Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Masterarbeit

Verbindung von Variability-aware Code-Smells zu Problemen in der Softwareentwicklung

Autor:

Hannes Klawuhn

1. Mai 2016

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake

Institut für technische und betriebliche Informationssysteme

Dipl.-Inf. Wolfram Fenske

Institut für technische und betriebliche Informationssysteme

M.Sc. Reimar Schröter

Institut für technische und betriebliche Informationssysteme

Klawuhn, Hannes:
Verbindung von Variability-aware Code-Smells zu Problemen in der Softwareentwicklung
Masterarbeit, Otto-von-Guericke-Universität Magdeburg, 2016.

Inhaltsangabe

Code-Smells sind ein etabliertes Konzept um schlecht designten Code in einem Software-System zu beschreiben. Um schlecht designten Code in hochkonfigurierbaren Software-Systemen beschreiben zu können wurden die Variability-aware Code-Smells eingeführt. Wir untersuchen den Einfluss von Variability-aware Code-Smells auf Dateien in hochkonfigurierbaren Software-Systemen, indem wir die Beziehung zwischen dem Vorkommen von Code-Smells und der Fehleranfälligkeit der Dateien überprüfen. Zu diesem Zweck bringen wir Smell-behaftete Dateien und Bugfixes miteinander in Verbindung. Zur Smell-Detektion benutzen wir den Prototypen SKUNK und zur Bugfix-Klassifizierung wurde ein textbasierter Ansatz gewählt. Wir untersuchen für insgesamt sechs Systeme, inwieweit Smell-behaftete Dateien eine höhere Wahrscheinlichkeit haben fehlerbehebenden Änderungen unterzogen zu werden. Wir zeigen, dass Dateien, die von Variability-aware Code-Smells betroffen sind, häufiger fehlerbehebend geändert werden als nicht Smell-behaftete Dateien. Weiterhin zeigen wir, dass die Größe einer Datei alleine nicht verantwortlich für eine höhere Fehleranfälligkeit ist. Der Einfluss von Variability-aware Code-Smells auf die Fehleranfälligkeit ist somit zumindest teilweise unabhängig von der Dateigröße. Wir liefern mit unserer Studie empirische Beweise für den vermuteten negativen Einfluss von Variability-aware Code-Smells auf Bereiche der Software-Entwicklung.

Inhaltsverzeichnis

Abbildungsverzeichnis viii

Tabellenverzeichnis x

Quelltextverzeichnis xi

Abkürzungsverzeichnis xiii

1 Einführung 1

2 Grundlagen 5
2.1 Hochkonfigurierbare Software-Systeme . . . 5
2.2 Code-Smells . . . 8
2.2.1 Erkennung von Code-Smells . . . 10
2.3 Statistische Tests . . . 12

3 Konzept 17
3.1 Variability-aware Code-Smells . . . 17
3.1.1 Annotation Bundle und Annotation File . . . 19
3.1.2 Large Feature . . . 22
3.2 Zielstellung . . . 24
3.3 Vorgehen . . . 25
3.4 Umsetzung . . . 28
3.4.1 Phase 1 – Smell-Analyse . . . 28
3.4.2 Phase 2 – Korrelation der Daten . . . 31
3.5 Zusammenfassung . . . 32

4 Evaluierung 33
4.1 Allgemeines Setup . . . 33
4.1.1 Parameter für die Smell-Analyse in SKUNK . . . 33
4.1.2 Festlegen des Commit-Fensters . . . 35
4.1.3 Fallbeispiele . . . 36
4.2 Methodik . . . 38
4.3 Ergebnisse . . . 39
4.3.1 RQ1 – Einfluss von Variability-aware Code Smells auf die Fehleranfälligkeit . . . 40
4.3.2 RQ2 – Einfluss spezifischer Code-Smells auf die Fehleranfälligkeit . . . 45

4.3.3 RQ3 – Zusammenhang von Dateigröße, Präsenz von Code-Smells und Fehleranfälligkeit 48
4.4 Diskussion 51
4.5 Threats to Validity 54
4.6 Zusammenfassung 55

5 Verwandte Arbeiten **57**

6 Zusammenfassung und Zukünftige Arbeiten **59**

A Anhang **63**
A.1 Detaillierte Daten für RQ 1 und RQ 3 63

Literaturverzeichnis **69**

Abbildungsverzeichnis

2.1 Beispiel eines Feature-Diagramms 7

3.1 Vereinfachter Prozessüberblick 26
3.2 Schematische Darstellung der Phase der Smell-Analyse 28
3.3 Auszug aus den gewonnenen Daten der Commit-Analyse 29
3.4 Auszug aus dem Severity-Report des Beispielsystems OpenVPN . . . 30
3.5 Schematische Darstellung der Phase der Datenkorrelation und Evaluation 31
3.6 Auszug aus den korrelierten Daten eines Snapshots aus OpenVPN . 31

4.1 Anteil an Bugfix-Commits in den Fallbeispielen 38
4.2 Grafische Darstellung der Odds Ratios unter Einbeziehung der Ergebnisse des exakten Tests nach Fisher 41
4.3 Vergleich der Odds Ratios aller Systeme 42
4.4 Übersicht über die Anteile der Smell-behafteten Dateien (blau) und der Bugfixes in Smell-behafteten Dateien (rot) aller Beispielsysteme . 43
4.5 Vergleich der Snapshots 1-15 und 16-41 des Systems BusyBox für die Werte des Fisher-Tests (links) und der Odds Ratios (rechts) 44
4.6 Grafische Darstellung der prozentualen Anteile von Smell-behafteten Dateien für die Variability-aware Code-Smells Annotation Bundle, Annotation File, Large Feature und den Fall, dass eine Datei von mindestens einem Smell betroffen ist 46
4.7 Darstellung der Korrelation zwischen Annotation Bundle und Annotation File in den sechs Systemen mit Anzahl der Snapshots als Beschriftung 47
4.8 Grafische Darstellung der ORs unter Einbeziehung der Ergebnisse des exakten Tests nach Fisher für jeden Code-Smell und jedes System . . 48
4.9 Vergleich der Dateigrößen von Smell-behafteten und nicht Smell-behafteten Dateien aller untersuchten Systeme 49

4.10 Grafische Darstellung der Odds Ratios unter Einbeziehung der Ergebnisse des exakten Tests nach Fisher bei großen Dateien 50

4.11 Übersicht über die Anteile der Smell-behafteten Dateien (blau) und der Bugfixes in Smell-behafteten Dateien (rot) aller Beispielsysteme (nur große Dateien) 51

Tabellenverzeichnis

2.1 Beispieldaten für die Spearman-Korrelation 13

2.2 Beispiel: Kontingenztabelle, Menschen klassifiziert nach Geschlecht und Schulleistung 14

2.3 übrige Kontigenztabellen des Beispiels 14

3.1 Beispiel: Kontingenztabelle, Dateien klassifiziert nach Smell-behaftung und fehlerbehebenden Änderungen 32

4.1 Analysierte Systeme und ihre Anwendungsdomänen 37

4.2 Übersicht über die Charakteristika der Fallbeispiel-Systeme 37

4.3 Überblick über den minimalen, maximalen und mittleren prozentualen Anteil an Smell-behafteten Dateien für Annotation Bundle, Annotation File, Large Feature und den Fall, dass eine Datei von mindestens einem Smell betroffen ist 45

4.4 Übersicht über die Effektstärke der Größe von einzelnen Code-Smells über alle Beispielsysteme, rot markiert sind kleine Effektstärken . . . 49

A.1 Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System BusyBox (RQ1) 63

A.2 Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System Apache (RQ1) 64

A.3 Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System OpenLDAP (RQ1) 64

A.4 Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System OpenVPN (RQ1) 64

A.5 Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System Pidgin (RQ1) 65

A.6 Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System SQLite (RQ1) 65

A.7 Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System BusyBox (RQ3) 65

A.8 Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System Apache (RQ3) 66

A.9 Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System OpenLDAP (RQ3) 66

A.10 Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System OpenVPN (RQ3) 66

A.11 Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System Pidgin (RQ3) 67

A.12 Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System SQLite (RQ3) 67

Quelltextverzeichnis

2.1 Beispiel der Präprozessor-Nutzung – Kompletter Quellcode 8

2.2 Beispiel der Präprozessor-Nutzung – mit ROTATE_THIRD 9

2.3 Beispiel der Präprozessor-Nutzung – ohne ROTATE_THIRD 9

2.4 Beispiel des Code-Smells Long Method. Auszug aus screen.c aus VIM. 10

2.5 Beispiel des Code-Smells Duplicated Code. Auszug aus osd.c aus MPlayer. 11

3.1 Potentiell Smell-behafteter annotierter Code mit sechs verschiedenen Features 19

3.2 Beispiel einer Funktion die mit dem Annotation Bundle Code-Smell behaftet ist. Auszug aus einer MySQL-Funktion 20

3.3 Beispiel einer Funktion die mit dem Large Feature Code-Smell behaftet ist. Auszug aus OpenVPN. 23

Abkürzungsverzeichnis

AB	Annotation Bundle
AF	Annotation File
CPP	C-Präprozessor
LF	Large Feature
LOAC	Anzahl der annotierten Codezeilen
LOC	Codezeilen
NOFC	Anzahl der Feature Konstanten
NOFL	Anzahl der Feature Locations
NS	Nesting-Summe
OR	Odds Ratio

1. Einführung

„Code-Smells“ sind ein von den erfahrenen Software-Entwicklern Fowler et al. eingeführtes, etabliertes Konzept für schlecht geschriebenen bzw. schlecht strukturierten Code [Fow99]. Zwar ist Smell-behafteter Code kein fehlerhafter Code jedoch wird in der Fachliteratur angenommen, dass Code-Smells einen negativen Einfluss auf Bereiche der Softwareentwicklung wie Programmverständnis, -wartung und -evolution haben [KDPGA12, FSMS15, FS15]. Die Forschung hat sich in den letzten Jahren damit beschäftigt wie diese Smells im Programmcode entdeckt werden können, unter anderem auch automatisiert [MGDLM10] und mit Toolsupport (bspw. jCOSMO) [VEM02]. Weitere Studien untersuchen die Entfernung von Code-Smells [Moh07], also die Verbesserung der Codestruktur [Fow99], und deren Wirkung auf die Softwareentwicklung [KDPGA12]. Diese Arbeiten identifizieren Code-Smells als problematisch für die Softwareentwicklung. Die Studie von Khomh et al. liefert sogar empirische Beweise für eine von Code-Smells verursachte höhere Fehleranfälligkeit. Alle zuvor beschriebenen Arbeiten, inklusive der von Fowler et al., haben dabei gemein, dass sie nur *objektorientierte Programmierung* untersuchen bzw. auf *nicht-konfigurierbare Software-Systeme* ausgelegt sind.

Hochkonfigurierbare Software-Systeme (auch bekannt als Software-Produktlinien) gewinnen sowohl in der Forschung als auch in der Industrie immer mehr an Bedeutung [ABKS13]. Software-Produktlinien haben eine gemeinsame Code-Basis, die für verschiedene Anforderungen erweitert werden kann. Dadurch lassen sich ähnliche Programme schnell und mit wenig Aufwand entwickeln, ohne jedes Produkt von Grund auf neu zu konzipieren. Dies spart Zeit und Geld, beinhaltet jedoch auch neue Risiken für schlechte Designentscheidungen.

Fenske und Schulze haben in ihrer Arbeit gezeigt [FS15], dass die etablierten Code-Smells nicht mehr ausreichen um die Schwächen hochkonfigurierbaren Programmcodes treffend zu beschreiben sobald die Variabilität mit einbezogen wird. Daraufhin haben sie, abgeleitet von den von Fowler et al. eingeführten Smells, vier „Variability-aware“ Code-Smells vorgeschlagen [FS15]. Darauf aufbauend haben Fenske et al. das Tool SKUNK zur automatisierten Smell-Erkennung entwickelt und anschließend mithilfe dessen gezeigt, dass der neu eingeführte Smell „Annotation Bund-

LE“ in Software-Systemen aus der Praxis auftritt [FSMS15]. Des weiteren haben sie in einer Umfrage mit 15 Forschern aus dem Bereich der Software-Produktlinien die Annahme bestätigt, dass die Variability-aware Code-Smells in realen System vorhanden sind und einen negativen Einfluss auf Bereiche der Softwareentwicklung, beispielsweise Programmverständnis, -wartung und -evolution, besitzen. Diese subjektive Einschätzung der Forscher soll in der folgenden Arbeit mit einer empirischen Studie untermauert werden.

Um zu untersuchen, inwiefern Variability-aware Code-Smells einen messbaren Einfluss auf die Softwarequalität haben, wird in dieser Arbeit insbesondere geprüft, ob Smell-behafteter Code fehleranfälliger ist als anderer. Für den Bereich der objektorientierten Entwicklung haben Khomh et al. bereits eine empirische Studie zu diesem Thema durchgeführt [KDPGA12]. Diese hat gezeigt, dass in der objektorientierten Programmierung Code-Smells einen negativen Einfluss auf die Softwareentwicklung haben und die Fehleranfälligkeit erhöhen. In der vorliegenden Arbeit soll diese Studie auf hochkonfigurierbare Software-Systeme übertragen werden. So wurde auch ein Teil der Forschungsfragen aus der Studie von Khomh et al. für diese Arbeit adaptiert. Dementsprechend sollen folgende Fragen beantwortet werden:

RQ 1 Gibt es einen Zusammenhang zwischen Code-Smells und der Fehleranfälligkeit?

RQ 2 Gibt es einen Zusammenhang zwischen bestimmten Arten von Code-Smells und der Fehleranfälligkeit?

RQ 3 Ist der Zusammenhang zwischen Code-Smells und der Fehleranfälligkeit unabhängig von der Dateigröße?

Zu diesem Zweck müssen Smell-behaftete Codestellen in hochkonfigurierbaren Software-Systemen gefunden und mit den Fehlern im Code in Verbindung gebracht werden. Da es nicht trivial ist, Fehler in einem Quellcode festzustellen – schon gar nicht automatisch – gehen wir einen anderen Weg. Wir suchen nach fehlerbehebenden Änderungen, sogenannten Bugfixes, und klassifizieren jede Datei, in der mindestens ein Bugfix stattfand, als fehlerhaft. Die dafür nötigen Daten gewinnen wir in unserer Fallstudie von sechs mit Präprozessor entwickelten hochkonfigurierbaren Software-Systemen. Die sechs untersuchten Systeme BusyBox, Apache, OpenLDAP, OpenVPN, Pidgin und SQLite, haben unterschiedliche Größen und kommen aus unterschiedlichen Anwendungsdomänen, um die gewonnenen Ergebnisse später zumindest teilweise generalisieren zu können. Da sich Dateien über die Zeit ändern und Code-Smells eingeführt oder entfernt werden, müssen diese in regelmäßigen Zeitabständen ihres Entwicklungsstandes auf Variability-aware Code-Smells untersucht werden. Zusätzlich wird die Commithistorie jedes Systems analysiert und auf Bugfix-Commits untersucht, da wir diese als Indikatoren für fehlerhafte Dateien nutzen. Abschließend werden die Smell-behafteten und fehlerhaften Dateien korreliert. So kann klassifiziert werden, ob fehlerhafte Dateien Smell-behaftet waren oder nicht. Durch die regelmäßig durchgeführten Smellanalysen kann für jeden Zeitabschnitt das Verhältnis von Smell-behafteten Dateien zu fehlerbehebenden Änderungen festgestellt werden.

In dieser Studie zeigen wir, dass (1) Smell-behaftete Dateien fehleranfälliger sind als nicht Smell-behaftete Dateien und (2) der Variability-aware Code-Smell Large Feature einen größeren Einfluss auf die Fehleranfälligkeit hat als die anderen untersuchten Smells. Zusätzlich zeigen wir, dass der Zusammenhang von Fehleranfälligkeit und dem Vorkommen von Code-Smells auch in großen Dateien besteht und somit teilweise unabhängig der Dateigröße ist.

Die Arbeit ist wie folgt aufgebaut: Zu Beginn der Arbeit werden in Kapitel 2 die zum Verständnis notwendigen Grundlagen dargelegt. In Kapitel 3 folgt die Darstellung unseres Konzepts. Zuerst wird das grundsätzliche Vorgehen, wie wir die benötigten Daten gewinnen, korrelieren und auswerten wollen, präsentiert. Anschließend wird an einem Beispielsystem gezeigt, wie das Konzept in eine Studie umgesetzt wird. Kapitel 4 beschreibt die Testumgebung, stellt die Ergebnisse der Studie vor und wertet diese aus. Als Abschluss der Arbeit wird auf verwandte Arbeiten eingegangen und eine Zusammenfassung, sowie ein Ausblick auf mögliche weiterführende Arbeiten gegeben.

2. Grundlagen

Der folgende Abschnitt liefert grundlegende Hintergrundinformationen um die Konzepte der Code-Smells und der hochkonfigurierbaren Software-Systeme vollständig verstehen zu können. Der erste Teil präsentiert Informationen über die Motivation und Vorteile von hochkonfigurierbaren Software-Systemen. Weiterhin wird der Annotations-basierte Ansatz solche Systeme zu implementieren näher beleuchtet. Der zweite Teil beinhaltet Details über Definitionen, Probleme und Arten von Code-Smells. Zusätzlich werden verschiedene Ansätze zur Detektion dieser Smells besprochen. Abschließend wird ein Überblick über statistische Methoden, die in dieser Arbeit benutzt werden, gegeben.

2.1 Hochkonfigurierbare Software-Systeme

Unterschiedliche Nutzer haben unterschiedliche Bedürfnisse und Wünsche an ein Produkt. Dies hat nicht nur die Automobilindustrie, die ihre Produktpalette in den letzten Jahrzehnten von ein paar Modellen mit wenigen Konfigurationsmöglichkeiten auf mehr als 10^{20} verschiedene Varianten eines Autos erweitert haben, erkannt. Der Käufer kann sich beispielsweise allein bei den Lenkrädern zwischen bis zu 50 verschiedenen Varianten entscheiden. Auch die Softwareindustrie ist in den letzten Jahren zu dieser Einsicht gekommen [FS15]. Hochkonfigurierbare Software-Systeme (auch als Software-Produktlinien (SPL) bekannt) verfolgen die Idee, auf den Nutzer maßgeschneiderte Software mit annehmbarem Kosten- und Arbeitsaufwand zu entwickeln [ABKS13, CN01]. Zu diesem Zweck sollen die Gemeinsamkeiten der einzelnen Produkte ausgenutzt werden. Eine Software-Produktlinie wird von Northrop wie folgt definiert [Nor02]:

> „A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.“

Ein hochkonfigurierbares Software-System besteht somit aus einer Menge von wiederverwendbaren Software-Artefakten, aus denen der Nutzer sich sein Wunschsystem zusammenstellt. In der Analogie der Automobilindustrie ist das beispielsweise das Lenkrad. Jedes Auto benötigt eins und doch hat der Käufer die Wahl zwischen vielen verschiedenen Varianten (Aussehen, Material, Funktionen etc.). In einem Datenbanksystem, das als SPL angelegt ist, könnte zum Beispiel der Sortieralgorithmus eines dieser Artefakte sein. Dieser ließe sich dann an die speziellen Anforderungen des Nutzers anpassen (z.B. geringere Rechenleistung in eingebetteten Systemen). So müssen Software-Systeme nicht für jeden einzelnen Anwendungsfall komplett neu implementiert werden, sondern benutzen ein und denselben Kern und erweitern diesen angepasst an die Bedürfnisse und Wünsche der Nutzer. Dabei bleiben die Software-Systeme meist auf ein Anwendungsgebiet (Domäne) spezialisiert. So gibt es variable Datenbanksysteme mit Optionen, die der Nutzer wählen kann. Diese unterscheiden sich beispielsweise bei Sortieralgorithmen oder ähnlichem. Das Anwendungsgebiet bleibt bei jeder gewählten Option jedoch im Datenbanksegment.

Die beschriebenen wiederverwendbaren Artefakte werden auch *Features* genannt. Sie repräsentieren Anforderungen und Gemeinsamkeiten, beziehungsweise Unterschiede, zwischen den einzelnen Produktvarianten. Features können optional oder obligatorisch sein, sie können einander bedingen oder ausschließen, oder aufeinander aufbauen. Je mehr Features und Abhängigkeiten es gibt, umso schwieriger wird es das System im Ganzen zu verstehen. Daher werden Beziehungen der Features untereinander unter anderem in einem *Feature-Diagramm* (auch Feature-Modell) dargestellt [KCH+90]. In Abbildung 2.1 auf der nächsten Seite ist ein Beispiel eines fiktiven Feature-Diagramms zu sehen. In solch einem Modell werden die Features durch blaue Rechtecke mit ihrem Namen dargestellt. Dabei wird zwischen abstrakten (hellblau) und konkreten (dunkelblau) Features unterschieden. Abstrakte Features sind meist Features die mehrere andere zusammenfassen und beschreiben. Während abstrakte Features nur im Feature-Modell und in der Konstruktion von Programmen zu finden sind, sind konkrete Features immer auch im Quellcode des Programms implementiert. Hier haben wir das abstrakte Basisfeature *Base*. Dieses ist immer ausgewählt und notwendig für die, sich in der Hierarchie darunter befindenden, Features *UI*, *Sounds*, *Lesen* und *Schreiben*. *UI* und *Lesen* sind dabei, erkennbar an dem grauen Kreis, verpflichtend und müssen gewählt werden, *Sounds* und *Schreiben* (weißer Kreis) hingegen sind optional. Weiterhin zeigt der Halbkreis unter *UI*, dass eins der beiden Features *Grafisch* und *Textuell* ausgewählt werden muss, wobei nicht beide gleichzeitig ausgewählt sein dürfen. Die Auswahl einer Feature-Menge definiert eine Variante eines hochkonfigurierbaren Software-Systems und wird *Konfiguration* genannt. Feature-Modelle helfen den Entwicklern bei der Erstellung von „korrekten“ Programmvarianten.

Es gibt verschiedene Implementationsansätze um hochkonfigurierbare Software-Systeme zu entwickeln. Beispielsweise gibt es Plugin-Architekturen, Feature-orientierte Programmierung und Aspekt-orientierte Programmierung, die sich alle der Klasse der *Kompositions-basierten Mechanismen* zuordnen lassen [Gri00, ABKS13]. Das Konzept der Kompositions-basierten Implementation ist die größtmögliche Modularisierung des Feature-Codes.

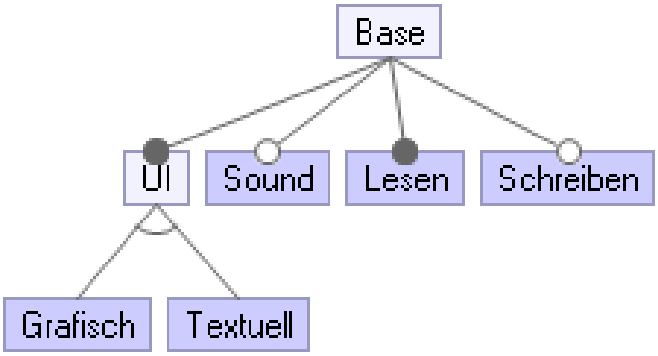


Abbildung 2.1: Beispiel eines Feature-Diagramms

Ein gegensätzliches Konzept verfolgt die Klasse der *Annotations-basierten Mechanismen*, die auch in der Praxis die vorherrschende Klasse für Software-Variabilität ist [LAL+10]. Zu diesen Mechanismen gehört insbesondere der Präprozessor, auf den sich die Untersuchungen in dieser Arbeit beziehen. Der Präprozessor folgt einem simplen Muster: Alle Features eines hochkonfigurierbaren Software-Systems sind in einer einzigen Code-Basis vereint. Dabei wird Code, der zu bestimmten Features gehört, beispielsweise mit Präprozessor-Anweisungen (z.B. `#ifdef`) annotiert. Bei der Erstellung des spezifischen (Software-)Produktes werden die annotierten Code-Fragmente dann modifiziert, beziehungsweise entfernt, je nachdem welche Features vom Nutzer ausgewählt wurden [ABKS13].

In Quelltext 2.1 und den folgenden Quelltexten wird ein simples Beispiel mit CPP-Direktiven und deren Effekt auf ein Software-System präsentiert. Der Zweck der Methode `rotate` ist das rotieren/verschieben von zwei, beziehungsweise drei, Werten. In Quelltext 2.1 sind drei `#ifdef`-Blöcke in Zeile 4, 13 und 18 zu sehen. Wenn das Feature `ROTATE_THIRD` angewählt wurde, sorgt der C-Präprozessor dafür, dass der Compiler den Quellcode aus Quelltext 2.2 erhält. In diesem Fall bekommt die Methode drei Eingabewerte und rotiert diese. Wenn das Feature nicht definiert ist, entfernt der Präprozessor die Zeilen 5, 14, 19, 20 und 21 vor der Kompilation, wie in Quelltext 2.3 zu erkennen ist. Unter diesen Umständen benutzt die Methode `rotate` lediglich zwei Parameter und rotiert/vertauscht diese zwei Werte. Zusammenfassend gibt es mit dem Feature `ROTATE_THIRD` zwei Programmvarianten, die auf der gleichen Quellcode-Basis operieren.

Das Nutzen von `#ifdef`-Blöcken im Quellcode wird oft für seine Unverständlichkeit und Anfälligkeit für kleine, schwierig zu findende Bugs, kritisiert [EBN02, SC92, MRG13, MKR+15]. Diese Probleme werden in Abschnitt 3.1 näher besprochen. Trotzdessen ist der C-Präprozessor nach wie vor eine beliebte Wahl um Variabilität in Software-Systemen einzuführen. Denn zum einen ist das Einführen und Entfernen von Features mithilfe von `#ifdef`-Direktiven leicht zu verstehen und erfordert wenig Vorplanung. Das Modell der `if-else`-Statements ist bei jedem Entwickler bekannt und kann direkt auf `#ifdef`-Direktiven angewandt werden. Zum anderen ist der C-Präprozessor Teil des C-Compilers und erfordert keine weiteren Tools zur erfolgreichen Kompilierung [KA09].

```
1  void rotate
2  ( int n1 ,
3  int n2
4  #ifdef ROTATE_THIRD
5  , int n3
6  #endif
7  )
8  {
9      \\ copy original values
10     int tn1 = n1;
11     int tn2 = n2;
12
13     #ifdef ROTATE_THIRD
14     int tn3 = n3;
15     #endif
16
17     \\ move
18     #ifdef ROTATE_THIRD
19     n1 = tn3 ;
20     n2 = tn1 ;
21     n3 = tn2 ;
22     #else
23     n1 = tn2 ;
24     n2 = tn1 ;
25     #endif
26 }
```

Quelltext 2.1: Beispiel der Präprozessor-Nutzung – Kompletter Quellcode

2.2 Code-Smells

Code-Smells, in der englischsprachigen Fachliteratur oft auch als „anti-patterns“ bezeichnet, sind ein etabliertes Konzept um schlecht designten Code in einem Software-System zu beschreiben [Fow99]. Diese Smells kennzeichnen dabei nicht direkt fehlerhaften Code, sie haben jedoch einen Einfluss auf Programmverständnis, -wartung und -evolution [KDPGA12]. Eingeführt wurde der Begriff der Code-Smells von Fowler et al. um Anhaltspunkte für schlecht strukturierten Programmcode, der ein Refactoring nötig hat, zu erarbeiten [Fow99]. Während ihre Arbeit auf das Refactoring spezialisiert ist, stellen sie zusätzlich Codemuster vor, die sie für problematisch, beziehungsweise Smell-behaftet halten. Beispiele für diese Muster, beziehungsweise Code-Smells sind: Duplizierter Code, zu große/lange Funktionen, oder Klassen, die zu viel Funktionalität beinhalten.

Lange Methode

Der Code-Smell Lange Methode beschreibt Methoden oder Funktionen, die zu viel Funktionalität beinhalten. Das sind typischerweise Methoden mit einer hohen Anzahl von Codezeilen. Die Problematik des Smells besteht darin, dass je länger beziehungsweise größer eine Methode ist, umso schwerer ist sie zu verstehen und zu pflegen. So können beispielsweise Änderungen am Anfang einer Methode zu unerwarteten Fehlern am Ende dieser führen. Eine Art dem entgegen zu wirken ist die Methode in mehrere kleine Methoden zu unterteilen. Dies ist auch eine von Fowler

```
1  void rotate
2  (int n1, int n2, int n3)
3  {
4      \\ copy original values
5      int tn1 = n1;
6      int tn2 = n2;
7      int tn3 = n3;
8
9      \\ move
10     n1 = tn3 ;
11     n2 = tn1 ;
12     n3 = tn2 ;
13 }
```

Quelltext 2.2: Beispiel der Präprozessor-Nutzung – mit ROTATE_THIRD

```
1  void rotate
2  (int n1, int n2)
3  {
4      \\ copy original values
5      int tn1 = n1;
6      int tn2 = n2;
7
8      \\ move
9      n1 = tn2 ;
10     n2 = tn1 ;
11 }
```

Quelltext 2.3: Beispiel der Präprozessor-Nutzung – ohne ROTATE_THIRD

vorgestellte Art des Refactorings. In Quelltext 2.4 ist ein Auszug aus dem Code aus einer 2864 Zeilen langen VIM-Methode zu sehen, die von dem Smell Lange Methode betroffen ist.

Duplizierter Code/Code-Klone

Ein weiteres Smell-Beispiel ist Duplizierter Code, beziehungsweise Code-Klon. Ein Code-Klon ist eine Code-Sequenz, die öfter als einmal im Software-System auftritt. Dieser kann entweder komplett identisch oder oftmals auch leicht verändert sein. Die Änderungen können unter anderem die Namensgebung der Variablen, Typänderungen oder unterschiedliche Funktionsaufrufe betreffen. Die Problematik des Smells ist der gestiegene Wartungsaufwand. Sollte beispielsweise in einem Teil des Codes, der mindestens einen Klon besitzt, ein Bug gefunden werden, muss dieser unter Umständen in mehreren Stellen im System gefixed werden. Der Entwickler muss somit mehr Zeit zum Finden und Beheben der Bugs investieren. Ein Codebeispiel ist in Quelltext 2.5 auf Seite 11 zu sehen. Das Beipiel zeigt zwei annähernd identische Methoden die sich nur in den Methodenaufrufen und ihrem Namen unterscheiden. Ihre Funktion unterscheidet sich nur in dem Farbraum, den sie adressieren - die eine adressiert den 24-Bit Farbraum, die andere den 32-Bit Farbraum. Die minimalen Änderungen können zu zwei Problemen führen. Da sich die Namen der Methoden nur minimal unterscheiden kann es leicht passieren, dass der Entwickler die falsche

```
2859 static int
2860 win_line(
2861     win_T    *wp,
2862     linenr_T     lnum,
2863     int      startrow,
2864     int      endrow,
2865     int      nochange UNUSED)    /* not updating for changed text
         */
2866 {
2867     int      col;            /* visual column on screen */
2868     unsigned     off;            /* offset in ScreenLines/
         ScreenAttrs */
2869     int      c = 0;          /* init for GCC */
```

```
5713
5714 #ifdef FEAT_SPELL
5715     /* After an empty line check first word for capital. */
5716     if (*skipwhite(line) == NUL)
5717     {
5718     capcol_lnum = lnum + 1;
5719     cap_col = 0;
5720     }
5721 #endif
5722
5723     return row;
5724 }
```

Quelltext 2.4: Beispiel des Code-Smells Long Method. Auszug aus screen.c aus VIM.

Methode aufruft. Jeder falsche Aufruf muss später vom Entwickler behoben werden. Ein weiteres Problem kann in einem der `if`-Statements auftreten, wenn sich während der Systemevolution die Bedingung ändert. Während es bei zwei Klonen noch einfach zu überblicken ist, erhöht sich die Bearbeitungszeit immer weiter je mehr Klone des gleichen Codes auftreten.

Die Forschung hat sich vermehrt in den letzten Jahrzehnten mit der Erkennung von Code-Smells in Objekt-orientierten Software-Systemen beschäftigt, worauf in dem folgenden Abschnitt eingegangen wird.

2.2.1 Erkennung von Code-Smells

Die automatische Erkennung von Code-Smells ist ein wichtiges Forschungsgebiet. Ob ein Code-Stück Smell-behaftet ist oder nicht, kann nur von einem menschlichen Entwickler definitiv entschieden werden. Travassos et al. haben beispielsweise einen Prozess vorgeschlagen, der auf manueller Inspektion und Lesetechniken basiert [TSFB99]. Die Lesetechniken geben spezifische und praktische Instruktionen zur Erkennung eines Smells. Jedoch ist die manuelle Suche sehr zeitaufwendig und für die Analyse eines kompletten Software-Systems daher nicht praktikabel. Darum werden weitere Wege zur automatischen Detektion von potentiell Smell-behafteten Codeteilen gesucht.

```
1  void vo_draw_alpha_rgb24 ( int w, int h, unsigned char * src ,
       unsigned char * srca , int srcstride , unsigned char * dstbase
       , int dststride ){
2   if( gCpuCaps . hasMMX2 )
3   vo_draw_alpha_rgb24_MMX2 (w, h, src , srca , srcstride , dstbase
       , dststride );
4  else if( gCpuCaps . has3DNow )
5   vo_draw_alpha_rgb24_3DNow (w, h, src , srca , srcstride , dstbase
       , dststride );
6   else if( gCpuCaps . hasMMX )
7   vo_draw_alpha_rgb24_MMX (w, h, src , srca , srcstride , dstbase ,
       dststride );
8   else
9   vo_draw_alpha_rgb24_X86 (w, h, src , srca , srcstride , dstbase ,
       dststride );
10  }
11
12  void vo_draw_alpha_rgb32 ( int w, int h, unsigned char * src ,
       unsigned char * srca , int srcstride , unsigned char * dstbase
       , int dststride ){
13  if( gCpuCaps . hasMMX2 )
14  vo_draw_alpha_rgb32_MMX2 (w, h, src , srca , srcstride , dstbase
       , dststride );
15  else if( gCpuCaps . has3DNow )
16  vo_draw_alpha_rgb32_3DNow (w, h, src , srca , srcstride , dstbase
       , dststride );
17  else if( gCpuCaps . hasMMX )
18  vo_draw_alpha_rgb32_MMX (w, h, src , srca , srcstride , dstbase ,
       dststride );
19  else
20  vo_draw_alpha_rgb32_X86 (w, h, src , srca , srcstride , dstbase ,
       dststride );
21  }
```

Quelltext 2.5: Beispiel des Code-Smells Duplicated Code. Auszug aus osd.c aus MPlayer.

Eine Methode, die den manuellen Ansatz erweitert, wurde von Van Emden und Moonen entwickelt. Sie zerlegen Smells in einzelne Smell-Aspekte, die in den Packages, Klassen, Methoden und Ähnlichem zu finden sind. Werden alle Aspekte eines Smells gefunden gilt er als erkannt [VEM02]. Mit ihrem Ansatz ist es möglich den Smell-behafteten Code automatisiert zu erkennen und zu visualisieren. Damit werden vorhandene Methoden zur automatischen Code-Inspektion, die die Qualität eines Codes beurteilen sollen, um einen strukturellen Aspekt erweitert. Der Entwickler erhält so einen visuellen Hinweis auf den Ort von potentiellen Smells im Code und kann diese gezielt beurteilen ohne das komplette System Zeile für Zeile zu überprüfen.

Fortschrittlichere Techniken um Smells automatisch zu entdecken basieren auf Metriken oder Heuristiken, die entscheiden sollen ob ein Code Smell-behaftet ist oder nicht. Moha et al. haben, aufbauend auf mehreren Arbeiten zum Thema Code-Smells, ein Verfahren zur Spezifikation und Erkennung von Smells entwickelt, das auf Metriken aufbaut [MGDLM10]. Dieses Verfahren, DECOR genannt, benutzt Re-

geln um Code-Smells zu beschreiben. Diese Regeln bestehen aus Wörtern, die oft in Beschreibungen der Smells zu finden sind. Diese Wörter werden dann mit Operatoren aus DECOR kombiniert. So kann der Smell Lange Methode beispielsweise wie folgt beschrieben werden: `RULE:LongMethod {METRIC LOC_METHOD VERY HIGH 10.0};`. Diese Regel beschreibt, dass eine Lange Methode durch eine hohe Anzahl von Codezeilen definiert ist. Komplexere Code-Smells können durch Kombination von verschiedenen Regeln beschrieben werden. Diese Regeln benutzt DECOR dann um ein Nachweisverfahren zu generieren, welches nach der Ausführung eine Liste von potentiell Smell-behafteten Codeteilen ausgibt.

Eine auf Heuristiken basierende Technik, die ein Bayesian Belief Network (BBN) [Pea14] verwendet, schlagen Khom et al. vor [KVGS09]. Sie betrachten die Code-Smell-Suche als ein Klassifikations-Problem, bei dem die Entscheidung gefällt wird, ob ein Codefragment smellbehaftet ist oder nicht. Die Entscheidung wird anhand mehrerer Vektoren, die diese beiden Klassen beschreiben, gefällt. In ihrer Arbeit nutzen sie ein BBN um Modelle von Smells zu entwickeln, die jeder Klasse eine Wahrscheinlichkeit zuordnen. Das Ergebnis des BBN ist die Smellwahrscheinlichkeit p(smell=true) für jede Klasse des Software-Systems. Diese Wahrscheinlichkeit p kann anschließend benutzt werden, um potentiell smellbehaftete Code-Fragmente zu identifizieren.

2.3 Statistische Tests

Im Evaluierungsteil dieser Arbeit werden verschiedene statistische Tests angewandt um beispielsweise Beziehungen zwischen den verschiedenen Variability-aware Code-Smells zu zeigen. Dafür werden in den folgenden Abschnitten diese Tests erklärt. Zuerst werden die *Spearman-Korrelation* und die *Cohen d Effektstärke* beschrieben. Abschließend wird der, für die Beantwortung unserer Forschungsfragen essentielle, *exakte Test nach Fisher* und die damit zusammenhängenden *Odds Ratios* ausführlich erklärt.

Spearman-Korrelation
Der Spearman-Korrelation Koeffizient ist ein parameterfreier Maßstab über die Stärke und Richtung eines Zusammenhangs zwischen zwei Variablen. Der Test kann für ordinale oder stetige Variablen eingesetzt werden. Beispielsweise kann die Spearman-Korrelation dazu benutzt werden, um den Zusammenhang zwischen dem Alter einer Person und ihrer Performance beim 100-Meter-Lauf festzustellen. Dazu messen wir für sechs Personen das Alter in Jahren und die Zeit für 100 Meter in Sekunden und teilen diesen Werten ihre jeweiligen Ränge zu (siehe Tabelle 2.1).

Mit den Werten aus Tabelle 2.1 kann man nun mit der Formel 2.1 die Spearman-Korrelation berechnen. Das mögliche Ergebnis reicht von -1 (negative Korrelation) über 0 (keine Korrelation) bis 1 (positive Korrelation). In dem gegebenen Beispiel ist der berechnete Koeffizient $r_{\mathrm{Sp}} = 0.83$, was bedeutet, dass ein relativ starker Zusammenhang zwischen dem Alter einer Person und seiner Zeit auf 100 Metern besteht.

$$r_{\mathrm{Sp}} = 1 - \frac{6 \cdot \sum_{i=1}^{n} d_i^2}{n \cdot (n^2 - 1)} \tag{2.1}$$

Person i	Alter x_i	Rang des Alters rang(x_i)	Zeit in Sekunden y_i	Platzierung rang(y_i)	Rangdifferenz rang(x_i)-rang(y_i) d_i
A	59	6	14.61	6	0
B	35	3	11.80	2	1
C	43	5	14.34	5	0
D	23	1	13.03	3	-2
E	42	4	14.18	4	0
F	27	2	11.02	1	1

Tabelle 2.1: Beispieldaten für die Spearman-Korrelation

Cohen d Effektstärke
Eine Effektstärke gibt die Stärke eines statistischen Effekts an. Die Cohen d Effektstärke wird genutzt um Mittelwertunterschiede zweier Gruppen zu beschreiben. Sie ist der Unterschied der Mittelwerte M_1 und M_2 geteilt durch die Standardabweichung $\sigma = \sqrt{(\sigma_1^2 + \sigma_2^2)/2}$ beider Gruppen: $d = (M_1 - M_2)/\sigma$. Die Effektstärke ist klein für $0.2 \leq d < 0.5$, mittel für $0.5 \leq d < 0.8$, und groß für $d \geq 0.8$.

Exakter Test nach Fisher
Die Forschungsfragen dieser Arbeit sollen mithilfe von *Nullhypothesen* beantwortet, beziehungsweise bewertet, werden. Dazu ist es nötig die Grundlagen von statistischen *Signifikanztests* zu klären. Für die Bewertung der Signifikanz wird *Fischers Exakter Test* benutzt. Dieser testet auf die Unabhängigkeit von zwei *nominalen Variablen*. Nominale Variablen klassifizieren Beobachtungen in diskreten Kategorien, dazu gehören beispielsweise das Geschlecht (*männlich* und *weiblich*) oder Schulnoten (*sehr gut* bis *ungenügend*). Solche Datensätze sind oft in sogenannten *Kontingenztabellen* angeordnet. Diese beinhalten die absoluten oder relativen Häufigkeiten von Merkmalsausprägungen der nominalen Variablen. Ein Beispiel einer Kontingenztabelle ist in Tabelle 2.2 auf der nächsten Seite dargestellt. Die gewöhnlichste, und in dieser Arbeit einzige genutzte Variante des Fisher Tests, ist der Test mit 2x2-Tabellen. Er hat gegenüber anderen ähnlichen Signifikanztests, wie dem Chi-Quadrat-Test, den Vorteil keine Anforderungen an die Stichprobengröße zu stellen. Er liefert selbst bei geringer Anzahl von Beobachtungen zuverlässige Ergebnisse [HEK09].

Der Test wird genutzt um Daten, die in zwei verschiedenen Arten klassifiziert wurden, auf signifikante Korrelation zu überprüfen. In Tabelle 2.2 auf der nächsten Seite sind beispielsweise Menschen einmal nach Geschlecht und einmal nach ihrer Leistung in der Schule kategorisiert. Der Ziel des Tests ist es nun zu zeigen, dass die Unabhängigkeit bzw. Abhängigkeit der Leistung vom Geschlecht, statistisch signifikant ist.

Dazu wird zunächst eine Nullhypothese aufgestellt. In diesem Falle lautet diese: Das Verhältnis von guten zu schlechten Leistungen ist bei männlichen und weiblichen Schülern gleich. Der Signifikanztest kann diese nun bestätigen oder verwerfen. Sollte die Nullhypothese verworfen werden, besteht ein signifikanter Zusammenhang zwischen Geschlecht und Leistung eines Schülers. Diese Tests können gerichtet oder ungerichtet sein. Das bedeutet bei ungerichteten (zweiseitigen) Tests wird nur auf die Unabhängigkeit getestet. Bei gerichteten (einseitigen) Test fließt in den Test

	männlich	weiblich	
genügend	a = 5	b = 0	5
ungenügend	c = 1	d = 4	5
	6	4	n = 10

Tabelle 2.2: Beispiel: Kontingenztabelle, Menschen klassifiziert nach Geschlecht und Schulleistung

	m	w		m	w		m	w		m	w	
genügend	1	4	5	2	3	5	3	2	5	4	1	5
ungenügend	5	0	5	4	1	5	3	2	5	2	3	5
	6	4	10	6	4	10	6	4	10	6	4	10

Tabelle 2.3: übrige Kontigenztabellen des Beispiels

schon die Vermutung, wie diese Unabhängigkeit aussieht, mit ein. Beispielsweise wenn man nicht testen möchte ob die Leistung überhaupt abhängig vom Geschlecht ist, sondern bereits die Vermutung hat, dass ein bestimmtes Geschlecht besser ist. Dann kann gezielt auf dieses Geschlecht geprüft werden.

In dem gewählten Beispiel wird davon ausgegangen, dass die Randbedingungen festgelegt sind. Das bedeutet: bekannt ist, dass 6 von 10 Jugendlichen männlich sind und 5 der 10 genügende Schulleistung vollbringen. Mit den festgelegten Randbedingungen gibt es somit fünf mögliche verschiedene Kontingenztabellen. Diese sind zum einen das Testergebnis aus dem Beispiel in Tabelle 2.2, zum anderen die vier in Tabelle 2.3 abgebildeten Kontingenztabellen. Diese Tabellen entstehen durch Beachtung der feststehenden Randbedingungen und Verändern der anderen Werte. Diese fünf Verteilungen wären somit theoretisch möglich gewesen.

Mit Formel 2.2 wird nun die Wahrscheinlichkeit berechnet, mit der jede dieser Verteilungen auftreten kann.

$$p = \frac{\binom{a+b}{a}\binom{c+d}{c}}{\binom{n}{a+c}} = \frac{(a+b)!\,(c+d)!\,(a+c)!\,(b+d)!}{n!a!b!c!d!} \tag{2.2}$$

Für das gegebene Beispiel ergibt sich somit der p_{cutoff} Wert von

$$p_{\text{cutoff}} = \frac{(5)!\,(5)!\,(6)!\,(4)!}{10!5!0!1!4!} = 0.0238 \tag{2.3}$$

und die p-Werte der anderen möglichen Matrizen:

$$\begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix} p = 0.2381 \tag{2.4}$$

$$\begin{bmatrix} 3 & 2 \\ 3 & 2 \end{bmatrix} p = 0.4762 \tag{2.5}$$

$$\begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} p = 0.2381 \tag{2.6}$$

$$\begin{bmatrix} 1 & 4 \\ 5 & 0 \end{bmatrix} p = 0.0238 \tag{2.7}$$

Diese p-Werte aus Formel 2.3 bis Formel 2.7 summieren sich zu 1, wie für Wahrscheinlichkeiten üblich. Nun gibt es die zwei Möglichkeiten der ungerichteten bzw. gerichteten Untersuchung. Für die ungerichtete Untersuchung, also die Untersuchung ob überhaupt ein Zusammenhang zwischen den Daten besteht, werden alle p-Werte, die kleiner oder gleich dem p_{cutoff} Wert sind, summiert. Wenn die Summe dieser Werte (in diesem Fall 0.0476) kleiner als ein vorher festgelegtes Signifikanzniveau ist, beispielsweise 5% bzw. 0.05, kann die Nullhypothese verworfen werden. In dem fiktiven Beispiel aus Tabelle 2.2 mit $a = 5$ kann die Nullhypothese also verworfen werden, das heißt es besteht ein signifikanter Zusammenhang zwischen Geschlecht und Schulleistung. Für die gerichtete Untersuchung, beispielsweise wenn man davon ausgeht, dass männliche Studenten bessere Leistungen vollbringen, werden alle p-Werte summiert, deren Kontingenztabellen noch extremere Werte aufweisen. In diesem Fall gibt es keine extremeren Werte als die der Ausgangstabelle. Der endgültige p-Wert ist somit 0.0238 und auch dieser liegt unter dem Signifikanzniveau. Somit kann abermals die Nullhypothese verworfen werden. Das heißt es besteht ein signifikanter Zusammenhang zwischen dem Geschlecht und der Schulleistung, mit besonderem Fokus darauf, dass männliche Schüler bessere Leistungen erbringen.

Als zusätzlichen Indikator berechnen wir des Weiteren die *Odds Ratio* (OR). Mit ihr kann bei ungerichteten Tests gezeigt werden, wie der Zusammenhang aussieht und bei gerichteten Tests, wie stark dieser Zusammenhang ist. Sie gibt die Wahrscheinlichkeit an, dass ein bestimmtes Ereignis eintritt, beispielsweise gute Leistungen in der Schule. Die OR ist definiert als das Verhältnis der Chance p, dass das Ereignis in einem Sample auftritt, zu der Chance q, dass es in einem anderen Sample auftritt [KDPGA12]. Diese Samples könnten beispielsweise „Person ist männlich“ (Experimentalgruppe) und „Person ist weiblich“ (Kontrollgruppe) sein.

$$Odds = \frac{p/(1-p)}{q/(1-q)} = \frac{(a/c)}{(b/d)} \tag{2.8}$$

Eine Odds Ratio von genau 1 bedeutet, dass das Ereignis in beiden Gruppen gleich wahrscheinlich ist. Eine OR größer als 1 bedeutet, dass das Ereignis in der Experimentalgruppe wahrscheinlicher ist, wobei es bei einem OR kleiner als 1 in der Kontrollgruppe wahrscheinlicher auftritt. Je größer der Wert, desto größer ist die Wahrscheinlichkeit, dass das Ereignis in der Experimentalgruppe auftritt. Große ORs bedeuten somit bessere Ergebnisse für die in dieser Arbeit folgenden Untersuchungen.

3. Konzept

Im vorangegangenen Kapitel wurden das Konzept der hochkonfigurierbaren Software-Systeme eingeführt und die Grundlagen objektorientierter Code-Smells vorgestellt. Objektorientierte Smells können die Variabilität hochkonfigurierbarer Systeme nicht abbilden, da sie unter anderem nicht auf Besonderheiten der Implementationstechniken eingehen. Daher wurden für diese Systeme die Variability-aware Code-Smells entworfen [FS15]. In diesem Kapitel wird zuerst das Konzept der Variability-aware Code-Smells eingeführt. Anschließend werden die Forschungsfragen und das Konzept der durchgeführten Fallstudie zur Untersuchung des Einflusses von Code-Smells auf Bereiche der Software-Entwicklung präsentiert. Abschließend wird die Umsetzung des Konzepts inklusive der genutzten externen Programme präsentiert.

3.1 Variability-aware Code-Smells

Der in Abschnitt 2.2 besprochene Begriff der Code-Smells bezieht sich nur auf die objektorientierte Programmierung und lässt Variabilität in Software komplett unberücksichtigt. Sowohl in der Forschung als auch in der Industrie kommen diese hochkonfigurierbaren Software-Systeme immer häufiger zum Einsatz. Durch die Variabilität im Code entstehen auch neue Möglichkeiten für schlechte Designentscheidungen. Da hochkonfigurierbare Softwaresysteme auf Langlebigkeit ausgelegt sind, kann das Risiko für Code-Verfall auch wachsen. Die Code-Smells von Fowler et al. sind jedoch nur auf *Single Software Products (SSPs)* ausgelegt und können dieses Risiko nicht auf *Software Produktlinien (SPLs)* abbilden. Um dieses Problem zu beheben haben Fenske und Schulze begonnen das Konzept der Code-Smells in Bezug auf Variabilität zu erweitern [FS15]. Sie nennen diese neue Art Smells *Variability-aware Code-Smells*. Nicht jeder der vorgestellten Variability-aware Code-Smells kann auf jede der im Grundlagenkapitel vorgestellten Implementationstechniken angewandt werden. Der Fokus der vorliegenden Masterarbeit liegt auf den Smells, die in Annotations-basierten Systemen zu finden sind. Durch die Nutzung der Detektionssoftware SKUNK sind die Untersuchungen auf C-Präprozessor-basierte Systeme beschränkt.

Hochkonfigurierbare Softwaresysteme werden erstellt, indem bestimmte Stellen im Quellcode variabel gestaltet werden. In C-Systemen werden dafür `#ifdef`-Anweisungen, die den variablen Code umschließen, benutzt. Zum einen ist durch die Variabilität eine einfache Möglichkeit gegeben, mehrere Softwaresysteme mit nur einer gemeinsamen Code-Basis zu entwickeln [ABKS13]. Zum anderen ist es jedoch nötig für jede variable Code-Stelle zwei zusätzliche Code-Zeilen einzufügen (`#ifdef` und `#endif`). Einige wenige `#ifdef`-Blöcke verringern noch nicht direkt die Verständlichkeit des Quellcodes. In realen Systemen aus der Industrie können jedoch tausende dieser, unter Umständen zusätzlich verschachtelter, Blöcke innerhalb des Codes auftreten. So kann es problematisch werden, denn während die Technik den Entwicklern erlaubt flexibel mit der Variabilität zu arbeiten, greift sie auch direkt in den normalen Quellcode ein und wird mit ihm vermischt. CPP-Direktiven, die falsch (beziehungsweise schlecht) eingesetzt werden, können beispielsweise die Programmverständlichkeit verringern [MKR+15].

Um die verschiedenen Arten von Smells beschreiben zu können, wird zunächst ein Beispiel präsentiert. In dem Beispiel aus Quelltext 3.1 auf der nächsten Seite wird eine simple Berechnung mit sechs verschiedenen Features (`USE_A`, `USE_B`, `USE_C`, `DEFAULT`, `MULTIPLY`, `ADDITION`) gezeigt. Abhängig von der Konfiguration, addiert oder multipliziert diese Funktion eine Eingabe mit entweder einem Defaultwert oder einer weiteren Eingabe. Diese Funktion kann in mindestens vier einzelne Funktionen (bspw. `add`, `add_def`, `mul`, `mul_def`) aufgeteilt werden. Da jedoch nicht jede Funktionalität in jedem Produkt genutzt wird, haben sich die Entwickler dafür entschieden sie zu vereinen und mit variablen Features auszustatten. Man sieht, dass die Anzahl der CPP-Direktiven die Anzahl der C-Statements übersteigt. Von den 32 Codezeilen werden 18 für `#ifdef`-Anweisungen und lediglich 14 für Funktionalität benutzt. Von diesen 14 Zeilen C-Code sind wiederum nur fünf statisch – der Rest variabel. Jede mögliche Konfiguration benutzt nur bestimmte C-Anweisungen. Beispielsweise wenn das Feature `USE_A` aktiv ist, werden nur sieben Zeilen Code (die Zeilen 1, 5, 9, 13, 16, 31 und 32) benutzt. Zusätzlich beinhaltet die Funktion verschachtelte `#ifdef`-Blöcke zwischen Zeile 18 und 29 und eine Negation und Kombination von Features in Zeile 2. So kommt es dazu, dass das Feature `USE_C` Zeile 3 aktiviert, die Zeilen 22 und 28 jedoch unaktiviert bleiben, wenn die Features `MULTIPLY` oder `ADDITION` nicht angewählt sind. Insgesamt können aus diesem Codebeispiel 64 Varianten gebildet werden, da sich die Features nicht gegenseitig bedingen.

Entwickler müssen viele verschiedene Konfigurationen beachten, wenn sie Funktionen ändern oder erweitern wollen. Zusätzlich muss sichergestellt werden, dass die Funktion in allen möglichen Konfigurationen syntaktisch und semantisch korrekt ist. Daher erschweren viele `#ifdef`-Blöcke das schnelle Verstehen einer Funktion. Außerdem sind so Zusammenhänge verschiedener variabler Teile in einem Softwaresystem schwieriger erkennbar. Fenske und Schulze haben daher einen hohen Anteil von CPP-Direktiven und der daraus resultierenden schlechten Lesbarkeit des Codes als typisches Merkmal eines Variability-aware Code-Smells festgelegt [FS15].

In den Arbeiten von Fenske und Schulze [FS15] und Daniel Meyer [Mey15] werden drei Code-Smells vorgestellt, die auf C-Präprozessor basierten Systemen anwendbar und daher für die Untersuchungen der vorliegenden Arbeit interessant sind. In den

```
1  int calculate ( int a, int b
2  #if defined ( USE_C ) && ! defined ( DEFAULT )
3  int c
4  # endif
5  ) {
6  #if defined ( DEFAULT )
7  int c = 30;
8  # endif
9  int result =
10 #if defined ( DEFAULT )
11 5;
12 # else
13 0
14 # endif
15 #if defined ( USE_A )
16 result += a;
17 # endif
18 #if defined ( MULTIPLY )
19 #if defined ( USE_B )
20 result *= c + b
21 # elif defined ( USE_C )
22 result *= c * c
23 # endif
24 # elif defined ( ADDITION )
25 #if defined ( USE_B )
26 result *= c + b
27 # elif defined ( USE_C )
28 result *= c * c
29 # endif
30 # endif
31 return result ;
32 }
```

Quelltext 3.1: Potentiell Smell-behafteter annotierter Code mit sechs verschiedenen Features

folgenden Abschnitten werden die drei Smells Annotation Bundle, Annotation File und Large Feature näher beschrieben und erklärt.

3.1.1 Annotation Bundle und Annotation File

Die Variability-aware Code-Smells Annotation Bundle und Annotation File beschreiben die gleiche Art von fehlerhaftem Design, jedoch auf unterschiedlichen Granularitätsebenen. Sie sind abgeleitet von dem OOP-Smell Long Method [Fow99] und auf Annotations-basierte Produktlinien beschränkt. Das Annotation Bundle beschreibt eine Funktion, die überwiegend aus variablen bzw. annotierten Code-Teilen besteht. Die hohe Anzahl an Annotationen (z.B. `#ifdef`-Blöcken) und die möglichen Verschachtelungen dieser sind dafür verantwortlich, dass Funktionen viele verschiedene Features enthalten und es schwierig ist schnell und einfach zu erkennen, welche Teile des Codes bei aktueller Konfiguration genutzt werden.

Das Code-Beispiel in Quelltext 3.2 auf der nächsten Seite zeigt einen Auszug aus der MySQL-Version 5.6.17 und weist Ähnlichkeiten zu dem einführenden Beispiel

```
1  sig_handler process_alarm(int sig __attribute__((unused))) {
2      sigset_t old_mask;
3      if (thd_lib_detected == THD_LIB_LT &&
4          !pthread_equal(pthread_self(),alarm_thread)) {
5  #if defined(MAIN) && !defined(__bsdi__)
6      printf("thread_alarm in process_alarm\n");
7      fflush(stdout);
8  #endif
9  #ifdef SIGNAL_HANDLER_RESET_ON_DELIVERY
10     my_sigset(thr_client_alarm, process_alarm);
11 #endif
12     return;
13     }
14 #ifndef USE_ALARM_THREAD
15     pthread_sigmask(SIG_SETMASK,&full_signal_set,&old_mask);
16     mysql_mutex_lock(&LOCK_alarm);
17 #endif
18     process_alarm_part2(sig);
19 #ifndef USE_ALARM_THREAD
20 #if !defined(USE_ONE_SIGNAL_HAND) && defined(
21     SIGNAL_HANDLER_RESET_ON_DELIVERY)
22     my_sigset(THR_SERVER_ALARM,process_alarm);
23 #endif
24     mysql_mutex_unlock(&LOCK_alarm);
25     pthread_sigmask(SIG_SETMASK,&old_mask,NULL);
26 #endif
27     return;
28 }
```

Quelltext 3.2: Beispiel einer Funktion die mit dem Annotation Bundle Code-Smell behaftet ist. Auszug aus einer MySQL-Funktion [FS15].

aus Quelltext 3.1 auf der vorherigen Seite auf. Es zeigt ein typisches Beispiel eines Annotation Bundle. Das gegebene Beispiel hat 29 Zeilen Code, jedoch sind davon nur 19 Zeilen C-Anweisungen. Die restlichen zehn Zeilen sind `#ifdef`-Direktiven. Innerhalb dieser Blöcke befinden sich neun Zeilen Code, woraus sich ergibt, dass nur zehn Zeilen zum Basis-Code gehören und nicht variabel sind. Somit ist lediglich die Hälfte des Codes statisch, während die andere Hälfte von Features abhängig ist. Genauer enthält die Methode fünf #ifdef-Blöcke (Zeilen 6-9, 10-12, 15-19, 21-27 und 22-24) mit insgesamt fünf verschiedenen Features, wovon drei negiert (Zeilen 6, 15 und 22) sind. Zusätzlich gibt es eine verschachtelte CPP-Direktive (Zeilen 22-24) und zwei Annotationen mit Kombinationen von verschiedenen Features (Zeile 20). Fenske et al. behaupten, dass solch eine Funktion, die mit einer großen Menge an Features und deren Implementation vermischt ist, schwer zu verstehen ist [FSMS15]. Weiterhin erschwert eine hohe Menge an variablem Code den Blick auf die Hauptfunktionalität eines Software-Systems.

Um die vorgestellten Charakteristika automatisch überprüfen zu können, werden aus den Software-Systemen Metriken extrahiert. Diese Metriken, wie beispielsweise die Anzahl der Feature Kontstanten (NOFC), werden genutzt um die Code-Smells formal zu beschreiben. Folgend werden die Smells Annotation Bundle

und Annotation File anhand von Metriken erklärt. Im Falle des Annotation Bundle haben Fenske et al. drei Charakteristika definiert [FSMS15]. Zuerst, ein stark annotierter Funktionskörper bedeutet, dass die Anzahl der annotierten Codezeilen (LOAC) einen großen Teil der Codezeilen (LOC) der Funktion einnehmen. Wenn diese Menge an annotiertem Code durch mehrere Feature Locations (NOFL), das heißt mehrere `#ifdef`-Anweisungen, eingeführt werden, wird der Code noch undurchsichtiger. Weiterhin, vermuten Fenske et al., dass eine geringe Anzahl an Features in einer Funktion nicht so verwirrend, wie eine Funktion mit einer hohen Anzahl an Features verteilt über mehrere Feature Locations, ist [FSMS15]. Das bedeutet, je mehr Features in einer Funktion eingeführt werden umso unverständlicher wird der Code. Letztlich muss eine Funktion mit einer großen Menge an LOAC und NOFC nicht zwingenderweise schlecht programmiert sein. Beispielsweise kann eine Initialisierungs-Funktion viele Settings per Feature initialisieren. Die Features in solch einer Datei sind meist voneinander unabhängig und dienen nur der Einführung beziehungsweise Initialisierung von, beispielsweise, globalen Variablen. Diese Funktion kann unter Umständen einen sehr großen Anteil an Feature Code haben, da sie nichts anderes macht als Settings zu initialisieren, und kann trotzdem einfach zu verstehen sein. Eine Funktion mit vielen verschachtelten `#ifdef`-Blöcken wiederum ist für den Entwickler schwieriger zu lesen und zu verstehen. Dementsprechend resultiert eine größere Nesting-Summe (NS), das heißt die Summe aller geschachtelten `#ifdef`-Blöcke, in verwirrenderem Code. Aus diesen Charakteristika wurde die folgende Formel 3.1 abgeleitet (ω 1 bis 3 geben die Gewichtung an) [FSMS15].

$$AB_{\text{Smell}} = \omega_1 * \frac{LOAC}{LOC} * NOFL + \omega_2 * \frac{NOFC_{\text{dup}}}{NOFL} + \omega_3 * \frac{NS}{NOFL} \tag{3.1}$$

Der Smell Annotation File ist ähnlich dem Annotation Bundle, jedoch auf Datei- statt Funktionsebene bezogen. Das bedeutet, dass nicht nur das Auftreten von Features in Funktionen gezählt werden, sondern jedes Auftreten von Features in einer kompletten Datei. Daher wurde für diesen Smell die folgende Formel 3.2 aufgestellt (ω 1 bis 3 geben die Gewichtung an).

$$AF_{\text{Smell}} = \omega_1 * \frac{LOAC}{LOC} * NOFL + \omega_2 * \frac{NOFC_{\text{dup}}}{NOFL} + \omega_3 * \frac{NS}{NOFL} \tag{3.2}$$

Der Wert der sich aus diesen Formeln ergibt, ist lediglich ein Indikator für die Stärke eines Smells und werden daher auch *Severity-Werte* genannt. Ohne weitere Kriterien für die Auswahl potenzieller Smells würden alle Funktionen beziehungsweise Dateien als Smell-behaftet gelten, da die berechneten Severity-Werte keine Auswahl treffen. Zu diesem Zweck müssen für jeden Code-Smell Schwellwerte und Verhältnisse festgelegt werden. Diese müssen erfüllt werden, so dass die Datei oder Funktion als Smell-behaftet bewertet wird. Ausschlaggebend für die beiden Smells Annotation Bundle und File sind die gleichen Metriken nur einmal auf Funktions- und einmal auf Dateiebene. Folgend werden diese Kriterien beschrieben. Die genauen Schwellwerte können individuell festgelegt werden. Die für unseren Versuch gewählten Werte werden in Abschnitt 4.1 eingeführt.

- Verhältnis von LOAC zu LOC einer Funktion beziehungsweise Datei, da sich die Code-Smells auf den problematischen Anteil von `#ifdef`-Direktiven beziehen. Ein zu kleiner Schwellwert könnte zu vielen fälschlicherweise als Smell-behafteten Stellen führen, wohingegen ein zu großer Schwellwert Smell-behaftete Stellen übersieht.

- Die Anzahl der Feature Konstanten (NOFC) ist wichtig, da wir die Streuung und Verknüpfung von Features als essentielle Charakteristik von Annotation Bundles betrachten. Eine hohe Anzahl an Feature Konstanten bedeutet die Einführung vieler verschiedener Features innerhalb einer Funktion beziehungweise Datei.

- Die Anzahl an Feature Locations (NOFL) wird benutzt um Dateien und Funktionen auf die Anzahl der eingeführten `#ifdef`-Blöcke zu überprüfen. Eine Mindestanzahl ist wichtig da beispielsweise Dateien mit nur einer großen Sektion von annotiertem Code selten als Smell-behaftet erachtet werden.

- Die Nesting Summe wird als Schwellwert festgelegt, da geschachtelte Feature Locations die Komplexität eines Smells sehr erhöhen. Daher kommen nur Funktionen und Dateien mit einer Mindestanzahl an geschachtelten `#ifdef`-Blöcken als potentiell Smell-behaftet in Frage.

Je höher der Wert, der sich aus den Formeln Formel 3.1 und Formel 3.2 ergibt, umso wahrscheinlicher ist es, dass die untersuchte Funktion beziehungsweise Datei Smell-behaftet ist [Mey15]. Daraus ergibt sich die Frage, ob diese Charakteristika und Code-Smells einen Einfluss auf Bereiche der Softwareentwicklung haben. Die Vermutung ist, dass die hohe Variabilität das Verstehen des Codes erschwert. Je höher die Variabilität ist, desto schwieriger wird der für bestimmte Konfigurationen aktive Teil erkannt. Hinzu kommt, dass bei Änderungen im Code jede mögliche Konfiguration geprüft werden muss. Die Wartung und Evolution eines Programmes wird dadurch erschwert. Unsere Untersuchungen überprüfen diese Vermutungen, indem sie die Fehlerhäufigkeit in Smell-behafteten und nicht Smell-behafteten Dateien vergleicht.

3.1.2 Large Feature

Der Variability-aware Code-Smell Large Feature wurde von Daniel Meyer in dessen Masterarbeit [Mey15] eingeführt und ist von dem ursprünglich für OOP-Programme eingeführten Smell Large Class [Fow99] abgeleitet. Meyer beschreibt Features, die zu viel Funktionalität besitzen und somit zu viel Verantwortung innerhalb eines Softwaresystems tragen. Typische Charakteristika großer Features sind zum einen eine hohe Menge an Feature-Code und zum anderen die Einführung von vielen Programmelementen, wie Variablen und Funktionen. Weiterhin ist auch eine starke Streuung des Features Codes (Scattering) eine Charakteristik, die ein Large Feature beschreiben kann. Scattering bedeutet, dass sich das Feature über eine große Anzahl an Files erstreckt.

```
2719
2720 #elif defined ( WIN32 ) // line 2719
2721
2722 int
2723 tun_read_queue ( struct tuntap *tt , int maxsize ) {...}
2724
2725 int
2726 tun_write_queue ( struct tuntap *tt , struct buffer * buf ) {...}
```

```
5309
5310 const char *
5311 ipset2ascii ( int index ) {...}
5312
5313 const char *
5314 ipset2ascii_all ( struct gc_arena *gc) {...}
5315
5316 # else /* generic */ // line 5315
```

Quelltext 3.3: Beispiel einer Funktion die mit dem Large Feature Code-Smell behaftet ist. Auszug aus OpenVPN.

In folgendem Beispiel in Quelltext 3.3 ist ein Auszug aus dem Feature `WIN32` aus dem Programm OpenVPN dargestellt [Mey15]. Es zeigt einen 2569 Zeilen langen Teil des `WIN32` Features aus dem File tun.c. Der Auszug aus Quelltext 3.3 führt bereits über 50 verschiedene Funktionen ein, wobei das File noch drei weitere `WIN32`-Teile beinhaltet. Alles in allem besitzt das Feature 6248 Codezeilen, was einen Anteil von 11,2% der gesamten Codezeilen des Softwaresystems OpenVPN (55687 Zeilen) ausmacht. Weiterhin ist es auf 21 Dateien und insgesamt 173 einzelne `#ifdef`-Blöcken aufgeteilt, was die Komplexität des Features aufzeigt.

Wie bereits in Abschnitt 3.1.1 geschehen, kann auch das Large Feature mit Metriken beschrieben werden. Aus den Charakteristika können zwei Verhältnisse abgeleitet werden. Zum einen, je höher die Anzahl der Feature-Code-Zeilen (LOFC) eines Features verglichen zu den LOC des Systems ist, umso mehr Funktionalität wird von diesem Feature eingeführt. Zum anderen, je öfter das Feature (NOFC) über das komplette Software-System (NOFL) eingesetzt wird, umso mehr Teile des Quellcodes sind von diesem Feature betroffen. Beide zusammen führen zu der folgenden Formel 3.3 (ω_1 und ω_2 geben die Gewichtung an).

$$LF_{\text{Smell}} = \omega_1 * \frac{LOFC}{LOC} + \omega_2 * \frac{NOFC}{NOFL} \tag{3.3}$$

Meyer nimmt in seiner Arbeit an, dass Features, die über viele Files und `#ifdef`-Blöcke verteilt sind, die Softwarewartung erschweren. Je mehr Code und Funktionalität ein Feature beinhaltet und je verstreuter das Feature ist, umso schwieriger wird es das Feature, beziehungsweise den Code, im Ganzen zu verstehen. Schon kleine Änderungen können ungewollte Nebeneffekte an ungeahnten Stellen hervorrufen [Mey15]. Wir wollen in dieser Arbeit untersuchen, inwieweit diese Argumentation stimmt.

3.2 Zielstellung

Bisher gibt es keine wissenschaftlichen Belege dafür, dass Variability-aware Code-Smells tatsächlich zu einer höheren Fehleranfälligkeit führen. Das Ziel der Arbeit ist es daher, den Zusammenhang von Fehleranfälligkeit und Smell-behafteten Dateien zu untersuchen. Da Fehler, beziehungsweise Bugs, im Quellcode nicht automatisch erkannt werden können, gehen wir den Weg über die Fehlerbehebungen. Wir gehen davon aus, dass Dateien in denen eine Fehlerbehebung stattfand vorher fehlerhaft waren und setzen so Fehler mit Fehlerbehebungen gleich. Um den Einfluss von Code-Smells auf die Fehleranfälligkeit festzustellen, werden folgende Forschungsfragen mit den zusammenhängenden Nullhypothesen adressiert:

> *RQ 1: Gibt es einen Zusammenhang zwischen Code-Smells und der Fehleranfälligkeit?*

Es wird untersucht, ob Dateien die mindestens einen Variability-aware Code-Smell besitzen, fehleranfälliger sind als andere. Zu diesem Zweck überprüfen wir die folgende Nullhypothese:

H_{01}: Das Verhältnis von Dateien, die in einem festgelegten Zeitfenster mindestens einmal fehlerbehend geändert wurden, ist gleich, zwischen Dateien die Smell-behaftet sind, und denen, die es nicht sind.

> *RQ 2: Gibt es einen Zusammenhang zwischen bestimmten Arten von Code-Smells und Fehleranfälligkeit?*

Zusätzlich zur Untersuchung, ob Code-Smells generell einen Einfluss haben, wird der Einfluss bestimmter Code-Smells auf die Fehleranfälligkeit untersucht. Somit werden die Variability-aware Code-Smells einzeln betrachtet. Zur Untersuchung dieses Aspekts stellen wir folgende Nullhypothese auf:

H_{02}: Das Verhältnis von Dateien, die in einem festgelegten Zeitfenster mindestens einmal fehlerbehend geändert wurden, ist gleich, zwischen Dateien die von einem bestimmten Smell betroffen sind, und denen die nicht von diesem Smell betroffen sind.

> *RQ 3: Ist der Zusammenhang zwischen Code-Smells und der Fehleranfälligkeit unabhängig von der Dateigröße?*

Diese Frage kommt von den Beobachtungen von El Emam et al., dass Metriken, von denen ein Einfluss auf die Software-Qualität vermutet wird, oft in Beziehung zur Größe stehen [EEBGR01]. Wir untersuchen ob die Fehleranfälligkeit von Smell-behafteten Dateien an der Größe der Datei oder an der Präsenz von Code-Smells liegt, indem wir die folgende Nullhypothese testen:

H_{03}: Das Verhältnis von großen Dateien, die in einem festgelegten Zeitfenster mindestens einmal fehlerbehend geändert wurden, ist gleich, zwischen Dateien die Smell-behaftet sind, und denen die es nicht sind.

Die Nullhypothesen H_{01} bis H_{03} sind einseitig, da wir daran interessiert sind zu untersuchen, ob Variability-aware Code-Smells eine *höhere* Fehleranfälligkeit verursachen. Eine zweiseitige Untersuchung, ob generell ein Einfluss, egal ob negativ oder positiv, besteht, wäre in unserem Fall nicht zielführend.

Es gibt bereits mehrere Arbeiten, die die Fehleranfälligkeit beziehungsweise Qualität von Software-Artefakten aufgrund bestimmter Umstände untersuchen. Beispielsweise nutzen Basili et al. Metriken als Qualitätsindikator [BBM96]. Metriken werden dabei als Anzeichen für höhere Fehleranfälligkeit untersucht. So zeigen Cartwright und Shepperd, dass Klassen mit Vererbungsbeziehungen fehleranfälliger sind [CS00]. Insbesondere die Anzahl der Codezeilen wird dabei oft als Indikator für die Fehleranfälligkeit diskutiert [KZL07, EEBGR01]. Beispielsweise haben El Emam et al. den Zusammenhang von Objekt-orientierten Metriken und der Größe untersucht [EEBGR01]. Sie haben festgestellt, dass viele Metriken mit der Größe korrelieren und somit, unter Einbezug des Störfaktors Größe, der Einfluss der Metriken auf die Fehleranfälligkeit verschwindet. Jedoch sind Forscher sich in diesem Punkt uneinig. So kritisiert Evanco die Arbeit von El Emam et al. und sagt, dass die Fähigkeit die Größe zu messen nicht zeitlich vor der Fähigkeit der Messung vieler Metriken besteht [Eva03]. Somit sei die Bedingung, dass ein Störfaktor (Größe) kausal vor der erklärenden Variable (Metrik) auftreten muss, nicht gegeben. Da der Zusammenhang von Metriken, der Größe von Systemen und dem Auftreten von Fehlern nicht eindeutig ist, ist unsere Untersuchung der Unabhängigkeit von Größe, Smells und der Fehleranfälligkeit in RQ 3 sinnvoll.

Die Resultate können zum einen für Entwickler, die Entwicklungs- und Wartungsaufgaben ausführen und deren Aufwand beachten müssen, interessant sein. Zum anderen kann es wissenswert für Tester, die wissen wollen auf welche Dateien sie sich konzentrieren müssen, sein. Weiterhin können Qualitätsprüfer oder Abteilungsleiter von den Ergebnissen profitieren. Sie könnten beispielsweise mithilfe von Code-Smell-Detektionssoftware die Qualität von eigenem oder zu erstehendem Quellcode überprüfen und so weitere Wartungskosten abschätzen. Letztlich ist es auch für Forscher, die den Zusammenhang zwischen Code-Smells und Software-Entwicklungsmerkmalen untersuchen, interessant. Die vorgestellten Forschungsfragen sind von einer bereits durchgeführten ähnlichen Studie adaptiert. Diese Studie von Khomh et al. beschränkt sich jedoch auf den objektorientierten Software-Bereich und betrachtet die Variabilität nicht [KDPGA12]. Nun sollen diese Fragen für hochkonfigurierbare Sotware-Systeme beantwortet werden.

3.3 Vorgehen

Das folgende Konzept und der in dieser Arbeit durchgeführte Versuch adaptiert eine früher durchgeführte Untersuchung von Khomh et al. [KDPGA12]. Die Annahme von erfahrenen Software-Entwicklern, dass Code-Smells einen negativen Einfluss auf Programmverständnis, -wartung und -evolution haben, wird zwar allgemein akzeptiert [Fow99], jedoch existieren wenig wissenschaftliche Belege, die dies bestätigen [DBDV+06, KPG09]. Daher haben Khomh et al. [KDPGA12] in ihrer Arbeit diesen Einfluss für *objektorientierte Systeme* untersucht, indem sie das Aufkommen von Code-Smells und die Fehleranfälligkeit von Klassen in Verhältnis gesetzt

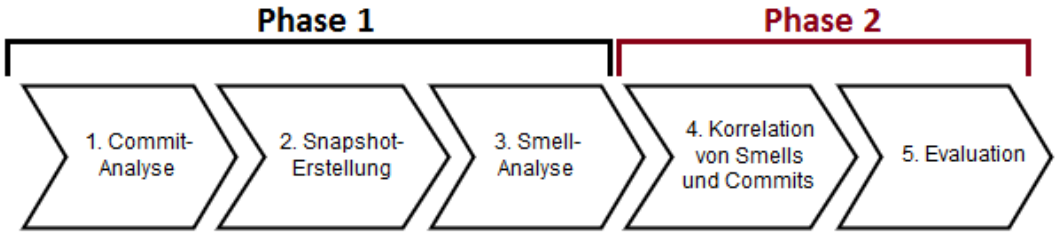


Abbildung 3.1: Vereinfachter Prozessüberblick

haben. Sie analysierten insgesamt 54 Releases von vier verschiedenen Softwaresystemen (u.a. Eclipse). Dabei wurde untersucht, ob Klassen, die in einem Release einen Code-Smell besaßen, bis zum darauf folgenden Release mindestens einmal fehlerbehend geändert worden sind. In jedem Release wurden einmal die Smell-behafteten Klassen bestimmt und anschließend überprüft, ob diese geändert wurden. Um Code-Änderungen in strukturelle, erweiternde und fehlerbehebende Änderungen zu klassifizieren haben Khomh et al. sowohl Commit-Nachrichten als auch Issuetracking-Systeme analysiert. So wurde beispielsweise in Commitnachrichten nach bestimmten Stichwörtern wie „Fix“, „Bug“ oder „Defect“ gesucht. Zusammenfassend zeigt die Studie, dass Smell-behaftete Klassen fehleranfälliger sind als andere. Dieser Zusammenhang konnte selbst dann beobachtet werden, wenn man in Betracht zieht, dass größere Dateien eine höhere Wahrscheinlichkeit haben, fehlerbehebend geändert zu werden, als kleinere Dateien. Unsere Arbeit untersucht diesen Einfluss auf die Softwareentwicklung von *hochkonfigurierbaren Software-Systemen* unter Rücksichtnahme auf Variabilität. Dazu wurde der Ablauf des Versuchs von Khomh et al. mit individuell nötigen Anpassungen übernommen.

Zuerst ist festzustellen, dass in unserer Arbeit auf der Datei-Granularität gearbeitet wird. Das bedeutet, dass eine Datei als Smell-behaftet gilt, sobald es mindestens eine Stelle, die als Smell-behaftet bewertet wurde, im Code dieser Datei gibt. Dabei ist egal wo und wie groß dieser Abschnitt ist. Genauso wird bei den Bugfixes gearbeitet. Das heißt, eine Änderung, die eine Smell-behaftete Datei betrifft, gilt unabhängigd davon, ob wirklich die konkrete Smell-behaftete Stelle geändert wurde, als sogenannter *Smelly-Fix*. Diese vereinfachte Darstellung hat vor allem den Grund, dass für die weitere Auswertung das Verhältnis zwischen Smell-behafteten und nicht Smell-behafteten Abschnitten benötigt wird. Da Smell-behaftete Stellen je nach Smell jedoch nur auf Funktions- oder auch Dateiebene eingegrenzt werden können, ist die Bildung des Verhältnisses von Smell-behafteten Zeilen zu nicht Smell-behafteten Zeilen nicht praktikabel.

Die Abbildung 3.1 gibt einen schematischen Überblick über den Ablauf des Versuchs. Dieser besteht zum einen aus einer Vorverarbeitungs- und Smell-Analyse-Phase in den ersten drei Schritten und zum anderen aus der Verarbeitung der Ergebnisse und der Evaluierung in den beiden letzten Schritten. Diese werden in den folgenden Abschnitten erklärt.

Im ersten Schritt wird die Commit-Historie des jeweiligen Repositories analysiert. Aus der Analyse werden die Hashwerte jedes Commits und die geänderten Dateien, inklusive des relativen Pfads, gewonnen. Zusätzlich wird anhand der Commit-

Nachricht eine Klassifizierung durchgeführt. Dabei werden die Dateiänderungen des jeweiligen Commits in fehlerbehend oder nicht-fehlerbehebend unterschieden. Weitere Daten, die gespeichert werden, sind der Zeitpunkt des Commits und die Zeilenangabe der Änderungen.

Über die Zeit verändern sich Dateien durch die voranschreitende Entwicklung. Durch Änderung an den Dateien können Code-Smells behoben werden oder neue hinzukommen. Da eine Smell-Analyse nach jeder Dateiänderung nicht in einem zeitlich annehmbaren Rahmen durchführbar ist, muss dies in gewissen zeitlichen Abständen passieren. Zu diesem Zweck werden im zweiten Schritt die Hashwerte aus Schritt 1 genutzt um den Entwicklungsstand des Repositories zu bestimmten Zeitpunkten auszuchecken und einen sogenannten *Snapshot* zu erstellen. Auf diesen Snapshots, die aus den .c-Dateien des Repositories zu diesem Commitzeitpunkt bestehen, werden anschließend die Smell-Analysen durchgeführt. Damit wird sichergestellt, dass die Smell- und Fehler-Analyse immer auf relativ aktuellen Daten beruhen.

Sobald alle Snapshots erstellt wurden, folgt in Schritt 3 die Smell-Analyse. Jeder Snapshot eines Repositories wird auf alle vorgestellten Code-Smells untersucht. Dazu werden die in Abschnitt 3.1 eingeführten Metriken verwendet. Die Smell-Analyse speichert die Dateinamen und den relativen Pfad inklusive der berechneten Smellstärke in eine Datei pro Snapshot.

In den beiden abschließenden Schritten werden die aus der Vorverarbeitung und Smell-Analyse gewonnen Daten zusammengefasst und anschließend ausgewertet. Dazu werden die Smell-behafteten Dateien mit den geänderten Dateien in Verbindung gebracht, um die benötigten Daten für die Evaluation bereitzustellen. Die Daten werden für jeden Snapshot zusammengefasst und jeder Snapshot einzeln evaluiert. Zur Beantwortung aller Forschungsfragen werden Nullhypothesen getestet. Um diese Hypothesen verwerfen oder unterstützen zu können, wird der exakte Test nach Fisher benutzt. Für den Fisher-Test werden vier spezielle Werte benötigt. Dies ist das Verhältnis von Dateien, die während eines Snapshots mindestens einmal fehlerbehebend geändert wurden. Dabei wird wiederum zwischen Smell-behafteten und nicht Smell-behafteten Dateien unterschieden. Die vier Werte sind somit folgende:

- Anzahl der Smell-behafteten Dateien, die mindestens einmal fehlerbehebend geändert wurden
- Anzahl der Smell-behafteten Dateien, die nicht fehlerbehebend geändert wurden
- Anzahl der nicht Smell-behafteten Dateien, die mindestens einmal fehlerbehebend geändert wurden
- Anzahl der nicht Smell-behafteten Dateien, die nicht fehlerbehebend geändert wurden

Diese Daten werden jeweils für jeden der drei Smells einzeln erhoben, um den Einfluss von den einzelnen Smells in RQ 2 untersuchen zu können. Zur Beantwortung von RQ 1 und RQ 3 werden die Daten für den Fall, dass ein File mindestens von einem Code-Smell betroffen ist, gespeichert. Mit diesem Schritt sind alle nötigen Daten für

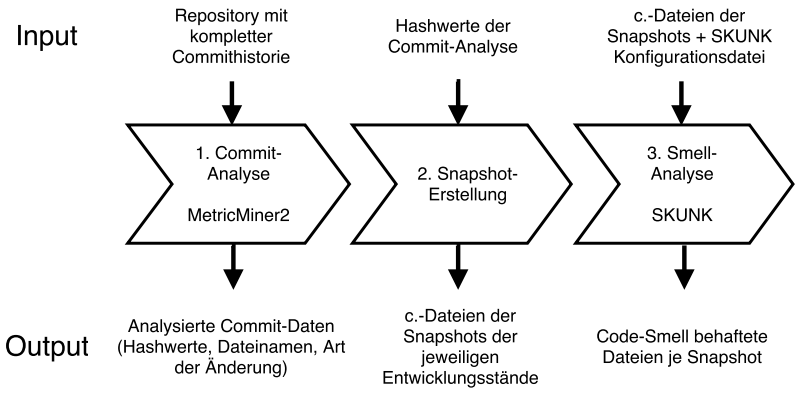


Abbildung 3.2: Schematische Darstellung der Phase der Smell-Analyse

den in der Evaluierung genutzten exakten Test nach Fisher gewonnen. Im folgenden Abschnitt wird die Umsetzung des Konzepts anhand eines Beispiels dargestellt.

3.4 Umsetzung

Das im vorhergehenden Abschnitt vorgestellte Konzept ist aufgeteilt in zwei Phasen. Die ersten drei Schritte aus Abbildung 3.1 gehören zu Phase 1, die mit dem Begriff Smell-Analyse beschrieben werden kann. Dazu wird zuerst die Commit-Historie analysiert, die Snapshots erstellt und abschließend die Smell-Analyse auf allen Snapshots durchgeführt. Die gewonnenen Daten werden jeweils zwischengespeichert, um in Phase 2 verarbeitet werden zu können. Dieser Abschnitt „Smell-Analyse“ ist in Abbildung 3.2 dargestellt. Anschließend werden in Phase 2 diese Daten korreliert und ausgewertet. Die zwei Schritte dieser Phase werden in Abbildung 3.5 schematisch präsentiert. Folgend wird die Umsetzung des Konzepts exemplarisch an dem Software-System OpenVPN vorgestellt.

3.4.1 Phase 1 – Smell-Analyse

Folgend wird die erste Phase des Konzepts vorgestellt. Sie ist in drei Schritte aufgeteilt und wird in Abbildung 3.2 schematisch dargestellt. Die drei Schritte sind eine Commit-Analyse mit MetricMiner2[1] [SFAG+13], die Snapshot-Erstellung und die Smell-Analyse mit SKUNK[2] [Mey15].

Im ersten Schritt wird zunächst der neueste Stand des jeweiligen Repositories genommen. Wichtig ist, dass auf die komplette Commithistorie zurückgegriffen werden kann, da die Suche nach Fehlern von den Commitnachrichten abhängt. Zur Commit-Analyse benutzen wir das von Aniche et al. entwickelte Framework MetricMiner2 [SFAG+13]. Aufgrund der Nutzung von MetricMiner2 zur Commitanalyse und der Verfügbarkeit der Fallbeispiele werden in dieser Arbeit lediglich GitHub[3]-Repositories benutzt. Theoretisch wäre jedoch auch die Benutzung anderer Versionskontrollsysteme möglich, solange sie Zugriff auf die vollständige Commithistorie inklusive der Commitmessages bieten. Anschließend müssen die Commits analysiert werden. Zu diesem Zweck muss für MetricMiner2 eine sogenannte „Study“

[1] http://www.metricminer.org.br/
[2] https://github.com/Dwelgaz/Skunk
[3] https://www.github.com/

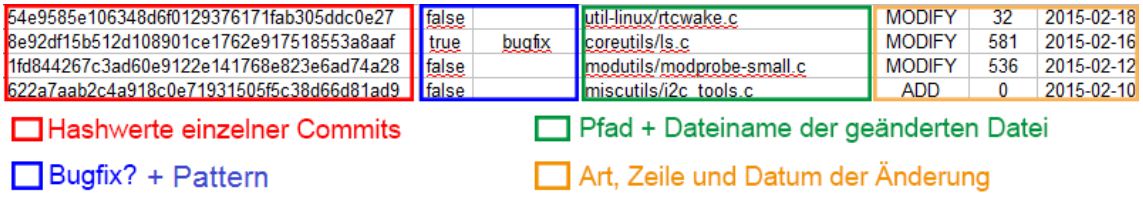
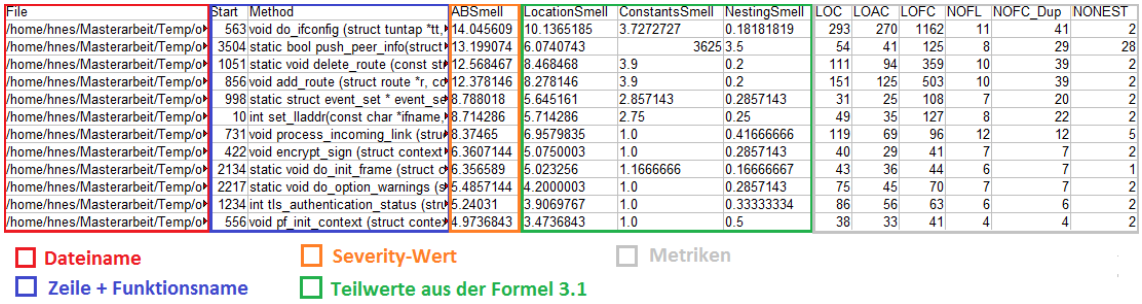


Abbildung 3.3: Auszug aus den gewonnenen Daten der Commit-Analyse

geschrieben werden, in der individuell konfiguriert werden kann, welche Daten aus den Commits extrahiert werden und wie diese gespeichert werden. Wie bereits in Abschnitt 3.3 beschrieben, werden die Hashwerte, die geänderten Dateien und die Entscheidung, ob es sich um einen Bugfix handelt oder nicht, von jedem Commit abgespeichert. Die Klassifikation zwischen Bugfix-Commit und Nicht-Bugfix-Commit wird ebenfalls in der selbtgeschriebenen „Study“ getroffen. Dazu werden die Commitmessages auf die Signalwörter „bug“, „fix“, „patch“ und „error“, wie von Mockus und Votta [MV00] und Pan et al. [PKWJ09] vorgeschlagen, durchsucht. Sollte eines dieser Worte vorkommen, werden der Commit und damit alle geänderten Dateien als fehlerbehebend markiert. Es werden nur Dateien mit der Endung „.c“ berücksichtigt. Dies ist möglich, da sich die untersuchten Code-Smells nur in diesen Dateien befinden können. Für das Beispielsystem OpenVPN mussten 1689 Commits analysiert werden. Davon wurden 574 als Bugfix-Commits klassifiziert. In Abbildung 3.3 ist ein Ausschnitt aus den gewonnenen und in einer extra Datei zwischengespeicherten Daten abgebildet.

In Schritt 2 wird das Repository in gewissen zeitlichen Abständen ausgecheckt und ein Snapshot des Entwicklungsstands zu diesem Zeitpunkt erstellt. In einem Abstand von 100 Bugfix-Commits werden die Snapshots erstellt. Wie bereits im Schritt zuvor werden auch bei der Snapshot-Erstellung nur .c-Dateien berücksichtigt. Dies spart vor allem bei der Code-Smell-Analyse im nächsten Schritt viel Zeit, denn so müssen deutlich weniger Dateien untersucht werden. Bei 574 Bugfix-Commits des Beispielsystems werden somit fünf Snapshots angelegt. Wir haben uns entschieden, nur volle Snapshots mit 100 Bugfix-Commits zu untersuchen, um nicht Gefahr zu laufen, dass zu kleine Snapshots entstehen. Die übrigbleibenden Bugfix-Commits verfallen somit. Es werden alle c.-Dateien des jeweiligen Entwicklungsstands extrahiert und auch der relative Pfad innerhalb des Systems bleibt erhalten.

Der Schritt der Small-Analyse schließt die Phase 1 ab. Für die Smell-Analyse nutzen wir den von Daniel Meyer in seiner Masterarbeit entwickelten Prototypen SKUNK. SKUNK ist ein Tool um Variability-aware Code-Smells in hochkonfigurierbaren, speziell C-Präprozessor-basierten, Software-Systemen zu detektieren. Es basiert auf Metriken und verfolgt einen simplen Ansatz der Detektion. Zuerst werden die Metriken aus dem System extrahiert und berechnet. Anschließend wird anhand von Schwellwerten und Verhältnissen entschieden, ob es potentielle Smell-behaftete Stellen im Quellcode gibt. Metriken und Formeln zur Detektion wurden bereits in Abschnitt 3.1 vorgestellt. Da SKUNK von Haus aus pro Durchgang ein System nur nach einem Code-Smell durchsuchen kann, muss der Schritt für jeden Smell wiederholt werden. Mit dem ersten Durchlauf des für dieses Konzept geschriebenen Programms werden alle Schritte von Phase 1 durchgeführt und die Snapshots erstellt. Wir haben uns



File	Start	Method	ABSmell	LocationSmell	ConstantsSmell	NestingSmell	LOC	LOAC	LOFC	NOFL	NOFC_Dup	NONEST
/home/hnes/Masterarbeit/Temp/o	563	void do_ifconfig (struct tuntap *tt,	14.045609	10.1365185	3.7272727	0.18181819	293	270	1162	11	41	2
/home/hnes/Masterarbeit/Temp/o	3504	static bool push_peer_info(struct	13.199074	6.0740743	3625	3.5	54	41	125	8	29	28
/home/hnes/Masterarbeit/Temp/o	1051	static void delete_route (const st	12.568467	8.468468	3.9	0.2	111	94	359	10	39	2
/home/hnes/Masterarbeit/Temp/o	856	void add_route (struct route *r, co	12.378146	8.278146	3.9	0.2	151	125	503	10	39	2
/home/hnes/Masterarbeit/Temp/o	998	static struct event_set * event_se	8.788018	5.645161	2.857143	0.2857143	31	25	108	7	20	2
/home/hnes/Masterarbeit/Temp/o	10	int set_lladdr(const char *ifname,	8.714286	5.714286	2.75	0.25	49	35	127	8	22	2
/home/hnes/Masterarbeit/Temp/o	731	void process_incoming_link (stru	8.37465	6.9579835	1.0	0.41666666	119	69	96	12	12	5
/home/hnes/Masterarbeit/Temp/o	422	void encrypt_sign (struct context	6.3607144	5.0750003	1.0	0.2857143	40	29	41	7	7	2
/home/hnes/Masterarbeit/Temp/o	2134	static void do_init_frame (struct c	6.356589	5.023256	1.1666666	0.16666667	43	36	44	6	7	1
/home/hnes/Masterarbeit/Temp/o	2217	static void do_option_warnings (s	5.4857144	4.2000003	1.0	0.2857143	75	45	70	7	7	2
/home/hnes/Masterarbeit/Temp/o	1234	int tls_authentication_status (stru	5.24031	3.9069767	1.0	0.33333334	86	56	63	6	6	2
/home/hnes/Masterarbeit/Temp/o	556	void pf_init_context (struct conte	4.9736843	3.4736843	1.0	0.5	38	33	41	4	4	2

Abbildung 3.4: Auszug aus dem Severity-Report des Beispielsystems OpenVPN

bewusst gegen eine Änderung von SKUNK entschieden. Der zeitliche Aufwand um es so zu erweitern, dass mehrere Smells mit einem Durchlauf untersucht werden können stand in keinem Verhältnis zum Nutzen. Unter anderem aus dem Grund, dass SKUNK bereits die Funktion bietet die Vorverarbeitungsschritte zu überspringen. Dies konnten wir ausnutzen indem wir unser Programm so angepasst haben, dass die Schritte 1 und 2 ebenfalls übersprungen werden können. SKUNK erstellt für jeden Snapshot sogenannte *Severity-Reports* und *Location-Files*. In den Severity-Reports wird für jeden potentiellen Smell die Smell-Stärke, die mit den aus Abschnitt 3.1 bekannten Formeln berechnet werden, gespeichert. In Abbildung 3.4 ist ein Auszug aus einem Severity-Report aus OpenVPN für den Code-Smell Annotation Bundle dargestellt. In den Location-Files werden die genauen Zeilen und Dateien, in denen die Smells zu finden sind, angegeben. Dies ist in unserem Fall besonders wichtig für den Smell Large Feature, denn in den Severity Reports der Large Features werden nicht die Dateinamen der potentiellen Smells gespeichert, sondern nur der Name des Smell-behafteten Features. Die somit ebenfalls Smell-behafteten Dateien werden aus den Location Files extrahiert. Für jeden Snapshot werden alle Vorkommen des untersuchten Smells in jeweils eine Datei mit dem Snapshot-Datum als Dateinamen gespeichert. Diese Dateien beinhalten den Dateinamen und dessen relativen Pfad von Smell-behafteten Dateien inklusive deren Severity-Rating. Nach den insgesamt drei Durchläufen (jeweils einmal für Annotation Bundle, Annotation File und Large Feature) sind für jeden Code-Smell und jeden Snapshot alle Smell-behafteten Files inklusive dem Severity-Rating gespeichert.

Mit diesem Schritt ist der Prozess der Vorverarbeitung und Smell-Analyse abgeschlossen. Nach dieser Phase sind folgende Daten für die anschließende Evaluation vorhanden:

- Filenamen und relativer Pfad aller fehlerbehebend geänderter Dateien inkl. Änderungsdatum
- Filenamen und relativer Pfad aller Smell-behafteter Dateien inkl. Art des Smells, Severity Rating und Datum des Snapshots
- Filenamen und relativer Pfad aller .c-Dateien jedes Snapshots

In Phase 2 werden diese Daten weiter verarbeitet und miteinander in Verbindung gebracht, um daraus Antworten auf die Forschungsfragen zu erarbeiten.

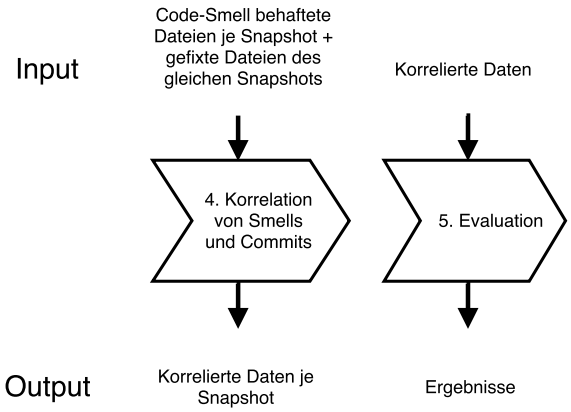


Abbildung 3.5: Schematische Darstellung der Phase der Datenkorrelation und Evaluation

Filename	Annotation Bundle	Annotation File	Large Feature	Smell Count	hasS	hasF	FixCount	File Size
src/openvpn/packet_id.c	0	1	1	2	YES	NO	0	10667
src/openvpn/mtcp.c	0	1	1	2	YES	NO	0	13644
src/openvpn/crypto_polarssl.c	0	1	1	2	YES	YES	2	9180
src/openvpn/tun.c	1	1	1	3	YES	YES	10	96518
src/openvpn/ps.c	0	0	1	1	YES	YES	2	16414
src/openvpn/fdmisc.c	0	0	1	1	YES	NO	0	759
src/openvpn/mstats.c	0	0	1	1	YES	NO	0	1448
src/openvpn/perf.c	0	1	1	2	YES	NO	0	4630
src/openvpn/console.c	0	0	1	1	YES	NO	0	3444

Abbildung 3.6: Auszug aus den korrelierten Daten eines Snapshots aus OpenVPN

3.4.2 Phase 2 – Korrelation der Daten

In der zweiten Phase werden die gewonnenen Daten aus Phase 1 zusammengefasst. Für jeden Snapshot wird eine Datei erstellt in der jede .c-Datei des Snapshots aufgelistet wird. Zusätzlich wird zu jeder Datei gespeichert, von welchem Code-Smell und von wie vielen insgesamt sie betroffen ist. Dazu wird jeder Dateiname mit den gefundenen Dateinamen aus der Smell-Analyse verglichen. Weiterhin wird überprüft ob und wie oft ein File während des Zeitfensters des aktuellen Snapshots fehlerbehebend geändert wurde. Hierzu werden wiederum die Dateinamen mit denen aus der Commitanalyse bekannten Dateien verglichen. Als letztes Merkmal wird die Größe der Datei ermittelt, in dem die Anzahl der Zeichen gemessen wird. In Abbildung 3.6 ist ein Ausschnitt dieser gespeicherten Daten dargestellt.

Um die benötigten Daten für den in der Evaluation benutzten Fisher-Test zu generieren, werden die Daten der einzelnen Snapshots zusammengefasst. Durch die Auswertung der Snapshotdaten aus dem vorherigen Schritt werden folgende Daten gewonnen:

- Anzahl an Smell-behafteten und nicht Smell-behafteten Dateien
- Anzahl der Bugfixes, die in Smell-behafteten bzw. nicht Smell-behafteten Dateien stattfanden
- Durchschnittliche Größe der Dateien, die Smell-behaftet bzw. nicht Smell-behaftet sind

	Smell-behaftet	nicht Smell-behaftet
fehlerbehebend geändert	22	7
nicht fehlerbehebend geändert	17	29

Tabelle 3.1: Beispiel: Kontingenztabelle, Dateien klassifiziert nach Smell-behaftung und fehlerbehebenden Änderungen

Die vier Werte, die für den Fisher-Test benötigt werden (siehe Abschnitt 3.3), werden ebenfalls gespeichert. Diese Daten werden jeweils für jeden der drei Smells einzeln erhoben (für RQ2) und einmal für den Fall, dass ein File mindestens von einem Code-Smell betroffen ist (für RQ1). Außerdem werden die gleichen Daten noch einmal für die 25% größten Dateien (für RQ3) erhoben. Mit diesem Schritt sind alle nötigen Daten für die Evaluationsmethoden gewonnen. Als exemplarisches Beispiel sind in Tabelle 3.1 einmal die vier Werte für einen Snapshot aus OpenVPN abgebildet. Zunächst berechnen wir mit der Formel 3.4 die Odds Ratio, also die Wahrscheinlichkeit, dass eine fehlerbehebende Änderung häufiger in Smell-behafteten Dateien auftritt als in nicht Smell-behafteten. Anschließend wird mit dem exakten Test nach Fisher der p-Wert berechnet. Liegt dieser unter dem gewählten Signifikanzniveau von 95% – also 0.05 – ist das Ergebnis signifikant. Für das gegebene Beispiel ergibt sich eine Odds Ratio von über 5, was darauf schließen lässt, dass die Fehleranfälligkeit in Smell-behafteten Dateien größer ist als in nicht Smell-behafteten Dateien. Weiterhin ergibt sich ein p-Wert von $\approx 0.001 \leq 0.05$. Das Ergebnis ist somit signifikant und die Nullhypothese H_{01} kann für diesen Snapshot verworfen werden. Das bedeutet, dass für diesen Snapshot die Annahme der Nullhypothese, dass Smell-behaftete und nicht Smell-behaftete Dateien gleichermaßen oft fehlerbehebend geändert werden, nicht gilt. Da die Nullhypothese einseitig aufgestellt ist, ist dies ein empirischer Beweis dafür, dass in diesem Snapshot Smell-behaftete Dateien signifikant häufiger Fehler beinhalten als nicht Smell-behaftete Dateien.

$$Odds = \frac{(a/c)}{(b/d)} = \frac{(22/17)}{(7/29)} \approx 5.23 \tag{3.4}$$

3.5 Zusammenfassung

In diesem Kapitel haben wir unsere Forschungsfragen vorgestellt und präzisiert. Um diese Fragen zu beantworten haben wir drei Nullhypothesen aufgestellt, die wir mithilfe des exakten Tests nach Fisher überprüfen. Weiterhin wurde ein allgemeingültiges Konzept vorgestellt, um hochkonfigurierbare Software-Systeme auf den Einfluss von Variability-aware Code-Smells auf die Fehleranfälligkeit von Dateien zu untersuchen. Das Konzept ist in eine Smell-Analyse-Phase und eine Evaluierungs-Phase unterteilt. Anhand eines Beispielsystems haben wir zusätzlich Details über die genutzten Tools beschrieben. In der Umsetzung des Konzepts kommen der Code-Smell-Detektor SKUNK und das Repository-Analysetool MetricMiner2 zum Einsatz. Im nächsten Kapitel präsentieren wir das Setup unserer Untersuchung, wie beispielsweise das gewählte Commit-Fenster und Paramter für die Smell-Detektion, und evaluieren die Untersuchungsergebnisse.

4. Evaluierung

In diesem Kapitel wird eine Fallstudie, die den Einfluss von Variability-aware Code-Smells auf Bereiche der Software-Entwicklung untersucht, vorgestellt. Dabei wird speziell die Fehleranfälligkeit von Dateien in Bezug auf das Vorkommen von Variability-aware Code-Smells untersucht. Zu diesem Zweck wurden in Abschnitt 3.2 drei Forschungsfragen aufgestellt, die in dieser Untersuchung addressiert werden. Die Evaluierung basiert auf dem in Abschnitt 3.3 vorgestellten Konzept und unterscheidet sich im Setup leicht von Forschungsfrage zu Forschungsfrage. Die Untersuchung wird auf sechs Software-Systemen aus unterschiedlichen Domänen durchgeführt. Es wird zunächst das grundlegende Setup und die Methodiken zur Beantwortung der Fragen beschrieben. Zusätzlich wird die Auswahl der sechs Systeme, die in der Fallstudie untersucht werden, diskutiert. Zur Beantwortung der Fragen werden die Spearman-Korrelation, die Cohen d Effektstärke und der exakte Test nach Fisher mit unterschiedlichen Daten berechnet (siehe Abschnitt 2.3). Abschließend werden die Ergebnisse präsentiert und bezüglich der Forschungsfragen diskutiert.

4.1 Allgemeines Setup

Im folgenden Abschnitt wird das grundlegende Setup für die Durchführung des im Kapitel 3 vorgestellten Konzepts präsentiert. Dafür müssen zuerst die Parameter, die SKUNK zur Smell-Analyse nutzt, für jeden Code-Smell festgelegt und erklärt werden. In Abschnitt 3.3 wurde beschrieben, dass die Analyse auf sogenannten Snapshots durchgeführt wird, welche in gewissen zeitlichen Abständen des Entwicklungsstands eines Systems genommen werden. Die Größe dieses sogenannten Commit-Fensters diskutieren wir in einem der folgenden Abschnitte. Zusätzlich werden Auswahlkriterien für die Fallbeispiele beschrieben und die darauf basierende Auswahl vorgestellt.

4.1.1 Parameter für die Smell-Analyse in SKUNK

In SKUNK, einem metrikbasierten Prototypen zur Detektion von Variability-aware Code-Smells, hat der Nutzer die Möglichkeit Schwellwerte und Verhältnisse als Parameter zu übergeben, um die Smell-Erkennung manuell anzupassen und zu optimieren. Für die drei Smells Annotation Bundle (AB), Annotation File (AF)

und Large Feature (LF) gibt es verschiedene Kriterien, die erfüllt werden müssen, um als Smell-behaftet erkannt zu werden. Für unsere Studie nutzen wir die optimierten Parameter, die Meyer in seiner Thesis erarbeitet hat [Mey15]. Die gewählten Schwellwerte wurden von Meyer durch Testläufe mit verschiedenen Systemen optimiert. Sie sind ein Kompromiss zwischen zu niedrigen Schwellwerten, die möglicherweise zu viel false-positives hervorrufen und zu hohen Schwellwerten, bei denen zu viele potentielle Smells ausgelassen werden könnten. Jeder der folgenden Parameter ist verpflichtend, was bedeutet, dass SKUNK nur Codeelemente (Features, Funktionen und Dateien), die alle diese Bedingungen erfüllen, in die Analyse einbezieht.

Annotation Bundle
Die folgenden Parameter für Annotation Bundle werden alle auf Funktions-Granularität überprüft.

- Verhältnis von LOAC zu LOC einer Funktion = 0.5 : Der Code-Smell bezieht sich auf Funktionen mit problematischem Anteil von `#ifdef`-Direktiven, daher stellen wir die Anforderung von mindestens 50% annotiertem Code. Ein kleiner Schwellwert könnte zu vielen false-positives führen und daher fälschlicherweise nicht Smell-behaftete Dateien als solche zu kennzeichnen.
- NOFC = 2 : Da wir die Streuung von Features als essentielle Charakteristik von Annotation Bundles betrachten, sehen wir eine Mindestanzahl von zwei verschiedenen Features in der Funktion als nötig an.
- NOFL = 2 : Um Funktionen mit nur einer großen Sektion von annotiertem Code aus dem Weg zu gehen, ist es zwingend notwendig, dass mindestens zwei verschiedene Feature Locations in der Funktion eingeführt werden.
- NS = 1 : Geschachtelte Feature Locations erhöhen die Komplexität dieses Smells sehr. Daher kommen nur Funktionen mit mindestens einem geschachtelten `#ifdef`-Block als potentiell Smell-behaftet in Frage.

Annotation File
Die folgenden Parameter für Annotation File werden alle auf Granularitätsebene von Dateien überprüft.

- Verhältnis von LOAC zu LOC einer Datei = 0.5 : Eine Datei mit großem Anteil an annotiertem Code – mit dem möglicherweise auch neue Funktionen und Parametern eingeführt werden – ist wahrscheinlich schwierig zu verstehen und zu warten. Daher wählen wir nur Dateien mit mindestens 50% annotiertem Code als potentiell Smell-behaftet.
- NOFC = 2 : Wie bereits beim Annotation Bundle ist auch im Annotation File die Streuung von Features von Bedeutung. Ein Feature allein hat einen geringeren Einfluss auf Lesbarkeit des Quellcodes, da nur eine Variante mit und eine Variante ohne Feature Code existieren würde. Mehrere Features, vor allem in komplexen und geschachtelten Feature Locations, erhöhen die Komplexität des Quellcodes. Daher werden nur Dateien mit mindestens zwei verschiedenen Feature Locations ausgewählt.

- NOFL = 2 : Auch im Annotation File wollen wir Dateien mit nur einer großen Sektion von annotiertem Code vermeiden. Somit müssen in einer Datei mindesten zwei verschiedene Feature Locations eingeführt werden, um für unsere Untersuchung als Smell-behaftet gelten zu können.

- NS = 1 : Da geschachtelte Feature Locations die Komplexität dieses Smells erhöhen, müssen Dateien mindestens einen geschachtelten `#ifdef`-Block enthalten, um als potentiell Smell-behaftet in Frage zu kommen.

Large Feature
Für das Large Feature gibt es eine Besonderheit. Alle Dateien, die mindestens ein Feature enthalten, werden zuerst als potentiell Smell-behaftet gekennzeichnet. Die folgenden Schwellwerte, die erreicht werden müssen, werden global über das ganze System akkumuliert und müssen nach der Untersuchung manuell in den Severity Reports überprüft werden. Erst nach dieser abschließenden Überprüfung werden dann die Dateien endgültig als Smell-behaftet klassifiziert. Die expliziten Werte für LOFC und NOFC haben wir von Daniel Meyer, der diese Werte empirisch anhand der der Analyse von sieben Systemen ermittelt hat, übernommen [Mey15].

- LOFC = 1122 und NOFC = 56 im ganzen System: In den Severity Reports werden dann die Schwellwerte überprüft. Es werden alle Features als Large Feature angesehen, die mehr als 1122 LOFC und 56 NOFC über das gesamte System hinweg besitzen.

4.1.2 Festlegen des Commit-Fensters

Ein essentieller Punkt in der Versuchsdurchführung ist es ein geeignetes Commit-Fenster, also den Abstand zwischen zwei Snapshots, zu wählen. Die richtige Größe des Commit-Fensters ist wichtig, um eine möglichst genaue Smell-Analyse durchführen zu können. Durch Änderungen der Dateien kann es passieren, dass Smell-behaftete Stellen in Dateien behoben werden. So kann es sein, dass spätere Änderungen an der gleichen Datei fälschlicherweise als Änderungen an einer Smell-behafteten Datei gezählt werden, obwohl der Smell dort gar nicht mehr vorhanden ist. Ebenfalls ist der umgekehrte Fall möglich, in dem eine Änderung erst zu einer Smell-behafteten Datei führt, diese jedoch vorher als nicht Smell-behaftet klassifiziert wurde. Es muss somit ein Kompromiss gefunden werden, so dass das Window nicht so klein ist und es kaum Änderungen gab und nicht so groß ist und die Dateien wie zuvor beschrieben veraltet sind.

Im Vergleich zur adaptierten Studie von Khomh et al. liegt in der Wahl des Commit-Fensters ein wesentlicher Unterschied zu unserem Versuch. In ihrer Arbeit wählten Khomh et al. als Fenster die Zeit zwischen zwei Releases eines Softwaresystems. Die Releasehistorien, beziehungsweise die Abstände der einzelnen Releases der Beispielsysteme, die wir untersuchen, sind sehr unterschiedlich und zum Teil sehr kleinteilig. Beispielsweise besitzt das System OpenVPN 55 Releases bei lediglich 674 Bugfix-Commits. Daher haben wir uns gegen die Wahl dieser Art des Windows entschieden. Stattdessen haben wir einen konstanten Abstand zwischen den Snapshots gewählt.

Eine konstante Größe des Commitwindows gewährleistet untereinander vergleichbare Ergebnisse. Dabei gibt es verschiedene Größen, die benutzt werden können um konstante Windows zu erstellen. Beispielsweise die Zeit, die Anzahl an Commits oder Bugfix-Commits. Da wir im speziellen die Fehleranfälligkeit untersuchen, sind die Bugfix-Commits von besonderer Bedeutung für die Studie. Bei einer konstanten Anzahl an generellen Commits oder einem kontanten Zeitfenster besteht die Gefahr, dass in diesem Fenster keine oder nur sehr wenige Bugfix-Commits getätigt wurden. Daher kommt nur eine konstante Anzahl an Bugfix-Commits als Commit-Fenster in Frage, um sicher zu gehen, dass jeder Snapshot eine Menge an untersuchbaren Bugfixes besitzt.

Mit der Entscheidung die Bugfix-Commits als Indikator für die Commit-Fenster zu nehmen, stellt sich als nächstes die Frage wie groß das Fenster zu wählen ist. Da der zur Auswertung genutzte Fisher-Test keine Anforderungen an die Größe der Werte stellt und auch bei kleinen Populationen robuste Ergebnisse liefert, ist diese Entscheidung nicht kritisch. So gibt es bei Khomh et al. beispielsweise Fenster mit lediglich zehn Bugfixes, deren Auswertung trotzdem ein brauchbares Ergebnis liefert. Die Wahl, die wir getroffen haben, ist somit ein Kompromiss zwischen der Aktualität der Snapshots und der Performance bei der Untersuchung. In der Untersuchung von Khomh et al. wurden in den drei Systemen, die eine ähnliche Größe wie die Systeme, die wir untersuchen, haben, zwischen 30 und 200 Bugfixes im Schnitt pro Release analysiert. In Anlehnung an die Größe dieser Fenster haben wir unser Commit-Fenster auf eine konstante Anzahl von 100 Bugfixcommits festgelegt.

4.1.3 Fallbeispiele

Zum Durchführen der Studie werden Softwaresysteme benötigt, die bestimmte Kriterien erfüllen. Um die Variability-aware Code-Smells mit SKUNK detektieren zu können, werden Systeme benötigt, deren Variabilität durch C-Präprozessoren implementiert sind. Weiterhin muss der Quellcode frei und über Github verfügbar sein, da unsere Umsetzung des Konzepts auf die Untersuchung von GitHub-Repositories aufbaut. Außerdem sollten die Systeme aus unterschiedlichen Domänen kommen um auszuschließen, dass bestimmte Eigenschaften von spezifischen Anwendungsdomänen die Ergebnisse beeinflussen. Ein weiteres Kriterium ist die Größe des Software-Systems. Es sollte von mittlerer Größe sein, das heißt, genügend Lines of Code besitzen, um eine ausreichende Menge an Code-Smells zu ermöglichen. Jedoch ist die Größe aus Performance-Gründen auch beschränkt. Weiterhin sollten die Software-Systeme eine erhebliche Anzahl an Lines of Annotated Code besitzen. Da die überprüften Smells größtenteils auf starke Annotierung aufbauen, haben Softwaresysteme mit mehr annotiertem Code eine größere Wahrscheinlichkeit Smells zu besitzen. Schlussendlich ist eine ausreichende Zahl an Commits bzw. Bugfix-Commits nötig um genügend Snapshots generieren zu können.

Die getroffene Auswahl ist eine Teilmenge der 40 von Liebig et al. analysierten C-Präprozessor basierten Softwaresysteme [LAL+10]. Eine Übersicht der gewählten Systeme ist in Tabelle 4.1 und Tabelle 4.2 gegeben und zusätzlich in Abbildung 4.1 grafisch aufgearbeitet. Für jedes System werden das Datum des ersten und letzten Snapshots, die Spanne der Files und Lines of Code, die Anzahl der Commits und Bugfix-Commits gezeigt. Die Anzahl der Files und LOC (nur nicht-leere und

System	Aktuellste Version	Datum	Domäne
BusyBox	1.24.1	Jan 2000 – Feb 2014	UNIX Werkzeugsammlung
Apache	2.4.17	Mar 1996 – Aug 2015	Web Server
OpenLDAP	2.4.40	Nov 1998 – Jul 2014	LDAP Verzeichnisdienst
OpenVPN	2.3.8	Nov 2008 – Mar 2015	Sicherheitsanwendung
Pidgin	2.10.9	Jun 2000 – Mar 2015	Mitteilungsdienst (IM)
SQLite	3.9.2	Mar 2001 – Aug 2010	Datenbanksystem

Tabelle 4.1: Analysierte Systeme und ihre Anwendungsdomänen

System	`.c` Files	kLOC	Commits	Fixes	Snapshots
BusyBox	144–714	33,1–177,9	13 948	4 136	41
Apache	54–315	15,5–174,3	28 413	5 413	52
OpenLDAP	254–506	52,4–251,4	21 877	6 062	58
OpenVPN	73–83	41,6–47,0	1 644	574	5
Pidgin	67–624	22,8–313,0	37 570	8 372	83
SQLite	27–146	14,5–102,7	15 587	1 815	18

Tabelle 4.2: Übersicht über die Charakteristika der Fallbeispiel-Systeme

nicht-kommentierte C-Code Zeilen) wurden mit dem CLOC[1]-Tool gemessen, wobei die Anzahl der Commits und Bugfix-Commits mithilfe der selbst geschriebenen MetricMiner2-Studie ermittelt wurden (siehe Abschnitt 3.4). In der letzten Spalte ist angegeben, wie viele Snapshots erstellt und untersucht wurden.

Bei der Betrachtung der Anzahl der Snapshots fällt auf, dass in den Systemen Apache und OpenLDAP jeweils zwei Snapshots mehr möglich wären. Wir mussten diese vier Snapshots jedoch durch Unregelmäßigkeiten in der Releasehistorie der beiden Systeme entfernen. In den vier betroffenen Commits wurden, höchstwahrscheinlich durch Fehler in der Commitdurchführung, Versionen ausgecheckt, die für die Systeme nicht repräsentativ sind. Im Fall von Apache wurde beispielsweise zwei mal ein SSL-Modul von Apache ausgecheckt, welches normalerweise innerhalb der Ordnerstruktur von Apache liegt. In diesen beiden Fällen wurden jedoch nur die Moduldateien ausgecheckt. Da sich dieser Zustand reproduzieren lässt, vermuten wir eine Unregelmäßigkeit in der Commitdurchführung.

Die Systeme sind alle aus unterschiedlichen Domänen und in verschiedenen Hinsichten unterschiedlich groß. Das System BusyBox[2] ist von der Dateianzahl her das größte, während Pidgin[3] die meisten LOC und Commits besitzt. Apache[4] wiederum hat die längste Entwicklungszeit. OpenVPN[5] ist, sowohl was die Dateien, die LOC

[1] http://cloc.sourceforge.net/
[2] https://github.com/mirror/busybox
[3] https://github.com/tieto/pidgin
[4] https://github.com/apache/httpd
[5] https://github.com/OpenVPN/openvpn

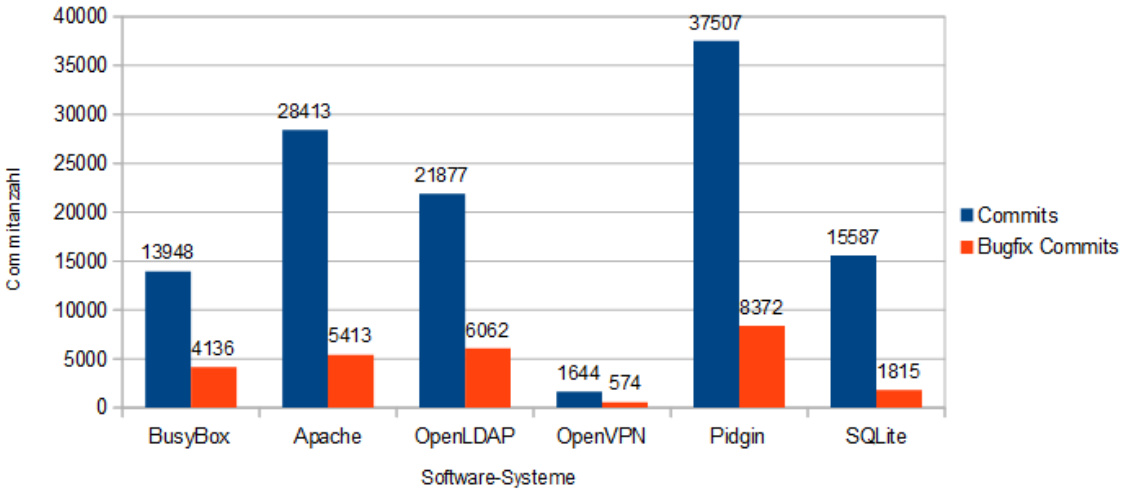


Abbildung 4.1: Anteil an Bugfix-Commits in den Fallbeispielen

als auch die Commits betrifft, das kleinste untersuchte System. SQLite[6] besitzt den kleinsten absoluten Anteil an Bugfix-Commits auf die gesamten Commits gesehen. Der LDAP Verzeichnisdienst OpenLDAP[7] ist in jeder Kategorie im Mittelfeld, und hat, abgesehen von OpenVPN, das größte Verhältnis von Bugfix-Commits zu Commits. Im Bereich der mittelgroßen Software-Systeme ermöglichen diese Variation eine gewisse Generalisierung der Ergebnisse.

4.2 Methodik

Im folgenden Abschnitt wird die Methodik zur Durchführung der Studie für jede Forschungsfrage (siehe Abschnitt 3.2) einzeln erklärt. Zum einen werden die statistischen Methoden, die zur Beurteilung der Ergebnisse genutzt werden, vorgestellt. Zum anderen werden Besonderheiten und Anpassungen am Setup des in Abschnitt 3.3 vorgestellten Konzept, sofern sie nötig sind, präsentiert.

RQ 1: *Gibt es einen Zusammenhang zwischen Code-Smells und der Fehleranfälligkeit?*
Wir untersuchen, ob Fehler in Dateien im Zusammenhang mit Smell-behafteten Dateien stehen. Der Zusammenhang sei dabei unabhängig von der Art des betroffenen Code-Smells. Daher testen wir, ob die Verhältnisse von Dateien, die mindestens einmal fehlerbehebend geändert wurden, zu denen, die nicht geändert wurden, sich signifikant unterscheiden zwischen Dateien, die Smell-behaftet sind, und denen, die es nicht sind. Zu diesem Zweck nutzen wir den exakten Test nach Fisher für die Nullhypothese H_{01}. Können wir die Nullhypothese ablehnen, unterstützt dies unsere Annahme, dass Smell-behaftete Dateien häufiger fehlerbehebend geändert werden als andere Dateien.

Zusätzlich zur Signifikanz der Ergebnisse berechnen wir die Odds Ratio (OR), welche die Wahrscheinlichkeit angibt, dass ein bestimmtes Ereignis auftritt. In unserer Untersuchung ist dieses Ereignis der Fall, dass ein Fehler in einer Smell-behafteten

[6] https://github.com/mackyle/sqlite
[7] https://github.com/osstech-jp/openldap

Datei vorkommt. Die Odds Ratio ist definiert als das Verhältnis der Chance p, dass ein Ereignis in einer Stichprobe (Experimentalgruppe) auftritt, zu der Chance q, dass es in der anderen Stichprobe (Kontrollgruppe) vorkommt: $OR = \frac{p/(1-p)}{q/(1-q)}$. In unserem Fall ist die Experimentalgruppe die Menge an Dateien die von einem Code-Smell betroffen ist, wohingegen die Kontrollgruppe die Menge an Dateien ohne einen Smell ist. Eine Odds Ratio von 1 sagt aus, dass das Ereignis gleichwahrscheinlich in beiden Gruppen auftritt. Ist die OR > 1 zeigt dies, dass es wahrscheinlicher ist, dass das Ereignis in der Experimentalgruppe auftritt, während eine OR < 1 das Gegenteil indiziert.

RQ 2: *Gibt es einen Zusammenhang zwischen bestimmten Arten von Code-Smells und Fehleranfälligkeit?*
Wir wollen die Beziehung von spezifischen Code-Smells zur Fehleranfälligkeit verstehen und untersuchen. Wir untersuchen zuerst mit der parameterfreien Spearman-Korrelation (siehe Abschnitt 2.3), ob es Korrelationen und den Smells gibt, um zu überprüfen, ob diese sich untereinander beeinflussen. Dies muss bei der Beantwortung der Frage beachtet werden.

Zur Beantwortung von RQ 2 wird der exakte Test nach Fisher angewandt. In diesem Fall werden jedoch nicht Dateien nach generell Smell-behaftet und nicht Smell-behaftet kategorisiert, sondern für jeden der Variability-aware Code-Smells einzeln. Das bedeutet, der Test wird drei mal durchgeführt – für jeden Smell einzeln – wobei die Dateien aufgeteilt werden in „von diesem Smell betroffen“ und „von diesem Smell nicht betroffen“. Die Dateien, die von dem Smell nicht betroffen sind, können jedoch von den anderen Variability-aware Code-Smells betroffen sein.

RQ 3: *Ist der Zusammenhang zwischen Code-Smells und der Fehleranfälligkeit unabhängig der Dateigröße?*
Bei der Untersuchung des Zusammenhangs von Variability-aware Code-Smells und der Dateigröße haben wir zuerst für jeden Snapshot die durchschnittliche Dateigröße von Smell-behafteten und nicht Smell-behafteten Dateien berechnet. Die Größenermittlung erfolgt dabei auf den untersuchten .c-Dateien je Snapshot. Mit diesen Daten wurde dann die Cohen d Effektstärke berechnet (siehe Abschnitt 2.3).

Für RQ 3 wird ebenfalls, wie bei den Untersuchungen zur ersten Forschungsfrage, der exakte Test nach Fisher genutzt um die Odds Ratios und statistische Signifikanz zu überprüfen. Der einzige Unterschied besteht darin, dass nur Dateien untersucht werden, die größer sind als 75% aller Dateien des jeweiligen Snapshots.

4.3 Ergebnisse

In diesem Abschnitt präsentieren wir die Ergebnisse der durchgeführten empirischen Studie. Die Ergebnisse werden für jede Forschungsfrage sowohl grafisch, als auch tabellarisch aufgearbeitet und präsentiert. Demenstprechend ist diese Sektion in die einzelnen Forschungsfragen gegliedert. Weiterhin diskutieren wir die Ergebnisse, stellen Besonderheiten und gefundene Muster vor.

4.3.1 RQ1 – Einfluss von Variability-aware Code Smells auf die Fehleranfälligkeit

Zur Untersuchung und Beantwortung der ersten Forschungsfrage werden zunächst die Darstellungen der Ergebnisse erläutert. Anschließend werten wir die Ergebnisse aus und beschreiben Besonderheiten und gefundene Muster. Abschließend ziehen wir eine Schlussfolgerung aus den Untersuchungsergebnissen.

Die Ergebnisse des exakten Tests nach Fisher sind in Abbildung 4.2 abgebildet. In diesem Diagramm sind die Odds Ratios und die statistische Signifikanz zusammengefasst. Jeder untersuchte Snapshot wurde auf eine der drei folgenden Klassen aufgeteilt:

- Blau: Der Snapshot hat eine OR > 1 und das Ergebnis ist statistisch signifikant (p-Wert $<= 0.05$)
- Gelb: Der Snapshot hat eine OR > 1, jedoch ist das Ergebnis nicht statistisch signifikant (p-Wert > 0.05)
- Rot: Der Snapshot hat eine OR ≤ 1 und das Ergebnis ist nicht statistisch signifikant (p-Wert > 0.05)

Für jedes System kann sowohl der prozentuale Anteil als auch die absolute Anzahl an Snapshots zu den zugehörigen Klassifizierungen abgelesen werden. Ein detaillierter Einblick in die Ergebnisse jedes einzelnen Snapshots ist in Tabelle A.1 bis Tabelle A.6 im Kapitel A gegeben. Zusätzlich sind die Odds Ratios in Abbildung 4.3 mithilfe von Boxplots visualisiert. Zur Berechnung der Boxplots benutzen wir die Defaultwerte von R. Das bedeutet, dass die Box die mittleren 50% aller Werte umschließt. Der Strich innerhalb der Box gibt den mittleren Wert (*Median*) an, das heißt die Box unterhalb des Striches beinhaltet genausoviele Werte wie die Box oberhalb des Striches. Den Abstand der unteren Begrenzung der Box zur oberen Begrenzung der Box wird *Interquartilsabstand* genannt. Die *Antennen* (*Whisker*) stellen die außerhalb der Box liegenden Daten dar. In unserem Fall ist die Länge auf das 1,5-fache des Interquartilabstands beschränkt. Dabei enden die Whisker jedoch nicht genau nach dieser Länge, sondern bei den Werten aus den Daten, die noch innerhalb dieser Grenzen liegen. Alle Werte die außerhalb dieses Wertebereichs liegen sind Ausreißer, die wir zur besseren Darstellung der Boxen nicht mit abbilden. In Abbildung 4.4 sind für jedes untersuchte System jeweils der Verlauf des Anteils an Smell-behafteten Dateien (blaue Linie) und der Bugfixes in Smell-behafteten Dateien (rote Linie). Für das System BusyBox werden in Abbildung 4.5 zwei Snapshot-Abschnitte verglichen. Dabei wird auf der linken Seite die Signifikanz in bekannter Art und Weise präsentiert. Auf der rechten Seite werden die Odds Ratios der beiden Abschnitte mit Boxplots dargestellt.

In Abbildung 4.2 ist an den roten Balken zu sehen, dass nicht in jedem Snapshot die Wahrscheinlichkeit, dass Smell-behaftete Dateien gefixed werden, größer ist als die Wahrscheinlichkeit, dass dies in nicht Smell-behafteten Dateien passiert. Über alle Systeme zusammengezählt ist dies jedoch nur in 18 von 257 Snapshots, und somit in etwa 7%, der Fall. Über die Systeme hinweg variieren die Odds Ratios. In

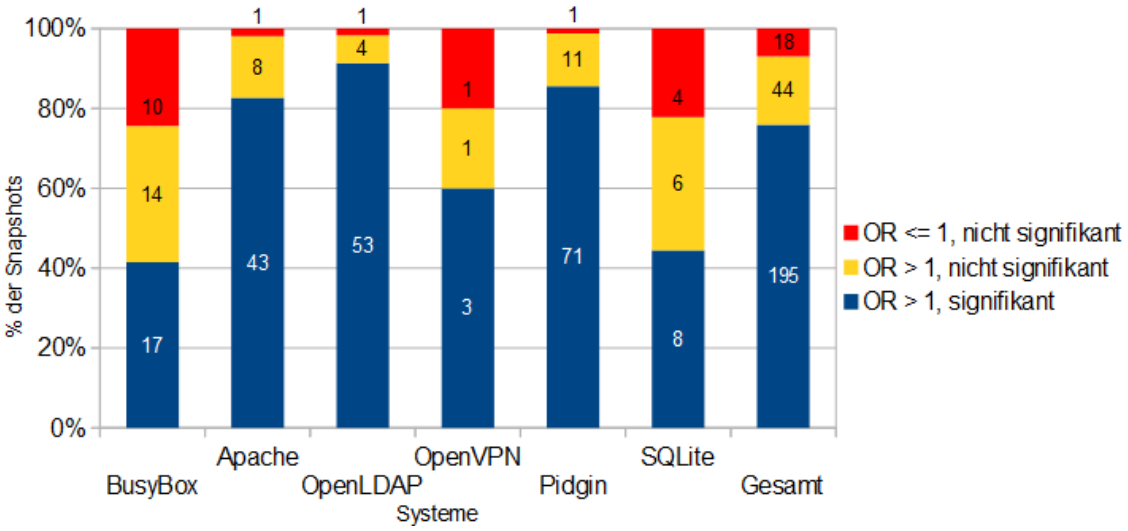


Abbildung 4.2: Grafische Darstellung der Odds Ratios unter Einbeziehung der Ergebnisse des exakten Tests nach Fisher

wenigen Fällen sind die ORs nahe 1 oder liegen darunter. In diesen Fällen bieten die Odds Ratios keine Anhaltspunkte für einen signifikanten Einfluss von Code-Smells. Im Falle negativer Odds Ratios indizieren diese sogar einen gegenteiligen Effekt, was bedeutet, dass in diesen Snapshots die Wahrscheinlichkeit kleiner ist, dass Smell-behaftete Dateien fehlerhaft sind, als die Wahrscheinlichkeit, dass nicht Smell-behaftete Dateien Fehler beinhalten. In Abbildung 4.3 ist zu erkennen, dass die mittleren Odds Ratios für alle Systeme bei etwa 3 (genau: 2,94) liegen. Folgend bezeichnen wir den Median als mittleren Wert. Die mittleren Odds Ratios aller einzelnen Systeme weichen nur gering davon ab. Die Werte liegen zwischen 2,87 und 3,62. Einzige Ausnahme ist hier das System BusyBox. Die mittleren Odds Ratios sind in diesem System 1,57 und damit fast halb so groß wie bei den restlichen Systemen. Diese niedrigen Odds Ratios überschneiden sich mit der Beobachtung, dass BusyBox den kleinsten Anteil an signifikanten Ergebnissen (41,5%) besitzt (siehe Abbildung 4.2). Das System SQLite hat ähnlich geringe Anteile an signifikanten Ergebnissen (44,4%) jedoch mittlere Odds Ratios von 3,08. An der Größe der Box von SQLite ist zu erkennen, dass die Streuung der ORs in diesem System sehr groß ist. Die Hälfte aller Daten liegen zwischen 1,66 und 6,07. Die Streuungen der anderen Systeme sind sichtbar geringer. Zusätzlich ist die obere Begrenzung die höchste aller sechs Systeme. Die geringe Menge an signifikanten Ergebnissen kann somit die geringen ORs in BusyBox somit nicht allein erklären. Das System BusyBox scheint sich in Bezug auf unsere Untersuchungen von den anderen Systemen zu unterscheiden, was die weiteren Beobachtungen zu RQ 1 und RQ 2 bestätigen. Interessant ist, dass die Boxen der drei größten Systemen Apache, OpenLDAP und Pidgin ähnlich groß und in ähnlichen Wertebereichen liegen. Die unteren und oberen Begrenzungen unterscheiden sich jeweils um weniger als 0,15. Nehmen wir die mittleren ORs aller Systeme als Indikator, heißt das, dass die Chance Fehler in Smell-behafteten Dateien zu finden um 194% größer ist als Fehler in nicht Smell-behafteten Dateien zu finden. Lediglich das System BusyBox weicht mit einer 57% höheren Chance, stark

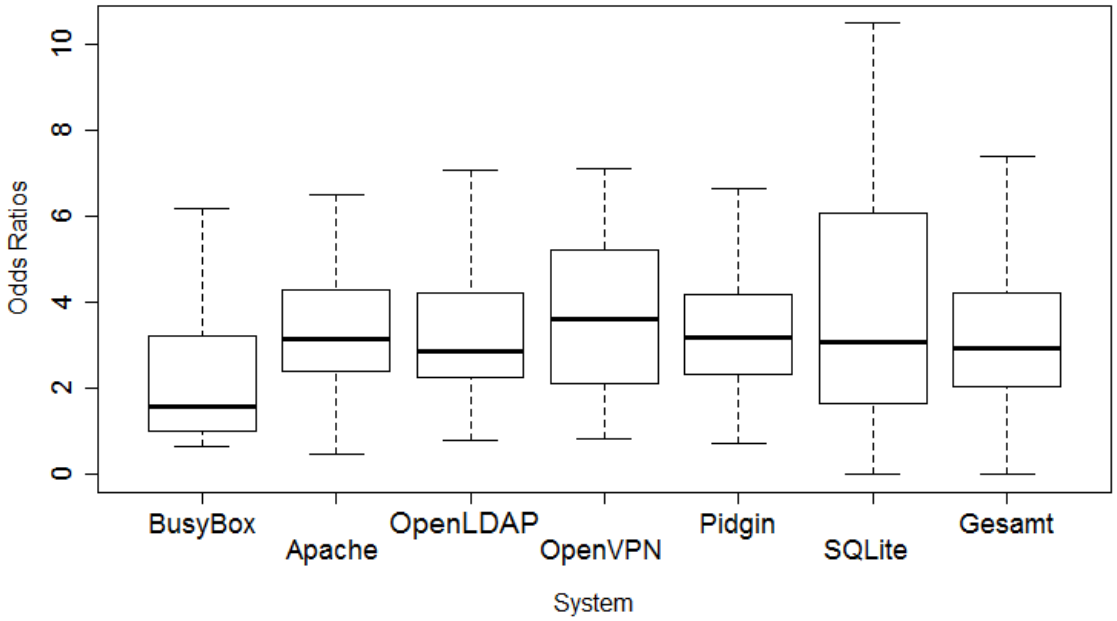


Abbildung 4.3: Vergleich der Odds Ratios aller Systeme

von den anderen Systemen ab. Wie statistisch signifikant diese ermittelten Chancen sind, klären wir mithilfe des exakten Tests nach Fisher.

Mit der Nutzung des Fisher-Tests kommt das Problem einher, dass dieser nur zwischen Dateien, die mindestens einmal geändert wurden, und denen, die nicht geändert wurden, unterscheidet. Eine Datei kann in dem untersuchten Fenster jedoch mehr als einmal geändert werden. In Abbildung 4.4 zeigen wir daher den Anteil von fehlerbehebenden Änderungen in Smell-behafteten Dateien und beziehen dabei auch mehrfache Änderungen auf ein und derselben Datei mit ein. Die rote Kurve steht dabei für den Anteil der Bugfixes in Smell-behafteten Dateien und die blaue Kurve für den Anteil an Smell-behafteten Dateien. Hier ergibt sich ein ähnliches Bild wie es zuvor bei den Odds Ratios beobachtet wurde. Das Verhältnis von fehlerbehebenden Änderungen in Smell-behafteten Dateien zu denen in nicht Smell-behafteten Dateien ist sehr selten kleiner als das Verhältnis von Smell-behafteten zu nicht Smell-behafteten Dateien. Dies ist ein Indiz dafür, dass Smell-behaftete Dateien häufiger fehlerbehebend geändert werden als andere Dateien. Auch bei dem System BusyBox ist dies zu beobachten. Das lässt vermuten, dass die niedrigen ORs für dieses System daraus resultieren, dass viele Änderungen in einem Window häufig auf den gleichen Dateien stattfanden.

In rechten Teil der Abbildung 4.5 ist an den Boxplots zu erkennen, dass in dem Software-System BusyBox ab dem Snapshot 15 deutlich mehr Snapshots niedrige ORs besitzen. Während der mittlere Wert (Median) in den ersten 15 Snapshots bei 3,77 liegt, beträgt er in den restlichen Snapshots lediglich 1,19. Der deutliche Unterschied ist auch an der unteren Begrenzung der ersten Box (2,54) und der oberen Begrenzung der zweiten Box (1,57) zu erkennen. Das bedeutet, dass mindesten 50% aller Odds Ratios der ersten 15 Snapshots über mindestens 50% der Odds Ratios der folgenden Snapshots liegen. Diesen Eindruck bestätigt auch der linke Teil der Abbil-

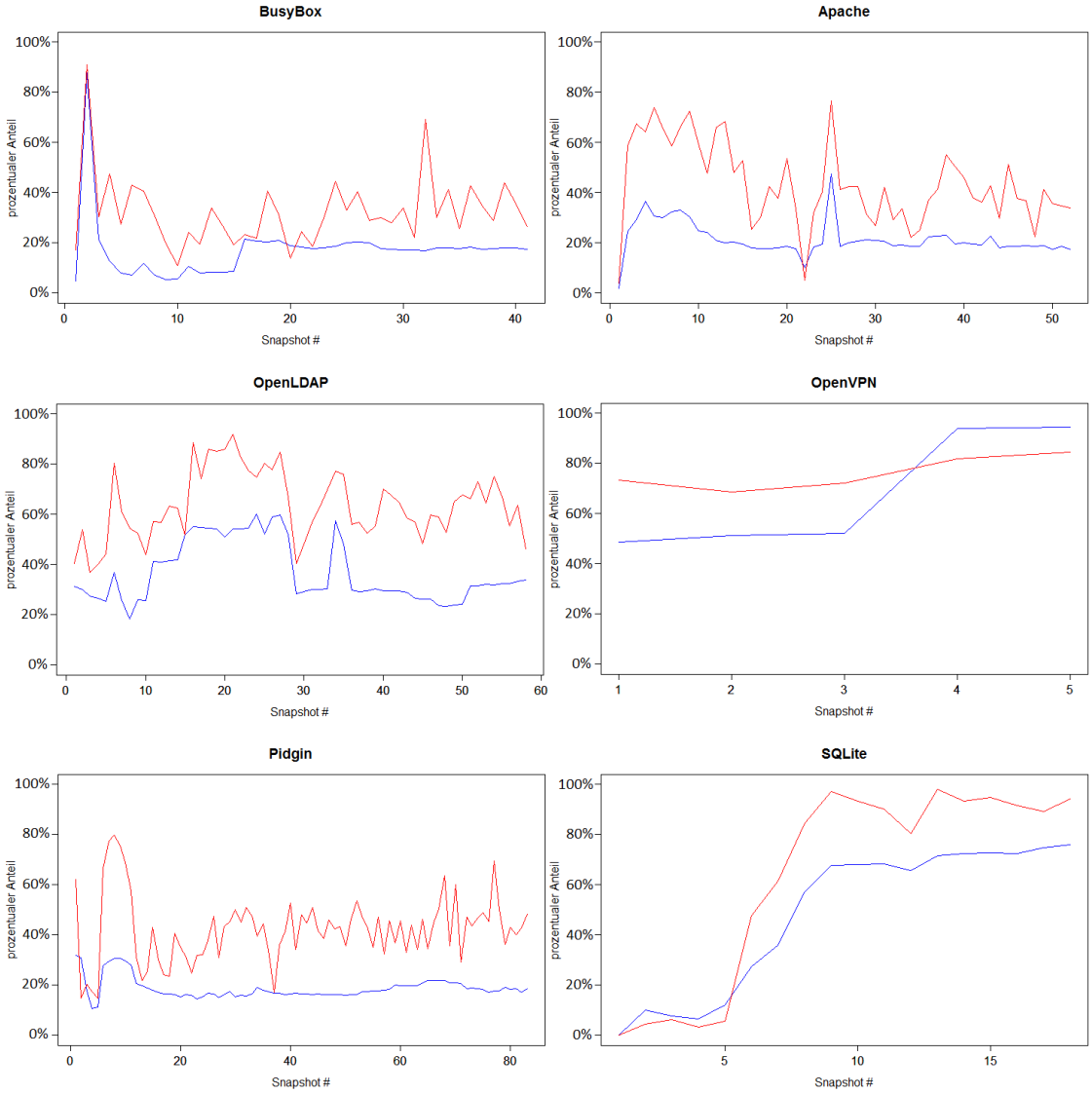


Abbildung 4.4: Übersicht über die Anteile der Smell-behafteten Dateien (blau) und der Bugfixes in Smell-behafteten Dateien (rot) aller Beispielsysteme

dung 4.5. Dieses Diagramm zeigt, dass die Snapshots nach dem 15. deutlich seltener signifikante Ergebnisse liefern. Dies ist in 20 von 26 Snapshots nach Nummer 15 der Fall, während bei den ersten 15 Snapshots lediglich 3 keine signifikanten Ergebnisse liefern. Dabei können wir in Abbildung 4.4 beobachten, dass ab Snapshot Nummer 16 der Anteil der Smell-behafteten Dateien von 9 auf 21% (von 51 auf 138 Dateien) ansteigt. Wir gehen davon aus, dass dies ein möglicher Grund für die unterschiedlichen Ergebnisse über das System hinweg ist. Das gleiche Phänomen ist ebenfalls im System OpenVPN zu beobachten. Dort springt der Anteil Smell-behafteter Dateien für die beiden letzten Snapshots von 52% auf 94%. Dies sind genau die beiden Snapshots, die keine signifikanten Ergebnisse liefern (siehe Tabelle A.4). In dem System SQLite steigt der Anteil ebenfalls innerhalb von wenigen Snapshots, ab dem fünften, von 6% auf über 60% an. Hier ist es jedoch im Gegensatz zu den anderen beiden Beispielen so, dass die ersten fünf Snapshots vor dem Anstieg alle keine si-

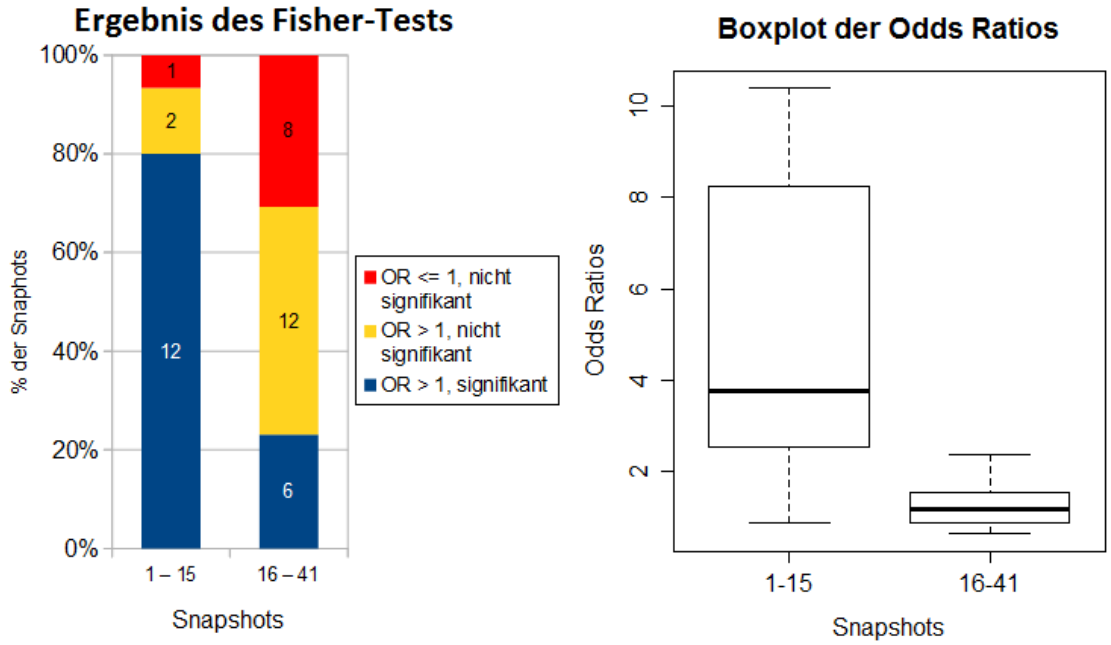


Abbildung 4.5: Vergleich der Snapshots 1-15 und 16-41 des Systems BusyBox für die Werte des Fisher-Tests (links) und der Odds Ratios (rechts)

gnifikante Ergebnisse bringen, wie in Tabelle A.6 zu erkennen ist. Den sprunghaften Anstieg an Smell-behafteten Dateien in diesen drei Systemen verursacht der Code-Smell Large Feature, während der Anteil der Smells Annotation Bundle und Annotation File relativ konstant bleibt, beziehungsweise nur langsam wächst. In den Systemen Apache, OpenLDAP und Pidgin ist kein Muster zu erkennen um die vereinzelt niedrigen ORs und schlechten p-Werte zu erklären.

Ausgehend von den gefundenen Ergebnissen folgern wir, dass es einen Zusammenhang von Code-Smells und der Fehleranfälligkeit gibt. In Abbildung 4.2 kann man sehen, dass in etwa 75% (195 von 267) aller Snapshots die Nullhypothese H_{01} abgelehnt werden kann. Eine abgelehnte Nullhypothese bedeutet, dass wir für diese Snapshots die statistisch fundierte Aussage treffen können, dass Variability-aware Code-Smells eine größere Fehleranfälligkeit bedeuten. Auffällig sind hier die Systeme BusyBox und SQLite, bei denen die Daten seltener – in etwas weniger als 50% der Snapshots – als bei den anderen Systemen signifikante Ergebnisse liefern. In den Systemen Apache, Pidgin und OpenLDAP wiederum liegt der Anteil an signifikanten Ergebnissen bei mehr als 80%. Dies sind auch die drei Systeme mit den meisten Snapshots. Die Resultate zeigen somit, dass Smell-behaftete Dateien häufiger fehlerbehend geändert werden als andere Dateien. Die Ablehnung von H_{01} und die ORs bieten einen empirisch belegten Beweis, dass Code-Smells eine höhere Fehleranfälligkeit verursachen und somit einen negativen Einfluss auf die Software-Entwicklung haben haben.

	AB			AF			LF			mind. 1 Smell		
System	min	max	mean	min	max	mean	min	max	mean	min	max	mean
BusyBox	0,0	5,1	3,7	1,4	6,0	4,7	1,4	85,2	12,9	4,9	87,6	17,1
Apache	1,9	11,0	5,7	0,0	5,5	3,0	0,0	44,9	17,1	1,9	47,5	21,5
OpenLDAP	2,8	10,0	6,0	4,8	10,6	7,3	14,8	58,2	33,3	18,5	60,0	36,7
OpenVPN	13,9	20,5	17,1	26,4	49,4	36,2	43,1	94,3	65,1	48,6	94,3	68,1
Pidgin	0,6	3,0	2,1	0,4	6,1	1,7	8,6	31,8	18,0	10,1	31,8	18,8
SQLite	0,0	9,9	6,4	0,0	24,6	15,5	0,0	73,4	44,2	0,0	76,1	48,2

Tabelle 4.3: Überblick über den minimalen, maximalen und mittleren prozentualen Anteil an Smell-behafteten Dateien für Annotation Bundle, Annotation File, Large Feature und den Fall, dass eine Datei von mindestens einem Smell betroffen ist

4.3.2 RQ2 – Einfluss spezifischer Code-Smells auf die Fehleranfälligkeit

Zur Beantwortung der zweiten Forschungsfrage werden wir zunächst die Präsentation der Ergebnisse beschreiben, bevor wir diese auswerten und diskutieren. Abschließend folgt eine Zusammenfassung und eine Schlussfolgerung, die wir aus den Resultaten ziehen.

In RQ 2 untersuchen wir den Einfluss einzelner Variability-aware Code-Smells auf die Fehleranfälligkeit. Zu diesem Zweck haben wir uns zuerst in Tabelle 4.3 und Abbildung 4.6 einen Überblick über die Verteilung der einzelnen Smells in den Systemen verschafft. Für jedes System werden in Tabelle 4.3 die minimalen, maximalen und mittleren prozentualen Anteile an Smell-behafteten Dateien dargestellt. In Abbildung 4.6 werden die prozentualen Anteile für jeden Variability-aware Code-Smell mithilfe von Boxplots visualisiert. Anschließend haben wir die Korrelation zwischen den Smells untersucht und in Abbildung 4.7 dargestellt. In Abbildung 4.8 sind die ORs und die Signifikanz des exakten Tests nach Fisher zusammengefasst. Für jedes System und jeden Code-Smell ist dort das Verhältnis der drei Klassifizierungen abgebildet. Die dargestellten Klassifizierungen sind identisch zu denen aus RQ 1 in Abschnitt 4.3.1.

In Tabelle 4.3 und Abbildung 4.6 ist zu sehen, dass, mit Ausnahme von OpenVPN und SQLite, die Code-Smells Annotation Bundle und Annotation File nur in geringer Menge (weniger als 10%) in den Testsystemen vorkommen. Weiterhin ist zu beobachten, dass der Smell Large Feature immer deutlich mehr Dateien als die restlichen Smells betrifft. Durch den deutlich geringeren Anteil an Dateien, die von Annotation Bundle und Annotation File betroffen sind, wird das Gesamtergebnis von ihnen weniger beeinflusst.

Die wenigen Korrelationen, die mit der Spearman Korrelation entdeckt werden konnten, sind in Abbildung 4.7 für die sechs untersuchten Software-Systeme dargestellt. Es wurden lediglich Korrelationen zwischen den Code-Smells Annotation Bundle und Annotation File gefunden. Für drei der Systeme (OpenVPN, Pidgin und SQLite) und die anderen Smell-Kombinationen wurden keine Korrelationen festgestellt. Die Abbildung zeigt für die Systeme jeweils die Anzahl an schwachen Korrelationen (zwischen 0,5 und 0,7) in Weiß. Der hellblaue Anteil zeigt die Snapshots in denen keine Korrelation festgestellt wurde. Dass es keine Korrelationen zwischen Large

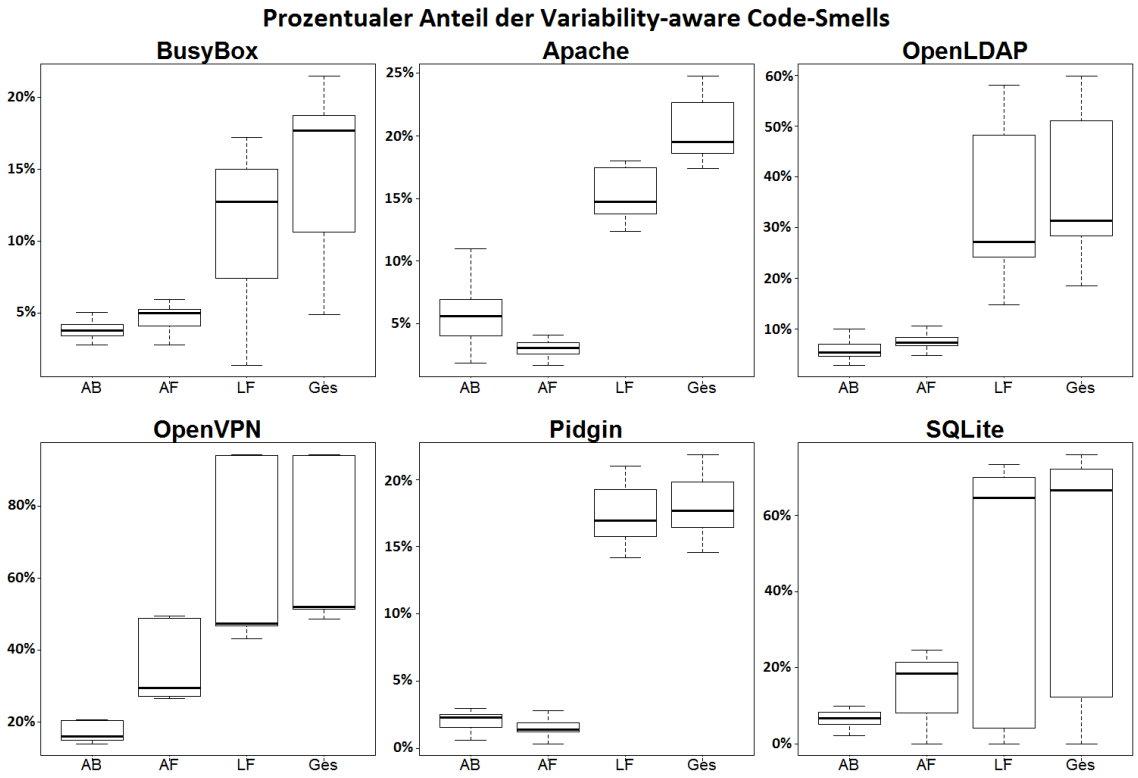


Abbildung 4.6: Grafische Darstellung der prozentualen Anteile von Smell-behafteten Dateien für die Variability-aware Code-Smells Annotation Bundle, Annotation File, Large Feature und den Fall, dass eine Datei von mindestens einem Smell betroffen ist

Feature und einem anderen Smell gibt, haben wir erwartet, da sie verschiedene Arten von Designschwächen ansprechen. Die geringe Korrelation zwischen den Smells Annotation Bundle und Annotation File ist aufgrund der ähnlichen Definition der Smells überraschend. Ein Grund dafür könnten die verschiedenen Granularitätsebenen sein, auf denen diese Smells arbeiten. Die geringen, beziehungsweise nicht vorhandenen, Korrelationen zwischen den Code-Smells bedeuten, dass alle drei Code-Smells voneinander unabhängig sind und einen Einfluss auf das Endergebnis haben können. Somit charakterisieren sie unterschiedliche Designschwächen und es ist sinnvoll alle drei Variability-aware Code-Smells zu detektieren und zu betrachten.

Die Abbildung 4.8 zeigt die zusammengefassten Ergebnisse des exakten Tests nach Fisher für die einzelnen Variability-aware Code-Smells und Systeme. Es ist anhand der blauen Balken zu erkennen, dass, bis auf BusyBox und OpenVPN, bei jedem System die Untersuchungen zum Large Feature deutlich mehr signifikante Ergebnisse liefern als die Untersuchungen zu den anderen beiden Smells. Zusammengenommen mit den gelben Balken sind die Anteile an ORs > 1 jedoch ähnlich bei allen drei Code-Smells. Ausnahmen sind hier wiederum BusyBox und der Code-Smell Annotation Bundle in den Systemen Apache und Pidgin, bei denen der Anteil an ORs > 1 kleiner ist. Die geringeren Anteile an signifikanten Ergebnissen für die Code-Smells Annotation Bundle und Annotation File liegt möglicherweise an der geringen Anzahl an betroffenen Dateien (siehe Tabelle 4.3). In den Systemen OpenLDAP, OpenVPN, Pidgin und SQLite stimmt die Anzahl an signifi-

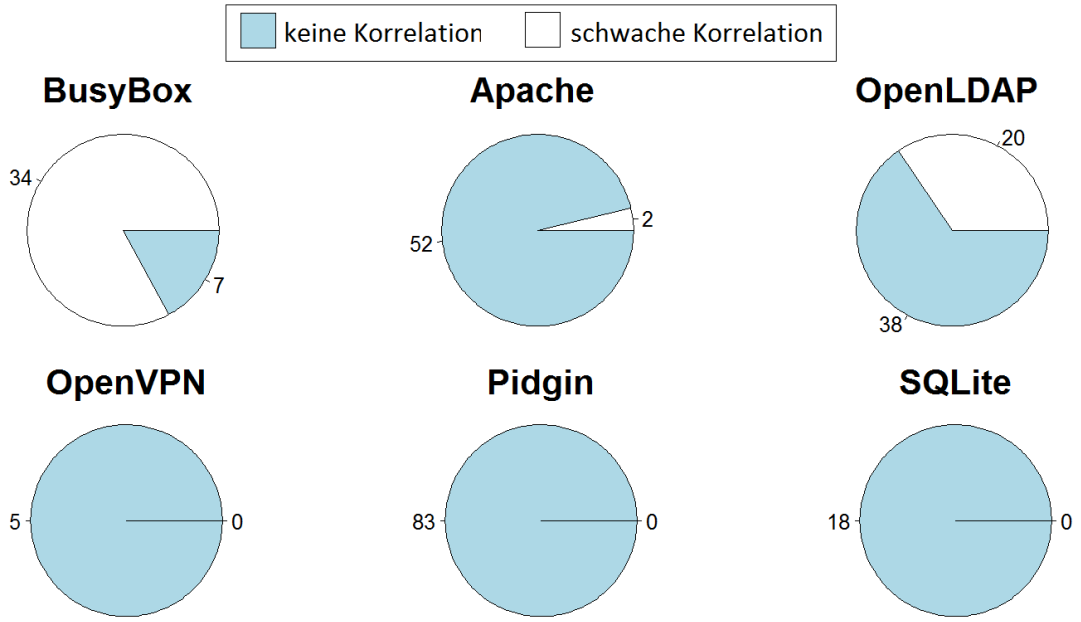


Abbildung 4.7: Darstellung der Korrelation zwischen Annotation Bundle und Annotation File in den sechs Systemen mit Anzahl der Snapshots als Beschriftung

kanten Odds Ratios (blaue Balken) für Large Feature und dem Gesamtergebnis überein. Auch die Anzahl der Snapshots die eine OR > 1 und nicht signifikant sind, weicht höchstens um 1 voneinander ab in diesen Systemen. Ein möglicher Grund dafür könnte sein, dass in den Systemen OpenLDAP und Pidgin die Menge an Dateien, die mit Annotation Bundle und Annotation File betroffen sind im Vergleich zu den Large Feature-behafteten Dateien noch geringer ist als in den anderen Systemen. Bei den Systemen OpenVPN und SQLite wiederum sind Dateien mit den beiden Annotation Smells oftmals auch von Large Feature betroffen. Somit haben die beiden Annotation Smells entweder aufgrund der geringen Menge oder der Überschneidungen einen geringeren Einfluss auf das Gesamtergebnis. Auffallend ist in dem System BusyBox, dass im Gegensatz zu allen anderen Systemen deutlich mehr Snapshots signifikante Ergebnisse und ORs > 1 bei den Smells Annotation Bundle und Annotation File lieferten, das Gesamtergebnis jedoch ähnlicher dem Ergebnis von Large Feature ist.

Aus den gewonnenen Ergebnissen schlussfolgern wir, dass alle drei untersuchten Variability-aware Code-Smells einzeln einen Einfluss auf die Fehleranfälligkeit haben. Wenn wir die signifikanten Werte zur Ablehnung der Nullhypothese H_{02} betrachten, sehen wir, dass wir sie für Large Feature in 70% der Fällen ablehnen können. Während wir H_{02} für Annotation Bundle in lediglich 49,4% und für Annotation File in 31,5% der Snapshots ablehnen können. Somit ist zu erkennen, dass der Smell Large Feature den größten Einfluss auf die Fehleranfälligkeit hat. Dieser Einfluss ist auch nicht ausschließlich auf die hohe Anzahl an Dateien, die von diesem Smell betroffen sind, zurückzuführen. Zusammenfassend gibt es somit einen Zusammenhang zwischen bestimmten Arten von Code-Smell und der Fehleranfälligkeit, jedoch nicht konsistent über alle Systeme und Snapshots hinweg.

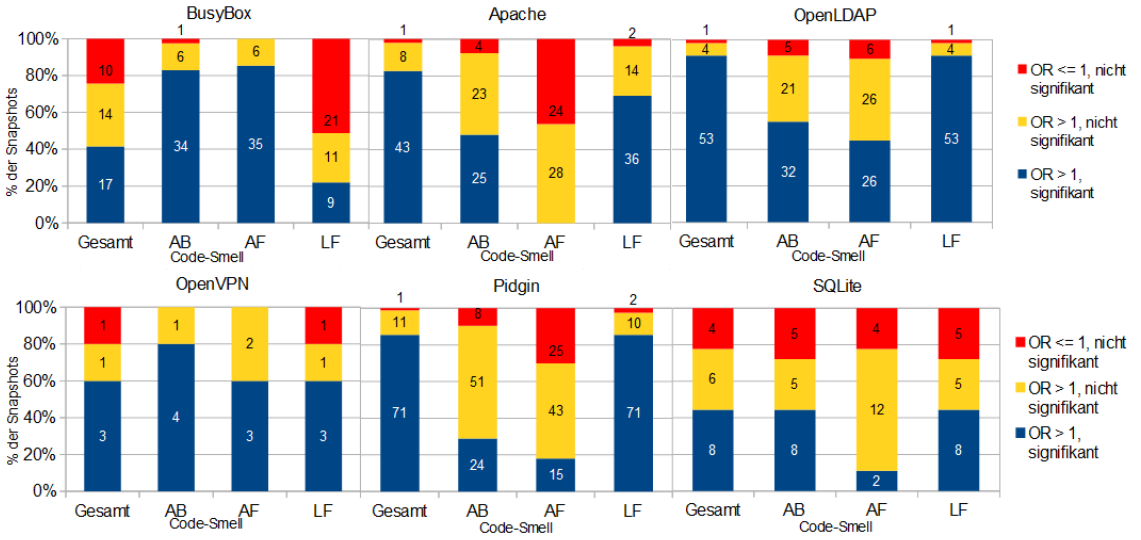


Abbildung 4.8: Grafische Darstellung der ORs unter Einbeziehung der Ergebnisse des exakten Tests nach Fisher für jeden Code-Smell und jedes System

4.3.3 RQ3 – Zusammenhang von Dateigröße, Präsenz von Code-Smells und Fehleranfälligkeit

Zur Betrachtung der dritten Forschungsfrage stellen wir zunächst die Darstellungen der Ergebnisse vor, um diese anschließend zu diskutieren und zu bewerten. Abschließen präsentieren wir eine Zusammenfassung und unsere Schlussfolgerung.

Um zu untersuchen, ob der Zusammenhang von Variability-aware Code-Smells und der Fehleranfälligkeit unabhängig der Dateigröße ist, haben wir in Abbildung 4.9 die durchschnittliche Größe von Smell-behafteten Dateien (blauer Balken) und nicht Smell-behafteten (oranger Balken) Dateien in jedem System gegenübergestellt. Zusätzlich sind in Tabelle 4.4 die berechneten Cohen d Effektstärken bezüglich der einzelnen Code-Smells und der Dateigröße in den einzelnen Software-Systemen abgebildet. Die rot markierten Werte zeigen eine geringe Effektstärke für den Smell in dem spezifischen System an. Das bedeutet in diesen Fällen sind Dateien, die mit dem spezifischen Code-Smell behaftet sind, nicht größer als Dateien, die nicht betroffen sind.

Die Ergebnisse des exakten Tests nach Fisher, der für RQ 3 nur Dateien, die größer als 75% der restlichen Dateien eines Snapshots sind, mit einbezieht, sind in Abbildung 4.10 aufgeführt. Wie in den beiden vorrangegangenen Abschnitten werden in diesem Diagramm die Odds Ratios und die statistische Signifikanz zusammengefasst. Jeder untersuchte Snapshot wurde auf eine von drei Klassen aufgeteilt. Im Diagramm kann sowohl der prozentuale Anteil als auch die absolute Anzahl an Snapshots zu den zugehörigen Klassifizierungen abgelesen werden. Ein detaillierter Einblick in die Ergebnisse jedes einzelnen Snapshots ist in Tabelle A.7 bis Tabelle A.12 im Kapitel A gegeben. In Abbildung 4.11 sind für jedes untersuchte System jeweils der Verlauf des Anteils an Smell-behafteten Dateien (blaue Linie) und der Bugfixes in Smell-behafteten Dateien (rote Linie) abgebildet. Hier werden für den Anteil der Bugfixes auch mehrfache Änderungen in einer Datei gewertet um das Verhältnis von allen Bugfixes bewerten zu können.

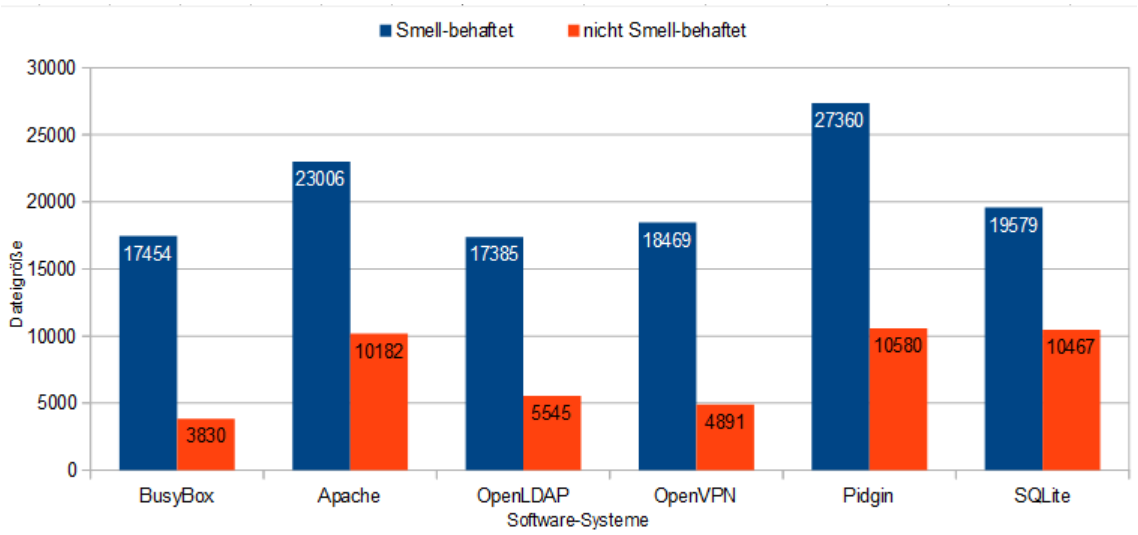


Abbildung 4.9: Vergleich der Dateigrößen von Smell-behafteten und nicht Smell-behafteten Dateien aller untersuchten Systeme

System	Annotation Bundle	Annotation File	Large Feature
BusyBox	3,9	4,1	1,6
Apache	3,7	0,1	2,2
OpenLDAP	0,8	1,5	1,8
OpenVPN	7,1	4,2	3,6
Pidgin	2,7	-0,2	4,0
SQLite	1,3	1,1	0,8

Tabelle 4.4: Übersicht über die Effektstärke der Größe von einzelnen Code-Smells über alle Beispielsysteme, rot markiert sind kleine Effektstärken

In Abbildung 4.9 ist zu sehen, dass die durchschnittliche Größe (arithmetisches Mittel) von Smell-behafteten Dateien größer ist als die von nicht Smell-behafteten Dateien. Diese Beobachtung bestätigt sich durch die Untersuchung der Effektstärken der einzelnen Code-Smells bezüglich der Dateigröße, welche in Tabelle 4.4 abgebildet sind. Hier ist zu sehen, dass für jeden einzelnen Code-Smell die betroffenen Dateien signifikant größer – mit einer hohen Effektstärke – sind als die Dateien, die nicht von dem jeweiligen Code-Smell betroffen sind. Lediglich die beiden rot markierten Werte zeigen eine geringe Effektstärke. Diese beiden Ausnahmen betreffen den Code-Smell Annotation File in den Systemen Apache und Pidgin.

Folgend vergleichen wir die Ergebnisse mit denen aus RQ 1 (siehe Abschnitt 4.3.1). Wenn man die abgebildeten Daten in Abbildung 4.10 mit denen aus Abbildung 4.2 vergleicht, sieht man an der Größe der roten Balken, dass es bei der Betrachtung von großen Dateien mehr Snapshots gibt bei denen die ORs < 1 sind als bei der Betrachtung aller Daten. Trotz dessen haben noch 204 von 257 Snapshots (79%) ORs > 1 (blauer und gelber Balken), und somit eine höhere Wahrscheinlichkeit, dass in großen Smell-behafteten Dateien Fehler behoben werden als in großen nicht Smell-behafteten Dateien. Man erkennt an den blauen Balken, dass der exakte Test nach Fisher jedoch nur in 30% der Fälle eine statistische Signifikanz hervor bringt. Ein

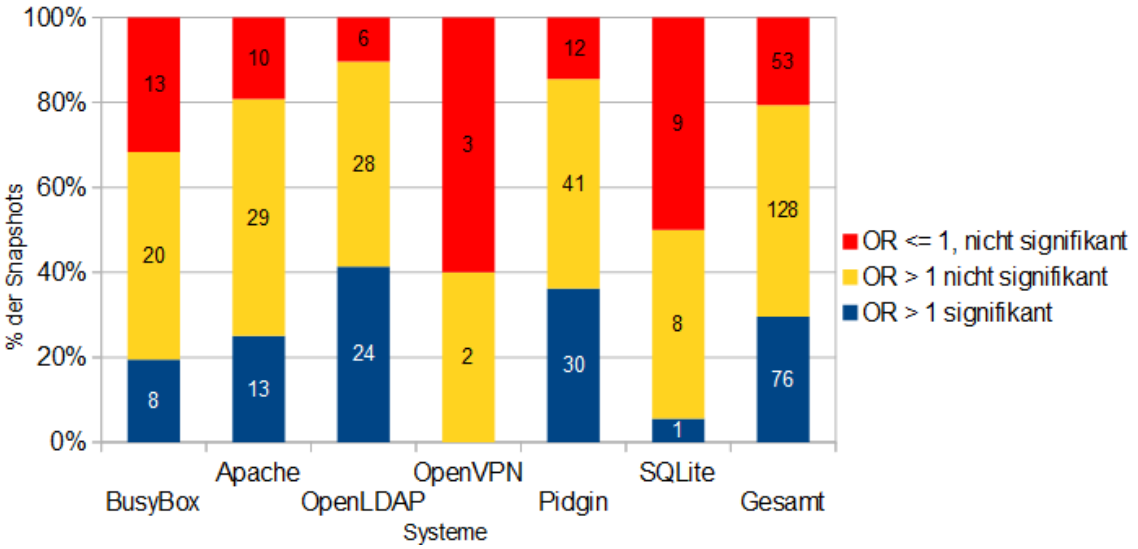


Abbildung 4.10: Grafische Darstellung der Odds Ratios unter Einbeziehung der Ergebnisse des exakten Tests nach Fisher bei großen Dateien

Grund dafür könnte die limitierte Anzahl an Dateien, die größer als 75% der Dateien sind, sein. Die Tabellen aus Kapitel A zeigen, dass immer wenn der Test statistisch signifikante Unterschiede zwischen den Proportionen von fehlerbehebenden Änderungen in Smell-behafteten und nicht Smell-behafteten Dateien erkannte, die Odds Ratios über 1,5 (meist sogar über 2) sind und Spitzenwerte von 25,67 erreichen.

Wie bereits für RQ 1 haben wir in Abbildung 4.11 den Anteil an Smell-behafteten Dateien und das Verhältnis von Bugfixes in Smell-behafteten und nicht Smell-behafteten Dateien gegenüber gestellt. Hier bietet sich ein ähnliches Bild. Nur in wenigen Fällen übersteigt der Anteil an Smell-behafteten Dateien den Anteil an Bugfixes in diesen Dateien. Dies ist in Indiz dafür, dass auch unter großen Dateien Smell-behaftete Dateien öfter fehlerbehebend geändert werden als nicht Smell-behaftete Dateien.

Die empirisch ermittelten Odds Ratios und der direkte Vergleich der Anteile der Smell-behafteten Dateien und Bugfixes in Abbildung 4.11 zeigen, dass große Dateien die von Variability-aware Code-Smells betroffen sind, fehleranfälliger sind als große Dateien, die von keinem Code-Smell betroffen sind. Die Nullhypothese H_{03} kann nur in 30% der Fällen verworfen werden. Das bedeutet, dass in den restlichen Fällen keine statistisch belegte Aussage darüber getroffen werden kann, ob Smell-behaftete große Dateien fehleranfälliger sind als nicht Smell-behaftete. Jedoch bieten die weiteren Daten, wie die häufigen ORs > 1 (Abbildung 4.10) und die Verhältnisse der gesamten Bugfixes (Abbildung 4.11) deutliche Indizien für unsere Vermutung, dass Smell-behaftete große Dateien fehleranfälliger sind als nicht Smell-behaftete große Dateien. Weiterhin haben wir ermittelt, dass Smell-behaftete Dateien generell größer sind als nicht Smell-behaftete Dateien. Dies ist ebenfalls, mit zwei Ausnahmen, der Fall für Dateien die nur jeweils von einem der Smells betroffen sind (siehe Tabelle 4.4). Wir schlussfolgern daher, dass die Größe allein nicht verantwortlich für eine höhere Fehleranfälligkeit ist. Der Zusammenhang zwischen Code-Smells und der Fehleranfälligkeit ist daher zumindest teilweise unabhängig von der Dateigröße.

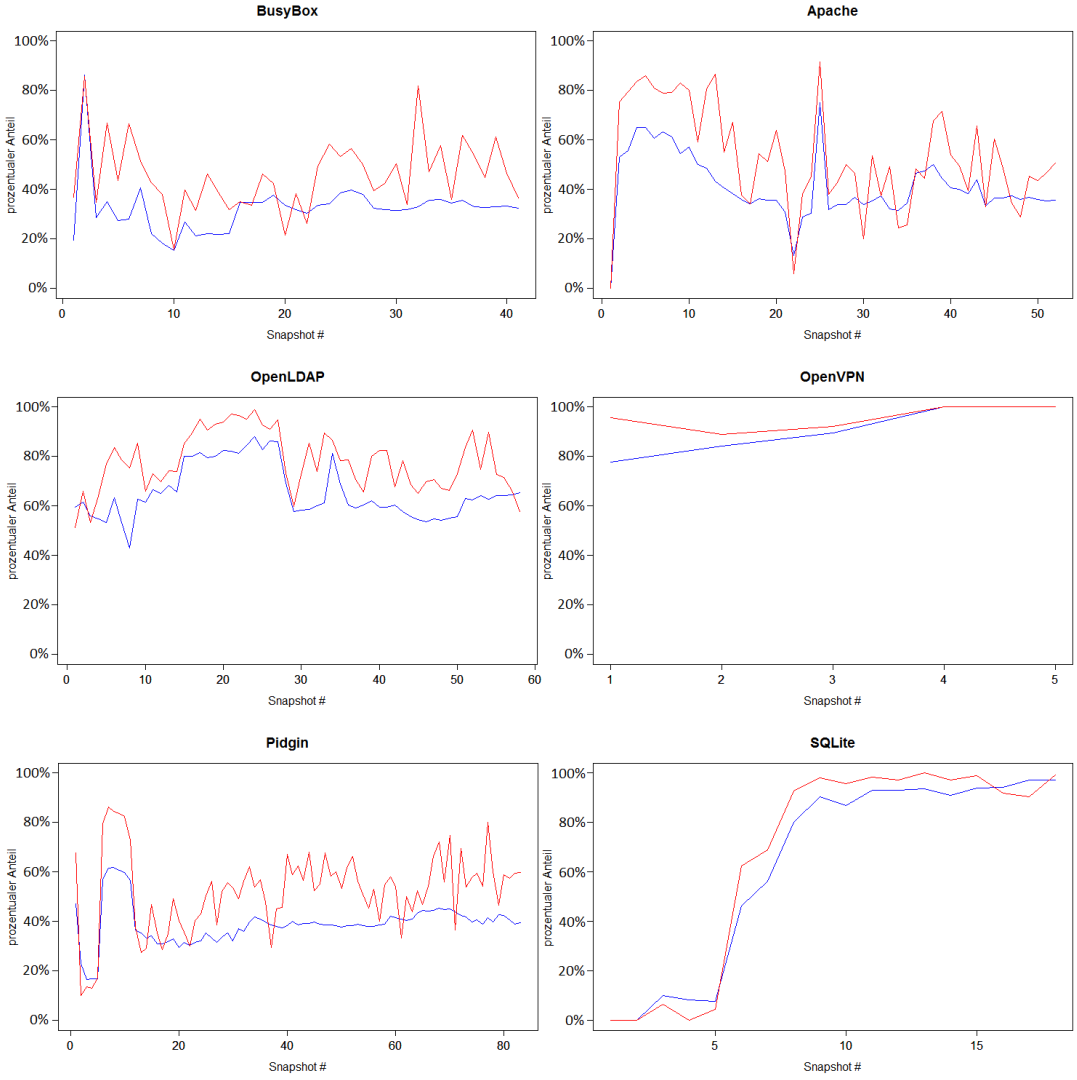


Abbildung 4.11: Übersicht über die Anteile der Smell-behafteten Dateien (blau) und der Bugfixes in Smell-behafteten Dateien (rot) aller Beispielsysteme (nur große Dateien)

4.4 Diskussion

Wir haben in diesem Kapitel eine Studie zum Einfluss von den Variability-aware Code-Smells Annotation Bundle, Annotation File und Large Feature auf die Fehleranfälligkeit durchgeführt. In dieser Sektion diskutieren wir die gefundenen Ergebnisse in Bezug auf die in Kapitel 1 vorgestellten Forschungsfragen. Dabei diskutieren wir, ob Variability-aware Code-Smells generell einen Einfluss auf die Fehleranfälligkeit haben, oder ob sie eventuell nur spezifische Muster der Verwendung von Präprozessor-Annotationen sind, welche keine negativen Auswirkungen auf die Software-Entwicklung haben. Weiterhin diskutieren wir den Einfluss einzelner Smells auf die Fehleranfälligkeit und ob der Zusammenhang zwischen Code-Smells und Fehleranfälligkeit unabhängig der Dateigröße bestehen bleibt.

RQ 1: Gibt es einen Zusammenhang zwischen Code-Smells und der Fehleranfälligkeit?

Ja, es gibt einen Zusammenhang zwischen Variability-aware Code-Smells und der Fehleranfälligkeit. In 239 von 257 (93%) aller Snapshots zeigen die Ergebnisse, dass Smell-behaftete Dateien häufiger fehlerbehebend geändert werden als nicht Smell-behaftete Dateien. Von diesen Ergebnissen sind lediglich 44 nicht signifikant. Somit kann in 195 von 257, also mehr als 75% aller Fälle, die Nullhypothese H_{01} abgelehnt werden. Das heißt, dass in diesen Snapshots signifikant mehr Fehler in Smell-behafteten Dateien als in nicht Smell-behafteten Dateien behoben werden. Die Ergebnisse variieren über die untersuchten Systeme, sind jedoch für die drei Systeme Apache, OpenLDAP und Pidgin ähnlich. Diese drei Systeme sind die drei größten Systeme, was Anzahl der Bugfix-Commits betrifft, und drei der vier größten Systeme, was die LOC betrifft.

In Abbildung 4.4 und Abbildung 4.6 ist zu erkennen, dass die Systeme OpenVPN und SQLite einen besonders hohen Anteil an Smell-behafteten Dateien besitzen. Im Schnitt haben diese Systeme 68,1% (OpenVPN) und 48,2% (SQLite) von Variability-aware Code-Smell betroffene Dateien. Dies liegt vor allem an dem Smell Large Feature der in diesen Systemen mit 65,1% (OpenVPN) und 44,2% (SQLite) annähernd für alle Smell-behafteten Dateien verantwortlich ist. Dieser mit den anderen Systemen verglichenen ungewöhnlich hohe Anteil an betroffenen Dateien lässt vermuten, dass die gewählten Schwellwerte für Large Feature für diese beiden Systeme nicht passend waren. Wir haben die über sieben Testsysteme empirisch optimierten Schwellwerte von Daniel Meyer [Mey15] für all unsere Untersuchungen übernommen. Jedoch argumentieren Moha et al. in ihrer Arbeit [MGMD08], dass Smell-Erkennung an Kontext und Charakteristika des jeweiligen untersuchten Systems angepasst werden sollte. In dem System BusyBox gibt es einen sprunghaften Anstieg von Smell-behafteten Dateien nach dem 15. Snapshot. Der Anteil an Smell-behafteten Dateien ist danach zwar nicht so hoch wie bei OpenVPN und SQLite jedoch ist in Abbildung 4.5 zu sehen, dass die Odds Ratios nach diesem 15. Snapshot deutlich abfallen. Das Phänomen von geringen Odds Ratios bei hohem Smell-Anteil ist schwierig zu erklären, da wir dies, außer bei zwei weiteren Snapshots in dem System OpenVPN, nicht noch einmal beobachten konnten.

RQ 2: Gibt es einen Zusammenhang zwischen bestimmten Arten von Code-Smells und Fehleranfälligkeit?

Ja, es gibt einen Zusammenhang zwischen den einzelnen Arten von Variability-aware Code-Smells und der Fehleranfälligkeit. Die gefundenen Ergebnisse deuten daraufhin, dass insbesondere der Code-Smell Large Feature einen Einfluss auf die Fehleranfälligkeit hat. Sieht man von dem System BusyBox ab, zeigen die Ergebnisse aus Abbildung 4.8 für Large Feature in 205 von 216 (94,9%) Snapshots, dass Dateien, die von Large Feature betroffen sind, häufiger fehlerbehebend geändert werden als die, die nicht betroffen sind. Verglichen mit dem Gesamtergebnis (208 von 216 Snapshots), sieht man, dass der Einfluss von Large Feature nicht von der Hand zu weisen ist. Wenn wir die signifikanten Werte zur Ablehnung der Nullhypothese H_{02} betrachten, sehen wir, dass wir sie für Large Feature in 70% der Fällen ablehnen können. Während wir H_{02} für Annotation Bundle in lediglich

49,4% und für Annotation File in 31,5% der Snapshots ablehnen können. Die geringe Anzahl an signifikanten Ergebnissen für die Annotation Smells in den Systemen Apache, OpenLDAP und Pidgin ist unter anderem mit den geringen Mengen an betroffenen Dateien und somit den kleinen Testmengen zu begründen. In Tabelle 4.3 sieht man, dass diese drei Systeme sowohl bei Annotation Bundle, als auch bei Annotation File die drei Systeme mit den durchschnittlich geringsten Mengen an betroffenen Dateien sind. Das System BusyBox ist auch in diesem Fall wieder eine Ausnahme. Es hat eine ähnlich geringe Menge an Smell-behafteten Dateien, jedoch deutlich mehr signfikante Ergebnisse in den Annotation Smells. Durch den geringen Anteil und der schlechteren Ergebnisse des exakten Tests nach Fisher schlussfolgern wir, dass die Variability-aware Code-Smells Annotation Bundle und Annotation File einen deutlich geringeren Einfluss haben als das Large Feature.

Obwohl Smell-behaftete Dateien fehleranfälliger sind als nicht Smell-behaftete Dateien, gibt es Situationen in denen ein Smell der beste Weg sein könnte um Anforderungen oder Funktionalität variabel implementieren zu können. Beispielsweise besitzen mehrere Systeme (beispielsweise OpenVPN und Pidgin) ein Large Feature mit dem Namen WIN32, welche Betriebssystem-spezifische Funktionalität implementiert. Diese Features sind zwar den Schwellwerten gemäß Large Feature, bieten jedoch leicht zu implementierenden Zugang zu Betriebssystem-spezifischen Funktionen. Sie führen beispielsweise unterschiedliche Dateipfade ein oder kontrollieren die unterschiedliche Thread-Behandlung für jedes Betriebsystem. Die Implementation dieser Features kann somit die Portierung von Software-Systemen auf andere Betriebssysteme erleichtern.

RQ 3: Ist der Zusammenhang zwischen Code-Smells und der Fehleranfälligkeit unabhängig der Dateigröße?

Ja, der Zusammenhang zwischen Variability-aware Code-Smells und der Fehleranfälligkeit existiert auch dann, wenn man den Einfluss der Dateigröße mit in Betracht zieht. Sowohl große Dateien, als auch die Gesamtheit der Dateien zeigen das gleiche Merkmal von fehleranfälligeren Dateien, wenn diese von Code-Smells betroffen sind. Desweiteren haben wir feststellen können, dass Smell-behaftete Dateien größer als nicht Smell-behaftete Dateien sind. Der Einfluss von Variability-aware Code-Smells ist in großen Dateien nicht mehr so groß wie in der Gesamtheit der Dateien, wie man in Abbildung 4.2 und Abbildung 4.10 sieht. Das sieht man daran, dass der Anteil an Snapshots deren Ergebnisse eine höhere Fehleranfälligkeit indizieren, von 93% (Gesamt) auf 79% (große Dateien) sinkt. Die Anzahl der signifikanten Ergebnisse ist noch deutlicher gesunken. Während wir für RQ 1 die Nullhypothese noch in 75,9% aller Fälle ablehnen konnten, können wir H_{03} nur noch in 30% aller Snapshots verwerfen. Dies ist jedoch unter anderem auf die geringe Anzahl an untersuchten Dateien zurückzuführen. Die weiteren Ergbnisse aus Abschnitt 4.3.3 deuten jedoch an, dass die Fehleranfälligkeit von Smell-behafteten großen Dateien höher ist als in nicht Smell-behafteten großen Dateien. Die Größe allein ist somit nicht verantwortlich für eine höhere Fehleranfälligkeit. Dies bestätigt unsere Vermutung, dass der Zusammenhang von Variability-aware Code-Smells und der Fehleranfälligkeit zumindest teilweise unabhängig von der Dateigröße ist.

4.5 Threats to Validity

Während alle Untersuchungen sorgfältig durchgeführt und die Methodiken gewissenhaft gewählt wurden, bleiben gewisse Risiken bestehen. Mit unserer Studie zeigen wir, dass Variability-aware Code-Smells die Fehleranfälligkeit von betroffenen Dateien beeinflussen. Wir können jedoch nicht behaupten, dass Code-Smells Fehler, beziehungsweise Bugs, verursachen. Tatsächlich kann die Studie keine Aussage darüber treffen was die Gründe dafür sind, dass Dateien von Code-Smells betroffen sind und dafür, dass Fehler in diesen Dateien auftreten. Sie zeigt lediglich, dass Smell-behaftete Dateien fehleranfälliger sind als andere Dateien. Um die Variability-aware Code-Smells detektieren zu können, wird der Prototyp SKUNK benutzt. SKUNK und die genutzten Schwellwerte zur Smell-Erkennung bergen dabei gewisse Risiken entweder Smell-behaftete Stellen nicht zu erkennen oder nicht Smell-behaftete Stellen fälschlicherweise als problematisch zu kennzeichnen. Idealerweise sollte Smell-Erkennung eine manuelle Validierung durchgeführt werden. Dies ist jedoch aufgrund der Menge an untersuchten Systemen und Snapshots in keinem akzeptablem Zeitrahmen durchführbar. Um diese fehlerhaften Kennzeichnungen zu minimieren greifen wir bei der Smell-Untersuchung auf in der vorrangegangenen Arbeit von Meyer optimierte Parameter zurück [Mey15].

Verschiedene Entwickler folgen verschiedenen Mustern beim „commiten“ ihrer Änderungen und Bugfixes und beim dokumentieren derer. Daher besteht bei der textbasierten Suche nach Bugfix-Commits das Risiko Bugfixes zu übersehen oder fälschlicherweise als Bugfix zu klassifizieren. Wie bereits bei der Smell-Erkennung sollte idealerweise auch hier eine manuelle Validierung der gefundenen Bugfixes erfolgen. Das Problem der Durchführbarkeit besteht jedoch auch hier. Für die textbasierte Suche von Bugfixes nutzen wir die von Pan et al. [PKWJ09] vorgeschlagenen Suchterme, um fehlerhafte Klassifizierungen zu minimieren.

Ein weiterer Punkt, der unsere Ergebnisse beeinflussen kann, sind die gewählten Tests, da statistische Tests möglicherweise nicht robust auf Eigenarten der Testmenge reagieren. Beispielsweise können zu kleine Testmengen oder ähnliches statistische Tests beeinflussen. Wir haben darauf geachtet, dass keine Anforderungen an die statistischen Tests verletzt werden, indem möglichst immer parameterfreie Tests benutzt wurden, die von vornherein keine Anforderungen an die Testobjekte stellen. Für OpenVPN sind wir uns bewusst, dass die Untersuchung lediglich auf 5 Snapshots durchgeführt wurde, und es daher schwierig ist starke Schlussfolgerungen für dieses System und den Einfluss von Variability-aware Code-Smells zu ziehen.

Unsere Erkenntnisse lassen sich nicht ohne weiteres auf andere Systeme und Hostsprachen übertragen, da unter anderem für andere Sprachen und Variabilitätsmechanismen die Smell-Definitionen und die Detektion dieser fehlen. Um auszuschließen, dass die Resultate der Untersuchung von einer Domäne verzerrt werden, haben wir Systeme von verschiedener Größe und aus unterschiedlichen Domänen untersucht. Somit können die Ergebnisse zumindest auf ähnliche Domänen und Systemgrößen generalisiert werden. Um die Ergebnisse noch weiter verallgemeinern zu können, müssen Untersuchungen auf weiteren Systemen und Sprachen durchgeführt werden.

4.6 Zusammenfassung

In diesem Kapitel haben wir sechs hochkonfigurierbare Software-Systeme mit insgesamt 257 Snapshots basierend auf dem vorgestellten Konzept untersucht, und haben unsere gestellten Forschungsfragen beantwortet. Wir haben gezeigt, dass Dateien die von Variability-aware Code-Smells betroffen sind fehleranfälliger sind als nicht Smell-behaftete Dateien. Dies haben wir vor der Untersuchung vermutet und konnten es durch die Ablehnung der Nullhypothese H_{01} empirisch belegen. Die Untersuchungen zur zweiten Forschungsfrage haben gezeigt, dass einzelne Code-Smells ebenfalls einen Einfluss auf die Softwareentwicklung haben. Über die sechs Systeme hinweg konnten wir für den Smell Large Feature die Nullhypothese H_{02} in 70% der Snapshots ablehnen, während dies bei den anderen beiden Smells in weniger als 50% der Fall ist. Diese Ergebnisse belegen, dass der Smell Large Feature einen größeren, negativen Einfluss auf die Fehleranfälligkeit von Dateien hat. Weiterhin haben wir gezeigt, dass Smell-behaftete Dateien größer sind als nicht Smell-behaftete Dateien. Mit unseren Untersuchungen konnten wir bestätigen, dass der Effekt der höheren Fehleranfälligkeit in Smell-behafteten Dateien auch in großen Dateien bestehen bleibt.

5. Verwandte Arbeiten

In dieser Arbeit wurde das Ziel verfolgt, den Zusammenhang zwischen Variability-aware Code-Smells und der Fehleranfälligkeit zu zeigen. Ein ähnliches Ziel verfolgte die Arbeit von Khomh et al. [KDPGA12]. Ihre Studie überprüfte und zeigte, dass Code-Smell-behaftete Klassen im OOP-Bereich häufiger geändert werden und fehleranfälliger sind als andere Klassen. Der Unterschied zu unserer Arbeit ist, dass sich ihre Untersuchungen dabei auf den Objekt-orientierten Bereich und Objekt-orientierte Code-Smells ohne jegliche Variabilität in den Systemen bezogen. Weiterhin untersuchten sie zusätzlich zur Fehleranfälligkeit auch die Änderungshäufigkeit.

Für hochkonfigurierbare Software-Systeme haben bereits Apel et al. 14 sogenannte „*variability smells*“ präsentiert [ABKS13]. Weitere Arbeiten beschäftigen sich ebenfalls mit variabilitätsbezogenen Problemen, abseits von expliziten Code-Smells [KMPY06, Kru07]. Im Gegensatz zu den von uns untersuchten Variability-aware Code-Smells, sind diese Smells und Probleme jedoch nicht auf Probleme im Quellcode beschränkt. Stattdessen betreffen sie auch andere Abschnitte der Entwicklung hochkonfigurierbarer Software-Systeme, wie beispielsweise unüberschaubare Feature-Modelle oder ungenutzte Variabilität.

Abilio et al. haben drei Code-Smells für Feature-orientierte Software-Systeme, inklusive Metriken zur Detektion, vorgestellt [APFC15]. Wie die von uns untersuchten Smells sind auch diese von den bekannten Objekt-orientierten Code-Smells abgeleitet. Die angesprochenen Probleme, wie Code Tangling, treten auch in Präprozessor-basierten Systemen auf. Die vorgeschlagenen Smells sind jedoch nicht für die in unserer Studie untersuchten Implementationsmechanismen entworfen. Aus diesem Grund finden sie hier keine weitere Verwendung.

Mehrere Arbeiten haben sich mit Problemen des C-Präprozessors befasst. Sie diskutieren den negativen Einfluss von CPP-Direktiven unter anderem auf Verständlichkeit und Fehleranfälligkeit in hochkonfigurierbaren Software-Systemen [SC92, EBN02, SLSA13]. Auch Liebig et al. präsentieren ihre Ergebnisse zur Nutzung von #ifdef-Direktiven in Bezug auf problematische Metriken, wie Nesting-Tiefe, Codezeilen und ähnliches [LAL+10]. Während diese Arbeiten sich zwar mit Problemen

des C-Präprozessors auf die Software-Entwicklung beschäftigen liefern sie keine empirischen Beweise zwischen expliziten Mustern im Quellcode (Code-Smells) und der Fehleranfälligkeit.

Metriken als Qualitätsindikator von Software-Systemen ist das Thema mehrerer Studien [BBM96, CS00]. Oft wird eine große Zahl an Codezeilen als starker Einfluss auf die Fehleranfälligkeit von Systemen diskutiert [KZL07, EEBGR01]. In unserer Studie untersuchen wir statt Metriken jedoch den Einfluss von Variability-aware Code-Smells.

6. Zusammenfassung und Zukünftige Arbeiten

In der Entwicklung von hochkonfigurierbaren Software-Systemen spielt die Variabilität eine wichtige Rolle. Sie macht eine schnelle und verlässliche Entwicklung möglich, da sie die Wiederverwertung von existierenden und getesteten Codesegmenten unterstützt. Bedingte Kompilation durch C-Präprozessor-Annotationen ist ein gebräuchlicher Mechanismus, um Variabilität im Quellcode eines hochkonfigurierbaren Software-Systems zu integrieren. Durch die steigende Bedeutung von hochkonfigurierbaren Software-Systemen, wird es auch wichtiger den Einfluss von Präprozessor-Direktiven auf die Software-Qualität zu untersuchen. Insbesondere der unsachgemäße Gebrauch von Präprozessor-Annotationen kann zu Variability-aware Code-Smells führen. Es wird davon ausgegangen, dass diese Muster im Quellcode einen negativen Einfluss auf Verständlichkeit und Wartbarkeit des Codes haben [Mey15, FSMS15, FS15]. Jedoch fehlen empirische Beweise um diese Thesen zu untermauern.

In dieser Arbeit haben wir zunächst ein Konzept entwickelt um den Einfluss von Variability-aware Code-Smells auf Bereiche der Software-Entwicklung, insbesondere auf die Fehleranfälligkeit, zu untersuchen. Dabei setzen wir auf den Prototypen SKUNK zur Detektion der Variability-aware Code-Smells in C-Präprozessor-basierten Systemen. Weiterhin nutzen wir einen textbasierten Ansatz um Bugfix-Commits zu klassifizieren, aus denen wir die Fehler in den Systemen abstrahieren. Wir haben in unserer Studie die Variability-aware Code-Smells Annotation Bundle, Annotation File und Large Feature untersucht. Die Studie wurde auf insgesamt 257 Snapshots aus den sechs hochkonfigurierbaren Software-Systemen BusyBox, Apache, OpenLDAP, OpenVPN, Pidgin und SQLite durchgeführt. Mithilfe des exakten Tests nach Fisher haben wir die korrelierten Daten anschließend in einer Studie ausgewertet um die aufgestellten Forschungsfragen zu beantworten.

Die erste Forschungsfrage soll klären ob es einen Zusammenhang zwischen Variability-aware Code-Smells und der Fehleranfälligkeit gibt. Wir konnten die mit RQ 1 zusammenhängende Nullhypothese H_{01} in mehr als 75% aller Snapshots ablehnen,

was unsere Vermutung unterstützt. So konnten wir zeigen, dass Dateien, die von mindestens einem der drei untersuchten Smells betroffen sind, signifikant häufiger fehlerbehebend geändert werden als andere Dateien. In den drei größten Systemen Apache, OpenLDAP und Pidgin konnten wir sogar jeweils in über 80% der Fälle H_{01} ablehnen, während dies in den Systemen BusyBox und SQLite in weniger als 50% der Fälle möglich war. Mit der zweiten Forschungsfrage, die sich mit dem Einfluss einzelner Variability-aware Code-Smells beschäftigt, konnten wir zeigen, dass der spezifische Code-Smell Large Feature einen größeren Einfluss auf die Software Evolution hat als die anderen beiden Smells. Dies belegen wir zum einen mit der deutlich höheren Ablehnrate der Nullhypothese H_{02} (Large Feature 70%, Annotation Bundle 49,4%, Annotation File 31,5%) und zum anderen mit der deutlich größeren Anzahl an von Large Feature betroffenen Dateien. Die dritte Forschungsfrage untersucht, ob der Zusammenhang aus RQ 1 unabhängig von der Größe von Dateien besteht. Wir haben aufgezeigt, dass Dateigröße allein nicht verantwortlich für eine höhere Fehleranfälligkeit ist und Variability-aware Code-Smells dem Entwickler somit zusätzliche und ergänzende Informationen liefern.

Die Studie unterstützt innerhalb der Grenzen der Validität die Vermutung aus der Literatur [Mey15, FSMS15, FS15], dass Code-Smells einen negativen Einfluss auf die Software-Entwicklung haben. Die Studie zeigt Entwicklern ebenfalls, dass sie Aufmerksamkeit auf Systeme mit einer hohen Anzahl an Smell-behafteten Dateien legen sollten, da diese Dateien eher Fehler beinhalten.

Zukünftige Arbeiten

Wir sind daran interessiert die Forschung in andere Richtungen fortzuführen. Weitere Arbeiten können den Zusammenhang zwischen Code-Smells und anderen Charakteristika und Metriken von hochkonfigurierbaren Software-Systemen und ihren Einfluss auf die Software-Entwicklung untersuchen. So ist zusätzlich zur Fehlerhäufigkeit auch die generelle Änderungshäufigkeit und die Art der Änderungen für die Forschung von Interesse.

Während wir die erhöhte Fehleranfälligkeit von Smell-behafteten Dateien in hochkonfigurierbaren Software-Systemen gezeigt haben, ist die Evaluierung lediglich auf sechs Open-Source-Systemen durchgeführt worden. In zukünftigen Forschungsarbeiten muss die Evaluierung auf eine größere Auswahl an Systemen ausgeweitet werden. Interessant wären auch Untersuchungen von Systemen aus der Industrie, da diese anderen Entwicklungszyklen unterliegen können. Weiterhin standen zum Zeitpunkt der Durchführung der Studie nur drei untersuchbare Variability-aware Code-Smells zur Verfügung. Es empfiehlt sich daher, dass zukünftige Arbeiten auf den Bereich der Variability-aware Code-Smells konzentrieren. Zum einen müssen neue Variability-aware Code-Smells definiert werden, zum anderen sollte die durchgeführte Studie um weitere Smells erweitert werden. Der benutzte Prototyp zur automatischen Erkennung von Variability-aware Code-Smells sollte in weiteren Untersuchungen noch verbessert oder durch andere Smell-Erkennungstools ersetzt werden. Auch die Bugfix-Klassifizierung kann noch erweitert und verbessert werden. Eine Möglichkeit wäre beispielsweise Issue-Datenbanken wie BugZilla mit einzubeziehen um genauere Bugfix-Klassifizierungen vornehmen zu können. Durch eine genauere Smell- und Bugfix-Erkennung erhoffen wir uns genauere und noch bessere Ergebnisse, die unsere Vermutungen unterstützen.

In der durchgeführten Studie wurden aufgrund fehlender Detektionsalternativen lediglich C-Präprozessor-basierte Systeme, die in der Hostsprache C geschrieben sind, durchgeführt. In zukünftigen Arbeiten sollten weitere Hostsprachen berücksichtigt werden. Weiterhin sollten auch andere Implementationsansätze für hochkonfigurierbare Software-Systeme, beispielsweise Feature- oder Aspekt-orientierte Programmierung, auf Variability-aware Code-Smells und deren Einfluss auf diese Art Systeme untersucht werden. Somit wäre eine Generalisierung der Ergebnisse auch auf weitere Systeme, Implementationsmechanismen und Hostsprachen möglich.

A. Anhang

A.1 Detaillierte Daten für RQ 1 und RQ 3

In den folgenden Tabellen werden für jedes untersuchte Software-System und jeden Snapshot sowohl der p-Wert und die zugehörige OR dargestellt.

Busybox								
#	P-Wert	Odds Ratio	#	P-Wert	Odds Ratio	#	P-Wert	Odds Ratio
1	0,27	Inf	15	0,03	2,18	29	0,71	0,88
2	0,10	1,97	16	0,87	0,75	30	0,13	1,43
3	0,71	0,88	17	0,83	0,79	31	0,75	0,86
4	0,00	29,17	18	0,00	3,24	32	0,01	2,38
5	0,00	7,20	19	0,56	1,00	33	0,51	1,02
6	0,00	9,26	20	0,95	0,65	34	0,01	1,88
7	0,00	10,37	21	0,21	1,26	35	0,46	1,05
8	0,00	6,18	22	0,95	0,67	36	0,08	1,53
9	0,01	3,91	23	0,58	0,98	37	0,32	1,20
10	0,03	2,87	24	0,09	1,50	38	0,50	1,03
11	0,00	3,26	25	0,29	1,18	39	0,01	2,01
12	0,05	2,20	26	0,02	1,68	40	0,01	1,93
13	0,00	3,48	27	0,74	0,88	41	0,06	1,57
14	0,00	3,77	28	0,18	1,30			

Tabelle A.1: Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System BusyBox (RQ1)

Apache								
#	P-Wert	Odds Ratio	#	P-Wert	Odds Ratio	#	P-Wert	Odds Ratio
1	0,55	Inf	19	0,02	2,43	37	0,00	2,91
2	0,00	11,87	20	0,02	2,51	38	0,00	5,25
3	0,04	3,00	21	0,02	2,41	39	0,00	3,34
4	0,12	2,08	22	0,94	0,46	40	0,00	5,29
5	0,00	7,82	23	0,04	2,23	41	0,00	3,13
6	0,03	2,39	24	0,00	3,72	42	0,00	3,16
7	0,02	2,48	25	0,05	3,02	43	0,00	3,92
8	0,00	3,05	26	0,01	2,87	44	0,01	2,47
9	0,00	4,31	27	0,00	3,35	45	0,00	5,06
10	0,00	10,53	28	0,00	5,35	46	0,00	3,49
11	0,00	6,49	29	0,02	3,10	47	0,00	4,36
12	0,00	10,35	30	0,02	2,30	48	0,14	1,55
13	0,00	8,08	31	0,00	3,48	49	0,00	3,53
14	0,00	4,29	32	0,02	2,31	50	0,00	3,90
15	0,00	4,01	33	0,01	2,55	51	0,00	3,20
16	0,01	3,31	34	0,25	1,46	52	0,18	1,50
17	0,32	1,32	35	0,02	2,18			
18	0,04	2,12	36	0,05	1,87			

Tabelle A.2: Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System Apache (RQ1)

OpenLDAP											
#	P-Wert	Odds Ratio	#	P-Wert	OR	#	P-Wert	OR	#	P-Wert	OR
1	0,00	2,92	16	0,00	9,92	31	0,00	2,22	46	0,00	2,85
2	0,00	4,56	17	0,00	2,32	32	0,00	3,75	47	0,00	4,04
3	0,13	1,45	18	0,00	4,55	33	0,00	5,13	48	0,00	2,90
4	0,00	2,30	19	0,00	3,64	34	0,01	2,18	49	0,00	5,22
5	0,00	2,25	20	0,00	7,07	35	0,00	3,07	50	0,00	7,99
6	0,00	7,83	21	0,00	8,89	36	0,00	1,99	51	0,00	4,21
7	0,00	3,89	22	0,00	2,94	37	0,00	2,28	52	0,00	5,10
8	0,00	4,06	23	0,00	2,12	38	0,00	2,48	53	0,00	4,58
9	0,04	1,62	24	0,00	3,21	39	0,00	2,68	54	0,00	3,47
10	0,00	2,83	25	0,00	3,03	40	0,00	4,20	55	0,00	4,55
11	0,00	2,76	26	0,01	1,93	41	0,00	2,78	56	0,11	1,56
12	0,00	2,34	27	0,00	4,04	42	0,00	4,58	57	0,03	2,25
13	0,00	2,16	28	0,00	1,89	43	0,00	2,25	58	0,23	1,32
14	0,00	1,90	29	0,13	1,37	44	0,00	4,38			
15	0,87	0,80	30	0,00	2,61	45	0,00	2,41			

Tabelle A.3: Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System OpenLDAP (RQ1)

OpenVPN		
#	P-Wert	Odds Ratio
1	0,01	3,62
2	0,00	7,10
3	0,00	5,23
4	0,75	0,84
5	0,44	2,13

Tabelle A.4: Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System OpenVPN (RQ1)

Pidgin										
#	P-Wert	Odds Ratio	#	P-Wert	OR	#	P-Wert	OR	#	P-Wert	OR
1	0,37	1,37	22	0,22	1,34	43	0,00	4,44	64	0,00	3,74
2	0,82	0,72	23	0,00	5,37	44	0,00	3,59	65	0,00	2,24
3	0,00	2,99	24	0,00	3,32	45	0,00	4,86	66	0,00	2,64
4	0,13	2,02	25	0,01	2,28	46	0,00	2,94	67	0,00	3,29
5	0,51	1,11	26	0,00	5,13	47	0,00	3,93	68	0,00	4,24
6	0,00	2,84	27	0,00	3,00	48	0,00	4,33	69	0,01	1,77
7	0,00	7,83	28	0,00	3,21	49	0,00	2,92	70	0,00	6,65
8	0,00	7,58	29	0,00	2,31	50	0,00	3,60	71	0,04	1,70
9	0,00	5,62	30	0,00	4,79	51	0,00	4,47	72	0,00	3,20
10	0,00	7,41	31	0,00	5,54	52	0,00	4,10	73	0,00	2,99
11	0,00	3,51	32	0,00	6,09	53	0,00	6,19	74	0,00	4,32
12	0,09	1,67	33	0,00	2,88	54	0,00	3,34	75	0,00	4,34
13	0,23	1,34	34	0,00	2,50	55	0,00	2,37	76	0,00	3,76
14	0,27	1,30	35	0,00	3,40	56	0,00	3,72	77	0,00	6,42
15	0,00	3,67	36	0,00	2,62	57	0,00	2,36	78	0,00	5,02
16	0,15	1,49	37	0,49	1,08	58	0,00	2,76	79	0,00	2,10
17	0,13	1,59	38	0,00	2,48	59	0,00	2,98	80	0,00	3,50
18	0,14	1,49	39	0,00	4,06	60	0,00	4,15	81	0,00	2,75
19	0,00	2,60	40	0,00	5,06	61	0,00	2,29	82	0,00	3,99
20	0,01	2,30	41	0,02	1,88	62	0,00	3,24	83	0,00	3,15
21	0,00	2,73	42	0,00	3,60	63	0,00	2,79			

Tabelle A.5: Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System Pidgin (RQ1)

SQLite					
#	P-Wert	Odds Ratio	#	P-Wert	Odds Ratio
1	1,00	0,00	10	0,00	4,88
2	0,94	0,36	11	0,08	1,92
3	0,44	Inf	12	0,14	1,66
4	0,98	0,18	13	0,00	10,49
5	0,90	0,49	14	0,06	2,12
6	0,06	6,07	15	0,03	2,58
7	0,01	4,62	16	0,00	6,33
8	0,01	3,59	17	0,01	3,72
9	0,00	14,30	18	0,10	2,04

Tabelle A.6: Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System SQLite (RQ1)

Busybox								
#	P-Wert	Odds Ratio	#	P-Wert	Odds Ratio	#	P-Wert	Odds Ratio
1	0,51	Inf	15	0,39	1,30	29	0,83	0,75
2	0,91	0,40	16	0,89	0,67	30	0,19	1,45
3	0,99	0,28	17	0,93	0,61	31	0,90	0,68
4	0,00	Inf	18	0,06	1,95	32	0,07	2,10
5	0,22	3,41	19	0,81	0,76	33	0,64	0,94
6	0,03	3,44	20	0,84	0,73	34	0,03	2,05
7	0,01	4,13	21	0,16	1,56	35	0,96	0,60
8	0,02	8,97	22	0,97	0,52	36	0,22	1,39
9	0,12	2,55	23	0,20	1,46	37	0,46	1,11
10	0,26	1,72	24	0,34	1,25	38	0,48	1,08
11	0,02	2,87	25	0,30	1,28	39	0,02	2,18
12	0,26	1,55	26	0,13	1,56	40	0,08	1,67
13	0,11	2,06	27	0,63	0,94	41	0,37	1,19
14	0,04	2,61	28	0,66	0,92			

Tabelle A.7: Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System BusyBox (RQ3)

Apache								
#	P-Wert	Odds Ratio	#	P-Wert	Odds Ratio	#	P-Wert	Odds Ratio
1	1,00	0,00	19	0,16	2,31	37	0,74	0,83
2	0,21	Inf	20	0,45	1,34	38	0,03	2,94
3	0,44	Inf	21	0,01	11,63	39	0,02	3,30
4	0,11	Inf	22	0,93	0,39	40	0,02	3,18
5	0,11	Inf	23	0,23	2,02	41	0,32	1,45
6	0,15	5,61	24	0,09	2,93	42	0,49	1,15
7	0,62	1,18	25	0,27	3,19	43	0,01	3,97
8	0,04	4,94	26	0,70	0,88	44	0,60	1,00
9	0,09	3,31	27	0,65	0,95	45	0,03	3,21
10	0,00	Inf	28	0,06	3,18	46	0,03	3,07
11	0,50	1,45	29	0,39	2,48	47	0,24	1,65
12	0,01	12,34	30	0,79	0,74	48	0,56	1,06
13	0,02	5,22	31	0,57	1,06	49	0,58	1,04
14	0,40	1,55	32	0,27	1,64	50	0,13	1,97
15	0,10	3,77	33	0,02	3,76	51	0,05	2,71
16	0,27	1,93	34	0,80	0,72	52	0,45	1,21
17	0,86	0,62	35	0,62	0,99			
18	0,28	2,02	36	0,60	1,01			

Tabelle A.8: Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System Apache (RQ3)

OpenLDAP											
#	P-Wert	Odds Ratio	#	P-Wert	OR	#	P-Wert	OR	#	P-Wert	OR
1	0,67	0,91	16	0,00	5,06	31	0,05	2,23	46	0,34	1,27
2	0,06	2,60	17	0,03	3,55	32	0,09	1,85	47	0,03	2,14
3	0,81	0,75	18	0,07	2,55	33	0,00	5,64	48	0,09	1,84
4	0,38	1,26	19	0,01	3,64	34	0,30	1,53	49	0,25	1,41
5	0,03	2,40	20	0,05	2,91	35	0,28	1,40	50	0,00	3,00
6	0,00	6,09	21	0,00	7,43	36	0,07	1,95	51	0,01	2,97
7	0,00	5,03	22	0,00	5,79	37	0,49	1,09	52	0,01	6,23
8	0,01	3,49	23	0,12	2,23	38	0,13	1,70	53	0,12	2,00
9	0,12	1,87	24	0,02	8,23	39	0,01	2,93	54	0,05	2,64
10	0,10	1,90	25	0,02	3,86	40	0,01	2,89	55	0,13	1,83
11	0,08	2,02	26	0,08	2,78	41	0,16	1,61	56	0,95	0,50
12	0,57	1,01	27	0,03	3,47	42	0,48	1,11	57	0,86	0,65
13	0,17	1,69	28	0,40	1,22	43	0,08	1,99	58	0,92	0,59
14	0,16	1,66	29	0,51	1,07	44	0,03	2,25			
15	0,71	0,86	30	0,05	2,18	45	0,41	1,18			

Tabelle A.9: Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System OpenLDAP (RQ3)

OpenVPN		
#	P-Wert	Odds Ratio
1	0,11	10,41
2	0,30	6,32
3	1,00	0,00
4	1,00	0,00
5	1,00	0,00

Tabelle A.10: Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System OpenVPN (RQ3)

Pidgin											
#	P-Wert	Odds Ratio	#	P-Wert	OR	#	P-Wert	OR	#	P-Wert	OR
1	0,53	Inf	22	0,75	0,81	43	0,06	1,95	64	0,09	1,70
2	0,67	1,07	23	0,01	5,03	44	0,04	2,16	65	0,28	1,34
3	0,35	1,97	24	0,06	2,90	45	0,07	1,85	66	0,18	1,44
4	0,90	0,51	25	0,08	2,48	46	0,15	1,67	67	0,05	1,88
5	0,55	1,33	26	0,00	4,16	47	0,01	2,90	68	0,02	2,33
6	0,19	2,20	27	0,06	2,21	48	0,03	2,24	69	0,30	1,28
7	0,00	7,66	28	0,12	1,83	49	0,11	1,67	70	0,00	4,53
8	0,05	2,99	29	0,28	1,41	50	0,05	1,98	71	0,88	0,67
9	0,02	3,76	30	0,02	2,53	51	0,00	3,22	72	0,00	3,17
10	0,00	6,94	31	0,05	2,07	52	0,06	1,96	73	0,09	1,66
11	0,02	3,53	32	0,01	2,92	53	0,00	3,12	74	0,00	3,29
12	0,53	1,12	33	0,29	1,34	54	0,22	1,41	75	0,03	2,12
13	0,71	0,87	34	0,09	1,91	55	0,52	1,05	76	0,15	1,53
14	0,87	0,64	35	0,09	1,93	56	0,10	1,66	77	0,00	3,62
15	0,18	1,89	36	0,24	1,43	57	0,68	0,90	78	0,00	2,62
16	0,91	0,55	37	0,62	0,99	58	0,26	1,38	79	0,58	0,98
17	0,68	0,90	38	0,50	1,08	59	0,03	2,22	80	0,02	2,10
18	0,03	3,00	39	0,41	1,17	60	0,04	2,00	81	0,20	1,40
19	0,37	1,34	40	0,00	3,28	61	0,69	0,89	82	0,02	2,10
20	0,57	1,05	41	0,12	1,70	62	0,26	1,34	83	0,03	1,95
21	0,74	0,82	42	0,06	1,99	63	0,07	1,91			

Tabelle A.11: Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System Pidgin (RQ3)

SQLite					
#	P-Wert	Odds Ratio	#	P-Wert	Odds Ratio
1	1	0	10	0,03	25,67
2	1	0	11	1,00	0,00
3	1	0	12	0,48	2,74
4	1	0	13	0,10	Inf
5	1	0	14	0,58	1,62
6	0,54	Inf	15	0,52	2,37
7	1,00	0,00	16	0,55	2,14
8	0,16	6,10	17	1	0
9	0,35	4,76	18	1	0

Tabelle A.12: Odds Ratios und p-Werte aus dem exakten Test nach Fisher für das System SQLite (RQ3)

Literaturverzeichnis

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Science & Business Media, 2013. (zitiert auf Seite 1, 5, 6, 7, 18 und 57)

[APFC15] Ramon Abílio, Juliana Padilha, Eduardo Figueiredo, and Heitor Costa. Detecting Code Smells in Software Product Lines – An Exploratory Study. In *Proceedigns of the International Conference on Information Technology - New Generations (ITNG)*, pages 433–438. IEEE, 2015. (zitiert auf Seite 57)

[BBM96] Viktor Basili, Lionel Briand, and Walcelio Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *Transactions on Software Engineering*, 22(10):751–761, 1996. (zitiert auf Seite 25 und 58)

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001. (zitiert auf Seite 5)

[CS00] Michelle Cartwright and Martin Shepperd. An Empirical Investigation of an Object-Oriented Software System. *Transactions on Software Engineering*, 26(8):786–796, 2000. (zitiert auf Seite 25 und 58)

[DBDV+06] Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. Does God Class Decomposition Affect Comprehensibility? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 346–355. IASTED, 2006. (zitiert auf Seite 25)

[EBN02] Michael Ernst, Greg Badros, and David Notkin. An Empirical Analysis of C Preprocessor Use. *Transactions on Software Engineering*, 28(12):1146–1170, 2002. (zitiert auf Seite 7 und 57)

[EEBGR01] Khaled El Emam, Saida Benlarbi, Nishith Goel, and Shesh Rai. The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *Transactions on Software Engineering*, 27(7):630–650, 2001. (zitiert auf Seite 24, 25 und 58)

[Eva03] William Evanco. Comments on The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *Transactions on Software Engineering*, 29(7):670–672, 2003. (zitiert auf Seite 25)

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. (zitiert auf Seite 1, 8, 19, 22 und 25)

[FS15] Wolfram Fenske and Sandro Schulze. Code Smells Revisited: A Variability Perspective. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, pages 3–10. ACM, 2015. (zitiert auf Seite 1, 5, 17, 18, 20, 59 und 60)

[FSMS15] Wolfram Fenske, Sandro Schulze, Daniel Meyer, and Gunter Saake. When Code Smells Twice as Much: Metric-Based Detection of Variability-Aware Code Smells. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 171–180. IEEE, 2015. (zitiert auf Seite 1, 2, 20, 21, 59 und 60)

[Gri00] Martin Griss. In *Proceedings of the International Conference of Software Product Lines (SPLC)*, pages 271–288. Springer, 2000. (zitiert auf Seite 6)

[HEK09] Joachim Hartung, Bärbel Elpelt, and Karl-Heinz Klösener. *Statistik: Lehr-und Handbuch der angewandten Statistik*. Walter de Gruyter, 2009. (zitiert auf Seite 13)

[KA09] Christian Kästner and Sven Apel. Virtual Separation of Concerns - A Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009. (zitiert auf Seite 7)

[KCH+90] Kyo Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (zitiert auf Seite 6)

[KDPGA12] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An Exploratory Study of the Impact of Antipatterns on Class Change-and Fault-Proneness. *Empirical Software Engineering*, 17(3):243–275, 2012. (zitiert auf Seite 1, 2, 8, 15, 25 und 57)

[KMPY06] Ronny Kolb, Dirk Muthig, Thomas Patzke, and Kazuyuki Yamauchi. Refactoring a Legacy Component for Reuse in a Software Product Line: a Case Study. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):109–132, 2006. (zitiert auf Seite 57)

[KPG09] Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An Exploratory Study of the Impact of Code Smells on Software

Change-Proneness. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 75–84. IEEE, 2009. (zitiert auf Seite 25)

[Kru07] Charles Krueger. The 3-Tiered Methodology: Pragmatic Insights from New Generation Software Product Lines. In *Proceedings of the Software Product Line Conference (SPLC)*, pages 97–106. IEEE, 2007. (zitiert auf Seite 57)

[KVGS09] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A Bayesian Approach for the Detection of Code and Design Smells. In *Proceedings of the International Conference on Quality Software (QSIC)*, pages 305–314. IEEE, 2009. (zitiert auf Seite 12)

[KZL07] Akif Günes Koru, Dongsong Zhang, and Hongfang Liu. Modeling the Effect of Size on Defect Proneness for Open-Source Software. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 115–124. IEEE, 2007. (zitiert auf Seite 25 und 58)

[LAL+10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 105–114. ACM, 2010. (zitiert auf Seite 7, 36 und 57)

[Mey15] Daniel Meyer. Metrics-Based Code Smell Detection in Highly-Configurable Software Systems. Masterarbeit, Otto-von-Guericke Universität, Germany, 2015. (zitiert auf Seite 18, 22, 23, 28, 34, 35, 52, 54, 59 und 60)

[MGDLM10] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *Transactions on Software Engineering*, 36(1):20–36, 2010. (zitiert auf Seite 1 und 11)

[MGMD08] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, and Laurence Duchien. A Domain Analysis to Specify Design Defects and Generate Detection Algorithms. In *Proceedings of the International Conference of Fundamental Approaches to Software Engineering (FASE)*, pages 276–291. ETAPS, 2008. (zitiert auf Seite 52)

[MKR+15] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The Love/Hate Relationship With the C Preprocessor: An Interview Study. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 495–518. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015. (zitiert auf Seite 7 und 18)

[Moh07] Naouel Moha. Detection and Correction of Design Defects in Object-Oriented Designs. In *Proceedings of the Companion to the Conference*

on Object-oriented Programming Systems, Languages & Applications (OOPSLA), pages 949–950. ACM, 2007. (zitiert auf Seite 1)

[MRG13] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. Investigating Preprocessor-Based Syntax Errors. In *Proceedings of the International Conference of Generative Programming: Concepts and Experiences (GPCE)*, number 3, pages 75–84. ACM, 2013. (zitiert auf Seite 7)

[MV00] Audris Mockus and Lawrence Votta. Identifying Reasons for Software Changes Using Historic Databases. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 120–130. IEEE, 2000. (zitiert auf Seite 29)

[Nor02] Linda Northrop. SEI's Software Product Line Tenets. *IEEE Software*, 19(4):32–40, 2002. (zitiert auf Seite 5)

[Pea14] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 2014. (zitiert auf Seite 12)

[PKWJ09] Kai Pan, Sunghun Kim, and James Whitehead Jr. Toward an Understanding of Bug Fix Patterns. *Empirical Software Engineering*, 14(3):286–315, 2009. (zitiert auf Seite 29 und 54)

[SC92] Henry Spencer and Geoff Collyer. # ifdef Considered Harmful, or Portability Experience With C News. In *Proceedings of Technical Conference*. USENIX, 1992. (zitiert auf Seite 7 und 57)

[SFAG+13] Francisco Zigmund Sokol, Mauricio Finavaro Aniche, Marco Gerosa, et al. MetricMiner: Supporting Researchers in Mining Software Repositories. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 142–146. IEEE, 2013. (zitiert auf Seite 28)

[SLSA13] Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment . In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pages 65–74. ACM, 2013. (zitiert auf Seite 57)

[TSFB99] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor Basili. Detecting Defects in Object-Oriented Designs: Using Reading Techniques to Increase Software Quality. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, volume 34, pages 47–56. ACM, 1999. (zitiert auf Seite 10)

[VEM02] Eva Van Emden and Leon Moonen. Java Quality Assurance by Detecting Code Smells. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 97–106. IEEE, 2002. (zitiert auf Seite 1 und 11)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 02. Mai 2016