Universität Passau

Fakultät für Informatik und Mathematik

Bachelorarbeit

Empirische Untersuchung des Zusammenhangs zwischen Fehlerhäufigkeit und ausgewählten Softwaremaßen

Verfasser:

Stefan John

21.03.2012

Betreuer:

Dr.-Ing. Sven Apel

Universität Passau
Fakultät für Informatik und Mathematik
Innstraße 41, D–94032 Passau

John, Stefan:
Empirische Untersuchung des Zusammenhangs zwischen Fehlerhäufigkeit und ausgewählten Softwaremaßen
Bachelorarbeit, Universität Passau, 2012.

Zusammenfassung

Softwaremaße werden mittlerweile vermehrt in den Entwicklungsprozess von Softwaresystemen eingebunden. Durch ihre Verwendung können in der Regel schnell Informationen über die Eigenschaften eines Systems beschafft werden, die zur Verbesserung desselben beitragen. Um allerdings zu erkennen, über welche Eigenschaften durch die Betrachtung eines Maßes Aussagen getroffen werden können, sind oftmals empirische Untersuchungen notwendig.

In der vorliegenden Arbeit wurden zu diesem Zweck vier Softwaremaße untersucht. Ziel war es zu bestimmen, welche Aussagekraft diese bezüglich der Komplexität eines Programms besitzen. Als Bemessungsgrundlage der Komplexität wurde hierzu die Fehlerhäufigkeit von Programmen betrachtet. Es wurden fehlerhafte und fehlerfreie Versionen dreier großer Softwareprojekte mit diesen Maßen gemessen, ihre Messwerte verglichen und die resultierenden Ergebnisse statistisch ausgewertet.

Trotz kleinerer Unterschiede zwischen den Ergebnissen der analysierten Maße ergab die Untersuchung, dass sich alle vier als Indikator für die Komplexität eines Programms eignen. Zudem konnte als mögliches Einsatzgebiet der Maße die systematische Fehlersuche in Programmen identifiziert werden. Einen eindeutigen „Testsieger“ unter den betrachteten Softwaremaßen konnte die Anaylse jedoch nicht zu Tage fördern.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Software ist heutzutage allgegenwärtig. Von der Handy-Applikation bis zum Betriebssystem, vom Spiel bis zur Steuerung medizinischer Anlagen wird der Mensch in allen Lebensbereichen mit ihr konfrontiert. Abhängig vom Anwendungsgebiet wird dabei oft eine Reihe von Anforderungen an die Softwareprodukte gestellt. Die wohl intuitivste und grundlegendste ist der fehlerfreie Betrieb der Programme. Betrachtet man die Größe heutiger Softwaresysteme[1] wird schnell deutlich, dass schon die Erfüllung einer solch elementaren Forderung keine leichte Aufgabe ist.

Der Bereich der Softwaretechnik hat über die Jahre eine Vielzahl von „Prinzipien, Methoden und Werkzeugen“ [Bal00, S. 36] hervorgebracht, die den Entwickler bei der Bewältigung dieser Aufgabe unterstützen sollen. Eines dieser Hilfsmittel sind Softwaremaße. Mit ihrer Hilfe können durch einfaches Messen der Artefakte eines Softwaresystems (zum Beispiel der Funktionen oder Quelltextzeilen) Aussagen über die Eigenschaften des Systems getroffen werden. Im Vergleich zu anderen Verfahren der Softwaretechnik ist die Durchführung einer solchen Messung eine schnell durchführbare, unkomplizierte Aufgabe. So ist es oftmals sogar möglich, die nötigen Berechnungen nahezu synchron zur Weiterentwicklung des Quelltextes durchzuführen und somit die Auswirkungen einer Änderung auf die betrachteten Eigenschaften des Programms sofort zu erfassen. Besonders aus diesem Grund finden Softwaremaße vermehrt Anwendung in heutigen Entwicklungsprozessen.

Sinnvoll ist der Einsatz eines Softwaremaßes natürlich nur dann, wenn es zur Beurteilung einer Systemeigenschaft beiträgt, die für den Entwickler von Interesse ist. Daher stellt sich sowohl für bestehende als auch für neu entwickelte Maße häufig die Frage, für welche Einsatzgebiete diese geeignet sind bzw. welche Merkmale durch sie beschrieben werden. Zur Beantwortung dieser Frage ist eine formale Beweisführung in der Regel nicht möglich, da meist eine Vielzahl unvorhersehbarer Faktoren (z. B. individueller Programmierstil, Motivation des Entwicklers etc.) die Aussagekraft der Maße beeinflussen. Empirische Untersuchungen sind also unerlässlich, um mehr über die Aussagekraft von Softwaremaßen zu erfahren und somit eventuell auch neue Einsatzmöglichkeiten zu erkennen.

[1] Debian 5.0: ca. 320 Mio. Source Lines Of Code
Quelle: `http://libresoft.dat.escet.urjc.es/debian-counting/lenny`

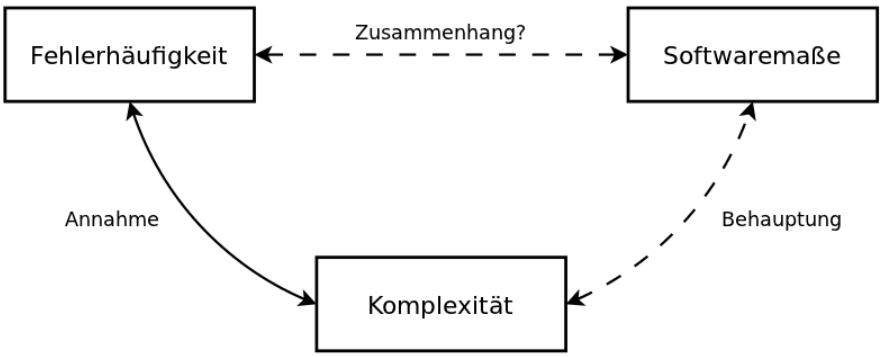

Abbildung 1.1: Modell der durchgeführten Untersuchung

1.2 Ziel der Arbeit

Ziel dieser Arbeit war, vier bestehende Softwaremaße auf ihre Aussagekraft bezüglich der Komplexität eines Programms hin zu untersuchen. Da Komplexität eine abstrakte, nicht messbare Größe ist, war es zur Durchführung einer solchen Untersuchung notwendig, Artefakte eines Programms zu betrachten, die ihrerseits etwas über die Komplexität desselben aussagen. Die Wahl fiel dabei auf Programmfehler bzw. deren Häufigkeit.

Dieser Entscheidung liegt die Annahme zugrunde, dass komplexe, unüberschaubare Programmteile fehleranfälliger sind, als solche, die vom Entwickler leicht verstanden werden können, sich also die Komplexität eines Programms in dessen Fehlern widerspiegelt. Aus dem Zusammenhang zwischen der Fehlerhäufigkeit und den Softwaremaßen kann unter dieser Annahme auf den Zusammenhang zwischen den Softwaremaßen und der Komplexität eines Programms geschlossen werden. Abbildung 1.1 soll dieses Vorgehen verdeutlichen.

1.2.1 Frühere Untersuchungen

Die hier untersuchten Softwaremaße wurden bereits in früheren Arbeiten auf ihre Aussagekraft hin überprüft. In diesen kamen jedoch andere Repräsentationen der Programmkomplexität zum Einsatz. In einer bisher unveröffentlichten Studie der Universität Passau wurde zum Beispiel eine Personenbefragung durchgeführt, um die Komplexität verschiedener Quelltexte über den subjektiven Eindruck der Testpersonen zu bestimmen. D. Balzer hingegen betrachtet in seiner Masterarbeit [Bal10] in der Praxis durchgeführte Refaktorierungen, um die Eignung der Maße als Indikatoren für die Komplexität zu untersuchen.

Die hier durchgeführte Anaylse, mit dem Blick auf die Fehlerhäufigkeit eines Programms, ist also als weiterer Schritt zu einer ganzheitlichen Betrachtung der Maße zu verstehen.

1.2.2 Fehler vs. Defekt

Häufig wird in der Literatur zwischen Fehlern und Defekten eines Programms unterschieden. Daher soll an dieser Stelle genauer erklärt werden, was in der vorliegenden Arbeit unter dem Begriff „Fehler“ zu verstehen ist.

Als *Defekte* werden fehlerhafte Stellen im Quelltext des Programms bezeichnet. Diese können zur Laufzeit zu einem unerwarteten Verhalten des Programms, fehlerhaften Berechnungen oder Abstürzen führen. Der Begriff des *Fehlers* hingegen umschließt mehr als das. Er bezieht sich nicht auf den Quelltext, sondern umfasst ganz allgemein jegliches unerwartete und unerwünschte Verhalten eines Programms. Auch während der Ausführung korrekter Quelltexte können Fehler auftreten, wenn beispielsweise Diskrepanzen zwischen der Spezifikation und den tatsächlichen Erwartungen der Anwender bestehen. Wird im Folgenden also von der Behebung eines Fehlers gesprochen, ist darunter nicht nur das Berichtigen fehlerhafter Quelltextstellen zu verstehen. Auch funktionale Erweiterungen oder Änderungen können Teil einer Fehlerbehebung sein.

Kapitel 2

Softwaremaße

Formal handelt es sich bei einem Softwaremaß um eine Funktion, die einem Artefakt der Softwaretechnik (z. B. Entwickler, Klasse, Funktion) eine Maßzahl zuordnet.

> I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the state of *science*, whatever the matter may be [Tho89].

Softwaremaße sind heute oft ein wichtiger Bestandteil der Softwareentwicklung. Sie dienen der Quantifizierung von Eigenschaften eines Produkts oder Prozesses und sollen dadurch die Entwickler bei ihrer Arbeit unterstützen. Merkmale eines Systems durch die Verwendung von Softwaremaßen in Zahlenwerte abzubilden ermöglicht es, Vergleiche anzustellen, Abschätzungen zu treffen und Fortschritte sowie Schwachpunkte der Entwicklung im Auge zu behalten.

2.1 Komplexitätsmaße

Die in dieser Arbeit untersuchten Maße fallen in die Kategorie der Komplexitätsmaße. Als Komplexitätsmaße werden jene Softwaremaße bezeichnet, die dabei helfen sollen, die Komplexität eines Programms zu bewerten. Diese soll zum Ausdruck bringen, wie schwer es fällt, ein System zu verstehen, zu ändern oder zu erweitern. Bei der Softwareentwicklung kommen heute oft Komplexitätsmaße zum Einsatz. Grund dafür sind die negativen Auswirkungen, die eine zu hohe Komplexität auf den Entwicklungsprozess haben kann. Die Produktivität der Entwickler sinkt, je mehr Zeit diese auf das Verstehen des Quelltextes investieren müssen. Zudem steigt mit der Komplexität in der Regel die Fehleranfälligkeit eines Programms, ein Aspekt, der auch als Ausgangspunkt dieser Untersuchung diente (siehe 1.2).

2.2 Untersuchte Maße

Im Folgenden werden die in dieser Arbeit untersuchten Softwaremaße vorgestellt und die zum Verständnis notwendigen Begriffe anhand eines Quelltext-Beispiels (Abb. 2.1) erläutert.

```
1  public static int[] bubbleSort(int[] array) {
2          int temp;
3          int n = array.length-1;
4          boolean changed = true;
5
6          while (changed) {
7                  changed = false;
8                  for (int i=0; i<n; i++) {
9                          if(array[i] > array[i+1]) {
10                                 swap(array,i,i+1);
11                                 changed = true;
12                         }
13                 }
14                 n--;
15         }
16         return array;
17 }
```

Abbildung 2.1: Java-Implementierung des Bubblesort-Algorithmus

2.2.1 Dependency Degree

Dependency Degree (DepDegree oder DD) [BF10] ist das jüngste der in dieser Arbeit untersuchten Maße. Wie der Name bereits vermuten lässt, werden bei ihm Abhängigkeiten zwischen den Operationen eines Programms betrachtet. Als Operationen werden hierbei Zuweisungen, bedingte Anweisungen, Funktionsaufrufe und die Rückkehr von Funktionen bezeichnet.

Kontrollflussgraph

Da der Kontrollfluss [ALSU07] bei der Ermittlung der Abhängigkeiten eine entscheidende Rolle spielt, wird zunächst jede Funktion des Programms durch ihren *Kontrollflussgraphen* (CFG – englisch: Control Flow Graph) dargestellt. Ein solcher Graph $G = (B, F)$ besteht aus der Knotenmenge B, welche die Operationen der zugrunde liegenden Funktion repräsentiert, und der Menge $F \subseteq B \times B$ der Kontrollflusskanten, die die möglichen Wege durch das Programm beschreiben. Üblicherweise werden zusätzlich ein Eintritts- und ein Austrittsknoten hinzugefügt, die den Aufruf bzw. das Verlassen des repräsentierten Programmteils darstellen. Abbildung 2.2 zeigt einen zum Quelltext-Beispiel passenden Kontrollflussgraphen.

Erreichende Definitionen

Zur Ermittlung der für das Maß DepDegree relevanten Abhängigkeiten wird, in Anlehnung an das gleichnamige Verfahren zur Analyse des Datenflusses [ALSU07], die Funktion *Reaching Definitions* $rd_G : B \times X \to 2^B$ definiert, wobei X die Menge der Variablen des Programms repräsentiert. Durch jene wird einem Tupel (b_u, x)

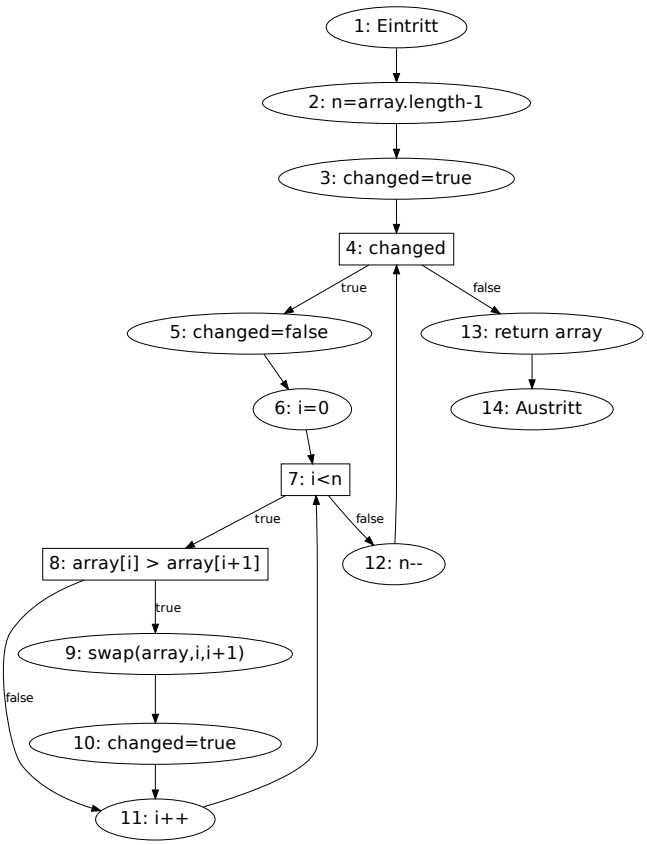

Abbildung 2.2: Kontrollflussgraph G der in Abb. 2.1 dargestellten Methode. Zur besseren Verständlichkeit wurden die ausgehenden Kanten der bedingten Anweisungen passend beschriftet.

die Menge aller Definitionen von x zugewiesen, die b_u erreichen. Als Definition einer Variablen x werden alle Operationen betrachtet, die x einen Wert zuweisen. Eine Definition b_d einer Variablen erreicht dabei eine Operation b_u genau dann, wenn im CFG ein Weg von b_d nach b_u existiert, auf dem die betrachtete Variable nicht durch eine andere Definition überschrieben wird.

Use-Def-Graph

Die Bestimmung der erreichenden Definitionen ist Voraussetzung für die Erstellung eines *Use-Def-Graphen* $S_G = (B, E)$. Die Knoten des zugrunde liegenden Kontrollflussgraphen G bilden dabei auch die Knoten des Use-Def-Graphen. Da der Austrittsknoten des CFG weder eine Definition noch die Verwendung einer Variablen beinhalten kann, ist er für diesen Graphen nicht von Bedeutung. Die Menge E der Use-Def-Kanten bildet die durch die erreichenden Definitionen gegebenen Beziehungen ab. Eine Kante zwischen den Operationen b_u und b_d existiert genau dann, wenn in b_u eine Variable verwendet wird, welcher in b_d ein Wert zugewiesen wurde und zusätzlich b_d eine erreichende Definition für b_u ist, also $b_d \in rd_G(b_u, x)$ gilt. Ein zum Quelltext-Beispiel passender Use-Def-Graph ist in Abbildung 2.3 zu sehen.

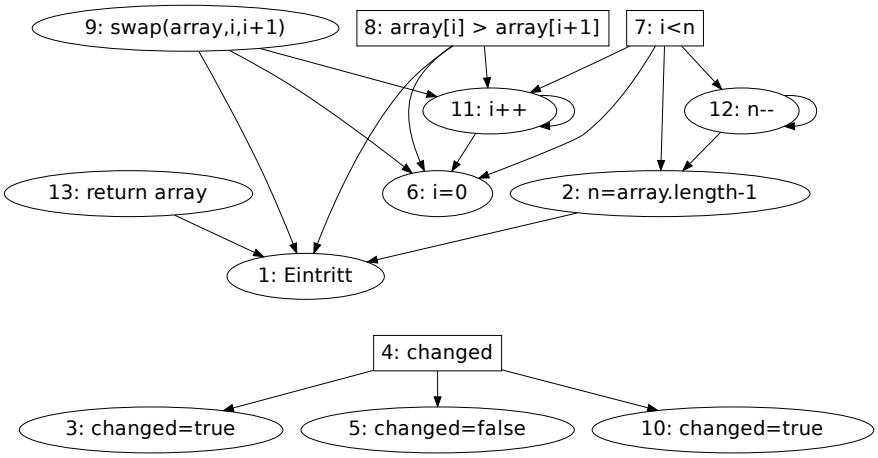

Abbildung 2.3: Use-Def-Graph S_G der in Abb. 2.1 dargestellten Methode. Der Eintrittsknoten symbolisiert die Definition des Funktionsparameters.

DepDegree

Der DepDegree-Wert $dd_G : B \to \mathbb{N}$ einer Operation ist definiert als die Anzahl ihrer ausgehenden Kanten im Use-Def-Graphen. Betrachtet man eine Funktion f im Programm als Menge von Operationen, errechnet sich der DepDegree-Wert dieser Funktion entsprechend aus der Summe der DepDegree-Werte ihrer Operationen.

$$dd_G(f) = \sum_{b \in f} dd_G(b) \tag{2.1}$$

Beispiel

Betrachtet man den Use-Def-Graphen des Bubblesort-Algorithmus (Abb. 2.3) und zählt dessen Kanten, so ergibt sich für die zugrunde liegende Funktion ein DepDegree-Wert von $dd_G(bubbleSort) = 19$.

2.2.2 Cyclomatic Complexity

Das Maß *Cyclomatic Complexity* (CC) wurde bereits 1976 von Thomas J. McCabe entwickelt [McC76]. Bei diesem spielt nicht der Datenfluss, sondern vielmehr die Struktur des untersuchten Programms eine Rolle. Die Grundidee besteht darin, die Anzahl der möglichen Pfade durch das Programm als Indikator für die Komplexität desselben zu verstehen. Da ein Entwickler bei der Wartung oder Erweiterung eines Systems schlimmstenfalls die Auswirkungen seiner Modifikationen auf sämtliche Programmpfade berücksichtigen muss, erscheint dies sinnvoll.

Ein Programm kann potentiell unendlich viele Pfade besitzen. Um dieser Problematik aus dem Weg zu gehen, wird beim Maß Cyclomatic Complexity nur die maximale Anzahl der linear unabhängigen Pfade ermittelt. Dies entspricht dem Grundgedanken, da sich aus der Menge der linear unabhängigen auch alle anderen Pfade eines Programms als Linearkombination erzeugen lassen.

Zur Berechnung wird auch bei diesem Maß ein Kontrollflussgraph (siehe 2.2.1) zur Repräsentation eines Programms (bzw. einer Funktion) herangezogen. Für einen CFG G ergibt sich der Wert des Maßes Cyclomatic Complexity aus der Anzahl n der Knoten, der Anzahl e der Kanten und der Anzahl p der zusammenhängenden Komponenten des Graphen.

$$CC(G) = e - n + 2p \tag{2.2}$$

Man kann zeigen, dass jede bedingte Anweisung (beispielsweise ein Fall einer Switch-Anweisung) die Zahl der linear unabhängigen Pfade um eins erhöht. Dieser wesentlich intuitivere Ansatz zur Bestimmung des Maßes findet auch im Werkzeug DepDigger (siehe 3.1) Anwendung, welches gleichfalls im Rahmen dieser Arbeit verwendet wurde.

Beispiel

Der Kontrollflussgraph des Bubblesort-Algorithmus (Abb. 2.2) besitzt 16 Kanten, 14 Knoten und besteht aus einer zusammenhängenden Komponente. Der Wert der Cyclomatic Complexity beträgt demnach $CC(G) = 16 - 14 + 2 = 4$. Zum selben Ergebnis kommt man durch Betrachtung des Quelltextes (Abb. 2.1). Zum Ausgangspfad der Funktion wird durch die drei bedingten Anweisungen (while, for und if) jeweils ein zusätzlicher linear unabhängiger Pfad hinzugefügt.

2.2.3 Halstead Difficulty

Ein Jahr nach McCabe veröffentlichte auch Maurice H. Halstead eine Reihe von Softwaremaßen [Hal77], die der Einschätzung verschiedener Eigenschaften eines Softwaresystems dienen sollen. Im Rahmen der vorliegenden Arbeit wurde nur das Maß *Halstead Difficulty* (HD) untersucht. Dieses dient als Indikator dafür, wie schwer ein Programm zu schreiben und zu verstehen ist.

Die Berechnung aller Halstead-Maße basiert auf der Überlegung, die syntaktischen Elemente eines Programms in Operatoren und Operanden zu unterteilen. Diese Kategorisierung hängt in erster Linie von der gewählten Programmiersprache ab. Die Maße setzen anschliessend die Anzahlen dieser Bausteine geeignet in Beziehung.

Im Fall des Maßes Halstead Difficulty fließen zwei Faktoren in die Berechnung mit ein: Zum einen die Hälfte der Anzahl n_1 der unterschiedlichen Operatoren, zum anderen das Verhältnis der Anzahl N_2 aller Operanden zur Anzahl n_2 der unterschiedlichen Operanden.

$$HD = \frac{n_1}{2} \cdot \frac{N_2}{n_2} \tag{2.3}$$

Dieses Maß geht also davon aus, dass sich sowohl die Verwendung vieler Operatoren, als auch die mehrmalige Verwendung des gleichen Operanden negativ auf die Verständlichkeit eines Programms auswirken.

```
1  public static int[] bubbleSort(int[] array) {
2          int temp;
3          int n = array.length-1;
4          boolean changed = true;
5
6          while (changed) {
7                  changed = false;
8                  for (int i=0; i<n; i++) {
9                          if(array[i] > array[i+1]) {
10                                 swap(array,i,i+1);
11                                 changed = true;
12                         }
13                 }
14                 n--;
15         }
16         return array;
17 }
```

Abbildung 2.4: Darstellung einer Unterteilung der Bubblesort-Implementierung in Operatoren und Operanden. Während Operanden kursiv und in grüner Farbe dargestellt sind, sind Operatoren rot gefärbt. Die Schlüsselworte „public“ und „static“ werden nicht beachtet.

Beispiel

Um die Berechnung des Maßes am Beispiel des Bubblesort-Algorithmus zu verdeutlichen, sind in Abbildung 2.4 Operatoren und Operanden des Quelltextes, ausgehend von der durch das Programm DepDigger (siehe 3.1) vorgenommenen Unterteilung, farblich sowie durch unterschiedliche Schriftlagen markiert. Betrachtet man diese Abbildung, lassen sich $n_1 = 22$ unterschiedliche Operatoren identifizieren. Die Anzahl unterschiedlicher Operanden n_2 beträgt 12. Insgesamt werden $N_2 = 34$ Operanden verwendet. Für die Funktion bubbleSort ergibt sich somit ein Messwert von $HD = \frac{22}{2} \cdot \frac{34}{12} = 31,167$.

2.2.4 Method Lines of Code

Das Maß *Method Lines of Code* (MLOC) ist analog zu dem bekannteren Softwaremaß *Total Lines of Code* (TLOC) definiert. Gemessen wird die Länge des Programms anhand der Anzahl der physischen Zeilen seines Quelltextes. Im Gegensatz zur Messung logischer Zeilen können sich dabei innerhalb einer Zeile mehrere oder auch gar keine Anweisungen befinden. Kommentare und Leerzeilen werden nicht gewertet. Im Unterschied zu TLOC werden nur Zeilen innerhalb einer Methode berücksichtigt.

Da der Quelltext des Bubblesort-Beispiels (Abb. 2.1) aus 17 Zeilen besteht, ergibt sich durch den Wegfall der enthaltenen Leerzeile ein MLOC-Wert von 16.

Kapitel 3

Durchführung der Analyse

Nachdem im letzten Kapitel die Softwaremaße vorgestellt wurden, deren Beziehung zur Fehlerhäufigkeit von Programmen untersucht werden soll, wird im Folgenden die Durchführung der Analyse erläutert. Hierzu wird zunächst das Werkzeug DepDigger beschrieben, welches zur Berechnung der Maße verwendet wurde. Desweiteren werden die Softwareprojekte vorgestellt, die als Basis für die Untersuchung herangezogen wurden. Abschließend wird erklärt, welche Anpassungen und Einschränkungen der Softwaresysteme notwendig waren, um eine automatisierte Gewinnung der Messdaten zu ermöglichen.

3.1 DepDigger

Zur Berechnung der hier untersuchten Maße kam das Programm DepDigger[1] zum Einsatz. Es handelt sich dabei um ein in Java geschriebenes Anaylsewerkzeug, welches als Plugin für das Eclipse-Framework[2] und als Kommandozeilenversion verfügbar ist. Neben der reinen Berechnung der Maße, welche auf Methodenebene erfolgt, verfügt die Plugin-Variante über verschiedene Möglichkeiten zur Visualisierung der Maße in der Entwicklungsoberfläche und ist daher gut zur Integration in den Entwicklungsprozess eines Systems geeignet. Im Rahmen dieser Arbeit wurde zur automatisierten Bearbeitung der Testsysteme die dafür zweckmäßigere Kommandozeilenversion verwendet. Zur Durchführung der Analyse waren einige Anpassungen des Programms notwendig, die hier allerdings nicht weiter besprochen werden.

Kürzlich wurde die Unterstützung von C-Programmen in DepDigger integriert. Zum Zeitpunkt der Analyse war jedoch eine Berechnung der implementierten Softwaremaße lediglich für Java-Programme möglich. Hierfür wird zunächst ein AST (Abstract Syntax Trees) des zu untersuchenden Programms generiert. Dieser dient dann als Grundlage für die Erstellung weiterer Strukturen (bspw. des Kontrollflussgraphen; siehe 2.2.1) und die Berechnung der Maße. Da zur Generierung eines korrekten AST ein syntaktisch fehlerfreies Programm Voraussetzung ist, wurden mit Syntaxfehlern behaftete Programmdateien von der hier durchgeführten Analyse ausgeschlossen, um verfälschte Messergebnisse zu vermeiden. Zur Beurteilung der Syntax war die Java-Sprachspezifikation maßgebend. Erweiterungen der Sprache, wie beispielsweise die aspektorientierte Programmierung mit AspectJ, konnten nicht berücksichtigt werden.

[1] `http://www.sosy-lab.org/~dbeyer/DepDigger/`
[2] `http://www.eclipse.org/`

3.2 Untersuchte Softwaresysteme

Entscheidend für die Wahl einer geeigneten Datenbasis für diese Arbeit waren folgende Anforderungen:

1. Aufgrund der zur Verfügung stehenden Implementierung der Maße durch DepDigger war die Auswahl auf Java-Progamme beschränkt.

2. Der Quelltext der Programme musste zugänglich sein.

3. Um eine Aussage über den Zusammenhang zwischen den Softwaremaßen und der Fehlerhäufigkeit treffen zu können, war es notwendig, Programme zu finden, die bezüglich eindeutig identifizierter Fehler sowohl in einer fehlerbehafteten als auch in einer fehlerfreien Version vorliegen.

Während die ersten zwei Kriterien von einer Fülle im Internet zugänglicher Programme erfüllt werden, wird die Auswahl durch die dritte Bedingung stark eingeschränkt. Zwar ist heutzutage die Verwendung von Versionsverwaltungssystemen und Software zur Dokumentation von Fehlern gerade auch im Bereich der Open-Source-Software üblich, die Fehlerbehebung jedoch erfolgt nicht immer in separat erfassten Aktualisierungen des Programms. Die in der Versionsverwaltung dokumentierten Schritte sind häufig ein Konglomerat aus Funktionserweiterungen, der Refaktorierung des Systems und der Behebung von Fehlern, wobei sich eine einzelne Fehlerbehebung über mehrere solche Schritte verteilen kann.

Letztlich fiel die Wahl auf drei Softwaresysteme, welche durch ein Projekt der Universität des Saarlandes (in Kooperation mit der Universität von Calgary) unter dem Titel *iBugs*[3] bereits in nahezu idealer Weise für die hier durchgeführte Analyse vorbereitet waren.

AspectJ

Beim Projekt *AspectJ*[4] handelt es sich um einen Compiler, der die aspektorientierte Programmierung in Java ermöglicht. Sowohl bezüglich der Lines of Code als auch gemessen an der Anzahl der dokumentierten Fehler ist es das größte der drei Systeme (Tabelle 3.1) und wurde in der Version Version 1.3 im Projekt iBugs integriert. AspectJ wird von der Eclipse Foundation betreut.

[3] `http://www.st.cs.uni-saarland.de/ibugs/`
[4] `http://www.eclipse.org/aspectj/`

	AspectJ	Rhino	Joda-Time
Anz. dokumentierter Fehler	350	32	8
Größe des Repositorys (in MB)	260	7	6
Größe des Quelltextes (in kLOC)	75	49	73

Tabelle 3.1: Vergleich der Softwaresysteme AspectJ, Rhino und Joda-Time

Joda-Time

Joda-Time[5] ist eine Java-Bibliothek, welche die Verwendung von Datums- und Zeitangaben in Java leichter und effizienter gestalten soll. Im iBugs-Projekt ist diese derzeit in Version 0.1 eingepflegt. Sie besitzt nur wenige dokumentierte Fehler, wurde aber der Vollständigkeit halber in die Analyse miteinbezogen.

Rhino

Rhino[6] ist ein in Java geschriebener Interpreter für JavaScript und wurde ebenfalls in der Version 0.1 von der Universität des Saarlandes bearbeitet. Da die anderen Testsysteme sich in der Anzahl ihrer Quelltextzeilen sehr ähnlich sind, ist das diesbezüglich deutlich kleinere Rhino eine willkommene Erweiterung der Testumgebung.

Vorteile von iBugs

Alle drei Softwaresysteme werden von der Universität des Saarlandes als sogenannte Repositorys des Versionsverwaltungssystems Subversion[7] angeboten. Zu jedem der Systeme ist eine Reihe von Fehlern dokumentiert, die aus der Versionsgeschichte des jeweiligen Systems gewonnen wurden. Die Repositorys sind derart gestaltet, dass zu jedem der dokumentierten Fehler eine komplette Version der Software vor und nach der Behebung des Fehlers erzeugt und betrachtet werden kann. Dies ermöglicht es, den Einfluß von Fehlern und Fehlerbehebungen auf die Softwaremaße zu erfassen und zu bewerten.

Im Gegensatz zu anderen Projekten, die auf ihre Eignung für die vorliegende Arbeit hin untersucht wurden, handelt es sich bei iBugs um eine Sammlung „realer“ Softwareprodukte. Weder die Programme selbst noch ihre Fehler wurden synthetisch zu Testzwecken erstellt.

Auch die im Projekt enthaltenen Metadaten erwiesen sich für die Analyse als äußerst nützlich. Zu jedem der drei Programme ist eine Auflistung der dokumentierten Fehler, samt einer Beschreibung derselben, enthalten. Zusätzlich sind für jeden dieser Fehler die im Zuge der Fehlerbehebung veränderten Dateien verzeichnet. Die vorhandenen Daten erleichterten die automatisierte Bearbeitung erheblich.

3.3 Ermittlung der Messwerte

Die Ermittlung der Messwerte sowie die spätere statistische Anaylse wurde für jedes der Testsysteme getrennt durchgeführt. Dazu wurden zunächst zu jedem dokumentierten Fehler die beiden zugehörigen Programmversionen aus den Repositorys extrahiert: Die Version vor und nach der Behebung des Fehlers, im Folgenden Prefix- und Postfix-Version genannt. Jegliche Berechnungen durch DepDigger

[5] `http://joda-time.sourceforge.net/`
[6] `http://www.mozilla.org/rhino/`
[7] `http://subversion.apache.org/`

Project	BugID	Method
rhino	114493	./mozilla/js/rhino/examples/Shell.java // Shell.quit()
rhino	137181	./mozilla/js/rhino/examples/Shell.java // Shell.quit()
rhino	137181	./mozilla/js/rhino/examples/File.java // File.File()

Tabelle 3.2: Verkürzt dargestellter Auszug aus der Ergebnistabelle für das Projekt Rhino. Am Beispiel soll nur die Identifikation einer mehrfach gemessenen Methode verdeutlicht werden. Die Spalten der Messergebnisse wurden daher weggelassen.

erfolgten im Weiteren auf allen Prefix- und Postfix-Versionen aller dokumentierten Fehler eines Testsystems. Aufgrund dieser Tatsache wurden die Maße einer Methode, abhängig vom Kontext, mehrmals berechnet. Um Methoden dennoch eindeutig einer spezifischen Programmversion zuordnen zu können, wurden diese anhand des Projektnamens, der Fehlernummer und der zugrunde liegenden Verzeichnisstruktur identifiziert (Tabelle 3.2).

In einem ersten Durchlauf von DepDigger wurden all jene Dateien herausgesucht, für die aufgrund eines Syntaxfehlers innerhalb ihres Quelltextes keine korrekte Berechnung der Maße gewährleistet werden konnte (siehe 3.1). Derartige Dateien wurden sowohl aus der Prefix- als auch der Postfix-Version des zugehörigen Fehlers gelöscht. Besonders das Projekt AspectJ war davon betroffen, da in einem großen Teil der Testfälle des Systems aspektorientierte Elemente enthalten waren, die nicht von der Java-Sprachspezifikation abgedeckt werden.

Nachdem in einem erneuten Durchlauf die Softwaremaße der Testsysteme errechnet worden waren, mussten aus den Messergebnissen alle Methoden anonymer Klassen entfernt werden. Solche Methoden waren nur durch die Zeilenangabe des Methodenkopfes im Quelltext eindeutig identifizierbar. Da Änderungen von Programmstellen, die im Quelltext vor einer derartigen Methode stehen, diesen Identifikator unvorhersehbar verändern, war es nicht möglich, die Prefix- und Postfix-Versionen anonymer Methoden zu vergleichen.

Um bei der statistischen Auswertung erkennen zu können, welche Methoden an der Behebung eines Fehlers beteiligt waren, wurden zusätzlich alle Methoden, die beide der folgenden Bedingungen erfüllten, entsprechend gekennzeichnet:

- Der Wert eines Maßes der Methode wurde durch die zugehörige Fehlerbehebung verändert.
- Die Methode war Teil einer Datei, die in den Metadaten des Testsystems als am Fehler beteiligt gekennzeichnet war.

Im Anschluß an die angesprochenen Einschränkungen und Modifikationen wurde schließlich eine statistische Anaylse der Messdaten durchgeführt. Diese wird im folgenden Kapitel besprochen.

Kapitel 4

Auswertung

In den folgenden Abschnitten werden der Ablauf der statistischen Auswertung der Messdaten sowie die Ergebnisse derselben vorgestellt. Außerdem wird der Zusammenhang zwischen der Fehlerhäufigkeit von Programmen und den untersuchten Softwaremaßen diskutiert. Desweiteren wird die Validität der Resultate besprochen und dabei insbesondere Probleme aufgezeigt, welche die Gültigkeit der hier getroffenen Aussagen gefährden können.

4.1 Statistische Auswertung

Die gesammelten Messdaten wurden mithilfe des Werkzeugs R[1] statistisch untersucht. Hierbei waren zwei Fragestellungen von Interesse:

1. Besitzen fehlerhafte Methoden höhere Messwerte als fehlerfreie?

2. Wie entwickeln sich die Messwerte bei der Behebung eines Fehlers?

Zu deren Beantwortung wurden die Messergebnisse für jedes Testsystem getrennt betrachtet und bezüglich der einzelnen Softwaremaße separat analysiert. Für die statistischen Tests wurde ein Signifikanzniveau von 5 % gewählt.

4.1.1 Vergleich fehlerhafter und fehlerfreier Methoden

Um die erste Frage zu klären, wurden zu jedem dokumentierten Fehler jene Methoden betrachtet, die aus der zugehörigen Prefix-Version stammen. Die Menge all dieser Methoden wurde zunächst in zwei Partitionen unterteilt. Einerseits die der fehlerhaften, welche zur Behebung des jeweiligen Fehlers verändert werden mussten, andererseits die der fehlerfreien Methoden. Für beide Partitionen wurde anschließend für alle Maße sowohl das arithmetische Mittel als auch der Median errechnet und die Mittelwerte der fehlerfreien mit denen der fehlerhaften Methoden verglichen. Zudem wurde mithilfe des Mann-Whitney-U-Tests [Geo09] geprüft, ob der Unterschied der beiden Partitionen im Bezug auf die Messwerte signifikant ist.

[1] `http://www.r-project.org/`

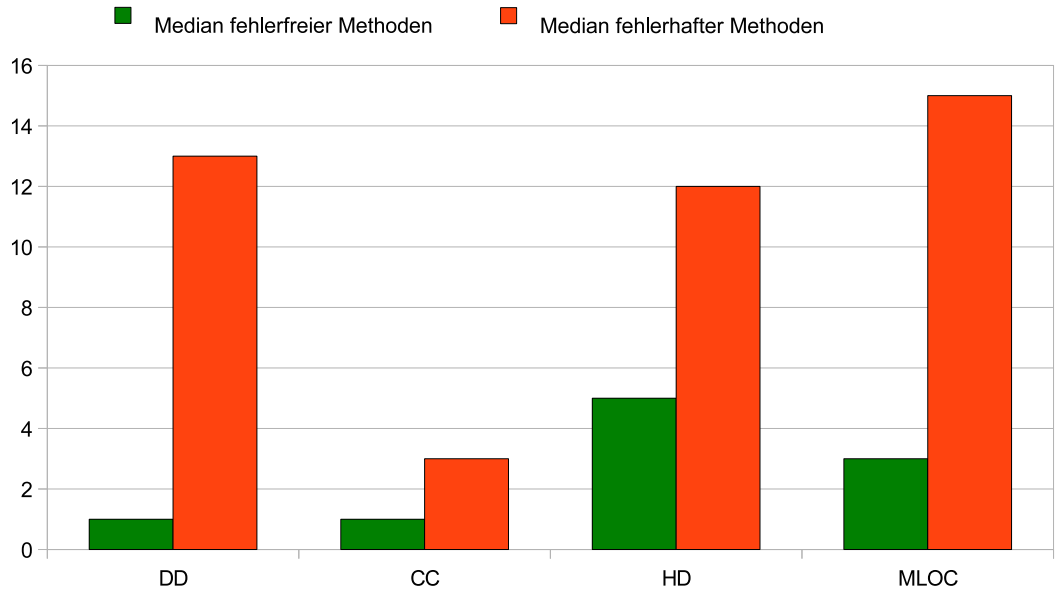

Abbildung 4.1: Vergleich der Mediane fehlerfreier und fehlerhafter Methoden am Beispiel AspectJ

Ergebnis

Die Analyse lieferte hierzu ein eindeutiges Ergebnis. Die Mittelwerte der fehlerhaften Methoden lagen bezüglich aller Softwaremaße über denen der fehlerfreien. Dies war sowohl bei der Betrachtung der arithmetischen Mittelwerte als auch der Mediane (Abb. 4.1) der Fall. Desweiteren war der Unterschied der Messwerte beider Partitionen in allen Fällen signifikant.

4.1.2 Auswirkung einer Fehlerbehebung auf die Maße

Im zweiten Teil der Auswertung wurde untersucht, welche Auswirkung die Behebung eines Fehlers auf die Softwaremaße hat. Da sich trivialerweise die Maße einer fehlerfreien (also nicht von der Fehlerbehandlung betroffenen) Methode nicht ändern, wurden nur fehlerhafte betrachtet. Für jeden dokumentierten Fehler wurden die Prefix- mit den Postfix-Versionen verglichen. Danach wurden die Methoden für jedes Maß bezüglich ihrer Tendenz in verbesserte, verschlechterte und unveränderte kategorisiert. Ein durch die Fehlerbehebung bedingtes Absinken des Maßes bedeuted eine Verbesserung, die Erhöhung eine Verschlechterung. Schließlich wurden die Kategorien auf die Anzahl der in ihnen enthaltenen Methoden hin untersucht. Abschließend wurde unter Verwendung des Chi-Quadrat-Tests [Geo09] errechnet, ob sich die gegebene Verteilung auf die Kategorien signifikant von einer Gleichverteilung unterscheidet.

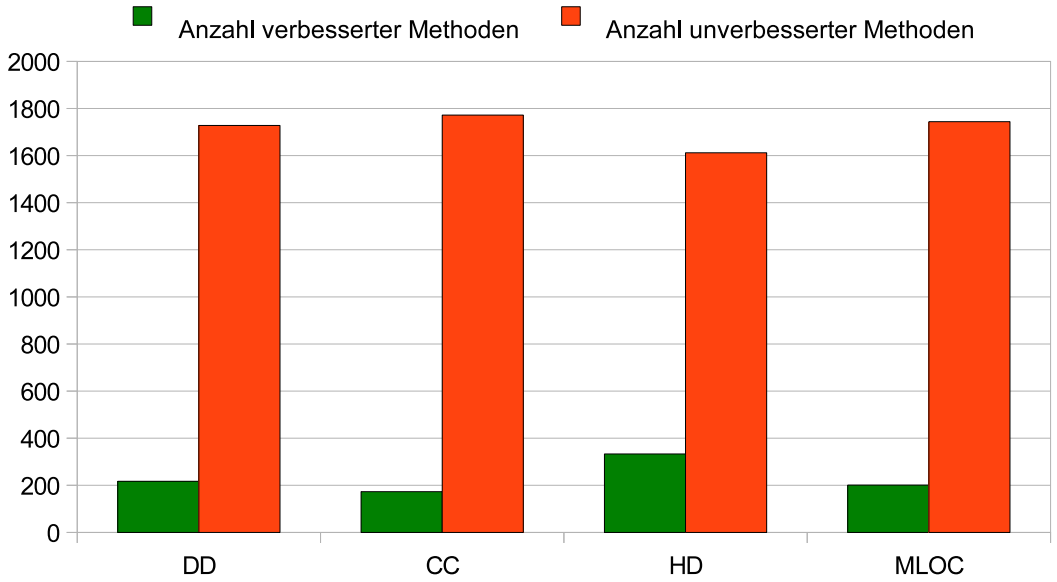

Abbildung 4.2: Vergleich der Anzahl verbesserter und unverbesserter Methoden am Beispiel AspektJ

Ergebnis

In einer ersten Untersuchung sollte geklärt werden, ob im Zuge einer Fehlerbehebung eher mit einer Verbesserung oder Verschlechterung der Maße zu rechnen ist. Daher wurden zunächst die Anzahl der verbesserten Methoden der Anzahl der verschlechterten gegenübergestellt. Diese Betrachtung lieferte jedoch keine eindeutigen Ergebnisse. Während in den Systemen AspectJ und Joda-Time die Anzahl der verschlechterten Methoden weit über der Anzahl der verbesserten lag, war im Programm Rhino entweder das Gegenteil der Fall oder die beiden Werte hielten sich die Waage.

Aus diesem Grund wurden in einer zweiten Untersuchung die Kategorien der verschlechterten und der unveränderten Methoden zusammengefasst und somit lediglich analysiert, ob eine Verbesserung der Maße zu erwarten ist. Nach dieser Einschränkung bot sich ein im wesentlichen einheitliches Bild (Abb. 4.2). Bei allen Testsystemen war die Anzahl der verbesserten Methoden bezüglich aller Maße, bis auf eine Ausnahme (Tabelle 4.1), signifikant geringer als die Anzahl derer, die entweder verschlechtert wurden oder unverändert blieben.

4.2 Interpretation

Aus den Ergebnissen der statistischen Auswertung lassen sich einige Schlüsse ziehen, die zur Klärung der Grundfrage dieser Arbeit, nach dem Zusammenhang zwischen der Fehlerhäufigkeit von Programmen und den untersuchten Softwaremaßen, beitragen.

	DD	CC	HD	MLOC
Anz. verbesserter Methoden	131	118	180	134
Anz. unverbesserter Methoden	185	198	136	182

Tabelle 4.1: Gegenüberstellung der Methoden des Programms Rhino, die sich durch eine Fehlerbehebung verbessert bzw. nicht verbessert haben. Ausschließlich beim Maß Halstead Difficulty überwiegt die Anzahl der verbesserten Methoden.

Wie sich gezeigt hat, besitzen fehlerhafte Methoden im Mittel deutlich höhere Messwerte als fehlerfreie. Daraus lässt sich schließen, dass das Auftreten eines Fehlers in Methoden, die hohe Messwerte erreichen, wahrscheinlicher ist, als in solchen, deren Werte niedrig sind. Fehler treten also häufiger in durch die Maße schlecht bewerteten Methoden auf. Diese Aussage kann für alle der hier untersuchten Maße getroffen werden. Unter der schon in der Zielsetzung angesprochenen Annahme (siehe 1.2), dass Fehler eines Programms üblicherweise in komplexen, schwer verständlichen Programmteilen vorkommen, bestätigen die Ergebnisse also die Eignung der analysierten Softwaremaße als Indikator für die Komplexität eines Programms.

Weiterhin ergibt sich aus den Untersuchungen, dass die Reduzierung der Fehlerhäufigkeit durch die Behebung eines Fehlers im allgemeinen keinen positiven Einfluss auf die untersuchten Maße hat. Betrachtet man die Maße als Indikator der Programmkomplexität, so folgt, dass im Gegensatz zur Refaktorierung eines Programms nach einer Fehlerbehebung nicht mit einer geringeren Komplexität und somit besseren Verständlichkeit zu rechnen ist. Die unterschiedlichen Ergebnisse der drei Testsysteme bezüglich des direkten Vergleichs verbesserter und verschlechterter Methoden sind ein Indiz dafür, dass die durch eine Fehlerbehebung zu erwartende Auswirkung auf die Komplexität stark von der Art und Struktur des untersuchten Programms und der beteiligten Entwickler abhängt.

Beim Vergleich der Maße untereinander konnten zwei Beobachtungen gemacht werden. Zum einen hob sich das Maß *Halstead Difficulty* bei zwei der drei Testsysteme durch eine wesentlich höhere Anzahl verbesserter Methoden deutlich von den anderen Maßen ab. Dies spielt jedoch für die Beurteilung seiner Aussagekraft bezüglich der Komplexität keine Rolle. Zum anderen waren die Unterschiede der Mediane fehlerhafter und fehlerfreier Methoden beim Maß *Cyclomatic Complexity* relativ gering. Zu einer genaueren Differenzierung der Komplexität verschiedener Programmteile empfiehlt es sich daher, eines der anderen Maße zu wählen. Besonders bemerkenswert ist in diesem Zusammenhang der Vergleich der DepDegree-Mittelwerte der fehlerhaften Methoden des Projekts Rhino (Tabelle 4.2). Einige starke Ausreißer führten hier zu einer enormen Abweichung zwischen den Mittelwerten. Derart extreme Einzelfälle waren bei keinem der anderen Maße zu beobachten. Im Hinblick auf die Aussagekraft bezüglich der Komplexität konnte sich letztlich keines der Maße von den anderen abheben.

Anwendung der Maße

Aufgrund der Ergebnisse kommt als eines der möglichen Anwendungsgebiete der Maße die Fehlersuche in Softwaresystemen in Betracht. Zwar können durch die hier aufgeführten Maße keine definitiven Aussagen über das Vorhandensein eines Fehlers in einem bestimmten Programmabschnitt erfolgen, das Auffinden eines solchen kann jedoch eventuell beschleunigt werden, wenn der Fokus der Entwickler zunächst auf Programmteile mit hohen Messwerten gerichtet wird. Ob eine derartige Anwendung tatsächlich von Nutzen ist, wird in dieser Arbeit allerdings nicht überprüft.

4.3 Validität

Zum Zeitpunkt der Analyse war, wie bereits in Unterkapitel 3.1 angesprochen, lediglich die Untersuchung von Java-Programmen möglich. Von einer Allgemeingültigkeit der Ergebnisse bezüglich anderer Programmiersprachen kann daher nicht ausgegangen werden. Dennoch wurde versucht, durch die Wahl großer, unabhängiger und vor allem realer Testsysteme zumindest im Bereich der Java-Entwicklung einen ganzheitlichen Blick auf die untersuchten Maße zu ermöglichen.

Mit der Wahl realer Systeme waren allerdings auch einige Schwierigkeiten verbunden, die aufgrund ihres eventuellen Einflusses auf die Ergebnisse nicht unerwähnt bleiben dürfen. Die Erstellung der Repositorys der Testsysteme durch die Universität des Saarlandes erfolgte großteils automatisiert [DZ07]. Das dazu verwendete Verfahren stützt sich auf die Analyse der Kommentare, die Entwickler im Zuge einer Änderung im Versionsverwaltungssystem hinterlegen. Da es letztlich in der Verantwortung der Entwickler liegt, welche Informationen diese Kommentare enthalten, kann nicht garantiert werden, dass zu jedem der dokumentierten Fehler tatsächlich nur Änderungen erfasst wurden, die zur Behebung des jeweiligen Fehlers notwendig waren. Dies und die Tatsache, dass Methoden anonymer Klassen ebenso unberücksichtigt bleiben mussten, wie Dateien, die nicht der Java-Sprachspezifikation genügten (siehe 3.3), können Ursachen möglicher Ungenauigkeiten sein. Aufgrund stichprobenartiger Überprüfungen der betroffenen Dateien besteht jedoch Grund zu der Annahme, dass die angesprochenen Faktoren nur geringfügigen Einfluss auf das Gesamtergebnis hatten.

	DD	CC	HD	MLOC
Arith. Mittel	531	22	27	86
Median	14	4	14	16

Tabelle 4.2: Vergleich der Mittelwerte fehlerhafter Methoden im Projekt Rhino

4.4 Schlusswort

Die durchgeführte Untersuchung zeigt, dass die betrachteten Softwaremaße durchaus zur Bewertung der Komplexität eines Programmes herangezogen werden können. Allerdings wurde hier ausschließlich die Fehlerhäufigkeit als Bemessungsgrundlage betrachtet. Es ist daher sicherlich sinnvoll, die hier entstandenen Ergebnisse mit denen der anderen zu diesem Thema vorhandenen Arbeiten (siehe 1.2) zu verknüpfen und letztlich eine ganzheitlichere Bewertung der Softwaremaße vorzunehmen. Auch eine Ausweitung der Testumgebung auf C-Programme ist dank der kürzlichen Weiterentwicklung des Werkzeugs DepDigger denkbar und sollte in Erwägung gezogen werden.

Literaturverzeichnis

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, second edition, 2007.

[Bal00] Helmut Balzert. *Lehrbuch der Software-Technik I, Software Entwicklung*. Spektrum Akademischer Verlag, second edition, 2000.

[Bal10] Dmitry Balzer. Werkzeugunterstützung für verstehen und monitoring von software-abhängigkeiten. Master's thesis, Universität Passau, 2010.

[BF10] Dirk Beyer and Ashgan Fararooy. A simple and effective measure for complex low-level dependencies. In *Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC)*, pages 80–83. IEEE Computer Society Press, 2010.

[DZ07] Valentin Dallmeier and Thomas Zimmermann. Automatic extraction of bug localization benchmarks from history. Technical report, Saarland University, 2007.

[Geo09] Hans-Otto Georgii. *Stochastik, Einführung in die Wahrscheinlichkeitstheorie und Statistik*. de Gruyter, fourth edition, 2009.

[Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[McC76] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, pages 407–. IEEE Computer Society Press, 1976.

[Tho89] William Thomson. *Popular Lectures and Addresses*, volume 1. Macmillan, 1889.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und nur mit erlaubten Hilfsmitteln angefertigt habe. Sie wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Passau, 21. März 2012 Stefan John