Analyse und Konzept zur Verbesserung der statischen Fehlereingrenzung

Masterarbeit zur Erlangung des
akademischen Grades Master of Science (M.Sc.)

Autor: Lars K.W. Gohlke, Diplom-Inf. (FH)
Betreuer 1: Prof. Dr. rer. nat. Gabriele Schmidt
Betreuer 2: Dr. rer. nat. Raik Nagel
Eingereicht: April/2012

Eingereicht an der Fachhochschule Brandenburg
Fachbereich Informatik und Medien.

Inhaltsverzeichnis

Danksagung

An dieser Stelle möchte ich die Gelegenheit nutzen, auf eine von der Form abweichende Art und Weise meinen persönlichen Dank folgenden Personen auszudrücken:

- Natürlich zuerst meinem alten Arbeitgeber der Sprd.net AG.
- Weiterhin meinen Betreuern, **Fr. Prof. Dr. Gabriele Schmidt** und **Hrn. Dr. Raik Nagel**, denen ich das gute Gelingen dieser Arbeit zu verdanken habe.

Dann möchte ich mich auch bedanken bei ...

- ... **Hrn. Dr.-paed. Wolfgang Gohlke** – meinem aufopferungsvollen Großvater – als Wächter über den Kommateufel, Hüter des ordentlichen Ausdrucks und dem Austreiber des Konjunktivs.
- ... meiner liebevollen **Großmutter**, als technische Assistenz für meinen Großvater.
- ... meinen Kollegen, welche ohne Murren, regelmäßig meine zwei Tage Abwesenheit pro Woche hingenommen haben.
- ... der Firma SonarSource S.A. unter `http://www.sonarsource.com/` und der Gemeinschaft, die mit ihrem Opensource-Produkt *Sonar* ganz entscheidend die technische Grundlage für das Gelingen dieser Arbeit gelegt haben.
- ... allen, die hier unerwähnt bleiben.

Es hat Spaß gemacht ... und doch ist es schön es abgeschlossen zu haben.

1 Einleitung

„Miss alles, was sich messen lässt und mach alles messbar, was sich auch nicht messen lässt.“

Galileo Galilei, 1564-1642

Fehlersuche auf der Grundlage fehlgeschlagener Tests kann ein mühsames Unterfangen sein. Softwareentwickler kennen die Situation, in der neue Anpassungen mit vorhandenen Tests zu überprüfen sind, aber eine Menge von Fehlern hervorrufen, deren Ursache nicht auf den ersten Blick ersichtlich ist. Dadurch, dass die Tests intern verschiedene Hierarchien abdecken, um somit unterschiedliche Sichten und Schichten zu überprüfen, entsteht aus wenigen Fehlerursachen durch Fehlerfortpflanzung meist ein störendes Rauschen von wesentlich mehr fehlgeschlagenen Tests. Nun muss die Fehlereingrenzung jedoch in einem, womöglich komplexen oder gar völlig unbekannten Projekt, durchgeführt werden. Dabei ist die Frage zu beantworten, wie man den Aufwand und damit die Kosten zur Fehlereingrenzung sinnvoll und wirksam reduzieren kann?

In Zeiten, in denen zunehmend eine testgetriebene Softwareentwicklung betrieben wird, sind die nötigen Voraussetzungen[1] gegeben, um die Fehlersuche auf Tests auszuweiten und somit die notwendige Effizienz durch verfeinerte Werkzeuge in der Produktionskette[2] zu erreichen.

Die vorliegende Arbeit befasst sich deshalb mit der statischen Fehlereingrenzung durch intelligente Vorsortierung der fehlgeschlagenen Tests im

[1] Damit ist das Vorhandensein einer ausreichenden Menge von gepflegten Tests gemeint.

[2] Der Begriff „Produktion“ tangiert den Diskurs, ob Softwareentwicklung (noch) ein kreativer schöpferischer oder mittlerweile schon ein industrialisierter Prozess ist (Rei11; Rob09; Hof08, S.544f).

Softwareentwicklungsprozess und richtet sich vornehmlich an Softwareentwickler.

Betriebliches Umfeld

Diese Masterarbeit entstand im Rahmen meiner Tätigkeit bei der Sprd.net AG als Qualitätsingenieur in der Softwareentwicklung. Die Sprd.net AG ist im Bereich des Onlineversandhandels mit u.a. individuell gestaltbaren Textilien tätig. Zur Unterstützung dieser Tätigkeit sind Softwareentwicklungen für die nach außen sichtbare Handelsplattform, aber auch für die interne Produktionssteuerung erforderlich. Für weiterhin hohe Innovationszyklen in der Entwicklung der Softwareplattform ist die Qualitätssicherung zuständig. Dafür sind in dieser Arbeit weiterführende bzw. unterstützende Erkenntnisse zu gewinnen.

Aufgabenstellung

Aus den bisherigen Ausführungen ergibt sich folgende Zielstellung: Erarbeitung der Analyse und eines Konzeptes zur Verbesserung der statischen Fehlereingrenzung mit Hilfe von Softwaremetriken. Die statische Analyse der Tests soll den Fehleranalyseprozess in Bezug auf vorhandene Tests beschleunigen. Auf diese Weise wird die Werkzeugkette des Softwareentwicklungsprozesses wirksam unterstützt und effizienter gestaltet.
Dabei sind folgende Fragen zu beantworten:

- Wie kann der Fehleranalyseprozess nachhaltig und wirksam verbessert werden?
- Wie lassen sich Softwaremetriken in den Fehleranalyseprozess integrieren?
- Kann eine Verbesserung der Effizienz bei der Fehlereingrenzung beobachtet werden?

Im Rahmen dieser Arbeit wird ein Konzept ausgearbeitet, um auf diese Weise Techniken der konstruktiven Qualitätssicherung und dynamische

Verfahren sinnvoll zu verbinden. Auf der Grundlage der Ausführung von Tests werden mit Hilfe der statischen Analyse, einem Verfahren aus der konstruktiven Qualitätssicherung, die fehlgeschlagenen Tests zueinander in Beziehung gesetzt. Dadurch wird der Fehleranalyseprozess wirksam unterstützt.

So wird in der Analysephase untersucht, ob sich mit Hilfe von Softwaremetriken Merkmale zur Unterscheidung von Tests finden lassen. Darauf aufbauend, wird diese Unterscheidung zu einer Klassifikation von Tests führen, aus der sich eine Reihenfolge berechnen läßt. Diese empfohlene Reihenfolge bildet die Grundlage im Fehleranalyseprozess, um so Fehler effizienter lokalisieren zu können.

Die Reihenfolge steht *deshalb* im Mittelpunkt, weil vermutet wird, dass verschiedene Tests hinsichtlich ihrer Beziehung zur Fehlerursache unterschiedlich zu wichten sind.

Darstellung des Problems in der Literatur

In der Literatur werden zwei Vorgehensweisen auf dem Gebiet der Fehlereingrenzung beschrieben. Einerseits existiert die konstruktive Qualitätssicherung (präventiv) als eine statische Verfahrensweise. Andererseits werden dynamische Verfahren zur automatischen Isolierung von Fehlern aufgrund statistischer Berechnungen (reaktiv) eingesetzt.

Die konstruktive Qualitätssicherung beabsichtigt bereits im Vorfeld, Probleme bzw. *bad practices* grundsätzlich durch geeignete Techniken bzw. *best practices* zu vermeiden. Es werden Hinweise gegeben (u.a. zu den Themen Software-Richtlinien, Typisierung, Vertragsbasierte Programmierung, Fehlertolerante Programmierung, Portabilität und Dokumentation), wie sich Probleme im Ansatz vermeiden lassen (Hof08, S.65ff).

Das Verfahren der konstruktiven Qualitätssicherung beinhaltet u.a.:

- eine formale Syntaxprüfung zur Vermeidung von Kompilierungsfehlern und
- eine Prüfung auf potentielle Fehlermuster, die, statistisch betrachtet, häufig auftreten (z.B. komplexe Ausdrücke durch unnötige Negation und sprachliche Verwechslungen, die formal korrekt sind[3]).

Die dynamischen Verfahren setzen auf Informationen, die während der Laufzeit gewonnen wurden. Auf diese Weise versuchen sie, den Fehler einzugrenzen (Lib+05; Lib04; SO05; NL10). Dabei steht die voll- bzw. semiautomatische Isolation der Fehler im Mittelpunkt.

So wird z.B. beim *Delta Debugging* (siehe Kapitel 2.1.3, S.10) durch Minimierung der Eingabedaten bzw. Konfigurationen und Vergleichen von Versionen eines Projektes versucht, den Fehler zu isolieren. Dabei ist jedoch für die Durchführung der Testläufe ein teilweise immenser Aufwand notwendig[4](Lib+05, S.6).

Ein weiteres dynamisches Verfahren ist die „Kooperierende Fehlerisolierung" (engl: cooperative bug isolation, siehe (Lib04)). Diese versucht z.B., Fehlern durch Laufzeitanalysen auf die Spur zu kommen. Dabei greift man jedoch invasiv in die Ablaufstruktur der Programme ein, um so interne Operationen zu beobachten.

Beide Verfahren der dynamischen Fehlereingrenzung setzen auf mehrfache Programmausführungen. Keiner der beiden Ansätze äußert sich zu Bedingungen im Arbeitsumfeld, die sich nicht am Optimum orientieren, aber so in der Realität häufig anzutreffen sind, wie z.B. :

- bestehende Softwareprojekte, deren Qualitätsniveau kurz- bzw. langfristig nicht verbessert werden kann, so dass dem Auftreten von Fehlern vorgebeugt wird
- lang laufende Tests, die eine mehrfache Ausführung für eine automatisierte Isolierung des Fehlers nicht wirtschaftlich erscheinen lassen.

[3]Dazu zählt z.B. der Java-Ausdruck $if(i = 1)\{\}$.
[4]„In each study we ran the programs on about 32,000 random inputs" (Lib+05, S.6).

Die in der Literatur beschriebenen zwei Vorgehensweisen werden den Anforderungen der Praxis nicht voll gerecht. Aus diesem Grund wird versucht, aus einer Kombination dynamischer Verfahren und konstruktiver Qualitätssicherung eine effizientere Fehlersuche zu erreichen.

Aufbau der Arbeit

Für die Erarbeitung der Analyse und des Konzeptes zur Verbesserung der statischen Fehlereingrenzung werden in Kapitel 2 die theoretischen Hintergründe erläutert.

In Kapitel 3 erfolgt die Analyse der fehlgeschlagenen Tests. Diese sind die Grundlage für die sortierte Empfehlungsliste, die dem Entwickler zugänglich gemacht wird.

Kapitel 4 behandelt die Evaluation und Implementierung des Prototypen. Dabei wird die Frage beantwortet, welche vom Verfasser im Vorfeld gesammelten Erfahrungen die Herausbildung der in dieser Arbeit gewählten Schwerpunkte beeinflusst haben und wie deren technische Umsetzung erfolgte.

In Kapitel 5 wird die Weiterentwicklung der Methodik, die als Schwerpunkt dieser Arbeit anzusehen ist, ausführlich darlegt.

Anhand eines Fallbeispiels (Kapitel 6) werden in Kapitel 7 in Form einer Untersuchung Hinweise gesammelt, die eine Verbesserung der Effizienzsteigerung des Fehleranalyseprozesses durch den Einsatz der weiterentwickelten Methodik nachweisen.

In Kapitel 8 werden in Form eines Fazits und Ausblicks die Ergebnisse dieser Arbeit zusammengefasst.

2 Theoretischer Hintergrund

2.1 Begriffsdefinitionen

In diesem Kapitel müssen für das weitere Verständnis bestimmte Begriffe erläutert werden. Im Anschluss erfolgt eine Beschreibung der Test-Klassifikation, danach des Erklärungsmodells und Prozesses der Fehlerbehandlung.

2.1.1 Fehler

In der Norm ISO 9000:2005 wird ein Fehler allgemein als Nichterfüllung einer festgelegten Anforderung definiert. Für den Bereich Software trifft Peter Liggesmeyer für Fehler in (Lig09, S.7) folgende Aussage: „Ein Fehler oder Defekt (*fault*,*defect*) ist bei Software die statisch im Programmcode vorhandene Ursache eines Fehlverhaltens oder Ausfalls".

Andreas Zeller definiert Fehler in (Zel03, S.10) als

- „jede Abweichung der tatsächlichen Ausprägung eines Qualitätsmerkmals von der vorgesehenen Soll-Ausprägung",
- „jede Inkonsistenz zwischen Spezifikation und Implementierung".

Diesen Definitionen kann uneingeschränkt zugestimmt werden. Da Fehler immer wieder auftreten werden, gilt es jedoch, sich mit den Ursachen zu beschäftigen, um in der Folge diese einzugrenzen.

2.1.2 Fehlereingrenzung

Die Fehlereingrenzung (engl.: *bug isolation*) ist Teil der Fehlersuche und -behebung (engl.: *debugging*) (Bal99, S.170). Dieser Begriff beschreibt das Vorgehen zur Ermittlung der unmittelbaren Ursache eines Fehlverhaltens.

2.1.3 Delta-Debugging

Delta-Debugging[5], ein Konzept zur automatischen Fehlereingrenzung, wurde von Andreas Zeller entwickelt. Er geht in dem Artikel *Yesterday, my program worked. Today, it does not. Why?* (Zel99) anschaulich auf ein Problem der aktuellen Softwareentwicklung ein, nämlich der Komplexität der Fehlerbehebung im Vergleich zur Entwicklungsgeschwindigkeit. Darin beschreibt er die Situation der täglichen Entwicklungsprozesse so, dass einerseits immer schneller Neuerungen eingebracht werden können, andererseits jedoch der Prozess der Fehlersuche sich dieser Geschwindigkeit nicht angepasst hat.

Die automatische Fehlereingrenzung erfolgt mit Hilfe des Delta-Debugging, indem die Änderungen analysiert werden. Diese können z.B. Unterschiede zwischen zwei Versionen eines Quelltextes, zwei Mengen von Eingabedaten oder auch verschiedenen Zustandswechseln beinhalten.

Delta-Debugging besteht im Wesentlichen aus zwei grundlegenden Algorithmen:

Vereinfachung

Die Menge der Änderungen wird soweit reduziert, bis letztendlich keine kleinere Menge, die den Fehler reproduziert, gefunden werden kann.

Isolierung

Hierbei wird die kleinste Änderung, die zu diesem Fehler geführt hat, gesucht.

[5] Abgeleitet von „Delta“ als Beschreibung eines Unterschiedes zwischen zwei Versionen.

Dieses Konzept der automatischen Fehlereingrenzung macht es möglich, mit vielen Änderungen und wahrscheinlichen Fehlern mit wenig Zeitaufwand, verglichen mit dem Einsatz klassischer Instrumente zur Fehlersuche, umzugehen.

2.1.4 Qualität

Der Begriff *Qualität* wird in (NNQ05) wie folgt definiert: *„Vermögen einer Gesamtheit inhärenter (lat. innewohnend) Merkmale eines Produkts, eines Systems oder eines Prozesses zur Erfüllung von Forderungen von Kunden und anderen interessierten Parteien"*. Dr. J.M. Juran[6] beschreibt in (SJ05, S.71) Qualität sehr prägnant als *„Bereit für den Einsatz"* (*„Juran broadened the definition of quality from conformance to specification to "fitness for use"."*). Qualität, so definiert, weist vor allem auf die praxisorientierte Bedeutung des Begriffs hin.

Wesentlich konkreter in Bezug auf Software bestimmt die Norm DIN ISO 9126 den Begriff Qualität: *„Software-Qualität ist die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen"*.

2.1.5 Qualitätsmerkmale

Qualität beinhaltet nach der DIN-ISO-Norm 9126 die relevanten Merkmalsausprägungen (Abbildung 2.1, S.12).

Peter Liggesmeyer definiert den Begriff *Qualitätsmerkmal* als „Eigenschaft einer Funktionseinheit, anhand derer ihre Qualität beschrieben und beurteilt wird, die jedoch keine Aussage über den Grad der Ausprägung enthält. Deshalb kann ein Qualitätsmerkmal über mehrere Stufen in Teilmerkmale verfeinert werden" (Lig09, S.515).

Diese Qualitätsmerkmale konkurrieren jedoch miteinander. Deshalb müssen auf der Grundlage eines Kompromisses Ziele formuliert werden, die zu einer vertretbaren Qualität des Softwareproduktes führen.

[6] Joseph Moses Juran war ein rumänisch-amerikanischer Wirtschaftsingenieur und zählte zu den Wegbereitern des Qualitätsmanagements (Zol10, S.96).

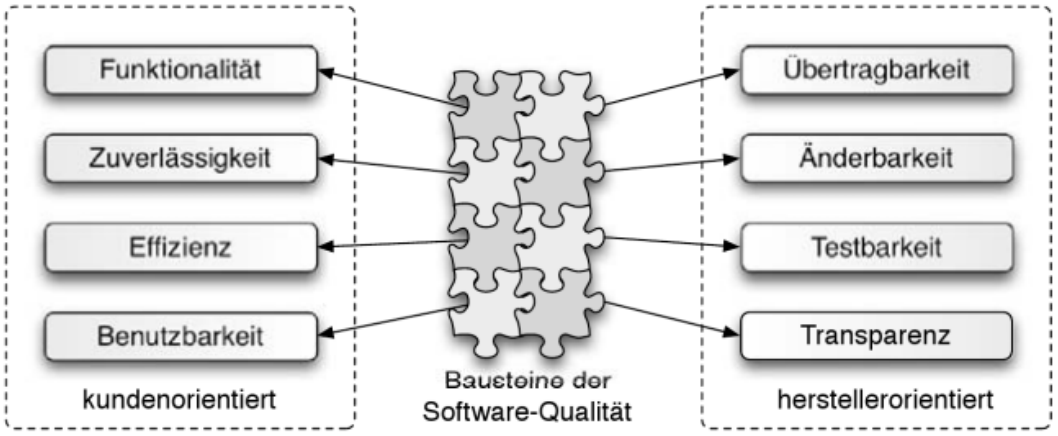


Abbildung 2.1: Qualitätsmerkmale eines Software-Produktes (Hof08, S.7).

Zunächst erfolgt die Beschreibung der Qualitätsmerkmale, die nach (Hof08, S.6ff) in der Praxis die größte Relevanz aufweisen. Danach werden Überschneidungen und Konflikte in einer Korrelationsmatrix dargestellt.

Funktionalität beschreibt die Bereitstellung von Funktionsweise eines Softwareproduktes für den Betrieb und die spezifizierten Rahmenbedingungen.

Laufzeit beschreibt die Fähigkeit eines Softwareproduktes, im Rahmen der Spezifikation eine ausreichende Leistung bereitzustellen. Das können u.a. interaktive Mensch-Maschine-Dialoge oder harte Zeitanforderungen im Bereich der Echtzeitsysteme sein.

Zuverlässigkeit beschreibt die Fähigkeit eines Softwareproduktes, ein festgelegtes Leistungsniveau unter den für die Benutzung geltenden Rahmenbedingungen zu erreichen und zu bewahren.

Benutzbarkeit beschreibt die Fähigkeit eines Softwareproduktes, vom Benutzer verstanden und benutzt werden zu können.

Wartbarkeit beschreibt die Fähigkeit eines Softwareproduktes, änderungsfähig zu sein. Änderungen können u.a. Korrekturen, Verbesserungen, Anpassungen an eine geänderte Umgebung bzw. Spezifikation sein.

Transparenz beschreibt das Kriterium, auf welche Art und Weise die äußerlich sichtbare Funktionalität intern implementiert wurde. In diesem Zusammenhang sind folgende Fragen zu beantworten: Wie sauber ist die interne Programmstruktur? Ist die Software durch zunehmende Software-Entropie[7] sichtbar gealtert?

Übertragbarkeit beschreibt die Fähigkeit eines Softwareproduktes, in eine veränderte Umgebung übertragen werden zu können. Dazu zählt u.a. die eventuell nötige Anpassung an unterschiedliche Hardware- und Softwarekomponenten für eine neue Arbeitsumgebung.

Testbarkeit beschreibt die Fähigkeit, eine Veränderung der Software zu validieren. Dazu ist es u.a. notwendig, interne Strukturen nach außen sichtbar zu machen und den Black-Box-Charakter vieler Komponenten mit zusätzlichem Code aufzubrechen.

Obwohl es möglich erscheint, diese Qualitätsmerkmale in gleichem Maße auszubilden, sind dem jedoch natürliche Grenzen gesetzt. Konflikte ergeben sich z.B. bei der Zielsetzung, bestmögliche Laufzeitwerte mit einer maximalen Übertragbarkeit zu kombinieren. Bestmögliche Laufzeitwerte würden voraussetzen, dass sehr viele Anpassungen an die verwendete Plattform erfolgen müssten. Im Gegensatz dazu erfordert eine maximale Übertragbarkeit sehr allgemeine und damit auch übertragbare Implementierungen, die ihrerseits jedoch nicht immer die besten Laufzeiten erreichen können. Solche *natürlichen* Konflikte sind in der Abbildung 2.2, S.14, dargestellt.

[7] „Der Grad der Unordnung eines Programms wird auch als Software-Entropie bezeichnet“ (Hof08, S.9).

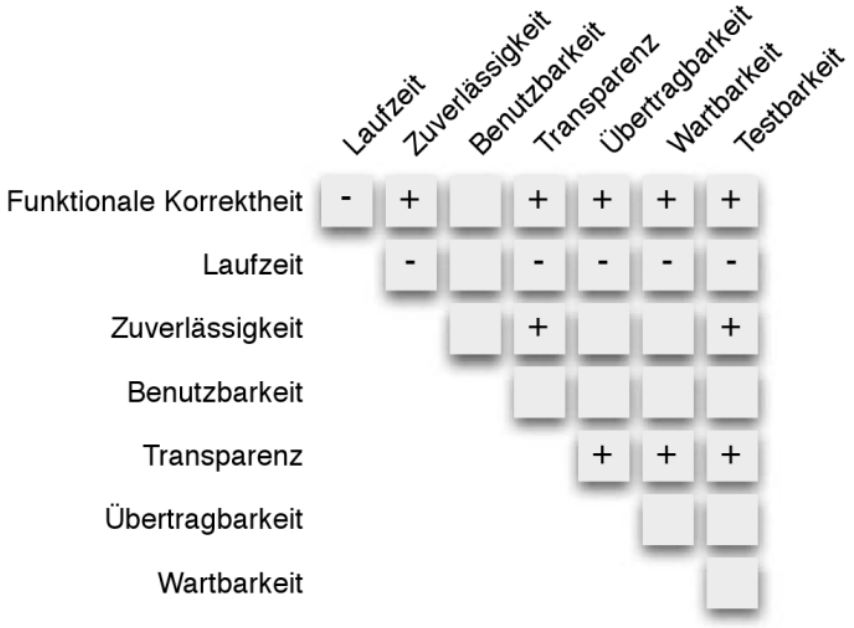


Abbildung 2.2: Korrelationsmatrix der Qualitätsmaße (Hof08, Abb. 1.3, S.11).

Neben dem Ziel, einen ausgewogenen Einsatz der Qualitätsmerkmale zu erreichen, stellt sich auch die Aufgabe, diese zu messen. So können Stand und Fortschritt der Qualitätssicherung, die den Softwareentwicklungsprozess begleitet, beobachtet werden (Bal08, S.377). Im Folgenden wird dargelegt, auf welche Art und Weise die Qualitätsmerkmale z.T. messbar gemacht werden.

2.1.6 Software-Metriken

Software-Metriken sind als leistungsfähiges Hilfsmittel in der Softwareentwicklung anerkannt (Hof08, S.247). „Das Wort *Metrik* entstammt dem Griechischen und bedeutet Kunst des Messens" (Lig09, S.233). Metriken[8] werden für die systematische Erfassung der quantitativen und qualitativen Merkmale eines Softwaresystems benötigt. Zur objektiven Beurteilung von

[8] Auf den Diskurs über die Verwendung der Begriffe *Maß* und *Metrik* wird nicht eingegangen (Lig09, S.233f; Bal08, S.377f), weil dies im Rahmen der Arbeit als nicht relevant angesehen wird.

Qualitätskriterien sind Kennzahlen notwendig, die diese quantifizierbar machen. Die Bereitstellung entsprechender Kennzahlen ist die Aufgabe von Software-Metriken. Somit ist eine formale Vergleichbarkeit der Software möglich.

Zum besseren Verständnis werden zunächst für Metriken geltende Gütekriterien erläutert. Danach erfolgt die Beschreibung der Skalentypen, in die die Werte der Metriken eingeordnet werden. Abschließend veranschaulicht eine Übersicht die Kategorien, nach denen Metriken gruppiert werden können.

Gütekriterien

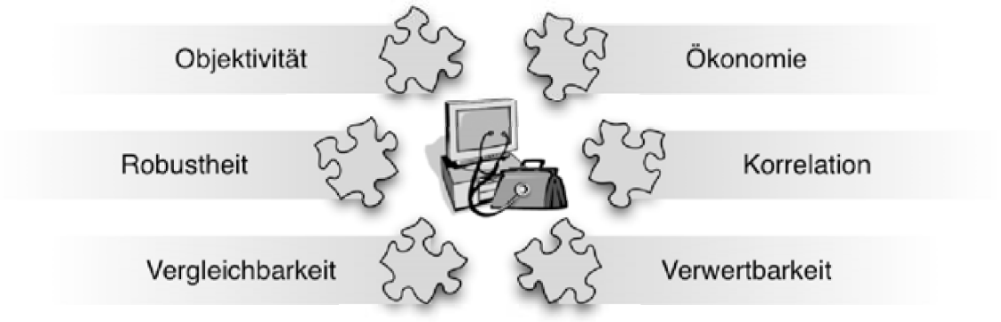


Abbildung 2.3: Gütekriterien, die eine Metrik erfüllen muss (Hof08, Abbildung 5.1, S.248).

Für den erfolgreichen Einsatz von Software-Metriken sind verschiedene Gütekriterien einzuhalten (Abbildung 2.3). Im Weiteren erfolgt eine Charakterisierung dieser Gütekriterien (Lig09, S.236; Hof08, S.248f; Bal08, S.383):

Objektivität (Reproduzierbarkeit)
: Eine Metrik gilt als objektiv, wenn sie von ihrem Wert unabhängig ist und ohne subjektive Einflüsse zustande kommt.

Robustheit (Stabilität, Zuverlässigkeit)
: Eine Metrik gilt als zuverlässig bzw. stabil, wenn sie unter gleichen Messbedingungen einen identischen Wert erreicht.

Vergleichbarkeit (Analysierbarkeit)
: Eine Metrik gilt als analysierbar, wenn ihre Werte zu anderen Werten der gleichen Software-Metrik in Relation gesetzt werden können.

Ökonomie (Rechtzeitigkeit)
: Eine Metrik gilt als ökonomisch, wenn sie mit vertretbarem Aufwand ermittelt werden und damit Einfluss auf das Produkt nehmen kann.

Korrelation (Eignung, Validität)
: Eine Metrik gilt als geeignet, wenn sie mit der zu messenden Eigenschaft in einem hinreichend starken Maß korreliert.

Verwertbarkeit (Nützlichkeit, Einfachheit)
: Eine Metrik gilt als nützlich, wenn sie mit einem angemessenen Interpretationsaufwand das Handeln beeinflusst und praktische Anforderungen erfüllt.

Helmut Balzert nennt zu den bereits oben genannten als weiteres siebtes Kriterium in (Bal08, S.383) explizit die *Normierung*: Eine Metrik gilt für ihn als normiert, wenn sie auf einer Skala abgebildet werden kann.

Skalentypen

Zwecks Vergleichbarkeit der Werte einer Software-Metrik ist es notwendig, diese einem Skalentyp zuzuordnen (Tabelle 2.1).

Skalentyp	Beschreibung	Beispiel
Nominalskala	freie Bezeichnung bestimmter Eigenschaften mit Markierungen	Matrikelnummern, Begriffsgruppen
Ordinalskala	Abbildung eines geordneten Aspekts einer Eigenschaft auf eine geordnete Menge von Messwerten unter Beibehaltung der Ordnung	Schulnoten, Windstärken
Rationalskala	Werte können zueinander in Relation gesetzt werden	Länge in Metern (doppelt so weit wie ...)
Absolutskala	Skala, auf der Werte als Zahl dargestellt werden und keine[9] Transformationen erlaubt sind	Häufigkeiten, Wahrscheinlichkeiten

Tabelle 2.1: Übersicht der Skalentypen mit einer Kurzbeschreibung und ausgewählten Beispielen nach (Lig09, S.238f).

Im Weiteren erfolgt in Form einer Übersicht die Kategorisierung von Software-Metriken.

Kategorisierung von Software-Metriken

Software-Metriken können anhand verschiedener Merkmalsausprägungen unterschieden werden. Im Folgenden werden drei für den weiteren Verlauf relevante Ausprägungen vorgestellt, nämlich Aufbau, Anwendung und Zweck. Die Kategorien von Software-Metriken sind in einer Tabelle dargestellt (Tabelle 2.2, S.18).

[9] Mit Ausnahme der Transformation mit einer Identitätsfunktion.

Kriterium	Ausprägung	Beschreibung
Aufbau		
	Basis-Metriken	Metriken, deren Berechnungsvorschrift nicht auf andere Metriken zurückgreift
	Hybride Metriken	Metriken, die eine oder mehrere Metriken zusammenfassen
Anwendung		
	statisch	Berechnung erfolgt aufgrund der Struktur des Quelltextes
	dynamisch	Berechnung erfolgt aufgrund zeitlich veränderlicher Einflüsse (u.a. Laufzeit, Nutzung)
Zweck		
	Umfangsmetriken	Messung erfolgt aufgrund des Umfangs des Quelltextes
	Halstead-Metriken[10]	Berechnung der textuellen Komplexität des Quelltextes
	Strukturmetriken	Berechnung der dem Klassenverbund zugrunde liegenden Struktur (Vererbung, Interaktionen)

Tabelle 2.2: Übersicht der Kategorien von Software-Metriken (Lig09; Hof08; Bal08).

Zwecks Unterscheidung der Tests anhand ihrer Typinformationen ist es notwendig, eine spezielle Metrik in Form der Typdistanz zu erstellen. Diese spezielle Software-Metrik ist allgemein einsetzbar, wird jedoch im Rahmen dieser Arbeit nur für die Testsortierung genutzt. Auf diese Weise kann die Wichtung der fehlgeschlagenen Tests erfolgen und die Fehlereingrenzung unterstützt werden.

[10] Halstead forderte Metriken (Vokabular, Minimalvokabular, Länge, Volumen, Minimalvolumen, Level, Schwierigkeit und Aufwand), die sich unmittelbar aus der lexikalischen Struktur und dem Programmtext herleiten (Hof08, S.259).

2.1.7 Typdistanz

Die *Typdistanz* ist eine statische objektorientierte Strukturmetrik und bezeichnet die Entfernung zwischen zwei Klassen (Typen). Die Entfernung wird hierbei auf einer Absolutskala angegeben.

Die Maßzahl $M_{distance}$ beschreibt die Entfernung zwischen zwei Typen im Namenraum-Baum (Abbildung 2.4).

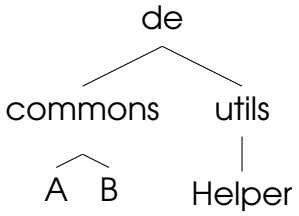


Abbildung 2.4: Darstellung eines Namenraumes als Baum.

Formal handelt es sich bei einem Namenraum um einen Baum[11], in dem die Blätter Typen sind. Dabei sei $G = (V, E)$ ein Graph mit dem Knoten V und den Kanten E.

$$
\begin{aligned}
V &= \{de, commons, A, B, utils, Helper\} \\
E &= \{\{de, commons\}, \{commons, A\}, \{commons, B\}, \{de, utils\}, \\
&\quad \{utils, Helper\}\} \\
M_{distance} &= d(V_m, V_n)
\end{aligned}
$$

Der Begriff *Typdistanz* lässt sich für einen Test in Form

- der Einzeltypdistanz und
- Gesamttypdistanz betrachten.

[11] Ein zusammenhängender und azyklischer Graph (Fel05, S.23).

Einzeltypdistanz

Die Einzeltypdistanz beschreibt eine einzelne Typdeklaration im Test. Somit kann ein Test aus mehreren Einzeltypdistanzen bestehen. Ist jedoch in einem Test keine Typdeklaration enthalten, so kann keine Einzeltypdistanz berechnet werden.

Die Berechnung der Einzeltypdistanz beschreibt die folgende Formel:

$$d(V_0, V_1) = \begin{cases} 0 & \text{wenn } V_0 \text{ oder } V_1 \text{ Vertreter eines Typs des Standard-sprachumfangs}[12] \text{ sind} \\ 1 & \text{wenn } V_0 \text{ und } V_1 \text{ zwei Nachbarblätter sind} \\ x & \text{Anzahl der Kanten zwischen } V_0 \text{ und } V_1 + 1 \end{cases}$$

In Tabelle 2.3 sind die Ergebnisse der berechneten Einzeltypdistanzen dargestellt.

V_0	V_1	$M_{distance}$
de.commons.A	de.commons.B	1
de.commons.A	de.utils.Helper	5 (4+1)
de.commons.A	java.lang.String	0

Tabelle 2.3: Einzeltypdistanz $M_{distance}$ für ausgewählte Typen.

[12] Dazu zählen z.B. bei Java alle Klassen aus den Paketen `java.*`.

Ein Schema für die Darstellung der Einzeltypdistanzen ist aus Abbildung 2.5, ersichtlich.

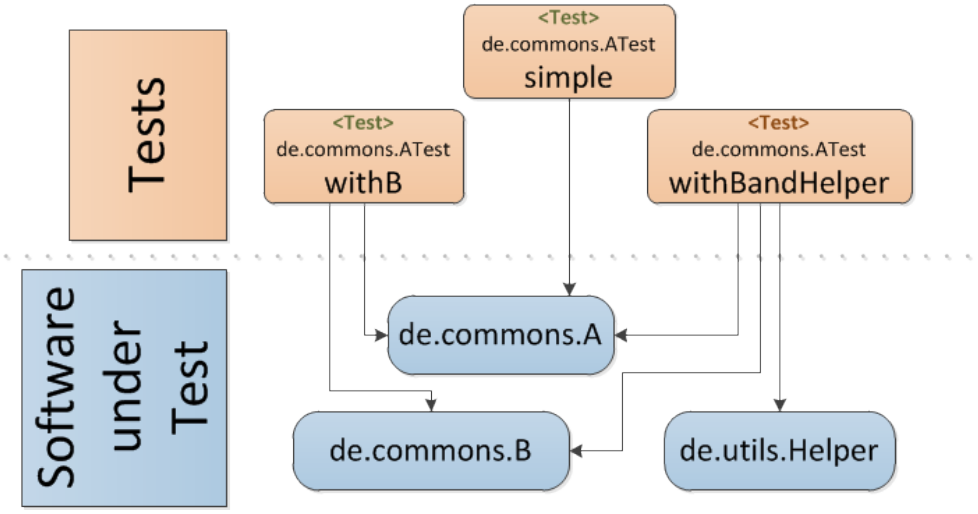


Abbildung 2.5: Schematische Darstellung der Einzeltypdistanzen am Beispiel unterschiedlicher Tests.

Gesamttypdistanz

Die Gesamttypdistanz beinhaltet die Zusammenfassung aller *Einzeltypdistanzen* eines Tests. Dieses Maß ergibt sich aus der Berechnungsvorschrift, auf die in Kapitel 5.4, S.56 näher eingegangen wird.

2.1.8 Fehlerfortpflanzung

Bei der Betrachtung von Fehlern kommt es vor, dass diese durch ursächlich andere hervorgerufen werden. In diesem Zusammenhang spricht man von dem der Physik entlehnten Begriff *Fehlerfortpflanzung*[13].

Es ist allgemein bekannt, dass ein Fehler weitere nach sich ziehen kann[14]. Bei der Softwareentwicklung kann entsprechend dem Prinzip der Fehlerfortpflanzung ein fehlgeschlagener Test u.U. nur als Symptom gewertet wer-

[13] „Da jeder Messwert der einzelnen Größen von seinem richtigen Wert abweicht, wird auch das Ergebnis der Rechnung von seinem richtigen Wert abweichen“ (Wik11a).

[14] „Es ist grundsätzlich nicht möglich, fehlerfrei zu messen. Die Abweichungen der Messwerte von ihren wahren Werten wirken sich auf ein Messergebnis aus, so dass dieses ebenfalls von seinem wahren Wert abweicht“ (Wik11b).

den. Die Fehlerfortpflanzung erzeugt durch die Kopplung der einzelnen Komponenten ein gewisses Fehler-Rauschen in Form weiterer fehlgeschlagener Tests. Es konnte beobachtet werden, dass zusätzliche Fehler, die durch die Fehlerfortpflanzung auftreten, eine größere Gesamttypdistanz aufweisen. Somit wird durch eine größere Anzahl von Fehlern die Analyse des Kernproblems erschwert. Es könnte zum Beispiel sein, dass die in Abbildung 2.6, S.22, dargestellte Übersicht der zehn fehlgeschlagenen Tests nur vier Fehler aufweist.

Wenn man die Tests nach ihrem Alter sortiert, ergeben sich vier Gruppen. Dabei könnte die Gruppe mit dem geringsten Alter von 1, die aus sieben fehlgeschlagenen Tests besteht, eventuell nur auf einen Fehler zurückgeführt werden.

Alle fehlgeschlagenen Tests

Testname	Dauer	Alter
>>> org.apache.tools.ant.taskdefs.JavaTest.testRunFail	0.072	1
>>> org.apache.tools.ant.taskdefs.JavaTest.testRunFailFoe	0.069	1
>>> org.apache.tools.ant.taskdefs.JavaTest.testRunFailFoeFork	0.576	1
>>> org.apache.tools.ant.taskdefs.JavaTest.testExcepting	0.078	1
>>> org.apache.tools.ant.taskdefs.JavaTest.testExceptingFork	0.577	1
>>> org.apache.tools.ant.taskdefs.JavaTest.testExceptingFoe	0.079	1
>>> org.apache.tools.ant.taskdefs.JavaTest.testRedirector2	182.092	1
>>> org.apache.tools.ant.util.SymlinkUtilsTest.testRootIsNoSymlink	0.0010	29
>>> org.apache.tools.ant.taskdefs.ManifestClassPathTest.testSameWindowsDrive	0.016	58
>>> org.apache.tools.ant.taskdefs.DeleteTest.test9	0.779	161

Abbildung 2.6: Liste von Fehlern mit der Angabe der Laufzeit (Dauer) und der Anzahl der Testläufe (Alter), in denen diese erstmalig auftraten.

2.1.9 Kleinster Test

Der kleinste Test beschreibt *den* Test, der im Rahmen aller fehlgeschlagenen Tests die geringste *Gesamttypdistanz* (S.21) aufweist.

2.1.10 Kleinster Fehler

Ausgehend von den Definitionen für *Fehler* und *kleinster Test* können diese, zusammengenommen, auch als *kleinster Fehler* bezeichnet werden. Sollte in einem *kleinsten Test* ein *Fehler* gefunden werden, so wird dieser als *kleinster Fehler* bezeichnet.

Die Zusammenfassung eines fehlgeschlagenen Tests und eines Fehlers (es kann sich um einen Programmfehler oder einen fehlerhaften Test (*false positives*) handeln) ist im Rahmen dieser Arbeit notwendig, weil jeweils die Ursache behoben werden muß. In beiden Fällen ist *eine* Anforderung nicht erfüllt.

Im Fall eines Programmfehlers ist offensichtlich eine Verletzung der Anforderungen eingetreten. Allerdings geschieht dies auch, wenn Tests unberechtigterweise fehlschlagen und somit die Erfüllung der Anforderungen angezweifelt werden muss.

2.1.11 Kontinuierliche Integration

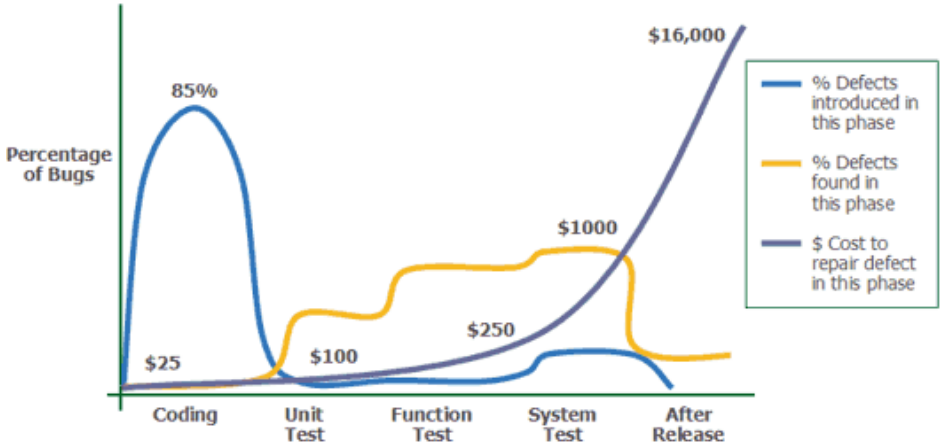


Abbildung 2.7: Entwicklung der Kosten für die Fehlerbereinigung in Abhängigkeit zur Zeit und Verteilung der entstandenen Fehler (Tec09).

Das Konzept der Kontinuierlichen Integration ist eine Vorgehensweise, die die Entwicklung von Software über den gesamten Prozess der Implementierung mit regelmäßigen Integrationsphasen begleitet. Dieses

Konzept stellt mit Hilfe der Testautomatisierung eine Säule im agilen Softwareentwicklungsprozess dar (Tho06; Goh10, S.5).

2.2 Test-Klassifikation

Softwaretests werden in allen Phasen des Entwicklungsprozesses durchgeführt. Einzelne Tests lassen sich laut Hoffmann anhand der folgenden Merkmale in Klassen einteilen (Hof08, S.158):

- **Prüfebene:** „In welcher Entwicklungsphase wird der Test durchgeführt?"
- **Prüfkriterium:** „Welche inhaltlichen Aspekte werden getestet?"
- **Prüfmethodik:** „Wie werden die verschiedenen Testfälle konstruiert?"

Zur Betrachtung der Tests kommt den Testklassen, entsprechend dem Merkmal der *Prüfebene*, im Rahmen dieser Arbeit eine besondere Bedeutung zu (Abbildung 2.8, S.25), wie sich das aus der Beschreibung des ursprünglichen Ansatzes in Kapitel 4.1, S.37 erkennen läßt.

Das Merkmal der *Prüfebene* wird in weitere Unterklassen unterteilt. Diese Unterklassen, eingeteilt in Prüfebenen, resultieren laut Hoffmann aus der Programmstruktur und dem Zeitpunkt der Entwicklung, zu dem ein Test durchgeführt wird (Hof08, Abb. 4.1 S.158).

Im Folgenden werden die vier Testklassen, die sich daraus ergeben, vorgestellt:

- Unittests,
- Integrationtests,
- Systemtests,
- Abnahmetests.

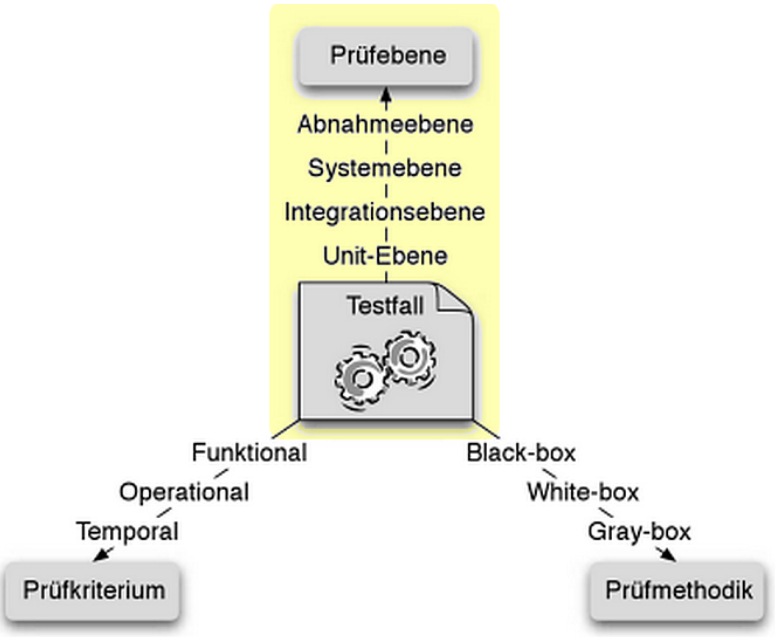


Abbildung 2.8: Die drei Merkmalsräume der Testklassifikation mit der Fokusierung auf die Prüfebene (Hof08, Abb. 4.1 S.158).

2.2.1 Unittests

Unittests, auch als Modultests bezeichnet, sind Tests für die kleinste eigenständig testbare Programmeinheit. Diese Begriffsdefinition der kleinsten Einheit führt dazu, dass mitunter Tests für ganze Programmbibliotheken dazu gehören und man, allgemeiner formuliert, von Komponententests sprechen sollte (Hof08, S.159). Infolge dieser sehr unklaren Trennung ist auch der Übergang von Unittests zu Integrationstests fließend.

2.2.2 Integrationstests

Integrationstests sind Tests, „die begleitend zur Zusammenfassung von Komponenten“(Lig09, S.512) zu einem Softwaresystem erstellt werden.

In (Hof08, S.163) werden Integrationstests als übergeordnete Unittests beschrieben. Ziel ist es, das Zusammenwirken mehrerer Komponenten zu testen, wobei es vor allem um die Funktionsfähigkeit der in Unittests überprüften Komponenten als Verbund geht.

2.2.3 Systemtests

Ein Systemtest prüft, im Gegensatz zu den Unit- und Integrationstests, die Einhaltung der Kriterien aus dem Pflichtenheft. Hoffmann beschreibt Systemtests – im Gegensatz zu den vorher genannten Testarten – als Übergang von einer internen Sicht des Testers zu einer externen kundenähnlichen Sicht auf das System (Hof08, S.166). Somit ist äußere Funktionalität wichtiger als die innere Struktur des Quelltextes bzw. der Softwarearchitektur. In (Lig09, S.376) wird der Systemtest ebenfalls als ein „Test des fertigen Systems gegen die in den Anforderungsdokumenten festgelegten Funktions-, Leistungs- und Qualitätsanforderungen" charakterisiert.

2.2.4 Abnahmetests

Der Abnahmetest ist ein „Test mit dem Ziel festzustellen, ob eine Software akzeptiert werden kann" (Lig09, S.509). Er ähnelt dem Systemtest darin, dass beide Prüfungen am fertigen System stattfinden. Im Gegensatz zum Systemtest, dessen Durchführung beim Hersteller erfolgt, wird der Abnahmetest durch den Kunden bzw. bei ihm in einer sehr realen Testumgebung durchgeführt (Hof08, S.168f). Darin besteht der wesentliche Unterschied zwischen den beiden Testarten.

2.3 Beschreibung des Erklärungsmodells

Für die Änderung der Methodik zur Fehlereingrenzung ist es notwendig, ein Modell anschaulich zu beschreiben, wodurch es möglich sein wird, die Reduzierung des Aufwandes nachzuvollziehen.

In diesem Modell sind folgende Sachverhalte definiert:

- Es gibt eine Software S.
- Diese Software S besteht aus mehreren Komponenten $K_1..K_n$.
- Diese Software S hat eine Reihe von Tests $T_1..T_m$.
- Die Tests $T_1..T_m$ beziehen sich willkürlich auf die Komponenten$K_1..K_n$.

- Die Beziehung eines Tests T zu einer Komponente K wird dabei als T > $\{K_1, .., K_n\}$ dargestellt.
- Jeder Test T steht zu mindestens einer Komponente K in Beziehung.

2.4 Prozess der Fehlerbehandlung

Obwohl allgemein bekannt, wird zunächst zwecks besseren Verständnisses und entsprechender Einordnung der weiterentwickelten Methodik (Kapitel 5, S.49) in den Gesamtprozess die Fehlerbehandlung (Abbildung 2.9, S.27) dargestellt.

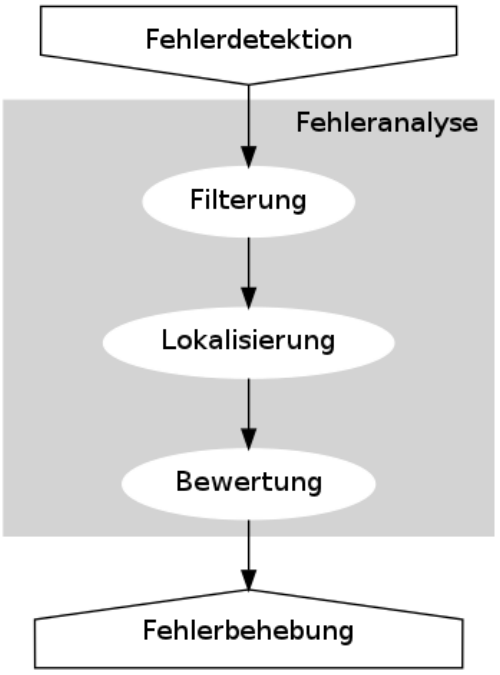


Abbildung 2.9: Allgemeine Darstellung des Prozesses der Fehlersuche.

1. **Fehlerdetektion**

   In diesem Schritt werden Fehler gefunden. Dies geschieht durch regelmäßige Tests (in sehr hohem Grad automatisiert) oder auch durch auftretende Unstimmigkeiten bei der Anwendung.

2. **Fehleranalyse**

   Dieser Prozessschritt analysiert die gefundenen Fehler.

   a) **Filterung**

      Häufen sich Fehler, weil eventuell durch die Fehlerfortpflanzung an unterschiedlichen Stellen weitere auftreten, erfolgt im Vorfeld eine Sortierung[15].

   b) **Lokalisierung**

      Nach der Filterung wird versucht, die Unstimmigkeit im Softwareprodukt (Quelltext und Konfiguration) zu lokalisieren.

   c) **Bewertung**

      Den Abschluss der Fehleranalyse bildet die Bewertung bzw. Einordnung des Problems hinsichtlich der Komplexität und Dringlichkeit.

3. **Fehlerbehebung**

   Jetzt wird der Fehler behoben. Fehlerbehebung kann in der Praxis auch Akzeptanz des erkannten Fehlverhaltens bedeuten.

[15] Dabei kann es sich auch um die chronologische Auftretensreihenfolge der Fehler handeln.

3 Analyse fehlgeschlagener Tests

Nachdem im Kapitel 2, S.9 Begriffsdefinitionen behandelt wurden, beleuchtet das folgende Kapitel Hintergründe der Analyse fehlgeschlagener Tests.

In Kapitel 2.1.11, S.23 wurde beschrieben, dass der Testautomatisierung eine wesentliche Bedeutung zukommt. Dabei ist allerdings weiterhin ungeklärt, wie man die erkannten Fehlschläge in den Tests mit geringerem Aufwand findet. Da sich eine Fehlerursache meist, ähnlich dem Domino-Effekt, durch viele Komponenten zieht (Kapitel 2.1.8, S.21), bleibt noch die mühsame Arbeit zur genauen Lokalisierung der Stelle, die den Fehler ausgelöst hat.

In diesem Kapitel wird die allgemein übliche Vorgehensweise zur Fehlereingrenzung vorgestellt.

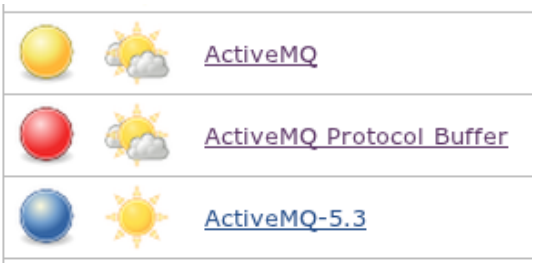
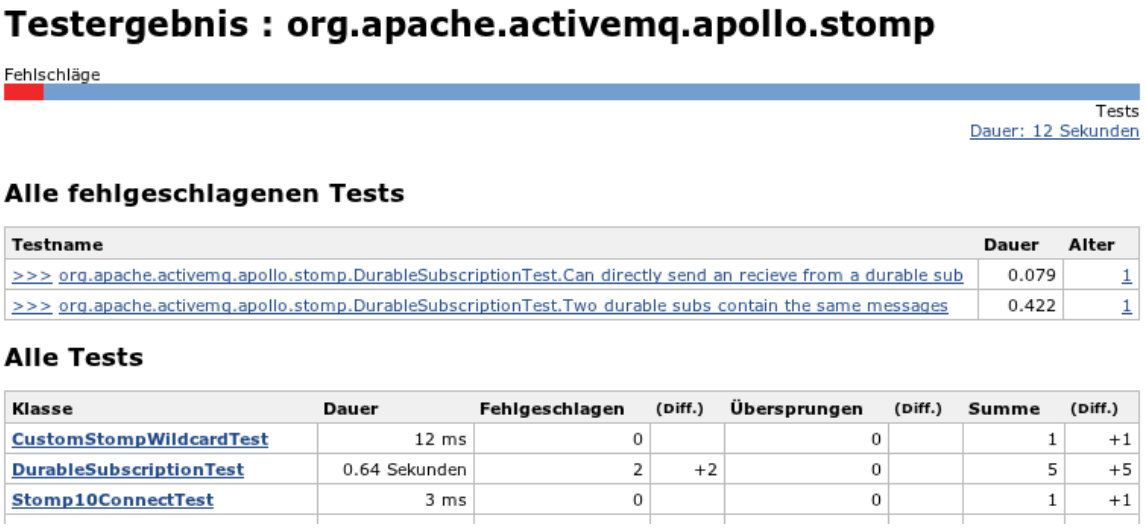


Abbildung 3.1: Ausschnitt aus der Übersicht eines Testberichts. Darin ist mit Hilfe gefüllter Kreise der Status dargestellt (v.o.n.u. instabil, fehlgeschlagen, erfolgreich) (`https://builds.apache.org/`, 12.08.2011 15:40).

Zur Zeit wird in Schritt 1 des Fehlerbehandlungsprozesses, der Fehlerdetektion (Kapitel 2.4, S.27f), ein Testlauf mit einem Bericht abgeschlossen

(lee83). Dieser enthält die Tests und deren jeweiligen Erfolgsstatus (Abbildung 3.1, S.29).

Ist einer der Tests nicht erfolgreich gewesen, so wird er gekennzeichnet und mit der entsprechenden Fehlermeldung versehen. Der Entwickler erfährt in einer abschließenden Zusammenfassung des Berichts, ob Fehler aufgetreten sind. Danach erhält er die Liste mit dem Verzeichnis der fehlgeschlagenen Tests (Abbildung 3.2).



Testergebnis : org.apache.activemq.apollo.stomp

Fehlschläge

Tests

Dauer: 12 Sekunden

Alle fehlgeschlagenen Tests

Testname	Dauer	Alter
>>> org.apache.activemq.apollo.stomp.DurableSubscriptionTest.Can directly send an recieve from a durable sub	0.079	1
>>> org.apache.activemq.apollo.stomp.DurableSubscriptionTest.Two durable subs contain the same messages	0.422	1

Alle Tests

Klasse	Dauer	Fehlgeschlagen	(Diff.)	Übersprungen	(Diff.)	Summe	(Diff.)
CustomStompWildcardTest	12 ms	0		0		1	+1
DurableSubscriptionTest	0.64 Sekunden	2	+2	0		5	+5
Stomp10ConnectTest	3 ms	0		0		1	+1

Abbildung 3.2: Beispiel eines Ausschnitts aus einem Testbericht (https://builds.apache.org/job/ActiveMQ-Apollo 12.08.2011 15:41). In dieser Liste sind die fehlgeschlagenen Tests durch einen Eintrag in der Spalte *Fehlgeschlagen* zu erkennen.

Im Weiteren erfolgt die Darstellung der bekannten Fehlereingrenzungsstrategien. Danach werden diese bewertet und deren Funktionsweise erläutert.

3.1 Allgemeine Strategien der Fehlereingrenzung

Das Fehlen einer systematischen Fehlereingrenzung macht es notwendig, die Analyse der ausgewählten Tests auf der Grundlage verschiedener Strategien durchzuführen.

1. **Sequentielle Strategie**

   Nach dieser Strategie werden die Fehler aufgrund ihrer Auftretensreihenfolge in der Liste untersucht (Abbildung 2.6, S.22).

2. **Erfahrungsbasierte Strategie**

   Mit dem Wissen der internen Abläufe[16] werden die wahrscheinlich verantwortlichen Tests ausgewählt.

3.2 Funktionsweise der Fehlereingrenzungsstrategien

3.2.1 Sequentiell

Wie bereits im Abschnitt Allgemeine Strategien der Fehlereingrenzung auf Seite 30 dargelegt, folgt diese Strategie dem naiven Ansatz der *linearen Suche* (OW02, S.165ff), wobei analog dazu der gleiche Aufwand notwendig ist.

3.2.2 Erfahrungsbasiert

Diese Strategie stützt sich auf die Erfahrung der Entwickler. Mit dem Wissen der internen Abläufe werden die wahrscheinlich verantwortlichen Tests ausgewählt. Somit ist diese Vorgehensweise eher als Heuristik[17] zu bezeichnen, die hauptsächlich auf eine vorangestellte Filterung setzt.

[16] Ähnlich dem Prinzip der *White-Box-Tests*, erfolgt die Orientierung im Gegensatz zu den *Black-Box-Tests* nicht anhand der Spezifikation, sondern anhand des Quelltextes.

[17] „Im Gegensatz zum Algorithmus stellt eine Heuristik eine Lösungsmöglichkeit dar, die, einer Daumenregel ähnlich, zu einer Problemlösung eingesetzt werden kann, ohne eine Lösung zu garantieren. Als Beispiel kann das Schachspiel herangezogen werden. Eine gute Heuristik dabei ist, das Mittelfeld zu kontrollieren, wobei dies kein Rezept zum Sieg ist, sondern lediglich eine Gewinnchancensteigerung" (Wes90, S.356).

Filterung

Die Filterung wird nach dem Prinzip *Teile und Herrsche* angewendet. Ziel dieses Vorverarbeitungsschrittes ist es, die vermeintlich nicht relevanten Fehler auszufiltern.

Nach der Filterung ist die Menge der Fehler auf die *mutmaßlich* wesentlichen reduziert. Die Entscheidung auf der Grundlage einer Mutmaßung, die mit Wissen unterlegt ist, jedoch als nicht gesichert gelten kann, stellt das größte Risiko bei Anwendung dieser Strategie dar.

Diese Strategie basiert in einem außerordentlich großen Maße auf der Erfahrung der Entwickler. Solch eine Abhängigkeit hat zur Folge, dass ein unerfahrener[18] Entwickler keine optimale Auswahl von fehlgeschlagenen Tests trifft und somit der Aufwand steigt.

Risiken der Filterung

Die vorangestellte Filterung birgt zwei Risiken, die sich im Wesentlichen auf den Faktor „Mensch" zurückführen lassen:

1. **Unzureichendes Wissen**

   Mangelhaftes Wissen über die Software führt zu nicht-optimalen Entscheidungen hinsichtlich der Auswahl der zu untersuchenden Fehler.

2. **Nichtdeterministische Verfahrensweise**

   Es kann nicht als gesichert gelten, dass ein bestimmter Entwickler bei einer von ihm wiederholten Untersuchung genau die gleiche Entscheidung treffen wird.

Diese beiden Risiken ergeben logischerweise eine gewisse Schwankungsbreite hinsichtlich des zu erwartenden Aufwandes, was allerdings auch aufgrund der Einordnung als „Heuristik" erwartet werden muß.

Die Tabelle 3.1 veranschaulicht zusammenfassend die Risiken und Vorteile der einzelnen Strategien.

[18] Hier ist der Begriff der „Unerfahrenheit" im Kontext der Software gemeint. Dem Entwickler fehlt es an konkretem Hintergrundwissen zur inneren Funktionsweise der von ihm zu untersuchenden Komponenten.

Strategie	Schwächen (Risiken)	Stärken (Vorteile)	aufwandsstabil[19]
sequentiell	Reihenfolge in der Fehlerliste	einfach	$\ominus$
erfahrungsbasiert	fehlerhafte Filterung, Erfahrungsmangel	nutzt Erfahrung	$neutral$

Tabelle 3.1: Zusammenfassende Übersicht der beschriebenen Fehlereingrenzungsstrategien.

3.3 Bewertung der Fehlereingrenzungsstrategien

Die Bewertung der verschiedenen Strategien erfolgt entsprechend dem Aufwand, der nötig ist, um *Fehler* einzugrenzen.

Im folgenden Verlauf wird der Aufwand zur Durchführung der Fehleranalyse der Einfachheit halber als *gleichverteilt* zwischen verschiedenen ursächlichen Fehlern betrachtet.[20] Allerdings verleiht diese offensichtlich vereinfachte Annahme den Aussagen ein zusätzliches Gewicht, da sich aufgrund der Verschiedenartigkeit die sowohl positiven als auch negativen Effekte verstärken.

Für das Verständnis der Bewertung sind zunächst folgende Begriffe zu erläutern:

1. **Aufwand**

[19] Der Aufwand bleibt bei der wiederholten Ausführung gleich. Dabei wird die Beeinflussung durch randomisierte Testausführung, Verwürfelungen durch parallele Testläufe und wechselnde Entwickler berücksichtigt.

[20] Es ist bekannt, dass diese Annahme der Realität nicht gerecht wird, es zu deutlichen Unterschieden zwischen den Fehlern kommen kann und in der Regel auch kommt. Die Unterschiede im Aufwand sind zumeist darin begründet, dass unterschiedliche Teile der Software verschieden sind und auch eine unterschiedliche Komplexität besitzen.

Der Aufwand ist gleichzusetzen mit der Dauer der Arbeit W des Entwicklers, nämlich die Tests zu analysieren und den Fehler einzugrenzen.

Da die mögliche Quantifizierung der Teilarbeiten, die die Dauer des Aufwandes verursachen, den Rahmen dieser Arbeit überschreiten würde, wird dieser Aufwand auf das intuitive Maß *Zeit* reduziert.

2. **Minimaler Aufwand**

   Der minimale Aufwand $W_{minimal}$ ist der Aufwand, der im Vergleich aller untersuchten fehlgeschlagenen Tests in der kürzesten Zeit zur Eingrenzung des Fehlers geführt hat.

3. **Maximaler Aufwand**

   Der maximale Aufwand $W_{maximal}$ ist der Aufwand, der im Vergleich aller untersuchten fehlgeschlagenen Tests die längste Zeit zur Eingrenzung des Fehlers beansprucht.

Die Bewertung der genannten Strategien erfolgt durch ihren Vergleich. Dabei werden die Strategien hinsichtlich ihrer Wahrscheinlichkeiten, dass ein Test mit $W_{minimal}$ zur Analyse herausgegriffen wird, verglichen.

$$
\begin{aligned}
n &= \text{Anzahl fehlgeschlagener Tests} \\
k &= \text{Anzahl der Tests mit } W_{minimal} \\
\binom{n}{k} &= \text{Wahrscheinlichkeit, einen von } k \text{ Tests mit } W_{minimal} \\
&\quad \text{aus } n \text{ fehlgeschlagenen Tests herauszugreifen} \\
\gamma &= \text{Wichtung der Erfahrung } \{\gamma \, \epsilon \, \mathbb{R} \mid 0 \leq \gamma \leq 1\} \\
F_{Ausschluss} &= \text{Anzahl der aufgrund von Erfahrung} \\
&\quad \text{ausgeschlossenen Tests}
\end{aligned}
$$

In Abbildung 3.3, S.35, sind die beiden Strategien hinsichtlich ihres Aufwandes zur Fehlereingrenzung vergleichend dargestellt. Es läßt sich somit die Aussage ableiten, dass aufgrund von $max(\Delta W_{sequentiell}) \geq 0$ $max(W_{erfahrungsbasiert}) \leq max(W_{sequentiell})$ gelten kann. Im schlechtesten Fall ist keine Erfahrung vorhanden ($\gamma F_{Ausschluss} = 0$), wodurch sich der Aufwand nicht reduziert und gleich dem Aufwand der sequentiellen Vorgehenswei-

se ($W_{erfahrungsbasiert} = W_{sequentiell}$) ist. Der Aufwand wird somit nie größer und dadurch können auch keine Nachteile auftreten.

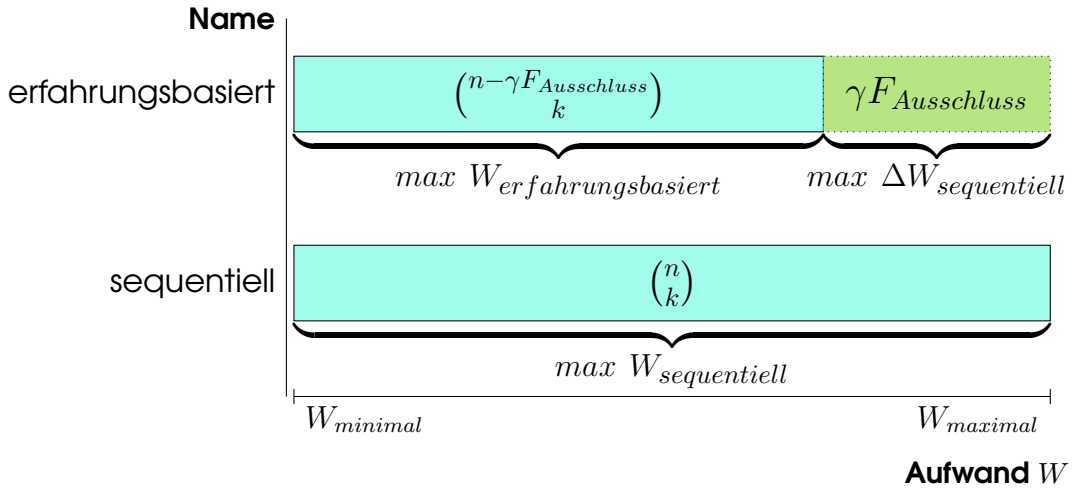


Abbildung 3.3: Vergleich des Aufwandes zur Fehlereingrenzung der beiden Strategien.

Das Problem bei der Entscheidung für eine dieser Verfahrensweisen besteht darin, dass die angewandte Strategie entweder einem primitiven Ansatz folgt (*sequentiell*) oder aber einer Heuristik zugrunde liegt (*erfahrungsbasiert*). Daraus folgt, dass beide Verfahren hinsichtlich ihres Aufwandes nicht optimal sind.

Nach Meinung des Verfassers ließe sich durch die Minimierung der Risiken während der Filterung eine signifikante Effizienzsteigerung erzielen. Vorschläge dazu werden im folgenden Kapitel gemacht (Tabelle 5.1, S.55) und in einer Untersuchung (Kapitel 7, S.67) erprobt.

4 Evaluation und Implementierung

In diesem Kapitel werden die Evaluation und die technische Implementierung beschrieben. Zunächst wird die Evaluationsphase der ursprünglichen Idee dargestellt. Danach wird die Implementierung des finalen Prototyps beschrieben.

4.1 Evaluationsphase

Der ursprüngliche Ansatz für die heuristische Fehlereingrenzung bestand darin, die Tests zu kategorisieren und daraus eine Ordnung zu entwickeln, die es dem Entwickler erleichtert, Fehler zu finden. Dabei sollten die Tests in Kategorien eingeteilt werden, so z.B. in *Unit-*, *Integrations-* und *Systemtests* (Kapitel 2.2, S.24). Aufgrund der Tatsache, dass diese Kategorien – zumindest in der Theorie – in einer hierarchischen Beziehung zueinander stehen und klar voneinander abzugrenzen sind, sollte die effizienteste Untersuchungsreihenfolge abgeleitet werden können. Diese Klassifizierung erschien durch die Wichtung von Metriken (u.a. Zyklomatische Komplexität, Umfang usw.) möglich.

Von dem in Kapitel 2.3, S.26 beschriebenen Modell ausgehend, haben erste Analysen mehrerer öffentlicher Opensource-Projekte (Apache Commons, Apache ActiveMQ, Apache Camel u.v.m.) ergeben, dass jedoch keine Anhaltspunkte für ein Gelingen dieses Ansatzes zu finden sind. Die Sichtung der Quelltexte der Tests ließ zwar die Schlußfolgerung zu, dass die Arten der Tests im Rahmen des Projektes einigermaßen klar unterschieden werden können. Allerdings scheint eine Verallgemeinerung dieser Erkenntnisse aufgrund der Unterschiedlichkeit der Tests und der ihnen zugrunde liegenden Projekte nicht zulässig. Hierbei könnten möglicherweise nur verfeinerte Data-mining Verfahren weiterhelfen.

Die Alternative dazu ist nämlich, die globale Klassifizierung von Tests auf eine Analyse lokaler Tests zu reduzieren. Dies erschien wesentlich erfolgversprechender. Daher wurde die Interpretation der Messergebnisse des im Folgenden beschriebenen Prototypen nicht mehr für die Klassifizierung der Testarten, sondern für die Sortierung der Tests benutzt.

4.2 Implementierung des Prototypen

Bei der Implementierung des Prototypen galt es, zunächst zwei grundsätzliche Fragen zu beantworten:

1. Gibt es eine Plattform, die, ähnlich einem Framework, eine infrastrukturelle Grundlage für weitere Analysen bietet?
2. Lassen sich die fachlichen Aufgaben mit Hilfe einer passenden Infrastruktur lösen?

Die Antwort auf die erste Frage heißt *Sonar*[21]. Mit Sonar als quelltextzentrierte Plattform zur Verwaltung der Qualität[22] wurde ein geeignetes Werkzeug als infrastrukturelle Grundlage gefunden. Darauf aufbauend verwendet die Referenzimplementierung eine Adaption des *JavaRunners*[23]. Damit ist es möglich, eigene Erweiterungen in den Analyseprozess einzubinden mit dem Ziel, die folgenden fachlichen Aufgaben zu lösen:

- Berechnung eigener Metriken,
- Erzeugung einer für diese Arbeit notwendigen Ausgabe,
- Erstellung von Adaptern für die Integration in gängige Testframeworks[24].

Im Folgenden wird *Sonar* als Werkzeug und Plattform vorgestellt. Danach wird die Lösung der fachlichen Aufgaben beschrieben.

[21] http://www.sonarsource.org/
[22] Gemeint sind noch bestehende Schwachstellen, aber auch bereits erreichte Fortschritte.
[23] Dabei handelt es sich um ein eigenständiges Programm.
[24] Dazu werden Junit und TestNG gezählt.

4.2.1 Sonar

Für den Zugriff auf den Quelltext und die Umsetzung der neuen Metrik *Typdistanz* (Kapitel 2.1.7, S.19) kommt die Plattform *Sonar* zum Einsatz.

Sonar beschreibt sich selbst als offene Plattform zur Verwaltung der Qualität des Quelltextes[25] (Son12).

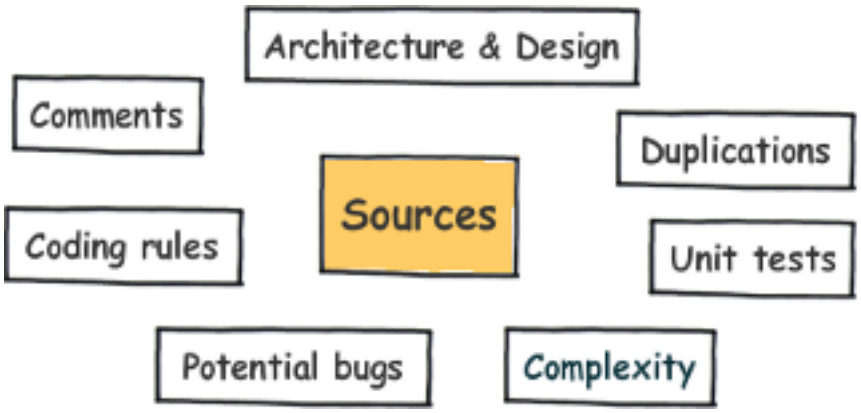


Abbildung 4.1: Darstellung der sieben Achsen der Quelltextqualität, auf die sich Sonar konzentriert(Son12).

Sonar ist als quelloffene Plattform konzipiert, die in Java geschrieben ist und sich mit Plugins erweitern läßt. Das Augenmerk liegt auf der technischen Qualitätskontrolle. So werden u.a. folgende Bereiche abgedeckt, wie aus Abbildung 4.1 hervorgeht:

- Hinweise zur Softwarearchitektur und zum Softwaredesign,
- Erkennung von wiederholten Quelltextblöcken,
- Einbindung von Modultests,
- Erkennung von potentiellen Fehlern,
- Einhaltung von Kodierrichtlinien,
- Analyse von Kommentaren,
- Bestimmung der Komplexität.

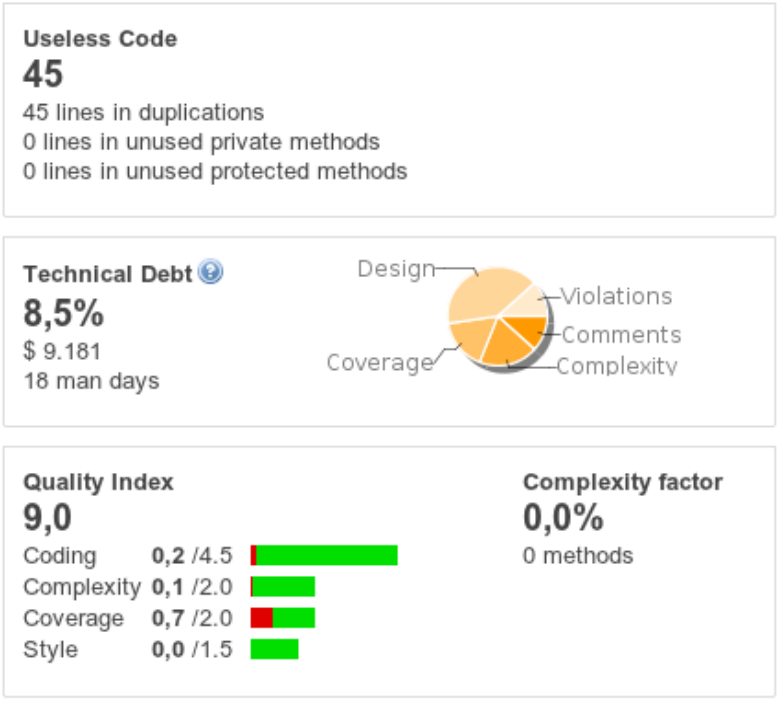


Abbildung 4.2: Darstellung einzelner Analyseergebnisse in der Sonar-Weboberfläche.

Neben Java wird die Analyse auch für weitere Programmiersprachen unterstützt[26]. Die Darstellung der Analysen und Metriken erfolgt über die mitgelieferte Weboberfläche (Abbildung 4.2). Alternativ dazu lassen sich auch viele Detailinformationen über die verfügbare REST-Schnittstelle bzw. durch die Nutzung der Softwarebibliotheken erhalten (Liste der von Sonar unterstützten Metriken im Anhang B, S.99). In der für diese Arbeit erstellten Referenzimplementierung erfolgt die Nutzung der Softwarebibliotheken.

Die Quelltextanalyse mit *Sonar* kann durch einen Ant-Task, ein Maven-Plugin oder einen sogenannten *Java-Runner* gestartet werden.

[25] „(...) open platform to manage code quality“ (Son12).
[26] Es werden weiterhin C, C#, Flex, Natural, PHP, PL/SQL, Cobol und Visual Basic6 unterstützt.

4.2.2 Definition neuer Metriken

Die Berechnung eigener Metriken ist für eine Evaluation sehr wichtig. Aufgrund der umfangreichen Quelltextmenge ist die Evaluation nur durch softwaretechnische Werkzeuge möglich. Daher war es am Anfang die Aufgabe, mit Hilfe von Sonar eigene Metriken zu definieren und für die Analyse zu verwenden.

Eigene Metriken können auf zwei Arten erstellt werden. Einerseits ist es möglich, bereits vorhandene Metriken zu kombinieren, andererseits können durch die Analyse des Abstrakten Syntax Baums[27] auch neue Basismetriken definiert werden.

In der Referenzimplementierung werden eigene Basismetriken definiert, die in Sonar mit Hilfe der Schnittstelle $JavaAstVisitor$ implementiert werden müssen.

In Listing 4.1 ist ein Beispiel für die Implementierung einer Metrik dargestellt, die die Anzahl der $Assert.assertXY$(-Anweisungen) im Quelltext zählt. Dieses Beispiel läßt die Komplexität im Umgang mit dem AST erahnen. Allerdings wird auch deutlich, wie gering der Aufwand mit Hilfe von Sonar ist.

Listing 4.1: Beispiel für die Implementierung der Schnittstelle *JavaAstVisitor*.

```
1  public class AssertCountVisitor extends JavaAstVisitor{
2    public static final List<Integer> WANTED_TOKENS
3          = Arrays.asList(TokenTypes.METHOD_CALL);
4    private final Registry            registry;
5    private AstHelper                 astHelper     = null;
6    private String                    keyFile;
7    private final JavaAstVisitorHelper      visitorHelper;
8
9    public AssertCountVisitor(final Registry registry)  {
10     this.registry = registry;
11     visitorHelper = new JavaAstVisitorHelper(this);
```

[27] Interne Repräsentation des Quelltextes als abstrakte Syntax in Form eines Baums (Krö02). Wird auch mit AST abgekürzt.

```
12    }
13
14    /*
15     * (non-Javadoc)
16     * @see JavaAstVisitor#getWantedTokens()
17     */
18    @Override
19    public List<Integer> getWantedTokens()  {
20      return WANTED_TOKENS;
21    }
22
23    /*
24     * (non-Javadoc)
25     * @see JavaAstVisitor#visitFile(DetailAST)
26     */
27    @Override
28    public void visitFile(final DetailAST ast)  {
29      File file = new File(getFileContents().getFilename());
30      keyFile = file.getAbsolutePath();
31      astHelper = new AstHelper(registry, keyFile, this);
32      visitorHelper.setAsthelper(astHelper);
33    }
34
35    /*
36     * (non-Javadoc)
37     * @see JavaAstVisitor#visitToken(DetailAST)
38     */
39    @Override
40    public void visitToken(final DetailAST ast)  {
41      SourceCode res = peekSourceCode();
42      VisitorKey visitorKey = visitorHelper.getVisitorKey(ast, res);
43
44      List<String> typeList = new ArrayList<String>();
45      if ((ast.getFirstChild() != null) &&
46          (ast.getFirstChild().getType() == TokenTypes.DOT))    {
47        astHelper.findCompleteDottedType(ast.getFirstChild(), typeList);
48      }
```

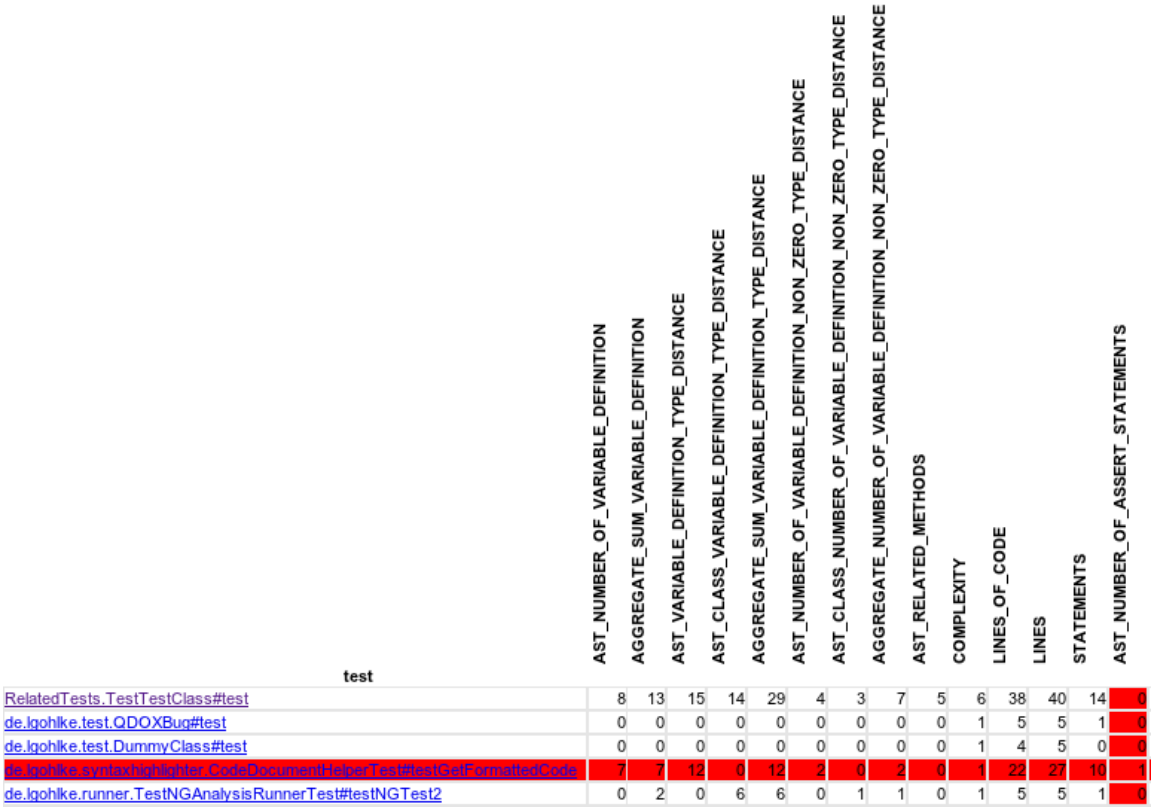
```
49      else    {
50        typeList.add(ast.getText());
51      }
52
53      if (typeList.size() > 0)    {
54        String lastName = typeList.get(typeList.size() - 1);
55        if (lastName.toLowerCase().contains("assert"))      {
56          ASTMetrics metric = ASTMetrics.AST_NUMBER_OF_ASSERT_STATEMENTS;
57          res.add(metric, 1);
58          registry.addMetricPerLine(visitorKey, metric, 1D);
59          registry.addDescription(visitorKey, metric,
60                  "found an assert-method-call");
61        }
62      }
63    }
64
65    /*
66     * (non-Javadoc)
67     * @see JavaAstVisitor#leaveToken(DetailAST)
68     */
69    @Override
70    public void leaveToken(final DetailAST ast){}
71  }
```

4.2.3 Erzeugung der Ausgabe

Die Ausgabe der Analyseergebnisse kann auf zwei Arten erfolgen:

Zum Einen kann sie als Darstellung der Maßzahlen mit den betreffenden Quelltextausschnitten für die Analyse der Metriken und deren Berechnungsgrundlage (Abbildung 4.3, S.44 und Anhang C, S.107) umgesetzt werden. Dafür wird eine Ausgabe benötigt, die einen Blick sowohl auf den Quelltext, als auch auf die Metriken ermöglicht.

Zum Anderen kann die Ausgabe als einfache sortierte Liste der fehlgeschlagenen Tests in einer Datei (Tabelle 4.1, S.44) erfolgen. Diese dient dem Entwickler als Werkzeug bei der Fehlereingrenzung.



test	AST_NUMBER_OF_VARIABLE_DEFINITION	AGGREGATE_SUM_VARIABLE_DEFINITION	AST_VARIABLE_DEFINITION_TYPE_DISTANCE	AST_CLASS_VARIABLE_DEFINITION_TYPE_DISTANCE	AGGREGATE_SUM_VARIABLE_DEFINITION_TYPE_DISTANCE	AST_NUMBER_OF_VARIABLE_DEFINITION_NON_ZERO_TYPE_DISTANCE	AST_CLASS_NUMBER_OF_VARIABLE_DEFINITION_NON_ZERO_TYPE_DISTANCE	AGGREGATE_NUMBER_OF_VARIABLE_DEFINITION_NON_ZERO_TYPE_DISTANCE	AST_RELATED_METHODS	COMPLEXITY	LINES_OF_CODE	LINES	STATEMENTS	AST_NUMBER_OF_ASSERT_STATEMENTS
RelatedTests.TestTestClass#test	8	13	15	14	29	4	3	7	5	6	38	40	14	0
de.lgohlke.test.QDOXBug#test	0	0	0	0	0	0	0	0	0	1	5	5	1	0
de.lgohlke.test.DummyClass#test	0	0	0	0	0	0	0	0	0	1	4	5	0	0
de.lgohlke.syntaxhighlighter.CodeDocumentHelperTest#testGetFormattedCode	7	7	12	0	12	2	0	2	0	1	22	27	10	1
de.lgohlke.runner.TestNGAnalysisRunnerTest#testNGTest2	0	2	0	6	6	0	1	1	0	1	5	5	1	0

Abbildung 4.3: Beispiel der Analyseübersicht für die Entwicklung.

Name	Score
net.sprd.qa.prototype.APIServiceTest#validateToken	201013
net.sprd.qa.prototype.APIServiceTest#invalidateToken	201013
net.sprd.qa.prototype.APIServiceTest#userNotExisting	202010
net.sprd.qa.prototype.APIServiceTest#removeUserNotExisting	202010
net.sprd.qa.prototype.Bus.BusConnectorTest#test	301008
net.sprd.qa.prototype.Bus.BusServiceTest#smokeTest	904506
net.sprd.qa.prototype.Bus.BusServiceTest#roundtrip	1001018

Tabelle 4.1: Beispiel einer Ausgabe der priorisierten Testliste am Ende eines Testlaufes mit Fehlern.

4.2.4 Integration

Am Ende des Testlaufs werden die fehlgeschlagenen Tests untersucht (Listing 4.2). Sollten Tests fehlschlagen, kann eine Analyse die fehlgeschlagenen priorisieren. Diese wird nur im Fehlerfall durchgeführt und entsprechend aufbereitet (Tabelle 4.1, S.44).

Listing 4.2: Auf das Wesentliche reduziertes Beispiel für die Integration in den Lebenszyklus von TestNG.

```
1  public class TestNGAnalysisRunner implements ITestListener{
2
3    @Override
4    public void onFinish(final ITestContext context)  {
5      if (failedTests.size() > 1)
6        runPostTestAnalysis();
7    }
8
9    private void runPostTestAnalysis(){
10     File directoryToAnalyse = new File(System.getProperty("user.dir"));
11     AnalysisTestFilter analysisTestFilter = new AnalysisTestFilter(TEST_TYPE
12       analysisTestFilter.setFilter(this);
13
14       new MetricAnalyzer().//
15           addDirectoryForAnalysis(directoryToAnalyse).//
16           writeTo(outputDir).//
17           filter(analysisTestFilter).//
18           analyze();
19   }
20 }
```

4.3 Typdistanz

Die Typdistanz, wie in Kapitel 2.1.7, S.19 dargelegt, wird an dem folgenden Beispiel erläutert.

Gegeben sind verschiedene Tests und die ihnen zugrunde liegende Paketstruktur (Abbildung 4.4). Zunächst wird die Paketstruktur gezeigt, die den Namenraumbaum darstellt. Anschließend werden anhand der Tests die Einzeltypdistanzen berechnet und mit den Zeilen im Test verknüpft (Listing 4.3 und Tabelle 4.2).

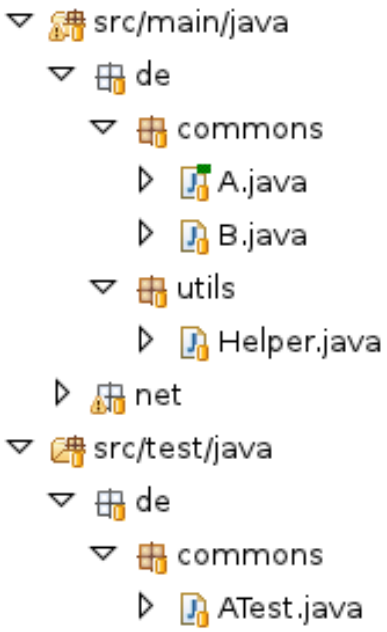


Abbildung 4.4: Darstellung eines Namenraumes mit der Unterscheidung zwischen Paketen und Klassen, zwischen Basisklassen und Testklassen (`src/main/java` und `src/test/java`).

Listing 4.3: Quelltextbeispiel für drei Tests, die sich hinsichtlich der Einzeltypdistanz unterscheiden.

```
1 public class ATest{
2   @Test public void simple(){
3     A a = new A();
4     String name = "name";
```

```
5       Assert.assertEquals(a, a);
6       Assert.assertFalse(a.equals(name));
7     }
8
9     @Test public void withB(){
10      A a = new A();
11      B b = new B();
12      Assert.assertFalse(a.equals(b));
13    }
14
15    @Test public void withBandHelper(){
16      A a = new A();
17      B b = new B();
18      Helper helper = new Helper();
19      helper.doSthWith(a, b);
20      Assert.assertFalse(a.equals(b));
21    }
22  }
```

Mit diesen Tests erhält man die in Tabelle 4.2 zeilenweise errechneten Einzeltypdistanzen.

Test	Zeile	Einzeltypdistanz
simple	13	$d(de.commons.ATest, de.commons.A) = 1$
	14	$d(de.commons.ATest, lang.java.String) = 0$
withB	22	$d(de.commons.ATest, de.commons.A) = 1$
	23	$d(de.commons.ATest, de.commons.B) = 1$
withBandHelper	30	$d(de.commons.ATest, de.commons.A) = 1$
	31	$d(de.commons.ATest, de.commons.B) = 1$
	32	$d(de.commons.ATest, de.utils.Helper) = 5$

Tabelle 4.2: Übersicht der den Tests zugeordneten Einzeltypdistanzen.

5 Weiterentwicklung der Methodik

Nachdem im vorangegangenen Kapitel die Evaluation und Implementierung im Mittelpunkt standen, wird an jetzt auf die Weiterentwicklung der Methode der statischen Fehlereingrenzung eingegangen.

Die im Kapitel 3, S.29 besprochenen Fehlereingrenzungsstrategien bergen Risiken (Kapitel 3.2.2, S.32), deren wirksame Verminderung das Ziel ist. Die beabsichtigte Reduzierung der Risiken soll den Prozess der Fehlerbehandlung (Kapitel 2, Abbildung 2.9, S.27) und damit den Prozesschritt der *Filterung* aufwandsstabiler gestalten.

In diesem Kapitel werden Lösungsansätze angeboten, die das Fehler-Rauschen in Folge der Fehlerfortpflanzung (Kapitel 2.1.8, S.21) für den Entwickler vermindern und so den Aufwand zur Lokalisierung des signifikantesten Tests reduzieren.

5.1 Einordnung im Fehlerbehandlungsprozess

Der Prozess der Fehlerbehandlung und die dabei einzuhaltende Reihenfolge der Schritte wurden bereits in Kapitel 2.4, S.27 dargelegt.

1. Die Fehlerdetektion, erster Schritt des Fehlerbehandlungsprozesses, zeigt zwar Fehler auf, es bleibt jedoch weiterhin ungeklärt, wo sich in der Liste der fehlgeschlagenen Tests der Test befindet, der dem Fehler am nächsten kommt.

2. Im zweiten Schritt des Fehlerbehandlungsprozesses werden die Fehler analysiert.

   Ab diesem Zeitpunkt muß der Entwickler entscheiden, wie er den Programmfehler zu lokalisieren beabsichtigt. Wenn man davon aus-

geht, dass es sich um keinen *trivialen*[28] Fehler handelt, sondern um einen komplexen[29], der ein störendes Fehlerrauschen erzeugt, wird ersichtlich, wie groß der Aufwand sein wird, um diesen einzugrenzen.

Im Gegensatz zu den Verfahren der voll- bzw. semiautomatischen Fehlerisolation beschäftigt sich diese Arbeit mit einem werkzeugunterstützten manuellen Ansatz:

a) Die *Filterung* der fehlgeschlagenen Tests soll die Lokalisierung der Fehlerursachen erleichtern.

b) Die *Lokalisierung* wird durch die Reduzierung der Fehlermenge in der vorausgegangenen Filterung vereinfacht.

3. Die nachfolgenden Schritte beinhalten die Bewertung und Fehlerbehebung.

5.2 Bedeutung der Filterung während der Fehleranalyse

Es wird im Folgenden noch einmal auf den Stellenwert des Filterungsschrittes innerhalb der Fehleranalyse im Fehlerbehandlungsprozess eingegangen. Erfolgt die Entwicklung nach dem Prinzip der *Kontinuierlichen Integration* (Kapitel 2.1.11, S.23), so kann aufgrund der zeitlichen Nähe zwischen dem erstmaligen Auftreten eines Fehlers und den entsprechenden Änderungen am Quelltext der Software ziemlich genau bestimmt werden, welche Einflüsse den Fehler hervorgerufen haben. Mit diesem Wissen läßt sich das Umfeld, das den Fehler verursacht hat, relativ gut eingrenzen.

[28] Dazu zählen Fehler, die leicht erkennbar sind, u.a. Vertauschungen, Verschreibungen, Weglassungen, Syntaxfehler. Sie lassen sich ohne viel Aufwand beheben und erfordern kein tieferes Veständnis des Produktes.

[29] Fehler, die ein umfangreiches Verständnis der fachlichen Logik voraussetzen, u.a. Seiteneffekte, Parallelität, Abhängigkeitsstrukturen.

Allerdings kann es vorkommen, dass Änderungen, die den Fehler hervorrufen, keine oder nur eine geringe Aussagekraft besitzen. Damit wäre nicht klar erkennbar, wie der Fehler[30] effizient eingegrenzt werden kann.

Besonderheiten, welche die Bedeutung des Filterungsschrittes innerhalb der Fehleranalyse im Fehlerbehandlungsprozess verdeutlichen und die zeitliche Nähe als Vorteil aufheben, wären u.a.:

1. **Umfangreiches Refactoring**

   Refactoring[31] wird nach (Fow) als *„a series of small behavior preserving transformations"* beschrieben, allerdings kommt es vor, dass dabei zu viele Bestandteile verändert wurden.

   Gründe, die die zeitliche Nähe als Vorteil für diese agile Methode (Wel09) aufheben, können sein:

   - Das Intervall, in dem Änderungen mit einem Testlauf geprüft werden, ist zu groß. Ursache dafür könnte sein, dass nicht regelmäßig oder in zu großen Abständen integriert wird.
     Durch die Anwendung des *Kontinuierlichen Testens*[32] könnten jedoch eventuelle Probleme schon frühzeitig erkannt werden.
   - Die Änderungen sind zu umfangreich, um aus dem Testlauf und der Veränderung zur vorletzten Version direkt auf die Fehlerursache schließen zu können.
     Wie aus dem Prinzip *Kontinuierliche Integration* allgemein bekannt (Kapitel 2.1.11, S.23), könnte diesem Problem einerseits dadurch begegnet werden, dass man die Größe der Änderungen, die dem Versionskontrollsystem übergeben werden, reduziert. Es könnte aber auch andererseits der Einsatz von Werkzeugen helfen, die den Entwickler nach dem Prinzip des

[30] Der Singular wird an dieser Stelle auch für die Möglichkeit des Auftretens mehrerer Fehler verwendet.

[31] „Der Begriff *Refactoring* bezeichnet die Erneuerung der inneren Strukturen eines Software-Systems, ohne sein äußeres Verhalten zu verändern." (Hof08, S.396).

[32] Nach (Saf06) wird dieses Prinzip folgendermaßen erklärt: „Continuous testing uses excess cycles on a developer's workstation to continuously run regression tests in the background, providing rapid feedback about test failures as source code is edited". Der Verfasser kann aus eigener Erfahrung berichten, dass das Arbeiten nach diesem Prinzip einen großen Gewinn darstellt.

Kontinuierlichen Testens während seiner Arbeit unterstützen und damit Fehler sofort aufzeigen.

2. **Lange Laufzeit der Tests**

   Bei einem Testlauf, der aufgrund seiner Komplexität[33] seltener durchgeführt wird als Änderungen an der zu testenden Software vorgenommen werden, lassen sich einzelne Änderungen nicht mehr eindeutig Fehlern zuordnen. Das ist häufig bei Integrationstests (Kapitel 2.2.2, S.25) und Systemtests (Kapitel 2.2.3, S.26) der Fall (Duv07; Fow06).

   Ein Beispiel verdeutlicht dies: Es gibt eine Testsuite[34], die 100 Tests umfasst. Die Laufzeit dieser Testsuite beträgt 20 Minuten. Werden nun während eines Testlaufs mehrere Änderungen eingecheckt, weil man parallel daran arbeitet, so wird bei anschließenden Testläufen mehr als eine Änderung einbezogen. Somit ist im Fehlerfall unklar, welche der Änderungen den Fehler verursacht hat.

   Formal kann festgestellt werden, dass die Abtastrate $f_{\text{Änderungsprüfung}}$, mit welcher Änderungen detektiert werden können, gegenüber der Änderungsrate f_{checkin} zu niedrig ist. Ausserdem setzt während der Laufzeit der Testsuite T_{Laufzeit} die Änderungsprüfung aus, also ist $f_{\text{Änderungsprüfung}} = 0$ (Abbildung 5.1).

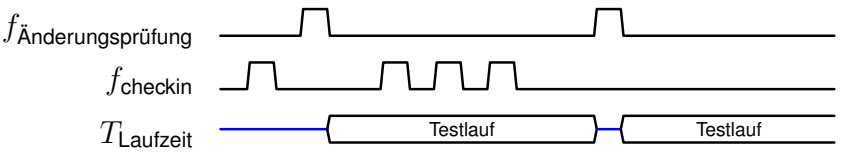


Abbildung 5.1: Schematische Darstellung der zeitlichen Abfolge während der Laufzeit der Testsuite.

[33] Damit sind der Aufwand bzw. die Kosten für die Ausführung eines Tests gemeint. Das kann z.B dann der Fall sein, wenn es sich um Tests handelt, die viele Ressourcen benötigen, wie z.B. Integrations- bzw. Oberflächentests.

[34] Eine Testsuite ist eine Sammlung von Tests.

An diesen Beispielen zeigt sich sehr deutlich der Nachteil einer fehlenden systematischen Fehlerlokalisierung.

5.3 Weiterentwicklung bekannter Fehlereingrenzungsstrategien

Trotz sehr hohen manuellen Anteils während des gesamten Prozesses der Fehlerbehandlung kann es für den Entwickler vorteilhaft sein, wenn bei der Filterung eine Unterstützung im Rahmen der iterativen Entwicklung (Kapitel 2.1.11, S.23) erfolgt. Dabei hat die Sortierreihenfolge der fehlgeschlagenen Tests eine besondere Bedeutung, wie im Weiteren noch gezeigt wird.

Die weiterentwickelte Fehlereingrenzungsstrategie wird als *bottom-up*[35] bezeichnet. Sie entwickelt den Schritt der Filterung aus der bekannten *erfahrungsbasierten* Strategie weiter (Kapitel 3.2.2, S.32).

Die Idee dieser Strategie besteht darin, zunächst den *kleinsten Test* in der Abhängigkeitsstruktur der Tests zu untersuchen (Kapitel 2.1.9, S.22). Mit der Annahme, dass der *kleinste Test* der Fehlerursache am nächsten ist, entfällt ein zusätzliches Überprüfen der *größeren* Tests.

Dieser Teil der Fehlereingrenzungsstrategie umfasst zwei Arbeitsschritte:

1. **Sortierung der Fehler**

   Die Sortierung der Fehler verfolgt das Ziel, anhand geeigneter Kriterien mit dem kleinsten Fehler die Fehlereingrenzung zu beginnen, um auf diese Weise den Aufwand zu minimieren. Diese Sortierung erfolgt idealerweise maschinell am Ende des Testlaufs und ist so optimiert, dass der Aufwand vernachlässigt werden darf.

2. **Analyse der Fehler**

   Die Analyse der kleinsten Tests macht weiterhin manuelle Arbeitsschritte notwendig und bedarf der eingehenden Expertise eines

[35] In Anlehnung an bekannte allgemeine Vorgehensweisen, u.a. in der Informatik.

Entwicklers. Die zeitaufwändige Analyse kann durch den Einsatz von *Delta-Debugging*[36] unterstützt werden. Damit ist es möglich, den Fehler u.U. automatisch einzugrenzen.

Mit der Aufteilung der Fehlereingrenzung in einen automatischen und manuellen Schritt verringert sich der Aufwand W im ersten Schritt bereits um den automatisierten Teil[37]. Im zweiten Schritt reduziert sich der Aufwand erneut um die Tests, die aufgrund von Erfahrungen als weniger relevant angesehen werden. Damit vergrößert sich die Wahrscheinlichkeit, einen *kleinsten Fehler* aus der reduzierten Menge der fehlgeschlagenen Tests herauszunehmen, erheblich. Dieser Aufwand wird in Abbildung 5.2, S.55, vergleichend dargestellt.

n	$=$	Anzahl fehlgeschlagener Tests
k	$=$	Anzahl kleinster Fehler
$\binom{n}{k}$	$=$	Wahrscheinlichkeit, einen von k kleinsten Fehler aus n fehlgeschlagenen Tests herauszugreifen
γ	$=$	Wichtung der Erfahrung $\{\gamma \epsilon \mathbb{R} \mid 0 \leq \gamma \leq 1\}$
$F_{Ausschluss}$	$=$	Anzahl der aufgrund von Erfahrung ausgeschlossenen Tests ($0 \leq F_{Ausschluss}$)
$\overline{W}$	$=$	eingesparter Aufwand
α	$=$	Wichtung der Qualität der Filterung $\{\alpha \epsilon \mathbb{R} \mid 0 \leq \alpha \leq 1\}$
F_{gross}	$=$	Anzahl herausgefilterter Tests ($0 \leq F_{gross}$)

Anhand der Abbildung 5.2, S.55, wird deutlich, auf welche Weise sich der Aufwand zur Fehlerlokalisierung weiter reduzieren läßt. Das Problem bei Anwendung dieser weiterentwickelten Strategie (bottom-up) besteht allerdings darin, geeignete Kriterien für eine vorbereitende Sortierung zu bestimmen. Die Richtigkeit der Reihenfolge entscheidet über die Effizienz der nachfolgenden Analyse. In Tabelle 5.1, S.55 wird auf Einschränkungen in der derzeitigen Implementierung hingewiesen.

[36] „In general, conventional debugging strategies lead to faster results. However, delta debugging becomes an alternative when the differences can be narrowed down automatically“ (Zel99, S.253).

[37] Es wird davon ausgegangen, dass der automatisierte Teil aus der Aufwandsbetrachtung herausgenommen werden kann, weil er mit Hilfe von Werkzeugen erfolgt und am Ende eines Testlaufs als Phase betrachtet wird.

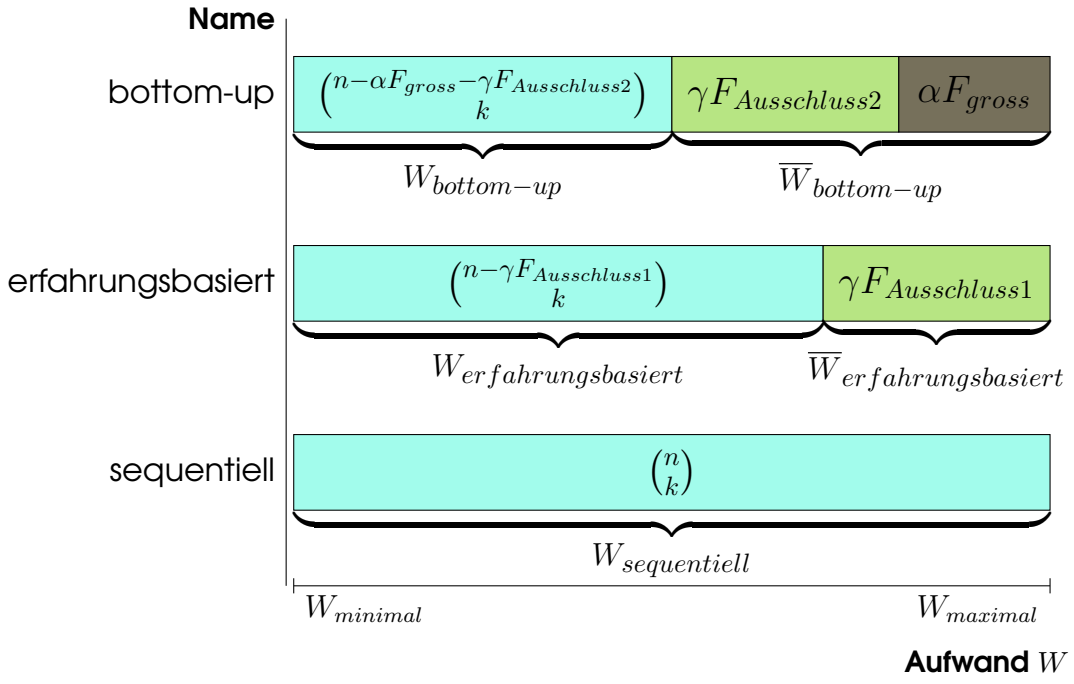


Abbildung 5.2: Vergleich des zeitlichen Aufwandes unter Einbeziehung der neuen Strategie.

Strategie	Schwächen (Risiken)	Stärken (Vorteile)	aufwandsstabil[38]
sequentiell	Reihenfolge in der Fehlerliste	einfach	$\ominus$
erfahrungsbasiert	fehlerhafte Filterung, Erfahrungsmangel	nutzt Erfahrung	$neutral$
bottom-up	Kriterien für Sortierung, Werkzeugunterstützung notwendig	geringster Aufwand	$\oplus$

Tabelle 5.1: Zusammenfassung der bekannten Fehlereingrenzungsstrategien mit dem neuen Ansatz *bottom-up*.

[38] Der Aufwand bleibt bei der wiederholten Ausführung gleich. Dabei wird die Beeinflussung durch randomisierte Testausführung, Verwürfelungen durch parallele Testläufe und wechselnde Entwickler berücksichtigt.

Ein Ansatz zur Bestimmung geeigneter Kriterien für eine vorbereitende Testsortierung wird im folgenden Kapitel erläutert.

5.4 Berechnungsvorschrift der Gesamttypdistanz

Die aktuelle Berechnungsvorschrift der Gesamttypdistanz eines Tests berücksichtigt drei Faktoren, die im Weiteren entsprechend ihrer Wichtung beschrieben werden:

1. **Maximale Einzeltypdistanz**

   Aufgrund des zentralen Kriteriums der Typdistanz wird das Maximum aller vorhandenen Einzeltypdistanzen $M_{distance_{max}}$ am höchsten gewertet, $M_{distance_{max}} \in \mathbb{N}_0$.

2. **Median der Einzeltypdistanzen**

   Für eine robuste Ermittlung der Gesamttypdistanz wird der Median $\tilde{M}_{distance}$ berechnet. Somit würde bei einer Gleichheit der maximalen Einzeltypdistanzen der Test höher gewichtet, dessen Median größer ist, $\tilde{M}_{distance} \in \mathbb{R}_0^+$.

3. **Anzahl der Anweisungen**

   Sind die unter 1.) und 2.) genannten Wichtungsfaktoren identisch, ist die einfache Wichtung nach Umfang der Anweisungen mit $|S|$, $|S| \in \mathbb{N}_0$ ausreichend.

Somit wird das Wichtungsmaß $Score_{FailedTests} \in \mathbb{N}_0$ wie folgt berechnet:

$$Score_{FailedTests} = M_{distance_{max}} * 10^5 + \tilde{M}_{distance} * 10^3 + |S|$$

Durch die abgrenzende Wichtung zwischen $M_{distance_{max}}$ und $\tilde{M}_{distance}$ um 10^2 findet keine Überlappung und Beeinflussung statt[39]. Ebenso verhält es sich mit $\tilde{M}_{distance}$ und $|S|$. Zwischen diesen beiden Faktoren wurde ein Abstand von 10^3 gewählt, um so den Nachkommastellen von $\tilde{M}_{distance}$ eine gewisse Bedeutung beizumessen.

[39] Bis zu einem Wert von 100 ist die Überlappung ausgeschlossen, wobei deren Bedeutung bei einem so hohen Medianwert der Typdistanzen ernsthaft in Frage gestellt werden kann.

Die errechneten Gesamttypdistanzen werden aufsteigend, beginnend mit den *kleinsten Tests*, sortiert.

5.5 Umfang und Abgrenzung der Analyse

Die Erfassung der Typdistanz basiert auf dem Quelltext, der dem Programm zugrunde liegt. Dieser wird in einer spezifischen Programmiersprache verfasst. Je nach Sprache gibt es unterschiedliche Möglichkeiten hinsichtlich der Nutzung von Datentypen. Da sich die Analyse in dieser Arbeit auf die statische Quellcodeanalyse stützt, beschränkt sich die konkrete Vorgehensweise auf statisch typisierte Sprachen.

Für die prototypische Implementierung wurde die Programmiersprache Java verwendet. Daher konzentriert sich die Analyse mit Hilfe dieses Prototypen auf Java-sprachige Quelltexte.

Innerhalb eines Java-Quelltextes sind mehrere Stellen für die Analyse von Typinformationen festgelegt, die für die Berechnung der Einzeltypdistanz (Kapitel 2.1.7, S.19) verwendet werden (Tabelle 5.2, S.57).

Syntaxelemente	Beispiel
Typdeklaration ...	`String x;`
... mit Generics[40]	`List<String> list;`
... und Initialisierung[41]	`List<String> list = new ArrayList<String>();`

Tabelle 5.2: Die in diesem Prototypen analysierten Syntaxelemente zur Berechnung der Einzeltypdistanz.

Neben den Syntaxelementen sind außerdem deren Geltungsbereiche zu unterscheiden. Dabei gibt es in unserem Fall den *sprachlichen* und den Geltungsbereich *per Konvention*.

[40] Bestimmt aus der Liste der Typen (*List*,*String*) die höchste Einzeltypdistanz.
[41] Ohne Berücksichtigung der instantiierten Klasse (ohne *ArrayList*).

1. **Sprachlicher Geltungsbereich**

   Der *sprachliche* Geltungsbereich wird durch die Sprachspezifikation bestimmt. Im Fall des implementierten Prototypen ist das „The Java™ Language Specification" (Gos+05). Als primärer Geltungsbereich in diesem Zusammenhang gilt der Kontext der Methode, da es in dieser Arbeit um die Vorsortierung fehlgeschlagener Tests geht, welche in Form von Methoden[42] auftreten. Außerdem darf der Klassenkontext, in dem diese Methode liegt, nicht vernachlässigt werden. Bei seiner Betrachtung muss auch die Vererbungshierarchie berücksichtigt werden. Mit der Vererbung ergeben sich Möglichkeiten, die Vererbungshierarchie von Oberklassen und implementierten Schnittstellen einzubeziehen (Abbildung 5.3, S.58).

```
 9  abstract class TestCase{
10    protected Helper helper;
11  }
12
13  public class BTest extends TestCase
14  {
15    private A a = new A();
16    private B b;
17
18    @BeforeMethod
19    public void beforeMethod()
20    {
21      b = new B();
22    }
23
24    @Test
25    public void simple()
26    {
27      String name = "name";
28      Assert.assertEquals(a, b);
29      Assert.assertFalse(a.equals(name));
30    }
31  }
```

Abbildung 5.3: Beispiel für den äußeren sprachlichen und den inneren Geltungsbereich per Konvention.

[42] Der Verfasser geht bei Java implizit von einer objektorientierten Betrachtungsweise aus, obwohl auch prozedural gearbeitet werden könnte.

2. **Geltungsbereich per Konvention**

   Der Geltungsbereich *per Konvention* wird in diesem Zusammenhang durch die Wahl des Test-Frameworks bestimmt, unterstützt werden *Junit*[43] und *TestNG*[44]. Beide Frameworks setzen zur Konfiguration des Lebenszyklus eines Tests Annotationen ein[45] (Tabelle 5.3). Daraus ergibt sich, dass außer dem eigentlichen Test ggf. weitere Methoden analysiert werden müssen (Listing 5.2, S.60).

Junit4	TestNG	Beschreibung
–	`@BeforeSuite`	wird vor allen Tests der Testsuite ausgeführt
`@BeforeClass`		wird vor allen Tests der aktuellen Klasse ausgeführt
`@Before`	`@BeforeMethod`	wird vor jeder Testmethode ausgeführt
`@Test`		ist ein Test
`@After`	`@AfterMethod`	wird nach jeder Testmethode ausgeführt
`@AfterClass`		wird nach allen Tests der aktuellen Klasse ausgeführt
–	`@AfterSuite`	wird nach allen Tests der Testsuite ausgeführt

Tabelle 5.3: Im Prototypen unterstützte Annotationen der Methoden zur Konfiguration des Lebenszyklus der Tests.

[43] http://www.junit.org
[44] http://www.testng.org
[45] Der Prototyp unterstützt Junit nur in der Version 4.

```
1 public abstract class AbstractTest{
2   @BeforeSuite
3   public void initSubSystem(){
4   }
5
6   @AfterSuite
7   public void cleanupDatabase(){
8   }
9 }
```

Listing 5.1: Quelltextbeispiel für einen Test, dessen Geltungsbereich per Konvention mehrere Methoden umfasst (abstrakte Oberklasse).

```
1  public class EmailTest extends AbstractTest implements IHandler
2  {
3     private int status = 0;
4
5    @BeforeMethod
6    public void before(){
7      Address add = new Address("test@qa.de");
8    }
9
10   @Test
11   public void testRoundTripOfMail(){
12     Address add = new Address("test@qa.de");
13   }
14
15   @AfterClass
16   public void afterClass(){
17   }
18 }
```

Listing 5.2: Quelltextbeispiel für einen Test, dessen Geltungsbereich per Konvention mehrere Methoden umfasst (Testklasse).

Im folgenden Abschnitt wird auf Beschränkungen eingegangen, denen dieser Prototyp unterliegt.

5.6 Einschränkungen

Da die Implementierung im Rahmen dieser Arbeit nur einen Prototypen darstellt, muss jedoch auf einige Einschränkungen hingewiesen werden. Folgende Restriktionen der prototypischen Berechnung der Typdistanz sind bekannt:

1. **Lokale Hilfsmethoden**

   Unterscheiden sich mehrere Tests nur hinsichtlich ihrer Parameter, gibt es eine lokale Hilfsmethode für deren Delegierung.

2. **Instantiierungen**

   Die Betrachtung der Typdeklaration ist wegen der Unterstützung der Polymorphie durch Java nicht ausreichend. Deutlich wird dies an dem letzten Beispiel, dargestellt in Tabelle 5.2, S.57. Daraus kann sich gegenüber der anfänglichen Typdeklaration eine potentiell höhere Einzeltypdistanz ergeben.

3. **Felder in Oberklassen**[46]

   Leitet sich die aktuelle Testklasse von einer anderen ab bzw. implementiert diese eine Schnittstelle, so wird deren Felddeklaration nicht berücksichtigt (Abbildung 5.3, S.58).

[46] Gilt zusammenfassend für Variable- und Konstantendeklarationen.

6 Fallbeispiel

In diesem Kapitel wird anhand eines komplexen Fallbeispiels die Ausprägung der Fehlerfortpflanzung und die Verbesserung der Methodik (Kapitel 5, S.49) durch eine automatisierte Filterung (Kapitel 5.4, S.56) erläutert.

Für eine repräsentative Anwendung nutzte der Verfasser eine ihm bekannte Architektur, die im Weiteren noch näher beschrieben werden wird. Dabei wurde diese zwecks Vereinfachung auf eine begrenzte Zahl von Anwendungsfällen reduziert. Im Folgenden werden zunächst das Anwendungsszenario, danach die Architektur und Funktionsweise der Beispielsoftware beschrieben.

Abbildung 6.1: Teil des T-Shirt-*Customizings* im Szenario.

6.1 Szenario

Dieses Fallbeispiel beinhaltet einen Webshop, der im Internet Kleidung anbietet. Dieser Webshop ist mittlerweile eine komplexe Plattform, auf der

verschiedene Dienste angeboten werden. Einer dieser Dienste besteht in der Massenbestellung bereits gestalter Kleidung, aber auch im Angebot des sehr beliebten *Customizing*[47] von T-Shirts (Abbildung 6.1, S.63).

6.2 Architektur

Die Architektur wurde zwar auf wesentliche Merkmale reduziert, aber es kann immer noch von einem hinreichend komplexen Beispiel ausgegangen werden.

Teil der Architektur ist eine private Zugriffsschicht (*API*), die das Produktanlegen und -verändern, aber auch die Authentifizierung und Verwaltung der Nutzer, bereitstellt. Der Dienst *Bus*[48] stellt die Kommunikation für die anderen beiden Dienste *API* und *Datenbank* zur Verfügung. Die *API* wird über den *Bus* mit der *Datenbank* verbunden. Die Abbildung 6.2, beinhaltet die Darstellung der Klassen.

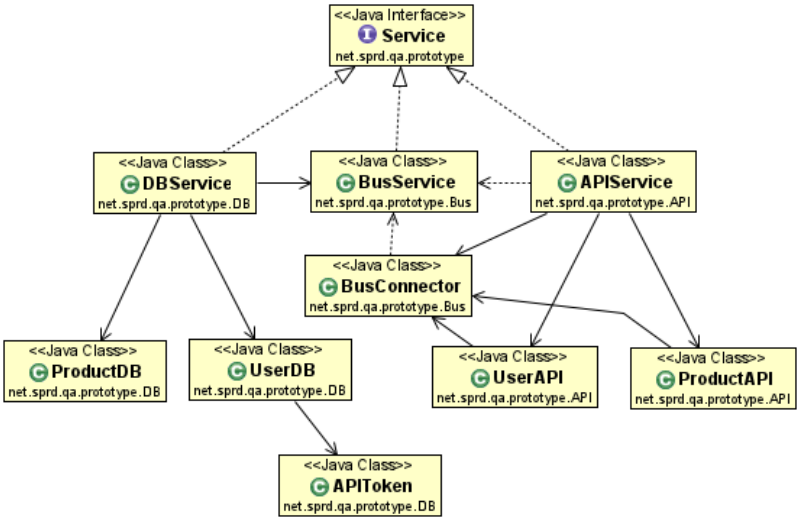


Abbildung 6.2: Klassendiagramm des Fallbeispiels.

[47] An dieser Stelle wird bewußt der Marketingbegriff kundenseitiger Gestaltung und Anpassung von Massenartikeln verwendet.

[48] „Der Enterprise Service Bus stellt das Rückgrat der SOA dar und dient als Integrationsplattform für verschiedene Dienste in einem verteilten System. Als Teil der Infrastruktur übernimmt er den Transport der Nachrichten in Form einer Middleware“ (Goh08, S.26ff).

Wichtige Eigenschaften der Architektur, die in diesem Fallbeispiel enthalten sind und explizit hervorgehoben werden, sind:

- **Mehrschichtenarchitektur**

  Es handelt sich um eine klassische 3-Schichtenarchitektur (Zugriffs-, Geschäftslogik- und Persistenzschicht) (Abbildung 6.3).

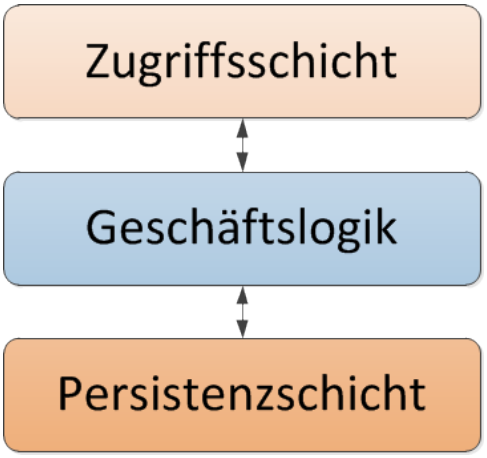


Abbildung 6.3: Darstellung einer klassischen 3-Schichtenarchitektur.

- **Serviceorientierte Architektur**

  Serviceorientierte Architektur (SOA) bezeichnet ein Architekturkonzept, das fachliche Dienste anbietet (Abbildung 6.4, S.66). Folgende Merkmale dieses Architekturkonzeptes sind enthalten: Einsatz eines Enterprise Servicebus und die lose Kopplung der Dienste (Goh08, S.26ff).

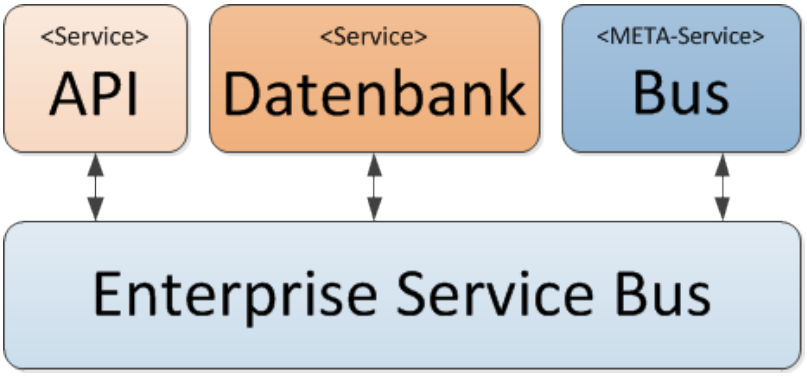


Abbildung 6.4: Darstellung als Serviceorientierte Architektur.

7 Untersuchung

Die mögliche Erhöhung der Effizienz bei der Fehlereingrenzung wird in diesem Kapitel durch eine erste Untersuchung angedeutet. Für eine abschließende Bewertung bzw. Aussage, dessen ist sich der Verfasser voll bewußt, bedarf es jedoch einer umfangreicheren Feldstudie.

Zunächst werden die Bedingungen für die Durchführung des Experiments dargelegt.

7.1 Rahmenbedingungen

Aufgrund der Tatsache, dass die Untersuchung in einem aktiven Wirtschaftsunternehmen stattfindet, sind die Rahmenbedingungen den Gegebenheiten vorort angepaßt. Das heißt, die Untersuchungen finden an den regulären Arbeitsrechnern der Probanden in ihren Räumlichkeiten statt. Somit ist jeder Proband mit dem Arbeitsplatz, einschließlich Entwicklungsumgebung und Betriebssystem, vertraut. Es bedarf auch keiner Einführung hinsichtlich der Benutzung besonderer Werkzeuge.

7.2 Konstellation

In dieser Untersuchung grenzen zwei Gruppen mit je sechs Probanden zwei Testfehler ein. Diese Fehler müssen von beiden Gruppen gefunden und korrigiert werden. Die Gruppen werden anhand der Erfahrungen ihrer Mitglieder in der Softwareentwicklung (Zeitdauer) zusammengestellt. Beide Gruppen weisen ähnliche Erfahrungswerte auf[49]. Je eine Gruppe grenzt einen

[49] Ähnliche Erfahrungswerte sind z.B. die Arbeitsjahre in der Entwicklung mit Java.

Fehler nach dem *alten (sequentiell)*, den zweiten Fehler nach dem *neuen* Verfahren *(bottom-up)* ein. Die Verfahren und Fehler werden von den Gruppen über Kreuz untersucht (Tabelle 7.1).

Fehler	Strategie der Gruppe 1	Strategie der Gruppe 2
Fehler 1	sequentiell (G1/F1)	bottom-up (G1/F2)
Fehler 2	bottom-up (G2/F1)	sequentiell (G2/F2)

Tabelle 7.1: Übersicht über die Durchführung der Untersuchung.

7.3 Ablauf

Teil 1 beinhaltet die Einweisung der Probanden, Teil 2 die Durchführung der Untersuchung.

7.3.1 Einweisung der Probanden (Teil 1)

Den Probanden wird der Rahmen der Untersuchung erläutert und ihnen eine genaue Anweisung zur Seite gestellt (Anhang A, S.81). Dabei wird auf zwei unterschiedliche Verfahren hingewiesen, die miteinander zu vergleichen sind. Den Probanden werden die beiden Verfahren nicht detailliert, sondern nur überblicksmäßig erläutert, um eine eventuelle Beeinflussung zu vermeiden.

Es wird bei der initialen Einrichtung des Versuchaufbaus geholfen. Abschließend werden die Probanden mit dem Fallbeispiel aus Kapitel 6, S.63, bekannt gemacht, welches die Grundlage für die Untersuchung bildet. Nachdem alle Fragen der Probanden beantwortet sind, wird das Startsignal gegeben, worauf diese mit der Durchführung der Untersuchung beginnen.

7.3.2 Durchführung der Untersuchung (Teil 2)

Während der Durchführung wird jegliche Kommunikation innerhalb der Gruppe untersagt, externe Störungen[50] werden vermieden. Für jeden Testlauf eines Probanden mit einem Fehler wird eine Bearbeitungsdauer von mindestens 15 Minuten angestrebt, um so eine gewisse Intensität der Beschäftigung zu erreichen. Dem Probanden ist dabei freigestellt, ob er mehr Zeit beansprucht oder aber schon früher die Durchführung abbricht.

7.4 Auswertung

Die Auswertung erfolgt ausschließlich durch den Vergleich der mit Hilfe beider Fehlereingrenzungsstrategien benötigten Durchführungszeiten. Im Ergebnis dieser Untersuchung gilt es nachzuweisen, ob es empirisch belegbare Hinweise für eine Effizienzsteigerung der beschriebenen Verfahrensoptimierung gibt.

Bei allen Untersuchungen wurden mit Blick auf die Verbesserung der Fehlereingrenzungsstrategien folgende Werte erfasst:

- Anzahl der untersuchten Fehler (F1 und F2),
- benötigte Zeit zur Behebung der Fehler,
- Auskunft der Probanden über die Anzahl ihrer Arbeitsjahre in der Entwicklung,
- Art der Vorgehensweise (sequentiell/bottom-up) bei der Fehlereingrenzung.

An der Untersuchung haben 11 Probanden teilgenommen. Drei Probanden haben beide Fehler gefunden, fünf haben jeweils erst im zweiten Versuch den Fehler entdeckt. Zwei Probanden fanden in beiden Versuchen keinen Fehler. Ein Proband ist wegen eines technischen Fehlers ausgeschieden.

[50] Gemeint sind u.a. eventuelle Unterbrechungen durch Fragen von Kollegen persönlich/via Email o.ä..

Im ersten Durchlauf beider Gruppen mit dem Fehler F1 haben nur 30% der Probanden den Fehler gefunden (Abbildung 7.1). Daraus kann die Schlussfolgerung gezogen werden, dass das Beispiel ausreichend komplex war, jedoch nicht so komplex, um allen Probanden das Finden des Fehlers unmöglich zu machen.

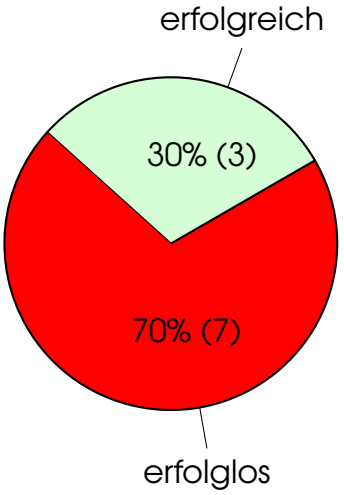


Abbildung 7.1: Verteilung der Erfolge bei der Fehlersuche im ersten Durchlauf beider Gruppen mit dem Fehler F1.

Den Probanden wurde nach einem jeweils erfolglosen Durchgang die Fehlerursache gezeigt. Es ist die Schlussfolgerung zu ziehen, dass fehlende Erfahrungen mit dem Projekt zu den Mißerfolgen im ersten Durchgang geführt haben.

Allerdings lassen die besseren Ergebnisse im zweiten Versuch den Schluss zu, dass die im ersten Durchgang gesammelten Erfahrungen zur Überwindung der hohen Lernkurve beigetragen haben. Hier waren bereits 80% der Probanden bei der Suche von Fehler F2 erfolgreich (Abbildung 7.2, S.71).

In Abbildung 7.3, S.72 werden die Ergebnisse der Untersuchung nach dem zweiten Durchlauf noch einmal in einem Diagramm zusammengefasst.

Aus dem sehr uneinheitlichen Ergebnis der beiden Durchgänge, unabhängig von der Wahl des Verfahrens, kann abgeleitet werden, dass ...

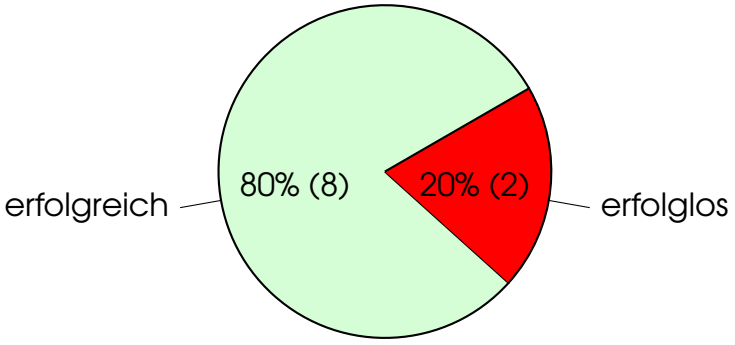


Abbildung 7.2: Verteilung der Erfolge bei der Fehlersuche im ersten Durchlauf beider Gruppen mit dem Fehler F2.

- ... die Eingewöhnung in ein neues Projekt nicht unterschätzt werden darf. Für diese Untersuchung wäre bei einer Wiederholung zu empfehlen, Aufgaben zur Eingewöhnung[51] zu stellen.
- ... eine umfangreichere Testmenge den Einfluß der Komplexität des Projektes zugunsten der Aussagekraft der eingesetzten Verfahren zurückdrängen würde.
- ... Fehler hinsichtlich ihrer Komplexität eingeschätzt und, wenn möglich, gleichverteilt[52] werden sollten.[53]
- ... Tests hinsichtlich ihrer Komplexität eingeschätzt und, wenn möglich, gleichverteilt werden sollten.[54]

[51] Die Aufgaben dienen dazu, die anfänglich sehr hohe Lernkurve während der Untersuchung zu vermeiden. Es wird vorgeschlagen, z.B. Einführungsaufgaben mit dem Projekt umzusetzen oder Aufgaben mit kleineren Fehlern zu lösen.

[52] Es hat sich gezeigt, dass die Art und Schwierigkeit eines Fehlers entscheidend die Dauer der Fehlersuche beeinflusst. Deshalb wird empfohlen, ähnliche Fehler in den Durchgängen zu verursachen. Damit wird die zu messende Dauer nicht an die individuelle Komplexität des Fehlers gekoppelt. Der Verfasser stellt fest, dass zwecks Vergleichbarkeit dieser Fehler weitere Ausführungen notwendig wären, diese aber den Rahmen der vorliegenden Arbeit überschreiten würden. Deshalb wird an dieser Stelle darauf verzichtet.

[53] Einer der beiden Fehler war sehr allgemein und daher wenig aussagekräftig (*NullpointerException*).

[54] Es wurden Tests vorgeschlagen, die aufgrund ihrer indirekten Ausformulierung nur sehr schwer zu verstehen waren: So wurde z.B. bei einem Test ein Fehler erwartet, allerdings trat nicht der erwartete, sondern ein anderer Fehler auf. Daher muss zunächst von den

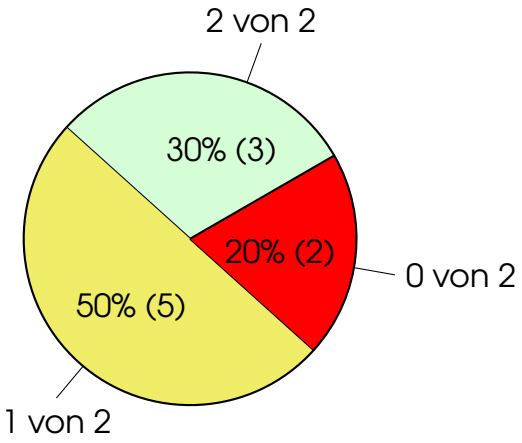


Abbildung 7.3: Verteilung nach beiden Durchläufen.

Die Ergebnisse der Untersuchung zeigen, dass aufgrund der kleinen Stichprobe und dem hohen nicht erfolgreichen Anteil im ersten Durchlauf keine qualitative Aussage über die Wirksamkeit des neuen Verfahrens gemacht werden kann. Somit ist die zu Beginn gestellte Frage (Kapitel 7.4, S.69), ob sich empirisch belegbare Hinweise zur Effizienzsteigerung mit Hilfe einer verbesserten Fehlereingrenzungsstrategie ergeben, abschließend nicht zu beantworten.

Die in dieser Untersuchung gesammelten Erfahrungen lassen jedoch die dargelegten Empfehlungen und Erkenntnisse als realistisch erscheinen. Allerdings müssen diese in einer umfangreichen Feldstudie überprüft werden.

Im Folgenden wird, trotz der eingeschränkten Aussagekraft hinsichtlich der Wirksamkeit der verbesserten Methodik, auf die Ergebnisse im Einzelnen eingegangen.

In Tabelle 7.2, S.73 ist die durchschnittliche Dauer der Fehlersuche dargestellt. Dabei ist zu erkennen, wie sich die Erfahrung mit diesem Projekt im zweiten Durchgang (Suche nach Fehler F2) bereits positiv ausgewirkt (Lerneffekt) und die Dauer der Bearbeitung im Durchschnitt abgenommen hat.

Probanden verstanden werden, warum ein Fehler erwartet wird. Erst dann kann dem „falschen“ Fehler nachgegangen werden.

Fehler	∅ Dauer	Probanden erfolgreich	Probanden erfolglos
F1	14:40 min	3	7
F2	10:18 min	8	2

Tabelle 7.2: Übersicht der Untersuchung beider Gruppen.

Der Lerneffekt läßt sich sowohl bei Gruppe G1 (Tabelle 7.3) als auch bei Gruppe G2 (Tabelle 7.4) beobachten.

Fehler	∅ Dauer	Probanden erfolgreich	Probanden erfolglos
F1	13:30 min	2	4
F2	8:35 min	4	2

Tabelle 7.3: Übersicht der Untersuchung von Gruppe 1 (G1).

Fehler	∅ Dauer	Probanden erfolgreich	Probanden erfolglos[55]
F1	17:00 min	1	3 (1)
F2	6:20 min	4	0 (1)

Tabelle 7.4: Übersicht der Untersuchung von Gruppe 2 (G2).

Beide Gruppen bestanden aus nahezu der gleichen Anzahl von Probanden (G1 sechs Probanden, G2 fünf Probanden).

Es folgt eine zusammenfassende Übersicht über die Ergebnisse der Untersuchung (Tabelle 7.5), welche sehr deutlich den Lerneffekt aus dem ersten Durchlauf (Fehler F1) im zweiten Durchlauf (Fehler F2) zeigt.

Gruppe	Fehler	Strategie	∅ Dauer	Anteil der erfolgreichen Probanden
G1	F1	sequentiell	13:30 min	33%
G1	F2	bottom-up	8:35 min	66%
G2	F1	bottom-up	17:00 min	33%
G2	F2	sequentiell	6:20 min	100%

Tabelle 7.5: Zusammenfassende Übersicht über die Ergebnisse der Untersuchung.

Von den Probanden in vielen Jahren gesammelte Erfahrungen, die in Vorbereitung der Untersuchung erfasst worden sind, wurden nicht gesondert ausgewertet. Grund dafür war die Tatsache, dass diese aufgrund der erreichten Ergebnisse nicht als relevant anzusehen sind.

Alle erhobenen Messdaten sind im Anhang aufgeführt (Anhang D, S.109).

8 Zusammenfassung und Auswertung

Thema der vorliegenden Arbeit war die Analyse und Erarbeitung eines Konzeptes zur Verbesserung der statischen Fehlereingrenzung. Dabei ging es darum, einen *leichtgewichtigen heuristischen* Gegenentwurf zu aufwendigen statistischen Verfahren (z.B. Delta Debugging) zu entwickeln.

Zusammenfassung

Im Rahmen dieser Arbeit wurden der Fehlerbehandlungsprozess im Allgemeinen (Kapitel 2.4, S.27) und die dazu bekannten Vorgehensweisen der Fehlereingrenzung im Speziellen untersucht. Es kristallisierten sich zwei allgemein bekannte Verfahrensweisen heraus, die in dieser Arbeit als *sequentiell* und *erfahrungsbasiert* bezeichnet und beschrieben wurden (Kapitel 3, S.29). Im Zusammenhang mit der Analyse der allgemeinen Strategien hat sich bei beiden Vorgehensweisen die Filterung (die einen Teilschritt des Fehlerbehandlungsprozesses darstellt) als Risiko erwiesen.

Die Abschwächung dieses Risikos war Teil der Evaluationsphase (Kapitel 4, S.37). Die Evaluation wurde maßgeblich durch das Werkzeug *Sonar* ermöglicht (Kapitel 4.2.1, S.39). Zunächst wurde versucht, Tests mit Hilfe von Software-Metriken zu klassifizieren, um dann Vorschläge für die Fehlereingrenzung zu entwickeln. Allerdings sind die bekannten Test-Klassen (Kapitel 2.2, S.24) nur unzureichend mit bekannten Software-Metriken beschreibbar. Deshalb wurde die neue Metrik *Typdistanz* entwickelt (Kapitel 2.1.7, S.19). Die *Typdistanz* ist eine statische objektorientierte Strukturmetrik und bezeichnet die Entfernung zwischen zwei Klassen (Typen). Sie beschreibt mit der von Tests berechneten *Gesamttypdistanz* ein Maß zur Nutzung von

Klassen aus deren Umgebung. Dabei korrelliert die *Typdistanz* mit der Entfernung zwischen dem Test und dem verwendeten Typ.

Das Ziel bestand darin, die Risiken bei der Weiterentwicklung der Methodik zur Fehlereingrenzung zu reduzieren, den Prozess der Fehlerbehandlung und damit den Prozesschritt der *Filterung* aufwandsstabiler zu gestalten. Die Instabilität bei der Filterung ergibt sich aus dem Fehler-Rauschen in Folge der Fehlerfortpflanzung und erhöht so den Aufwand zur Lokalisierung der signifikantesten Tests (Kapitel 5.3, S.53).

Die signifikantesten Tests sind die *kleinsten Tests* (Kapitel 2.1.9, S.22) im Umfeld eines Fehlers. Mit der Sortierung der (fehlgeschlagenen) Tests nach Signifikanz im Schritt der Filterung kann eine Effizienzsteigerung erzielt werden. Es ist anzumerken, dass diese angestrebte und wohl auch zu erreichende Verbesserung mit keinem Risiko einer Verminderung der Effizienz beim Einsatz der weiterentwickelten Methodik verbunden ist (Kapitel 5, Abbildung 5.2, S.55).

Zur Überprüfung bzw. Bestätigung des Konzeptes wurde an einem Fallbeispiel (Kapitel 6, S.63) eine Untersuchung durchgeführt (Kapitel 7, S.67). Dabei haben zehn Probanden bewiesen, dass das entwickelte Konzept eine tragfähige Grundlage bildet und einsetzbar ist. Allerdings konnten aufgrund der geringen Anzahl von Probanden und der Komplexität des Fallbeispiels nur erste Erfahrungen für eine weitergehende Studie gesammelt werden. Ein abschließendes Urteil über die Wirksamkeit der verbesserten Methodik ist somit anhand des Ergebnisses der Untersuchung nicht möglich (Kapitel 7.4, S.69). Sie läßt jedoch zumindest die Richtigkeit des Konzeptes vermuten.

Fazit und Ausblick

Ziel der vorliegenden Arbeit war die Analyse und Erarbeitung eines Konzeptes zur Verbesserung der statischen Fehlereingrenzung. Dieses Ziel ist mit Hilfe des Einsatzes der selbstentwickelten Software-Metrik (Kapitel 2.1.7, Typdistanz auf Seite 19) und einer Weiterentwicklung der Methodik des Fehlerbehandlungsprozesses (Kapitel 3, Analyse fehlgeschlagener Tests auf Seite 29 und Kapitel 5, Weiterentwicklung der Methodik auf Sei-

te 49) erreicht worden. Es können folgende Schlussfolgerungen gezogen werden:

- Der Einsatz von *Softwaremetriken* im Bereich der automatischen Testsortierung hat sich als geeignet erwiesen.
- Die Wirksamkeit dieses Konzeptes, auf dessen Grundlage eine effizientere und damit schnellere Fehleranalyse in der Qualitätssicherung erreicht werden kann, muss allerdings in weiteren Studien umfangreich untersucht werden.
- Der Aufwand der Installation ist überschaubar und die Lernkurve sehr niedrig.
- Die eingesparten Kosten können infolge freiwerdender Ressourcen durch einen effizienteren Fehlerbehandlungsprozess signifikant sein.

Die prototypische Umsetzung der neuen Softwaremetrik *Typdistanz* zur Analyse von Tests und deren Wirksamkeit müssen in weiteren Untersuchungen nachgewiesen werden. Dazu ist es unter Umständen wichtig und notwendig, Einschränkungen der aktuellen Implementierung der *Typdistanz* (Kapitel 5.6, S.61) zu beseitigen.

Des Weiteren sollten zukünftige Untersuchungen mit einem Fallbeispiel berücksichtigen, dass der Lerneffekt, insbesondere bei der Fehlersuche in neuen Projekten, nicht zu unterschätzen ist und berücksichtigt werden muss.

Das Konzept der Testsortierung ist insofern universal, weil es auch auf andere Sprachen übertragen werden kann. Somit können auch auf der Basis der Plattform *Sonar* Adapter für verschiedene Sprachen und Testframeworks umgesetzt werden.

Aus den Ausführungen geht hervor, dass der Einsatz von *Softwaremetriken* im Bereich der automatischen Testkategorisierung eine für die Zukunft durchaus sinnvolle Strategie darstellt. Auf diese Weise ist es möglich, existierende Risiken im Fehlerbehandlungsprozess durch Anwendung unterstützender Werkzeuge zu reduzieren, die Effizienz zu steigern und damit Kosten einzusparen.

Anhang

A Anweisungen für die Untersuchung

A.1 Anweisungen für Gruppe G1 / Test T1

A.1.1 Gegenstand der Untersuchung

Es soll untersucht werden, inwieweit sich Erfahrungen, Vorgehensweise und Werkzeugunterstützung bei der Fehlereingrenzung in einem neuen Projekt auswirken.

Für diesen Zweck ist ein Fehler eingebaut, den Sie bitte korrigieren! Es kann sein, dass der Fehler streut. Sie haben Ihre Arbeit beendet, wenn der Testlauf erfolgreich war. Die Dauer Ihrer Korrektur wird gemessen.

A.1.2 Vorbereitung

1. Bitte laden Sie sich das Projektarchiv von `http://www.lgohlke.de/dl/.dp/DemoProject_F1_s.zip` herunter. Es ist ein Maven-Projekt enthalten.

2. Entpacken Sie dieses in ein Projektverzeichnis, z.B. „Versuch T1".

3. Führen Sie in dem Verzeichnis „Versuch T1" in der Konsole den Befehl **mvn test -DskipTests=true** aus. Das ist wichtig, weil eventuell viele Abhängigkeiten heruntergeladen werden müssen und das Herunterladen unter Umständen etwas länger dauert.

4. Schließen Sie bitte Anwendungen, die Sie während dieser Untersuchung ablenken und dadurch das Untersuchungsergebniss beinflussen könnten. Dazu zählen u.a. Skype/Thunderbird/Facebookchat. Keine Angst, nach spätestens 30 Minuten sind Sie wieder erreichbar.

5. Beschreibung des zu untersuchenden Projektes

   Für eine repräsentative Anwendung nutzte der Verfasser eine ihm bekannte Architektur, die im Weiteren noch näher beschrieben werden wird. Diese wurde zwecks Vereinfachung auf eine begrenzte Zahl von Anwendungsfällen reduziert. Im Folgenden werden zunächst das Anwendungsszenario, danach die Architektur und Funktionsweise der Beispielsoftware beschrieben.

Abbildung A.1: Teil des T-Shirt-*Customizings* im Szenario.

Szenario

Dieses Fallbeispiel beinhaltet einen Webshop, der im Internet T-Shirts anbietet. Dieser Webshop ist mittlerweile eine große Plattform, auf der verschiedene Dienste angeboten werden. Einer dieser Dienste besteht in der Massenbestellung bereits gestalter Kleidung, aber auch im Angebot des sehr beliebten *Customizing* von T-Shirts (siehe Abbildung A.1).

Architektur

Die Architektur wurde zwar auf wesentliche Merkmale reduziert, es kann jedoch von einem hinreichend komplexen Beispiel ausgegangen werden.

Teil der Architektur ist eine private Zugriffsschicht (*API*), die das Produktanlegen und -verändern, aber auch die Authentifizierung und Verwaltung der Nutzer, bereitstellt. Der Dienst *Bus* stellt die Kommunikation für die anderen beiden Dienste `API` und `Datenbank` zur Verfügung. Die *API* wird über den *Bus* mit der *Datenbank* verbunden. Die Abbildung A.2, S.84 beinhaltet die Darstellung der Klassen.

Wichtige Eigenschaften, die in diesem Fallbeispiel enthalten sind und explizit hervorgehoben werden, sind:

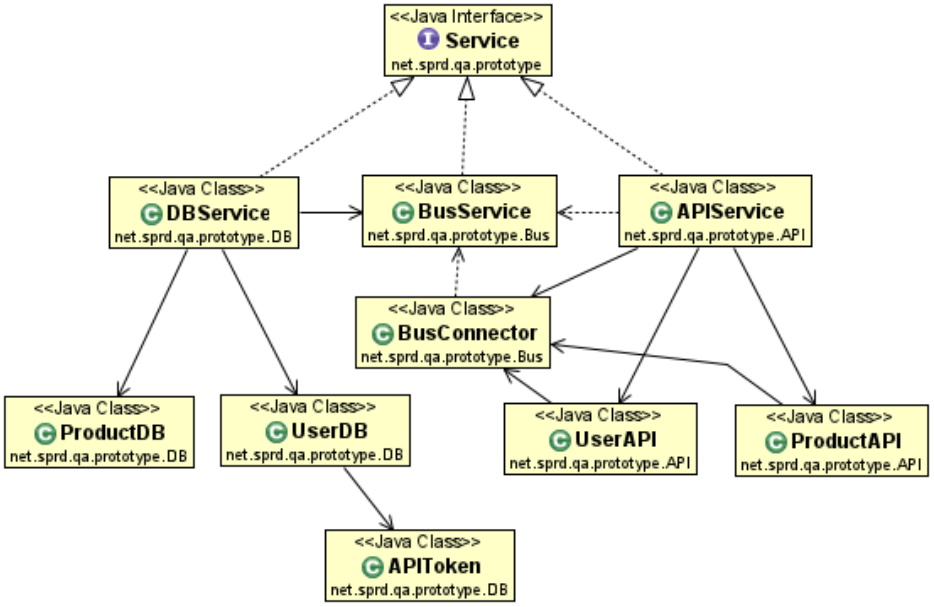


Abbildung A.2: Klassendiagramm des Fallbeispiels.

- **Mehrschichtenarchitektur**

  Es handelt sich um eine klassische 3-Schichtenarchitektur (Zugriffs-, Geschäftslogik- und Persistenzschicht).

- **Serviceorientierte Architektur**

  Serviceorientierte Architektur (SOA) bezeichnet ein Architekturkonzept, das fachliche Dienste anbietet (siehe Abbildung A.3). Folgende Merkmale dieses Architekturkonzeptes sind enthalten: Einsatz eines Enterprise Servicebus und die lose Kopplung der Dienste.

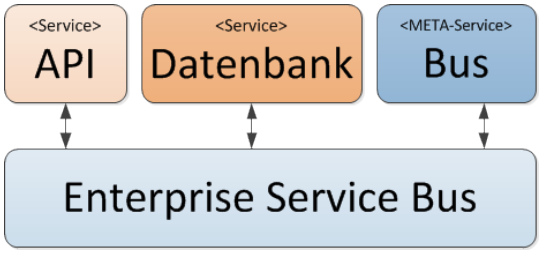


Abbildung A.3: Darstellung als Serviceorientierte Architektur.

6. Richten Sie dieses Projekt in Ihrer Entwicklungsumgebung ein, so dass keine Kompilierungsfehler entstehen.

7. Wenn alle Abhängigkeiten aufgelöst wurden, sind Ihre Vorbereitungen beendet.

HINWEIS

Zur Beobachtung wird die Zeit gemessen. Daher ist es wichtig, dem Versuchsleiter eindeutig zu signalisieren, wann Sie *beginnen* und wann Sie *aufhören*. Ein Abbruch ist erlaubt, wird allerdings frühestens nach 15 Minuten empfohlen.

A.1.3 Durchführung

1. Führen Sie in dem Verzeichnis „Versuch T1" in der Konsole zum Starten der Tests den Befehl **mvn test** aus. Korrigieren Sie alle fehlschlagenen Tests!

2. Teilen Sie dem Versuchsleiter unverzüglich mit, dass Sie Ihre Arbeit beendet haben.

A.1.4 Abschluß

Bitte schicken Sie das Protokollarchiv aus dem Wurzelverzeichnis des Projektverzeichnisses per Skype an **lkwg82** (der Dateiname ist *protokoll.G1.T1.changedCode.tar.bz2*)!

Danke für Ihre Teilnahme an der Untersuchung!

A.2 Anweisungen für Gruppe G1 / Test T2

A.2.1 Gegenstand der Untersuchung

Es soll untersucht werden, inwieweit sich Erfahrungen, Vorgehensweise und Werkzeugunterstützung bei der Fehlereingrenzung in einem neuen Projekt auswirken.

Für diesen Zweck ist ein Fehler eingebaut, den Sie bitte korrigieren! Es kann sein, dass der Fehler streut. Sie haben Ihre Arbeit beendet, wenn der Testlauf erfolgreich war. Die Dauer Ihrer Korrektur wird gemessen.

A.2.2 Vorbereitung

1. Bitte laden Sie sich das Projektarchiv von `http://www.lgohlke.de/dl/.dp/DemoProject_F2_b.zip` herunter. Es ist ein Maven-Projekt enthalten.

2. Entpacken Sie dieses in ein Projektverzeichnis, z.B. „Versuch T2".

3. Führen Sie in dem Verzeichnis „Versuch T2" in der Konsole den Befehl **mvn test -DskipTests=true** aus. Das ist wichtig, weil eventuell viele Abhängigkeiten heruntergeladen werden müssen und das Herunterladen unter Umständen etwas länger dauert.

4. Schließen Sie bitte Anwendungen, die Sie während dieser Untersuchung ablenken und dadurch das Untersuchungsergebniss beinflussen könnten. Dazu zählen u.a. Skype/Thunderbird/Facebookchat. Keine Angst, nach spätestens 30 Minuten sind Sie wieder erreichbar.

5. Beschreibung des zu untersuchenden Projektes

   Für eine repräsentative Anwendung nutzte der Verfasser eine ihm bekannte Architektur, die im Weiteren noch näher beschrieben werden wird. Diese wurde zwecks Vereinfachung auf eine begrenzte Zahl von Anwendungsfällen reduziert. Im Folgenden werden zunächst das Anwendungsszenario, danach die Architektur und Funktionsweise der Beispielsoftware beschrieben.

Abbildung A.4: Teil des T-Shirt-*Customizings* im Szenario.

Szenario

Dieses Fallbeispiel beinhaltet einen Webshop, der im Internet T-Shirts anbietet. Dieser Webshop ist mittlerweile eine große Plattform, auf der verschiedene Dienste angeboten werden. Einer dieser Dienste besteht in der Massenbestellung bereits gestalter Kleidung, aber auch im Angebot des sehr beliebten *Customizing* von T-Shirts (siehe Abbildung A.4).

Architektur

Die Architektur wurde zwar auf wesentliche Merkmale reduziert, es kann jedoch von einem hinreichend komplexen Beispiel ausgegangen werden.

Teil der Architektur ist eine private Zugriffsschicht (*API*), die das Produktanlegen und -verändern, aber auch die Authentifizierung und Verwaltung der Nutzer, bereitstellt. Der Dienst *Bus* stellt die Kommunikation für die anderen beiden Dienste `API` und `Datenbank` zur Verfügung. Die *API* wird über den *Bus* mit der *Datenbank* verbunden. Die Abbildung A.5, S.88 beinhaltet die Darstellung der Klassen.

Wichtige Eigenschaften, die in diesem Fallbeispiel enthalten sind und explizit hervorgehoben werden, sind:

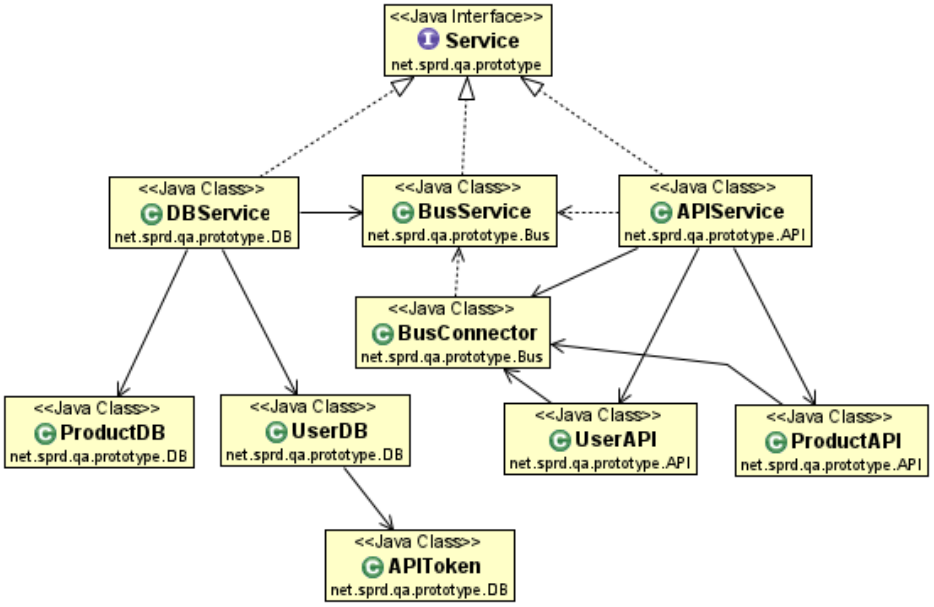


Abbildung A.5: Klassendiagramm des Fallbeispiels.

- **Mehrschichtenarchitektur**

  Es handelt sich um eine klassische 3-Schichtenarchitektur (Zugriffs-, Geschäftslogik- und Persistenzschicht).

- **Serviceorientierte Architektur**

  Serviceorientierte Architektur (SOA) bezeichnet ein Architekturkonzept, das fachliche Dienste anbietet (siehe Abbildung A.6). Folgende Merkmale dieses Architekturkonzeptes sind enthalten: Einsatz eines Enterprise Servicebus und die lose Kopplung der Dienste.

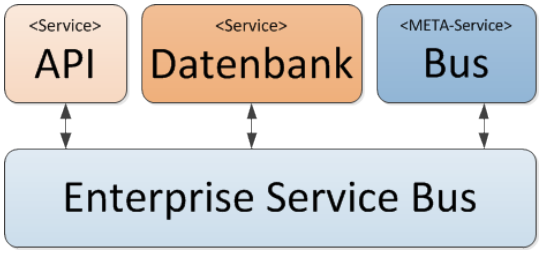


Abbildung A.6: Darstellung als Serviceorientierte Architektur.

6. Richten Sie dieses Projekt in Ihrer Entwicklungsumgebung ein, so dass keine Kompilierungsfehler entstehen.

7. Wenn alle Abhängigkeiten aufgelöst wurden, sind Ihre Vorbereitungen beendet.

HINWEIS

Zur Beobachtung wird die Zeit gemessen. Daher ist es wichtig, dem Versuchsleiter eindeutig zu signalisieren, wann Sie *beginnen* und wann Sie *aufhören*. Ein Abbruch ist erlaubt, wird allerdings frühestens nach 15 Minuten empfohlen.

A.2.3 Durchführung

1. Führen Sie in dem Verzeichnis „Versuch T2" in der Konsole zum Starten der Tests den Befehl **mvn test** aus. Korrigieren Sie alle fehlschlagenen Tests!

2. Bitte untersuchen Sie die fehlgeschlagenen Tests in der Reihenfolge (von oben nach unten), die in der Datei *target/analysis/orderedList.html* vorgegebenen ist.

3. Teilen Sie dem Versuchsleiter unverzüglich mit, dass Sie Ihre Arbeit beendet haben.

A.2.4 Abschluß

Bitte schicken Sie das Protokollarchiv aus dem Wurzelverzeichnis des Projektverzeichnisses per Skype an **lkwg82** (der Dateiname ist *protokoll.G1.T2.changedCode.tar.bz2*)!

Danke für Ihre Teilnahme an der Untersuchung!

A.3 Anweisungen für Gruppe G2 / Test T1

A.3.1 Gegenstand der Untersuchung

Es soll untersucht werden, inwieweit sich Erfahrungen, Vorgehensweise und Werkzeugunterstützung bei der Fehlereingrenzung in einem neuen Projekt auswirken.

Für diesen Zweck ist ein Fehler eingebaut, den Sie bitte korrigieren! Es kann sein, dass der Fehler streut. Sie haben Ihre Arbeit beendet, wenn der Testlauf erfolgreich war. Die Dauer Ihrer Korrektur wird gemessen.

A.3.2 Vorbereitung

1. Bitte laden Sie sich das Projektarchiv von `http://www.lgohlke.de/dl/.dp/DemoProject_F1_b.zip` herunter. Es ist ein Maven-Projekt enthalten.

2. Entpacken Sie dieses in ein Projektverzeichnis, z.B. „Versuch T1".

3. Führen Sie in dem Verzeichnis „Versuch T1" in der Konsole den Befehl **mvn test -DskipTests=true** aus. Das ist wichtig, weil eventuell viele Abhängigkeiten heruntergeladen werden müssen und das Herunterladen unter Umständen etwas länger dauert.

4. Schließen Sie bitte Anwendungen, die Sie während dieser Untersuchung ablenken und dadurch das Untersuchungsergebniss beinflussen könnten. Dazu zählen u.a. Skype/Thunderbird/Facebookchat. Keine Angst, nach spätestens 30 Minuten sind Sie wieder erreichbar.

5. Beschreibung des zu untersuchenden Projektes

   Für eine repräsentative Anwendung nutzte der Verfasser eine ihm bekannte Architektur, die im Weiteren noch näher beschrieben werden wird. Diese wurde zwecks Vereinfachung auf eine begrenzte Zahl von Anwendungsfällen reduziert. Im Folgenden werden zunächst das Anwendungsszenario, danach die Architektur und Funktionsweise der Beispielsoftware beschrieben.

Abbildung A.7: Teil des T-Shirt-*Customizings* im Szenario.

Szenario

Dieses Fallbeispiel beinhaltet einen Webshop, der im Internet T-Shirts anbietet. Dieser Webshop ist mittlerweile eine große Plattform, auf der verschiedene Dienste angeboten werden. Einer dieser Dienste besteht in der Massenbestellung bereits gestalter Kleidung, aber auch im Angebot des sehr beliebten *Customizing* von T-Shirts (siehe Abbildung A.7).

Architektur

Die Architektur wurde zwar auf wesentliche Merkmale reduziert, es kann jedoch von einem hinreichend komplexen Beispiel ausgegangen werden.

Teil der Architektur ist eine private Zugriffsschicht (*API*), die das Produktanlegen und -verändern, aber auch die Authentifizierung und Verwaltung der Nutzer, bereitstellt. Der Dienst *Bus* stellt die Kommunikation für die anderen beiden Dienste `API` und `Datenbank` zur Verfügung. Die *API* wird über den *Bus* mit der *Datenbank* verbunden. Die Abbildung A.8, S.92 beinhaltet die Darstellung der Klassen.

Wichtige Eigenschaften, die in diesem Fallbeispiel enthalten sind und explizit hervorgehoben werden, sind:

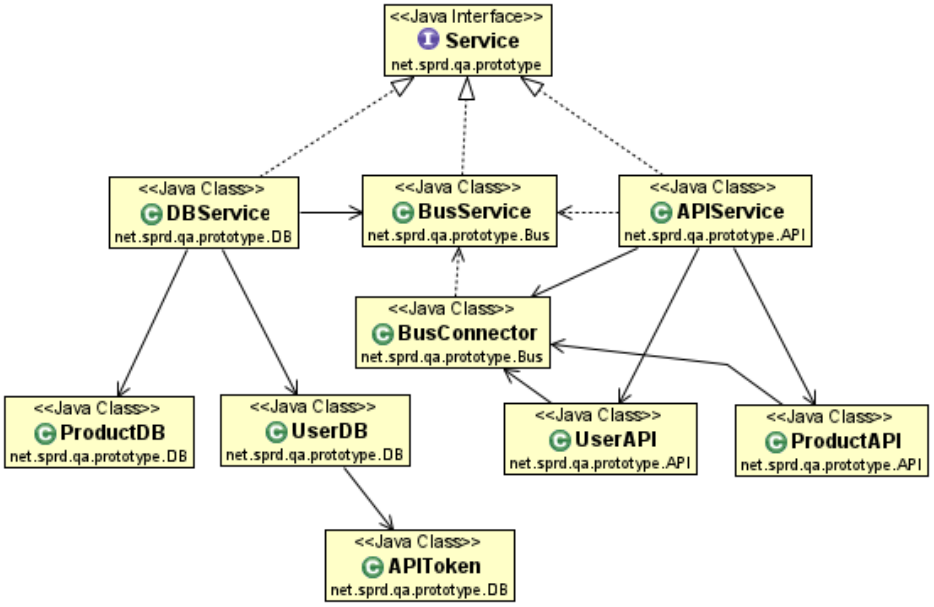


Abbildung A.8: Klassendiagramm des Fallbeispiels.

- **Mehrschichtenarchitektur**

  Es handelt sich um eine klassische 3-Schichtenarchitektur (Zugriffs-, Geschäftslogik- und Persistenzschicht).

- **Serviceorientierte Architektur**

  Serviceorientierte Architektur (SOA) bezeichnet ein Architekturkonzept, das fachliche Dienste anbietet (siehe Abbildung A.9). Folgende Merkmale dieses Architekturkonzeptes sind enthalten: Einsatz eines Enterprise Servicebus und die lose Kopplung der Dienste.

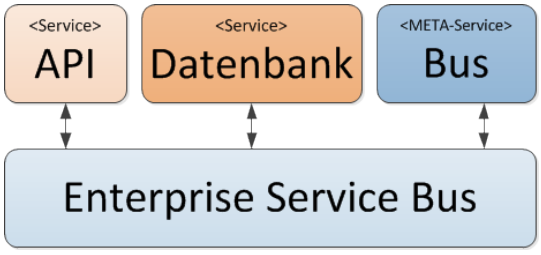


Abbildung A.9: Darstellung als Serviceorientierte Architektur.

6. Richten Sie dieses Projekt in Ihrer Entwicklungsumgebung ein, so dass keine Kompilierungsfehler entstehen.

7. Wenn alle Abhängigkeiten aufgelöst wurden, sind Ihre Vorbereitungen beendet.

HINWEIS

Zur Beobachtung wird die Zeit gemessen. Daher ist es wichtig, dem Versuchsleiter eindeutig zu signalisieren, wann Sie *beginnen* und wann Sie *aufhören*. Ein Abbruch ist erlaubt, wird allerdings frühestens nach 15 Minuten empfohlen.

A.3.3 Durchführung

1. Führen Sie in dem Verzeichnis „Versuch T1" in der Konsole zum Starten der Tests den Befehl **mvn test** aus. Korrigieren Sie alle fehlschlagenen Tests!

2. Bitte untersuchen Sie die fehlgeschlagenen Tests in der Reihenfolge (von oben nach unten), die in der Datei *target/analysis/orderedList.html* vorgegebenen ist.

3. Teilen Sie dem Versuchsleiter unverzüglich mit, dass Sie Ihre Arbeit beendet haben.

A.3.4 Abschluß

Bitte schicken Sie das Protokollarchiv aus dem Wurzelverzeichnis des Projektverzeichnisses per Skype an **lkwg82** (der Dateiname ist *protokoll.G2.T1.changedCode.tar.bz2*)!

Danke für Ihre Teilnahme an der Untersuchung!

A.4 Anweisungen für Gruppe G2 / Test T2

A.4.1 Gegenstand der Untersuchung

Es soll untersucht werden, inwieweit sich Erfahrungen, Vorgehensweise und Werkzeugunterstützung bei der Fehlereingrenzung in einem neuen Projekt auswirken.

Für diesen Zweck ist ein Fehler eingebaut, den Sie bitte korrigieren! Es kann sein, dass der Fehler streut. Sie haben Ihre Arbeit beendet, wenn der Testlauf erfolgreich war. Die Dauer Ihrer Korrektur wird gemessen.

A.4.2 Vorbereitung

1. Bitte laden Sie sich das Projektarchiv von `http://www.lgohlke.de/dl/.dp/DemoProject_F2_s.zip` herunter. Es ist ein Maven-Projekt enthalten.

2. Entpacken Sie dieses in ein Projektverzeichnis, z.B. „Versuch T2".

3. Führen Sie in dem Verzeichnis „Versuch T2" in der Konsole den Befehl **mvn test -DskipTests=true** aus. Das ist wichtig, weil eventuell viele Abhängigkeiten heruntergeladen werden müssen und das Herunterladen unter Umständen etwas länger dauert.

4. Schließen Sie bitte Anwendungen, die Sie während dieser Untersuchung ablenken und dadurch das Untersuchungsergebniss beinflussen könnten. Dazu zählen u.a. Skype/Thunderbird/Facebookchat. Keine Angst, nach spätestens 30 Minuten sind Sie wieder erreichbar.

5. Beschreibung des zu untersuchenden Projektes

   Für eine repräsentative Anwendung nutzte der Verfasser eine ihm bekannte Architektur, die im Weiteren noch näher beschrieben werden wird. Diese wurde zwecks Vereinfachung auf eine begrenzte Zahl von Anwendungsfällen reduziert. Im Folgenden werden zunächst das Anwendungsszenario, danach die Architektur und Funktionsweise der Beispielsoftware beschrieben.

Abbildung A.10: Teil des T-Shirt-*Customizings* im Szenario.

Szenario

Dieses Fallbeispiel beinhaltet einen Webshop, der im Internet T-Shirts anbietet. Dieser Webshop ist mittlerweile eine große Plattform, auf der verschiedene Dienste angeboten werden. Einer dieser Dienste besteht in der Massenbestellung bereits gestalter Kleidung, aber auch im Angebot des sehr beliebten *Customizing* von T-Shirts (siehe Abbildung A.10).

Architektur

Die Architektur wurde zwar auf wesentliche Merkmale reduziert, es kann jedoch von einem hinreichend komplexen Beispiel ausgegangen werden.

Teil der Architektur ist eine private Zugriffsschicht (*API*), die das Produktanlegen und -verändern, aber auch die Authentifizierung und Verwaltung der Nutzer, bereitstellt. Der Dienst *Bus* stellt die Kommunikation für die anderen beiden Dienste `API` und `Datenbank` zur Verfügung. Die *API* wird über den *Bus* mit der *Datenbank* verbunden. Die Abbildung A.11, S.96 beinhaltet die Darstellung der Klassen.

Wichtige Eigenschaften, die in diesem Fallbeispiel enthalten sind und explizit hervorgehoben werden, sind:

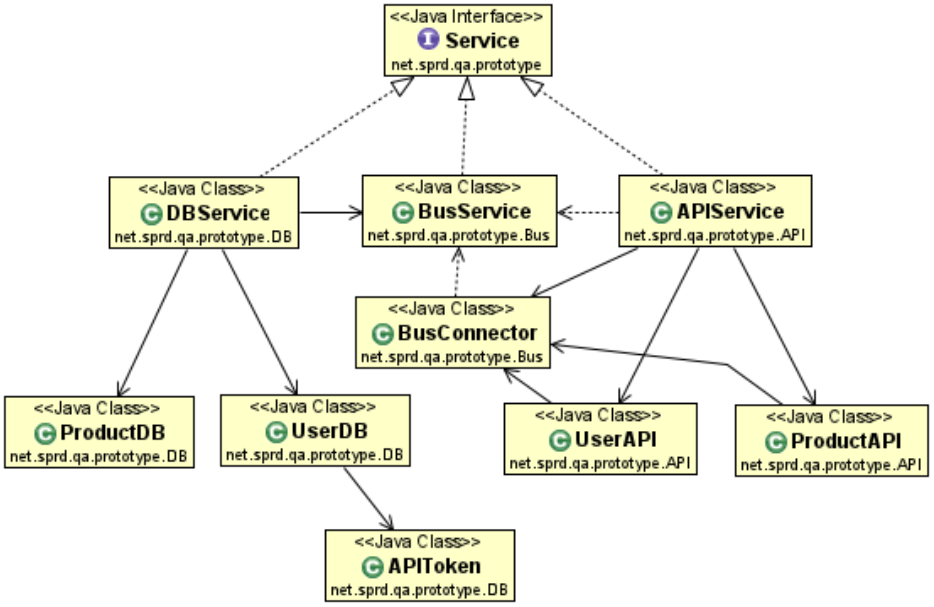


Abbildung A.11: Klassendiagramm des Fallbeispiels.

- **Mehrschichtenarchitektur**

  Es handelt sich um eine klassische 3-Schichtenarchitektur (Zugriffs-, Geschäftslogik- und Persistenzschicht).

- **Serviceorientierte Architektur**

  Serviceorientierte Architektur (SOA) bezeichnet ein Architekturkonzept, das fachliche Dienste anbietet (siehe Abbildung A.12). Folgende Merkmale dieses Architekturkonzeptes sind enthalten: Einsatz eines Enterprise Servicebus und die lose Kopplung der Dienste.

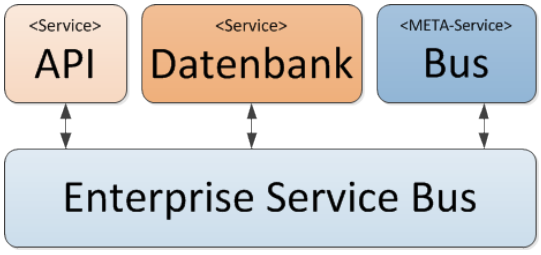


Abbildung A.12: Darstellung als Serviceorientierte Architektur.

6. Richten Sie dieses Projekt in Ihrer Entwicklungsumgebung ein, so dass keine Kompilierungsfehler entstehen.

7. Wenn alle Abhängigkeiten aufgelöst wurden, sind Ihre Vorbereitungen beendet.

HINWEIS

Zur Beobachtung wird die Zeit gemessen. Daher ist es wichtig, dem Versuchsleiter eindeutig zu signalisieren, wann Sie *beginnen* und wann Sie *aufhören*. Ein Abbruch ist erlaubt, wird allerdings frühestens nach 15 Minuten empfohlen.

A.4.3 Durchführung

1. Führen Sie in dem Verzeichnis „Versuch T2" in der Konsole zum Starten der Tests den Befehl **mvn test** aus. Korrigieren Sie alle fehlschlagenen Tests!

2. Teilen Sie dem Versuchsleiter unverzüglich mit, dass Sie Ihre Arbeit beendet haben.

A.4.4 Abschluß

Bitte schicken Sie das Protokollarchiv aus dem Wurzelverzeichnis des Projektverzeichnisses per Skype an **lkwg82** (der Dateiname ist *protokoll.G2.T2.changedCode.tar.bz2*)!

Danke für Ihre Teilnahme an der Untersuchung!

B Von Sonar unterstützte Metriken

Die auf der folgenden Seite beginnende Auflistung ist http://docs.codehaus.org/display/SONAR/Metric+definitions entnommen (Stand: 29.01.2012 22:11).

Metric definitions

Metrics are the heart of Sonar, using Sonar efficiently means perfectly understanding the definition and calculation algorithm of each one.

Size

Name	Key	Qualitative	Description
Physical lines	lines	no	Number of carriage returns

Comment lines	comment_lines	no	Number of javadoc, multi-comment and single-comment lines. Empty comment lines like, header file comments (mainly used to define the license) and commented-out lines of code are not included.
			`/**`
			` * This is a javadoc block`
			` *  <- empty comment line considered as a blank line`
			` */  <- empty comment line considered as a blank line`
			`/*`
			` * This is a multi-comment block`
			` */`
			`// This is a single-comment block`
			`// log("Debug information"); <- commented-out line of code is not a comment line`

Commented-out lines of code	commented_out_code_lines	yes	Number of commented-out lines of code. Javadoc blocks are not scanned.
			`/*`
			`* someoneCommentMeOutOneDay();`
			`* nobodyKnowWhatAmISupposedToDo();`
			`*/`

Lines of code	ncloc	no	Number of physical lines of code - number of blank lines - number of comment lines - number of header file comments - commented-out lines of code
Density of comment lines	comment_lines_density	yes	Number of comment lines / (lines of code + number of comments lines) * 100 With such formula : - 50% means that there is the same number of lines of code and comment lines - 100% means that the file contains only comment lines and no lines of code
Packages	packages	no	Number of packages
Classes	classes	no	Number of classes including nested classes, interfaces, enums and annotations
Files	files	no	Number of analyzed files

Directories	directories	no	Number of analyzed directories

Accessors	accessors	no	Number of getter and setter methods used to get(reading) or set(writing) a class' property .
			
			`// Getters`
			`public String getName(){`
			`  return this.name;`
			`}`
			`public boolean isParent(){`
			`  return this.isParent;`
			`}`
			`// Setters`
			`public void setName(String name){`
			`  this.name = name;`
			`}`
			`public void setIsParent(boolean isParent){`
			`  this.isParent = isParent;`
			`}`

Methods	functions	no	Number of Methods without including accessors. A constructor is considered to be a method.
Public API	public_api	no	Number of public classes, public methods (without accessors) and public properties (without public final static ones)
Public undocumented API	public_undocumented_api	yes	Number of public API without a Javadoc block
Density of public documented API	public_documented_api_density	yes	(Number of public API - Number of undocumented public API) / Number of public API * 100

Statements	statements	no	Number of statements as defined in the Java Language Specification but without block definitions. Statements counter gets incremented by one each time an expression, if, else, while, do, for, switch, break, continue, return, throw, synchronized, catch, finally is encountered :
			
			`//`
			`i = 0;`
			`if (ok)`
			`if (exit) {`
			`if (3 == 4);`
			`if (4 == 4) { ; }`
			`} else {`
			`try{}`
			`while(true){}`
			`for(...){}`
			`...`
			
			Statements counter is not incremented by a class, method, field, annotation definition or by a package and import declaration.

Tests

Name	Key	Qualitative	Description
Unit tests	tests	no	Number of unit tests
Unit tests duration	test_execution_time	no	Time required to execute unit tests
Unit test error	test_errors	yes	Number of unit tests that failed
Unit test failures	test_failures	yes	Number of unit tests that failed with an unexpected exception
Unit test success density	test_success_density	yes	(Unit tests - (errors + failures))/ Unit tests * 100
Skipped unit tests	skipped_tests	yes	Number of skipped unit tests

Line Coverage	line_coverage	yes	On a given line of code, line coverage simply answers the question: "Is this line of code executed during unit test execution?". At project level, this is the density of covered lines:
			`Line coverage = LC / EL`
			
			`where`
			
			`LC - lines covered (lines_to_cover - uncovered_lines)`
			`EL - total number of executable lines (lines_to_cover)`

New Line Coverage	new_line_coverage	yes	identical to line_coverage but restricted to new / update source code

Branch coverage	branch_coverage	yes	On each line of code containing some boolean expressions, the branch coverage simply answers the question: "Has each boolean expression evaluated both to true and false ?". At project level, this is the density of possible branches in flow control structures that have been followed.
			`Branch coverage = (CT + CF) / (2*B)`
			
			`where`
			
			`CT - branches that evaluated to "true" at least once`
			`CF - branches that evaluated to "false" at least once`
			`(CT + CF = conditions_to_cover - uncovered_conditions)`
			
			`B - total number of branches (2*B = conditions_to_cover)`

New Branch Coverage	new_branch_coverage	yes	identical to branch_coverage but restricted to new / update source code

Coverage	coverage	yes	Coverage metric is a mix of the two previous line coverage and branch coverage metrics to get an even more accurate answer to the question "how much of a source-code is being executed by your unit tests?".
			The Coverage is calculated with the following formula :
			`coverage = (CT + CF + LC)/(2*B + EL)`
			`where`
			`CT - branches that evaluated to "true" at least once`
			`CF - branches that evaluated to "false" at least once`
			`LC - lines covered (lines_to_cover - uncovered_lines)`
			`B - total number of branches (2*B = conditions_to_cover)`
			`EL - total number of executable lines (lines_to_cover)`

New Coverage	new_coverage	yes	identical to coverage but restricted to new / update source code
Conditions to Cover	conditions_to_cover	no	Total number of conditions which could be covered by unit tests.
New Conditions to Cover	new_conditions_to_cover	no	identical to conditions_to_cover but restricted to new / update source code
Lines to Cover	lines_to_cover	no	Total number of lines of code which could be covered by unit tests.
New Lines to Cover	new_lines_to_cover	no	identical to lines_to_cover but restricted to new / update source code
Uncovered Conditions	uncovered_conditions	no	Total number of conditions which are not covered by unit tests.
New Uncovered Conditions	new_uncovered_conditions	no	identical to uncovered_conditions but restricted to new / update source code
Uncovered Lines	uncovered_lines	no	Total number of lines of code which are not covered by unit tests.
New Uncovered Lines	new_uncovered_lines	no	identical to uncovered_lines but restricted to new / update source code

Duplication

Name	Key	Qualitative	Description
Duplicated lines	duplicated_lines	yes	Number of physical lines touched by a duplication

Duplicated blocks	duplicated_blocks	yes	Number of duplicated blocks of lines
Duplicated files	duplicated_files	yes	Number of files involved in a duplication of lines
Density of duplicated lines	duplicated_lines_density	yes	Duplicated lines / Physical lines * 100

Design

Name	Key	Qualitative	Description
Depth of inheritance tree	dit	no	The depth of inheritance tree (DIT) metric provides for each class a measure of the inheritance levels from the object hierarchy top. In Java where all classes inherit Object the minimum value of DIT is 1.
Number of children	noc	no	A class's number of children (NOC) metric simply measures the number of direct and indirect descendants of the class.
Response for class	rfc	no	The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. RFC is simply the number of methods in the set.
Afferent couplings	ca	no	A class's afferent couplings is a measure of how many other classes use the specific class.
Efferent couplings	ce	no	A class's efferent couplings is a measure of how many different classes are used by the specific class.
Lack of cohesion of methods	lcom4	yes	LCOM4 measures the number of "connected components" in a class. A connected component is a set of related methods and fields. There should be only one such component in each class. If there are 2 or more components, the class should be split into so many smaller classes.
Package cycles	package_cycles	yes	Minimal number of package cycles detected to be able to identify all undesired dependencies.
Package dependencies to cut	package_feedback_edges	no	Number of package dependencies to cut in order to remove all cycles between packages.
File dependencies to cut	package_tangles	no	Number of file dependencies to cut in order to remove all cycles between packages.
Package edges weight	package_edges_weight	no	Total number of file dependencies between packages.
Package tangle index	package_tangle_index	yes	Gives the level of tangle of the packages, best value 0% meaning that there is no cycles and worst value 100% meaning that packages are really tangled. The index is calculated using : 2 * (package_tangles / package_edges_weight) * 100.
File cycles	file_cycles	yes	Minimal number of file cycles detected inside a package to be able to identify all undesired dependencies.
Suspect file dependencies	file_feedback_edges	no	File dependencies to cut in order to remove cycles between files inside a package. Warning : cycles are not always bad between files inside a package.

File tangle	file_tangles	no	file_tangles = file_feedback_edges.
File edges weight	file_edges_weight	no	Total number of file dependencies inside a package.
File tangle index	file_tangle_index	yes	2 * (file_tangles / file_edges_weight) * 100.

Complexity

Name	Key	Qualitative	Description

Complexity	complexity	no	The Cyclomatic Complexity Number is also known as McCabe Metric. It all comes down to simply counting 'if', 'for', 'while' statements etc. in a method. Whenever the control flow of a method splits, the Cyclomatic counter gets incremented by one.
			Each method has a minimum value of 1 per default except accessors which are not considered as method and so don't increment complexity. For each of the following Java keywords/statements this value gets incremented by one:
			`if`
			`for`
			`while`
			`case`
			`catch`
			`throw`
			`return (that isn't the last statement of a method)`
			`&&`
			`||`
			`?`
			Note that else, default, and finally don't increment the CCN value any further. On the other hand, a simple method with a switch statement and a huge block of case statements can have a surprisingly high CCN value (still it has the same value when converting a switch block to an equivalent sequence of if statements).
			For instance the following method has a complexity of 5
			`public void process(Car myCar){ <- +1`
			`if(myCar.isNotMine()){ <- +1`
			`return; <- +1`
			`}`
			`car.paint("red");`
			`car.changeWheel();`
			`while(car.hasGazol() && car.getDriver().isNotStressed()){ <- +2`
			`car.drive();`
			`}`
			`return;`
			`}`

Average complexity by method	function_complexity	yes	Average cyclomatic complexity number by method

Complexity distribution by method	function_complexity_distribution	yes	Number of methods for given complexities
Average complexity by class	class_complexity	yes	Average cyclomatic complexity by class
Complexity distribution by class	class_complexity_distribution	yes	Number of classes for given complexities
Average complexity by file	file_complexity	yes	Average cyclomatic complexity by file

Rules

Name	Key	Qualitative	Description
Violations	violations	yes	Total number of rule violations
New Violations	new_violations	yes	Total number of new violations
xxxxx violations	xxxxx_violations	yes	Number of violations with severity xxxxx, xxxxx being blocker, critical, major, minor or info
New xxxxx violations	new_xxxxx_violations	yes	Number of new violations with severity xxxxx, xxxxx being blocker, critical, major, minor or info
Weighted violations	weighted_violations	yes	Sum of the violations weighted by the coefficient associated at each priority (Sum(xxxxx_violations * xxxxx_weight))
Rules compliance index	violations_density	yes	100 - weighted_violations / Lines of code * 100

SCM

Name	Key	Qualitative	Description
Commits	commits	no	The number of commits.
Last commit date	last_commit_date	no	The latest commit date on a resource.
Revision	revision	no	The latest revision of a resource.
Authors by line	authors_by_line	no	The last committer on each line of code.
Revisions by line	revisions_by_line	no	The revision number on each line of code.

C Ausgabe für die Entwicklung

Auf der folgenden Seite ist eine Gegenüberstellung von Metriken und Quelltexten für die Implementierung dargestellt.

de.lgohlke.syntaxhighlighter.CodeDocumentHelperTest#testGetFormattedCode()

```
@org.junit.Test
public void testGetFormattedCode(){
    String sourceCode = "public HtmlDivTable(final int columns, final int rows) \n" + //
            "{\n" + //
            "  this.columns = columns;\n" + //
            "  this.rows = rows;\n" + //
            "  cells = new ArrayList<String>(columns * rows);\n" + //
            "}\n" + //
            "",

    JavaMethod method = new JavaMethod();
    method.setParentClass(new JavaClass("dad.asdsad.asdas"));

    method.setSourceCode(sourceCode);
    TestMethod test = new TestMethod(method);

    {
        int startLineCount = 1;
        int showLine = 5;
        int showLinesBeforeAndAfter = 1;
        String formattedCode = new CodeDocumentHelper().getFormattedCode(test, showLine, showLinesBeforeAndAfter

        // System.out.println(formattedCode);
        Assert.assertEquals(1 + 2 + (showLinesBeforeAndAfter * 2), formattedCode.split("\\r?\\n").length);
    }
}
```

test	AST_NUMBER_OF_VARIABLE_DEFINITION	AGGREGATE_SUM_VARIABLE_DEFINITION	AST_VARIABLE_DEFINITION_TYPE_DISTANCE	AST_CLASS_VARIABLE_DEFINITION_TYPE_DISTANCE	AGGREGATE_SUM_VARIABLE_DEFINITION_TYPE_DISTANCE	AST_NUMBER_OF_VARIABLE_DEFINITION_NON_ZERO_TYPE_DISTANCE	AST_CLASS_NUMBER_OF_VARIABLE_DEFINITION_NON_ZERO_TYPE_DISTANCE	AGGREGATE_NUMBER_OF_VARIABLE_DEFINITION_NON_ZERO_TYPE_DISTANCE	AST_RELATED_METHODS	COMPLEXITY	LINES_OF_CODE	LINES	STATEMENTS	AST_NUMBER_OF_ASSERT_STATEMENTS	AGGREGATE_VARIABLE_DEFINITION_TYPE_DISTANCE_PER_LOC	AGGREGATE_MEDIAN_DEFINITION_TYPE_DISTANCE	AGGREGATE_MAX_DEFINITION_TYPE_DISTANCE	testOrderScore
RelatedTests.TestTestClass#test	8	13	15	14	29	4	3	7	5	6	38	40	14	4	0,67	4,00	6,00	6040014
de.lgohlke.test.QDOXBug#test	0	0	0	0	0	0	0	0	0	1	5	5	1	3	0,00	0,00	0,00	1
de.lgohlke.test.DummyClass#test	0	0	0	0	0	0	0	0	0	1	4	5	0	3	{ AST_NUMBER_OF_ASSERT_STATEMENTS			
de.lgohlke.syntaxhighlighter.CodeDocumentHelperTest#testGetFormattedCode()	7	7	12	0	12	2	0	2	0	1	22	30	1	0,59	0,00	8,00	8000010	
de.lgohlke.runner.TestNGAnalysisRunnerTest#testNGTest2	0	2	0	6	6	0	1	0	1	1	5	5	1	3	0,86	3,00	6,00	6003001
de.lgohlke.runner.TestNGAnalysisRunnerTest#testNGTest1	0	2	0	6	6	0	1	1	0	1	6	6	2	3	0,75	3,00	6,00	6003002
de.lgohlke.runner.TestNGAnalysisRunnerTest#test	2	4	6	6	12	1	1	2	2	3	24	25	12	3	0,52	3,00	6,00	6003012
de.lgohlke.runner.CLIRunnerTest#nullArguments	0	1	0	0	0	0	0	0	1	4	24	26	6	3	0,00	0,00	0,00	6
de.lgohlke.runner.CLIRunnerTest#mavenEnvironmentNonExistingDirectory	1	2	0	0	0	0	0	0	1	4	25	27	7	3	0,00	0,00	0,00	7
de.lgohlke.runner.CLIRunnerTest#mavenEnvironment	1	2	2	0	2	1	0	1	1	4	26	29	8	2	0,08	1,00	2,00	2201008

D Messdaten der Untersuchung

Person	Erfahrung mit Java[56]	Gruppe	Dauer $F1$	Dauer $F2$
P1	15 Jahre	G1	16 min	1 min
P2	13 Jahre	G1	11 min	10 min
P3	13 Jahre	G2	17 min	31 min
P4	11 Jahre	G1	x	9 min
P5	10 Jahre	G2	x	4 min
P6	8 Jahre	G1	x	9 min
P7	7 Jahre	G2	x	4 min
P8	6 Jahre	G1	x	x
P9	4 Jahre	G2	x	4 min
P10	3 Jahre	G1	x	x
P11	7 Jahre	G2	ungültig[57]	ungültig

Tabelle D.1: Übersicht der Rohdaten zur Untersuchung.

[56] Die Anzahl der Jahre bezeichnet den Zeitraum zwischen dem ersten *„Hello World"* und dem Zeitpunkt der Untersuchung. Nicht berücksichtigt wurden dabei Unterbrechungen in der Entwicklung mit Java zugunsten anderer Sprachen.

[57] Wegen eines technischen Fehlers sind beide Messungen ungültig.

E Veröffentlichung des Codes

Die Quelltexte für das Programm sind als Google-Code-Projekt öffentlich zugänglich unter http://code.google.com/p/metricanalyzer/.

Quellen- und Literaturverzeichnis

(Bal08) H. Balzert. *Lehrbuch Der Softwaretechnik: Softwaremanagement*. Lehrbuch der Software-Technik. Spektrum Akademischer Verlag, 2008.

(Bal99) H. Balzert. *Lehrbuch Grundlagen der Informatik: Konzepte und Notationen in UML, Java und C++, Algorithmik und Software-Technik, Anwendungen*. Spektrum Lehrbücher der Informatik. Spektrum Akad. Verl., 1999.

(Duv07) Paul Duvall. *Automation for the people: Continuous Integration anti-patterns*. englisch. (Online; Stand 7. Oktober 2011, 12:26). 2007. URL: http://www.ibm.com/developerworks/java/library/j-ap11297/.

(Fel05) Prof. Stefan Felsner. *GRAPHENTHEORIE*. (Online; Stand 12. November 2011, 17:49). 2005. URL: http://page.math.tu-berlin.de/~felsner/Lehre/GrTh05/Graphentheorie.pdf.

(Fow) Martin Fowler. *Refactoring*. (Online; Stand 2. Oktober 2011, 00:40). URL: http://refactoring.com/.

(Fow06) Martin Fowler. *Continuous Integration*. englisch. (Online; Stand 7. Oktober 2011, 12:40). 2006. URL: http://martinfowler.com/articles/continuousIntegration.html.

(Goh08) Lars Gohlke. "SOA für Multi-Messaging - Integrationskonzept am Beispiel Skype". (Online; Stand 22. November 2011, 18:33). Magisterarb. FH Brandenburg, 2008. URL: http://lgohlke.de/_/arbeiten/study-thesis-master.

(Goh10) Lars Gohlke. "Qualitätsmanagement mit Continuous Integration - Untersuchung anhand einer Machbarkeitsstudie in der Praxis". (Online; Stand 14. Oktober 2011, 18:53). Magisterarb. Fachhochschule Brandenburg, 2010. URL: http://lgohlke.de/_/arbeiten/diplomarbeit.

(Gos+05) James Gosling u. a. *The Java Language Specification, Third Edition*. 3. Aufl. Amsterdam: Addison-Wesley Longman, 2005, S. 688.

(Hof08) D.W. Hoffmann. *Software-Qualität*. Springer, 2008.

(Iee83) Ieee. *Standard for software test documentation*. Ieee, 1983.

(Krö02) Fred Kröger. *Interpreter - Gliederung*. (Online; Stand 20. März 2012). 2002. URL: http://www.pst.informatik.uni-muenchen.de/lehre/WS0203/psem/doku/georgieva_hand.pdf.

(Lib+05) Ben Liblit u. a. "Scalable statistical bug isolation". In: *PLDI 05* 40 (2005), S. 15–26.

(Lib04) B.R. Liblit. "Cooperative bug isolation". Diss. Citeseer, 2004.

(Lig09) P. Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2009.

(NL10) P.A. Nainar und B. Liblit. "Adaptive bug isolation". In: *Proc. of the 32nd Int. Conf. on Software Engineering*. 2010, S. 255–264.

(NNQ03) Statistik und Zertifizierungsgrundlagen Deutsches Institut für Normung. Normenausschuß Qualitätsmanagement. *DIN ISO 9126 „Software-Engineering - Qualität von Software-Produkten"*. DIN EN ISO. 2003.

(NNQ05) Statistik und Zertifizierungsgrundlagen Deutsches Institut für Normung. Normenausschuß Qualitätsmanagement. *Qualitätsmanagementsysteme: Grundlagen und Begriffe; (ISO 9000:2005)*. DIN EN ISO. Beuth, 2005.

(OW02) T. Ottmann und P. Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Lehrbuch. Spektrum, Akad. Verl., 2002.

(Rei11) Victoria Reitano. "Software development: manufacturing process or creative idea?" In: *SDTimes* (2011). (Online; Stand 2. Februar 2012, 12:49). URL: http://www.sdtimes.com/blog/post/2011/08/09/Software-Development-Manufacturing-Process-or-Creative-Idea.aspx.

(Rob09) Dave Robertson. "Is creative software development dead?" In: *ComputerWorlUK* (2009). (Online; Stand 2. Februar 2012, 12:50). URL: http://www.computerworlduk.com/in-depth/applications/2216/is-creative-software-development-dead/.

(Saf06) David Saff. *Continuous Testing*. (Online; Stand 2. Oktober 2011, 01:49). 2006. URL: http://groups.csail.mit.edu/pag/continuoustesting/.

(SJ05) K.S. Stephens und J.M. Juran. *Juran, quality, and a century of improvement*. Book series of International Academy for Quality. ASQ Quality Press, 2005.

(SO05) C.D. Sterling und R.A. Olsson. "Automated bug isolation via program chipping". In: *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM. 2005, S. 23–32.

(Son12) Sonarsource. *Put your technical debt under control*. (Online; Stand 29. Januar 2012, 15:31). 2012. URL: http://www.sonarsource.org.

(Tec09) Agitar Technologies. *Why Unit Testing?* (Online; Stand 16. Oktober 2011, 19:37). 2009. URL: http://www.agitar.com/solutions/why_unit_testing.html.

(Tho06) BT ThoughtWorks. *Agile Cookbook - A wiki guide to Agile Delivery in the 90 Day Cycle*. (Online; Stand 2. Oktober 2011, 12:53). Feb. 2006. URL: http://agilecookbook.com/.

(Wel09) Don Wells. *Refactoring*. (Online; Stand 2. Oktober 2011, 00:49). 2009. URL: http://www.extremeprogramming.org/.

(Wes90) M. Wessells. *Kognitive Psychologie*. E. Reinhardt, 1990, S. 356.

(Wik11a) Wikipedia. *Fehlerfortpflanzung — Wikipedia, Die freie Enzyklopädie*. (Online; Stand 13. Oktober 2011, 18:19). 2011. URL: http://de.wikipedia.org/w/index.php?title=Fehlerfortpflanzung&oldid=89052465.

(Wik11b) Wikipedia. *Fehlerrechnung — Wikipedia, Die freie Enzyklopädie*. (Online; Stand 13. Oktober 2011, 18:18). 2011. URL: http://de.wikipedia.org/w/index.php?title=Fehlerrechnung&oldid=90146294.

(Zel03) Andreas Zeller. *Software-Test*. (Online; Stand 13. September 2011, 13:35). 2003. URL: http://www.st.cs.uni-saarland.de/edu/einst/12-testen.pdf.

(Zel99) Andreas Zeller. "Yesterday, my program worked. Today, it does not. Why?" In: *Proc. ESEC/FSE 99* Vol. 1687 of LNCS (1999). (Online; Stand 03. Oktober 2011, 12:55), S. 253–267. URL: http://www.infosun.fim.uni-passau.de/st/papers/tr-99-01/esec99.pdf.

(Zol10) H.D. Zollondz. *Grundlagen Qualitätsmanagement: Einfuehrung in Geschichte, Begriffe, Systeme und Konzepte*. Edition Management. Oldenbourg, 2010.

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Erklärung zur Masterarbeit

Hiermit erkläre ich, die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst zu haben. Auf die verwendeten Quellen habe ich in entsprechender Weise hingewiesen und diese im Literaturverzeichnis aufgeführt.

Brandenburg a. d. H., April 2012

Lars Gohlke