Electronic version of an article published as **Liggesmeyer P, Engels G, Münch J, Dörr J, Riegel N (Hrsg): Software Engineering 2009, 02-06. März 2009 in Kaiserslautern, LNI P-143, pp. 139-150**

Testfokusauswahl im Integrationstestprozess

Lars Borner, Barbara Paech

Lehrstuhl für Software Systeme
Universität Heidelberg, Institut für Informatik
Im Neuenheimer Feld 326
69120 Heidelberg
borner@informatik.uni-heidelberg.de
paech@informatik.uni-heidelberg.de

Abstract: Im Integrationstestprozess werden Bausteine eines Softwaresystems schrittweise zusammengesetzt und deren Abhängigkeiten untereinander getestet. Die große Anzahl Abhängigkeiten heutiger Systeme macht es unmöglich, alle Abhängigkeiten zu testen. Tester müssen sich daher auf die fehlerhaften Abhängigkeiten fokussieren. In diesem Beitrag stellen wir einen Ansatz vor, der Informationen über Abhängigkeitseigenschaften nutzt, um den Testfokus für den Integrationstest festzulegen. Der Ansatz umfasst eine Liste von Eigenschaften, um Abhängigkeiten zu klassifizieren. Darüber hinaus gibt er eine Methode vor, um mit Hilfe von Fehlerdaten früherer Versionen eines Softwaresystems sowie deren Eigenschaften fehlerhafte Abhängigkeiten zu identifizieren. Evaluiert wurde der Ansatz mit Hilfe der Entwicklungsumgebung Eclipse.

1 Einleitung

Softwaresysteme bestehen heutzutage aus einer großen Anzahl von (Software-) Bausteinen[1], die unzählige Anforderungen realisieren. Um diese Anforderungen zu erfüllen, müssen die realisierenden Bausteine miteinander interagieren oder sind voneinander abhängig. Das Überprüfen der Abhängigkeiten zwischen Bausteinen erfolgt innerhalb des Integrationstestprozesses. Ziel ist es, „… Fehler in Schnittstellen und im Zusammenspiel zwischen integrierten Komponenten[2] zu finden“ ([SL06], Seite 202). Nachdem die einzelnen Bausteine im Komponententest getestet wurden, werden sie im Integrationstest schrittweise zu einem Gesamtsystem zusammengesetzt und die Abhängigkeit zwischen ihnen getestet. Der Fokus liegt dabei auf typischen Fehlern im Zusammenspiel der Bausteine. Anschließend kann der Systemtestprozess durchgeführt werden, um die korrekte Umsetzung der Anforderungen zu überprüfen.

Innerhalb des Integrationstestprozesses werden verschiedene Entscheidungen getroffen (vgl. [BIP07]). Eine der wichtigsten Entscheidungen ist die Auswahl des Testfokus, d.h. die Auswahl der zu testenden Abhängigkeiten. Die Festlegung des Testfokus ist

[1] Ein Baustein repräsentiert einen Teil eines Softwaresystems (z.B. eine Klasse, eine Modul, eine Komponente, ein Teilsystem, eine Datei, …), der von anderen Teilen abhängig ist bzw. von dem andere Teile abhängig sind.
[2] In diesem Fall synonym mit dem Begriff Baustein

notwendig, da vollständiges Testen eines Softwaresystems nicht möglich ist (vgl. [Me79]). Diese Entscheidung schränkt die Anzahl der zu testenden Abhängigkeiten auf die vermutlich fehlerhaften ein. In unserer Arbeit wird eine Abhängigkeit als *fehlerhaft bezeichnet*, wenn mindestens einer der beteiligten Bausteine eine hohe Fehleranzahl aufweist, da das Fehlschlagen einer Abhängigkeit immer durch Fehler in den beteiligten Bausteinen verursacht wird. In der Literatur über den Integrationstest sind nur wenige Informationen über die Testfokusauswahl für den Integrationstest zu finden. Sneed und Winter verweisen in [SW02] auf eine Strategie zur Ermittlung der Integrationsreihenfolge auf Basis der Kritikalität der Bausteine. Ein Vorgehen, wie diese Kritikalität ermittelt werden kann, wird jedoch nicht beschrieben. Ansätze zur Fehlervorhersage können diese Lücke schließen. Die Ansätze verwenden statistische Verfahren, um Zusammenhänge zwischen der Fehleranzahl von Bausteinen und ihren Bausteineigenschaften (z.B. in [BBM96], [OW07]) in früheren Versionen eines Softwaresystems aufzudecken. Gefunden Zusammenhänge werden in der aktuellen Version ausgenutzt, um vorherzusagen, welche Bausteine fehleranfällig sein werden (z.B. [BBM96]) oder um die voraussichtliche Fehleranzahl pro Baustein zur ermitteln (z.B. [OW07]). Alle Ansätze konzentrieren sich nur auf einzelne Bausteine und nicht auf die Abhängigkeiten und sind für den Integrationstest nur bedingt einsetzbar. Unser Ansatz legt den Fokus auf die Abhängigkeiten und ihre Eigenschaften und sagt fehleranfällige Abhängigkeiten voraus, die als Testfokus für den Integrationstest ausgewählt werden. Wir stellen eine Liste von möglichen Abhängigkeitseigenschaften zur Verfügung, um die Abhängigkeiten eines Softwaresystems zu klassifizieren. Wir verwenden ebenfalls statistische Verfahren, um Zusammenhänge zwischen der Fehleranzahl in den Bausteinen und den Abhängigkeitseigenschaften in früheren Versionen eines Softwaresystems aufzudecken. Diese nutzen wir aus, um den Testfokus für die aktuelle Version festzulegen. Der Ansatz wurde für die Entwicklungsumgebung Eclipse in zwei verschiedenen Versionen angewendet. In diesen Versionen deckten wir Zusammenhänge auf und legten basierend darauf den Testfokus für eine dritte Version fest. Nachfolgend werden die identifizierten Abhängigkeitseigenschaften vorgestellt (Kapitel 2). Im Kapitel 3 beschreiben wir den Ansatz zur Testfokusauswahl. Die Ergebnisse der Anwendung des Ansatzes für Eclipse stellen wir in Kapitel 4 vor. Abschließend gehen wir auf zukünftige Arbeiten ein und geben eine kurze Zusammenfassung.

2 Abhängigkeitseigenschaften

Eine Abhängigkeit ist eine unidirektionale Beziehung zwischen zwei Bausteinen. Ein Baustein übernimmt die Rolle des abhängigen und einer des unabhängigen Bausteins (vgl. Abbildung 1a). Es kann zwischen zwei Bausteinen maximal zwei Abhängigkeiten geben. In Abbildung 1b) existieren die Abhängigkeiten „ab“ und „ba“. In „ab“ ist der Baustein A der abhängige und Baustein B der unabhängige Baustein; in „ba“ entsprechend umgekehrt. Eine Abhängigkeit wird durch Abhängigkeitseigenschaften näher definiert (siehe unten), wobei eine Abhängigkeit mehrere Eigenschaften in sich vereinen kann. Baustein A kann beispielsweise verschiedene Dienste an Baustein B aufrufen und auf mehrere Attribute bei B zugreifen.

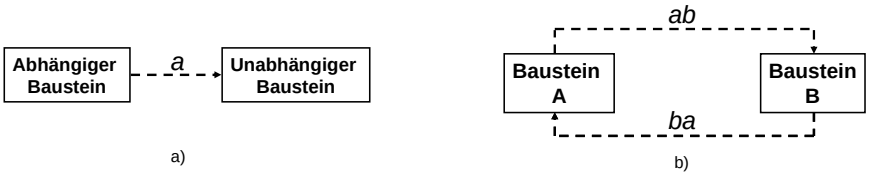

Abbildung 1: Abhängigkeit zwischen Bausteinen

Das Wissen über Abhängigkeitseigenschaften stellt einen Schlüsselfaktor für die erfolgreiche Durchführung eines Integrationstests dar. Eine detaillierte Beschreibung einer Abhängigkeit und ihrer Eigenschaften ermöglicht zum einen, die Abhängigkeit in eine fehlerhafte oder weniger fehlerhafte Abhängigkeit zu klassifizieren. Zum anderen geben die Eigenschaften Auskunft über mögliche Fehler, die innerhalb der Abhängigkeit auftreten können (z.B. wenn bei einem Dienstaufruf zwei Parameter gleichen Typs übergeben werden, kann das zu dem Fehler führen, dass die Parameterwerte versehentlich vertauscht werden, ohne dass es eine Fehlermeldung vom Compiler gibt [SPB+06]). Die Liste der Abhängigkeitseigenschaften wurde mit Hilfe einer Literaturrecherche und Gesprächen mit Entwicklern und Testern erarbeitet. Die identifizierten Eigenschaften lassen sich in drei Kategorien unterteilen: **Laufzeiteigenschaften**, **Entwicklungseigenschaften** und **Deploymenteigenschaften**. Nachfolgend werden die Eigenschaften dieser drei Kategorien kurz beschrieben.

2.1 Laufzeiteigenschaften

Auf oberster Ebene können die Laufzeiteigenschaften in *Client/Server-*, *Vererbungs-* und *indirekte Eigenschaften* eingeteilt werden. Bei einer Client/Server-Eigenschaft ruft ein Baustein bei einem zweiten Dienste auf und/oder greift auf dessen Attribute zu. Die Eigenschaft des *Dienstaufrufs* kann durch weitere Eigenschaften näher spezifiziert werden: übergebene oder zurück gegebene *Parameter*, einzuhaltende *Kommunikationsverträge*, die *Kommunikationsart* (asynchron, synchron) oder *„übergreifende" Zustandsänderungen*. Dienstaufrufe mit Eingabeparameter können eine mögliche Fehlerquelle sein. Die Parameter können von den beteiligten Bausteinen falsch interpretiert und verarbeitet werden [SPB+06] (z.B. ist ein Zahlenwert als Euro übergeben worden, wird aber als Dollar interpretiert). Die Komplexität der Parameter (Eingabe oder Ausgabe) kann ebenfalls ein Indiz auf die Fehleranfälligkeit sein, da sie komplexe Zustände in sich bergen können. Ein Kommunikationsvertrag (vgl. [VR02]) schränkt die Interaktion zwischen zwei Bausteinen ein. Eine vordefinierte Reihenfolge muss eingehalten werden, um einen bestimmten Dienst zu erbringen. Zeitliche Beschränkungen liegen vor, wenn der Server seine Antwort in einer bestimmten Zeit liefern muss. Darüber hinaus können weitere nichtfunktionale Anforderungen (NFA) an die Interaktion gestellt werden, z.B. die Genauigkeit der Ergebnisse. Für den Integrationstester ist es von Interesse, ob die Dienstaufrufe synchron oder asynchron verlaufen und die beteiligten Bausteine in einem oder in zwei parallelen Kontrollprozessen ausgeführt werden. Beispielsweise kann im Falle eines Kontrollprozesses eine wechselseitige Kommunikation zu einem Deadlock führen, wenn die Kommunikation synchron verläuft. Die übergreifende Zustandsänderung beschreibt die Folgen einer Kommunikation. Diese Folgen können das Erzeugen von Instanzen,

Modifikation an existierenden Instanzen und Bausteinen sowie das Löschen von Instanzen sein. Zur Laufzeit können die Änderungen an falschen Instanzen oder an falschen Bausteintypen durchgeführt werden. Attributzugriffe beschreiben das Lesen und/oder Modifizieren von Attributen (oder Variablen) am Server. Der Client hat dabei Zugriff auf Attribute des Servers und kann sie beliebig modifizieren. Ein Zugriff auf ein nicht initialisiertes Attribut beispielsweise (vgl. [Bi96]) kann eine mögliche Fehlerursache sein.

Unter einer Vererbungseigenschaft *„versteht man die Möglichkeit, ein neues Objekt von einem vorhandenen Objekt abzuleiten, wobei das neue Objekt alle Merkmale und Fähigkeiten des alten besitzt.*“ (in [MSH03], Seite 62). Der neue Baustein (Unterklasse) erbt alle Eigenschaften, d.h. alle Zustände, Attribute, Dienste und Abhängigkeiten des vorhandenen Bausteins (Oberklasse). Diese Art der Eigenschaft ermöglicht es, eine Instanz einer Oberklasse durch eine Instanz der Unterklasse zu ersetzen. Wichtige Eigenschaften in einer Vererbung sind die *Modifikation* und die *Vererbungstiefe*. Die Modifikation beschreibt das Ausmaß der Veränderung der Eigenschaften der Oberklasse durch die Unterklasse. Die Modifikation kann durch das Hinzufügen von neuen Diensten und Attributen sowie durch das Überschreiben von existierenden Diensten geschehen. Beim Überschreiben eines Dienstes implementiert die Unterklasse das Verhalten des Dienstes der Oberklasse neu und ändert damit das erwartete Verhalten. Dies bedeutet, dass das Verhalten der Unterklasse nicht zwangsläufig konsistent zum Verhalten der Oberklasse ist. Da jedoch jede Instanz einer Oberklasse durch eine Instanz einer Unterklasseklasse substituiert [Or98] werden kann, wird das erwartete Verhalten (das der Oberklasse) nicht garantiert. Die Vererbungstiefe beschreibt den „Abstand“ zwischen der betrachteten Unterklasse und der Oberklasse. Je größer der Abstand ist, desto mehr Klassen sind zwischen den beiden Klassen der Vererbung zu finden. Alle Klassen in der Klassenhierarchie müssen als potentielle Modifikatoren[3] angesehen werden. Je größer die Vererbungstiefe ist, desto stärker wurde die Oberklasse modifiziert.

Indirekte Abhängigkeiten entstehen, wenn zwei Bausteine voneinander abhängig sind, obwohl sie nicht direkt miteinander kommunizieren oder Vererbungseigenschaften besitzen. Die Abhängigkeit entsteht durch die gemeinsame Verwendung interner und externer Ressourcen, wodurch sich die Bausteine gegenseitig beeinflussen. Die betrachteten Ressourcen sind in dieser Abhängigkeit in keiner der beiden Bausteine zu finden. Verschiedene Arten von Ressourcen sind denkbar: *globale Variablen*, *externe Datenquellen*, gemeinsam genutzte *Hardware* oder *externe Dienste*.

2.2 Entwicklungs- & Deploymenteigenschaften

Die Klasse der Entwicklungs- und Deploymenteigenschaften ist im Vergleich zu den Laufzeiteigenschaften sehr klein. Aber sie liefern ergänzende Informationen über eine bereits vorliegende Abhängigkeit mit Laufzeiteigenschaften. Entwicklungseigenschaften

[3] Ein Modifikator ist eine Klasse, die die Eigenschaften der Oberklasse modifiziert (durch neue Attribute, Dienste oder das Überschreiben)

beschreiben näher, unter welchen Umständen die Abhängigkeit und die daran beteiligten Bausteine entstanden sind. Durch Offshoring [CR05] oder durch große Softwareentwicklungsprojekte entstehen *inhomogene Entwicklungsteams*, die *geographisch verteilt* sind und unter Umständen auch mit *unterschiedlichen Technologien* (z.B. Programmiersprachen) arbeiten. Die Realisierung von abhängigen Bausteinen an unterschiedlichen Standorten kann zu Missverständnissen und somit zu Fehlern führen. Beispielsweise kann die Verwendung von unterschiedlichen Maßzahlen, insofern sie nicht eindeutig spezifiziert wurden, zu unerwartetem Verhalten in der Kommunikation führen. Die Abhängigkeiten zwischen Bausteinen, die in unterschiedlichen Programmiersprachen realisiert werden, können aufgrund unterschiedlich realisiertet Programmiersprachenkonzepte fehlschlagen. Die Deploymenteigenschaften beziehen die Hardwareumgebung mit ein, in der die betrachteten Bausteine installiert sind und ausgeführt werden. Hierbei wird unterschieden, ob die Bausteine *verteilt installiert* sind und ausgeführt werden oder sich auf dem gleichen Hardwareknoten befinden. „*Two components that unconsciously interact via the infrastructure are potential source of failures*“ ([Ma03] Seite 7).

3 Testfokusauswahl

„There is no universal metric or prediction model hat applies to all projects“ (vgl. [ZN08] Seite 1). Daher stützt sich unser Ansatz zur Testfokusauswahl auf Erkenntnisse, die aus früheren Versionen eines Softwaresystems gewonnen werden. Wir empfehlen, die Daten, die über frühere Versionen bereits vorliegen, zu nutzen, um Erkenntnisse für die aktuelle Version des Softwaresystems zu gewinnen. Existierende Ansätze legen den Schwerpunkt auf die Identifikation von fehlerhaften Bausteinen. Unser Ansatz identifiziert fehleranfällige Abhängigkeiten. Hierfür setzen wir für frühere Versionen Abhängigkeitseigenschaften mit der Fehleranfälligkeit der beteiligten Bausteine in Beziehung. Die Fehleranfälligkeit definiert sich als Anzahl der Fehler innerhalb eines Bausteins für eine bestimmte Version. Mit Hilfe statistischer Verfahren decken wir Zusammenhänge zwischen der Fehleranzahl in den Bausteinen und den Abhängigkeitseigenschaften auf. Die statistischen Untersuchungen werden für mehrere frühere Versionen durchgeführt, um versionsübergreifende Korrelationen aufzudecken. Nur Korrelationen, die über mehrere Versionen Bestand haben, sollten zur Auswahl des Testfokus verwendet werden.

Die Testfokusauswahl mit Hilfe unserer Methode besteht aus drei Hauptschritten: *Abhängigkeitseigenschaften festlegen* (1), *frühere Versionen analysieren* (2) und *Testfokusauswahl für die aktuelle Version* (3) (vgl. Abbildung 2). Die Tatsache, dass Informationen über Abhängigkeitseigenschaften in den Artefakten verschiedener Entwicklungsphasen eines Projektes verstreut sind, erschwert die Identifikation der Abhängigkeiten und ihrer Eigenschaften. Aus diesem Grund müssen im ersten Schritt unseres Ansatzes die Abhängigkeitseigenschaften identifiziert werden, die für statistische Untersuchungen zur Verfügung stehen und zu denen schneller Zugang möglich ist. Der Quelltext der Bausteine liefert viele Informationen über die Laufzeiteigenschaften (z.B. Dienstaufrufe, Parameter, Vererbung und Modifikation). Der Projektplan informiert über Entwicklungseigenschaften. Architektur- und

Entwurfsmodelle geben einen Überblick über die Verteilung, die Art der Kommunikation (synchron, asynchron), Kommunikationsverträge usw. In unseren Untersuchungen haben wir uns nur auf den Quelltext als Informationsquelle konzentriert. Wir analysierten Abhängigkeiten (und ihre Eigenschaften) zwischen Quelltextdateien. (siehe Kapitel 4). Nachdem festgelegt wurde, welche Eigenschaften erhoben werden, müssen die Wertebereiche der Eigenschaften definiert werden. In den meisten Fällen handelt es sich bei dem Wertebereich einer Eigenschaft um eine bestimmte Anzahl (z.B. Anzahl aufgerufener Dienste vom abhängigen zum unabhängigen Baustein).

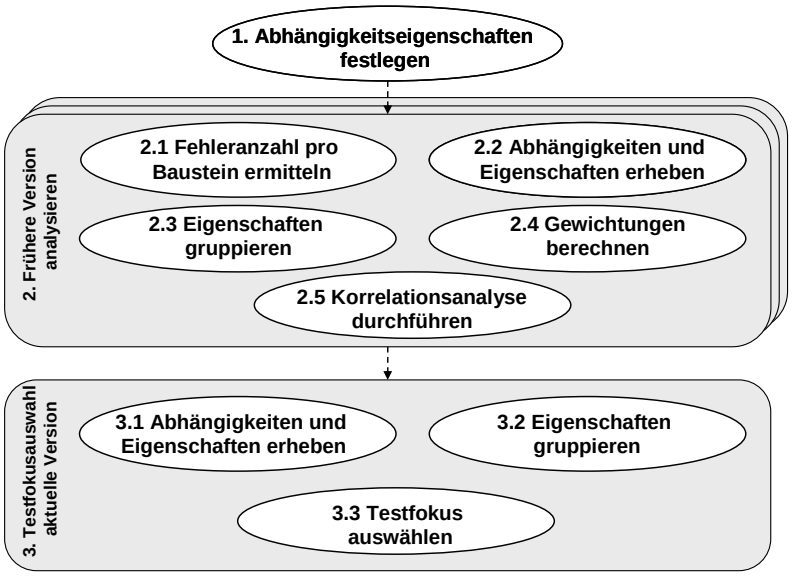

Abbildung 2: Einzelschritte der Testfokusauswahl

Im zweiten Schritt müssen frühere Versionen eines Softwaresystems analysiert werden, um Zusammenhänge zwischen Abhängigkeitseigenschaften und Fehleranzahl der beteiligten Bausteine aufzudecken (2). Für möglichst viele frühere Versionen müssen folgende Teilschritten ausgeführt werden (vgl. Abbildung 2): *Fehleranzahl pro Baustein ermitteln (2.1)*, *Abhängigkeiten und Eigenschaften erheben (2.2)*, *Eigenschaften gruppieren (2.3)*, *Gewichtungen berechnen (2.4)*, *Korrelationsanalyse durchführen (2.5)*. In unserer Methode verwenden wir das Verfahren aus [ZPZ07] um die Fehleranzahl pro Baustein zu berechnen (2.1). Es werden Informationen aus Bugtrackingsystemen, die zur Verwaltung von Fehlermeldungen dienen und Informationen aus Versionsveraltungssystemen (z.B. CVS, SVN) verwendet, um die Fehleranzahl pro Baustein und Version zu ermitteln. Parallel dazu können die Abhängigkeiten und ihre Eigenschaften erhoben werden (2.2). Hierbei werden nur die Eigenschaften betrachtet, die im ersten Schritt (*Abhängigkeitseigenschaften festlegen*) ausgewählt worden sind. Eine übersichtliche Darstellung in Tabellenform als Ergebnis bietet sich hierbei an (eine Tabelle pro Version), wobei jede Zeile eine Abhängigkeit und die Spalten die Eigenschaften darstellen. Diese Tabelle wird um die erhobene

Fehleranzahl der beteiligten Bausteine ergänzt. Die Fehleranzahl des abhängigen und des unabhängigen Bausteins wird für jede Abhängigkeit in die letzte Spalte der Tabelle eingetragen. Abbildung 3 stellt ein Beispiel für eine Abhängigkeitstabelle (Ausschnitt) dar. Die ersten zwei Spalten enthalten die Bezeichnung des abhängigen und unabhängigen Bausteins. Spalten zwei bis fünf listen die Ausprägungen der Eigenschaften für die einzelnen Abhängigkeiten auf. Es ist in Zeile zwei zu erkennen, dass in der Abhängigkeit zwischen dem Baustein `GC.java` und `OS.java` der abhängige Baustein 75 unterschiedliche Dienste am unabhängigen Baustein aufruft, auf 37 unterschiedliche Attribute zugreift und es zwischen beiden Bausteinen keine Vererbung gibt. Die letzten zwei Spalten geben die Fehleranzahl der beteiligten Bausteine wieder (2 für `GC.java` und 8 für `OS.java`). Die entstandene Tabelle kann anschließend von einem Statistik-Werkzeug (z.B. SPSS [SP08]) eingelesen und weiter verarbeitet werden. Bevor die statistischen Analysen durchgeführt werden können, müssen Eigenschaften jeweils gruppiert werden (2.3), da wir als statistisches Verfahren den Mittelwert verwenden. In unseren Untersuchungen hat sich gezeigt, dass eine 10er Einteilung gute Ergebnisse liefert. Dabei wird jeder Wert einer Eigenschaft in eine von 10 Gruppen umkodiert, wobei darauf geachtet wird, dass die entstehenden Gruppen annähernd gleich groß sind. Dezile bieten sich hierbei an. Bevor die statistischen Auswertungen durchgeführt werden können, müssen die einzelnen Abhängigkeiten gewichtet werden (2.4). Dies ist notwendig, da ein Baustein an mehreren Abhängigkeiten als abhängiger und/oder unabhängiger Baustein beteiligt sein kann. Die Gewichtung ergibt sich aus dem Kehrwert der Häufigkeit, mit der ein Baustein als abhängiger bzw. unabhängiger Baustein an einer Abhängigkeit beteiligt ist. Abschließend können wir die Korrelationsanalyse durchführen (2.5). Da an einer Abhängigkeit immer genau zwei Bausteine beteiligt sind, können die Korrelationsanalysen mit der Fehleranzahl des abhängigen, des unabhängigen oder der Summe der Fehleranzahl beider Bausteine durchgeführt werden. In unserem Ansatz verwenden wir die Fehleranzahl beider einzelnen Bausteine, d.h. wir prüfen, ob es einen Zusammenhang zwischen der Eigenschaft und der Fehleranzahl im abhängigen Baustein und zusätzlich ob es einen Zusammenhang zwischen Eigenschaft und Fehleranzahl im unabhängigen Baustein gibt.

Abhängige Datei	Unabhängige Datei	# Attributzugriffe	# Serviceaufrufe	Vererbung	...	# Fehler im abhängiger Baustein	# Fehler im unabhängiger Baustein
CodeAttribute.java	IBytecodeVisitor.java	0	205	0	...	12	2
GC.java	OS.java	37	75	0	...	2	8
Decorations.java	OS.java	79	72	0	...	5	8
Control.java	OS.java	164	71	0	...	8	8
BinaryExpression.java	CodeStream.java	1	64	0	...	12	2
ASTConverter.java	AST.java	0	63	0	...	3	24
AstMatchingNodeFinder.java	ASTVisitor.java	0	62	1	...	5	1
Display.java	OS.java	80	52	0	...	45	8
CodeStream.java	ConstantPool.java	4	51	0	...	2	0
Shell.java	OS.java	49	47	0	...	3	8
⋮	⋮	⋮	⋮	⋮	...	⋮	⋮

Abbildung 3: Ausschnitt einer Abhängigkeitstabelle eines Softwaresystems

Für das Aufdecken von Zusammenhängen zwischen Fehleranzahl und Eigenschaften verwenden wir den Mittelwertvergleich [JL02]. Aufgrund der großen Streuung in den Ergebnissen wurden Zusammenhänge, die durch den Mittelwertvergleich aufgedeckt

wurden, nicht von anderen Verfahren (z.B. Rangkorrelation nach Bravais-Pearson [JL02] oder Korrelationskoeffizient nach Spearman [JL02]) aufgedeckt. Beim Mittelwertvergleich wird die durchschnittliche Fehleranzahl in den gruppierten Eigenschaften (vgl. Schritt 2.3) miteinander verglichen. Gesucht wird nach positiven Zusammenhängen zwischen der durchschnittlichen Fehleranzahl und der untersuchten Eigenschaft, d.h. je größer der Wert (z.B. Anzahl der aufgerufenen Dienste), einer Eigenschaft ist, desto größer ist die durchschnittliche Fehleranzahl. Diese Korrelationsanalyse wird für alle Eigenschaften, die in Schritt 1 festgelegt wurden, durchgeführt. Nachdem der zweite Schritt (und alle seine Teilschritte 2.1-2.5) für mehrere frühere Versionen durchgeführt wurden, können Korrelationen ausgewählt werden, die über mehrere Versionen Bestand haben.

Im dritten Schritt wird die Testfokusauswahl für die aktuelle Version durchgeführt. Hierzu werden die Abhängigkeiten und Eigenschaften für die aktuelle Version identifiziert (3.1) und nach dem gleichen Schema wie in Schritt 2.3 gruppiert (3.2). Anschließend wird der Testfokus ausgewählt. (3.3) Hierzu wird jede Abhängigkeit markiert, die mindestens eine Eigenschaft besitzt, die 1.) eine Korrelation mit der Fehleranfälligkeit aufweist und 2.) deren Wert in den letzten (höchsten) Gruppen eingeordnet werden kann. Hierbei kann unterschieden werden, ob die Eigenschaft eine Korrelation mit der Fehleranzahl des abhängigen Bausteins, mit der Fehleranzahl des unabhängigen Bausteins oder mit der Fehleranzahl beider Bausteine aufweist. Anschließend werden Markierungen verwendet, um den Testfokus auszuwählen. Hat eine Abhängigkeit nur Eigenschaften, die keine Korrelation zur Fehleranzahl aufweisen oder deren Werte der Eigenschaften nicht in den höchsten Gruppen zu finden sind, so wird sie nicht zum Testfokus gezählt. Gibt es mindestens eine Eigenschaft in der Abhängigkeit, die sich in den höchsten Gruppen befindet und eine Korrelation zur Fehleranzahl mit genau einem der beteiligten Bausteine aufzeigt, so wird der Testfokus dieser Abhängigkeit mit einer „1“ kodiert. Bei einer Abhängigkeit, die mindestens einen Eigenschaftswert aufweist, der in den höchsten Gruppen ist, und diese Eigenschaft eine Korrelation mit der Fehleranzahl beider Bausteine aufweist, wird der Testfokus mit „2“ kodiert. Die Abhängigkeiten mit der Kodierung „2“ bekommen die höchste Testpriorität. Abhängigkeiten mit der Kodierung 1 erhalten eine geringere Testpriorität. Entsprechend der vergebenen Testprioritäten werden die zu testenden Abhängigkeiten ausgewählt.

4 Fallstudie Eclipse

Im Rahmen einer Fallstudie haben wir unsere Methode angewendet. Wir wollten zeigen, dass mit Hilfe der Methode, die Abhängigkeiten mit Eigenschaften, die auf fehlerhafte Bausteine hinweisen, ausgewählt werden. Verwendet wurde die Entwicklungsumgebung Eclipse [Ec08] in den Versionen 2.0, 2.1 und 3.0. Es standen die Quelltexte und die Fehleranzahl pro Quelltextdatei zur Verfügung [ZPZ07]. In einem ersten Schritt wurden die Abhängigkeiten und ihre Eigenschaften identifiziert (einfache statistische Daten über die drei Versionen sind in Abbildung 4 zusammengefasst).

	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0
Anzahl Quelltextdateien	6747	7908	10631
Anzahl Abhängigkeiten	56765	71182	96652
Durchschnittliche Fehleranzahl pro Datei	1,38	0,78	0,95

Abbildung 4: Statistische Daten der drei Eclipse-Versionen

12 Abhängigkeitseigenschaften wurden mit Hilfe eines Werkzeugs automatisch aus dem Quelltext extrahiert, um die Abhängigkeiten näher zu charakterisieren. Die untersuchten Eigenschaften waren:

- Vererbungseigenschaften: *Anzahl überschriebener Dienste*, *Anzahl neuer Dienste*, *Anzahl neuer Attribute*
- Client/Server-Eigenschaften: *Anzahl Attributzugriffe*, *Anzahl aufgerufener Dienste*, *Anzahl Eingabeparameter*, *Anzahl komplexer Eingabeparameter*, *Anzahl Dienste mit komplexen Ausgabeparametern*, *Anzahl Dienste mit mindestens zwei Eingabeparametern gleichen Typs*, *durchschnittliche Anzahl Eingabeparameter pro Dienstaufruf*, *durchschnittliche Anzahl komplexer Eingabeparameter pro Dienstaufruf*, *relative Häufigkeit von Dienstaufrufen mit komplexen Ausgabeparametern*

Im nächsten Schritt wurden für die Versionen 2.0 und 2.1 Mittelwertvergleiche durchgeführt. Dabei konnte nur für vier Eigenschaften (*Anzahl überschriebener Dienste* vgl. Abbildung 5, *Anzahl neuer Dienste*, *Anzahl neuer Attribute* und *Anzahl Dienstaufrufe mit komplexem Ausgabeparameter*) ein signifikanter[4] Zusammenhang hinsichtlich der Fehleranzahl im **abhängigen Baustein** aufgedeckt werden, der in beiden Versionen Bestand hatte.

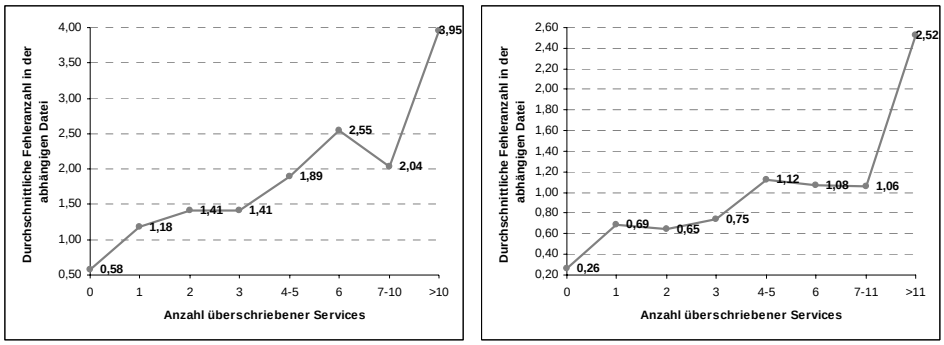

Abbildung 5: Zusammenhang zwischen Anzahl überschriebener Dienste und Fehleranzahl des abhängigen Bausteins (links Version 2.0, rechts: Version 2.1)

Für die **unabhängigen Bausteine** wurden ebenfalls vier Eigenschaften identifiziert, die auf eine erhöhte Fehleranzahl hindeuten (*Anzahl Dienstaufrufe*, *Anzahl Dienstaufrufe mit komplexem Ausgabeparameter*, *Anzahl Eingabeparameter*, *Anzahl komplexer Eingabeparameter*). Die gefundenen Zusammenhänge wurden abschließend verwendet,

[4] Im Rahmen der statistischen Untersuchungen wurde ein Signifikanzniveau von 0,01 verwendet.

um die zu testenden Abhängigkeiten für die Eclipse-Version 3.0 festzulegen. Nach der oben vorgestellten Methode wurden die Abhängigkeiten kodiert (0 = *nicht testen*, 1 = *testen mit geringer Priorität*, 2 = *testen mit hoher Priorität*). Aus den 96652 Abhängigkeiten der Version 3.0 wurden 5318 Abhängigkeiten mit hoher Testpriorität, 12980 Abhängigkeiten mit mittlerer Testpriorität und die verbleibenden 78354 Abhängigkeiten als "nicht testen" eingestuft. Abschließend wurde ein erneuter Mittelwertvergleich für die Version 3.0 durchgeführt, um die Unterschiede der Fehleranzahl in den drei Gruppen (0, 1, 2) miteinander zu vergleichen. Das Ergebnis ist in Abbildung 6 zu sehen. Es zeigt sich, dass die Bausteine, die an Abhängigkeiten beteiligt sind, die nicht als Testfokus ausgewählt wurden, die geringste durchschnittliche Fehleranzahl aufweisen. Für die zweite Gruppe ist die durchschnittliche Fehleranzahl höher als für die erste, aber kleiner als die dritte Gruppe. Die Gruppe mit der höchsten Priorität weist auch die höchste durchschnittliche Fehlerrate auf, sowohl für den abhängigen als auch für den unabhängigen Baustein.

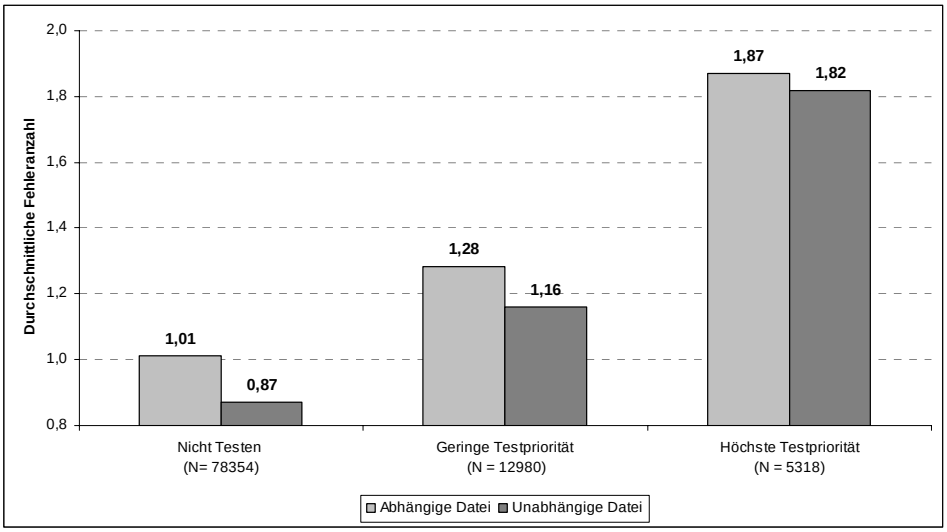

Abbildung 6: Durchschnittliche Fehleranzahl für Dateien nach Zuordnung des Testfokus

Die Untersuchung am Beispielsystem Eclipse hat die Anwendbarkeit unseres Ansatzes an einem großem System gezeigt. Es war möglich, mit Hilfe statischer Analysewerkzeuge die große Anzahl von Abhängigkeiten und ihrer Eigenschaften aufzudecken. Es zeigt sich auch, dass die Eigenschaften, die für Eclipse 2.0 und 2.1 auf fehlerhafte Bausteine hinweisen, auch in Eclipse 3.0 verwendet werden können, um den Testfokus sinnvoll festzulegen. Abbildung 6 zeigt, dass die Abhängigkeiten so ausgewählt worden sind, dass die beteiligten Bausteine eine höhere Fehleranzahl aufweisen. Zusätzlich ermöglicht die Einteilung in „nicht testen", „Testen mit geringer Priorität" und „Testen mit hoher Priorität" eine bessere Planung des Testprozesses. Abhängigkeiten mit hoher Priorität müssen auf jeden Fall getestet werden und wenn anschließend noch Ressourcen zur Verfügung stehen, kann die zweite Gruppe getestet werden.

5 Zusammenfassung

Das vorgestellte Verfahren zur Testfokusauswahl für den Integrationstest bietet eine Möglichkeit, aus den Erfahrungen früherer Versionen eines Softwaresystems zu lernen und dieses Wissen in der aktuellen Version des Systems einzusetzen. Inwieweit sich das gewonnen Wissen auf andere Softwaresysteme übertragen lässt, muss in nachfolgenden Studien untersucht werden.

In zukünftigen Arbeiten wollen wir das Verfahren für die Testfokusauswahl verfeinern, um eine feingranularere Einteilung der Abhängigkeiten nach Testnotwendigkeit durchzuführen. Hierzu werden nicht nur die gefundenen Korrelationen verwendet, sondern auch untersucht, inwieweit der Fehlerdurchschnitt in den einzelnen Gruppen einer Eigenschaft vom Fehlerdurchschnitt aller Dateien abweicht. Je größer die Abweichung ist, desto größer ist der Einfluss der Eigenschaft auf die Fehleranfälligkeit der beteiligten Bausteine. In Abbildung 5 beispielsweise ist zu erkennen, dass für die Version 2.0 die durchschnittliche Fehleranzahl der abhängigen Datei in der letzten Gruppe 3,95 beträgt. Die durchschnittliche Fehleranzahl aller Dateien dieser Version beträgt jedoch „nur“ 1,38 Fehler pro Datei. Somit liegt die Fehleranzahl der abhängigen Dateien in dieser Gruppe ca. 2,8mal höher. Diese Faktoren wollen wir uns zu Nutze machen, um die einzelnen Eigenschaften zu gewichten. Darüber hinaus wird derzeit erarbeitet, inwieweit nicht nur fehlerhafte Bausteine sondern direkt fehlerhafte Abhängigkeiten identifiziert werden können.

In weiteren Arbeiten werden die Ergebnisse der Testfokusauswahl dazu verwendet, geeignete Testentwurfstechniken auszuwählen, um die ausgewählten Abhängigkeiten gezielt testen zu können. Die Eigenschaften geben Auskunft darüber, welche Fehler in dieser Abhängigkeit auftreten können. So können diese mit systematischen Verfahren aufgedeckt werden. Darüber hinaus wird ein Verfahren zur Ermittlung einer optimalen Integrationsreihenfolge erarbeitet, welches den ausgewählten Testfokus berücksichtigt. Es ist das Ziel, Abhängigkeiten, die auf fehlerhafte Bausteine hinweisen, möglichst früh zu integrieren, um viele Fehler früh im Integrationstestprozess aufzudecken. Des Weiteren hat eine frühe Integration von fehlerhaften Bausteinen und Abhängigkeiten den Vorteil, dass sie im Rahmen weiterer Integrationstests indirekt mit getestet werden (vgl. [Bi00]).

Während der Anwendung des Ansatzes im Rahmen der Fallstudie hat sich gezeigt, dass sich der Zeitaufwand in Rahmen hält. Den größten Teil nahm dabei die Quelltextanalyse ein, um die Abhängigkeiten und ihre Eigenschaften zu extrahieren. Das von uns eingesetzte Werkzeug benötigte dafür acht bis zehn Stunden pro Version. Die Korrelationsanalyse wurde mit Hilfe des Werkzeugs SPSS durchgeführt. Darin wurden alle Analysen per Hand angestoßen. Allerdings war eine Version in ca. 120 Minuten analysiert und mögliche Korrelationen aufgedeckt. Da für frühere Versionen die Analyse jeweils nur einmal durchgeführt werden muss, kann gesagt werden, dass mit Hilfe der derzeitigen Werkzeugunterstützung ca. 10 - 12 Stunden für die Analyse einer Version benötigt werden. In naher Zukunft soll das gesamte Verfahren stärker automatisiert werden. Eine einheitliche Werkzeugumgebung, die die statischen Analysen des Quelltextes, das Identifizieren der Abhängigkeiten und ihrer Eigenschaften sowie die

statistischen Tests durchführt, wird derzeit erarbeitet. Das große Ziel ist es, den Integrationstestprozess in Projekten durch eine geeignete Unterstützung besser zu etablieren.

Literaturverzeichnis

[BBM96] Basili, V.R., Briand, L.C., Melo, W.L.: A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transaction on Software Engineering, Vol 22, No. 10, Oktober 1996

[Bi96] Binder, R.: "Testing Object-Oriented Software: A Survey" Journal of Software Testing, Verification and Reliability, Volume 6, Seiten 125-252, 1996.

[Bi00] Binder, R.: "Testing Object-Oriented Systems". Addison-Wesley, 2000.

[BIP07] Borner, L., Illes-Seifert, T., Paech, B.: "The Testing Process - A Decision Based Approach", International Conference on Software Engineering Advances (ICSEA) 25-31 Aug. 2007.

[CR05] Chakraborty, K., Remington, W.: "'Offshoring' of IT services: the impact on the US economy". J. Comput. Small Coll. 20, 4 (Apr. 2005), 112-125.

[JL02] Janssen, J., Laatz, W.: „Statistische Datenanalyse mit SPSS für Windows", Springer Verlag, 2002.

[Ma03] Mariani, L.: "A Fault Taxonomy for Component-based Software", Proceedings of the International Workshop on Test and Analysis of Component-Based Systems; TACoS 2003, September 2003.

[Me79] Meyers, G.J.: "The Art of Software Testing", John Wiley & Sons, New York, 1979.

[MSH03] Middendorf, S., Singer, R., Heid, J.: „JAVA Programmierhandbuch und Referenz", dpunkt.verlag, 2003.

[Or98] Orso, A.: "Integration Testing of Object-Oriented Software", Doktorarbeit, 1998.

[OW07] Ostrand, T.J., Weyuker, E.J.: How to Measure Success of Fault Prediction Models in Proceddings of the Fourth international workshop on Software quality assurance SOQUA 2007, September 2007

[SL06] Spillner, A., Linz, T.: "Basiswissen Softwaretest - Aus- und Weiterbildung zum Certified Tester", dpunkt.verlag, 2006.

[SPB+06] Suliman, D., Paech, B., Borner, L., Atkinson, C., Brenner, D., Merdes, M., Malaka, R.: "The MORABIT Approach to Runtime Component Testing", Proceedings of the 30th Annual International Computer Software and Applications Conference, 2006.

[SW02] Sneed, H., Winter, M.: "Testen objektorientierter Software – Das Praxishandbuch für den Test objektorientierter Client/Server - Systeme", Hanser Verlag, 2002.

[VR02] Vieira, M., Richardson, V.: "The Role of Dependencies in Component-based Systems Evolution" Proceedings of the International Workshop on Principles of Software Evolution 2002.

[ZPZ07] Zimmermann, T., Premraj, R., Zeller, A.: "Predicting Defects for Eclipse", Proceedings of PROMISE 2007.

[ZN08] Zimmermann, T., Nagappan, N.: „Predicting Defects using Nework Analysis on Dependency Graphs", Proceedings of International Conference on Software Engineering 2008.

[Ec08] Eclipse, www.eclipse.org, 2008

[SP08] SPSS, http://www.spss.com/, 2008