschaften:

- Aktivität auf GitHub zum Zeitpunkt der Ermittlung der Nutzer aus der Timeline (1 Tag vor Umfrage). Anders können sie kein Ereignis erzeugen, welches in der Timeline auftaucht.

[1] Der Aufwand für den Probanden steigt mit der Anzahl der zu sortierenden Projekte. Aber auch die Aussagekraft für unsere Heuristik. Daher wurde ein Kompromiss mit der Wahl von drei Projekten getroffen. Praktisch an drei Projekten ist auch, dass sie durch Angabe des 1. und 3. Projekts in eine Reihenfolge gebracht werden können.

[2] Siehe auch Web-Scraping im Einführungskapitel 2

- Angabe einer korrekten E-Mail Adresse auf ihrem GitHub-Profil.
- Verwaltung eines Projekts mit letztem Änderungsdatum von 2012. Dies Kriterium wurde ungeschickt gewählt und soll bei der nächsten Umfrage wegfallen.
- Vertrauen von Links zu Google Docs. Nutzer, die sehr empfindlich gegenüber der Sicherheit sind, werden ausgeschlossen.

12.5 E-Mail

Es wird für jeden Teilnehmer eine E-Mail generiert. Die E-Mail ist an einigen Stellen auf Basis seiner Daten auf GitHub individualisiert. Sie enthält für jeden Probanden seinen GitHub-Nutzernamen und seinen dort angegebenen Klarnamen. Nachfolgend wird der Inhalt der E-Mail samt Platzhaltern, welche spezifisch für einen Probanden ausgefüllt sind, angegeben.

12.5.1 Inhalt der E-Mail

Hi there, [Name]!

I noticed your profile on GitHub ([Login]), and I am currently doing research related to testing on GitHub. It would help me a lot if you could spare 5 min and fill out this short Google-form:

https://docs.google.com/forms/d/...

Your answer will help my research a lot. Your answers will be treated confidentially and results from this survey will only ever be published in anonymized form.

If you're not interested, just ignore this email – there won't be any follow up mails to bug you.

Thanks for considering,
André

–
André Schnabel,
MSc. student at Software Engineering Group, Leibniz Universität Hannover, Germany
http://www.se.uni-hannover.de

Falls ein GitHub-Nutzer seinen Klarnamen angegeben hat, wird `[Name]` durch den Klarname ersetzt. Ansonsten wird der Nutzername eingesetzt. `[Link]` ist ein Link auf den im nächsten Abschnitt vorgestellten Fragebogen.

Diese E-Mail wurde für die erste Umfrage am 19. März 2013 um 16 Uhr an 500 Nutzer versandt. Für die zweite Umfrage wurde dieselbe E-Mail am 22. April 2013 um 22 Uhr an 2000 Kandidaten verschickt.

12.6 Fragebogen (Questionnaire)

Der Fragebogen wurde über Google Forms realisiert, da dies zum Zeitpunkt der Erstellung dieser Arbeit eine populäre Methode für Online-Umfragen ist und somit eine höhere Vertraulichkeit besitzt als eine eigene oder unbekanntere Lösung.

12.6.1 Inhalt des Fragebogens

Ein Stern kennzeichnet verpflichtende Fragen. Die nicht mit einem Stern ausgezeichneten Fragen sind optional. In der ersten Umfrage wurde der vollständige Fragebogen verwendet. Für die zweite Umfrage wurde die 2., 8., 9. und 10. Frage entfernt.

I am currently researching testing on GitHub. I will treat your answers confidentially and only use them to evaluate my models related to testing on GitHub. Thank you for taking the time to answer my questions.

1. Are you responsible for software quality assurance measures on GitHub projects you maintain?* [Yes/No]

2. Does low testing effort already put into a project or a high number of estimated bugs make you want to test a project more? * [Yes/No]

The following questions are about GitHub projects you currently maintain. When answering the following questions, please regard your projects only on a technical level and put personal affection to these projects or importance of them to you aside.

3. Which of your GitHub projects was tested the most?* [Textfeld] 4. Which of your GitHub projects was tested the least?* [Textfeld]

5. Which of your GitHub projects do you suspect to have the biggest number of undetected bugs?* [Textfeld] 6. Which of your GitHub projects do you suspect to have the smallest number of undetected bugs?* [Textfeld]

7. Please tick all buzz words that you can explain to a friend.* [Checkboxes: TDD, Unit testing, xUnit, Function tests, Mocking, IoC]

8. Besides testing effort already put into a project and estimated bug count, what else makes you want to test a project more? [Textfeld]

9. Do you have any comments? [Textfeld]

10. May we contact you for further questions?* [Yes/No]

Thank you for taking the time to answer our questions!

12.7 Rücklauf

Der Fragebogen der ersten Umfrage wurde von 95 der 500 (19%) angeschrieben Nutzer ausgefüllt. Der Rücklauf des Fragebogens der zweiten Umfrage betrug mit 261 von 2000 etwa 13%.

12.8 Qualitative Auswertung

Es folgen die Ergebnisse des manuellen Teils der Auswertung, der sich mit den textuellen Angaben der Probanden in unstrukturierter textueller Form befasst.

12.8.1 Motivation für das Testen

Warum will man ein Projekt weiter testen (außer wegen Fehlerzahl und bisherigem Testaufwand?). Diese Fragestellung soll zu Antworten führen, welche die bisherige Definition des Testbedarfs (Fehlerzahl & Testaufwand prüfen und eventuell verbessern sollen.

Es folgt die Auflistung der genannten Anreize für weiteres Testen in Kategorien eingeteilt.

Fehler

- Fehlerberichte von Nutzern[3],
- selbst erkannte Fehler,
- Erhöhung der Korrektheit,

Bisheriger Testaufwand

- geringes Vertrauen in geschrieben Code,
- geringe Testabdeckung,
- Kundeneindruck von Qualität und Stabilität,

Lernen

- Erlernen von Modultests,
- Erlernen von TDD,
- Einarbeitung in fremden Code,

[3] Aber auch automatisierte Fehlerberichte wie Protokolle von Google Analytics.

Sozialer Aspekt

- Absicherung bei vielen Beitragenden (die nicht alle testen),
- Hohe Anzahl Nutzer, die Projekt koordinieren,
- Vereinfachung des Mergen/Beurteilen von Pull-Requests bei vielen Nutzern,
- Nutzerzahl bedeutet auch mehr wütende Leute, wenn deine Software fehlerbehaftet ist,
- Imaginäre Internet-Punkte,

Produktivcode

- Verbesserung von Komplexität und Struktur des Codes,
- Verbesserung der Architektur,
- Erhöhung der Wartbarkeit,
- Um “Qualitätssoftware” zu veröffentlichen,
- zur Dokumentation,
- Reifen und Verbessern von Code,

Regressionen

- Sicherstellung, ob bei Commit Fehler eingeführt,
- Einfügen neuer Features vereinfachen,
- Vermeidung zukünftiger Bugs,
- Vereinfachung zukünftiger Veränderungen am Projekt.
- Für sicheres Refaktorisieren,

Anforderungen

- Verwendungszweck,
- Sicherheitsanforderungen,
- Öffentlichkeit und Interesse am Code,

Diverses

- Warnungen von Werkzeugen zur statischen Analyse[4],
- Integration mit anderen Technologien/Werkzeugen,

[4] Erwähnt wurden rails_best_practices (`https://github.com/railsbp/rails_best_practices`) und reek (`https://github.com/troessner/reek/wiki`).

- Zur Sicherung der Performanz und Qualität,
- Performanz erhöhen,
- Zeit zum Testen vorhanden,
- Erhöhung der Professionalität.

12.8.2 Kommentare

Der Fragebogen der ersten Umfrage enthielt auch einen Bereich für Kommentare. Zwei häufige Kategorien von Kommentaren waren:

Nutzen von Tests

- Tests schreiben ist einfacher als Debuggen,
- Flüchtigkeit beim Testen kommen auf einen selbst oder andere Nutzer des Projekts zurück,
- TDD erzeugt stabilere SW als andere Ansätze,
- Tests helfen bei Warten von Anwendung in vieler Hinsicht und sind guter Spiegel für Code.
- wenn Code geringe Testbarkeit hat, ist wahrscheinlich was falsch dran.
- Zeit beim Testen bekommt man vielfach wieder zurück,
- einfache Refaktorisierung und Stabilität.
- Testen ist ein großartiges Werkzeug, das meistens ausgelassen wird.

Einschränkungen von Tests und Testbarkeit

- Testaufwand in Projekte mit geringer Beliebtheit zu stecken lohnt sich nicht.
- Programme auf Mikrokontroller-HW lässt sich schwier testen,
- Projektgröße ist großer Faktor in Fehlerzahl. Große Projekte haben unendliche Mengen von Ausführungspfaden, welche die Fehlerzahl groß beeinflussen. Selbst bei guten Tests.
- TDD braucht Disziplin, zuerst *wirklich* scheiternde Tests schreiben,
- Testen ist Gewohnheit,
- Der Prozess des Testens sollte einfacher sein.

12.8.3 Folgeumfragen

Eventuell “inlinen” des Formulars in die Aufforderungsmail der Umfrage, um sicherheitsbewusste Nutzer miteinzubeziehen.

In zukünftigen Umfragen sollte auch angegeben werden, in welchem Format die Projekte genannt werden. Einige gaben nur den Namen des Repositories an, andere einen vollständigen Link dazu. Dieser Link enthält auch den Namen des Verwalters (Probanden). Wären Einheitlich alle Repositories als vollständige Links angegeben worden, müsste der Verwalter nicht anhand der Antworten “geraten” werden.

12.9 Quantitative Auswertung

Da die Antworten der Umfrage zu den Projekten[5] lediglich die Qualität einer Ordinalskala besitzen, können keine Korrelationskoeffizienten bestimmt werden. Diese verlangen nämlich mindestens intervallskalierte Zufallsvariablen. In der zweiten Umfrage besteht die Möglichkeit diese Einschränkung aufzuheben. Dies bedarf aber einer feineren Beurteilung der Größen durch den Kandidaten der Umfrage. Es ist zweifelhaft, ob eine intuitive Schätzung mit dieser Genauigkeit überhaupt sinnvoll ist.

12.9.1 Säuberung der Ergebnisse

Das Formular der ersten Umfrage wurde insgesamt 94 mal ausgefüllt.

Die Antworten der Kandidaten auf den Fragebogen werden von Google Forms automatisch in eine Ergebnistabelle von Google Docs eingetragen. Diese Tabelle wurde in eine mit CSV formatierte Textdatei exportiert. Aus dieser Datei wurden alle 2-Tupel von Projekten mit geringstem und höchsten Testaufwand beziehungsweise vermuteter Fehlerzahl für die weitere Verarbeitung ausgelesen. Dies geschah mit einem speziell für diesen Zweck verfassten Programm. Die Verarbeitungsschritte transformieren zwischen XML und CSV.

Da in der Umfrage die Nutzer nicht explizit zur Angabe ihres Nutzernamens aufgefordert worden sind, musste die Zuordnung der genannten Projekte zu den Nutzern heuristisch erfolgen. Zuerst wurden für jeden Nutzer aus der Kandidatenliste jeweils alle seine Projekte von der GitHub-Webseite gescraped[6]. Dann wird für jedes 4-Tupel aus Projektnamen nach einem Nutzer gesucht, der diese als Teilmenge seiner Projekte enthält. Nun sind die Projekte eindeutig durch Nutzername und Namen des Repositories bestimmt.

Nutzer, welche die Projekte als vollständigen Link auf das Repository of GitHub angegeben haben, konnten direkt ermittelt werden. Der Name des verwaltenden Nutzers ist nämlich in so einem Link enthalten.

[5] Projekt mit geschätztem $\{\text{niedrigsten, höchsten}\} \times \{\text{Testaufwand, Fehlerzahl}\}$.
[6] Siehe 2.2.10 für die Definition von Web-Scraping

12.9.2 Bestimmung von Metrik-Werten

Jetzt wird jedes in der Umfrage genannte Projekt von GitHub geklont[7]. Dann wird es auf den Zustand zum Umfragezeitpunkt zurückgesetzt[8]. Dann wurden für das geklonte Projekt die Werte der sprachunabhängigen Metriken von *JProjectInspector* gemessen.

12.9.3 Sprachverteilung

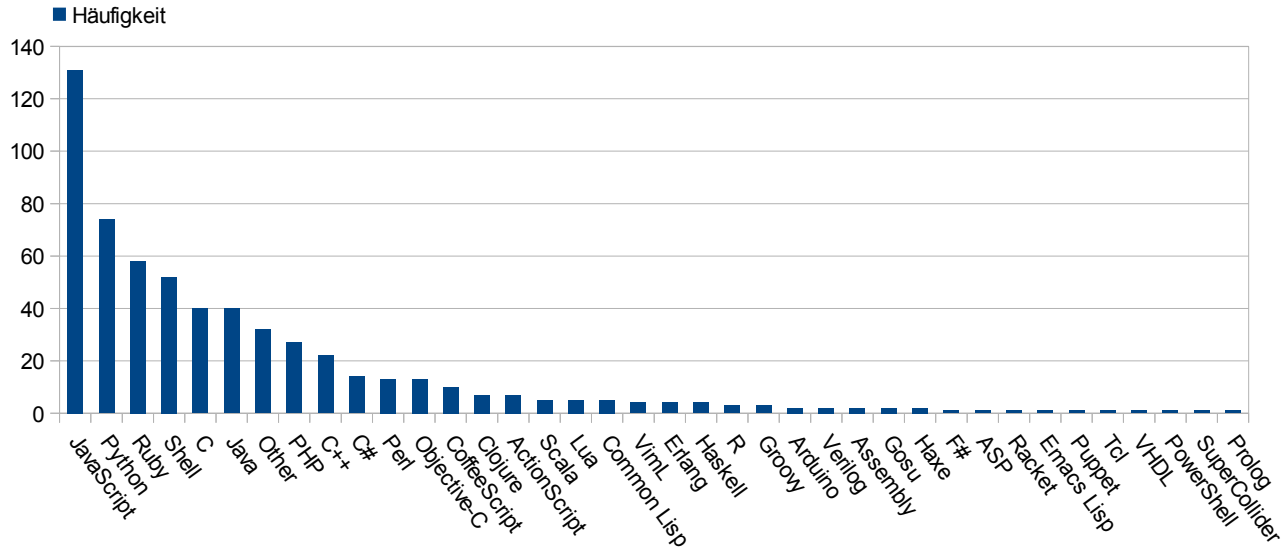


Abbildung 12.1: Sprachverteilung der in beiden Umfragen angegebenen Projekte.

Von den in beiden Umfragen genannten Projekten wurde die Sprachverteilung ermittelt. Dies geschah mithilfe der `LanguageDistributionRunner`-Klasse aus dem Paket

```
org.andreschnabel.jprojectinspector.evaluation.runners
```

Die Verteilung ist in Abbildung 12.1 dargestellt. Die Verteilung hat die Form einer Potenzfunktion[9].

[7] Klonen (cloning) bezeichnet das Erstellen einer identischen lokalen Kopie eines entfernten Repositories. Siehe auch 2.3.2 im Grundlagenkapitel.

[8] Dies geschieht über das zurücksetzen des Repositories mit `git reset --hard` auf den SHA1-Wert des neusten Commit zum gewünschten Zeitpunkt. An diesen wiederum gelangt man mit `git log -1 --until=YYYY-mm-dd`. Eine Umsetzung beider Aktionen befindet sich in der Klasse `GitHelpers` im *JProjectInspector*-Projekt.

[9] Das ist ähnlich zur Pareto-Verteilung. Die meisten Projekte sind in ein paar wenigen Sprachen verfasst und die restlichen Projekte bilden den “long tail”.

12.9.4 Differenzen der Metriken

Seien t und T die Projekte mit minimalem beziehungsweise maximalen bisherigen Testaufwand. Zudem seien b und B die Projekte mit minimaler respektive maximaler Fehlerzahl. Dann sollen für jede gewählte Metrik die Differenzen $\Delta\text{Fehler} = M(B) - M(b)$ und $\Delta\text{Testaufwand} = M(T) - M(t)$ berechnet werden. Liegen diese für alle Projekte deutlich < 0 oder > 0 kann dies als grober Hinweis auf eine negative respektive positive Korrelation gedeutet werden.

Jetzt wurden für die erste Umfrage die oben genannten Deltas berechnet.

Metrik	μ	Metrik	μ
tdloc	5400,19	bdloc	1895,81
tdtloc	379,2	bdtloc	-166,6
tdcontribs	1,81	bdcontribs	-0,11
tdcommits	63,45	bdcommits	34,83
tdissues	0,39	bdissues	0,81

Tabelle 12.1: Aggregierte Werte der Differenzen.

12.9.5 Beschränkte Felder

Die absoluten Häufigkeiten der gewählten Schlagwörter ist in Tabelle 12.2 dargestellt. Die relativen Anteil wurden in Abbildung 12.2 visualisiert.

Buzzword	Häufigkeit in 1. Umfrage /#Teilnehmer	Häufigkeit in 2. Umfrage /#Teilnehmer	$\sum$ Häufigkeit /$\sum$#Teilnehmer
Unit testing	91/95	250/261	341/356
TDD	78/95	200/261	278/356
Mocking	66/95	172/261	238/356
Function tests	45/95	140/261	185/356
IoC	35/95	85/261	120/356
xUnit	30/95	67/261	97/356

Tabelle 12.2: Häufigkeiten der Buzzwords in den Umfragen.

Auf die Frage zur Verantwortlichkeit für Qualitätssicherung im Projekt[10] antworteten in der ersten Umfrage 81 der Befragten mit Ja und die restlichen 12 mit Nein. In der zweiten Umfrage gaben 213 Befragte die Antwort Ja und 48 Befragte die Antwort nein.

Zur Frage, ob geringer bisheriger Testaufwand und eine hohe geschätzte Fehlerzahl zu mehr Testen motiviert[11], gab es 74 mal die Antwort Ja und 19 mal die Antwort Nein.

[10] "Are you responsible for software quality assurance measures on GitHub projects you maintain?"

[11] "Does low testing effort already put into a project or a high number of estimated bugs make you want

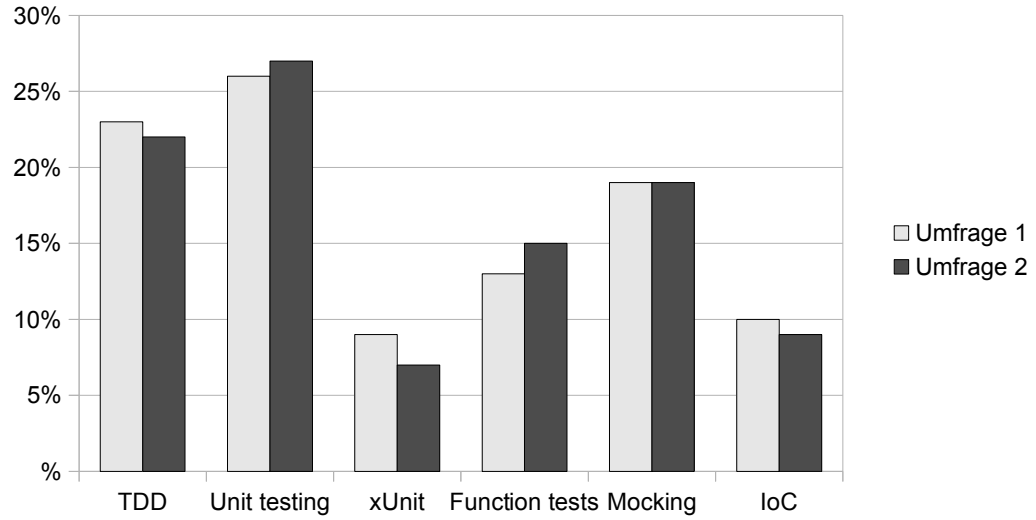


Abbildung 12.2: Anteil der Befragten je Umfrage, die angaben Schlagwort erklären zu können, abhängig vom Schlagwort.

12.9.6 Auswertung mit Excel

Die von *JProjectInspector* exportierten CSV-Tabellen können problemlos in Excel importiert werden. Dies erlaubt das einfache Berechnen von Mittelwert und Median für bestimmte Metriken bestimmter Projekte. Weiterhin lassen sich mit Excel besonders leicht Diagramme erzeugen, welche die in den Tabellen enthaltenen Werte visualisieren.

In der nachfolgenden Tabelle 12.3 befindet sich eine Zusammenfassung der aggregierten Werten aus den Ergebnissen ersten Umfrage. Hier wurden keine Deltas je Nutzer betrachtet sondern alle Ausprägungen der Metriken je Projekt und dies Nutzerübergreifend.

Metrik	μ	Median
LOC	7350,21	841
Test LOC	472,51	0
#Contributors	2,5	1
#Commits	93,73	27
#Issues	1,27	0

Tabelle 12.3: Aggregierte Werte der Metriken in der 1. Umfrage.

to test a project more?”

12.9.7 Visualisierungen

Mit der Java-Visualisierungsbibliothek JFreeChart werden die Ergebnistabellen in weiteren Diagrammen dargestellt. Diese Form der Visualisierung ist technisch besonders gut für das in dieser Arbeit zu entwickelnde Visualisierungs-Tool geeignet. Im Gegensatz zu der Visualisierung mit Diagrammen in Excel ist zum betrachten dieser Diagramme lediglich das mitgelieferte Werkzeug *JProjectInspector* nötig. Weiterhin ist die Verwendung einer Visualisierungsbibliothek programmatisch einfacher als die Generierung von Excel-Tabellen mit Diagrammen. Excel kann zwar neben dem proprietären Binärformat auch Tabellen in einer XML-Darstellung einlesen. Doch ist diese wie für XML üblich sehr umständlich und mit sehr viel unnötiger Redundanz versehen.

12.9.8 Benchmark

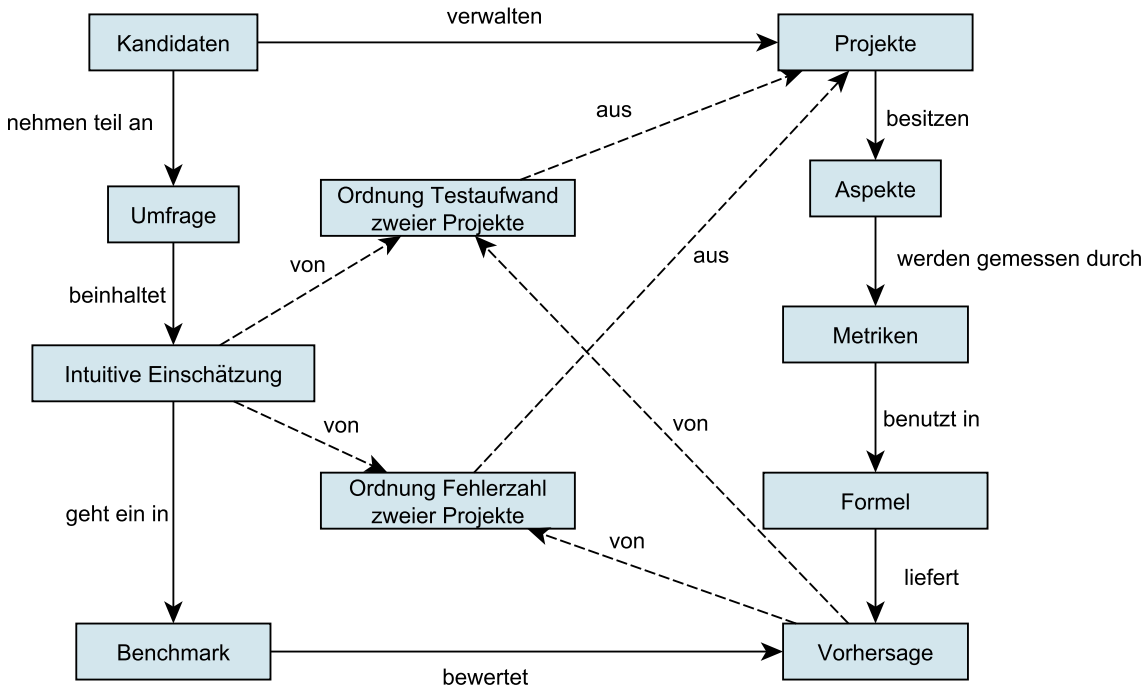


Abbildung 12.3: Gerichteter Graph mit beschrifteten Kanten zur Verdeutlichung des Benchmarks.

Ein Programm zur Beurteilung von Kandidaten für die Heuristik wurde entwickelt. Es bekommt jeweils eine Formel für die Vorhersage des Testaufwands und der Fehlerzahl auf Basis der Metrik-Werte eines Projekts. Dann wendet es diese Formel auf die Projekte des Nutzers eines Umfrageergebnisses an, welche genannt worden sind, und prüft ob eines oder zwei der vom Nutzer angegebenen Projekte korrekt vorhergesagt werden konnte. Das Beurteilungsmaß ist dann das Verhältnis der korrekt vorhergesagten Ordnung zweier Projekte dividiert durch die Gesamtzahl der Projektpaare, was

der Anzahl der antwortenden Nutzer entspricht. Es wurden nur für in Antworten zum Fragebogen genannte Projekte die Metriken gemessen.

Dadurch kann die Festlegung der Heuristik als eine Suche beziehungsweise ein Optimierungsproblem betrachtet werden, für das es gilt einen möglichst guten Kandidaten zu finden. Idealerweise sollte die Formel eine 100%ige Vorhersage liefern. Aufgrund von psychologischen Einflüssen bei der Beurteilung des Testaufwands und der Fehlerzahl durch die Nutzer, ist dies jedoch praktisch nicht umsetzbar. Da es sich hierbei um eine Heuristik handelt sind auch einige falsche Vorhersagen akzeptabel. Die Vorhersage sollte aber eine signifikant bessere Beurteilung erzielen als uninformiertes Raten.

Wenn ein Nutzer ein einziges Projekt als Projekt mit minimalem und maximalem Testaufwand angegeben hat, so soll dieser beim Benchmark nicht beachtet werden, da keine Metrik auf einem einzigen Projekt unterschiedlichen Werte liefern kann (und auch muss). Bei Fehlerzahl wird analog vorgegangen. Weiterhin wurden die Kandidaten so gewählt, dass sie mindestens drei Projekte besitzen. Dass davon ein einziges maximale und minimale Fehlerzahl beziehungsweise Testaufwand hat ist nicht plausibel, da ansonsten alle weiteren Projekte genau dieselbe Fehlerzahl und Testaufwand besitzen müssten.

Uninformiertes Raten würde einer rein zufälligen Ordnung der genannten zwei Projekte entsprechen. Bei Annahme einer Gleichverteilung für die rein zufällige Festlegung der Ordnung folgt, dass die zu übertreffende Wahrscheinlichkeit für die Korrektheit bei 50% liegt.

Zusammenfassend werden die einzelnen Bestandteile des Benchmarks und ihre Beziehungen in Abbildung 12.3 dargestellt.

Die für die Auswertung verfassten und benutzten Funktionen finden sich ebenfalls im *JProjectInspector*-Programm. Nämlich in der Klasse `CandidateTapper` im Paket

`org.andreschnabel.jprojectinspector.evaluation`.

Die Ergebnisse des Benchmarks für einige Formel-Kandidaten befinden sich in Tabellen 12.4 und 12.5.

Testaufwand	$\frac{\#\textbf{Korrekt}}{\#\textbf{Anwendbar}}$	**%**
$\frac{\text{TestLinesOfCode}+\text{NumRevisions}}{(\text{AverageChurnPerRevision}+1)}$	114/129	88.37%
TestLinesOfCode	84/129	65.12%
AvgLocPerTestFile	28/43	65.12%
$\frac{\text{TestLinesOfCode}}{\text{LinesOfCode}}$	80/124	64.52%

Tabelle 12.4: Beurteilung von Formel-Kandidaten für Testaufwand durch Benchmark.

Fehlerzahl	$\frac{\#\textbf{Korrekt}}{\#\textbf{Anwendbar}}$	**%**
$\frac{\text{NumIssues+AverageChurnPerRevision}}{\text{AvgLocPerTestFile+NumTestCommitMessages}+1}$	34/46	73.91%
AvgLocPerSourceFile	56/100	56.00%
LinesOfCode	71/130	54.62%
$\frac{\text{NumStars}}{\text{AvgLocPerTestFile}+1}$	24/46	52.17%

Tabelle 12.5: Beurteilung von Formel-Kandidaten für Fehlerzahl durch Benchmark.

In beiden Fällen (Testaufwands-Schätzung, Fehlerzahl-Schätzung) hat der erste Kandidat gewonnen. Die anwendbaren Projekte sind diese, wo keine der Terme einen ungültigen “N/A”-Wert aufweißt. Dieser taucht bspw. auf, wenn keine Testdateien gefunden werden konnten.

13 Heuristik

In diesem Kapitel werden die Ergebnisse der Auswertung der beiden Umfragen mit vorherigen Überlegungen zum Konzept des Testbedarfs zu einem Maß für den Testbedarf kombiniert.

Am Ende des Umfragekapitels wurden zwei Formeln zur Vorhersage der intuitiven Einschätzung des relativen Testaufwands von Projekten eines Nutzers und der geschätzten Fehlerzahl der Projekte eines Nutzers angegeben. Die Kombination dieser beiden Formeln zum festgelegten Testbedarfswert wurde dabei mit den hier vorgenommenen Überlegungen und den Antworten der ersten Umfrage begründet. Falls ein Umfeld in dem die Heuristik benutzt werden soll aufgrund spezifischer Anforderungen eine abgewandelte Defintion des Testbedarfs erfordert, wäre es möglich die Fehlerzahl und den Testaufwand in anderer Form zum Testbedarf zu kombinieren. Diese beiden Formeln sind als Vorhersagemodelle durch die beiden Umfragen empirisch begründet.

13.1 Modelle für Fehlerzahl und Testaufwand

Von den besten Kandidaten aus dem Benchmark zwei ausgewählt.

1. Maß für Fehlerzahl: $=_{\text{Def.}} \frac{\text{NumIssues+AverageChurnPerRevision}}{\text{AvgLocPerTestFile+NumTestCommitMessages+1}}$
2. Maß für Testaufwand: $=_{\text{Def.}} \frac{\text{TestLinesOfCode+NumRevisions}}{\text{(AverageChurnPerRevision+1)}}$

Das Maß für die Fehlerzahl sagte 73.91% der Sortierungen zweier Projekte eines Nutzers bezüglich der Fehlerzahl korrekt voraus. Das Maß für den Testaufwand sagte 88.37% der Sortierungen zweier Projekte eines Nutzers bezüglich des Testaufwands korrekt voraus.

13.2 Maß für Testbedarf

Basierend auf den in der Literatur gefundenen Resultaten zu Modellen für die Fehlervorhersage, dem Qualitätsmodell vom GQM-Ansatz sowie den Ergebnissen und der Auswertung der beiden Umfrage definieren wir nun ein Maß. Dieses Maß liefert dann eine Heuristik für den Testbedarf.

Der Testbedarf wird nun motiviert durch Abbildung 7.1 zusammengesetzt aus Fehlerzahl (Stellvertreter für Fehlerkosten) und Testaufwand (Stellvertreter für Testkosten).

In der Abbildung ergeben sich die Gesamtkosten aus Fehlerkosten und Fehlerbehebungskosten (Testkosten). Der Testbedarf ist positiv korreliert mit der Fehlerzahl und negativ korreliert mit den Fehlerbehebungskosten. Daher wurde in Kapitel 7 bereits als Formel für den Testbedarf vorgeschlagen:

$$\text{Testbedarf} =_{\text{Def.}} \frac{\text{Fehlerzahl}}{\text{Testaufwand}}.$$

Einsetzen ergibt als konkretes Maß für den Testbedarf:

$$\frac{(\text{NumIssues} + \text{AverageChurnPerRevision})(1 + \text{AverageChurnPerRevision})}{(\text{AvgLocPerTestFile} + \text{NumTestCommitMessages} + 1)(\text{TestLinesOfCode} + \text{NumRevisions})}$$

Die Terme der Formel lassen sich für alle Projekte auf einer Social Coding Site ermitteln.

Die Heuristik ist dann eine Menge von Projekten auf einre SCS in Reihenfolge des absteigenden Werts dieser Formel zu testen.

14 Fazit und Ausblick

Es folgt nun im letzten Kapitel eine kritische Reflexion. Mögliche Anknüpfungspunkte für zukünftige Arbeiten werden skizziert. Abschließend gibt es eine rückblickende Zusammenfassung der wichtigsten Ergebnisse.

14.1 Kritische Würdigung

14.1.1 Heuristik

Die entwickelte Heuristik besitzt lediglich eine Ordinalskala. Von den 356 Antworten beider Umfragen konnten für das Benchmark lediglich 135 verwendet werden. Dies kann zwei Ursachen haben. Der zu den 2-4 genannten Projekten zugehörige Nutzer konnte nicht ermittelt werden. Eines der genannten Projekte fehlte in der Projektliste der angeschrieben Nutzer. Hilfreich waren hierbei Nutzer, welche ihr Projekt als (Owner,Repo)-Tupel oder mit vollständigem Link angegeben haben.

14.1.2 Umsetzung

Die Umsetzung wurde zuerst in der Programmiersprache Ruby begonnen. Diese besitzt eine hohe Verbreitung auf GitHub. Zusätzlich ist die Ruby-Community im Zusammenhang mit Rails[1] in [Pham et al. '13] besonders positiv bezüglich ihrer Testkultur aufgefallen. In der universitären Ausbildung hat jedoch Java eine weit höhere Verbreitung. Daher wurde ein Wechsel auf Java vollzogen. Dadurch sollen eventuell notwendige Wartungsarbeiten am Werkzeug durch Studenten erleichtert werden.

Im Nachhinein stellte sich jedoch die Wahl der Programmiersprache Java für die Umsetzung als problematisch heraus. Die Aufgabenstellung erforderte sehr häufige Verarbeitung von Listen. Um hierbei die Absichten einer Listenoperation besonders explizit im Code auszudrücken, wurde auf Funktionen höherer Ordnung wie `map`, `filter` und `reduce` zurückgegriffen. Diese Funktionen wurden als Methoden mit generischen Schnittstellen als Parameter implementiert. Die Verwendung dieser Methoden erfordert jedoch eine sehr umständliche Syntax. Denn Java verfügt bisher noch über keine Syntax für Lambda-Ausdrücke.[2]

[1] Ein Framework für Webservices. (`http://rubyonrails.org`)

[2] Mehr über Funktionen höherer Ordnung und die Problematik ihrer Verwendung in Java findet sich im fünften Kapitel von [Sestoft '12] sowie in [Six '07].

Eine bessere Wahl wäre die Programmiersprache C# gewesen. Sie ist syntaktisch sehr ähnlich zu Java und sollte deshalb von Studenten mit Vorkenntnissen in Java schnell zu erlernen sein. Mit LINQ-Anfragen besitzt C# eine eingebettete domänenspezifische Sprache für die Verarbeitung von Daten (speziell auch Listen). Zudem verfügt C# über eine kompakte Syntax für Lambda-Ausdrücke.

Die vom Programm mitgelieferten Java-spezifischen Metriken hätten entfallen können. Sie spielten bei der Entwicklung der Heuristik keine Rolle.

14.2 Ausblick

14.2.1 Coverage und kontinuierliche Integration

Momentan führen Dienste zur kontinuierlichen Integration[3] automatisch die Test-Suites von Projekten bei Änderungen aus. Die Ergebnisse des Testlaufs werden in einem öffentlichen Protokoll wiedergegeben. In Zukunft wäre wünschenswert, dass diese Dienste zusätzlich die Testabdeckung ermitteln. Dann könnten fehleranfällige Annäherungsverfahren wie in dieser Arbeit oder in [Alves und Visser '09] vermieden werden. Die genaue dynamische Testabdeckung eines Projekts ließe sich einfach von der Webseite des benutzten CI-Dienstes abgreifen. Weiterhin lässt sich vermuten, dass automatisch ermittelte Testabdeckung ein besseres Bewusstsein für Teststrategien bei den Entwicklern hervorrufen kann. Speziell die Veränderung der Abdeckung (aber auch anderer Metriken) durch Änderungen eines einzelnen Commits wären für die konstruktive Qualitätssicherung interessant.

14.2.2 Weitere Einflussfaktoren

Es könnten in eine Formel für den Testbedarf noch eine Reihe weiterer konkreter Aspekte als Einflussfaktoren eingebracht werden. Beispielsweise ließen sich statisch für eine gegebene Programmiersprache Entwurfsmuster erkennen. Man könnte versuchen Mocking zu erkennen. Das deutet auf fortgeschrittene Testkultur. Man könnte zur Beurteilung des Testbedarf eines Projekts auch den Verwalter näher betrachten. Hat ein Projektverwalter in früheren Projekten bereits Erfahrungen mit Tests oder gar TDD gemacht? In ferner Zukunft wäre es auch denkbar Testkonventionen aus prosaischen Dokumenten und Wikis über eine Mustererkennung zu extrahieren. Dies würde eine automatische Suche und Erkennung von Richtlinien für Tests ermöglichen. Automatische Testerzeugung und Testminimierung könnte bei Anwendbarkeit auf fehlende Tests oder redundante Tests hinweisen. Dadurch könnte die Qualität der Tests eingeschätzt werden.

Die Häufigkeit von Refaktorisierungen könnte man anhand von Commit-Nachrichten erraten. Man könnte vermuten, dass Refaktorisierungen den Testbedarf verringern.

[3] Continuous Integration (CI). Bekanntester Vertreter aktuell TravisCI `https://travis-ci.org`.

Das aber nur, wenn der betroffene Code *vollständig* von *korrekten* Testfällen abgedeckt ist. Andernfalls können Regressionsfehler eingeführt worden sein.

14.2.3 Feinere Heuristik

Feingranulare Fehlervorhersagemodelle erlauben einer Vorhersage der vermutlich fehleranfälligsten Module/Klassen. Dadurch könnte man die Klassen eines Projekts in Reihenfolge des absteigenden Testbedarfs bringen. Würde man dann die bedürftigsten Klassen zuerst testen, könnte sich die Chance Fehler zu finden (und somit die Testeffizienz) weiter erhöhen.

14.2.4 Verbesserung der Auswertung

Es gibt eine Möglichkeit die Umfrageergebnisse besser auszuwerten. Nämlich indem für *alle Projekte* jedes Nutzers die Metrik-Werte berechnet werden. Dann wird geprüft, ob eine gegebene Formeln die Angabe des Projekts mit dem geringsten/höchsten Testbedarf/Fehlerzahl korrekt vorhersagt.

Dies hätte die Bewertung der Formeln verbessert. Denn wenn ein Nutzer einem Projekt einen maximalen Testaufwand beziehungsweise eine maximale Fehlerzahl zuordnet, impliziert dies für alle anderen Projekte des Nutzers einen geringeren Testaufwand respektive Fehlerzahl.

Andererseits wäre dadurch die Auswertung in dieser Arbeit enorm erschwert worden. Denn allein das Klonen der genannten Projekte hatte mehrere Stunden Zeit in Anspruch genommen. Das Klonen aller ca. 40.000 Projekte der Befragten würde vermutlich mehrere Tage in Anspruch nehmen. Eine Anpassung des Messwerkzeugs auf eine verteilte Ausführung könnte dieses Problem eventuell lösen.

14.2.5 Java-spezifische Metriken

Das mitgelieferte Werkzeug implementiert bereits einige Java-spezfische Metriken, welche nicht in die Heuristik eingeflossen sind. Dies wurde mit der geforderten Allgemeingültigkeit der zu entwickelnden Heuristik begründet. Eine zukünftige Arbeit könnte eine möglicherweise bessere Heuristik nur für Java-Projekte entwickeln. Das Vorgehen könnte analog zu dieser Arbeit erfolgen. Hierbei müsste man dann in einer Umfrage speziell Nutzer mit Java-Projekten zu diesen befragen und für diese Projekte die java-spezifischen Metriken messen. Besonders die statische Codeanalyse PMD ist ein vielversprechende Kandidat für die Vorhersage der Fehlerzahl.

Als weitere java-spezifische Metrik könnten ein Erkennungstool für die Test-Smells aus [Breugelmans und Rompaey ’08] implementiert werden. Anhand der Anzahl der gefundenen Smells, wäre eine Aussage über die Qualität der Tests möglich. Je weniger Smells, desto besser.

14.2.6 Erkennung von testgetriebener Entwicklung

Verwandt mit Co-Evolution[4] von Produktionscode und Testcode ist Erkennung der testgetriebenen Entwicklung. In [Zaidman et al. '08] konnten Hinweise auf testgetriebene Entwicklung in Checkstyle[5] in der Veränderungshistorie gefunden werden.

Ein Indikator für testgetriebene Entwicklung ist, dass Revisionen mit neuen Modulen auch automatisch die dazugehörigen Testfälle enthalten. Würde man dies spezifisch für Java und das Framework JUnit betrachten, gibt es eine Konvention die zu einer Klasse gehörige Unit-Test-Klasse genau wie die getestete Klasse zu nennen mit Hinzufügung des Suffix "Test". Für Repositories, welche mit dem verteilten Versionsverwaltungssystem Git verwaltet werden, könnte man dies realisieren, indem man die erste Revision einer Klasse mit dem `git log`-Kommando bestimmt und dies ebenfalls für die zugehörige Testklasse (falls vorhanden) tut. Besitzt jede Klasse eine zugehörige Testklasse, und ist die erste Revision der Testklasse nicht jünger als die der getesteten Klasse, dann lässt sich eine testgetriebene Entwicklung beim Projekt begründet vermuten.

14.3 Zusammenfassung

Eine explorative Voranalyse des Testen auf GitHub ergab, dass weniger als die Hälfte der Projekte auf GitHub überhaupt testet.

Im Rahmen dieser Arbeit wurde ein erweiterbares Werkzeug für die Messung, Analyse und Visualisierung von Metriken auf GitHub entwickelt. Zusätzlich liefert das Werkzeug einen Bewertungsmechanismus von Vorhersagemodellen für den intuitiv geschätzten relativen Testaufwand von Projekten und die intuitiv geschätzte relative Zahl der verbleibenden Defekte in einem Projekt. Dieser Mechanismus enthält seine Daten aus zwei Umfragen auf GitHub.

Es wurden verschiedene Formeln mit Termen aus Produktionscode-, Testcode- und Repository-Metriken als Vorhersagemodelle evaluiert. Dabei wurden speziell auch Metriken benutzt, welche sich in der Literatur bereits als gute Stellvertreter für Fehlerzahl und Testaufwand qualifizierten.

Es wurden aus der Umfrage jeweils eine Formel für die Fehlerzahl und den Testaufwand abgeleitet. Diese wurden zu einer Formel für einen Schätzwert des Testbedarfs kombiniert. Die Formel induziert eine Heuristik für den Testbedarf.

[4] Mehr dazu in [Zaidman et al. '10].

[5] Werkzeug zur Überprüfung der Einhaltung von Richtlinien für den Programmierstil. (`http://checkstyle.sourceforge.net/`)

Literaturverzeichnis

[Afzal et al. ’09] W. Afzal, R. Torkar, und R. Feldt,
Search-Based Prediction of Fault Count Data,
1st International Symposium on Search Based Software Engineering, pp. 35–38, May 2009.

[Ahsan und Wotawa ’11] S. N. Ahsan und F. Wotawa,
Fault Prediction Capability of Program File’s Logical-Coupling Metrics,
Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement, pp. 257–262, Nov. 2011.

[Alves und Visser ’09] T. L. Alves und J. Visser,
Static estimation of test coverage, Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 55–64, 2009.

[Basili et al. ’94] G. Caldiera, V. Basili, und D. Rombach,
The goal question metric approach,
Encyclopedia of software engineering 2, pp. 528-532, 1994.

[Bird et al. ’09] C. Bird, P. Rigby, E. Barr, D. Hamilton, D. German, P. Devanbu,
The Promises and Perils of Mining Git,
Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, pp. 1-10, 2009.

[Breugelmans und Rompaey ’08] M. Breugelmans und B. Van Rompaey,
TestQ: Exploring structural and maintenance characteristics of unit test suites,
ASE Workshops 2008. 23rd IEEE/ACM International Conference on 15-16 Sept 2008. pp. 11-20. 2008.

[Chao und Yang ’93] A. Chao und M. Yang,
Stopping rules and estimation for recapture debugging with unequal failure rates,
Biometrika vol. 80, No. 1, pp. 193–201, 1993.

[Dalal und Mallows ’88] S. Dalal und C. Mallows,
When should one stop testing software?
Journal of the American Statistical Association, vol. 83, no. 403, pp. 872–879, 1988.

[Danial ’13] A. Danial,
CLOC - Count Lines of Code,
`http://cloc.sourceforge.net/`, 2013.
Letzter Abruf: 01.05.2013

[Eaddy und Zimmermann '08] M. Eaddy und T. Zimmermann,
Do crosscutting concerns cause defects?,
IEEE Transactions on Software Engineering, vol. 34, no. 4, pp. 497–515, July-Aug. 2008.

[Fenton und Neil '00] N. Fenton und M. Neil,
Software metrics: Roadmap,
ICSE '00 Proceedings of the Conference on the Future of Software, pp. 357-370, 2000.

[Fitzsimmons und Love '78] A. Fitzsimmons und T. Love,
A review and evaluation of software science,
ACM Computing Surveys (CSUR), vol. 10, no. 1, 1978.

[Gaffney '84] J. Gaffney,
Estimating the number of faults in code,
IEEE Transactions on Software Engineering, vol. 00, no. 4, pp. 459–464, 1984.

[Hall et al. '11] T. Hall, S. Beecham, D. Bowes, D. Gray, und S. Counsell, *A systematic literature review on fault prediction performance in software engineering*,
IEEE Transactions on Software Engineering, vol. 38, no. 6, pp. 1276-1304, Nov.-Dec. 2012.

[Halstead '77] M. H. Halstaed,
Elements of Software Science,
Elsevier Science Inc. New York, New York, USA, 1977.

[Hamer und Frewin '82] P. Hamer und G. Frewin,
MH Halstead's Software Science-a critical examination,
Proceedings of the 6th international conference on Software, pp. 197–206, 1982.

[Hata '12] H. Hata,
Fault-prone Module Prediction Using Version Histories,
Dissertation, Graduate School of Information Science and Technology, Osaka University, 2012.

[Hata et al. '12] H. Hata, O. Mizuno, und T. Kikuno,
Bug prediction based on fine-grained module histories,
34th International Conference on Software Engineering (ICSE), pp. 200–210, Jun. 2012.

[Herraiz und Hassan '10] I. Herraiz und A. Hassan,
Beyond Lines of Code: Do We Need More Complexity Metrics?,
Making Software: What Really Works, and Why We Believe It, O'Reilly Media, Chapter 8, pp. 125-141, First Edition, 2010.

[Hoffmann '08] D. Hoffmann,
Software-Qualität
Springer-Verlag Berlin Heidelberg, 2008.

[ISO/IEC '99] International Organization for Standardization,
14598-1 Information technology - Software product evaluation - Part 1: General overview,
`http://www.iso.org/iso/catalogue_detail.htm?csnumber=24902`,
1999. Letzter Abruf: 01.05.2013

[Joshi et al. '07] H. Joshi, C. Zhang, S. Ramaswamy, und C. Bayrak,
Local and Global Recency Weighting Approach to Bug Prediction,
4th International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007), no. 3, pp. 33–33, May 2007.

[Kochhar et al. '13] P. Kochhar, T. Bissyandé, D. Lo, und L. Jiang,
Adoption of Software Testing in Open Source Projects: A Preliminary Study on 50,000 Projects,
17th European Conference on Software Maintenance and Reengineering (CSMR), 2013.

[Li et al. '98] M. Li, Y. Malaiya, und J. Denton,
Estimating the number of defects: a simple and intuitive approach,
Proeedings of the 7th International Symposium on Software Reliability Engineering (ISSRE), pp. 307-315, 1998.

[Lipow '82] M. Lipow,
Number of faults per line of code,
IEEE Transactions on Software Engineering, vol. 1, no. 4, pp. 437–439, 1982.

[Mancoridis '09] S. Mancoridis,
Topics in Metrics for Software Testing,
Lecture slides, Software Verification and Validation (SE 320), Drexel University, 2009,
`https://www.cs.drexel.edu/~spiros/teaching/SE320/slides/metrics.pdf`.
Letzter Abruf: 01.05.2013.

[McCabe '76] T. McCabe,
A Complexity Measure,
IEEE Transactions on Software Engineering, Volume SE-2, Issue 4, pp. 308-320, 1976.

[Meneely und Williams '12] A. Meneely und O. Williams,
Interactive Churn Metrics: Socio-Technical Variants of Code Churn,
ACM SIGSOFT Software Engineering Notes archive, Volume 37 Issue 6, November 2012, Pages 1-6, 2012.

[Mizuno und Kusumoto '97] O. Mizuno und S. Kusumoto,
Estimating the number of faults using simulator based on generalized stochastic Petri-net model,
6th Asian Test Symposium Proceedings, 1997 (ATS'97), pp. 269–274, 1997.

[Myers ’04] G. Myers,
The Art of Software Testing,
John Wiley & Sons, Inc. Hoboken, New Jersey, Second Edition, 2004.

[Nagappan ’05a] N. Nagappan,
A Software Testing and Reliability Early Warning (STREW) Metric Suite,
Dissertation, Graduate Faculty of North Carolina State University, 2005.

[Nagappan und Ball ’05] N. Nagappan und T. Ball,
Use of relative code churn measures to predict system defect density,
Proceedings of 27th International Conference on Software Engineering. ICSE 2005., pp. 284–292, 2005.

[Nagappan et al. ’06] N. Nagappan, T. Ball, und A. Zeller,
Mining metrics to predict component failures,
Proceeding of the 28th international conference on Software Engineering. ICSE 2006, p. 452, 2006.

[Nikora und Munson ’98] A. Nikora und J. Munson,
Estimating Rates of Fault Insertion and Test Effectiveness in Software Systems,
Proceedings of the 4th ISSAT International Conference on Quality and Reliability in Design, Seattle, WA, 1998.

[Ostrand et al. ’05] T. J. Ostrand, E. J. Weyuker, und R. M. Bell,
Predicting the location and number of faults in large software systems,
IEEE Transactions on Software Engineering, vol. 31, no. 4, pp. 340–355, 2005.

[Ottenstein et al. ’76] L. Ottenstein, V. Schneider, und M. Halstead,
Predicting the number of bugs expected in a program module,
Computer Science Department, Purdue University, Technical Report, 1976.

[Peng und Wallace ’93] W. Peng und D. Wallace,
Software Error Analysis,
NIST Special Publication 500-209, US Department of Commerce, NIST, 1993.

[Pham et al. ’13] R. Pham, L. Singer, O. Liskin, F. Filho, und K. Schneider
Creating a Shared Understanding of Testing Culture on a Social Coding Site,
Software Engineering Group, Leibniz University Hanover, Technical Report, 2013.

[Pham et al. ’13a] R. Pham, L. Singer, und K. Schneider,
Building Test Suites in Social Coding Sites by Leveraging Drive-By Commits,
Software Engineering Group, Leibniz University Hanover, Technical Report, 2013.

[Robles et al. ’04] G. Robles, J. Gonzalez-Barahona, und R. Ghosh,
Gluetheos: Automating the retrieval and analysis of data from publicly available software repositories
Proceedings of the International Workshop on Mining Software Repositories, 2004.

[Rosenberg ’97] J. Rosenberg,
Some misconceptions about lines of code,
Proceedings of the 4th International Symposium on Software Metrics, METRICS ’97, pp. 137–142, 1997.

[Schneider ’07] K. Schneider,
Abenteuer Softwarequalität: Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement,
dpunkt.verlag GmbH, Heidelberg, 2007.

[Sestoft ’12] P. Sestoft,
Programming Language Concepts,
Springer Undergraduate Texts in Computer Science, London, 2012.

[Shepperd und Kadoda ’01] M. Shepperd und G. Kadodam,
Comparing software prediction techniques using simulation,
IEEE Transactions on Software Engineering, vol. 27, no. 11, pp. 1014–1022, 2001.

[Sink ’11] E. Sink,
Version Control by Example,
Pyrenean Gold Press, 2011.

[Six ’07] B. Six,
Functional Collection Patterns in Java,
`http://www.ugrad.cs.jhu.edu/~wsix/collections.pdf`, 2007.
Letzter Abruf: 01.05.2013

[Stetter ’86] F. Stetter,
Comments on “number of faults per line of code”,
IEEE Transactions on Software Engineering, vol. SE-12, no. 12, pp. 1145–1145, 1986.

[Stoliar ’13] Y. Stoliar,
Entwurf und Entwicklung einer Ranking Website von Open Source Projekten für Crowdsourcing Zwecke,
Bachelor’s thesis, Leibniz University Hanover, 2013.

[Thung et al. ’13] F. Thung, T. Bissyandé, D. Lo, und L. Jiang,
Network Structure of Social Coding in GitHub,
17th European Conference on Software Maintenance and Reengineering (CSMR), 2013.

[Tiwari und Pandey ’12] V. Tiwari und R. Pandey,
Open Source Software and Reliability Metrics,
International Journal of Advanced Research in Computer and Communication Engineering (IJARCCE), vol. 1, no. 10, pp. 808–815, 2012.

[Weyuker und Ostrand ’10] E. Weyuker und T. Ostrand,
An Automated Fault Prediction System,
Making Software: What Really Works, and Why We Believe It, O’Reilly Media, Chapter 9, pp. 145-159, First Edition, 2010.

[Yip ’95] P. Yip,
Estimating the number of errors in a system using a martingale approach,
IEEE Transactions on Reliability, vol. 44, no. 2, pp. 322–326, 1995.

[Yip und Xi '99] P. Yip und L. Xi,
Sensitivity-analysis and estimating number-of-faults in removal debugging,
IEEE Transactions on Reliability, vol. 48, no. 3, pp. 300–305, 1999.

[Zaidman et al. '08] A. Zaidman, B. Van Rompaey, S. Demeyer, und A. Van Deursen,
Mining Software Repositories to Study Co-Evolution of Production & Test Code,
2008 International Conference on Software Testing, Verification, and Validation, pp. 220–229, 2008.

[Zaidman et al. '10] A. Zaidman, B. Rompaey, A. Deursen, und S. Demeyer,
Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining,
Technical Report TUD-SERG-2010-035, Delft University of Technology, SE Research Group, 2010.

Abbildungsverzeichnis

Tabellenverzeichnis

Compact Disc

Jedem Exemplar dieser Arbeit liegt eine CD mit den folgenden Daten bei:

- LaTeX-Quellen dieser Arbeit im Verzeichnis `TexSrc`.
- PDF-Version der Arbeit `Schnabel2013.pdf`.
- Quellcode von *JProjectInspector* im gleichnamigen Verzeichnis.
- Handbuch `Manual.pdf` von *JProjectInspector* in englischer Sprache. Liefert Hilfestellung bei der Einrichtung des Werkzeugs.
- HTML-Dokumentation von *JProjectInspector* in Verzeichnis `JavaDoc`. Wurde mithilfe des Werkzeugs `javadoc` aus Kommentaren im Quelltext generiert.
- Aus Umfragen gewonnene CSV-Datensätze für Benchmark-Funktionalität im Unterverzeichnis `JProjectInspector/data/benchmark`.
  - `WeightedEstimatesUmfrage{1,2,Combined}.csv` beinhaltet die Vermutungen von Projektinhabern über das von ihnen verwaltete Projekt mit kleinstem/größten Testaufwand/Fehlerzahl aus der ersten, zweiten Umfrage oder beiden Umfragen kombiniert.
  - `MetricResultsUmfrage{1,2,Combined}.csv` enthält Messergebnisse einiger Metriken für die genannten Projekte aus der ersten, zweiten Umfrage oder beiden Umfragen kombiniert.
- Rohantworten der beiden Umfragen auf Google Forms im CSV-Format im Verzeichnis `Survey`. Vertraulich zu behandeln.
- Eigenständig ausführbare `.jar`-Datei `JProjectInspector.jar` in Verzeichnis `JProjectInspector`. Benötigt Laufzeitumgebung für Java in Version 6 oder höher.
- Quellcode vom Werkzeug *EvalMassMailer*, welches zur Durchführung der Umfragen entwickelt und benutzt worden ist. Befindet sich im gleichnamigen Verzeichnis.
- Kopien der referenzierten Online-Quellen im Stand, wie sie dem Autor beim Verfassen dieser Arbeit vorlagen. In `WebArchive`.

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 06.05.2013

_____

André Schnabel