INAUGURAL – DISSERTATION

zur
Erlangung der Doktorwürde
der Naturwissenschaftlich-Mathematischen Gesamtfakultät
der Ruprecht – Karls – Universität
Heidelberg

vorgelegt von

Diplom-Informatiker Lars Borner

aus Hennigsdorf

Tag der mündlichen Prüfung: 03.03.2010

Integrationstest

Testprozess, Testfokus und Integrationsreihenfolge

Gutachter: Prof. Dr. Barbara Paech (Universität Heidelberg)
Prof. Dr. Peter Liggesmeyer (Universität Kaiserslautern)

Kurzbeschreibung

Im Integrationstest werden die Abhängigkeiten zwischen den Bausteinen eines Softwaresystems getestet. Die große Anzahl Abhängigkeiten heutiger Systeme stellt für die beteiligten Rollen des Integrationstests eine große Herausforderung dar. Die vorliegende Promotionsarbeit stellt neue und innovative Ansätze vor, um diese Rollen zu unterstützen.
Im ersten Teil der Arbeit wird ein Testprozess definiert, der die spezifischen Eigenheiten des Integrationstests berücksichtigt. Der definierte Integrationstestprozess setzt dabei seinen Schwerpunkt auf die im Prozess zu treffenden Entscheidungen. Er beschreibt, welche Entscheidungen in welcher Reihenfolge von welcher Rolle getroffen werden und welchen Einfluss diese Entscheidungen auf weitere Entscheidungen besitzen.
Im weiteren Verlauf der Arbeit werden neue Ansätze vorgestellt, die das Treffen von zwei Entscheidungen im Integrationstestprozess unterstützen: die Testfokusauswahl und die Integrationsreihenfolge.
Das Testen aller Abhängigkeiten ist aufgrund der Ressourcenbeschränkungen in realen Softwareprodukten nicht möglich. Die wenigen verfügbaren Ressourcen müssen daher für das Testen der fehleranfälligen Abhängigkeiten eingesetzt werden. Für das Identifizieren der fehleranfälligen Abhängigkeiten, und somit für die Testfokusauswahl, stellt die Promotionsarbeit einen neuen Ansatz vor. Der Ansatz verwendet Informationen über die Fehleranzahl von Bausteinen und die Eigenschaften von Abhängigkeiten aus früheren Versionen der zu integrierenden Software, um statistisch signifikante Zusammenhänge zwischen den Eigenschaften und der Fehleranzahl aufzudecken. Diese Zusammenhänge werden in der aktuellen Version ausgenutzt, um den Testfokus, d.h. die zu testenden Abhängigkeiten, auszuwählen.
Im Integrationstest werden Bausteine schrittweise zu einem Gesamtsystem zusammengesetzt, um die Lokalisation der Fehlerursache beim Auftreten eines Fehlers zu erleichtern. Der Nachteil dieses schrittweisen Vorgehens ist, dass Bausteine, die noch nicht integriert, aber für das Ausführen der Tests notwendig sind, simuliert werden müssen. Das Ziel ist es daher, eine Integrationsreihenfolge zu ermitteln, die einen minimalen Simulationsaufwand bedeutet. Zusätzlich sollten Abhängigkeiten, die als Testfokus ausgewählt wurden, frühzeitig integriert werden, um eventuelle Fehler frühzeitig aufzudecken. In dieser Promotionsarbeit wurde der erste Ansatz entwickelt, eine Integrationsreihenfolge zu ermitteln, die sowohl den Testfokus als auch den Simulationsaufwand berücksichtigt.
Die in der Arbeit entwickelten Ansätze wurden in Fallstudien mit mehreren realistisch großen Softwaresystemen evaluiert.

Abstract

The goal of integration testing is to test the dependencies between the components of a software system. The huge number of dependencies of today's systems challenges the roles participating in the integration testing process. This PhD thesis describes new and innovative approaches to support these roles.

The first part of the thesis defines a testing process that takes the specific characteristics of integration testing into account. This defined integration testing process focuses on the decisions to be made during the process. It describes the decisions that have to be made by individual roles. For every decision the dependent and depending decisions are worked out.

The second part of the thesis introduces new approaches to support two important decisions of the process: the test focus selection and the determination of the integration testing order.

Due to resource limitations of real software development projects, the testing of all dependencies is not possible. The few available resources have to be spent on error prone dependencies. This PhD thesis introduces a newly developed approach to identify these error prone dependencies and with that the test focus for the integration testing process. This new approach uses previous versions of a software system to uncover statistically significant correlations between the properties of a dependency and the number of errors of the participating components. These correlations are used to select the dependencies of the current version that have to be tested.

In integration testing the components of the systems are stepwise integrated to test the dependencies between them. This stepwise approach eases the detection of the error cause, in case an error is uncovered. The disadvantage of this approach is the simulation effort of components which have not yet been integrated, but are being used by components in the current integration step. Therefore, the goal is to find an integration testing order that causes a minimal simulation effort. Additionally, the order has to take into account the test focus, i.e. the dependencies that were selected as test focus have to be integrated as early as possible. This PhD thesis introduces a newly developed approach that determines an integration testing order which considers the test focus as well as the simulation effort. This approach uses heuristic algorithms like the Simulated Annealing or Genetic Algorithm.

All newly developed approaches were evaluated in several case studies. They were applied to real, large-sized software systems.

Danksagung

Allen Personen, die mich während der Entstehung dieser Arbeit begleitet haben, möchte ich an dieser Stelle meinen Dank ausdrücken.

An erster Stelle möchte ich mich bei Prof. Dr. Barbara Paech bedanken, die mich auf dem Weg der Themenfindung und Lösungserarbeitung sowohl geführt als auch begleitet hat. Die zahlreichen Anregungen und Diskussionen haben maßgeblich zum Entstehen dieser Arbeit beigetragen.

Großen Dank gilt auch Prof. Dr.-Ing. Peter Liggesmeyer für seine Bereitschaft, die vorliegende Promotionsarbeit zu begutachten.

Bei meinen ehemaligen Arbeitskollegen und Arbeitskolleginnen möchte ich mich für ihre produktive Zusammenarbeit in einer angenehmen Atmosphäre bedanken. Timea Illes-Seifert danke ich für die zahlreichen Gespräche und Diskussionen über die statistischen Tests und die Bereitstellung der Fehlerdaten ausgewählter Programme. Markus Oswald möchte ich für die entscheidenden Denkanstöße danken, die mich bei der Entwicklung des Ansatzes zur Integrationsreihenfolgeermittlung auf den richtigen Weg brachten. Frau Yulia Kosolapova danke ich für ihre Hilfe bei der Nachimplementierung der existierenden Algorithmen zur Integrationsreihenfolgeermittlung. Dr. Wilhelm Springer gebührt mein Dank für seine Unterstützung bei zahlreichen technischen Fragen und Problemen.

Den Mitgliedern des Arbeitskreises „Testen Objektorientierter Programme“ möchte ich für die Möglichkeit danken, frühe Ergebnisse meiner Arbeit dort zu präsentieren. Die Diskussionen im Anschluss an diese Präsentationen brachten viele neue Denkanstöße, die in neue Ideen und Ergebnisse der Arbeit einflossen.

Großer Dank gilt auch Michael Katzer und Holger Hanisch, die als kritische Korrekturleser viele kleinere Unklarheiten und Fehler in der Promotionsschrift aufdeckten.

Ganz herzlich bedanke ich mich abschließend bei meiner Frau, Anke Borner, für ihre Unterstützung, ihr Verständnis und ihre Geduld. Insbesondere möchte ich mich bei ihr für die Ermutigungen und Motivationsschübe bedanken, ohne die ich diese Arbeit nie hätte beenden können.

Inhaltsverzeichnis

1. Einleitung

In heutigen Softwareentwicklungsprozessen bildet die Qualitätssicherung einen wichtigen Bestandteil um die Qualität des Softwareprodukts zu erhöhen. Spillner schätzt in [SP00]: "*30 to 40% of all software activities are testing related*" (Seite 2) und wird in seiner Schätzung durch den NIST Report (National Institute of Standards & Technology) in [Ta02] bestätigt. Trotz dieses hohen Anteils an Qualitätssicherungsmaßnahmen im Entwicklungsprojekt ist es nicht möglich, ein Softwaresystem vollständig zu testen [Me79]. Aus diesem Grund müssen die zu testenden Bestandteile sorgfältig ausgewählt und darauf die Qualitätssicherungsmaßnahmen systematisch angewendet werden. Eine Möglichkeit, die Qualität in Softwaresystemen sicherzustellen, ist das Testen. Das Testen findet auf verschiedenen Teststufen statt. Diese Teststufen sind unter anderem der Unittest, der Integrationstest und der Systemtest [SL05]. Der Unittest legt seinen Schwerpunkt auf einzelne (Software-) Bausteine und prüft, ob diese ihre Funktionalität korrekt umsetzen. Der Integrationstest überprüft, ob die Abhängigkeiten zwischen den Bausteinen korrekt realisiert wurden. Der Systemtest überprüft, ob das fertige Gesamtsystem die Anforderungen an das System korrekt umsetzt.

Während für die Planung und die Durchführung des Unit- und Systemtests bereits eine Vielzahl von Ansätzen existiert (z.B. [Li90] oder [SBS08]) und in der Praxis eingesetzt wird, gibt es im Integrationstest noch viele offene Herausforderungen. Die vorliegende Promotionsarbeit beschäftigt sich mit einem Teil dieser Herausforderungen.

Das Ziel dieser Promotionsarbeit ist es, den Integrationstest durch neue Ansätze zu unterstützen. Diese Arbeit umfasst die Definition eines Integrationstestprozesses, eine Vorgehensweise zur Auswahl der zu testenden Systemteile für den Integrationstestprozess (=Abhängigkeiten) sowie einen neuen Ansatz zur Ermittlung der Integrationsreihenfolge. In dieser Promotionsarbeit werden existierende Testprozesse und ihre Eignung für den Integrationstestprozess betrachtet. Diese Testprozesse umfassen die an den Testprozessen beteiligten Rollen, die auszuführenden Aktivitäten und die zu erstellenden Artefakte. Diese Prozesse sind jedoch allgemein gehalten, um sie auf alle Teststufen anwenden zu können. Aufgrund der Tatsache, dass sich die allgemeinen Testprozesse auf alle Teststufen anwenden lassen, fehlt ihnen die Bezugnahme zu den Besonderheiten der einzelnen Teststufen. Die vorliegende Promotionsarbeit geht über den Inhalt der allgemeinen Testprozesse hinaus, indem sie diese Prozesse um Testentscheidungen erweitert. Das ermöglicht es, diese Entscheidungen bewusst zu treffen und explizit in den Testartefakten zu dokumentieren. Darüber hinaus definiert sie einen Testprozess speziell für den Integrationstest. Der Schwerpunkt dieses Integrationstestprozesses liegt dabei auf den speziell am Integrationstest beteiligten Rollen und den zu treffenden Entscheidungen. Die Entscheidungen werden in einer Hierarchie dargestellt. Die Hierarchie ist zum einen eine Vorgabe, in welcher Reihenfolge die Entscheidungen getroffen werden sollten. Zum anderen gibt sie Auskunft darüber, welche Entscheidungen durch frühere Entscheidungen beeinflusst werden.

Das Ziel eines jeden Testprozesses ist es, die zur Verfügung stehenden Ressourcen für den Test optimal auszunutzen, d.h. möglichst viele Fehler mit diesen Ressourcen zu finden. Zwei Entscheidungen innerhalb des definierten Integrationstestprozesses haben auf das Erreichen dieses Ziels einen großen Einfluss. Dies ist zum einen die Auswahl der zu testenden Abhängigkeiten innerhalb des Integrationstestprozesses, zum anderen die Entscheidung über eine geeignete Integrationsreihenfolge.

Da das vollständige Testen aller Abhängigkeiten nicht möglich ist (vgl. [Me79]), müssen die zu testenden Abhängigkeiten sorgfältig ausgewählt werden. Es werden die Abhängigkeiten ausgewählt, die eine erhöhte Wahrscheinlichkeit besitzen, fehlzuschlagen und somit fehleranfällig sind. Die zu testenden Abhängigkeiten werden als Testfokus bezeichnet. Das Festlegen des Testfokus für den Integrationstestprozess setzt detaillierte Kenntnisse über Eigenschaften der Abhängigkeiten voraus. Diese Abhängigkeitseigenschaften erlauben es, Abhängigkeiten näher zu beschreiben und zu klassifizieren. Die vorliegende Arbeit stellt einen Ansatz vor, der diese Eigenschaften ausnutzt, um den Testfokus für den Integrationstest auszuwählen. Der Ansatz ermöglicht es, Zusammenhänge zwischen Abhängigkeitseigenschaften und der Fehleranzahl der beteiligten Bausteine einer Abhängigkeit in früheren Versionen eines Softwaresystems aufzudecken. Diese Zusammenhänge werden in der aktuellen Version des Softwaresystems ausgenutzt, um die zu testenden Abhängigkeiten auszuwählen. Hierzu werden diejenigen Abhängigkeiten in der aktuellen Version ausgewählt, die Eigenschaften besitzen, die in früheren Versionen eine hohe aufgedeckte Fehleranzahl in den beteiligten Bausteinen der Abhängigkeiten aufwiesen. Zur Unterstützung dieses Ansatzes stellt die Arbeit einen Katalog von Eigenschaften bereit, die in Abhängigkeiten auftreten können.
Eine Integrationsreihenfolge beschreibt, in welcher Reihenfolge das System schrittweise integriert und getestet wird. Innerhalb eines Schrittes werden die Abhängigkeiten zwischen den neuen Bausteinen und den bereits integrierten Bausteinen getestet. Bausteine, die von bereits integrierten Bausteinen benötigt, aber noch nicht integriert sind, müssen simuliert werden. Das Simulieren von Bausteinen bedeutet zusätzlichen Aufwand innerhalb des Integrationstestprozesses. Aus diesem Grund sollte die Reihenfolge so gewählt werden, dass sie möglichst wenig Simulationen benötigt. Auf der anderen Seite sollen aber viele Fehler möglichst früh entdeckt werden. Aufgrund dessen muss eine Integrationsreihenfolge auch die zu testenden Abhängigkeiten berücksichtigen, die besonders fehleranfällig sind, und sie frühzeitig in die Testreihenfolge integrieren. Aktuelle Ansätze (z.B. [BLW03]) berücksichtigen bei der Integrationsreihenfolgeermittlung nur den Simulationsaufwand. Die zu testenden Abhängigkeiten werden nicht berücksichtigt. Diese Promotionsarbeit stellt einen neuen Ansatz zur Ermittlung der Integrationsreihenfolge vor. Der in dieser Arbeit neu entwickelte Ansatz bietet die Möglichkeit, eine Integrationsreihenfolge zu ermitteln, die zum einen die zu testenden Abhängigkeiten berücksichtigt und zum anderen den Aufwand des Simulierens nicht integrierter Bausteine minimiert.

1.1. Struktur der Arbeit

In Kapitel 2 werden wichtige Grundlagen für das Verständnis der Arbeit vorgestellt. Das Kapitel definiert wichtige Begriffe, die im Laufe der Arbeit häufig verwendet werden (2.1). Anschließend werden ausgewählte Probleme des Integrationstests erläutert (2.2). Sie dienen als Motivation für die in dieser Promotion bearbeiteten Themen. Kapitel 2.3 stellt ein Beispielsystem vor. Dieses Beispielsystem wird in der Arbeit häufiger verwendet, um die entwickelten Ansätze und Konzepte zu veranschaulichen.
Das 3. Kapitel befasst sich mit den Testprozessen. Im ersten Teil (3.1) wird ein allgemeiner Testprozess vorgestellt, der den Schwerpunkt auf die beteiligten Rollen und zu treffenden Entscheidungen legt. Kapitel 3.2 stellt den Integrationstestprozess und seine Besonderheiten vor. Auch hier liegt der Schwerpunkt auf den Rollen und den Entscheidungen. Das Kapitel 3.3 befasst sich mit Anwendungsmöglichkeiten des vorgestellten Integrationstestprozesses.
Der Katalog der Abhängigkeitseigenschaften wird in Kapitel 4 vorgestellt. Das Ziel ist es, Abhängigkeiten anhand der Eigenschaften klassifizieren zu können. Die einzelnen Eigenschaften werden im ersten Teil dieses Kapitels näher erläutert. Im Anschluss daran wird in Kapitel 4.2 ein Überblick über aktuelle Arbeiten gegeben. Es wird auf die

Literaturquellen näher eingegangen, die für die Erstellung des Katalogs Informationen geliefert haben. Abgeschlossen wird Kapitel 4 mit Einsatzmöglichkeiten des Katalogs.
In Kapitel 5 wird der neue Ansatz zur Testfokusauswahl vorgestellt. Da dieser Ansatz eine Reihe von statistischen Verfahren verwendet, gibt Kapitel 5.1 eine kurze Einführung in die wichtigsten statistischen Grundlagen. Das anschließende Kapitel 5.2 gibt einen Überblick über vergleichbare Arbeiten. Kapitel 5.3 stellt die einzelnen Schritte des Ansatzes im Detail vor. Die Anwendbarkeit des neuen Ansatzes wurde in zwei Fallstudien evaluiert. Die Beschreibung der Durchführung der Fallstudien sowie ihre Ergebnisse sind in Kapitel 5.4 zu finden.
Die Integrationsreihenfolge ist Schwerpunkt des 6. Kapitels. Es beginnt mit einem Überblick existierender Ansätze (6.1). Diese Ansätze werden kurz vorgestellt und klassifiziert. In Kapitel 6.2 wird ein neuer Ansatz vorgestellt, mit dem gemessen werden kann, wie gut eine gegebene Integrationsreihenfolge den Testfokus berücksichtigt. Für eine ausgewählte Anzahl existierender Algorithmen wird für die von diesen Algorithmen ermittelte Reihenfolge bestimmt, wie gut sie den Testfokus berücksichtigen. Im gleichen Kapitel wird ein weiterer Ansatz vorgestellt, der den Testfokus zu 100 Prozent berücksichtigt. Das letzte Unterkapitel des 6. Kapitels zeigt verschiedene Ansätze auf, wie sowohl der Testfokus als auch der Simulationsaufwand in einer Integrationsreihenfolge berücksichtigt werden können.
Kapitel 7 gibt eine kurze Beschreibung existierender Werkzeugunterstützungsmöglichkeiten. Das erste Unterkapitel stellt die Werkzeugunterstützung für die Testfokusauswahl vor. Kapitel 7.2 beschreibt die Werkzeugunterstützung für die Ermittlung der Integrationsreihenfolge.
Abgeschlossen wird diese Promotionsarbeit durch eine Zusammenfassung, die in Kapitel 8 zu finden ist. Dieses Kapitel gibt darüber hinaus einen Überblick über offene Forschungsfragen und zukünftige Forschungsaufgaben.

2. Grundlagen

Dieses Kapitel liefert Grundlagenwissen, das für das Verständnis der Arbeit notwendig ist. In Kapitel 2.1 werden wichtige Begriffe der Arbeit definiert. Das anschließende Kapitel 2.2 beschreibt aktuelle Probleme des Integrationstest. Diese Probleme motivieren die Inhalte der nachfolgenden Kapitel, indem ein Teil der aufgelisteten Probleme abgehandelt wird. Kapitel 2.3 beschreibt ein Beispielsoftwaresystem. In den späteren Kapiteln dieser Arbeit werden die erarbeiteten Konzepte anhand dieses Beispiels veranschaulicht.

2.1. Definitionen

Ein Softwaresystem besteht aus einer Vielzahl von **Bausteinen**, die voneinander abhängen. Innerhalb des Systems können Bausteine auf verschiedenen Abstraktionsebenen betrachtet werden. So kann ein Baustein ein Service, eine einzelne Klasse, eine Menge von Klassen (Klassencluster [Bi00]), ein Modul, eine Komponente, eine Systemschicht oder ein Teilsystem sein. Damit ein Softwaresystem seine Anforderungen erfüllen kann, existieren zwischen den Bausteinen **Abhängigkeiten**. Leo Savernik definiert in [Sa07] eine Abhängigkeit wie folgt: „*Eine Abhängigkeit zwischen zwei Artefakten liegt vor, wenn Artefakt A Artefakt B zu seiner korrekten Funktionsweise benötigt.*“ ([Sa07] S. 3). Die beiden Kernaussagen dieser Definition sind, dass Abhängigkeiten zum ersten die „*korrekte Funktionsweise“* von Bausteinen (Artefakten) ermöglichen und dass sie zweitens immer zwischen genau zwei Bausteinen existieren. Die dritte Aussage, die in der Definition nur implizit enthalten ist, ist die Richtung einer Abhängigkeit. Eine Abhängigkeit zwischen zwei Bausteinen ist immer gerichtet. Diese Tatsache ist mit der in [CL04] definierten Sichtweise vergleichbar, in der zwischen eingehenden und ausgehenden Abhängigkeiten unterschieden wird. Die beteiligten Bausteine nehmen dabei unterschiedliche Rollen ein (vgl. Abbildung 1 a). Es gibt einen abhängigen Baustein und einen unabhängigen Baustein. Der abhängige Baustein benötigt den unabhängigen für seine korrekte Funktionsweise. Aus Sicht des abhängigen Bausteins handelt es sich um eine ausgehende und aus Sicht des unabhängigen Bausteins um eine eingehende Abhängigkeit. Diese Abhängigkeit ist somit unidirektional.
Tritt die Konstellation auf, dass sowohl Baustein A von der Funktionsweise von Baustein B als auch Baustein B von der Funktionsweise von Baustein A abhängig ist (vgl. Abbildung 1 b), so werden beide Abhängigkeiten getrennt voneinander betrachtet, d.h. als zwei unabhängige Abhängigkeiten. Für die Abhängigkeit „*ab“* ist Baustein A der abhängige und Baustein B der unabhängige Baustein. Für die Abhängigkeit „*ba*“ ist es umgekehrt. Daraus ergibt sich auch, dass es zwischen zwei Bausteinen nicht mehr als zwei Abhängigkeiten geben kann.

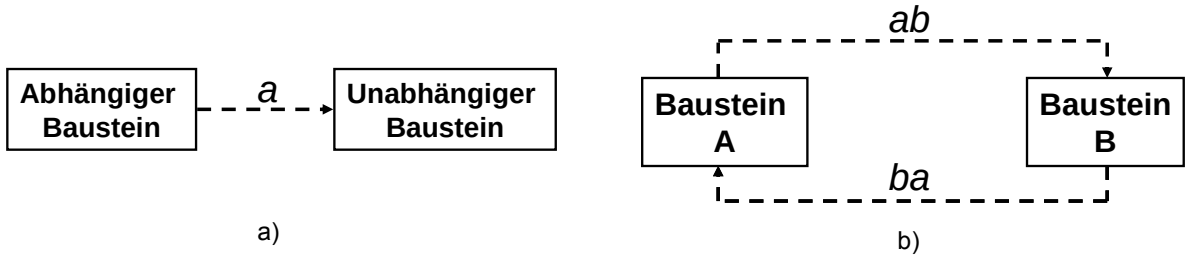


Abbildung 1: Abhängigkeit zwischen Bausteinen

Das vollständige Testen eines Systems ist nicht möglich. Daher müssen die zu testenden Teile des Systems sorgfältig ausgewählt werden. Die ausgewählten Systemteile werden als **Testfokus** bezeichnet. Im Integrationstestprozess sind die potenziell zu testenden Systemteile die Abhängigkeiten zwischen den Bausteinen. Aus der Gesamtmenge aller Abhängigkeiten müssen diejenigen ausgewählt werden, die dem Testfokus zugeteilt, d.h. die im Integrationstestprozess getestet werden. Diese Abhängigkeiten sind so auszuwählen, dass sie mit großer Wahrscheinlichkeit mindestens einen Fehler enthalten. In dieser Arbeit enthält eine Abhängigkeit einen Fehler, wenn mindestens einer der beiden beteiligten Bausteine der Abhängigkeit einen Fehler enthält. Diese Definition ist notwendig, da ein aufgetretener Fehler schwer einer Abhängigkeit zugeordnet werden kann. Das Zuordnen von Fehlern zu Bausteinen hingegen ist einfacher. Eine Abhängigkeit, die mindestens einen Fehler enthält, wird als **fehleranfällige Abhängigkeit** bezeichnet. Hierbei wird unterschieden, ob sich der Fehler der Abhängigkeit im abhängigen und/oder im unabhängigen Baustein der Abhängigkeit befindet. Diese Unterscheidung ist für den in Kapitel 5 vorgestellten Ansatz zur Testfokusauswahl notwendig.
Die korrekte Funktionsweise der Abhängigkeiten wird im Integrationstest geprüft. Der **Integrationstest** „*… insures the consistency of component interfaces and whether the components pass data and control correctly, which results in successful integration of dependent components*“ ([ER96], Seite 65). Mit anderen Worten ist der Integrationstest ein Test zwischen bereits getesteten Modulen [Be90], um Fehler „… *in Schnittstellen und im Zusammenspiel zwischen integrierten Komponenten zu finden*“ ([SL05], Seite 243). Der Integrationstest ist heutzutage notwendig, da Softwaresysteme sehr groß und komplex sind und daher nicht mehr von einzelnen Personen gehandhabt und entwickelt werden können. Die Bausteine, aus denen das spätere System besteht, werden von einer Vielzahl von Personen entwickelt. Durch die hohe Anzahl an beteiligten Personen ist zusätzliche Kommunikation zwischen den Beteiligten notwendig. Diese zusätzliche Kommunikation stellt eine neue Fehlerquelle dar, da Missverständnisse zwischen den beteiligten Personen auftreten können. Diese übertragen sich in die Bausteine und ihre Abhängigkeiten. Solche Missverständnisse bzw. die daraus resultierenden Fehler werden während der Integration sichtbar, denn eine „… *component integration can result in architectural mismatches when trying to assemble components with incompatible interaction behaviour*” ([BCI+00], Seite 220). Der Integrationstest kann diese Interkomponentenfehler [Bi00] systematisch aufdecken. Dies geschieht, indem das Softwaresystem schrittweise aus seinen Bausteinen zusammengesetzt wird [Be90], [Bi00]. Begonnen wird mit zwei Bausteinmengen. Die Abhängigkeiten zwischen den Bausteinen in den zwei Bausteinmengen werden getestet. Anschließend wird eine weitere Bausteinmenge zum integrierten System hinzugefügt und die Abhängigkeiten zwischen der neu hinzugefügten Bausteinmenge und den bereits integrierten Bausteinen überprüft. So wird weiter verfahren, bis alle Bausteine zum System hinzugefügt und die Abhängigkeiten systematisch überprüft wurden. Das Hinzufügen eines oder mehrerer Bausteine zu einer bereits integrierten Menge von Bausteinen und das Testen der Abhängigkeiten zwischen ihnen wird **Integrationsschritt** genannt. Innerhalb eines solchen Integrationsschritts kann eine beliebige Anzahl an Bausteinen gleichzeitig zum bereits integrierten Teilsystem hinzugefügt werden. Die Anzahl der Bausteine die in einem Integrationsschritt zum integrierten System hinzugefügt werden, wird als **Integrationsschrittgröße** bezeichnet. Im Idealfall ist die Integrationsschrittgröße 1, d.h. in jedem Integrationsschritt wird jeweils nur ein Baustein neu hinzugefügt und die Abhängigkeiten dieses Bausteins zu den bereits integrierten Bausteinen getestet.
Abhängig davon, in welcher Reihenfolge und mit welcher Integrationsschrittgröße das spätere System zusammengesetzt wird, kann eine Simulation von Bausteinen notwendig werden. Dies ist der Fall, wenn bereits integrierte Bausteine von (noch) nicht integrierten Bausteinen abhängig sind. Die nicht integrierten Bausteine werden durch Stubs (Platzhalter)

[SL05] simuliert. Diese Stubs müssen zusätzlich zu den „echten“ Bausteinen realisiert werden, wobei ihr Funktionsumfang geringer ist, als der Funktionsumfang ihrer Originale. Sie simulieren das Verhalten der Originale und ermöglichen so die Ausführung der bereits integrierten Bausteine.
Die Reihenfolge in der ein System aus seinen Bausteinen schrittweise zusammengesetzt wird, wird als **Integrationsreihenfolge** bezeichnet. Solche Integrationsreihenfolgen können Optimierungskriterien berücksichtigen. So ist beispielsweise eine möglichst geringe Anzahl an zu erstellenden Stubs ein Kriterium für eine gute Reihenfolge, um die Mehraufwand für die Erstellung dieser Stubs zu minimieren. Integrationsreihenfolgen können mit Integrationsstrategien ermittelt werden. Hierzu ist in der Literatur eine Reihe von Ansätzen zu finden. Ein Überblick über existierende Ansätze wird in Kapitel 6.1 gegeben.
Obwohl aktuell existierende Ansätze sich bei der Ermittlung der Integrationsreihenfolge nur auf die Integration von Bausteinen einer Abstraktionsebene konzentrieren ist es durchaus denkbar, auch Bausteine verschiedener Abstraktionsebenen in einem Integrationstestprozess schrittweise zu integrieren und die Abhängigkeiten zwischen ihnen testen.

2.2. Probleme des Integrationstests

Eines der größten Probleme im Integrationstest ist die Existenz von Abhängigkeiten zwischen einer Vielzahl von Bausteintypen. Der Integrationstest kann somit auf jeder Abstraktionsstufe angesiedelt werden, angefangen von den kleinsten Bausteinen eines Softwaresystems, den Services (bzw. Methoden, Funktionen, …), bis hin zu ganzen Softwaresystemen, die voneinander abhängig sind. Binder beispielsweise unterscheidet in [Bi00] vier Typen von Bausteinen in objektorientierten Systemen, die integriert werden müssen: Methoden, Klassen, Cluster und Teilsysteme. Daraus folgt, dass erst das Zusammenspiel der Methoden innerhalb einer Klasse getestet wird. Anschließend die Abhängigkeiten zwischen Klassen innerhalb eines Clusters (z.B. einer Komponente). Mehrere Cluster bilden anschließend ein Teilsystem. Die Abhängigkeiten zwischen den Clustern werden während des Zusammensetzens des Teilsystems überprüft. Abschließend wird das Gesamtsystem aus den Teilsystemen zusammengefügt. Das Zusammensetzen von Bausteinen zu Bausteinen höherer Abstraktionsebenen (z.B. Methoden zu Klassen) sollte stets mit einer Integrationsschrittgröße von eins geschehen.

Daraus ergibt sich auch eines der großen Probleme des Integrationstests. In jedem Schritt wird immer nur ein Baustein hinzugefügt und die Abhängigkeiten zwischen dem neu integrierten und den bereits integrierten Bausteinen überprüft [Be90]. Das Ziel des schrittweisen Vorgehens ist es, einen gefundenen Fehler in der Abhängigkeit schnell den fehlerhaften Bausteinen zuordnen zu können. Dieser Fehler wird mit großer Wahrscheinlichkeit im neu hinzugefügten Baustein oder in den Bausteinen, von denen der neue Baustein abhängig ist, zu finden sein. Integriert man das gesamte System in einem Integrationsschritt, d.h. werden alle Bausteine gleichzeitig zu einem System zusammengesetzt (Big-Bang Integration [SW02]), können die Abhängigkeiten nur unzureichend getestet und gefundene Fehler schwer dem verursachenden Baustein zugeordnet werden. Die Suche nach den fehlerhaften Bausteinen beim Aufdecken eines Fehlers erschwert sich drastisch. Existierende Forschungsarbeiten zielen darauf ab, Ansätze zu entwickeln, die es ermöglichen, eine Integrationsreihenfolge zu ermitteln, die möglichst einfach anzuwenden ist und das Testen der Abhängigkeiten mit möglichst wenigen zusätzlichen Ressourcen ermöglicht. Diese Ansätze beschäftigen sich also damit, die Anzahl der zu simulierenden Bausteine so gering wie möglich zu halten, und reduzieren somit den Simulationsaufwand.

Ein weiteres Problem im Integrationstest ist das Fehlen von Ansätzen, die Tester bei der Auswahl des Testfokus zu unterstützen. Integrationstester müssen sich auf die Teile der Software konzentrieren, die eine höhere Wahrscheinlichkeit besitzen, fehlzuschlagen und/oder die im Fehlerfall zu weitreichenden Schäden (Personenschäden, finanzielle Schäden …) führen. Aus beiden Werten kann ein Risiko bestimmt werden (Schaden * Wahrscheinlichkeit). Die Systemteile mit einem hohen Risiko müssen dann getestet werden. In [ER96], [Or98] weisen die Autoren darauf hin, dass auch die Integrationsreihenfolge so gewählt werden sollte, dass kritische Bausteine möglichst früh integriert werden. Jedoch gibt es keinen Hinweis darauf, wie die Kritikalität für die Bausteine ermittelt werden kann. In dieser Arbeit wird diese Idee von [ER96] und [Or98] aufgegriffen. Jedoch wird anstatt, die Kritikalität von Bausteinen im Integrationstest zu verwenden, die Kritikalität von Abhängigkeiten berücksichtigt, da im Integrationstest weniger die Bausteine, sondern vielmehr die Abhängigkeiten überprüft werden. Hier stellt sich jetzt die Frage, ob es eine Möglichkeit gibt, Abhängigkeiten auszuwählen, die eine höhere Fehlerwahrscheinlichkeit in sich tragen. Da der Schaden für das Fehlschlagen einer Abhängigkeit nur schwer abzuschätzen ist, konzentriert sich diese Arbeit auf die Wahrscheinlichkeit, dass eine Abhängigkeit fehleranfällig ist. Um jedoch die Wahrscheinlichkeit für Abhängigkeiten abschätzen zu können, sind Informationen über die Abhängigkeiten notwendig.

Ein viertes Problem des Integrationstestprozesses ergibt sich zum einen aus der unzureichenden Existenz von Informationen über Abhängigkeiten zwischen Bausteinen in den Entwicklungsartefakten und zum anderen aus der Streuung der wenigen Informationen über eine Vielzahl von Entwicklungsartefakten. Zu diesen Entwicklungsartefakten zählen Architekturmodelle, Entwurfsmodelle, Quelltexte, Projektpläne und Anforderungen. **Architekturmodelle** umfassen alle Modelle, die verwendet werden, um die Architektur eines Softwaresystems zu spezifizieren. Sie umfassen unter anderem die in der UML [UML09] gebräuchlichen Deploymentdiagramme, Komponentendiagramme, Architektur-beschreibungssprachen oder natürlichsprachlichen Text. **Entwurfsmodelle** dienen als Vorlage für die Implementierung und beschreiben die innere Struktur der späteren Software und ihrer Bausteine. Dies können Klassen-, Sequenz- oder Aktivitätsdiagramme, aber auch Schnittstellen- und Kontraktbeschreibungen sein. **Quelltexte** liefern Informationen über die tatsächliche innere Struktur der einzelnen Bausteine und ihrer Abhängigkeiten. Hierzu gehören auch die Informationen aus dem Softwarekonfigurationsmanagement (z.B. CVS [CV09], SVN [Su09]). **Projektpläne** dokumentieren wichtige Informationen, die sich nicht aus der Beschreibung des späteren Softwaresystems ableiten lassen. Sie enthalten Informationen über den Fertigstellungszeitpunkt von Bausteinen, das eingesetzte Personal, Kosten, usw. **Anforderungen** spezifizieren, was das zu integrierende Softwaresystem erfüllen muss. Sie sind häufig in natürlicher Sprache verfasst. Sie können Informationen über kritische (oder wichtige) Funktionen der Software liefern.

Ein weiteres Problem für den Integrationstester ist die fehlende Möglichkeit, Abhängigkeiten systematisch nach ihren Eigenschaften klassifizieren zu können. Eine Abhängigkeit kann eine Vielzahl von Eigenschaften besitzen. Informationen über diese Abhängigkeitseigenschaften sind für die Integrationstester von großem Wert. Sie liefern Hinweise, was während einer Abhängigkeit für Fehler auftreten können und wie schwierig es ist, eine solche Abhängigkeit zu simulieren, falls einer der Bausteine im aktuellen Integrationsschritt noch nicht integriert worden ist.

In [ER96] und [Or98] wird zwar beschrieben, dass die Integration in der Reihenfolge der Kritikalität der Bausteine erfolgen soll. Sie berücksichtigen jedoch nicht, dass sich dadurch

die Anzahl der zu simulierenden Bausteine unter Umständen stark erhöhen kann, wodurch die benötigten Ressourcen für den Test stark ansteigen. Es fehlt ein guter Mittelweg für die Integrationsreihenfolge, der die Kritikalität der Abhängigkeiten und den Aufwand für das Simulieren von nicht integrierten Bausteinen berücksichtigt.

Zur Unterstützung der Testprozesse auf beliebiger Teststufe stellen Spillner und Linz in [SL05] einen allgemeinen Testprozess vor, der die wichtigsten Testaktivitäten, die in jedem Testprozess auf jeder Teststufe ausgeführt werden sollten, einbezieht. Da der Prozess auf alle Teststufen anwendbar ist, kann er nicht die Eigenheiten der einzelnen Teststufen berücksichtigen. Für den Integrationstest bedeutet dies, dass es keinen definierten Testprozess gibt, der sich speziell mit der Problematik des Integrationstest auseinander setzt. Spezielle, auszuführende Aktivitäten oder zu treffende Entscheidungen, die ausschließlich im Integrationstestprozess anzutreffen sind, werden in [SL05] und auch in anderen Arbeiten nicht berücksichtigt.

2.3. Beispielsystem Onlineumfrage

Die in dieser Arbeit vorgestellten Ansätze werden an vielen Stellen anhand eines Beispielsystems veranschaulicht. Als Beispielsystem findet die technische Realisierung einer Onlineumfrage des Arbeitskreises „Testen objektorientierter Programme“ der Gesellschaft für Informatik e.V. Verwendung ([Too09]). Diese Onlineumfrage hatte die Erfassung von Fehlerhäufigkeiten in objektorientierten Systemen zum Ziel. Sie wurde vom 19. August bis 31. Oktober 2005 durchgeführt. An ihr nahmen 1219 Personen teil. Die Teilnehmerinnen hatten die Möglichkeit, mit Hilfe des WEB-Browsers die Häufigkeiten von vorgegebenen Fehlerkategorien zu bewerten. Eine genaue Beschreibung sowie die Basisauswertungen können in [BFH+06] gefunden werden. Weitergehende Auswertungen sind in [BEJ+08] nachzulesen.
Die Anforderungen an den Onlinefragebogen wurden im Rahmen vieler Diskussionen im Arbeitskreis identifiziert und festgelegt. Hierbei sollte in erster Linie die Möglichkeit bestehen, die Umfrage mit Hilfe eines „handelsüblichen“ *Web-Browser* durchzuführen. Um innerhalb des kurzen Zeitraums vielen Personen die Teilnahme zu ermöglichen, mussten diese auch gleichzeitig auf das System zugreifen können. Das System musste daher in hohem Maß die *parallelen* Beantwortungen des Fragebogens unterstützen. Weiterhin sollte es leicht möglich sein, die zu bewertenden *Fehlerkategorien dynamisch* in das System einzubinden, ohne den Quelltext des Systems verändern zu müssen. Die einzelnen Antworten sollten in einem Format so gespeichert werden, dass sie von verschiedenen Analysewerkzeugen leicht eingelesen und verarbeitet werden können. Dies waren die Werkzeuge Microsoft Excel [Ex09] und SPSS [SP09].
Um die geforderten Anforderungen zu realisieren, fiel die Entscheidung auf eine Two-Tier Client/Server Architektur mit einem Thin Client [Bu02]. Die Architektur ist in Abbildung 2 dargestellt. Der Web-Browser auf dem Client ist für das Anzeigen der HTML[1]-Inhalte, die ihm der Server liefert, verantwortlich. Als Server wurde ein Apache Tomcat-Container [Tom09] verwendet.
Die Bausteine innerhalb des Containers sind für die Erzeugung des HTML-Quelltextes verantwortlich. Die Definition der Fehlerkategorien, die von den Teilnehmerinnen zu bewerten waren, konnten mit Hilfe einer XML[2]-Datei außerhalb des Systems definiert und von den Elementen im Tomcat-Container eingelesen und verarbeitet werden. Die Antworten

[1] HTML … Hyper Text Markup Language
[2] XML … Extensible Markup Language

der Teilnehmer wurden in eine externe Datei im CSV[1]-Format exportiert. Dieses Format kann sowohl von Excel als auch von SPSS importiert und weiter verarbeitet werden.

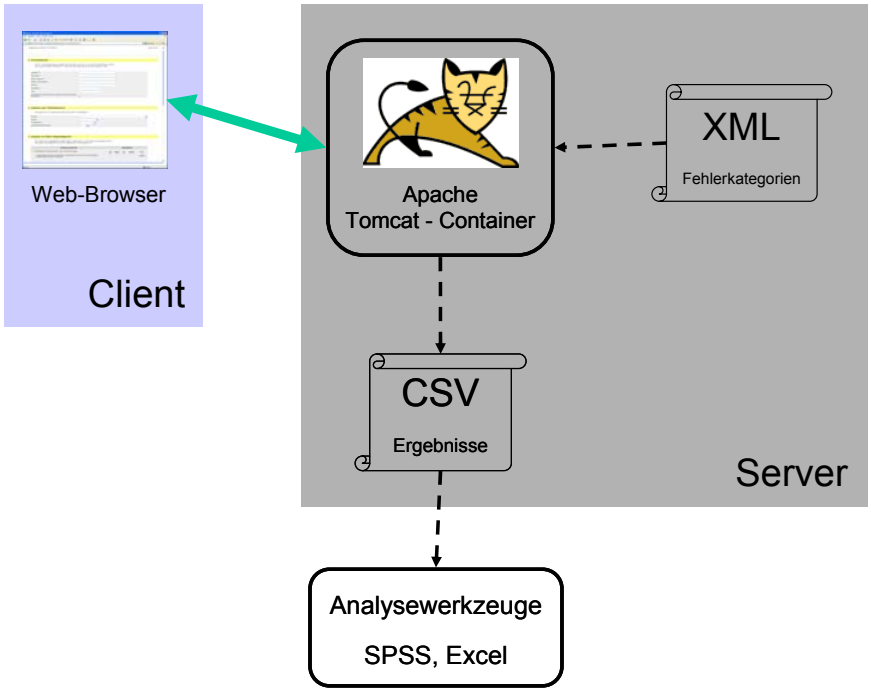


Abbildung 2: Architektur der Onlineumfrage

Bei den Bausteinen innerhalb des Tomcat-Containers handelt es sich um Servlets [Se09] und Java-Klassen [Ja09]. Das System besteht aus sechs Klassen strukturiert in drei Paketen. Das Klassendiagramm in Abbildung 3 stellt die einzelnen Klassen und ihre Beziehungen untereinander dar. Die Klassen, die zur Anwendung gehören und selbst implementiert wurden, sind rot gekennzeichnet, Klassen und Schnittstellen in externen Standardbibliotheken sind hingegen grün dargestellt. Blaue Bausteine repräsentieren externe Dateien, die von der Anwendung gelesen oder geschrieben werden. Die Pakete sind durch graue Kästen dargestellt.
Die Klasse `QuestionnaireServlet` ist die wichtigste Klasse. Sie ist für das Erzeugen der HTML-Inhalte und somit für das Anzeigen der Inhalte des Fragebogens verantwortlich. Zur besseren Trennung von Java-Quelltext und HTML-Quelltext wurde die Klasse `HTML` eingeführt, die den HTML-Quelltext in Java-Operationen kapselt, welche von weiteren Klassen verwendet werden können. Die Klasse `XMFileLParser` liest die Fehlerkategorien aus der XML-Datei aus und erstellt daraus eine Liste (`ErrorCategoryList`) mit zu bewertenden Fehlerkategorien (`ErrorCategory`). Mit Hilfe der Klasse `Result` können die Antworten der Teilnehmer in die CSV-Datei geschrieben werden. Die Beziehungen zu anderen Klassen (Standard-Java-Klassen) und Bausteinen (Externe Dateien) sind ebenfalls eingezeichnet.
Die einzelnen Klassen sind in einer Schichtenarchitektur [So07], [HR02] strukturiert, wobei die Klassen `HTML` und `QuestionnaireServlet` die Oberflächenschicht darstellen. Die Oberflächenschicht kommuniziert mit der darunter liegenden Anwendungsschicht, welche für das Einlesen der zu bewertenden Fehlerkategorien (`XMLFileParser`, `ErrorCategoryList`, `ErrorCategory`) und für das Bereitstellen der Speicherfunktionalität (`Result`) verantwortlich ist. Hierbei gibt es nur eine

[1] CSV … Comma Separated Values

Kommunikationsrichtung zwischen den beiden Schichten: von oben nach unten. D.h. die Klassen aus der Oberflächenschicht kommunizieren nur mit den Klassen aus der Anwendungsschicht, jedoch nicht umgekehrt.
Das Beispiel des Onlinefragebogens wird innerhalb der nachfolgenden Kapitel verwendet, um die Konzepte des Integrationstestprozesses (Kapitel 3.2), der Abhängigkeitseigenschaften (Kapitel 4), der Testfokusauswahl (Kapitel 5), und der Integrationsreihenfolge (Kapitel 6) zu veranschaulichen.

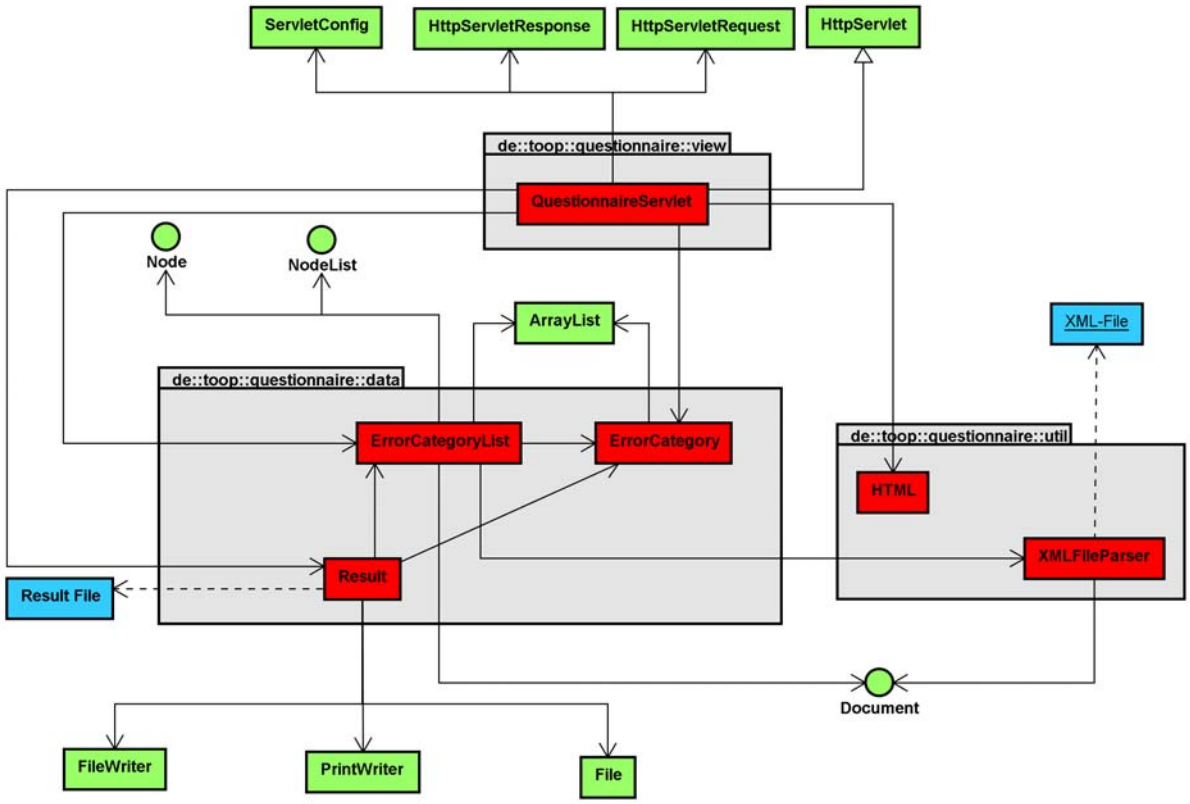


Abbildung 3: Interne Struktur des Onlinefragebogens als Klassendiagramm

3. Der Testprozess

Die erfolgreiche Durchführung eines Testprozesses innerhalb eines großen Softwarenentwicklungsprojekts setzt einen wohl definierten Prozess voraus. Der Prozess definiert, welche Rollen beteiligt sind, welche Aktivitäten ausgeführt und welche Entscheidungen getroffen werden müssen.
Während der Betrachtung eines Testprozesses müssen zwei Dimensionen unterschieden werden. Die erste Dimension beschäftigt sich mit der Eingliederung des Testprozesses in den Softwareentwicklungsprozess und bezeichnet die Gesamtheit aller Teststufen [BD04]. Im V-Modell nach [Bo79] ist dies der komplette rechte Ast, im W-Modell nach [Sp01] und [Sp02] der mittlere Teil des W's. Dieser Testprozess legt den Fokus auf die Koordination, Planung und Durchführung der Testaktivitäten über Teststufen hinweg. Die zweite Dimension betrachtet den Testprozess innerhalb einer Teststufe. Er konzentriert sich auf die Rollen, Entscheidungen und Aktivitäten für das „eigentliche" Testen der Entwicklungsartefakte. Nachfolgend bezieht sich der Begriff „Testprozess" auf den Testprozess innerhalb einer Teststufe. Testprozesse über Teststufen hinweg werden nicht betrachtet.
Für jede Teststufe sollte ein eigener Testprozess definiert und verwendet werden, z.B. ein Unittest-, System-, Abnahme- oder auch Integrationstestprozess. In [SL05] wird ein allgemeiner Testprozess beschrieben, der sich auf jede Teststufe anwenden lässt, aber so allgemein gehalten ist, dass er die spezifischen Eigenheiten der jeweiligen Teststufen nicht berücksichtigt. Weiterhin gibt der Prozess auch keine Hinweise darauf, welche Entscheidungen im Testprozess zu treffen sind.
Das vorliegende Kapitel beschäftigt sich speziell mit dem Integrationstestprozess und seinen Eigenheiten. Zuvor jedoch werden die gemeinsamen Rollen, Aktivitäten und Entscheidungen eines generischen Testprozesses vorgestellt, d.h. eines Testprozesses, der als Vorlage für einen Testprozess auf einer beliebigen Teststufe angewendet werden kann. Das Ziel dieses generischen Prozesses ist es, als „Schablone" für die jeweiligen Testprozesse der einzelnen Teststufen zu dienen, indem Eigenheiten der betrachteten Teststufe ergänzt werden. Ähnlich dem Muster *Abstrakte Oberklasse* [GHJ+01] in objektorientierten Softwaresystemen dient der generische Testprozess als eine Art abstrakte Schablone, welche bereits Rollen, Aktivitäten und Entscheidungen vorgibt, die in den einzelnen Unterklassen (die einzelnen konkreten Teststufen-Testprozesse) spezialisiert werden.
Im Nachfolgenden wird ein generischer Testprozess vorgestellt, der seinen Schwerpunkt auf die zu treffenden Entscheidungen legt. Dies ermöglicht es, die Entscheidungen, die während der Durchführung eines Testprozesses getroffen werden müssen, bewusst zu treffen und explizit in den Testartefakten zu dokumentieren. Die Betrachtung der Entscheidungen im Testprozess dient der Erweiterung und Konkretisierung der existierenden Testprozessbeschreibungen (z.B. in [SL05], [PKS00]), die die auszuführenden Testaktivitäten betrachten.
Im Anschluss wird die Spezialisierung dieses generischen Testprozesses für den Integrationstestprozess vorgestellt.

3.1. Generischer Testprozess

In verschiedenen Literaturquellen können Informationen über den Testprozess gefunden werden. Diese Informationen konzentrieren sich auf die im Testprozess beteiligten Rollen, die auszuführenden Phasen und Aktivitäten sowie vereinzelt auf die zu erstellenden

Testartefakte. Wichtige, in der Literatur genannte, Testaktivitäten sind die *Testplanung* ([Be90], [KFN99], [BD04]), der *Testentwurf* ([Be90], [Bi00], [GSW03], [FG99]), die *Testimplementierung* ([Be90], [Bi00], [FG99]), die *Testausführung* ([Be90], [Bi00], [GSW03], [FG99]) und die *Testauswertung* ([GSW03], [FG99]). Eine Einbettung der Aktivitäten „Testplanung“ und „Testausführung“ in den Softwareentwicklungsprozess ist im W-Modell dargestellt ([Sp01], [Sp02]). Ein durchgängiges Testprozessmodell, das alle Aktivitäten des Testprozesses enthält und die Abhängigkeiten zwischen diesen darstellt, kann in [SL05] gefunden werden (vgl. Abbildung 4). Die Autoren vereinigen die in älteren Literaturquellen verstreut gefundenen Aktivitäten zu einem durchgängigen Testprozess. Der dort beschriebene Testprozess beginnt mit der *Planung und Steuerung*, gefolgt von *Analyse und Design*, *Realisierung und Durchführung* und wird nach *Auswertung und Bericht* beendet. Hierbei wird der Testprozess als eigenständiges „Teilprojekt“ behandelt, welches einen eigenen Projektstart (Beginn) und einen Projektende (Ende) besitzt. Die Aktivität *Planung und Steuerung* wird parallel zu den übrigen ausgeführt, um die Durchführung des Prozesses zu überwachen und gegebenenfalls steuernd einzugreifen.

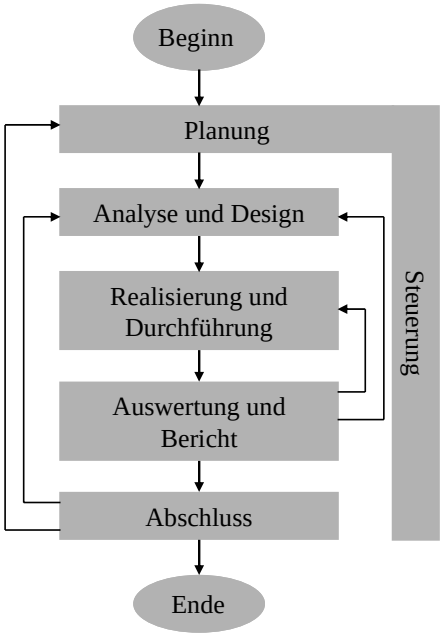


Abbildung 4: Testprozess nach [SL05]

Die in [SL05] dargestellten Testaktivitäten wurden in [IHP+05] im Rahmen der Identifikation von Kriterien für die Testwerkzeugauswahl um einige Testaktivitäten erweitert. Die Autoren fügten die Aktivitäten *capturing and comparing test results*, *reporting results*, *tracking software problem reports/defects* und *managing the test ware* in den Testprozess ein. Ergänzend zu diesen neuen Aktivitäten wurden in [IHP+05] die beteiligten Rollen des Testprozesses identifiziert. Diese sind der *Testmanager, Testdesigner, Testautomatisierer, Testkonfigurationsmanager* und *der Testadministrator*.

Der nachfolgend vorgestellte generische Testprozess ([BIP07a], [BIP07b]) basiert auf den Beschreibungen in [PKS00], [IHP+05] sowie [SL05] und ergänzt diese um Entscheidungen, die innerhalb des Prozesses getroffen werden. Die Entscheidungen werden den verantwortlichen Rollen und den Aktivitäten, in denen sie getroffen werden, zugeordnet. Hierfür werden in 3.1.1 die beteiligten Testrollen und in 3.1.2 die Testaktivitäten des generischen Testprozesses vorgestellt. 3.1.3 beschreibt im Detail, welche Entscheidungen von wem, in welchen Aktivitäten getroffen werden und welche Informationen für das Treffen der Entscheidungen notwendig sind.

3.1.1. Testrollen

Am Testprozess sind verschiedene Rollen beteiligt, welche in *ausführende Testrollen* und *unterstützende Rollen* unterschieden werden können. Eine ausführende Testrolle ist für die Ausführung von Testaktivitäten sowie für das Treffen von Testentscheidungen verantwortlich. Eine unterstützende Rolle ist eine Rolle, die nicht direkt an der Ausführung der Testaktivitäten bzw. für das Treffen von Testentscheidungen verantwortlich ist. Sie unterstützt die ausführenden Testrollen, indem sie Informationen und Entwicklungsartefakte für das Treffen von Entscheidungen zur Verfügung stellt. Die in [IHP05] beschriebenen Rollen des Testmanagers, Testdesigners, Testautomatisierers, Testkonfigurationsmanagers und Testadministrators sind im Kontext dieser Arbeit ausführende Testrollen. Anforderungsingenieure, Architekten, Entwickler, Programmierer und Projektmanager stellen hingegen Informationen und Entwicklungsartefakte für das Ausführen von Testaktivitäten und das Treffen von Entscheidungen zur Verfügung und nehmen somit die unterstützende Rolle ein.
Die ausführenden Testrollen des Testprozesses werden nachfolgend näher beschrieben und auf ihre Verantwortungsbereiche näher eingegangen. Die Rollen des Testautomatisierers, des Testkonfigurationsmanagers und des Testadministrators aus [IHP05] werden dabei im Rahmen dieser Arbeit zu einer gemeinsamen Rolle „*Tester*“ zusammengefasst.
Im Idealfall nimmt in einem Entwicklungsprojekt eine Person genau eine Rolle ein. In Projekten mit wenigen Personen ist es jedoch möglich, dass eine Person auch mehrere Rollen innehaben kann.

3.1.1.1 Testmanager

Die Aufgabe des Testmanagers[1] ist es, die verschiedenen Testaktivitäten zu koordinieren und den weiteren Rollen die Ausführung ihrer Aufgaben zu ermöglichen. Er legt fest, welche Bestandteile der Software wie intensiv zu testen sind. Weiterführend ist er für die *Planung und Steuerung* der verschiedenen Testaktivitäten verantwortlich. D.h. er legt fest, wann welche Aktivitäten ausgeführt werden und welche Ressourcen hierfür zur Verfügung stehen. Am Ende des Testprozesses ist er für die *Auswertung und den Bericht* sowie für den *Abschluss* verantwortlich. Er priorisiert die gefundenen Fehler, wertet den aktuell durchgelaufenen Testzyklus aus und entscheidet, ob die Testaktivitäten abgeschlossen werden können oder ob weitere Testaktivitäten ausgeführt werden müssen.

3.1.1.2 Testdesigner

Der Testdesigner ist in nahezu alle Aktivitäten und Entscheidungen des Testprozesses involviert. Er unterstützt den Testmanager beim Festlegen der zu testenden Teile und bei der Priorisierung der gefundenen Fehler. Er ist vorrangig für die *Analyse* der Testbasis und das *Design* der Testfälle verantwortlich. Darüber hinaus werden von ihm die Testfälle priorisiert und in entsprechender Weise vorbereitet, so dass sie vom Tester realisiert und ausgeführt werden können. Hierzu spezifiziert der Testdesigner neben den Testfällen auch die notwendige Testumgebung für die Testläufe.

3.1.1.3 Tester

Der Tester ist für die *Realisierung* und *Durchführung* der Testfälle und des Management der realisierten Testfälle verantwortlich. Er realisiert, verwaltet und konfiguriert darüber hinaus die notwendigen Testumgebungen für die einzelnen Testfälle. Während und nach der

[1] Im weiteren Verlauf wird im Text die männliche Form des Rollennames verwendet. Es ist damit jedoch sowohl die weiblich und männliche Form einzuschließen.

Durchführung der Testfälle wertet er die Ergebnisse aus und fasst sie in einem Bericht für den Testmanager zusammen.

3.1.2. Testaktivitäten

Die in Abbildung 4 zusammengefassten Testaktivitäten werden nachfolgend kurz vorgestellt. Für eine detailliertere Beschreibung wird auf [IHP+05] und [SL05] verwiesen.

3.1.2.1 Planung und Steuerung

Im Rahmen dieser Testaktivität plant der Testmanager die notwendigen Ressourcen für den Testprozess, d.h. wie viel finanzielle Mittel und Zeit stehen zur Verfügung oder welche Mitarbeiter werden benötigt. Die Ergebnisse der Testplanung werden im Testkonzept (vgl. [SL05]) festgehalten. Die Aktivität des Planens umfasst darüber hinaus das Festlegen der Teststrategie. Sie beschreibt, wie die vorhandenen Ressourcen auf die zu testenden Systemteile verteilt werden. Grundlage hierfür ist eine Risikoeinschätzung, welche sich aus dem „*… erwarteten Risiko und der Schwere der Auswirkung beim Eintreten einer Fehlerwirkung …*“ ([SP06] S. 21) ergibt. Der Testmanager legt in dieser Aktivität auch die Testintensität der einzelnen Teilsysteme fest, priorisiert die Testaktivitäten und Tests, wählt die einzusetzenden Werkzeuge aus, definiert Metriken, um den Testprozess zu überwachen und zu steuern und definiert Testendekriterien. Der Testmanager synchronisiert ergänzend die Kommunikation und den Informationsfluss mit den unterstützenden Rollen im Softwareentwicklungsprozess.

3.1.2.2 Analyse und Design

Der Testdesigner prüft im Rahmen dieser Aktivität, ob die Testbasis alle Informationen enthält, um daraus Testfälle abzuleiten. Basierend auf der Teststrategie aus der Aktivität „Planung und Steuerung“ werden Testentwurfstechniken ausgewählt, um mit Hilfe der Testbasis die logischen Testfälle und Testdaten abzuleiten. Diese Testfälle beinhalten Informationen über die Vorbedingung, die einzelnen Test- und Prüfschritte in abstrakter Form und ein Beschreibung der Nachbedingung. Die Prüfschritte beschreiben die erwartete Reaktion des Testobjekts auf die in den Testschritten durchgeführten Aktionen. Nach einer Später in der Aktivität „Realisierung und Durchführung“ werden die tatsächlichen Reaktionen des Testobjekts mit den erwarteten Ergebnissen des Testobjekts verglichen.
Im Gegensatz zum Ansatz von [SL05] werden die konkreten Testfälle und Testdaten im vorliegenden Testprozess (wie auch in [IHP+05]) erst zum Zeitpunkt der Realisierung der Testfälle vollständig definiert, da sie erst zu diesem Zeitpunkt benötigt werden und so flexibler auf Abweichungen zwischen der Testbasis und dem tatsächlichen Testobjekt reagiert werden kann.

3.1.2.3 Realisierung und Durchführung

Wie bereits in 3.1.2.2 erwähnt, werden in dieser Aktivität die logischen Testfälle und -daten durch konkrete Testfälle und -daten verfeinert. Diese Testfälle können anschließend durch den Tester für die Testausführung in Form von Testskripts realisiert werden, um sie automatisiert ausführen zu können. Es ist jedoch auch möglich, die Testfälle manuell auszuführen. In den meisten Fällen wird für die Ausführung der Testfälle eine spezielle Testumgebung benötigt. Dies umfasst zum einen das Simulieren von nicht verfügbaren Systemteilen, z.B. weil sie noch nicht fertig gestellt wurden oder weil Systemteile aus der späteren Systemumgebung im Testumfeld nicht vorhanden sind. Zum anderen müssen Monitore realisiert werden, die die Reaktionen und die Ergebnisse des Testobjekts aufzeichnen, um sie später mit den erwarteten Reaktionen vergleichen zu können.

Nachdem die Testumgebung und die Testfälle realisiert wurden, werden die Testfälle ausgeführt. Hierbei sollten nach [SL05] zuerst die Testfälle ausgeführt werden, die die Hauptfunktionen des Testobjekts überprüfen, um sicherzustellen, dass diese lauffähig sind und die nachfolgenden Testfälle nicht blockiert werden. Ein weiterer Einflussfaktor auf die Ausführungsreihenfolge der Testfälle ist die Priorität der Tests, die der Testmanager in der Aktivität „Planung und Steuerung“ festgelegt hat.
Für jeden Testlauf, d.h. für jeden ausgeführten Testfall wird ein Testprotokoll erstellt, das die tatsächliche Reaktion des Testobjekts dokumentiert. Ohne ein Testprotokoll besitzt ein Testlauf keine Gültigkeit. Mit Hilfe des Testprotokolls kann überprüft werden, ob das Testobjekt das erwartete Verhalten im Testfall gezeigt hat oder ob es eine Abweichung gab.

3.1.2.4 Auswertung und Bericht

Auf Basis der Testprotokolle können Testdesigner, Tester und Testmanager im Anschluss festlegen, ob eine Abweichung zwischen dem erwarteten Soll-Verhalten und dem beobachteten Ist-Verhalten aufgetreten ist. Bei einer Abweichung müssen sie feststellen, ob es sich bei der Abweichung um einen Fehler im Testobjekt, im Testfall und seiner Realisierung oder in der Dokumentation des Testobjekts (Testbasis) handelt. Im ersteren Fall war der Testfall erfolgreich, da er einen Fehler im Testobjekt selbst aufgedeckt hat. Im zweiten Fall müssen die Testfälle korrigiert und der Testfall erneut ausgeführt werden. Im letzen Fall, d.h. im Fall, dass die Testbasis fehlerhaft ist, müssen die Testbasis und gegebenenfalls auch die Testfälle korrigiert und anschließend die Testfälle neu ausgeführt werden.
Im Rahmen dieser Aktivität werden alle Testprotokolle aller Testläufe betrachtet und für jeden Testlauf entschieden, ob er einen Fehler im Testobjekt aufgedeckt hat oder nicht. Die Erkenntnisse werden in einem Bericht zusammengefasst, den der Testmanager in der nächsten Testaktivität nutzen kann, um festzustellen, ob der Testprozess für diese Teststufe abgeschlossen werden kann oder weitere Tests notwendig sind.

3.1.2.5 Abschluss

In dieser Testaktivität prüft der Testmanager auf Basis des Testberichts aus der vorangegangenen Aktivität, ob der Testprozess für diese Teststufe abgeschlossen werden kann oder ob er teilweise wiederholt werden muss. Hierzu überprüft er, ob die in der Aktivität „Planung und Steuerung“ festgelegten Testendekriterien erreicht worden sind. Sind die Testendekriterien erreicht, so erstellt er eine Zusammenfassung des durchgeführten Testprozesses. Anschließend kann der Testprozess beendet werden. Sind die Testendekriterien jedoch nicht erreicht worden, muss er festlegen, welche Testaktivitäten erneut durchgeführt werden müssen, um die Testendekriterien zu erreichen. Anschließend wird der Testprozess mit der jeweiligen Testaktivität fortgesetzt.

3.1.3. Entscheidungen

Neben den verantwortlichen Rollen und auszuführenden Testaktivitäten stellen die im Testprozess zu treffenden Testentscheidungen eine neue Sichtweise auf den Testprozess dar.

Definition: *Eine **Entscheidung** bezeichnet eine Auswahl (getroffen von einer Person oder einer Gruppe), aus möglichen Alternativen unter Beachtung von Erfolgskriterien. Diese Auswahl kann sowohl bewusst als auch unbewusst getroffen werden.*

Definition: *Eine **Testentscheidung** ist eine Entscheidung, die im Rahmen eines Testprozesses getroffen wird.*

Testentscheidungen beeinflussen den Ablauf des Testprozesses und die darin auszuführenden Aktivitäten und zu erstellenden Artefakte. Eine bewusst getroffene Entscheidung wird explizit und eine unbewusst getroffene Entscheidung implizit in den Artefakten des Testprozesses dokumentiert.
Die Zusammenhänge zwischen Testentscheidungen, Testaktivitäten, Testrollen und Testartefakten sind in einem Meta-Modell in Abbildung 5 dargestellt. Eine Testentscheidung wird im Rahmen des Testprozesses in genau einer Testaktivität (z.B. Analyse und Design) getroffen. Eine Testaktivität benötigt verschieden viele Testartefakte als Input, um neue Testartefakte zu erstellen sowie um die in ihr getroffenen Entscheidungen zu ermöglichen und zu begründen. Eine Testaktivität wird von einer oder mehreren Rollen ausgeführt, die während der Ausführung mehrere Testentscheidungen treffen. Hierbei wird jede Testentscheidung von mindestens einer ausführenden Testrolle getroffen. Eine solche Testentscheidung ist entweder implizit oder explizit in einem Testartefakt dokumentiert. In einem Testartefakt können mehrere Entscheidungen dokumentiert sein. In Testaktivitäten erstellte Testartefakte fließen als Input in nachfolgende Testaktivitäten ein oder sind für das Treffen von Testentscheidungen als Eingabe notwendig. Welche Ausgabe eine Aktivität erstellt, hängt von den früher erstellen Testartefakten und den früher getroffenen Testentscheidungen ab.

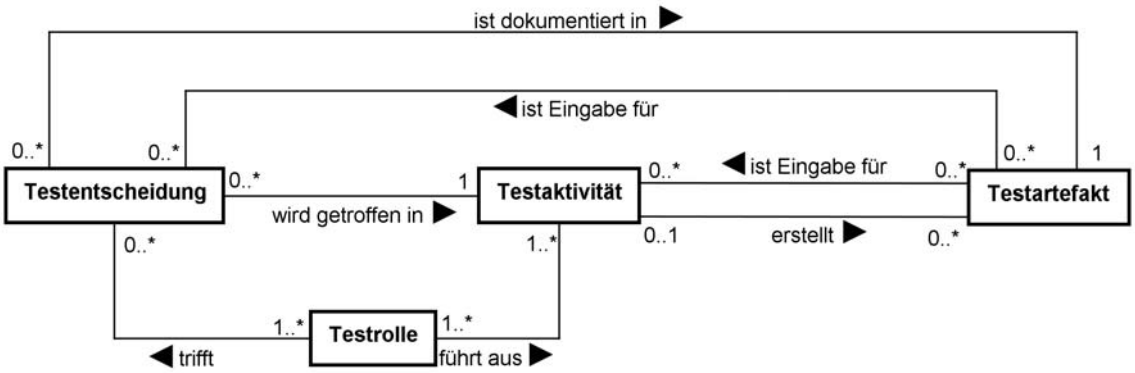


Abbildung 5: Zusammenhänge zwischen Rollen, Artefakten, Aktivitäten, Entscheidungen

Im Rahmen des Testprozesses kann zwischen Prozessbezogenen und Testobjektbezogenen Testentscheidungen unterschieden werden:

Definition: ***Prozessbezogene Entscheidungen*** *sind Testentscheidungen, die den Ablauf des Testprozesses beeinflussen, d.h. durch diese Entscheidungen wird festgelegt, welche Testaktivitäten wann und wie durchzuführen sind und welche Testartefakte innerhalb dieser Testaktivitäten erstellt werden. Diese Entscheidungen werden im Rahmen von Planungsaktivitäten erstellt und im Testplan explizit dokumentiert.*

Definition: ***Testobjektbezogene Entscheidungen*** *sind Testentscheidungen, die sich auf das zu testende Softwaresystem selbst beziehen. Sie beeinflussen, wie das System getestet wird und sind in den Artefakten implizit dokumentiert.*

Die Prozess- und Testobjektbezogenen Testentscheidungen können in einer Entscheidungshierarchie angeordnet werden. Eine Entscheidungshierarchie, wie sie in Abbildung 6 zu sehen ist, gibt eine zeitliche Reihenfolge vor, in der die Entscheidungen getroffen werden sollten. Entscheidungen auf der oberen Ebene sollten vor Entscheidungen auf den unteren Ebenen getroffen werden, da die „nachfolgenden“ Entscheidungen durch sie

beeinflusst werden. Entscheidungen, die auf der gleichen Ebene angeordnet sind, können parallel und bis zu einem bestimmten Grad unabhängig voneinander getroffen werden. Diese Erkenntnisse lassen sich in folgenden Regeln zusammenfassen:

Regel der Entscheidungsabhängigkeit: *Entscheidungen der unteren Ebenen in der Hierarchie sind abhängig von den Entscheidungen der oberen Ebenen, d.h. frühe Entscheidungen wirken sich direkt auf nachfolgende Entscheidungen aus. Dies führt dazu, dass fehlende explizite Entscheidungen auf höheren Ebenen, in Entscheidungen auf unteren Ebenen implizit enthalten sind.*

Regel der Parallelität: *Alle Entscheidungen innerhalb einer Ebene können parallel getroffen werden, da sie bis zu einen gewissen Grad unabhängig voneinander sind. Hierbei kann es jedoch zu Wechselwirkungen zwischen den Entscheidungen kommen.*

Die Entscheidungshierarchie in Abbildung 6 besteht aus sieben unterschiedlichen Entscheidungsebenen: *Spezifikations-*, *Testziel-*, *Teststrategie-*, *Testdesign-*, *Testrealisierungs-*, *Testlauf-* und *Testauswertungsebene*. Jeder Ebene sind Testentscheidungen zugeordnet. Die Art der Entscheidung ist durch die farbliche Markierung hervorgehoben. Prozessbezogene Entscheidungen sind durch einen gelben und Testobjektbezogene Entscheidungen durch einen grauen Hintergrund gekennzeichnet.

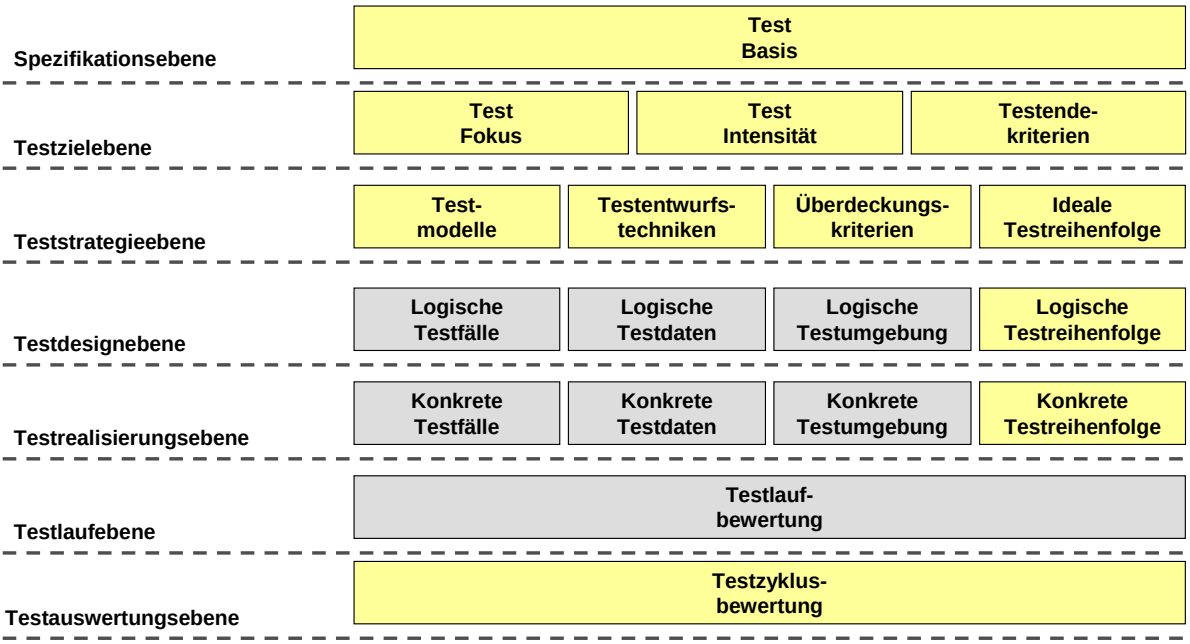


Abbildung 6: Entscheidungshierarchie des generischen Testprozesses

Es ist zu erkennen, dass die Entscheidungen der oberen drei Ebenen (Spezifikations-, Testziel- und Teststrategieebene) sowie die Entscheidung der untersten Ebene (Testauswertungsebene) Prozessbezogene Entscheidungen sind. Diese Entscheidungen beeinflussen den Verlauf des Testprozesses und geben vor, welche Testaktivitäten wie auszuführen und welche Testartefakte zu erstellen sind. Auf der Testdesign- und Testrealisierungsebene hingegen sind überwiegend Testobjektbezogene Entscheidungen (einzige Ausnahme sind die Entscheidungen zur logischen und konkreten Testreihenfolge) zu finden. Sie betreffen das zu testende Testobjekt und geben vor, wie es zu testen ist.
In den Regeln der *Parallelität* und der *Entscheidungsabhängigkeit* wird definiert, dass zwischen Entscheidungen Abhängigkeiten auftreten können. Diese Abhängigkeiten können

zum einen unidirektional sein. Als Beispiel sei hier die Abhängigkeit zwischen der Entscheidung Testbasis und Testentwurfstechnik genannt. Die Entscheidung zur Testbasis muss vor der *Entscheidung* zur *Testentwurfstechnik* getroffen werden. Sollte dies nicht der Fall sein, so wird die Entscheidung zur Testbasis implizit in der Entscheidung zur Testentwurfstechnik getroffen.
Zum anderen gibt es zwischen zwei Entscheidungen auch bidirektionale Abhängigkeiten. Eine bidirektionale Abhängigkeit kann nur innerhalb einer Ebene auftreten. Dies führt dazu, dass eine Entscheidung innerhalb dieser Abhängigkeit eine andere Entscheidung überflüssig macht oder einschränkt. Wie sich die einzelnen Entscheidungen innerhalb solcher Abhängigkeiten beeinflussen, hängt von der Reihenfolge ab, in der die Entscheidungen getroffen werden. Häufig sollten Entscheidungen innerhalb einer bidirektionalen Abhängigkeit gleichzeitig und im Rahmen derselben Testaktivitäten gefällt werden. Die Abhängigkeiten zwischen den Entscheidungen des allgemeinen Testprozesses sind in Abbildung 7 (Seite 35) veranschaulicht. Unidirektionale Abhängigkeiten sind durch einen einfachen Pfeil zwischen der unabhängigen Entscheidung (ausgehende Pfeilrichtung) und der abhängigen Entscheidung (eingehe Pfeilrichtung) gekennzeichnet. Bidirektionale Abhängigkeiten sind an den „Doppelpfeilen“ zu erkennen und tragen darüber hinaus eine Identifikationsnummer. Die Identifikationsnummer stellt in Kurzform die sich gegenseitig beeinflussenden Eigenschaften dar, z.B. I-EK bezeichnet die bidirektionale Abhängigkeit der Entscheidungen zur Test**I**ntensität und der Entscheidung zu den Test**E**nde**K**riterien.
Die einzelnen Testentscheidungsebenen, ihre enthaltenen Testentscheidungen und die Abhängigkeiten zwischen den Entscheidungen werden nachfolgend näher beschrieben.

3.1.3.1 Spezifikationsebene

Auf der Spezifikationsebene werden die vorliegenden Informationen in den Artefakten der Entwicklungsphasen des Softwareprojekts überprüft. Es muss entschieden werden, ob auf Basis der gegebenen Informationen der Testprozess begonnen werden kann. Die Artefakte, die als Basis für die Planung und Durchführung des Testprozesses benötigt und verwendet werden, werden als Testbasis bezeichnet. Die Testbasis besteht aus den Informationen, Spezifikationen, Modellen (z.B. Anforderungen, Architektur und Systemdesign, Quelltexten) und Entscheidungen des Entwicklungsprozesses und dient als Grundlage für die Auswahl des Testfokus, der Testintensität und der Testendekriterien der nachfolgenden Testzielebene. Die Entscheidung über Vollständigkeit der **Testbasis** ist Voraussetzung für den Start des Testprozesses und wird vom Testmanager mit Unterstützung des Testdesigners getroffen. Ohne ausreichende Informationen kann der Testprozess nicht beginnen. Im Fall, dass Informationen in der Testbasis fehlen, sollten diese erst ergänzt werden, bevor weitere Entscheidungen im Testprozess getroffen werden können. Die Entscheidung über die Testbasis ist von keinen weiteren Testentscheidungen abhängig.

3.1.3.2 Testzielebene

Vollständiges Testen von Programmen ist nicht möglich [Me79]. Daher müssen die zu testenden Teile der Software ausgewählt und es muss entscheiden werden, wie intensiv diese zu testen sind. Diese Entscheidungen werden auf der Testzielebene getroffen. Die Testbasis dient hierbei als Grundlage für den Testdesigner und -manager zu entscheiden, wie der **Testfokus** gelegt ist. Mit der Festlegung des Testfokus kann die **Testintensität** festgelegt werden, d.h. mit welcher Intensität die Testobjekte des Testfokus getestet werden. In die Entscheidung über die Testintensität fließen Informationen über verfügbare Testressourcen ein. Beide Entscheidungen (Testfokus, Testintensität) beeinflussen einander auf dieser Ebene. Das Festlegen einer bestimmten Testintensität vor dem Festlegen des Testfokus kann die Entscheidungen des Testfokus einschränken oder gar überflüssig machen (z.B. die Entscheidung: *alle Systemteile müssen gleich intensiv getestet werden*,

macht die Auswahl des Testfokus überflüssig, da sie bereits implizit im „alle“ getroffen wurde). Umgekehrt kann eine bestimmte Entscheidung zum Testfokus die Entscheidungen über die Testintensität einschränken. Dies ist der Fall, wenn die Entscheidung zum Testfokus festlegt, dass genau ein Testobjekt zu testen ist. Somit würden die vorhandenen Testressourcen diesem Testobjekt zugewiesen und es ist keine gesonderte Betrachtung der Testintensität notwendig. Die bidirektionale Abhängigkeit zwischen diesen Entscheidungen ist in Abbildung 7 (Seite 35) mit der Bezeichnung „*F-I*“ gekennzeichnet.
Auf dieser Ebene entscheidet der Testmanager ebenfalls, welche **Testendekriterien** am Ende des Testprozesses erreicht sein müssen, um den Testprozess erfolgreich abschließen zu können. Diese Entscheidung wird durch die Entscheidung über die Testintensität beeinflusst und umgekehrt („*I-EK*“).

3.1.3.3 Teststrategieebene

Auf der Teststrategieebene werden vom Testmanager und Testdesigner vier wichtige Entscheidungen getroffen. Aufbauend auf den Entscheidungen zur Testbasis, zum Testfokus und zur Testintensität legt der Testdesigner fest, welche **Testentwurfstechniken** im Testprozess verwendet werden, um die Testfälle und Testdaten aus der Testbasis abzuleiten. Diese Entscheidung beeinflusst und wird beeinflusst durch die Entscheidungen zu den **Testmodellen** („*M-ET*“) und zu den **Überdeckungskriterien** („*ET-ÜK*“). Mit der Entscheidung zu den Testmodellen legt der Testdesigner fest, welche Informationen (Modelle, Spezifikationen) zusätzlich zu den in der Testbasis bereits enthaltenen Informationen benötigt werden und in welcher Form diese vorliegen müssen. Diese zusätzlichen Informationen (=Testmodelle) werden benötigt, um mit Hilfe der Testentwurfstechnik Testfälle abzuleiten. Hierbei spielt die Reihenfolge des Treffens der Entscheidungen über die Testmodelle und über die Testentwurfstechnik eine entscheidende Rolle. Sollten erst die Entscheidungen über die Testmodelle getroffen werden, so schränken diese Entscheidungen die Auswahl der zur Verfügung stehenden Testentwurfstechniken ein. Im umgekehrten Fall kann das Festlegen von Testentwurfstechniken die Entscheidung über die zu erstellenden Testmodelle stark einschränken. Die Entscheidung zu den Überdeckungskriterien wird vom Testmanager festgelegt und ist abhängig von den Entscheidungen über den Testfokus und die Testintensität. Sie legt Vorgaben fest, wie viele Testfälle mit Hilfe der Testentwurfstechniken erstellt werden müssen. Diese Entscheidung wird stark durch die Entscheidung der Testentwurfstechniken beeinflusst und umgekehrt.
Eine weitere Testentscheidung, die der Testdesigner auf dieser Ebene treffen muss, ist die Entscheidung zur **idealen Testreihenfolge**. Diese, von den Entscheidungen zum Testfokus und zur Testintensität abhängige Entscheidung legt fest, welche Teile des Systems in welcher Reihenfolge zu testen sind. Hierbei kann eine konkrete Reihenfolge erstellt werden, was wann zu testen ist, oder aber es werden nur Regeln (z.B. „*Kritische Teile müssen vor unkritischen Teilen getestet werden*“) definiert, die das Bestimmen einer Reihenfolge zu einem späteren Zeitpunkt ermöglichen.

3.1.3.4 Testdesignebene

Die Testdesignebene enthält Entscheidungen, mit denen der Testdesigner festlegt, wie das System bzw. Teile des Systems im Einzelnen zu testen sind. Auf dieser Ebene werden Entscheidungen zu logischen Testfällen und Testdaten, zur logischen Testumgebung und zur logischen Testreihenfolge getroffen. Die Entscheidungen zu den **logischen Testfällen** legen für jeden einzelnen Testfall fest, wie dieser abläuft, d.h. welche Test- und Prüfschritte darin ausgeführt werden. Diese Testfälle definieren jedoch noch keine konkreten Handlungen, wie z.B. „*der Benutzer gibt in das Feld beschriftet mit dem Wert ‚Alter’ das aktuelle Alter ein und betätigt anschließend den Button mit der Aufschrift ‚Senden’*“, sondern stellen die Handlungen abstrahiert dar, beispielsweise „*Eingabe der Persönlichen Daten*“.

Ebenso abstrahiert beschrieben werden auch die Prüfanweisungen, die in diesem Testfall ausgeführt werden, um zu überprüfen, ob eine Abweichung zwischen IST- und SOLL-Verhalten aufgetreten ist. Eng verknüpft mit den Entscheidungen zu den Testfällen sind die Entscheidungen zu den **logischen Testdaten**. Hierbei wird festgelegt, welche Testdaten zu verwenden und zu prüfen sind. Beide Entscheidungen beeinflussen sich sehr stark („*LTF-LTD*“), da zum einen die einzugebenden Daten in den Testschritten und die zu prüfenden Daten in den Prüfschritten durch die logischen Testdaten definiert werden, aber im Gegenzug die Test- und Prüfschritte bestimmte Daten verlangen. Die Entscheidungen zu den logischen Testdaten und Testfällen sind abhängig von den Entscheidungen über die Testmodelle, die Testentwurfstechniken und die Überdeckungskriterien.
Die Entscheidung über die **logische Testumgebung** ist eine weitere Entscheidung, die auf dieser Ebene getroffen werden muss. Sie legt fest, welche Anforderungen die Testumgebungen erfüllen müssen, damit später die Testfälle darin ausgeführt werden können. Die **logische Testreihenfolge** legt fest, welche Testfälle in welcher Reihenfolge ausgeführt werden bzw. welche Teile der Software in welcher Reihenfolge zu testen sind. Somit hängt diese Entscheidung von der bereits getroffenen Entscheidung über die ideale Testreihenfolge ab. Jedoch wird diese Entscheidung durch die Entscheidung zur logischen Testumgebung beeinflusst und umgekehrt („*LT-LR*“).

3.1.3.5 Testrealisierungsebene

Die Testrealisierungsebene beschäftigt sich mit den Fragestellungen, wie die konkreten Testfälle und Testdaten aussehen und wie diese umgesetzt werden können. Der Tester setzt hierbei die Vorgaben aus den logischen Testfällen in **konkrete Testfälle** um. Er entscheidet, wie die abstrakt beschriebenen Test- und Prüfschritte durch konkrete Angaben spezialisiert werden können. Ebenso wie für die Testfälle müssen die abstrakt beschriebenen logischen Testdaten durch **konkrete Testdaten** ersetzt werden. Hierbei trifft der Tester Entscheidungen, welche konkreten Werte in den einzelnen Testfällen benötigt werden. Aus den Entscheidungen zu den logischen Testumgebungen muss der Tester sich für **konkrete Testumgebungen** entscheiden. Er legt beispielsweise fest, welche konkreten Werkzeuge eingesetzt werden, welche Monitore zum Aufzeichnen von Ergebnissen verwendet werden oder wie die Protokolle zu erstellen sind. Wann welcher Testfall ausgeführt wird, wird vom Testmanager im Rahmen der **konkreten Testreihenfolge** festgelegt. Hierbei erhält er Unterstützung vom Testdesigner und vom Tester, um möglichst viele Synergieeffekte beim Aufbau der Testumgebung und bei der Realisierung/Ausführung der Testfälle zu haben. Hierbei bauen die Entscheidungen zur konkreten Testreihenfolge auf den Entscheidungen über die logische Testreihenfolge auf.

3.1.3.6 Testlaufebene

Auf der Testlaufebene muss für jeden Testfall entschieden werden, ob er erfolgreich war, d.h. ob er einen Fehler gefunden hat. Hierzu wird vom Tester überprüft, ob es eine Abweichung zum spezifizierten Soll-Verhalten im konkreten Testfall gab. Im Fall einer Abweichung wird vom Testdesigner entschieden, ob es sich tatsächlich um einen Fehler im zu testenden System handelt oder um Fehler in den Testartefakten. Weiterhin sollte im Rahmen dieser Entscheidung für jeden identifizierten Fehler vom Testmanager mit Unterstützung des Testdesigners eine Klassifikation vorgenommen werden, z.B. sollte jedem Fehler eine Schwere und Priorität zugeordnet werden, um später die Korrekturmaßnahmen besser planen zu können. Um die Entscheidung zu dieser **Testlaufbewertung** durchführen zu können, benötigen die Tester und der Testdesigner die Spezifikation der konkreten Testfälle und Testdaten sowie die Ergebnisse der Testläufe.

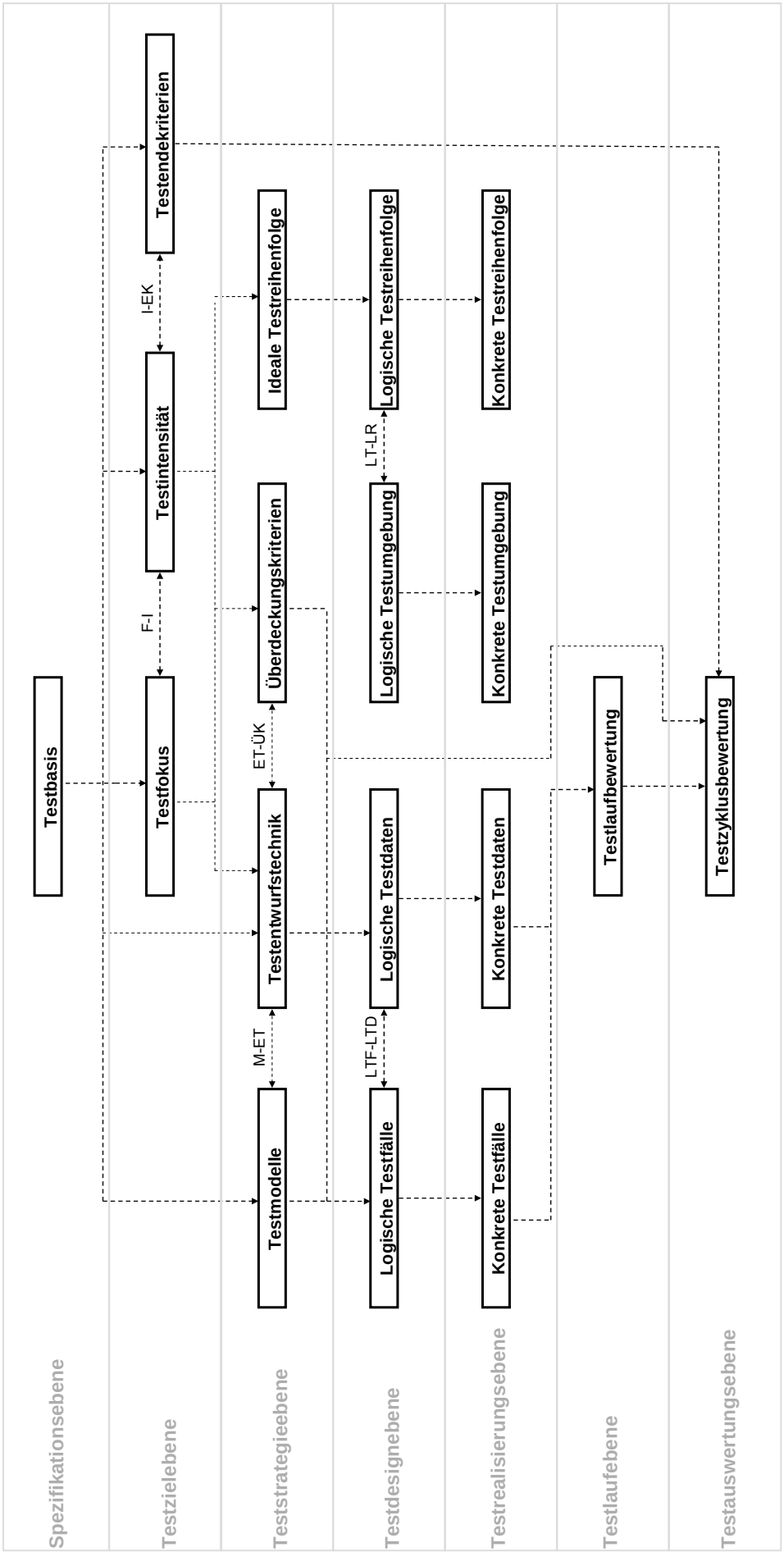


Abbildung 7: Abhängigkeiten zwischen den Entscheidungen des allgemeinen Testprozesses

3.1.3.7 Testauswertungsebene

Auf der untersten Ebene der Entscheidungshierarchie entscheidet der Testmanager über den Erfolg bzw. Misserfolg des aktuellen Testzyklusses bzw. Testprozesses. Diese **Testzyklusbewertung** umfasst das Analysieren der aktuellen Ergebnisse und den Vergleich mit den vorgegebenen Überdeckungs- und Testendekriterien. Er entscheidet, ob diese Kriterien erreicht worden sind und die Testaktivitäten abgeschlossen werden können oder ob diese Kriterien nicht erreicht worden sind und der Testprozess an ausgewählten Aktivitäten erneut ansetzen muss (z.B. Spezifikation der logischen Testfälle) und Testentscheidungen zu überdenken oder neu zu treffen sind.

Die Zuordnung der vorgestellten Testentscheidungen zu den in [SL05] beschriebenen (und in 3.1.2 vorgestellten) Testaktivitäten ist in Abbildung 8 dargestellt. Aktivitäten sind durch graue Ellipsen und Entscheidungen durch weiße Rechtecke gekennzeichnet. Durchgezogene Pfeile von einer Aktivität zu einer Entscheidung symbolisieren, dass eine Entscheidung in der entsprechenden Aktivität getroffen wird. Ein gestrichelter Pfeil von einer Entscheidung zu einer Aktivität gibt an, dass eine Entscheidung für die Ausführung der Aktivität benötigt wird.
In der Prozessvorbereitung („Beginn“) wird die Entscheidung getroffen, ob der Testprozess durchgeführt werden kann, d.h. ob die Testbasis ausreichend detailliert ist, um die nachfolgenden Testentscheidungen zu treffen und die Testaktivitäten auszuführen. Diese Entscheidung fließt direkt in die Aktivität „Planung und Steuerung“ ein. In der initialen Planungsphase werden die Entscheidungen über den Testfokus, die Testintensität, die Testendekriterien und die ideale Testreihenfolge gefällt. Im weiteren Verlauf des Projekts wird sowohl die logische als auch die konkrete Testreihenfolge festgelegt. Die Entscheidungen zum Testfokus, zur Testintensität, über die Testendekriterien und die ideale Testreihenfolge fließen in die Testaktivität „Analyse und Design“ ein. In ihr werden die Entscheidungen zu den Testmodellen, Testentwurfstechniken und Überdeckungskriterien genutzt. Diese Entscheidungen werden im Rahmen der gleichen Aktivität weiterverwendet, um die logischen Testdaten und Testfälle zu erstellen und die damit verbundenen Entscheidungen zu treffen. In der Aktivität „Realisierung und Durchführung“ werden diese Entscheidungen benötigt, um zusammen mit den Entscheidungen zur logischen und konkreten Testreihenfolge die Entscheidungen über die konkreten Testfälle, Testdaten und Testumgebungen zu treffen. Nach der Durchführung der Testfälle müssen die Testprotokolle ausgewertet und die Entscheidungen zur Testlaufbewertung getroffen werden. Diese Entscheidung wird anschließend für die Testzyklusbewertung genutzt.

Die ergänzende Betrachtung der Entscheidungen neben den Rollen und Aktivitäten bringt den Vorteil, dass sich die ausführenden Personen mit den Entscheidungen bewusster auseinander setzen. Aus Abbildung 8 kann entnommen werden, welche Entscheidungen in welchen Aktivitäten getroffen werden müssen und auf welche weiteren Aktivitäten diese Entscheidungen Einfluss haben. Durch die Veranschaulichung dieser Zusammenhänge sollen die beteiligten Personen motiviert werden, die Entscheidungen explizit zu treffen, da die ihnen bewusst gemacht wird, wie weitreichend ihre Entscheidungen sind. Dies führt im besten Fall dazu, dass sie sich mit den verschiedenen Lösungsalternativen auseinander setzen und anschließend gezielt die beste Lösung auswählen können, d.h. die beste Entscheidung treffen.

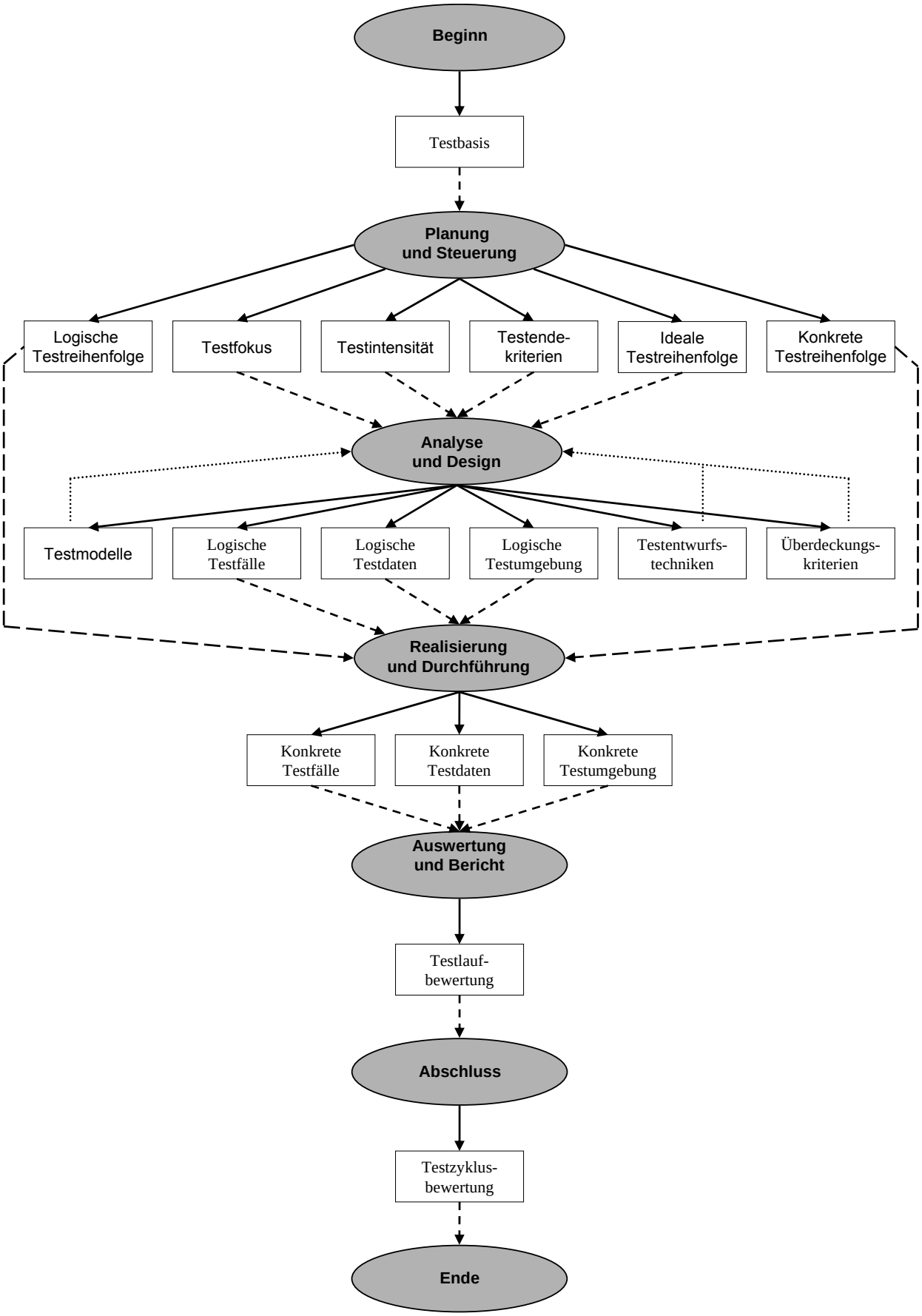


Abbildung 8: Zuordnung der Testentscheidungen zu den Testaktivitäten

3.1.4. Anwendungsmöglichkeiten der Entscheidungsebenen

Eine kurze Zusammenfassung der Entscheidungsebenen des generischen Testprozesses sowie eine Liste der Anwendungsmöglichkeiten sind in [BIP07a] und [BIP07b] zu finden. Diese Anwendungsmöglichkeiten werden nachfolgend kurz zusammengefasst, um die Mächtigkeit der vorangegangenen Entscheidungshierarchie für den generischen Testprozess zu demonstrieren. Die Anwendungsmöglichkeiten stehen jedoch nicht im Fokus dieser Arbeit.

In erster Linie dient der vorgestellte generische Testprozess als Vorlage für die Planung und Durchführung von Testprozessen verschiedener Teststufen. Er liefert wichtige Aktivitäten und Entscheidungen, die im Rahmen eines Testprozesses durchgeführt bzw. getroffen werden müssen, und gibt einen Überblick über die beteiligten Rollen. Die einzelnen Entscheidungen der Entscheidungshierarchie können in einer Checkliste vorgegeben werden, um sich die zu treffenden Entscheidungen im Testprozess vor Augen zu führen. Im Laufe des Prozesses können so die einzelnen Entscheidungen bewusst getroffen, dokumentiert und auf der Liste abgehakt werden. Es ist jedoch zu beachten, dass der generische Testprozess gezielt allgemein gehalten wurde, um alle Entscheidungen, die im Testprozess der unterschiedlichen Teststufen gemeinsam enthalten sind, zusammenzufassen. Für die Anwendung auf einer bestimmten Teststufe müssen die teststufenspezifischen Entscheidungen ergänzt werden. Im Rahmen dieser Arbeit wurden die spezifischen Entscheidungen des Integrationstestprozesses identifiziert und in einer integrationstestspezifischen Entscheidungshierarchie strukturiert. Diese Hierarchie wird im Detail in Kapitel 3.2 vorgestellt. Im Rahmen einer weiteren Studie wurden zusätzlich die spezifischen Entscheidungen des Systemtest untersucht und den Entscheidungen des Integrationstests gegenübergestellt. Das Ergebnis dieser Gegenüberstellung ist in [BIP07b] nachzulesen.

Die Vorarbeiten der in [IPR+06] beschriebenen Studie zur Evaluation von Testmanagementwerkzeugen wurden durch die Entscheidungshierarchie unterstützt. Die Hierarchie unterstützte das Erstellen der Fragen für die Umfrage.

Als abschließende Anwendungsmöglichkeit des generischen Testprozesses soll die Evaluation von Literatur über testrelvante Themen genannt werden. Entscheidungen, Rollen und Aktivitäten können verwendet werden, um die in der Literatur adressierten Testthemen zu kategorisieren. Wie in [BIP07b] dargestellt, können beispielsweise Literaturquellen, die sich mit Testentwurfstechniken auseinander setzen, danach klassifiziert werden, welche Entscheidungen sie unterstützen bzw. für welche Entscheidungen sie Informationen liefern.

3.2. Integrationstestprozess

Nachfolgend wird der in Kapitel 3.1 vorgestellte generische Testprozess instanziiert und um die integrationstestprozessspezifischen Eigenheiten ergänzt. Hierzu werden in einem ersten Schritt die beteiligten Rollen und ihre Kommunikationspartner näher betrachtet (3.2.1). Die zu treffenden Entscheidungen werden in 3.2.2 beschrieben. Diese werden mit Hilfe eines Integrationstestprozesses für das in Kapitel 2.3 vorgestellte Beispiel der Onlineumfrage exemplarisch veranschaulicht. Diese Beispiele werden mit „Beispiel-Onlineumfrage“ eingeleitet und sind an einer kleineren Schriftart zu erkennen.

3.2.1. Testrollen

Der Integrationstestprozess stellt ebenso wie der System- oder Unittest eine große Herausforderung für die beteiligten Rollen des Testprozesses dar. Die verschiedenen ausführenden Rollen benötigen eine Vielzahl von Informationen, in Form von Spezifikationen und Modellen als Eingabe für die auszuführenden Aktivitäten und die zu treffenden

Entscheidungen. In Abbildung 9 sind die ausführenden Testrollen Rollen im grauen Kasten mit abgerundeten Ecken dargestellt. Die unterstützenden Rollen sind Rollen der Projektleitung, dargestellt mit einem weißen Kopf, Rollen der Entwicklung, gekennzeichnet mit einem dunkelgrauen Kopf und Rollen aus anderen Testprozessen, dargestellt mit einem hellgrauem Kopf. Die Abbildung verdeutlicht, welche unterstützenden Rollen den ausführenden Rollen Informationen bereitstellen, um die Ausführung von Testaktivitäten und das Treffen von Testentscheidungen zu ermöglichen. Die Art der bereitgestellten Informationen ist an den gestrichelten Pfeilen zu erkennen. So stellt beispielsweise der Projektmanager dem Testmanager die Informationen aus den Projektplänen zur Verfügung.
Die ausführenden Testrollen werden nachfolgend vorgestellt. Es werden die Verantwortungsbereiche und die Kommunikationspartner sowie die benötigten Informationen für das erfolgreiche Treffen der Entscheidungen erläutert.

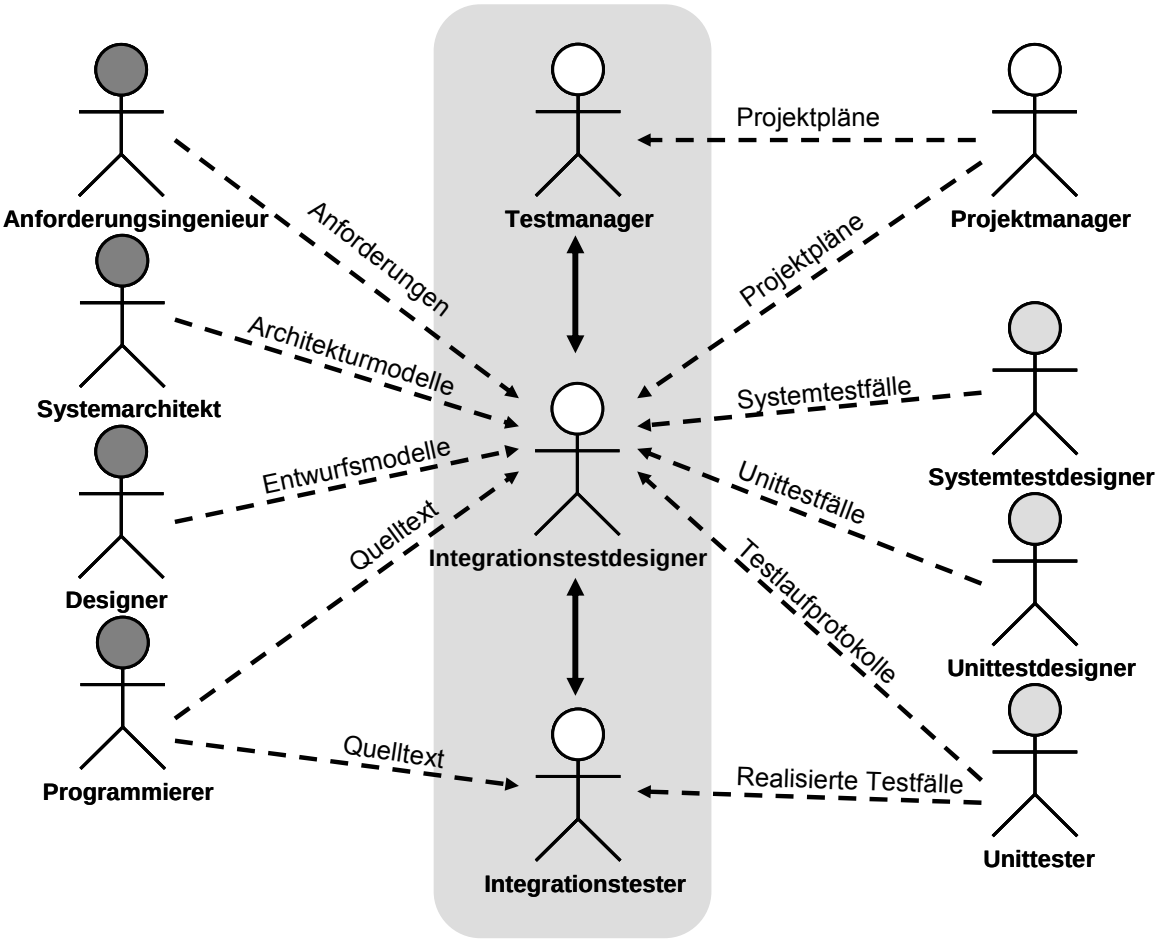


Abbildung 9: Rollen des Integrationstestprozesses und ihre Abhängigkeiten

3.2.1.1 Testmanager

Zusätzlich zu den Verantwortungsbereichen, die ein Testmanager des generischen Testprozesses innehat, ist er dafür verantwortlich, festzulegen, welche Bausteine und Abhängigkeiten im Integrationstest wie intensiv zu testen sind. Die Kommunikationspartner des Testmanagers sind, wie in Abbildung 9 erkennbar, der Integrationstestdesigner und der Projektmanager des gesamten Entwicklungsprojekts. Der Projektmanager liefert Informationen über die Planung der Entwicklungsphasen und somit Aussagen darüber, wann welche Bausteine für den Testprozess zur Verfügung stehen. Der Projektmanager verhandelt mit Testmanager über finanzielle, zeitliche und personelle Ressourcen, eingesetzt werden sollen. Der Integrationstestdesigner liefert wichtige Informationen über den Erfolg

durchgeführter Testaktivitäten und unterstützt den Testmanager beim Treffen der Entscheidungen mit Informationen.

3.2.1.2 Integrationstestdesigner

Der Integrationstestdesigner identifiziert zu Beginn des Testprozesses alle Abhängigkeiten zwischen den Bausteinen der Software, ermittelt die Reihenfolge, in der das System schrittweise zusammengesetzt und getestet wird und spezifiziert die Testfälle sowie die Testumgebung für die Tests. In erster Linie kommuniziert er mit dem Integrationstestmanager sowie Integrationstester. Für das Treffen von Entscheidungen, die in seinen Verantwortungsbereich fallen, benötigt der Integrationstestdesigner viele Informationen und Artefakte aus der Entwicklung. Der Anforderungsingenieur liefert z.B. die Anforderungen, die an das Gesamtsystem gestellt werden. Die Anforderungsspezifikation liefert Informationen, um das erwartete Verhalten der Bausteine vorhersagen zu können. Der Systemarchitekt, der Designer und der Programmierer liefern Informationen über die interne Struktur der zu integrierenden Systemteile. Mit zunehmender Nähe zum Quelltext werden Informationen detaillierter und verraten mehr über die tatsächliche Struktur und die tatsächlichen Abhängigkeiten zwischen den Bausteinen des späteren Softwaresystems. Aber nicht nur Rollen aus der Entwicklung liefern wichtige Informationen. Testrollen aus Testprozessen anderer Teststufen (z.B. Unit- und Systemtest) können Testartefakte dieser Teststufen[1] bereitstellen. Bereits spezifizierte (aber noch nicht ausgeführte Testfälle) des Systemtestdesigners (vgl. V-Modell in [Bo79]) können verwendet werden, um den Aufwand der Testspezifikation im Integrationstest zu verringern. Hierzu werden die spezifizierten Systemtestfälle für den Integrationstest angepasst und wieder verwendet. Die Artefakte aus dem Unittest, der zeitlich vor dem Integrationstest angesiedelt ist (vgl. V-Modell in [Bo79]), können im Integrationstest wieder verwendet werden. Bereits spezifizierte und realisierte Testfälle können für den Integrationstest adaptiert werden, um Ressourcen zu sparen.

3.2.1.3 Integrationstester

Der Integrationstester ist für die Realisierung der Testumgebung und der Testfälle verantwortlich. Er erstellt alle notwendige Stubs, Treiber, Monitore und instrumentiert die zu testenden Bausteine. Er führt die Testläufe aus und dokumentiert die Ergebnisse jedes einzelnen Testlaufs in den vorgeschriebenen Testprotokollen. Für das Treffen der Entscheidungen innerhalb seines Verantwortungsbereichs kommuniziert der Integrationstester mit dem Programmierer, dem Integrationstestdesigner und dem Unittester. Der Integrationstestdesigner liefert ihm die Spezifikation der logischen Testfälle, Testdaten und eine Beschreibung der notwendigen Testumgebungen für die einzelnen Testläufe. Für die Realisierung der Testfälle und Testumgebungen verwendet der Integrationstester Informationen vom Programmierer über die interne Struktur der Testobjekte und kann bereits realisierte Testumgebungen und Testfälle des Unittesters wieder verwenden, z.B. Stubs, Treiber und Monitore, die für den Unittest erstellt worden sind.

3.2.2. Entscheidungen im Integrationstestprozess

Die in Kapitel 3.1 vorgestellte Entscheidungshierarchie für den generischen Testprozess kann bei näherer Betrachtung des Integrationstestprozesses um integrationstestspezifische Entscheidungen verfeinert und ergänzt werden. Die Entscheidungsebenen des allgemeinen Testprozesses sind ebenso im Integrationstestprozess präsent. Die Entscheidungshierarchie des Integrationstestprozesses ist in Abbildung 10 dargestellt.

---

[1] Das Zusammenspiel der einzelnen Testprozesse verschiedener Teststufen ist nicht Gegenstand dieser Arbeit. Es soll jedoch angemerkt werden, dass dieses Forschungsgebiet ein großes Potential zur Ressourcenersparnis in Softwareentwicklungsprojekten beinhaltet.

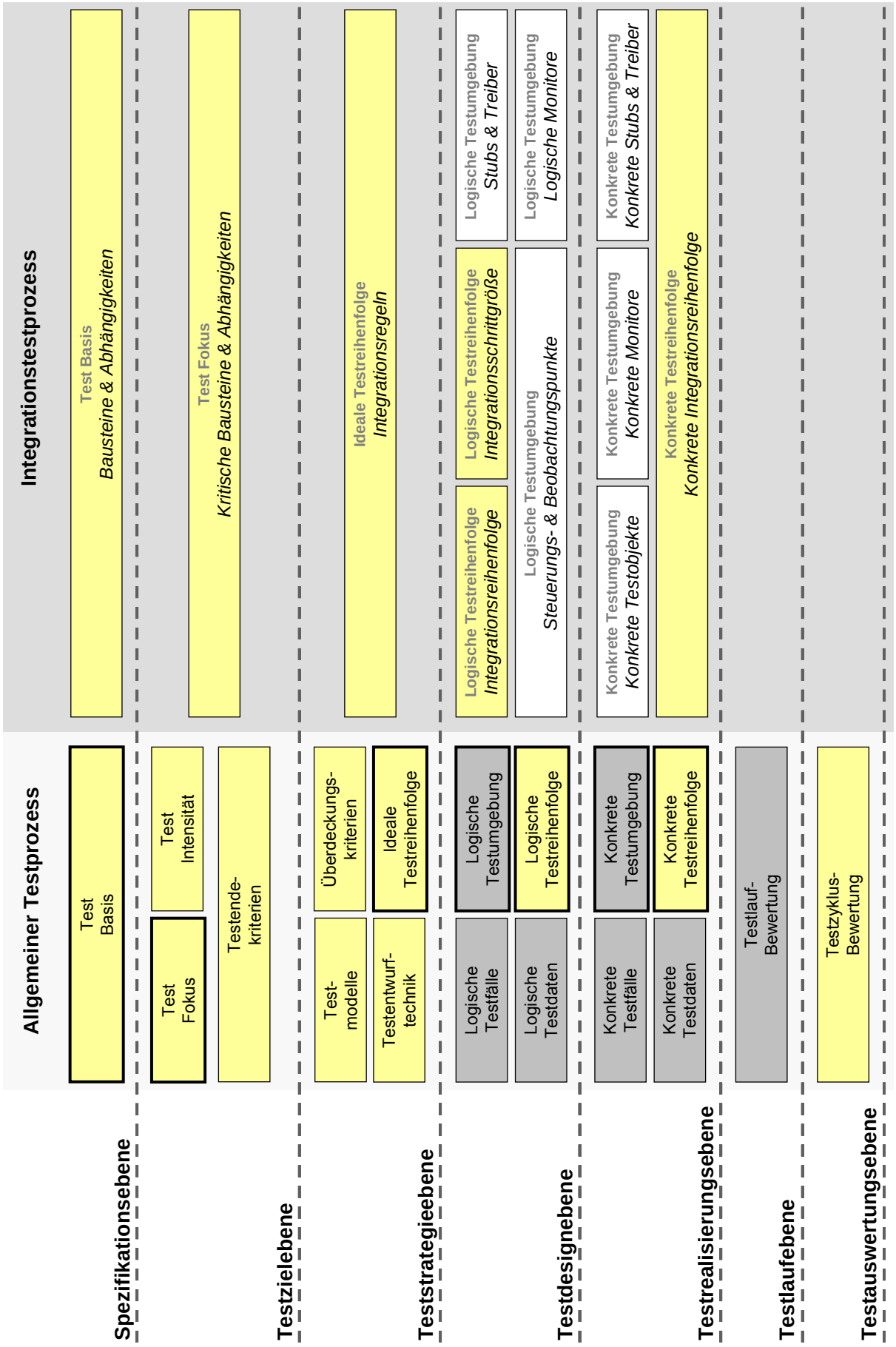


Abbildung 10: Entscheidungshierarchie des Integrationstestprozesses

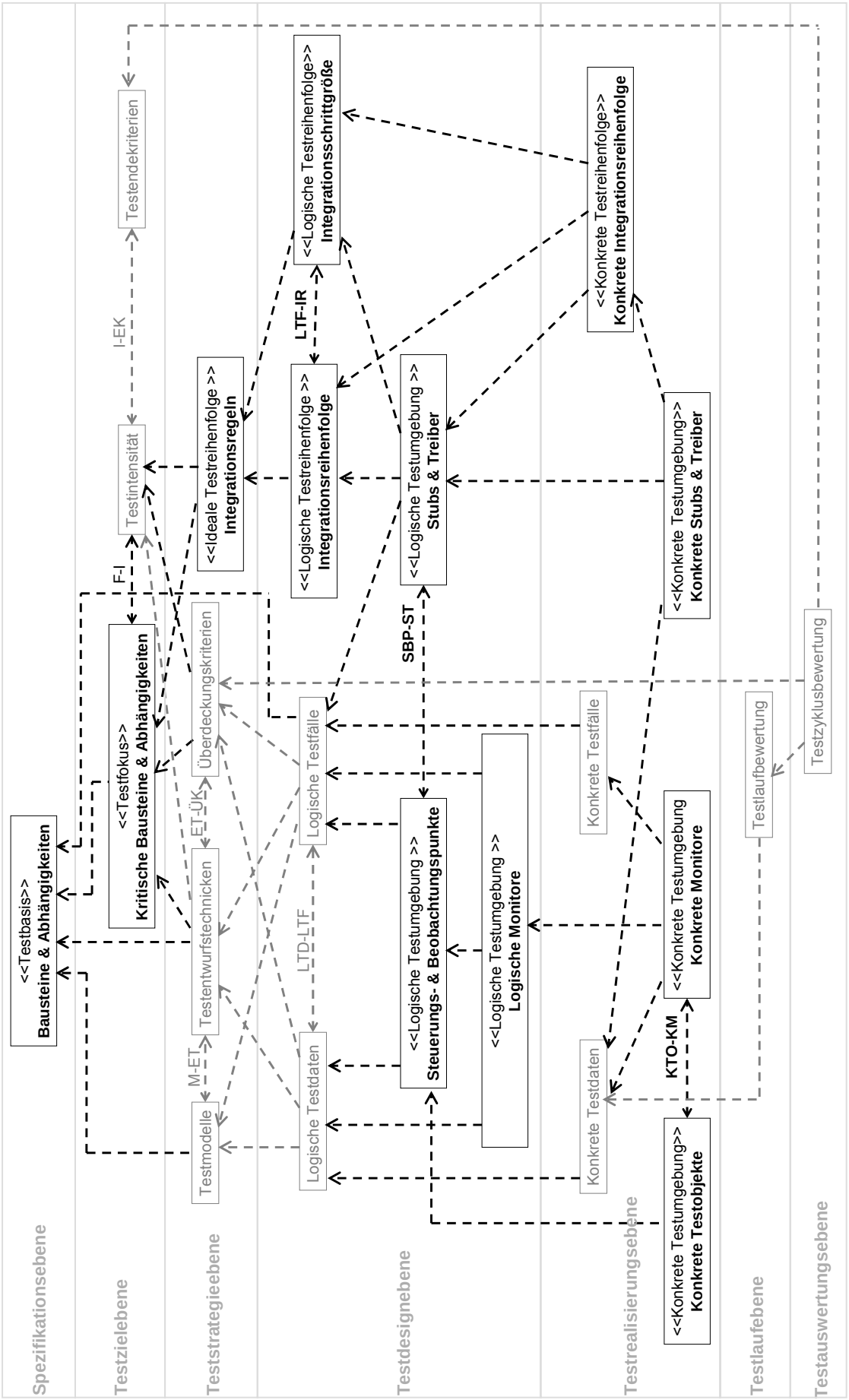


Abbildung 11: Abhängigkeiten zwischen den Entscheidungen im Integrationstestprozess

Sie stellt die Entscheidungen des allgemeinen und des Integrationstestprozesses gegenüber. Die rechte Seite der Abbildung beinhaltet die spezifischen Entscheidungen des Integrationstestprozesses. Die linke Seite zeigt die Entscheidungen des allgemeinen Testprozesses. Die Entscheidungen, die im Integrationstestprozess verfeinert werden, sind in der Abbildung auf der linken Seite durch einen breiten schwarzen Rahmen gekennzeichnet. Welche Entscheidung des allgemeinen Testprozesses verfeinert worden ist, ist zu jeder Integrationstestentscheidung in grauen Buchstaben über der Entscheidung vermerkt. So verfeinert die Entscheidung zur „Bausteinen & Abhängigkeiten“ die Entscheidung zur Testbasis.
In der Abbildung ist zu erkennen, dass speziell die Entscheidungen der Testdesign- und der Testrealisierungsebene stark verfeinert werden.
Wie auch im allgemeinen Testprozess lassen sich im Integrationstestprozess Testobjektbezogene und Prozessbezogene Entscheidungen identifizieren. Prozessbezogene Entscheidungen sind in der Abbildung gelb und Testobjektbezogene Entscheidungen grau hinterlegt. Für das Lesen der Hierarchie gelten ebenso die Regeln der *Entscheidungsabhängigkeit* und der *Parallelität*. Daraus ergibt sich, dass es im Integrationstestprozess zwischen den Entscheidungen zu unidirektionalen und bidirektionalen Abhängigkeiten kommen kann. Diese Abhängigkeiten sind in Abbildung 11 zusammengefasst. Die Entscheidungen und Abhängigkeiten, die aus dem generischen Testprozess im Integrationstestprozess übernommen wurden, sind mit grauer Schrift bzw. Pfeilen dargestellt. Die Entscheidungen und Abhängigkeiten die speziell im Integrationstest auftreten, sind mit schwarz dargestellt. Darüber hinaus sind die bidirektionalen Abhängigkeiten mit Namen gekennzeichnet, die einen grauen Hintergrund besitzen.
Die einzelnen Entscheidungsebenen, die enthaltenen Abhängigkeiten und die Abhängigkeiten zwischen ihnen werden nachfolgend beschrieben und am Beispiel der Onlineumfrage verdeutlicht.

3.2.2.1 Spezifikationsebene

Auf der Spezifikationsebene werden die Artefakte der Entwicklung genau untersucht und entschieden, ob mit den vorliegenden Artefakten der Integrationstestprozess begonnen werden kann. Hierbei wird entschieden, ob die Informationen aus der Testbasis, d.h. Informationen über die **Bausteine und Abhängigkeiten** der Software, ausreichend sind, um in den nachfolgenden Schritten den Testfokus und die Testintensität für den Integrationstestprozess definieren zu können. Die Informationen über die Abhängigkeiten müssen vorliegen, da sie die Testobjekte des Integrationstestprozesses sind. Die Bausteine werden schrittweise zu einem Gesamtsystem zusammengesetzt und ihre Abhängigkeiten untereinander getestet.

Beispiel-Onlineumfage
Während der Entwicklung wurden die typischen Phasen der Softwareentwicklung nicht durchlaufen, sondern auf eine agile Entwicklungsmethode zurückgegriffen. Dadurch stehen für den Integrationstest nur der Quelltext und die darin enthaltenen Kommentare zur Verfügung. Wir müssen jetzt entscheiden, ob auf Basis der gegebenen Artefakte aus dem Entwicklungsprozess der Integrationstestprozess begonnen werden kann. Diese Entscheidung können wir in Form folgender Frage mit zugehörigen Lösungsoptionen zusammenfassen:
Kann mit der gegebenen Testbasis der Testprozess begonnen werden?

a) Ja – Informationen sind ausreichend
b) Nein – Informationen sind nicht ausreichend ➔ Informationen müssen vervollständigt werden

In unserem Fall entscheiden wir, dass diese Informationen nicht ausreichend sind und weitere Informationen nachgereicht werden müssen. Aber welche Artefakte werden in unserem Integrationstestprozess benötigt?:

Welche Informationen müssen zur Auswahl des Testfokus und der Testintensität in der Testbasis ergänzt werden?

- ***a) Ein detailliertes Klassendiagramm mit allen Abhängigkeiten zwischen Bausteinen ist ausreichend***
- *b) Zusätzlich zu den statischen Modellen sind dynamische Modelle notwendig, z.B. Sequenzdiagramme*
- *c) Formale Kontraktspezifikationen für die Beschreibung der Interaktionen zwischen den Bausteinen ist notwendig*
- *d) …*

Die aufgezählten Lösungsalternativen stellen nicht alle möglichen Lösungsalternativen dar. Hier wäre noch eine Vielzahl weiterer Alternativen denkbar, die allerdings vom entsprechenden Projektkontext und den bereits vorliegenden Informationen abhängen. Aufgrund der geringen Größe unseres Softwaresystems entscheiden wir uns für die Alternative a), d.h. dass Entwurfsdokumente zu reproduzieren sind, um die beteiligten Klassen und ihre Abhängigkeiten zu veranschaulichen. Für das Reproduzieren verwenden wir den Quelltext, den wir mit Hilfe eines statischen Analysewerkzeugs analysieren, um die Bausteine und Abhängigkeiten zu identifizieren. Die Bausteine und Abhängigkeiten werden im Klassendiagramm modelliert (vgl. Abbildung 3, Kapitel 2.3). Die Richtung der Pfeile gibt an, welche Klasse von welcher Klasse abhängt.
Mit Hilfe der getroffenen Entscheidung(en) über die Testbasis und der daraus resultierenden Aktivitäten kann der Integrationstestprozess für das Beispielsystem begonnen werden.

3.2.2.2 Testzielebene

Auf dieser Ebene wird festgelegt, welche **kritischen Abhängigkeiten & Bausteine** es innerhalb des Softwaresystems gibt und somit getestet werden muss. Eine Abhängigkeit ist kritisch, wenn sie fehleranfällig ist, d.h. mit hoher Wahrscheinlichkeit viele Fehler besitzt. Die Entscheidung über kritische Abhängigkeiten und Bausteine ist abhängig von den Entscheidungen zu den *Bausteinen & Abhängigkeiten* der darüber liegenden Ebene. Einher mit der Entscheidung, was zu testen ist, geht die Entscheidung mit welcher **Testintensität** dieser Testfokus zu testen ist. Diese Entscheidung ist notwendig, um die verfügbaren Testressourcen auf die einzelnen zu testenden Abhängigkeiten zu verteilen. Innerhalb dieser Ebene sind in Abbildung 11 zwei bidirektionale Abhängigkeiten zu erkennen. Dies ist zum einen die Abhängigkeit zwischen den *kritischen Bausteinen und Abhängigkeiten* und der *Testintensität* („F-I“) und zum anderen die Abhängigkeit zwischen der *Testintensität* und den *Testendekriterien* („*I-EK*“). Diese Abhängigkeiten wurden aus der Entscheidungshierarchie des generischen Testprozesses übernommen.

Beispiel-Onlineumfrage
Auf dieser Ebene müssen wir entscheiden, welche Teile des Beispielsystems von uns wie intensiv zu testen sind und welche Testendekriterien wir innerhalb des Integrationstestprozesses erreichen wollen.
Mit Hilfe der erweiterten Testbasis, die alle Abhängigkeiten und Bausteine enthält, können wir überlegen, welche der Abhängigkeiten und Bausteine kritisch für das zu integrierende System sind. Hierzu muss in einem ersten Schritt festgelegt werden, welche Abhängigkeiten und Bausteine für uns als „testnotwendig“ erachtet werden. Diese Entscheidung lässt sich in Form der folgenden Frage und ihrer Lösungsmöglichkeiten dokumentieren:

Wie wählen wir aus, welche Abhängigkeiten wir testen wollen?

- *a) Zufällig, d.h. aus allen Abhängigkeiten wird zufällig eine bestimmte Anzahl von Abhängigkeiten ausgewählt*
- ***b) Durch Zuweisung einer Testnotwendigkeit zu jeder Abhängigkeit***
- *c) Gar nicht, jede Abhängigkeit muss getestet werden*
- *…*

Die angebotenen Lösungen haben alle ihre Vor- und Nachteile. Lösung „a)“ beispielsweise ermöglicht es, die geringen Ressourcen effizient auf wenige Abhängigkeiten zu verteilen, wobei die Art, wie die Abhängigkeiten ausgewählt werden, recht fraglich ist. Lösung „c“ sieht vor, alle Abhängigkeiten zu testen. Da aber nur beschränkte Ressourcen für das Testen zur Verfügung stehen, müssen diese wenigen Ressourcen später auf alle Abhängigkeiten verteilt werden, so dass Alles „nur ein wenig“ getestet werden kann, aber Nichts umfangreich.

Die Lösung „b)“ schwächt den Nachteil von Lösung „a)“ ab, indem die Auswahl der zu testenden Abhängigkeiten systematisch geschieht, was wiederum mit mehr Aufwand verbunden ist. Wir entscheiden uns in unserem Testprozess für die Alternative „b)“, da wir die zu testenden Abhängigkeiten gezielt auswählen wollen. Dabei wird jeder Abhängigkeit eine Testnotwendigkeit entsprechend ihres Typs zugewiesen. Hierfür betrachten wir jede Abhängigkeit, klassifizieren und bewerten sie. Diese Bewertungen sollten im Idealfall anhand von Erfahrungen aus früheren Projekten oder Versionen des zu integrierenden Softwaresystems durchgeführt werden[1]. Im Fall der Onlineumfrage verwenden wir das nachfolgende einfache Klassifikationsschema, das auf eigenen Erfahrungen der Tester der Onlineumfrage beruht:

1. Abhängigkeiten von externen Dateien, die das Verhalten des Systems beeinflussen, erhalten die höchste Priorität (1).
   Grund: In früheren eigenen Projekten hat sich gezeigt, dass externe Dateien, z.B. Konfigurationsdateien, die das Verhalten eines Softwaresystems beeinflussen, zu einer erhöhten Fehleranzahl führen.
2. Abhängigkeiten zwischen selbst erstellten Klassen erhalten eine mittlere Priorität (2).
   Grund: Selbst erstellte Bausteine sind nach eigenen Erfahrungen fehleranfälliger als Bausteine aus externen Bibliotheken, verursachen aber weniger Fehler als Abhängigkeiten zwischen Bausteinen und externen Dateien.
3. Abhängigkeiten zwischen selbst erstellten Klassen und Klassen aus Klassenbibliotheken erhalten die geringste Priorität (3)
   Grund: siehe Punkt 2.

Auf Basis dieser Priorisierung werden nun die Abhängigkeiten ausgewählt, die getestet werden müssen. Da nicht genügend Ressourcen für das Testen aller Abhängigkeiten zur Verfügung stehen, müssen die Abhängigkeiten ausgewählt werden, die aus unserer Sicht eine erhöhte Wahrscheinlichkeit aufweisen, Fehler zu enthalten. Diese Wahrscheinlichkeit wird durch die Priorität ausgedrückt, wobei Abhängigkeiten mit der Priorität 1 die höchste Wahrscheinlichkeit, Abhängigkeiten mit Priorität 2 eine mittlere Wahrscheinlichkeit und Abhängigkeiten mit der Priorität 3 die geringste Wahrscheinlichkeit besitzen. Für Abhängigkeiten, die eine geringe Wahrscheinlichkeit besitzen, legen wir fest, dass sie nicht getestet werden müssen, um Ressourcen zu sparen. Abhängigkeiten mit mittlerer und hoher Wahrscheinlichkeit (Priorität 1 und 2) werden im Integrationstest getestet.
Nun müssen wir noch festlegen, wie intensiv die einzelnen verbliebenen Abhängigkeiten zu testen sind. Hierbei kann die Testintensität in Anzahl Testfälle, Aufwand in Personentage, Geldmittel oder ähnlichen Kennzahlen angegeben werden. In unserem Fall wird die Testintensität in Anzahl der Testfälle angegeben[2]. Wir entscheiden uns dafür, dass alle Abhängigkeiten mit einer Priorität 1 mit 10 Testfällen und alle Abhängigkeiten mit einer Priorität 2 mit 5 Testfällen zu testen sind.
Die nächsten Entscheidungen betreffen die Testendekriterien. Wir müssen festlegen, wann unser Testprozess als erfolgreich beendet angesehen werden kann. Auch hier spielen wieder Erfahrungswerte eine große Rolle. Wir können festlegen, dass alle Testfälle ausgeführt werden müssen und keine Fehler gefunden werden dürfen oder dass nur 80% der Testfälle fehlerfrei ablaufen müssen. Hierbei dürfen jedoch die bereits getroffenen Entscheidungen nicht außer Acht gelassen werden, z.B. hinsichtlich des Testfokus. Die Testnotwendigkeit und die aufgewendete Testintensität sollten in die Festlegung der Testendekriterien einfließen. Dies bedeutet in unserem Testprozess, dass alle Testfälle, die Abhängigkeiten mit der Priorität 1 prüfen, zu 100% fehlerfrei durchlaufen müssen. Im Fall der Abhängigkeiten mit mittlerer Priorität reicht eine Quote von 80%.

3.2.2.3 Teststrategieebene

Auf der Teststrategieebene wird neben den Entscheidungen aus dem allgemeinen Testprozess eine integrationstestspezifische Entscheidung getroffen. Diese Entscheidung zu den **Integrationsregeln** konkretisiert die „allgemeine“ Entscheidung zur idealen Testreihenfolge. Mit Hilfe der Integrationsregeln wird festgelegt, nach welchen Regeln das System aus den einzelnen Bausteinen idealerweise zusammengesetzt wird. Diese Entscheidung hängt sowohl von der Entscheidung über die kritischen Abhängigkeiten und Bausteine als auch von der Entscheidung zur Testintensität ab. Darüber hinaus werden auf

[1] Ein systematisches Vorgehen zur Bestimmung des Testfokus wird in Kapitel 0 detailliert vorgestellt.
[2] Inwieweit die Anzahl der Testfälle als geeignete Kennzahl dient, wird in diesem Beispiel nicht diskutiert.

dieser Ebene Entscheidungen über die **Testmodelle**, die **Testentwurfstechnik** und die **Überdeckungskriterien** für den Integrationstestprozess getroffen. Diese Entscheidungen sind nur insoweit integrationstestspezifisch, als dass sie von den Entscheidungen zu den kritischen Abhängigkeiten und Bausteinen abhängig sind und nur Testmodelle, Testentwurfstechniken und Überdeckungskriterien berücksichtigen, die für den Integrationstest eingesetzt werden können.

Beispiel-Onlineumfrage
Für unseren Integrationstest legen wir im ersten Schritt die Testentwurfstechniken fest. Diese werden im Rahmen des Integrationstestprozesses benötigt, um Testfälle aus der gegebenen Testbasis abzuleiten. Der Einsatz einiger Testentwurfstechniken setzt die Existenz bestimmter Informationen, die in Form von Testmodellen abgelegt sind, voraus. Aus diesem Grund können die Entscheidungen zu den Testentwurfstechniken nicht unabhängig von der Entscheidung der Testmodelle getroffen werden.
Damit wir in unserem Fall eine Testentwurfstechnik auswählen können, prüfen wir in einem ersten Schritt, welche Informationen bereits vorliegen und ob sie ausreichend sind, um einige Testentwurfstechniken anwenden zu können. Derzeit liegen folgende Informationen vor:

- Testbasis:
  - Detailliertes Klassendiagramm, das die Abhängigkeiten zwischen den Bausteinen darstellt
  - Quelltext der Anwendungsklassen
- Testfokus (Kritische Abhängigkeiten & Bausteine) und Testintensität

Daraus ergibt sich die erste Frage, deren Beantwortung die Entscheidung zur Testentwurfstechnik explizit dokumentieren lässt.

Welche Testentwurfstechniken wenden wir an?

1. ***Kontrollflussbasierte Techniken***

a) Datenflussbasierte Techniken
b) Anforderungsbasierte Techniken
c) Zustandsorientierte Verfahren
d) Zufallsbasierte Techniken
e) …

In unserem Fall könnten wir Testentwurfstechniken anwenden, die sich auf die Existenz von Quelltexten stützen. Hierbei wären kontrollflussbasierte, datenflussbasierte oder zufallsbasierte Ansätze denkbar. Der Einsatz der ersten beiden Testentwurfstechniken erfordert das Erstellen von spezifischen Testmodellen, d.h. wir müssten noch die entsprechenden Testmodelle (Kontrollflussgraphen, Datenflussgraphen) erstellen, um mit Hilfe dieser Testmethoden die Testfälle abzuleiten. Bei der letzteren Testentwurfstechnik würden die Testfälle durch die Auswahl von zufälligen Parameterwerten für die einzelnen Operationen und Methoden abgeleitet werden.
Eine weitere Möglichkeit wäre das Ableiten der Testfälle anhand von Anforderungen oder die zustandsbasierten Testmethoden. Für beide Testentwurfstechniken liegen jedoch nicht genügend Informationen vor. Sie müssten in den Testmodellen nachträglich erstellt werden. Im Rahmen des kleinen Programms lohnt sich dieser Aufwand jedoch nicht.
Für unser kleines Programm haben wir uns für die kontrollflussbasierte Testtechnik entschieden. Eine Beschreibung für die kontrollflussbasierte Testtechnik im Integrationstest kann in [Sp90] gefunden werden.

Anmerkung: *Die Auswahl der Testentwurfstechnik(en) kann auch die Art der zu testenden Abhängigkeiten berücksichtigen. So wäre es denkbar, dass eine als kritisch eingestufte Vererbungsbeziehung mit speziellen Testentwurfstechniken für das Testen von Vererbungsbeziehungen getestet wird.*

Im nächsten Schritt muss entschieden werden, welche Testmodelle wir benötigen, um die ausgewählte Testentwurfstechnik zu unterstützen. Testmodelle sind Modelle, die die existierenden Informationen der Testbasis ergänzen und erweitern. Sie werden verwendet, um die notwendigen Informationen in der richtigen Form zur Verfügung zu stellen, um Testfälle daraus ableiten zu können.
In unserem Fall haben wir die Entscheidung über die Testmodelle bereits implizit getroffen. Durch die Entscheidung für eine kontrollflussbasierte Testentwurfstechnik haben wir festgelegt, dass wir Kontrollflussgraphen der einzelnen Klassen und kombinierte Kontrollflussgraphen benötigen, die die jeweiligen

Kontrollflüsse innerhalb des Systems beschreiben. Diese Graphen müssen jetzt vom Entwickler oder Integrationstestdesigner erstellt werden.
Anschließend müssen wir uns für die notwendigen Überdeckungskriterien entscheiden. Hier überprüfen wir, welche Überdeckungskriterien für die von uns gewählte Testentwurfstechnik anwendbar sind. In unserem Fall könnten folgende Alternativen in Frage kommen:

Welchen Überdeckungsgrad wollen wir erreichen?

- ***a) Alle Knoten des Kontrollflussgraphens***
- *b) Alle Kanten des Kontrollflussgraphens*
- *c) Alle Pfade im Kontrollflussgraphen*
- *d) …*

In unserem Fall entscheiden wir uns für Lösungsoption „a)“. Wir müssen die Testfälle also so wählen, dass jeder Knoten im Kontrollflussgraph mindestens in einem Testfall durchlaufen wird.
Als letztes legen wir die Integrationsregeln fest, d.h. welche Regeln die Integrationsreihenfolge einschränken. Hierbei sind für uns folgende Regeln denkbar:

Welche Integrationsregeln sollen für die Integrationsreihenfolge gelten?

- ***a) Testnotwendige Abhängigkeiten müssen möglichst früh getestet werden***
- *b) Unkritische Abhängigkeiten möglichst früh testen*
- ***c) Reihenfolge mit möglichst wenig Stubs***
- *d) …*

Hier wären viele Regeln denkbar. Beispielsweise könnten hier Kriterien aus dem Projektplan hineinspielen, d.h. wenn das Management eine frühzeitige Demonstration der Software wünscht, so müsste die Integration mit den Bausteinen beginnen, die die Schnittstelle zum Benutzer realisieren. Identifizierten Regeln lassen sich auch nach Belieben kombinieren, so dass die Regel „a)“ und Regel „c)“ gleichzeitig gelten müssen. In solchen Fällen ist jedoch sorgfältig zu prüfen, ob die Kombination sinnvoll ist. In unserem Fall streben wir eine Kombination an. Somit soll sich eine spätere Integrationsreihenfolge genau nach diesen Regeln richten, d.h. sie muss sowohl versuchen, Abhängigkeiten, die als Testfokus ausgewählt wurden, möglichst früh zu testen als auch eine Reihenfolge wählen, die möglichst wenig Stubs verwendet und somit den Simulationsaufwand minimiert.

3.2.2.4 Testdesignebene

Auf dieser Ebene gibt es eine Reihe von integrationstestspezifischen Entscheidungen, die die Entscheidungen zur logischen Testreihenfolge und zur logischen Testumgebung verfeinern. Die Festlegung einer **Integrationsreihenfolge** hängt von den definierten Integrationsregeln ab. Innerhalb dieser Entscheidung wird festgelegt, welcher Baustein wann zum System hinzugefügt und seine Abhängigkeiten getestet werden. Häufig basiert diese Entscheidung auf den Schätzungen der Fertigstellungszeitpunkte der einzelnen Bausteine und den definierten Integrationsregeln. Die Entscheidung über die **Integrationsschrittgröße** beeinflusst und wird beeinflusst („IR-ISG“) von der Entscheidung zur Integrationsreihenfolge. Sie legt fest, wie viele Bausteine in einem Integrationsschritt zum System hinzugefügt und getestet werden. Hierzu sind sowohl Informationen über die bereits geplante Integrationsreihenfolge als auch über die Entscheidungen der Integrationsregeln notwendig. Eine ermittelte Integrationsreihenfolge kann hinsichtlich der Erfüllung der Integrationsregeln eventuell nicht optimal sein. Durch die Integrationsschrittgröße ist es möglich, diesen Missstand zu beseitigen.
Aufbauend auf den Entscheidungen hinsichtlich der logischen Testfälle, der Integrationsreihenfolge und der Integrationsschrittgröße können die Entscheidungen zu den **Stubs & Treibern** gefällt werden. Hier wird festgelegt, welche Stubs und Treiber innerhalb der Testfälle unter der Voraussetzung der gewählten Integrationsreihenfolge und -schrittgröße benötigt werden. Für jeden Stub muss entschieden werden, ob es ein realistischer oder ein spezifischer Stub ist. Diese Entscheidung hat Einfluss auf die Wiederverwendbarkeit und Erstellungsaufwand. Ein realistischer Stub simuliert den Hauptteil der Funktionalität seines Originals und kann in vielen Testfällen eingesetzt werden. Ein

spezifischer Stub hingegen enthält nur ein Minimum der notwendigen Funktionalität und liefert im einfachsten Fall nur ein gewünschtes Ergebnis. Dadurch wird er weniger fehleranfälliger als der realistische Stub, kann aber in weniger Testfällen wieder verwendet werden. Die Entscheidung über den Typ des Stubs wird implizit in der Spezifikation der Stubs und Treiber dokumentiert. Die Herstellung und Beobachtung von internen Zuständen wird im Integrationstest durch Steuerungs- und Beobachtungspunkte realisiert. Hierbei werden die Testobjekte in einer späteren Aktivität instrumentiert, d.h. um Steuerungs- und Beobachtungspunkte erweitert. Auf der Testdesignebene muss entschieden werden, welche **Steuerungs- und Beobachtungspunkte** in den beteiligten Bausteinen der getesteten Abhängigkeiten benötigt werden, um die Testfälle ausführen zu können. Diese Entscheidung beeinflusst die Entscheidung der Stubs und Treiber, wird aber auch im gleichen Maße von diesen beeinflusst („*SBP-ST*“).
Nachdem nahezu alle Entscheidungen der Testdesignebene getroffen worden sind, muss abschließend noch die Entscheidung über die **logischen Monitore** gefällt werden. Hier wird definiert, welche Werkzeuge oder „zusätzliche Bausteine“ benötigt werden, um das tatsächliche Verhalten der Bausteine beobachten und aufzeichnen zu können. Hierbei werden die Entscheidungen zu den logischen Testdaten benötigt, um zu erfahren, welche Daten aufgezeichnet werden sollen. Die logischen Testfälle wiederum werden benötigt, um zu wissen, wann Daten oder innere Zustände aufgezeichnet und überprüft werden. Die Entscheidungen zu den Steuerungs- und Beobachtungspunkte geben Auskunft darüber, wie das Testobjekt die aufzuzeichnenden Daten an seine Umgebung abgibt.
Aus Abbildung 11 und den eben dargestellten Ausführungen zu den Entscheidungen der Testdesignebene kann entnommen werden, dass sich die Testdesignebene erneut in „Zwischenebenen“ gemäß den Regeln zur Parallelität und Entscheidungsabhängigkeit unterteilen lässt. Auf der obersten Zwischenebene befinden sich die Entscheidungen zu den logischen Testfällen und Testdaten, zur Integrationsreihenfolge und zur Integrationsschrittgröße. Die Entscheidungen zu Steuerungs- und Beobachtungspunkten sowie zu Stubs und Treibern sind von diesen Entscheidungen abhängig und befinden sich daher auf einer darunter liegenden Zwischenebene. Auf der untersten Zwischenebene ist die Entscheidung zu den logischen Monitoren zu finden. Durch diese Zwischenebene wird ersichtlich, dass die Entscheidungen des Integrationstestprozesses vielschichtiger sind, als die Entscheidungen des allgemeinen Testprozesses.

Beispiel-Onlineumfrage

Auf der Testdesignebene müssen wir für unser System die Entscheidungen über die Integrationsreihenfolge, die Integrationsschrittgröße, die Stubs und Treiber, die logischen Monitore sowie die Steuerungs- und Beobachtungspunkte treffen. In unserem Fall möchten wir uns in einem ersten Schritt Gedanken über die Integrationsreihenfolge machen. Hierbei müssen wir die festgelegten Integrationsregeln berücksichtigen: „*Testnotwendige Abhängigkeiten müssen möglichst früh getestet werden*“ und „*Reihenfolge mit möglichst wenig Stubs*“. Eine Reihenfolge mit wenig Stubs könnte durch die Verwendung der Bottom-Up Strategie (vgl. [Me79], [Bi00], [SW02], [SL05]) erreicht werden. Für das Beispiel der Onlineumfrage ist eine mögliche Reihenfolge in Tabelle 1 dargestellt, welche mit der Bottom-Up Strategie ermittelt wurde.
Der Integrationstestprozess könnte mit der vorgegebenen Reihenfolge in sieben Integrationsschritten abgeschlossen sein. Die ermittelte Reihenfolge legt dabei implizit die Größe der Integrationsschritte fest. Pro Schritt wird ein Baustein integriert. Für die Integration müssen die drei Stubs ErrorCategory, ResultFile und HTML realisiert werden. Die Reihenfolge erfüllt jedoch nicht vollständig die Forderung, dass „*Testnotwendige Abhängigkeiten müssen möglichst früh getestet werden*“ und „*Reihenfolge mit möglichst wenig Stubs*“ müssen. Die Abhängigkeit zwischen der Klasse XMLFileParse und XMLFile wird im ersten Integrationsschritt überprüft, aber die Abhängigkeit zwischen der Klasse Result und der Datei ResultFile wird erst im Schritt 5 geprüft. Um

die Forderung vollständige zu erfüllen, müsste die letzte Abhängigkeit früher integriert werden, was aber dazu führt, dass zusätzliche Stubs realisiert werden müssten[1].

Tabelle 1: Mögliche Integrationsreihenfolge für die Onlineumfrage

Integrations-schritt	Integrierte Bausteine	Getestete Abhängigkeit	Notwendige Stubs
1.	XMLFileParser, XMLFile	XMLFileParser→XMLFile	-
2.	ErrorCategoryList	ErrorCategoryList→XMLFileParser	ErrorCategory
3.	ErrorCategory	ErrorCategroyList→ErrorCategory	-
4.	Result	Result→ErrorCategoryList Result→ErrorCategory	ResultFile
5.	ResultFile	Result→ResultFile	-
6.	QuestionnaireServlet	QuestionnaireServlet→ErrorCategoryList QuestionnaireServlet→Result QuestionnaireServlet→ErrorCategory	HTML
7.	HTML	QuestionnaireServlet→HTML	-

Durch das Festlegen der Integrationsreihenfolge haben wir schon entschieden, welche Stubs und Treiber für die einzelnen Integrationstestschritte benötigt werden. Jedoch steht noch die Entscheidung aus, ob wir realistische oder spezifische Stubs verwenden wollen.

Von welcher Komplexität sollen die einzelnen Stubs sein?

- ***a) Spezifische Stubs, d.h. möglichst geringe Komplexität***
- *b) Realistische Stubs, d.h. möglichst hohe Wiederverwendbarkeit*
- *c) Stubs, die häufig benutzt werden, sollen möglichst realistisch sein, Stubs mit geringem Wiederverwendungsgrad sollen unkompliziert sein.*
- *d) …*

Der Vorteil eines realistischen Stubs ist die Fähigkeit, das Verhalten der zu simulierenden Klasse so naturgetreu wie möglich widerzuspiegeln. Somit kann er in vielen Testfällen wieder verwendet werden. Der Nachteil ist jedoch, dass der Aufwand für die Erstellung eines solchen Stubs sehr hoch ist und durch seine Komplexität, die Wahrscheinlichkeit steigt, bei der Realisierung einen Fehler zu machen. Im Gegenzug sind spezifische Stubs schnell erstellt, da sie nur einen Bruchteil der möglichen Funktionalität des zu simulierenden Bausteins realisieren. Diese Stubs können jedoch kaum in mehreren Testfällen wieder verwendet werden, wodurch eine große Anzahl dieser Stubs zu erstellen ist. In unserem Fall entscheiden wir uns für den spezifischen Stub, um die Fehleranfälligkeit des Stubs so gering wie möglich zu halten.
Die Entscheidung zu den logischen Testfällen, den logischen Testdaten und den Steuerungs- und Beobachtungspunkten werden jetzt im Rahmen dieses Beispiels nicht mehr näher betrachtet, da sie komplexe Kenntnisse über die ausgewählte Testmethode und den Quelltext des Beispielsystems voraussetzen würden. Es sei jedoch noch angemerkt, dass nun mit Hilfe der Testentwurfstechnik die logischen Testfälle und Testdaten spezifiziert werden. Hierbei müssen wir festlegen, welche Schritte in den einzelnen Testfällen auszuführen sind und welche Testdaten dabei an das integrierte System übergeben und vom integrierten System zurückgeliefert werden. Erst mit diesem Wissen können wir die Entscheidungen über notwendige Steuerungs- und Beobachtungspunkte treffen. In unserem Beispiel haben wir 60 Integrationstestfälle und die zugehörigen Testdaten spezifiziert. Abschließend müssen wir uns noch entscheiden, welche Monitore für die Ausführung der Testfälle notwendig sind. Da wir in jedem Schritt nur eine Abhängigkeit überprüfen wollen und es sich um ein sehr kleines Softwaresystem handelt, werden keine zusätzlichen Monitore benötigt. Die Aufzeichnung der Ergebnisse wird über die Treiber (die auch gleichzeitig die Testfälle realisieren) durchgeführt.

3.2.2.5 Testrealisierungsebene

Auf der Testrealisierungsebene werden die Entscheidungen zur konkreten Testreihenfolge und zur konkreten Testumgebung durch integrationstestspezifische Entscheidungen

[1] Ein Ansatz zur Berücksichtigung des Testfokus und des Simulationsaufwands in einer Integrationsreihenfolge wird in Kapitel 6.3 vorgestellt.

verfeinert. Die Entscheidung zur **konkreten Integrationsreihenfolge** spezialisiert die Entscheidung zur konkreten Testreihenfolge. In diese Entscheidungen fließen konkrete Projektbedingungen ein, die während der Planungs- und Designphase des Testprojekts noch nicht feststanden. So spielen hier tatsächliche Fertigstellungstermine der Bausteine eine Rolle. Durch Verzögerungen im Entwicklungsprozess oder durch Kürzungen des Etats kann es notwendig werden, die bereits aufgestellte Integrationsreihenfolge nochmals anzupassen. Hierbei fließen die Entscheidungen zur Integrationsreihenfolge, der Integrationsschritte und der Stubs und Treiber, aber auch die Priorität der logischen Testfälle ein. Parallel dazu werden auf dieser Ebene die Entscheidungen zu den **konkreten Testfällen** und **konkreten Testdaten** getroffen. Hierbei legt der Integrationstester fest, welche konkreten Test- und Prüfschritte in den Testfällen ausgeführt und welche konkreten Werte dabei verwendet werden.
Zu diesem Zeitpunkt liegen auch die implementierten Testobjekte vor. Hier muss der Integrationstester entscheiden, wie er die Entscheidungen über die Steuerungs- und Beobachtungspunkte in den Testobjekten umsetzt. Seine Entscheidungen sind implizit in den instrumentierten **konkreten Testobjekten** enthalten. Diese Entscheidungen beeinflussen jedoch die Auswahl der **konkreten Monitore** bzw. werden von den Entscheidungen über die konkreten Monitore beeinflusst („*KTO-KM*“). Bei der Entscheidung über die konkreten Monitore legt der Integrationstester fest, welche konkreten Werkzeuge er einsetzt bzw. wie die eigenen konkreten Monitore realisiert werden. Diese Entscheidungen können sowohl explizit dokumentiert werden (z.B. bei der Auswahl der konkreten Werkzeuge) aber auch implizit in den selbst erstellten konkreten Monitoren enthalten sein.
Die zuvor getroffenen Entscheidungen zur konkreten Integrationsreihenfolge, zu den konkreten Testdaten sowie zu den Stubs und Treibern fließen in die Entscheidung des Integrationstesters zu den **konkreten Stubs und Treiber** ein. Hier entscheidet er, wie die Stubs und Treiber konkret realisiert werden.
Alle Entscheidungen dieser Ebene werden benötigt, um die Ausführung der Testläufe zu ermöglichen. Dies schließt auch die Protokollierung der Ergebnisse ein.

Beispiel-Onlineumfrage
Auf die Entscheidungen dieser Ebene werden wir für die Onlineumfrage nicht näher eingehen, weil eine Übersicht über alle erstellten Testfälle und Testdaten notwendig wäre, um die Entscheidungen zu den konkreten Testfällen und Testdaten zu erläutern. Wir gehen noch kurz auf die Entscheidung der konkreten Integrationsreihenfolge ein. Da bereits alle zu integrierenden Bausteine vorliegen, ist keine Anpassung der zuvor festgelegten logischen Integrationsreihenfolge notwendig.

3.2.2.6 Testlaufebene

Auf der Testlaufebene führt der Integrationstester gemeinsam mit dem Integrationstestdesigner und dem Testmanager die **Testlaufbewertung** durch. Die Entscheidungen, ob ein Fehler gefunden wurde oder nicht, ist bereits im allgemeinen Testprozess beschrieben worden und hat keinen besonderen Charakter im Integrationstestprozess.

Beispiel-Onlineumfrage
Alle spezifizierten Testfälle unseres Beispielsystems liefen, bis auf drei Ausnahmen, fehlerfrei durch. Jetzt müssen wir für jeden fehlerhaften Testlauf entscheiden, ob es sich bei der Abweichung zwischen erwartetem Soll-Verhalten und beobachtetem Ist-Verhalten um einen Fehler in der Software oder um einen Fehler in unseren Testartefakten handelt. Es stellte sich heraus, dass zwei Testläufe tatsächlich Fehler in Abhängigkeiten identifizierten. Der dritte Fehler ist auf eine falsch hergestellte Vorbedingung für den Testlauf zurückzuführen. Beide gefundenen Fehler werden von uns mit einer hohen Priorität eingestuft, da sie die Kommunikation zwischen der Anwendungsklasse ErrorCategory und der Oberflächenklasse QuestionnaireServlet betrifft und zu einem gravierenden Fehlverhalten geführt hat.

3.2.2.7 Testauswertungsebene

Der Testmanager führt am Ende eines Testzyklusses eine **Testzyklusbewertung** durch. Hierzu entscheidet er, ob die Testaktivitäten erfolgreich waren und beendet werden können oder ob weitere Testaktivitäten notwendig sind. Diese Entscheidung ist bereits im Kapitel über den allgemeinen Testprozess (Kapitel 3.1.3.7, Seite 36) beschrieben worden und hat keine integrationstestspezifischen Eigenheiten.

Beispiel-Onlineumfrage

Auf dieser Ebene müssen wir entscheiden, ob wir das System ausreichend getestet haben. Hierbei helfen uns die auf der Testzielebene festgelegten Testendekriterien (100% fehlerfreie Testläufe für Abhängigkeiten mit Priorität 1, und 80% für Abhängigkeiten mit Priorität 2).

Haben wir ausreichend getestet, um die definierten Testendekriterien zu erreichen?

- ***a) Ja, Testendekriterien sind erreicht, Anpassung der Kriterien nicht notwendig, Testprozess kann beendet werden***
- *b) Ja, Testendekriterien sind erreicht, aber es ist eine Anpassung der Kriterien notwendig oder einige Entscheidungen müssen neu getroffen oder einige Testaktivitäten müssen neu ausgeführt werden.*
- *c) Nein, Testendekriterien sind nicht erreicht, einige Entscheidungen müssen neu getroffen und einige Testaktivitäten müssen erneut ausgeführt werden.*
- *d) …*

Da wir die definierten Testendekriterien erreicht haben und auch während des Testprozesses keine neuen, nicht berücksichtigten Informationen aufgetaucht sind, die das erneute Treffen von vorangegangenen Testentscheidungen verlangen, können wir festlegen, dass der Testprozess abgeschlossen werden kann.

Die in 3.2.1 vorgestellten Rollen sind für das Treffen der Entscheidungen im Integrationstestprozess verantwortlich. Diese Entscheidungen werden während der Ausführung verschiedener Testaktivitäten getroffen. Die Testaktivitäten des Integrationstests unterscheiden sich kaum von den Testaktivitäten des generischen Testprozesses (vgl. Abbildung 4). Die in Abbildung 8 dargestellte Zuordnung der Entscheidungen des generischen Testprozesses ist auch für den speziellen Integrationstestprozess anwendbar.
Der große Unterschied besteht jedoch darin, dass die zu treffenden Entscheidungen innerhalb der Aktivitäten verfeinert worden sind. Diese verfeinerte Entscheidungshierarchie zeigt, auf welche Eigenheiten die Rollen des Integrationstestprozesses achten müssen. Sie gibt ihnen Richtlinien in Form von Entscheidungen vor, worauf sie speziell im Integrationstestprozess achten müssen.

3.3. Anwendung des Integrationstestprozesses

Der beschriebene Ablauf des Integrationstestprozesses kann, ähnlich wie der generische Testprozess, für unterschiedliche Zwecke eingesetzt werden. In erster Linie dient er zur Planung, Durchführung und Steuerung des Integrationstestprozesses. Die Entscheidungshierarchie kann als Checkliste verwendet werden, die die zu treffenden Entscheidungen auflistet. Getroffene Entscheidungen können nach erfolgreichem Treffen auf der Liste abgehakt werden. Somit kann anhand der Checkliste jederzeit der aktuelle Stand des Integrationstestprozesses abgelesen werden. Sie zeigt, welche Entscheidungen bereits getroffen wurden und welche noch zu treffen sind. Die Entscheidungen des Prozesses sollten dabei bewusst getroffen und explizit in den Testartefakten dokumentiert werden. Hierbei kann der in [MYB+91] beschriebene QOC-Ansatz (Question, Option, Criteria) Verwendung finden. Die zu treffende Entscheidung wird als Frage (Question) formuliert. Zu dieser Frage werden alle möglichen Lösungsalternativen (Options) identifiziert und dokumentiert. Anschließend werden Bewertungskriterien (Criteria) definiert, gegen die die Lösungsalternativen bewertet werden. Als Bewertungsskala kann eine 5-Stufen-Skala (z.B.

„**++**“, „**+**“, „**+/-**„, „**-**„, „**--**", wobei „**++**“ die vollständige Erfüllung und „**--**„ die vollständige Nichterfüllung des Kriteriums bezeichnet) verwendet werden. Jede Lösungsalternative wird gegen jedes Kriterium bewertet. Anschließend wird die Lösungsalternative als finale Lösung ausgewählt, die die einzelnen Kriterien am besten unterstützt. Die Entscheidung wurde getroffen, dokumentiert und begründet. Der checklistengestützte Integrationstestprozess kann überall dort eingesetzt werden, wo ein Integrationstestprozess geplant und durchgeführt wird.
Darüber hinaus kann die beschriebene Entscheidungshierarchie verwendet werden, um die Einführung eines systematischen Integrationstestprozesses zu unterstützen. Diese Einführung setzt die Kenntnis des Ist-Prozesses voraus, d.h. wie der Integrationstestprozess derzeit durchgeführt wird. Die Entscheidungshierarchie kann bei der Ist-Analyse wertvolle Unterstützung leisten. Eine Checkliste mit allen zu treffenden Entscheidungen wird verwendet, um die im aktuellen Ist-Prozess getroffenen Entscheidungen zu identifizieren. Dabei kann unterschieden werden, ob die jeweilige Entscheidung gar nicht, unbewusst oder bewusst getroffen wird. Damit geht die Frage einher, ob eine Entscheidung implizit oder explizit in den Testartefakten enthalten ist. Aufbauend auf dieser Ist-Analyse kann ein Plan für einen Integrationstestprozess erstellt werden, der die zu treffenden Entscheidungen explizit fordert. Dabei müssen nicht sofort alle Entscheidungen auf einmal bewusst getroffen und explizit dokumentiert werden, vielmehr kann die Veränderung des Ist schrittweise geschehen. Auf diese Weise wird die Akzeptanz bei den beteiligten Personen erhöht werden, da ein Großteil der Artefakte und Entscheidungen wie gewohnt entsteht und sich nur ein kleiner Teil ändert. Dieser Änderungsprozess sollte sich langsam über einen größeren Zeitraum und über mehrere Projekte hinweg in einem Unternehmen vollziehen.

Der Integrationstestprozess dient auch zur Klassifikation von existierenden Ansätzen im Bereich des Integrationstests. Für einen vorgestellten Ansatz können die Entscheidungen identifiziert werden, die dieser Ansatz adressiert. Auf diese Weise können verschiedene Ansätze nach entsprechenden Entscheidungen klassifiziert und miteinander verglichen werden.
Ein letzter Punkt, der für die vorliegende Arbeit von großer Bedeutung ist, ist die Verwendung des Integrationstestprozesses, um offene Forschungsfragen im Rahmen des Integrationstestprozesses zu identifizieren. Für die wenigsten Entscheidungen gibt es derzeit zufrieden stellende Unterstützung, wie diese Entscheidungen am besten zu treffen sind.

Die nachfolgenden Kapitel konzentrieren sich auf die Unterstützung der Entscheidungen zum Testfokus und zur Integrationsreihenfolge. Bei der Entscheidung zum Testfokus existieren derzeit keine bekannten Ansätze, die es ermöglichen, die Abhängigkeiten gezielt als Testobjekte des Integrationstests auszuwählen. Für die Entscheidung zur Integrationsreihenfolge gibt es einige Ansätze, diese konzentrieren sich jedoch nur auf die Minimierung des Simulationsaufwands, vernachlässigen aber die Testfokusauswahl.

4. Klassifikation von Abhängigkeiten

Im Integrationstestprozess stehen die interne Struktur und das interne Verhalten eines Softwaresystems im Mittelpunkt. Doch bevor mit Hilfe von Testentwurfstechniken und Testmodellen Testfälle für die Testobjekte des Integrationstests abgeleitet werden können, muss den beteiligten Rollen bewusst sein, was im Integrationstest getestet werden kann. Die zu testenden Teile des Softwaresystems im Integrationstest sind die Abhängigkeiten. Detaillierte Informationen über eine Abhängigkeit (und ihre Eigenschaften) sind im Integrationstestprozess von großem Wert. Sie können Auskunft über mögliche Fehler liefern, die in dieser Abhängigkeit auftreten können. „It is important to identify possible dependencies […] to distinguish sources (roots) of potential problems, where more attention is needed“ ([VR02], S. 63). Abhängigkeitseigenschaften können aber auch Informationen bereitstellen, die für das Ermitteln der Integrationsreihenfolge und somit für das Realisieren der Testumgebung (Stubs, Treiber, Monitore) notwendig sind.
In der vorliegenden Arbeit wird einen Katalog von Abhängigkeitseigenschaften bereitgestellt, der es ermöglicht, Abhängigkeiten durch diese Eigenschaften zu klassifizieren. Dieser Katalog ist Basis für die Ansätze zur Testfokusauswahl und zur Ermittlung der Integrationsreihenfolge, die in den Kapiteln 5 und 6 vorgestellt werden. Aus diesem Grund wird für jede Eigenschaft angegeben, ob sie Einfluss auf die Testnotwendigkeit und somit auf die Testfokusauswahl und/oder auf den Simulationsaufwand und somit auf die Integrationsreihenfolge besitzt.

Definition: *Die **Testnotwendigkeit** einer Abhängigkeit beschreibt, ob die Abhängigkeit im Rahmen des Integrationstestprozesses getestet werden sollte. Die Testnotwendigkeit spiegelt die Wahrscheinlichkeit eines Fehlers, der durch die Ausprägung bestimmter Abhängigkeitseigenschaften eintreten kann, wider.*

Definition: *Der **Simulationsaufwand** einer Abhängigkeit beschreibt den Aufwand für das Simulieren des unabhängigen Bausteins für den Fall, dass er als Stub für die gewählte Integrationsreihenfolge realisiert werden muss. Bestimmte Ausprägungen von Abhängigkeitseigenschaften haben direkten Einfluss darauf, wie aufwändig es ist, den unabhängigen Baustein zu simulieren.*

Die einzelnen Abhängigkeitseigenschaften des Katalogs und ihr Einfluss auf die Testnotwendigkeit und Simulationsaufwand werden nachfolgend in Kapitel 4.1 vorgestellt. Kapitel 4.2 gibt einen Überblick über existierende Ansätze, welche eine Klassifikation von Abhängigkeiten ermöglichen. Abschließend in Kapitel 4.3 werden weitere Einsatzmöglichkeiten des Katalogs vorgestellt.

4.1. Eigenschaftskatalog für Abhängigkeiten

Der Eigenschaftskatalog für Abhängigkeiten unterstützt in erster Line die Entscheidungen zur Testfokusauswahl und zur Integrationsreihenfolgermittlung des Integrationstestprozess. Seine einzelnen Eigenschaften werden nachfolgend beschrieben. Für jede Eigenschaft wird ihr Einfluss auf den Integrationstestprozess geschildert und auf mögliche Quellen verwiesen, in denen Informationen über die Existenz dieser Eigenschaft zu finden sind. Zur Veranschaulichung der Eigenschaften werden an geeigneter Stelle Beispiele verwendet. An vielen Stellen wird auf das Beispiel der Onlineumfrage zurückgegriffen (vgl. Kapitel 2.3).

Die Abhängigkeitseigenschaften lassen sich in drei Kategorien unterteilen: Kategorie der Laufzeiteigenschaften, Kategorie der Entwicklungseigenschaften und Kategorie der Deploymenteigenschaften. **Laufzeiteigenschaften** spezifizieren die Eigenschaften der Abhängigkeit, die für die Laufzeit des Systems von Interesse sind. Sie beinhalten sowohl die statischen als auch dynamische Eigenschaften der Abhängigkeit. **Entwicklungseigenschaften** fassen Informationen aus dem Entwicklungsprozess zusammen. Die **Deploymenteigenschaften** beschreiben die Verteilung der beteiligten Bausteine, die an der Abhängigkeit beteiligt sind.

4.1.1. Kategorie der Laufzeiteigenschaften

Die Kategorie der Laufzeiteigenschaften stellt die größte Kategorie im vorliegenden Katalog dar. Sie enthält Eigenschaften zu Client/Server-Abhängigkeiten, zu Vererbungsabhängigkeiten und zu indirekten Abhängigkeiten. Die Eigenschaften der Kategorie sind in
Abbildung 12 dargestellt. Die Färbung der Eigenschaften gibt Auskunft über ihren Einfluss auf den Integrationstest. Eine graue Färbung symbolisiert, dass die Eigenschaft Einfluss auf die Testnotwendigkeit der Abhängigkeit hat. Die weiße Färbung steht für den Einfluss auf den Simulationsaufwand. Eigenschaften mit einer kontinuierlichen Färbung von weiß nach grau symbolisieren Eigenschaften, die sowohl Einfluss auf die Testnotwendigkeit als auch auf den Simulationsaufwand haben.
In der Abbildung wird die Unterscheidung zwischen atomaren und komplexen Eigenschaften getroffen. Eine komplexe Eigenschaft kann für sich betrachtet werden oder sie kann durch weitere Eigenschaften verfeinert werden. Die Eigenschaft *Modifikationen* kann für sich als eine Eigenschaft betrachtet werden und die Werte WAHR oder FALSCH annehmen. Wenn aber detaillierter Informationen zur Verfügung gestellt werden sollen, um die Modifikation näher zu beschreiben, dann können Informationen über die Anzahl der neuen oder überschriebenen Services hinzugezogen werden. Atomare Eigenschaften können nicht weiter aufgeteilt werden[1].
Die Laufzeiteigenschaften können bei näherer Betrachtung noch einmal in drei Kategorien aufgeteilt werden. **Client/Server** Abhängigkeiten beschreiben Abhängigkeiten, bei denen auf Services und Attribute zugegriffen wird. **Vererbungsabhängigkeiten** können nur in objektorientierten Systemen auftreten und beschreiben Eigenschaften, die bei der Vererbung zwischen zwei Klassen eine Rolle spielen. Eine **indirekte** Abhängigkeit ist eine Abhängigkeit, die durch Dritte hervorgerufen wird. Beide Bausteine haben hierbei keine direkte Abhängigkeit (Client/Server und/oder Vererbung) sondern hängen durch einen dritten Baustein voneinander ab (vgl. [Ma03].

4.1.1.1 Client/Server Eigenschaften

In einer Client/Server Beziehung/Abhängigkeit nimmt der abhängige Baustein die Rolle des Clients und der unabhängige Baustein (vgl. Abbildung 1, Seite 17) die Rolle des Servers ein. Der Client kann auf Services und Attribute des Servers zugreifen, um seine eigene Funktionalität zu realisieren. Dabei beeinflusst er nicht nur sein Verhalten, sondern gegebenenfalls auch das Verhalten des Servers. Die Eigenschaften von Interesse sind hierbei die **Anzahl der Serviceaufrufe** und die **Anzahl der Attributzugriffe** vom Client zum Server. Die Existenz eines **Kommunikationsvertrags** (vgl. [VR02]) kann die Zugriffe und Aufrufe einschränken.

[1] Allerdings erhebt der vorliegende Katalog keinen Anspruch auf Vollständigkeit. Bei einer Erweiterung des Kataloges kann es sinnvoll sein, existierende atomare Eigenschaften zu verfeinern. Zum Fertigstellungszeitpunkt der Arbeit gab es jedoch keine Anzeichen dafür, dass eine weitere Aufteilung notwendig wäre.

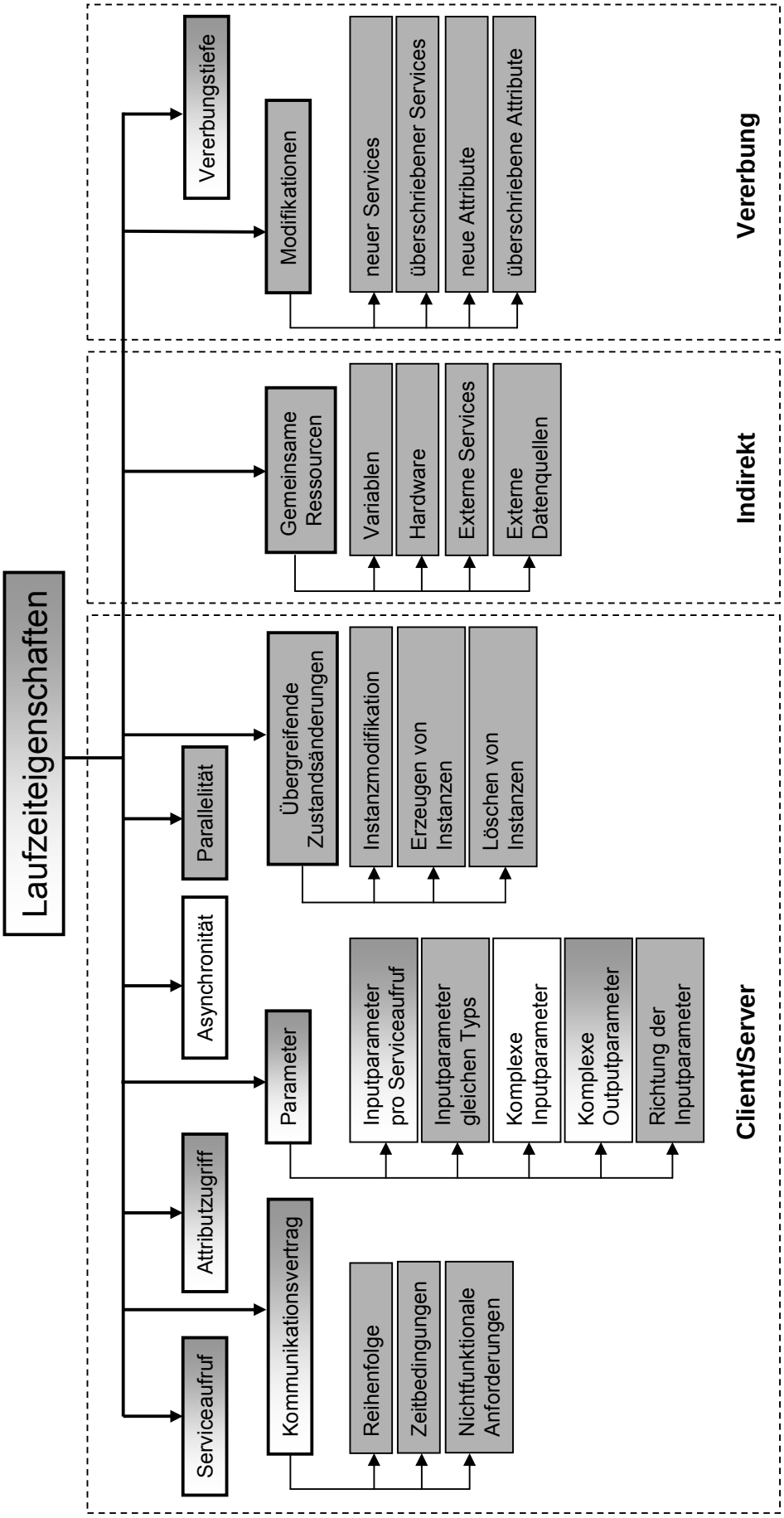


Abbildung 12: Abhängigkeitseigenschaften der Kategorie „Laufzeiteigenschaften

Parameter können während eines Serviceaufrufs übergeben bzw. entgegengenommen werden. Serviceaufrufe und Attributzugriffe können weitreichende Folgen im gesamten Softwaresystem nach sich ziehen. Diese **übergreifenden Zustandsänderungen** spezifizieren diese Änderungen im System näher. Die **Parallelität** und die **Asynchronität** ergänzen die Client/Server Beziehung, um zwei zusätzliche Eigenschaften. Die Parallelität beschreibt, ob die beteiligten Bausteine ihre Funktionalität parallel anbieten und auch ausführen können. Sie befinden sich somit in unterschiedlichen Systemprozessen. Die Asynchronität gibt an, inwiefern die Kommunikation in der Client/Server Beziehung synchron oder asynchron stattfindet.
Zwei Beispiele für Client/Server Abhängigkeiten in der Programmiersprache Java sind der Onlineumfrage entnommen und in Quelltext 1 auf der nachfolgenden Seite zu sehen. Die erste Zeile aus der Klasse `ErrorCategoryList` beschreibt einen Aufruf einer Instanz der Klasse `ErrorCategoryList` auf sich selbst (Attributabfrage) sowie einen Serviceaufruf auf eine Instanz der Klasse `ArrayList`. Die zweite Quelltextzeile aus der Klasse `QuestionnaireServlet` beschreibt einen Aufruf im Rahmen einer `if`-Anweisung. Die Instanz der Klasse `QuestionnaireServlet` ruft an einer Instanz der Klasse `ErrorCategory` den Service `getSubcategories()` ohne Parameter auf.
Die Eigenschaften von Client/Server Abhängigkeiten werden nachfolgend beschrieben.
Informationen über die Client/Server Abhängigkeiten zwischen zwei Bausteinen können in Entwurfs- und Architekturmodellen, aber vor allem im Quelltext gefunden werden.

```
this.categories.add(mainErrorCategory);

if(activeErrorCategory.getSubcategories()!=null)
```

Quelltext 1: Beispiele einer Client/Server Abhängigkeit im Quelltext repräsentiert

Serviceaufruf
Ein Serviceaufruf[1] stellt eine Kommunikation vom Client zum Server dar. Dabei ruft der Client mindestens einen Service beim Server auf. In jedem Softwaresystem, welches aus mehr als einem Baustein besteht, ist die Kommunikation notwendig, beispielsweise um Informationen und Daten auszutauschen oder vorhandene Funktionalität zu verwenden. Diese Eigenschaft hat sowohl Einfluss auf die Testnotwendigkeit als auch auf den Simulationsaufwand. Je größer die Anzahl der aufgerufenen Services in dieser Abhängigkeit ist, desto größer ist die Anzahl der Services, die simuliert werden müssen. Gleichzeitig kann eine hohe Anzahl an Serviceaufrufen ein erhöhtes Fehlerrisiko der Abhängigkeit in sich bergen. In [Bi96] sind mögliche Fehler, die während eines Serviceaufrufs auftreten können, zu finden (Seite. 162, nach Firesmith [Fi92]). „*Message sent to wrong supplier*“: die Nachricht wurde an die falsche Serverinstanz oder an eine Instanz eines falschen Servertyps geschickt. „*Message not in supplier*“: der aufgerufene Service ist nicht im Server realisiert. „*Documentation/code mismatch*“: die Dokumentation über den aufgerufenen Service ist falsch oder falsch vom Entwickler interpretiert worden. Dadurch wurde der falsche Service aufgerufen.

Attributzugriff
Ein Attributzugriff beschreibt das Lesen und/oder Modifizieren eines Attributes oder einer Variable im Server. Ähnlich zu den Serviceaufrufen erhöht die Anzahl der zugegriffenen Attribute den Simulationsaufwand einer Abhängigkeit. Je mehr Attribute zugegriffen werden, desto mehr Attributzugriffe müssen simuliert werden. Ähnlich verhält es sich mit der

[1] Der Begriff Service wird im Katalog synonym mit den Begriffen Dienst oder Methode verwendet.

Testnotwendigkeit. Je höher die Anzahl der Attributzugriffe, desto höher ist auch die Wahrscheinlichkeit, dass ein Fehler eintritt. Der Fehler könnte dabei sowohl im Client als auch im Server zu finden sein. Ein Zugriff auf ein nicht initialisiertes Attribut (vgl. [Bi96]) kann zu einem Fehler führen. Sowohl die Funktionalität des Clients als auch des Servers kann vom Zustand des Attributs abhängig sein. Eine Veränderung könnte somit ein nicht gewolltes Verhalten nach sich ziehen. Zusätzlich können ähnliche Probleme wie beim Serviceaufruf auftreten: es wurde auf das falsche Attribut, auf ein Attribut der falschen Serverinstanz oder des falschen Servertyps zugegriffen.

Parallelität

Parallelität gibt an, ob zwei Bausteine ihre Funktionalität parallel zueinander ausführen können. Dies spielt insbesondere dann eine Rolle, wenn beide Bausteine gleichzeitig auf gemeinsame Ressourcen zugreifen wollen oder gemeinsam den gleichen Service aufrufen möchten. Die häufigsten Fehler, die durch parallele Ausführung entstehen, sind Fehler, die durch Wettlaufbedingungen und Deadlocks hervorgerufen werden. Besonders schwierig ist das Testen von parallelen Prozessen und Bausteinen im Rahmen des Integrationstestprozesses. Viele Fehler, die durch Parallelität verursacht werden, lassen sich nur schwer aufdecken, da sie schwer zu reproduzieren sind. Darüber hinaus fehlt es an unterstützenden Integrationstestentwurfstechniken, um speziell diese Fehler zu adressieren.
Verstärkt wird das Problem durch die mangelnden Dokumentationsmöglichkeiten für Parallelität. Das Modellieren von Parallelität ist sehr aufwändig, aber nicht unmöglich. Die UML [UML09] bietet mit den Aktivitäts- und Sequenzdiagrammen sowie dem Zustandsautomaten drei Möglichkeiten an, Nebenläufigkeit zu modellieren [RHQ+05].
Die Eigenschaft Parallelität hat somit Einfluss auf die Testnotwendigkeit, wenn sich zwei abhängige Bausteine in unterschiedlichen Prozessen befinden.

Asynchronität

Die Eigenschaft Asynchronität beschreibt, ob eine Kommunikation zwischen zwei Bausteinen asynchron oder synchron ausgeführt wird. Im synchronen Fall ruft der Client am Server einen Service auf, unterbricht seine aktuell ausgeführte Funktionalität und wartet auf das Ergebnis des Serviceaufrufs. Das Steuerungsmuster „Call-Return“ aus dem Bereich des Architekturdesigns [So07], [Bu03] ist hierfür ein Beispiel. Im asynchronen Fall setzt der Client nach getätigtem Serviceaufruf seine Funktionalität fort und empfängt die Antwort des Aufrufs parallel zu seiner ausgeführten Funktionalität. Sollte im synchronen Fall der Server seine Funktionalität nicht beenden können und somit nicht in der Lage sein, seine Antwort an den Client zurückzusenden, kann es zu einem „endlosen“ Warten des Clients kommen. In diesem Fall kann der Client seine Funktionalität nicht beenden. In Softwaresystemen, die nur einen „Thread“ verwenden und keine parallelen Ausführungen beinhalten, können nur synchrone Aufrufe stattfinden.
Im asynchronen Fall kann der Client zwar seine Funktionalität fortsetzen, aber im Fall, dass die Antwort des Servers ausbleibt, können die Berechnungen des Clients, die auf den Antworten des Serviceaufrufes aufbauen, falsche Ergebnisse liefern. Diese Fehler können durch Wettlaufbedingungen (so genannte Race Conditions) auftreten, wenn der Client seine Berechnungen schneller ausführt als der Server die benötigte Antwort sendet. Das Ziel des Integrationstests ist es, die Fehler, die durch solche Wettlaufbedingungen auftreten können und zu Deadlocks in der Ausführung führen, frühzeitig zu erkennen. Die Identifikation der Informationen über die Eigenschaft der Asynchronität stellt jedoch die Integrationstester vor eine große Herausforderung. Hier helfen Modelle, die das dynamische Verhalten und die konkreten Aufrufbeziehungen zwischen den beteiligten Bausteinen darstellen. Dies können beispielsweise UML-Sequenzdiagramme sein. Ebenso können die Informationen aus dem

Quelltext entnommen werden. Dabei muss geprüft werden, ob sich die Bausteine in parallelen Kontrollprozessen (Threads) befinden.
Die zwei Aufrufe in Quelltext 1 sind ein Beispiel für synchrone Serviceaufrufe. Aus einem Klassendiagramm kann dies nicht herausgelesen werden. Das Sequenzdiagramm in Abbildung 13 beschreibt eine asynchrone Kommunikation zwischen Baustein A und Baustein B. Baustein A ruft an Baustein B einen Service auf und setzt anschließend seine Berechnungen fort. Während seiner Funktionsausführung ruft er an Baustein C einen weiteren Service auf und setzt seine Berechnung fort. Nach einiger Zeit empfängt er die Antworten zuerst von Baustein C und anschließend von Baustein B.

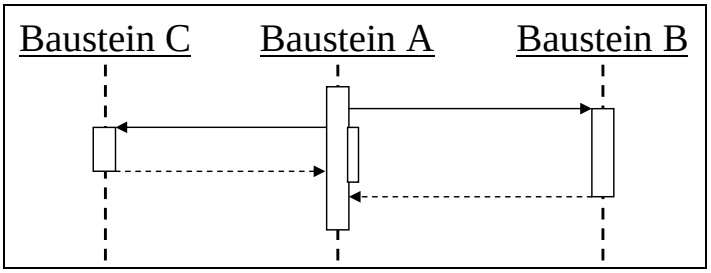


Abbildung 13: Beispiel einer asynchronen Kommunikation

Asynchrone Serviceaufrufe treten nur in Systemen auf, die auch die Eigenschaft der Parallelität erfüllen (vgl. Eigenschaft Parallelität). Die durch Parallelität und Asynchronität auftretenden Wettlaufbedingungen führen dazu, dass das Erstellen und die Ausführung von Testfällen umfangreich und kompliziert sind. Bei jedem Testlauf kann es zu unterschiedlichen Ergebnissen kommen, je nachdem, welche Serviceaufrufe wann getätigt wurden und wie verschieden schnell die einzelnen Bausteine darauf antworten. So könnte es im Beispiel in Abbildung 13 passieren, dass erst Baustein B und anschließend Baustein C die Antwort an Baustein A liefert. Hierdurch kann die Berechnung von Baustein A ein anderes Ergebnis liefern. Diese unerwünschten Nebeneffekte aufzudecken, ist eine komplizierte Aufgabe des Integrationstests.
Es gibt in der Literatur, und auch aus eigenen Erfahrungen, keine Hinweise darauf, dass asynchrone Abhängigkeiten fehleranfälliger sind als synchrone Abhängigkeiten. Daher hat die Eigenschaft keinen Einfluss auf die Testnotwendigkeit. In beiden Fällen muss getestet werden, ob die Kommunikation korrekt funktioniert. Hinsichtlich des Simulationsaufwandes kann gesagt werden, dass asynchrone Abhängigkeiten schwerer zu simulieren sind, als synchrone, da im simulierten Baustein der Aspekt der Asynchronität berücksichtigt werden muss. Zusätzlich erhöht eine asynchrone Abhängigkeit den Aufwand für das Erstellen von Monitoren, um das korrekte Verhalten aufzuzeichnen.

Kommunikationsvertrag
Diese Eigenschaft gibt an, ob es einen Vertrag (synonym auch Contract) zwischen dem Client und dem Server gibt. In diesem Vertrag können Informationen über einzuhaltende Reihenfolgen, Zeitbedingungen oder Qualitätsmerkmale definiert werden. Die **Reihenfolge** schreibt vor, in welcher Reihenfolge Serviceaufrufe und Attributzugriffe vom Client zum Server durchgeführt werden dürfen. Diese Reihenfolgen können explizit modelliert oder implizit im Quelltext der Bausteine vorhanden sein. Ein Beispiel für eine implizit dokumentierte Reihenfolge ist in der Onlineumfrage zu finden. Die Reihenfolge ist in Abbildung 14 als Sequenzdiagramm nachträglich modelliert worden. Innerhalb des Systems ist die Klasse `Result` dafür verantwortlich, die Antworten der Teilnehmer in eine externe Textdatei zu schreiben. Dies kann jedoch erst geschehen, wenn der Client (`QuestionnaireServlet`), der den Service des Schreibens ausführen möchte, in einem

ersten Schritt den Server (`Result`) initialisiert. Hierzu muss er den Speicherpfad setzen und anschließend die Ergebnisdatei öffnen. Erst dann ist es ihm möglich, die Ergebnisse in die angegebene Datei zu schreiben. Nachdem er die Ergebnisse in die Datei geschrieben hat, muss er die geöffnete Datei wieder schließen. Bei einer nachträglichen Modifikation des Quelltextes, die diese Reihenfolge ändert, können Probleme auftreten. Eine definierte Reihenfolge kann die Testnotwendigkeit von Abhängigkeiten erhöhen, da die Nichteinhaltung der Reihenfolge zu Fehlern führen kann.

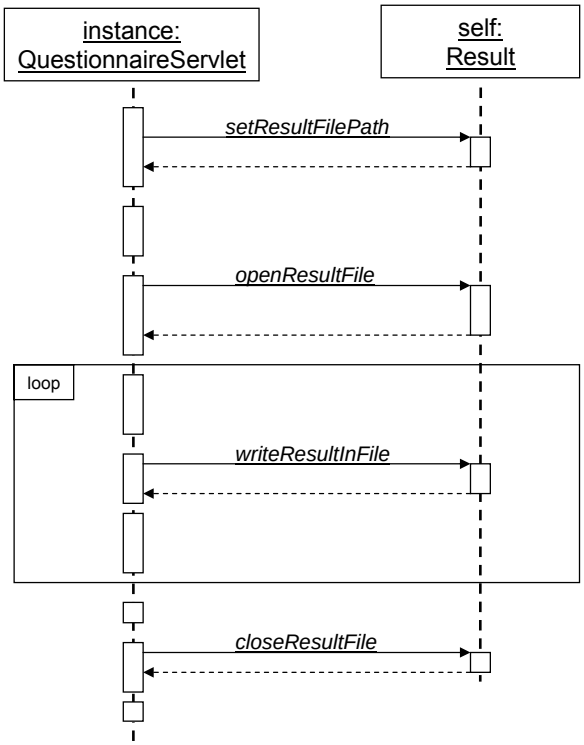


Abbildung 14: Protokollbeispiel aus der Onlineumfrage

Zeitbedingungen beschreiben Bedingungen an das Zeitverhalten der Abhängigkeit. Ein Server kann sich innerhalb des Vertrages dazu verpflichten, eine Funktionalität innerhalb einer definierten Zeit auszuführen (z.B. in 15 ms). Die Verpflichtung kann aber auch dem Client auferlegt werden. Der Client kann sich verpflichten, notwendige Daten, die der Server für die aufgerufenen Services benötigt, innerhalb einer definierten Zeitspanne an den Server zu senden. Die vereinbarten Zeitbedingungen müssen im Integrationstestprozess überprüft werden, da die Nichteinhaltung zum Vertragsbruch und zu schweren Fehlern in den beteiligten Bausteinen führt.

Nichtfunktionale Anforderungen beschreiben nichtfunktionale Kriterien, die an eine Abhängigkeit gestellt und in einem Vertrag definiert werden. Ein Beispiel für ein Qualitätsmerkmal ist die Genauigkeit der auszutauschenden Daten. Server und auch Client verpflichten sich, die Daten mit einer vordefinierten Genauigkeit zu liefern, beispielsweise auf die siebente Kommastelle genau. An diese Vereinbarung müssen sich Client und Server halten. Eine Nichteinhaltung kann zu Rechnungsfehlern führen.
Die Existenz der Eigenschaft Kommunikationsvertrag einschließlich ihrer Untereigenschaften erhöht die Testnotwendigkeit einer Abhängigkeit und deren Simulationsaufwand, da zusätzliche Monitore notwendig sind, um beispielsweise die Antwortzeit aufzuzeichnen.

Parameter

Die Eigenschaft Parameter klassifiziert eine Abhängigkeit hinsichtlich ihrer Input- und Outputparameter. Mit einem Serviceaufruf kann der Client Parameter, so genannte Inputparameter, an den Server übergeben. Welche Parameter während eines Aufrufs übergeben werden müssen, wird in der Schnittstellenbeschreibung des Services definiert. Hierbei kann es passieren, dass der Client die falschen Inputparameter übergibt oder der Server die Inputparameter falsch interpretiert, beispielsweise wird die übergebene Zahl als „Dollar“ anstatt als „Euro“ interpretiert [SBM+06]. Hierbei kann die Anzahl der **Inputparameter pro Serviceaufruf** die Wahrscheinlichkeit eines der oben genannten Fehler erhöhen, wodurch die Testnotwendigkeit der Abhängigkeit erhöht wird. Zusätzlich steigt mit zunehmender Anzahl der Inputparameter auch der Simulationsaufwand, da das Verhalten des Services von den Inputparametern abhängig ist.

Die Existenz von **Inputparametern gleichen Typs** während eines Serviceaufrufs kann zu Fehlern führen. In Quelltext 2 werden vier Parameter vom Typ String übergeben. Ein Vertauschen der Parameter [SPB+06] wäre denkbar und fiele dem Entwickler nicht auf, da der Compiler keine syntaktischen Fehler feststellen würde. Das Erstellen der Tabelle würde in vielen Fällen zu einer falschen Darstellung führen. Diese Fehlerquelle kann die Testnotwendigkeit der Abhängigkeit erhöhen.

Nicht nur der gleiche Typ, sondern auch der Typ der Inputparameter allgemein kann von Interesse sein. Bei der Simulation einer Abhängigkeit müssen gegebenenfalls auch **komplexe Inputparameter** simuliert werden. Dieser Umstand erhöht den Simulationsaufwand einer Abhängigkeit. Hinweise auf einen Einfluss auf die Testnotwendigkeit konnten nicht gefunden werden.

```
beginTable(String width, String border, String spacing, String padding)[1]
```

Quelltext 2: Beispiel einer Methode mit mehreren Inputparametern gleichen Typs

Die **Richtungen der Inputparameter** (vgl. [RHQ+05]) werden im Laufe der Entwurfsphase definiert und geben an, ob die Inputparameter während der Serviceaufrufe nur gelesen oder auch modifiziert werden. Die Übereinstimmung der Serviceimplementierung und der Servicespezifikation muss überprüft werden. Eine Nichteinhaltung der Spezifikation kann zu einem Fehler führen. Ein Beispiel: Ein Service modifiziert einen Parameter, obwohl er es nicht „dürfte. Die führt dazu, dass mit einem modifizierten Wert weitergearbeitet wird, obwohl dieser unverändert hätte bleiben müssen. Dies kann im weiteren Verlauf zu schweren Fehlern führen. Die Überprüfung, ob Parameter verändert oder nur gelesen werden, führt dazu, dass zusätzliche Überprüfungsmechanismen (Beobachtungspunkte, Monitore) realisiert werden müssen. Auf den Simulationsaufwand selbst hat das jedoch keinen Einfluss.

Wie Inputparameter können auch Outputparameter falsch interpretiert werden [SBM+06]. Dabei steigt die Gefahr einer Fehlerinterpretation mit der **Komplexität der Outputparameter**. Ein komplexer Datentyp beinhaltet Zustände, die über einfache und komplexe Datentypen realisiert werden können. Diese enthaltenen Datentypen bieten mehr Möglichkeit zur Fehlerinterpretation als ein einziger Datentyp als Rückgabewert. Somit erhöht sich die Testnotwendigkeit einer Abhängigkeit wenn es sich um komplexe Outputparameter in den Serviceaufrufen handelt. Darüber hinaus erhöht sich durch die Existenz von komplexen Outputparametern auch der Simulationsaufwand. Wenn eine

[1] beginTable definiert eine Tabelle auf der Oberfläche (Web-Seite) mit allen notwendigen Informationen.

Instanz eines Bausteins (Datentyps) als Rückgabewert gefordert ist, dieser aber noch nicht im aktuellen Integrationstestschritt integriert wurde, muss er simuliert werden.
Die Existenz von Input- und Outputparametern erhöht in vielen Fällen sowohl die Testnotwendigkeit als auch den Simulationsaufwand einer Abhängigkeit.

Übergreifende Zustandsänderungen
Die übergreifende Zustandsänderung beschreibt die Folgen einer Client/Server Abhängigkeit zwischen zwei Bausteinen und bezieht sich auf das betrachtete Softwaresystem. Solche Folgen können das Erzeugen von Instanzen, die Modifikation an existierenden Instanzen und Bausteinen sowie das Löschen von Instanzen sein.
Eine **Instanzmodifikation** beschreibt, ob durch die Kommunikation Änderungen an bereits im System existierenden Instanzen vorgenommen werden. Diese Änderung kann an den übergebenen Inputparametern sein, aber auch an Attributen innerhalb des Servers. Je mehr Instanzen von der Modifikation betroffen sind, desto höher ist die Wahrscheinlichkeit, dass eine Modifikation fehlschlägt. So könnten die Modifikationen an der falschen Instanz durchgeführt werden oder die Instanzen befinden sich nach der Modifikation in einem falschen Zustand.
Das **Erzeugen von Instanzen** und das **Löschen von Instanzen** stellen besondere Arten von Zustandsänderungen dar. Im ersten Fall werden Instanzen von bestimmten Bausteinen erzeugt. Dabei können die Instanzen vom falschen Typ instanziiert worden sein oder sie befinden sich nach der Instanziierung in einem falschen Zustand. Das Löschen einer Instanz beschreibt den entgegengesetzten Vorgang. Eine im System existierende Instanz wird gelöscht. Dieser Löschvorgang kann nicht vollständig abgeschlossen werden, so dass die Instanz dennoch vorhanden ist und zur Belegung von Speicher führt (vgl. Programmiersprache Java, in der der Garbage Collector [MSH03] den Speicher eines Objekts erst freigeben kann, wenn auf das zu löschende Objekt keine Referenz mehr besteht), oder der belegte Speicher wurde nicht freigegeben (vgl. Programmiersprache C++, in der der Speicher explizit vom Programmierer bei Löschung einer Instanz freigegeben werden muss). Weiterhin ist es denkbar, dass die falsche Instanz gelöscht wurde.
Die Eigenschaft der übergreifenden Zustandsänderung ist in komplexen Systemen schwer zu identifizieren. Um alle Zustandsänderungen eines Serviceaufrufs identifizieren zu können, muss das System vollständig dokumentiert sein, d.h. alle Abhängigkeiten zwischen Bausteinen müssen spezifiziert sein. Die Informationen können in Modellen der Entwurfsphase (Zustandsautomaten, Sequenzdiagramme, Aktivitätsdiagramme) oder im Quelltext gefunden werden.

4.1.1.2 Vererbung

„*Unter Vererbung versteht man die Möglichkeit, ein neues Objekt von einem vorhandenen Objekt abzuleiten, wobei das neue Objekt alle Merkmale und Fähigkeiten des alten besitzt.*“ ([MSH03], Seite 62). In einem solchen Fall stehen die beiden Objekte (Bausteine) in einer Vererbungsbeziehung. Der neue Baustein (Unterklasse) erbt hierbei alle Eigenschaften, d.h. alle Attribute und Services sowie die Abhängigkeiten des vorhandenen Bausteins (Oberklasse). Diese Art der Beziehung ermöglicht es, eine Instanz einer Oberklasse durch eine Instanz der Unterklasse im System zu ersetzen. Diese Eigenschaft führt dazu, dass eine Instanz einer Unterklasse an jeder Stelle im Programmquelltext eingesetzt werden kann, wenn eine Instanz der Oberklasse erwartet wird. Informationen über eine Vererbungsbeziehung und ihre Eigenschaften lassen sich leicht im Quelltext der Bausteine oder im UML-Klassendiagramm finden. Im Klassendiagramm für den Onlinefragebogen in Abbildung 4 in Kapitel 2.3 ist eine Vererbungsbeziehung zwischen der Klasse `QuestionnaireServlet` und der Klasse `HttpServlet` eingezeichnet, wobei erstere die Unterklasse und letztere die Oberklasse darstellt.

Aufgrund der Tatsache, dass eine Unterklasse alle Eigenschaften, insbesondere auch alle Abhängigkeiten, erbt, ergibt sich eine erhöhte Testnotwendigkeit. Es müssen nicht nur alle Abhängigkeiten zwischen der Oberklasse und ihren abhängigen Bausteinen überprüft werden, sondern auch zwischen allen Unterklassen und den abhängigen Bausteinen der Oberklasse. Hierbei können weitere Eigenschaften einer Vererbungsbeziehung diese Testnotwendigkeit näher spezifizieren.

Modifikation
Die Eigenschaft der Modifikation beschreibt das Ausmaß der Veränderung der Eigenschaften der Oberklasse durch die Unterklasse. Dies kann durch das Hinzufügen von neuen Services und Attributen sowie durch das Überschreiben oder Überladen von existierenden Services und Attributen geschehen. Beim Überschreiben eines Services implementiert die Unterklasse das Verhalten des Services der Oberklasse neu und ändert damit eventuell auch das erwartete Verhalten. Dies bedeutet, dass das Verhalten der Unterklasse nicht mehr zwangsläufig konsistent zum Verhalten der Oberklasse ist. Da jedoch jede Instanz einer Oberklasse durch eine Instanz einer Unterklasse substituiert werden kann [Or98], wird das erwartete Verhalten der Oberklasse nicht garantiert.
Für den Integrationstest ist es von Interesse, wie viele Modifikationen durch die Unterklasse vorgenommen wurden. Das Hinzufügen von **neuen Attributen** führt dazu, dass die Instanzen der Klasse neue Zustände einnehmen können, welche das Verhalten der Klassen verändern. **Überschriebene Services** und **überschriebene Attribute** führen zu einer Veränderung des Verhaltens der Klasse, welches nicht unbedingt konsistent zum Verhalten der Oberklasse ist. **Neue Services** sind in der Lage vorhandene oder auch neue Attribute zu manipulieren, wodurch Zustände und Zustandsübergänge der Oberklasse nicht konsistent zur Unterklasse sein müssen.
Die benötigten Informationen sind in den Entwurfsmodellen und direkt aus dem Quelltext ersichtlich. Klassendiagramme beispielsweise zeigen auf, welche Klasse von welcher erbt und in welcher Klasse Services und Attribute neu hinzugefügt oder überschrieben worden sind. Alle vier Untereigenschaften der Vererbung haben Einfluss auf die Testnotwendigkeit der Abhängigkeit. Je größer die Modifikation ist, desto größer ist die Wahrscheinlichkeit, dass das Verhalten nicht mehr konsistent zum Verhalten der Oberklasse ist.

Vererbungstiefe
Die Vererbungstiefe beschreibt den „Abstand“ zwischen der betrachteten Unterklasse und der Oberklasse, d.h. wie viele Klassen sich in der Vererbungshierarchie der betrachteten Klassen befindet. Alle Klassen dazwischen sind potenzielle Modifikatoren[1] der betrachteten Oberklasse. Je größer die Vererbungstiefe ist, desto stärker wurde die Oberklasse eventuell bereits modifiziert. Durch diese Modifikationen erhöht sich die Testnotwendigkeit dieser Abhängigkeiten. Gleichzeitig erhöht sich mit steigender Anzahl der Modifikationen der Simulationsaufwand. Je mehr Oberklassen die Unterklasse hat, desto mehr Services und Attribute müssen für den Integrationstest simuliert werden.
In der Onlineumfrage beträgt die Vererbungstiefe zwischen der Klasse `QuestionnaireServlet` und der Klasse `HTTPServlet` „1“. Die Vererbungstiefe zwischen `QuestionnaireServlet` und der Klasse `Object` beträgt „4“. Zwischen diesen Klassen befinden sich die Klassen `GenericServlet` und `Servlet`[2].

[1] Ein Modifikator ist eine Klasse, die die Eigenschaften der Oberklasse durch neue Attribute, Services oder das Überschreiben modifiziert
[2] Nicht im Klassendiagramm dargestellt, aber nachzulesen in der Java API [JA08]

4.1.1.3 Indirekte Abhängigkeit

Indirekte Abhängigkeiten entstehen, wenn zwei Bausteine voneinander abhängig sind, obwohl sie nicht direkt miteinander kommunizieren oder in einer Vererbungsbeziehung stehen. Die Abhängigkeit entsteht durch die gemeinsame Verwendung interner und externer Ressourcen, wodurch sich die beteiligten Bausteine gegenseitig beeinflussen. Die betrachteten Ressourcen sind in dieser Abhängigkeit in keinem der beiden Bausteine zu finden. In einer Abhängigkeit über Ressourcen ist es nicht möglich, einen eindeutigen abhängigen und unabhängigen Baustein zu definieren. In diesem Fall werden zwei Abhängigkeiten definiert: Für jeden Baustein eine eingehende und eine ausgehende Abhängigkeit.
Es sind verschiedene Arten von Ressourcen möglich: *Variablen, externe Datenquellen, Hardware* und *externe Services*.
Variablen beschreiben die gemeinsame Nutzung einer oder mehrerer (globaler) Variablen durch zwei Bausteine. Durch diese gemeinsamen Zugriffe kann es zu einer Beeinflussung der Funktionalität in einem der Bausteine oder in beiden Bausteinen kommen, sofern die Funktionalität von den Zuständen der globalen Variablen beeinflusst wird. Typische Fehler können durch falsche Zustände der globalen Variablen, durch Wettlaufbedingungen oder durch Deadlocks entstehen. Die Art der Zugriffe auf die Variablen spielt dabei eine entscheidende Rolle. Es kann zwischen gemeinsam gelesenen und gemeinsam modifizierten Variablen unterschieden werden. Sollten zwei Bausteine nur lesend auf die Variablen zugreifen, so ist der Einfluss, den beide Bausteine aufeinander ausüben, sehr gering. In den Fällen, in denen einer oder beide Bausteine die globalen Variablen modifizieren, kann es zu oben genannten Fehlern kommen.
Die gemeinsame Nutzung von **externe Datenquellen** kann zu ähnlichen Problemen führen wie die gemeinsame Nutzung von globalen Variablen. Wie auch im Fall der globalen Variablen können die gemeinsam gelesenen oder die gemeinsam modifizierten externen Datenquellen von Interesse sein. Im Beispiel der Onlineumfrage haben wir durch die zwei externen Dateien `Result-File` und `XML-File` zwar zwei Dateien, auf die einige Bausteine des Systems zugreifen, jedoch greift immer nur ein Baustein auf diese Dateien zu. Daher gibt es im Beispielsystem keine gemeinsame Nutzung von externen Datenquellen und somit keine Abhängigkeit über eine externe Datei.
Eine Abhängigkeit über die **Hardware** beschreibt die gemeinsame Nutzung von Systemressourcen durch zwei Bausteine, was eventuell zu Problemen und Fehlerverhalten führen kann. Sollten sich die Bausteine etwa auf dem gleichen Hardwaresystem befinden und somit gemeinsam die Rechnerkapazitäten nutzen, ist es möglich, dass die Bausteine nicht mehr in der Lage sind, ihre geforderte Funktionalität aufgrund knapper Rechnerressourcen zu erbringen. Insbesondere in Kombination mit nichtfunktionalen Anforderungen kann es zu einem gesonderten Problem führen, wenn ein Server nicht mehr in der Lage ist, die Antwort in geforderter Zeit auszuführen. Es lassen sich eine Vielzahl von Systemressourcen identifizieren, die auf die Funktionalität von Bausteinen Einfluss haben können, beispielsweise der Arbeitsspeicher, die Prozessorzeit, die Prozessoranzahl, der Festplattenspeicher, die Netzwerkbandbreite, die Batterieleistung.
Speziell im Zusammenhang mit mobilen Anwendungen [MMS+06] kann die Betrachtung der Abhängigkeit durch die gemeinsame Nutzung von Systemressourcen von Bedeutung sein, da mobile Endgeräte nur stark eingeschränkte Systemressourcen besitzen.
Externe Services beschreiben ergänzende Softwaresysteme, die Funktionalitäten anbieten, die von zwei Bausteinen gemeinsam genutzt werden. Zu dieser Kategorie gehören Services von Betriebssystemen, aber auch eingesetzte Middleware. Hierbei können Fehler in der eigentlichen Anwendung entstehen, weil die externen Systeme fehlerhaft sind oder falsch verwendet wurden

4.1.2. Kategorie der Entwicklungseigenschaften

Die Kategorie der Entwicklungseigenschaften (vgl. Abbildung 15 a)) stellt im Vergleich zu den Laufzeiteigenschaften eine sehr kleine Kategorie dar. Sie liefert jedoch ergänzende Informationen über eine bereits vorliegende Abhängigkeit mit Laufzeiteigenschaften. Die Entwicklungseigenschaften beschreiben Eigenschaften von Bausteinen, die sich auf die Abhängigkeit zwischen diesen Bausteinen auswirken kann. Durch Offshorings (vgl. [CR05]) oder durch große Softwareentwicklungsprojekte entstehen inhomogene Entwicklungsteams, die **geographisch verteilt** und eventuell auch mit unterschiedlichen Technologien (z.B. **Programmiersprachen**) arbeiten.

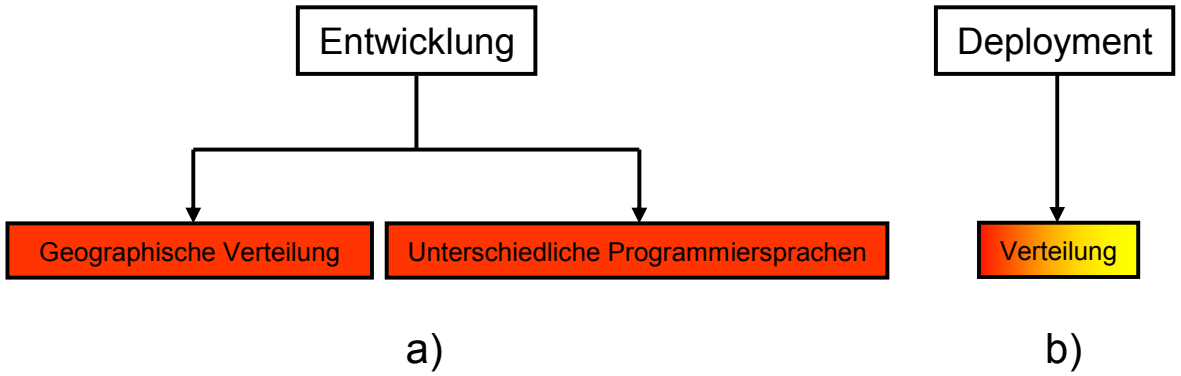


Abbildung 15: Eigenschaftsklassen „Entwicklung“ und „Deployment“

Geographische Verteilung

Die geographische Verteilung einer Entwicklung besagt, dass die Bausteine, die an einer Abhängigkeit beteiligt sind, unabhängig voneinander entwickelt worden sind. Dabei kann es sich um eine räumliche oder um eine zeitliche Unabhängigkeit handeln.

Eine räumliche Unabhängigkeit liegt vor, wenn die beteiligten Bausteine an geographisch unterschiedlichen Orten entwickelt wurden. Arbeiten Entwicklerteams an verschiedenen Orten, kann es durch Missverständnisse zu Fehlern kommen. Die Verwendung von unterschiedlichen Maßzahlen z.B. Zentimeter, Inch oder Zoll kann durch diese räumliche Verteilung zu Fehlern in Bausteinen und in der Abhängigkeit zwischen diesen führen.

Eine zeitliche Unabhängigkeit beschreibt, dass einer der beteiligten Bausteine Monate oder auch Jahre vor dem anderen Baustein erstellt worden ist. Durch diese zeitliche Trennung kann Wissen über den „älteren“ Baustein verloren gehen (z.B. durch defekte Festplatten, neue Werkzeuge, die alte Formate nicht mehr lesen können …). Dadurch kann es passieren, dass wichtige Informationen über diesen Baustein nicht mehr vorhanden sind.

Nicht selten sind auch kulturelle und sprachliche Unterschiede innerhalb der Entwicklerwelten Ursache für Fehler. Kommunikationsschwierigkeiten durch missverstandene und mehrdeutige Begriffe sind ein Beispiel hierfür. Die Informationen über die Unterschiede in der Entwicklung der einzelnen Bausteine können eventuell im Projektplan eingesehen werden. Dort ist eine Zuteilung von zu entwickelnden Bausteinen und den verantwortlichen Entwicklerteams zu erkennen.

Programmiersprache

In Softwaresystemen kann es vorkommen, dass Bausteine miteinander kommunizieren, die in unterschiedlichen Programmiersprachen realisiert sind. Diese Programmiersprachen können sich durch die Umsetzung bestimmter Programmierkonzepte voneinander unterscheiden. Beispielsweise wird die Behandlung von Arrays in verschiedenen Programmiersprachen unterschiedlich gehandhabt. Ein Beispiel: die Programmiersprache

Java beginnt die Nummerierungen der Elemente im Array mit der Zahl „0“. In der Programmiersprache Fortran beginnt die Zählung mit einer „1“. [Co71] So kann das Übergeben eines Arrays sowie eines Indexes auf ein Element innerhalb des Arrays von einem Baustein zu einem anderen dazu führen, dass das falsche Element ausgelesen wird. Die Konsequenz ist ein falsch ermittelter Wert, der in spätere Berechnungen einfließen wird.

4.1.3. Kategorie - Deploymenteigenschaften

Diese Kategorie der Eigenschaften (vgl. Abbildung 15 b)) bezieht die Hardwareumgebung ein, in der die betrachteten Bausteine installiert sind und ausgeführt werden. Hierbei kann unterschieden werden, ob die einzelnen Bausteine *verteilt* installiert sind und ausgeführt werden oder ob sie sich auf dem gleichen Hardwareknoten befinden. Im Fall der **Verteilung** kann die Testnotwendigkeit erhöht werden, da der Austausch von Informationen über zusätzliche Infrastruktur, beispielsweise ein Netzwerk, erfolgen muss. „Two components that unconsciously interact via the infrastructure are potential sources of failures“ ([Ma03] Seite 7). Weiterhin besitzen die eingesetzten Hardwareknoten ebenfalls eine bestimmte Wahrscheinlichkeit auszufallen oder fehlerhaft zu sein. Durch den Einsatz mehrerer Hardwareknoten wird somit die Fehleranfälligkeit der Software und der darin enthaltenen Abhängigkeiten zusätzlich erhöht. Gleichzeitig erhöht sich durch die Verteilung auch der Simulationsaufwand. Zusätzliche Hardware muss eingesetzt oder simuliert werden, um die Verteilung realistisch abzubilden

4.2. State of the Art – Klassifikation von Abhängigkeiten

Ein Katalog von Abhängigkeitseigenschaften, wie er auf den vorangegangenen Seiten vorgestellt wurde, kann so in der Literatur nicht gefunden werden. Vereinzelt werden in verschiedenen Literaturquellen jedoch Hinweise auf mögliche Abhängigkeitseigenschaften gegeben. Einen Überblick welche Eigenschaften aus welchen Quellen entnommen werden konnten, ist in Tabelle 2 dargestellt. Nachfolgend werden die Arbeiten aus der Literatur vorgestellt, aus den Eigenschaften von Abhängigkeiten für den Katalog entnommen werden konnten.

Tabelle 2: Literaturquellen der Abhängigkeitseigenschaften

Abhängigkeitseigenschaft	Literaturquellen
Laufzeit - Client/Server	[RHQ+05], [Or98], [UML09]
Serviceaufrufe	[Ov94], [PC89], [VR02]
Attributzugriffe	[Ov94], [PC89], [VR02]
Kommunikationsvertrag	[Ma03], [SPB+06], [VR02]
Parameter	[Ov94], [RHQ+05], [SPB+06], [WC03]
Asynchronität	[Ov94]
Parallelität	[Ov94]
Zustandsänderungen	[BTS+03]
Laufzeit - Indirekte Abhängigkeiten	[Ma03],
Gemeinsame Ressourcen	[Bi96]
Laufzeit - Vererbung	[RHQ+05], [UML09]
Vererbungstiefe	[Or98]
Modifikation	[HM92], [Or98],
Entwicklung	-
Deployment	[Ma03], [WPC01]

Eine weit verbreitete Klassifikation von Abhängigkeiten bietet die Unified Modeling Language (UML [UML09], [RHQ+05]) in der Version 2.0. Sie unterscheidet Assoziation, Aggregation,

Komposition und Vererbung. Diese Einteilung wurde in vielen Literaturquellen übernommen (z.B. [HM92], [Bi96], [Or98], [BLW01] oder [BTS+03]). Die Klassifikation kann bereits in sehr frühen Phasen des Softwareentwicklungsprozess eingesetzt werden, da sie die Abhängigkeit zwischen zwei Bausteinen nur sehr abstrakt darstellt.
Eine Klassifikationsmöglichkeit für (Programm-) Abhängigkeiten lässt sich in [PC89] finden. Die Autoren unterscheiden zwischen Steuerungs- und Datenabhängigkeit. Eine Steuerungsabhängigkeit fasst alle Eigenschaften der Kontrollstruktur, d.h. des dynamischen Ablaufs, zusammen. Eine Datenabhängigkeit umfasst die Verwendung von Variablen. Diese Abhängigkeitsarten wurden für (starke und schwache) syntaktische und semantische Abhängigkeiten verfeinert. Diese Abhängigkeitsarten beschreiben jedoch nur Abhängigkeiten zwischen Anweisungen[1] und lassen sich nur auf Anweisungen innerhalb eines Quelltextes anwenden. Auf Abhängigkeiten zwischen abstrakteren Bausteinen, z.B. Klassencluster, Teilsysteme oder Schichten lassen sie sich nicht anwenden.
Vieira und Richardson beschreiben in [VR02], welche Informationen in den Abhängigkeitsspezifikationen für komponentenbasierte Systeme enthalten sein müssen. Diese Spezifikationen umfassen die angebotenen und notwendigen Services der beteiligten Komponenten, die Beschreibung der einzuhaltenden Protokolle und die Komponentenbenutzung, d.h. welche Komponenten als Server benötigt werden. Die Spezifikation der Abhängigkeiten wird zusammen mit der Spezifikation der Komponenten beschrieben. Dabei werden die Abhängigkeiten mit Hilfe von regulären Ausdrücken spezifiziert.
In [WC03] definieren Wheeldon und Counsell fünf verschieden Typen von Klassen-Kopplungen in einem objektorientierten Softwaresystem. Neben der Vererbung und Aggregation, existieren die Interface-Beziehung (eine Klasse realisiert ein Interface), Parametertypen (eine Klasse ist ein Inputparameter für eine Methode einer anderen Klasse) und Rückgabetypen (eine Klasse ist Outputparameter einer Methode einer anderen Klasse).
Orso unterteilt in seiner Doktorarbeit [Or98] Abhängigkeiten zwischen Klassen in Client/Server Abhängigkeiten (diese fassen Assoziationen, Aggregationen und Kompositionen zusammen) und hierarchische Abhängigkeiten (Vererbung). Bei der Vererbung unterscheidet er in Subclassing und Subtyping. Subclassing repräsentiert dabei die uns bekannte Vererbung und kann als einfache Vererbung oder Mehrfachvererbung auftreten. Neben dieser Abhängigkeit kann die erbende Klasse neue Eigenschaften (im Vergleich zur Oberklasse) definieren und/oder existierende Eigenschaften neu definieren (überschreiben). Beim Überschreiben kann zusätzlich zwischen invariantem, kovariantem und kontravariantem Überschreiben unterschieden werden. Das invariante Überschreiben fordert, dass überschriebene Eigenschaften die gleiche Signatur wie die Oberklasse aufweisen. Beim kovarianten Überschreiben können alle Datentypen auch durch ihre Unterklassen ersetzt werden. Beim kontravarianten Überschreiben können, komplementär zum kovarianten Überschreiben, alle Datentypen durch ihre Oberklassen ersetzt werden. Subclassing ist eine syntaktische Beziehung zwischen zwei Klassen und wird nur zum Zweck der Wiederverwendung eingesetzt (eine Oberklasse stellt Funktionalität zur Verfügung, die in allen Unterklassen zur Verfügung steht). Subtyping hingegen repräsentiert eine semantische Beziehung zwischen zwei Klassen. Im Fall des Subtyping könnte anstelle der Oberklasse auch die Unterklasse verwendet werden. Hierbei kann noch einmal zwischen der schwachen und starken Form unterschieden werden. In den meisten objektorientierten Programmiersprachen ist die schwache Form des Subtyping realisiert. Dieses Konzept führt zum Phänomen des Polymorphismus. „*Polymorphism refers to the possibility for an entity to refer at run-time to objects of several types.*“ ([Or98] Seite 19). Hierbei handelt es sich um die

[1] Die Abhängigkeiten zwischen den Anweisungen können im Kontrollflussgraphen (vgl. [PC89]) dargestellt werden.

Möglichkeit, dass eine Klasse zur Laufzeit von unterschiedlichen Typen abhängen kann. D.h. wenn eine Klasse A eine Klasse B verwendet, könnte sie implizit alle Unterklassen von Klasse B verwenden. Dieser Fakt stellt für den Integrationstest eine besondere Herausforderung dar. Es müssen neben den Abhängigkeiten zwischen Baustein A und Baustein B auch alle (möglichen) Abhängigkeiten zwischen Baustein A und allen Unterklassen von Baustein B getestet werden. In seiner Arbeit unterscheidet Orso zwischen acht verschiedenen Arten des Polymorphismus, die sich hierarchisch anordnen lassen. Durch Polymorphismus können Abhängigkeiten in dynamische und statische Abhängigkeiten unterschieden werden [CL04].
Die in [Or98] beschriebenen Eigenschaften der Vererbung können durch die von Harrold und McGregor in [HM92] genannten Eigenschaften erweitert werden. Sie unterteilen das Hinzufügen von neuer Funktionalität (im Vergleich zur Oberklasse) und das Modifizieren von vorhandener Funktonalität. Neue Funktionalität entsteht durch das Hinzufügen von neuen (virtuellen) Attributen und Operationen. Geerbte (virtuelle) Operationen, d.h. das Modifizieren vorhandener Funktionalität, werden gesondert betrachtet. Harrold und McGregor gehen in ihrer Arbeit auch auf Schnittstellenfehler ein, aus denen sich weitere Eigenschaften für Abhängigkeiten ableiten lassen. Hierzu gehören Fehler im Eingabe/Ausgabe-Format der *Parameter*, die falsche *Reihenfolge* von Unterroutineaufrufen, Missverständnisse der Eingabe/Ausgabe ([Ov94]). Diese Fehler werden in [SPB+06] als „input interpretation misfit“, „output interpretation misfit“ und als „control state misfit“ bezeichnet. Ergänzt wird die Liste durch „input order misfit“, d.h. ein Fehler in der Reihenfolge der Inputparameter und „quality misfit“, d.h. es werden vorgegebene Qualitätskriterien nicht eingehalten. Diese Fehler lassen darauf schließen, dass für den Integrationstest auch Informationen über die Input- und die Outputparameter eine wichtige Rolle spielen. Leonardo Mariani geht in [Ma03] näher auf die Qualitätskriterien ein. Er unterscheidet hier Performanz (z.B. Zeit und Speicher), Quality of Service (z.B. Zuverlässigkeit), Entwicklungsprozess und Kosten. Weiterhin schreibt er: „*Two components that unconsciously interact via the infrastructure are a potential source of failure*“ und „*Two components may be in conflict because […] they could interact through services of the operating system […] by accessing the same file, or by using the same service, or by sharing the same disc.*“ ([Ma03], Seite 7). Diese Darstellung zeigt eine weitere Abhängigkeitseigenschaft, die für den Integrationstest von Bedeutung ist: Die Art, wie die Interaktion zwischen zwei Bausteinen stattfindet, beispielsweise mit Hilfe einer Middleware, dem Betriebssystem, von Dateien, von anderen Services oder der Festplatte. Gleichzeitig ist es von Interesse, wie die beteiligten Bausteine auf die Hardware verteilt sind. So können sich zwei Bausteine gegenseitig beeinflussen, wenn sie auf dem gleichem Hardwareknoten installiert sind (*„simply because they share the CPU“*, [Ma03], Seite 6).
In [BTS+03] unterscheiden die Autoren drei Arten der Interaktion: Klasseninteraktion, Selbst-Interaktion, Objektinteraktion. Ersteres betrachtet die Interaktion auf Klassenebene, d.h. welche Operation ruft Operationen in anderen Klassen auf. Letzteres erweitert diese Betrachtung auf die Objektebene, indem es konkrete Objektinstanzen berücksichtigt. Die Selbstinteraktion beschreibt die Interaktion einer Klasse mit sich selbst, d.h. das Aufrufen von Operationen an der eigenen Klasse. Für diese Interaktionen schlagen Baudry et al. die Stereotypen <<create>>, <<use>>, <<use_consult>> und <<use_def>> vor. In einer create-Interaktion werden nur Operationen aufgerufen, die eine Instanz einer Klasse erzeugen. In einer use-Interaktion werden alle Operationen außer den create-Operationen verwendet. Diese Interaktion wird durch <<use_consult>> und <<use_def>> verfeinert. Während in der ersten Interaktion der Zustand des aufgerufenen Servers unverändert bleibt, wird in der zweiten der innere Zustand des aufgerufenen Servers verändert. Die Betrachtung der Zustandsänderung kann auch auf die Inputparameter von Serviceaufrufen erweitert werden. Die UML 2.0 ermöglicht es, in Klassendiagrammen für jeden Inputparameter einer Operation eine Richtung zu definieren: in, out, inout [RHQ+05]. Im ersten Fall wird der Parameter nur

gelesen und nicht verändert. In zweiten Fall wird der Parameter verändert, aber nicht gelesen und im letzten Fall wird der Parameter gelesen und verändert.
Abschließend soll noch Robert Binder erwähnt werden, der in [Bi96] eine umfassende Literaturrecherche zum Thema „Testen von objektorientierten Programmen“ erstellt hat. In [Bi96] können Abhängigkeiten zwischen zwei Klassen durch die gleichzeitige Verwendung der gleichen globalen Daten existieren. Zwischen beiden Klassen gibt es keine Vererbungs- oder Aufrufbeziehung, dennoch hängt die Funktionalität der einen Klasse über die globale Variable von der Funktionalität der anderen Klasse ab.

Für die Klassifikation von Abhängigkeiten lassen sich in der Literatur keine geeigneten Ansätze finden, die für die Testfokusauswahl und die Integrationsreihenfolgeermittlung im Integrationstestprozess geeignet erscheinen. Die oben vorgestellten Literaturquellen liefern viele Hinweise über mögliche Abhängigkeitseigenschaften, die für den Integrationstestprozess Auskunft über die Testnotwendigkeit und den Simulationsaufwand einer Abhängigkeit geben können. Aus diesem Grund wurden die verschiedenen Hinweise aus der Literatur in dem in Kapitel 4.1 vorgestellten Katalog zusammengetragen. Dieser Katalog erlaubt es unter anderem die Entscheidungen der Testfokusauswahl und der Integrationsreihenfolgermittlung im den Integrationstestprozess zu unterstützen.

4.3. Einsatz der Klassifikation von Abhängigkeiten

Die Anwendung der Klassifikation für den Integrationstestprozess in einem Softwareentwicklungsprojekt erfolgt in drei Schritten. Im ersten Schritt müssen die zu spezifizierenden Abhängigkeitseigenschaften festgelegt werden. Dies erfolgt zu Projektbeginn. Im zweiten Schritt, während der Softwareentwicklung, werden die Abhängigkeiten und ihre Eigenschaften spezifiziert. Im dritten Schritt werden die spezifizierten Informationen ausgenutzt. Diese drei Schritte werden nachfolgend kurz vorgestellt.

Schritt 1 - Abhängigkeitseigenschaften festlegen
Zu Projektbeginn, spätestens aber nach der Anforderungsphase müssen die zu spezifizierenden Eigenschaften für die Abhängigkeiten definiert werden. Hierzu werden aus dem vorliegenden Katalog die Eigenschaften ausgewählt, die während des Softwareentwicklungsprozesses dokumentiert werden. Idealerweise sind das alle möglichen Eigenschaften, die innerhalb der späteren Software existieren werden. Je mehr Informationen über die Eigenschaften zur Verfügung stehen, desto mehr Informationen stehen für die Planung und Durchführung des Integrationstestprozesses zur Verfügung. Auf der anderen Seite erhöht eine steigende Anzahl Eigenschaften den Dokumentationsaufwand der Entwickler. Daher muss ein Kompromiss zwischen Anzahl der zu dokumentierenden Eigenschaften und dem daraus resultierenden Dokumentationsaufwand gefunden werden.
Ein wichtiges Kriterium zur Erleichterung der Auswahl der zu spezifizierenden Abhängigkeitseigenschaften ist die Existenz von guter Werkzeugunterstützung. Viele der Eigenschaften lassen sich mit sehr wenig Aufwand durch Werkzeuge automatisch aus existierenden Dokumentationselementen extrahieren. Modelle aus der Architektur und dem Entwurf liefern Anhaltspunkte darüber, welche Bausteine von welchen abhängen. Kontrakte, dynamische Modelle (z.B. Sequenzdiagramme, Interaktionsdiagramme), Schnittstellenbeschreibungen aber auch statische Modelle (Klassendiagramm, Komponentendiagramme) liefern Informationen über Eigenschaften, beispielsweise über Services, Attribute oder Parameter. Der Projektplan liefert Daten über die verteilte Entwicklung von Bausteinen oder die eingesetzten Programmiersprachen. Verteilungsdiagramme enthalten Informationen über die Verteilung der abhängigen

Bausteine auf verschiedene Hardwareknoten. Und letztendlich kann auch der Quelltext eine Vielzahl an Informationen über die Abhängigkeiten offen legen. Mit Hilfe von statischen Analysewerkzeugen können viele Eigenschaften automatisch aus den Modellen und dem Quelltext extrahiert werden.
Der Zeitpunkt, wann welche Informationen spezifiziert werden, hat Einfluss auf die Planung und Vorbereitung des Integrationstestprozesses. Je früher die Informationen über die Abhängigkeiten und ihre Eigenschaften vorliegen, desto früher kann die Planung des Integrationstestprozesses begonnen werden.
Für die Entscheidung, welche Eigenschaften zu erheben sind, muss im Vorfeld überprüft werden, welche CASE[1]-Werkzeuge zur Verfügung stehen und im Projekt eingesetzt werden. Es ist zu prüfen, welche Informationen in den verschiedenen Werkzeugen dokumentiert werden können und ob sie mit möglichst wenig Aufwand extrahiert und dem Integrationstest zur Verfügung gestellt werden können.

Schritt 2 - Abhängigkeitseigenschaften spezifizieren
Nachdem festgelegt worden ist, was spezifiziert werden muss, können die Entwickler mit der Erstellung der Entwicklungsartefakte und des Softwaresystems beginnen. Sie erstellen das System auf die ihnen bekannte Art und Weise unter Verwendung der ihnen bekannten Modelle, Modellierungstechniken und Werkzeuge. An Stellen, an denen die Modellierungstechniken und die Werkzeugunterstützung das Dokumentieren von geforderten Eigenschaften nicht unterstützt, muss parallel in zusätzlichen Artefakten dokumentiert werden.
In dem Moment, in dem die ersten Entscheidungen zum Integrationstestprozess fallen, werden alle Informationen über Abhängigkeiten zusammengetragen und in kompakter Form den verantwortlichen Rollen des Integrationstestprozesses zur Verfügung gestellt.

Schritt 3 - Abhängigkeitseigenschaften nutzen
In erster Linie werden die Informationen über Abhängigkeiten und ihre Eigenschaften im Integrationstestprozess verwendet. Sie liefern einen wichtigen Beitrag zur Auswahl des Testfokus (vgl. Kapitel 5). Sie enthalten Hinweise auf die Testnotwendigkeit der Abhängigkeit. Sie spielen aber durch die enthaltenen Informationen über den Simulationsaufwand auch eine Rolle bei der Bestimmung einer optimalen Integrationsreihenfolge (vgl. Kapitel 6). Eine Abhängigkeit mit einem hohen Simulationsaufwand wird früh im Integrationstest integriert, um den Simulationsaufwand für diese zu vermeiden.
Aber nicht nur für den Integrationstest können die Eigenschaften einen wichtigen Beitrag leisten. Sie können auch dazu beitragen, die von Vieira und Richardson in [VR02] geforderte explizite Beschreibung von Abhängigkeiten in komponentenorientierten Systemen zu realisieren. Die in [VR02] vorgestellte Beschreibung der Pomsets zur Beschreibung einer Abhängigkeit könnten um die noch nicht enthaltenen Eigenschaften erweitert werden. Für den MORABIT[2]-Ansatz [SPB+06], der darauf abzielt, mobile Komponenten zur Laufzeit zu testen, können die Informationen über Eigenschaften dabei helfen, geeignete Teststrategien, d.h. wann und wie viel muss zur Laufzeit getestet werden, auszuwählen. Gleichzeitig kann sie dem Komponentenentwickler Hinweise darauf geben, was in einem Build-In Test getestet werden muss.
Eine Möglichkeit, Abhängigkeiten kompakt darzustellen, bieten die Design-Struktur-Matrizen [Br01], [SJS+05]. Die Matrizen beschreiben, welcher Baustein von welchem Baustein abhängt. Diese Darstellung kann um eine Dimension erweitert werden, die die Eigenschaften

[1] CASE … Computer Aided Software Engineering
[2] MORABIT … **Mo**bil **Re**ssouce **A**daptive **B**uilt-**I**n **T**ests

einer Abhängigkeit repräsentiert. Diese Abhängigkeit wird in eine Matrix eingetragen, deren Zeilen und Spalten durch die Bausteine des Softwaresystems bezeichnet werden. Eine Abhängigkeit zwischen zwei Bausteinen wird durch einen Punkt in der Matrix repräsentiert. Dieser Punkt könnte durch einen Vektor ersetzt werden, um die zusätzlichen Eigenschaften zu berücksichtigen. Zur besseren Visualisierung kann auch alternativ aus den Eigenschaften ein Zahlenwert ermittelt werden, der die Komplexität, die Testnotwendigkeit oder den Simulationsaufwand für jede Abhängigkeit repräsentiert. Durch eine Darstellung im dreidimensionalen Raum lässt sich sofort erkennen, welche Abhängigkeiten die höchste Komplexität haben.

5. Testfokusauswahl

Die Testfokusauswahl befasst sich mit der Entscheidung, welche Teile des Softwaresystems zu testen sind. Die Auswahl ist notwendig, da in der Regel jedes Softwareprojekt mit beschränkten Ressourcen für die Qualitätssicherung auskommen muss. Diese Ressourcen müssen auf die zu testenden Systemteile aufgeteilt werden. Es gibt zwei Strategien, um diese Aufteilung vorzunehmen. Die erste Strategie sieht eine gleichmäßige Verteilung der geringen Ressourcen auf alle Systemteile vor. So können zwar alle Systemteile getestet werden, jedoch kann dies nur sehr oberflächlich getan werden, was der Nachteil dieser Strategie ist. Die enthaltenen Fehler in den Systemteilen können mit den wenigen Ressourcen pro Systemteil nicht aufgedeckt werden. Die zweite Strategie wird gut durch Zimmermann und Nagappan beschrieben: „*When managers want to spend resources most effectively, they would typically allocate them on the parts where they expect most defects or at least the most severe ones“ (*[ZN08], Seite 531*).* Das bedeutet, dass die Ressourcen auf eine ausgewählte Menge von Systemteilen verteilt werden, wodurch diese intensiver getestet werden können. Nachteilig ist aber, dass nicht-ausgewählte Systemteile nicht getestet werden. Bei der Verwendung dieser Strategie müssen die zu testenden Systemteile sorgfältig ausgewählt werden. Die Auswahl sollte sich daher auf die zu testenden Systemteile konzentrieren, die mit hoher Wahrscheinlichkeit eine hohe Fehleranzahl aufweisen.
Im Integrationstest sind die zu testenden Systemteile die Abhängigkeiten sowie die an den Abhängigkeiten beteiligten Bausteine. Da ein Softwaresystem häufig mehr Abhängigkeiten als Bausteine besitzt, ist es umso wichtiger, dass die zu testenden Abhängigkeiten sorgfältig ausgewählt werden. Das Ziel muss es sein, die Abhängigkeiten auszuwählen, die mit einer hohen Wahrscheinlichkeit einen Fehler enthalten.
Im Nachfolgenden wird ein Ansatz vorgestellt, der die Auswahl von zu testenden Abhängigkeiten für ein Softwaresystem ermöglicht. Die Abhängigkeiten, die als Testfokus ausgewählt werden, weisen eine höhere Wahrscheinlichkeit auf, einen Fehler zu enthalten, als nicht ausgewählte Abhängigkeiten. Hierbei werden die Eigenschaften der Abhängigkeiten sowie Informationen aus früheren Versionen des zu testenden Softwaresystems verwendet. Für frühere Versionen werden mit Hilfe statistischer Verfahren Abhängigkeitseigenschaften identifiziert, die auf eine hohe Fehleranzahl in der Abhängigkeit hinweisen. Aufbauend auf diesen Ergebnissen werden für die aktuelle (und die nachfolgenden) Versionen die Abhängigkeiten als Testfokus ausgewählt, die diese Eigenschaften besitzen. Ein Überblick über den gesamten Prozess ist in Abbildung 16 dargestellt.
In einem ersten Schritt (1.) werden die Eigenschaften von Abhängigkeiten festgelegt, die zur Testfokusauswahl herangezogen werden. Anschließend werden frühere Versionen des zu testenden Softwaresystems herangezogen, um Zusammenhänge zwischen Eigenschaften und Fehleranzahl in den beteiligten Bausteinen aufzudecken. Hierzu werden die Schritte 2.1-2.4 durchgeführt. Es wird die Fehleranzahl für die Bausteine des Softwaresystems ermittelt (2.1). Anschließend werden die Abhängigkeiten und ihre Eigenschaften erhoben (2.2). Zur Analyse des Zusammenhangs zwischen Fehleranzahl der Bausteine und der Abhängigkeitseigenschaften werden statistische Verfahren eingesetzt. Diese setzen voraus, dass die Abhängigkeiten nach ihren Eigenschaften gruppiert werden (2.3). Abschließend können die Korrelationsanalysen für jede Version durchgeführt werden (2.4). Gesucht wird dabei nach Zusammenhängen zwischen den Eigenschaften und der Fehleranzahl. Diese Zusammenhänge müssen in möglichst vielen früheren Versionen existieren. Sobald die Informationen der Korrelationsanalyse für verschiedene Versionen vorliegen, können die Ergebnisse zur Bestimmung des Testfokus für die aktuelle Version verwendet werden.

Hierzu werden die Abhängigkeiten und ihre Eigenschaften für die zu testende Version erhoben (3.1). Die erhobenen Abhängigkeiten werden nach dem gleichen Muster wie in Schritt 2.3 gruppiert (3.2). Anschließend werden die Abhängigkeiten als Testfokus ausgewählt, deren Eigenschaften eine Korrelation mit der Fehleranzahl in früheren Versionen aufweisen (3.3).

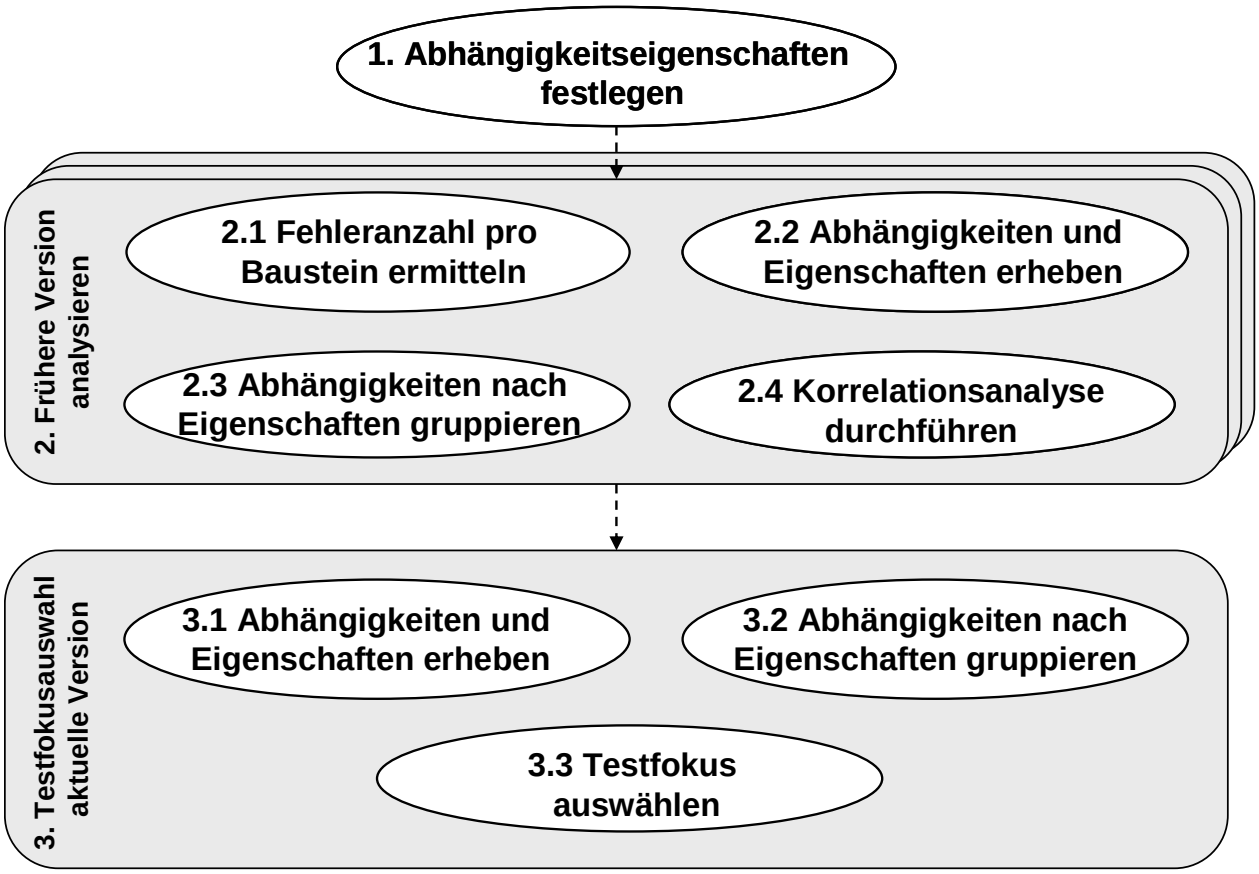


Abbildung 16: Vorgehen zur Testfokusauswahl

Das Vorgehen wird nachfolgend näher beschrieben. Kapitel 5.1 konzentriert sich hierbei auf statistische Grundlagen, die für das Verständnis des Vorgehens notwendig sind. Kapitel 5.2 gibt einen Überblick über verwandte Verfahren zur Ermittlung von Zusammenhängen zwischen Eigenschaften und Fehleranzahl. Kapitel 5.3 beschreibt den Ansatz zur Testfokusauswahl im Detail. Im Anschluss werden zwei Fallstudien vorgestellt (Kapitel 5.4), die die Anwendbarkeit des Ansatzes an realistisch großen Softwaresystemen demonstrieren.

5.1. Statistische Grundlagen

Für die Identifikation von fehleranfälligen Abhängigkeiten werden statistische Verfahren eingesetzt. Diese Identifikation bildet die Grundlage des vorgestellten Ansatzes der Testfokusauswahl.

Für die Testfokusauswahl wird überprüft, ob Eigenschaften einer Abhängigkeit Auskunft darüber geben können, ob die Abhängigkeit fehlerhaft ist. In der Statistik gibt es Verfahren zur Überprüfung von Zusammenhängen zwischen einer abhängigen Variablen (z.B. die Fehleranzahl in einem Baustein) und einer unabhängigen Variablen (z.B. eine Eigenschaft der Abhängigkeit). Hierfür wird für jeden Fall (= eine Abhängigkeit) innerhalb einer Stichprobe überprüft, ob es einen Zusammenhang gibt und wie stark dieser gegebenenfalls ausgeprägt ist. Die Stichprobe stellt eine Auswahl von Fällen aus der Gesamtmenge aller

möglichen Fälle (Grundgesamtheit) dar. Ein Fall beinhaltet alle abhängigen und unabhängigen Variablen (z.B. alle Eigenschaften einer Abhängigkeit, Informationen über Fehler in den beteiligten Bausteinen), die diesen Fall definieren. Diese Variablen können auf unterschiedlichen Mess- bzw. Skalenniveaus gemessen werden. In der Statistik werden fünf Skalenniveaus unterschieden. In aufsteigender Reihung sind dies das nominale, das ordinale, das Intervall-, das Ratio- und das absolute Skalenniveau ([Di02]). Abhängig vom Skalenniveau sind unterschiedliche mathematische Operationen sowie statistische Auswertungen und Verfahren erlaubt. Skalen höheren Niveaus enthalten die Eigenschaften der niedrigen Skalen. D.h. alle Operationen, die für eine Ordinalskala erlaubt sind, können auch für eine Intervall- oder Ratioskala verwendet werden. Umgekehrt ist dies nicht zulässig. Weiterhin lässt sich jede Skala höheren Niveaus in eine Skala niedrigeren Niveaus überführen, allerdings nur mit entsprechendem Informationsverlust.
Nominal skalierte Variablen können in keine sinnvolle Reihenfolge gebracht werden. Typische Beispiele für eine nominale Skala ist die Unterscheidung des Geschlechts einer Person (männlich, weiblich). Die *Ordinalskala* ermöglicht die Definition von Rangfolgen, d.h. für zwei Werte kann eindeutig differenziert werden, ob ein Wert größer oder kleiner als der zweite Wert ist. Die Ordinalskala kann keine Aussagen über die Abstände zweier Werte machen. Ein Beispiel für eine Ordinalskala ist die Einteilung von Personen nach ihrem Schulabschluss (kein Abschluss, Hauptschule, Realschule, Gymnasium). Die *Intervallskala* ergänzt die Ordinalskala um die fehlenden Abstände zwischen den Ausprägungen. Sie erlaubt exakte Aussagen über den Abstand zweier Werte. Der Nullpunkt in einer Intervallskala kann frei definiert werden. Ein Beispiel für eine Intervallskala ist die Temperatur in °C oder °F. Eine *Ratioskala* besitzt im Gegensatz zur Intervallskala einen natürlichen Nullpunkt. Ein Beispiel für eine Ratioskala ist das monatliche Einkommen. Die Ratioskala hat die Freiheit, sich die Skaleneinheiten selbst zu wählen. Beispielsweise kann die Länge einer Geraden in Zentimeter oder auch in Meter angegeben werden. Diese Freiheit ist in einer *Absolutskala* nicht gegeben. Beispiele für eine Absolutskala sind Häufigkeiten oder Wahrscheinlichkeiten.
Für die Überprüfung von Zusammenhängen zwischen abhängigen und unabhängigen Variablen steht eine Vielzahl von statistischen Tests zur Verfügung, z.B. *Signifikanztests*, *Zusammenhangsmaße*, *Regressionsanalysen* oder *Mittelwertvergleiche*.

Signifikanztest
Mit Hilfe von Signifikanztests kann überprüft werden, ob eine bestimmte Annahme zutrifft oder nicht. Ein Beispiel für solch eine Annahme wäre: „Es existiert ein Zusammenhang zwischen der abhängigen und der unabhängigen Variable.“ Hierzu werden zwei Hypothesen aufgestellt. Die Hypothese H_1 beschreibt die Annahme wohingegen die zweite Hypothese H_0 das Gegenteil darstellt (z.B. es existiert kein Zusammenhang). Mit Hilfe der Signifikanztests wird überprüft, welche Hypothese angenommen bzw. verworfen werden muss. Ein sehr häufig verwendeter Signifikanztest ist der Chi-Quadrat-Test. „Der Chi-Quadrat-Test ist ein Test, der überprüft, ob nach ihrer empirischen Verteilung zwei in einer Stichprobe erhobene Variablen voneinander unabhängig sind oder nicht.“ (vgl. [JL03], Seite 229). In einem ersten Schritt werden die beiden Hypothesen H_0 und H_1 aufgestellt. H_0 besagt, es gibt keine Abhängigkeit zwischen den zwei untersuchten Variablen. H_1 besagt, es besteht ein solcher Zusammenhang. Anschließend wird die statistische Prüfgröße Chi-Quadrat[1] ermittelt. Im Anschluss muss das Signifikanzniveau festgelegt werden, d.h. die Wahrscheinlichkeit, mit der H_1 angenommen wird. Üblicherweise wird ein Niveau von 5% (entspricht einem signifikanten Ergebnis) oder ein Niveau von 1% (entspricht einem hoch signifikanten Ergebnis) ausgewählt. Im nächsten Schritt werden die Freiheitsgrade für die Verteilung, in

[1] Zur Berechnung der Prüfgröße Chi-Quadrat sei auf die Formel in [JL03] Seite 229 verwiesen.

diesem Fall die Chi-Quadrat-Verteilung, festgelegt. Mit Hilfe der Freiheitsgrade und des Signifikanzniveaus werden die kritischen Bereiche der Verteilungsfunktion bestimmt. Dies kann in der Chi-Quadrat-Tabelle nachgelesen werden. Abschließend wird überprüft, ob die ermittelte Prüfgröße Chi-Quadrat in den kritischen Bereich fällt. Fällt die Prüfgröße in diesen Bereich, kann H_1 angenommen werden, d.h. es besteht ein signifikanter Zusammenhang zwischen den untersuchten Variablen. Anderenfalls wird H_1 verworfen und H_0 beibehalten, d.h. es konnte kein signifikanter Zusammenhang zwischen den untersuchten Variablen gefunden werden.
Mit Hilfe des Chi-Quadrat-Tests kann ermittelt werden, ob zwischen zwei Variablen ein Zusammenhang besteht. Er gibt aber keine Auskunft darüber, wie stark der Zusammenhang ausgeprägt ist.

Zusammenhangsmaß
Wenn mit Hilfe des Chi-Quadrat-Tests ein signifikanter Zusammenhang zwischen zwei Variablen ermittelt wurde, muss anschließend geprüft werden, wie stark dieser Zusammenhang ist und ob er eine positive oder negative Ausprägung besitzt. Hierüber können Zusammenhangsmaße Auskunft geben. „Die Mehrzahl der Zusammenhangsmaße ist […] so ausgelegt, dass der Wert 0 anzeigt, dass kein Zusammenhang zwischen den beiden Variablen vorliegt. Der Wert 1 dagegen indiziert einen perfekten Zusammenhang. Werte zwischen 1 und 0 stehen für mehr oder weniger starke Zusammenhänge“ (Seite [JL03], S. 234). Für unterschiedliche Skalenniveaus stehen verschiedene statistische Verfahren zur Verfügung. Ein Zusammenhang zwischen zwei nominalskalierten Variablen kann mit Hilfe des „Kruskal und Goodmann tau“ Tests (vgl. [JL03] Seiten 236ff) ermittelt werden. Für ordinalskalierte Variablen kann der Rangkorrelationskoeffizient r_s nach Spearman (vgl. [JL03] Seiten 242ff) zur Anwendung kommen. „Das bekannteste [Zusammenhangsmaß] ist der Pearsonsche Produkt-Moment-Korrelations-Koeffizient r“ (vgl. [JL03] Seite 246) für intervallskalierte Variablen. Während für nominalskalierte Variablen nur angegeben werden kann, wie stark der Zusammenhang ist, kann bei mindestens ordinalskalierten Variablen die Richtung angegeben werden. Ist das Zusammenhangsmaß positiv, gibt es eine positive Korrelation zwischen der unabhängigen und der abhängigen Variablen. D.h. je größer der Wert der unabhängigen Variablen ist, desto größer ist der Wert der abhängigen Variablen. Ist das Zusammenhangsmaß negativ, ist auch der Zusammenhang negativ. D.h. mit sinkenden Werten der unabhängigen Variable steigen die Werte der abhängigen Variable oder umgekehrt.

Regression
Die Regression [JL03] zielt darauf ab, mit Hilfe von Gleichungen den Zusammenhang zwischen der abhängigen und einer (oder mehrerer) unabhängigen Variablen zu beschreiben. Anschließend wird überprüft, wie gut die Gleichung diesen Zusammenhang abbildet. Für die Regression können verschiedene Gleichungen genutzt werden. Die lineare Regressionsanalyse verwendet eine lineare Gleichung. Abhängig davon, wie viele unabhängige Variablen in die Gleichung einfließen, spricht man von einer einfachen linearen Regression (nur eine unabhängige Variable) oder einer multiplen linearen Regression (mehrere unabhängige Variablen). Für die einfache lineare Regression hat die verwendete Gleichung die Form:

$$y'_i = x_i * b_1 + b_0$$

$\mathbf{y'_i}$ stellt den Vorhersagewert dar, d.h. den Wert, der mit Hilfe der Formel ermittelt wurde. Je stärker sich der Vorhersagewert $\mathbf{y'_i}$ der tatsächlichen Variablen $\mathbf{y_i}$ für möglichst viele Fälle **i** annähert, desto stärker ist der lineare Zusammenhang zwischen der abhängigen Variablen **y**

und der unabhängigen Variable **x**. Um dies zu erreichen, müssen $\mathbf{b_0}$ und $\mathbf{b_1}$ so ermittelt werden, dass die Fehler, d.h. die Abweichungen der $\mathbf{y'_i}$ von tatsächlichen $\mathbf{y_i}$, möglichst klein sind. Hierzu wird die Methode der kleinsten Quadrate verwendet, d.h. die Werte für $\mathbf{b_0}$ und $\mathbf{b_1}$ werden so gewählt, dass die Summe für alle $(y_i - y'_i)^2$ minimal ist. Mit Hilfe des Bestimmtheitsmaßes R^2 kann der Zusammenhang zwischen der abhängigen Variablen und den unabhängigen Variablen in Zahlen gefasst werden. R^2 kann einen Wert zwischen 0 und 1 annehmen und beschreibt, wie gut sich der Zusammenhang zwischen abhängiger Variable und unabhängigen Variablen durch die lineare Gleichung annähern lässt. Bei einem Wert von 1 liegen alle Werte auf einer Geraden und es gibt einen perfekten linearen Zusammenhang zwischen der abhängigen und der unabhängigen Variable. Bei einem R^2 von 0 liegen die Werte als Punktwolke vor und es gibt keinen Zusammenhang.
Eine weitere Form der Regression ist die binär logistische Regression. Die abhängige Variable muss für diese Form der Regression dichotom sein, d.h. sie darf nur zwei Werte annehmen dürfen. Diese zwei Werte müssen sich auf 1 und 0 abbilden lassen. Mit Hilfe von Exponentialfunktionen wird der Zusammenhang zwischen der/den unabhängigen Variable/n und der dichotomen abhängigen Variable beschrieben.

Mittelwertvergleich
Voraussetzung für die Anwendung von Mittelwertvergleichen ist das Vorliegen von gruppierten unabhängigen Variablen, d.h. sie müssen nominal- oder ordinalskaliert vorliegen. intervallskalierte Variablen müssen zuvor in Gruppen eingeteilt werden. Hierzu können Quantile verwendet werden. Die abhängige Variable hingegen kann intervall- oder ratioskaliert sein. Grundidee des Mittelwertvergleiches ist es, für jede Gruppe der abhängigen Variablen das arithmetische Mittel der abhängigen Variablen zu berechnen und mit den anderen Gruppen zu vergleichen. Sind die Mittelwerte der Gruppen gleich, gibt es keinen Zusammenhang zwischen der abhängigen und der unabhängigen Variablen. Gibt es einen Unterschied zwischen den Mittelwerten von mindestens zwei Gruppen, kann mit Hilfe eines Signifikanztests überprüft werden, ob dieser Zusammenhang zwischen der abhängigen und der unabhängigen Variablen signifikant, d.h. statistisch abgesichert, ist.
Als statistischer Test für die Überprüfung der Mittelwerte zweier Gruppen (zwei unabhängiger Stichproben) kann der T-Test verwendet werden. Er berechnet den Mittelwert für die zwei untersuchten Stichproben und ermittelt gleichzeitig die Signifikanz. Eine Erweiterung des T-Tests auf mehr als zwei Gruppen (k unabhängige Stichproben) stellt die einfaktorielle Varianzanalyse (ANOVA) dar. Sie überprüft, ob sich die Mittelwerte von mindestens zwei Gruppen signifikant voneinander unterscheiden. Sie kann jedoch keine Aussagen darüber machen, welche Gruppen sich voneinander unterscheiden.
Für die Anwendung des T-Tests und der ANOVA muss eine Reihe von Vorbedingungen erfüllt sein, damit die verwendeten mathematischen Formeln die richtigen Ergebnisse liefern:

1) Die abhängige Variable muss mindestens intervallskaliert sein.

2) Die abhängige Variable muss in der Grundgesamtheit normalverteilt[1] sein.

3) Die unabhängige Variable muss kategorial, d.h. entweder nominal- oder ordinalskaliert sein.

Zusätzlich zu diesen drei Bedingungen gilt für die ANOVA, dass die Gruppen der unabhängigen Variablen unabhängig voneinander sein müssen, d.h. die einzelnen Fälle müssen sich eindeutig in eine der Gruppen einordnen lassen und die Gruppen der

[1] Die Normalverteilung kann mit dem Kolmogorov-Smirnov-Test überprüft werden (vgl. [JL03]).

unabhängigen Variable sollten in etwa die gleiche Varianz[1] aufweisen. Durch diese Voraussetzung von Bedingungen werden die Tests auch als parametrische Tests bezeichnet.
Sind die Voraussetzungen für die Anwendung eines parametrischen Tests nicht gegeben, muss auf nichtparametrische (oder verteilungsfreie) Tests zurückgegriffen werden, da anderenfalls die parametrischen Tests falsche Ergebnisse liefern können.
Das nichtparametrische Gegenstück zum T-Test stellt der Mann-Whitney-U-Test dar. Er kann angewendet werden, wenn die abhängige Variable nicht mindestens intervallskaliert, sondern „nur" ordinalskaliert vorliegt oder wenn die Bedingung der Normalverteilung nicht erfüllt ist. (vgl. [JL03] Seite 497). Der Mann-Whitney-U-Test verwendet nicht die Messwerte der Variablen, sondern deren Rangplätze. D.h. die Fälle werden der Größe nach, aufsteigend sortiert. Ihre Position in der sortierten Liste stellt ihren Rangplatz dar. Besitzen mehrere Fälle den gleichen Wert, so wird der mittlere Rang gebildet und allen betreffenden Fällen zugeordnet. Diese Vorgehensweise ist in Tabelle 3 veranschaulicht. Für die Fälle F1 bis F9 wurden die Fälle nach der abhängigen Variable aufsteigend sortiert. Jedem Fall wurde ein Rang zugewiesen. Da die abhängige Variable der Fälle F3 und F9 den Wert 1 besitzen, wurde der mittlere Rang für beide Fälle vergeben: (1 + 2)/2 = 1,5.

Tabelle 3: Beispiel der Rangzuweisung

Fall	Wert abhängige Variable	Fortlaufender Rang	Mittlerer Rang	Unabhängige Variable
F3	1	**1**	**1,5**	**3**
F9	1	**2**	**1,5**	**4**
F7	2	**3**	**3**	**4**
F4	3	**4**	**4**	**3**
F1	4	**5**	**5**	**3**
F6	5	**6**	**7**	**4**
F2	5	**7**	**7**	**4**
F8	5	**8**	**7**	**4**
F5	10	**9**	**9**	**3**

Ähnlich zum T-Test ermittelt der Mann-Whitney-U-Test den Mittelwert der Ränge für zwei Gruppen und überprüft, ob sich die Mittelwerte der Ränge signifikant voneinander unterscheiden.
Wie die ANOVA bei parametrischen Tests können mit Hilfe des Kruskal-Wallis-H-Tests mehrere Gruppen gleichzeitig überprüft werden. Äquivalent zum Mann-Whitney-U-Test werden beim Kruskal-Wallis-H-Test die Ränge der abhängigen Variablen der einzelnen Gruppen verwendet und überprüft, ob sie sich signifikant voneinander unterscheiden. Hierzu wird ein Prüfwert H ermittelt. „Diese Prüfgröße ist approximativ chi-quadrat-verteilt mit k-1 Freiheitsgraden" (vgl. [JL03] Seite 505). Anhand der Chi-Quadrat-Tabelle kann ein kritischer Wert für die gegebenen Freiheitsgrade entnommen werden. Liegt der Prüfwert H über dem kritischen Wert, wird der gefundene Zusammenhang als signifikant angesehen. Wie auch bei der ANOVA kann der Kruskal-Wallis-H-Test keine Auskunft darüber geben, welche Gruppen sich signifikant unterscheiden.

Einige der statistischen Verfahren zur Ermittlung von Zusammenhängen setzen nominal- bzw. ordinalskalierte Variablen voraus. Liegen die Variablen jedoch auf einem höheren Skalenniveau vor, müssen diese in eine Ordinalskala überführt werden. Hierfür können Quantile verwendet werden. Quantile sind in der Statistik eine Art Streuungsmaß. Dabei

[1] Die Gleichheit der Varianzen kann mit Hilfe des Levene Tests überprüft werden (vgl [JL03]).

werden die Werte einer betrachteten Variablen aufsteigend oder absteigend sortiert. Anschließend werden die betrachteten Fälle in n gleich große Teile unterteilt. Abhängig davon, wie groß n gewählt wird, sprechen Statistiker von Quartilen (n=4), Dezilen (n=10) oder Perzentilen (n = 100). Wichtig dabei ist, dass jeder Wert eines Quantils nicht größer sein darf als alle Werte der darunter liegenden Quantile. Weiterhin darf jeder Fall nur genau in einem Quantil zu finden sein. Der Vorteil der Quantile ist, dass die Fälle in etwa gleich große Gruppen eingeteilt werden können.

5.2. State of the Art – Testfokusauswahl

In den letzten Jahren sind eine Vielzahl von empirischen Untersuchungen durchgeführt worden (z.B. in [BBM96], [OW07], [NBZ06], [Zh08]) um Zusammenhänge zwischen Eigenschaften von Bausteinen und ihrer Fehleranfälligkeit, d.h. der Anzahl der in ihnen enthaltenen Fehler, zu erforschen. Mit Hilfe von statistischen Verfahren wurden mehr oder weniger starke Korrelationen zwischen einzelnen Eigenschaften und der Fehleranfälligkeit aufgezeigt. Diese Korrelationen wurden verwendet, um Fehlerprognosen für nachfolgende Releases eines Softwaresystems zu erstellen. Das übergeordnete Ziel ist es, Korrelationen zu finden, die sich allgemeingültig auf alle Softwaresysteme und in allen Softwareprojekten anwenden lassen, um sie in jedem Softwareentwicklungsprojekt einsetzen zu können. Aber: "*There is no universal metric or prediction model that applies to all projects*" (vgl. [ZN08] Seite 531 mit Bezug auf [NBZ06]). Aus diesem Grund konzentrieren sich die Ansätze wie auch der Ansatz dieser Arbeit auf das Vorhersagen der Fehleranfälligkeit für nachfolgende Releases. Es werden Informationen über Eigenschaften und Fehler aus früheren Versionen verwendet, um Korrelationen zu ermitteln und daraus ein Vorhersagemodell zu konstruieren. Das Vorhersagemodell wird mit den Informationen früherer Releases „trainiert".

Die Modelle in der Literatur können die Fehleranfälligkeit mit unterschiedlicher Genauigkeit prognostizieren. Einige Modelle können Vorhersagen machen, welche Bausteine fehleranfällig sind und welche nicht. D.h. sie ermitteln, ob in einem Baustein mit einer gewissen Wahrscheinlichkeit mindestens einen Fehler enthalten sein wird oder nicht. Sie können aber nicht exakt vorhersagen, wie viele Fehler in einem Baustein vorhanden sein werden (z.B. [BBM96], [RSG08]). Andere Modelle können die Anzahl der in den Bausteinen enthaltenen Fehler vorhersagen (z.B. [Zh08], [OW07]). Die letzte Gruppe von Modellen sagen nicht die konkrete Anzahl der Fehler voraus, sondern die Fehlerdichte (z.B. [KPB06], [NB05]).

Je nachdem, wie genau die Fehler (mindestens ein Fehler, exakte Anzahl oder Dichte) prognostiziert werden sollen, werden unterschiedliche statistische Verfahren zur Vorhersage verwendet. Der in Kapitel 5.3 vorgestellte Ansatz lässt sich nicht eindeutig in eine der drei Kategorien einordnen. Ziel des Ansatzes ist es, zu differenzieren, ob eine Gruppe von Abhängigkeiten mit einer höheren Wahrscheinlichkeit mehr Fehler besitzen wird als eine andere Gruppe. Der Ansatz ermöglicht es nicht, die genaue Fehleranzahl für eine Abhängigkeit zu bestimmen oder die Fehlerdichte vorherzusagen. Der Ansatz ist entwickelt worden, um den Testfokus für den Integrationstest festlegen zu können.

Ein sehr einfaches Vorhersagemodell ist in [Zh08] für die Fehleranzahl zukünftiger Eclipse Versionen erstellt worden. Es wurde nur die Anzahl der Fehler früherer Versionen verwendet, um die Anzahl Fehler zukünftiger Versionen vorherzusagen. Hierzu wurde eine Polynomialfunktion in der Form

$$\mathbf{y = b_2 x^2 + b_1 x + b_0}$$

verwendet, wobei **y** die Anzahl der kumulierten Fehler aller früheren Fehler einschließlich der Fehler der vorhergesagten Version x darstellt. Die Faktoren b_2, b_1 und b_0 wurden mit

Hilfe der Fehleranzahl vorangegangener Versionen ermittelt. Aus dem Wert **y** kann anschließend der vorhergesagte Wert für die aktuelle Version ermittelt werden, indem die Summe der Fehler in vorangegangenen Versionen von **y** abgezogen wird.
In [RSG08] wurde der Zusammenhang zwischen Fehleranfälligkeit und der Anzahl der Modifikationen eines Bausteins untersucht. Dabei zeigte sich, dass „*The number of software defects decreases, if the number of refactorings increased.* (vgl. [RSG08] Seite 35). Dieser Zusammenhang konnte in allen fünf untersuchten Softwaresystemen nachgewiesen werden. Allerdings geben die Autoren keine genaue Auskunft über die verwendeten statistischen Verfahren. Sie verwenden ein Werkzeug namens WEKA zur Ermittlung des Zusammenhangs.
In [BBM96] führten die Autoren Studentenexperimente durch, um vorherzusagen, ob im Akzeptanztest ein Fehler gefunden wird. Mit Hilfe des Rangkorrelationskoeffizienten nach Spearman wurde überprüft, ob Bausteineigenschaften mit der Wahrscheinlichkeit einen Fehler im Akzeptanztest zu finden korrelieren. Die „univariate logistische Regression“ und die „multivariate logistische Regression“ wurden verwendet, um vorherzusagen, ob im Akzeptanztest ein Fehler gefunden wird.
Munson et al. kombinierten in [MK92] verschiedene Komplexitätsmetriken, z.B. Anzahl Quelltextzeilen, Anzahl Zeichen oder McCabes zyklomatische Zahl [Mc76]. Da die untersuchten Metriken stark voneinander abhingen, verwendeten die Autoren die Hauptkomponentenanalyse. Die Hauptkomponentenanalyse errechnet aus der Gesamtmenge der untersuchten Metriken eine Menge neuer Metriken, die die gleiche Aussagekraft haben, wie die untersuchten Metriken. Ziel ist es, die Anzahl der Metriken zu verringern. In [MK92] korrelierten die neuen Metriken sehr gut mit der Fehleranfälligkeit. Nachteil war allerdings, dass die durch die Hauptkomponentenanalyse ermittelten Metriken nicht interpretierbar waren.
In [FO00] und in [OW07] zeigen die Autoren auf, dass die Fehler nicht gleichmäßig auf die Bausteine der untersuchten Softwaresysteme verteilt sind. Vielmehr stellte sich heraus, dass im Durchschnitt 80% der Fehler des gesamten Softwaresystems in 20% der Bausteine zu finden sind. Diese Untersuchung zeigt, dass der Aufwand der Qualitätssicherung auf die Bausteine konzentriert werden muss, in denen die meisten Fehler enthalten sind.
In [KPB06] widerlegen die Autoren die weit verbreitete Annahme, dass eine hohe Anzahl an Quelltextzeilen (LOC) in einem Baustein zu einer hohen Anzahl an Fehlern führt. Viel aussagekräftiger als die LOC seien dagegen die Informationen über Änderungen an einem Baustein über die Zeit. Jedoch machen die Autoren lassen sich durch das Werkzeug WEKA ein Zusammenhangsmaß ermitteln. Leider geben sie keine Auskünfte darüber, mit welchen statistischen Verfahren dieses Zusammenhangsmaß ermittelt wird.
In [NB05] wurden die Ergebnisse der statischen Analyse von Quelltext verwendet, um die Fehleranfälligkeit von Bausteinen vorherzusagen. Die Kernidee war es, dass die statische Analyse und das dynamische Testen unterschiedliche Fehler finden. Mit Hilfe der Fehleranzahl, die durch die statische Analyse aufgedeckt wurde, wird die Anzahl der Fehler vorhergesagt, die im dynamischen Test gefunden werden können. Für ihre Analysen verwendeten die Autoren den Korrelationskoeffizienten nach Spearman.
Ein weiterer Überblick über acht empirische Untersuchungen ist in [SK03] zu finden. Sie zeigen alle, dass objektorientierte Metriken mit der Fehleranfälligkeit von Bausteinen korrelieren.
Alle empirischen Untersuchungen, die in der Literatur gefunden werden können, konzentrieren sich auf die Eigenschaften eines Bausteins. Untersuchungen, die die Abhängigkeit zwischen zwei Bausteinen betrachten, gibt es nicht. „*One drawback of most complexity metrics is that they only focus on single elements, but rarely take the interactions between elements into account*” (vgl. [ZN08] Seite 531). Zimmermann und Nagappan haben erkannt, dass nicht nur die Eigenschaften (Metriken) eines einzelnen Bausteins betrachtet

werden dürfen. Vielmehr müssen die Interaktionen zwischen den Bausteinen hinzugezogen werden. Die Untersuchungen in [ZN08] stützen sich aber nicht auf die Abhängigkeitseigenschaften einer Abhängigkeit zwischen zwei Bausteinen, sondern aggregieren die Eigenschaften (z.B. Anzahl der abhängigen Bausteinen (Clients) oder Anzahl der unabhängigen Bausteine (Server)) für einen Baustein. Die so erhobenen Metriken beziehen sich somit wieder nur auf einen Baustein und nicht auf die Abhängigkeit.
Kein in der Literatur beschriebenes Vorgehen betrachtet die Testobjekte des Integrationstests, um eine Fehlervorhersage und somit eine Testfokusauswahl im Integrationstest zu ermöglichen. Der in dieser Arbeit vorgestellte Ansatz zur Testfokusauswahl soll diese Lücke schließen. Er verwendet als erster Ansatz die Eigenschaften der Abhängigkeiten, um die Auswahl der Testobjekte des Integrationstests zu ermöglichen.

5.3. Vorgehen zur Ermittlung des Testfokus

Das Vorgehen zur Auswahl des Testfokus gliedert sich in zwei umfassende Aktivitäten. Zum einen müssen fehleranfällige Abhängigkeitseigenschaften mit Hilfe von statistischen Verfahren identifiziert werden. Zum anderen werden die Informationen über die fehleranfälligen Eigenschaften für die Testfokusauswahl in der zu testenden Version des Softwaresystems ausgenutzt.

5.3.1. Fehleranfällige Abhängigkeitseigenschaften

Die Testfokusauswahl für den Integrationstest setzt die Kenntnis über Zusammenhänge zwischen Abhängigkeitseigenschaften und Fehleranfälligkeit der Abhängigkeit voraus. Diese Zusammenhänge werden mit Hilfe der Informationen aus früheren Versionen eines Softwaresystems und statistischer Verfahren gewonnen. Die in Kapitel 5.2 vorgestellten Arbeiten zur Fehleranalyse bzw. Fehlervorhersage verwenden verschiedene Verfahren, um Korrelationen zwischen Eigenschaften und Fehleranzahl aufzudecken und die Signifikanz der Korrelation zu überprüfen. Sie alle konzentrieren sich nur auf Eigenschaften von Bausteinen selbst, z.B. auf Quelltextzeilen, angebotene Services, zyklomatische Komplexität nach McCabe [Mc76], notwendige Server und abhängige Clients. Ihre Ergebnisse können somit sehr gut im Unittest verwendet werden. Sie geben Auskunft darüber, welche Bausteine im Unittest intensiver zu testen sind.
Nachfolgend wird ein Ansatz, basierende auf den Ideen in [IP08a], [IP08b] und [IP09] vorgestellt, der Zusammenhänge zwischen den Eigenschaften einer Abhängigkeit und der Fehleranzahl der beteiligten Bausteine aufdeckt. Die Eigenschaften, die eine Korrelation mit der Fehleranzahl von mindestens einem der beteiligten Bausteine aufweist, werden für die Testfokusauswahl des Integrationstests verwendet. In diesem Kapitel wird beschrieben, wie solche Zusammenhänge zwischen Abhängigkeitseigenschaften und Fehleranzahl ermittelt werden können (vgl. Schritte 1.-2.4 in Abbildung 16).

5.3.1.1 Vorbereitende Schritte der Korrelationsanalyse

Bevor mit der Korrelationsanalyse begonnen werden kann, müssen die zu untersuchenden Abhängigkeitseigenschaften festgelegt werden (Schritt 1., vgl. Abbildung 16). Hierbei kann die in Kapitel 1 vorgeschlagene Klassifikation von Abhängigkeiten verwendet werden. Welche Eigenschaften für die Analyse herangezogen werden können, hängt im Wesentlichen von zwei Faktoren ab:

- Verfügbarkeit der Eigenschaften
- Aufwand zur Erhebung der Eigenschaften für Abhängigkeiten

Eigenschaften, die in dem zu untersuchenden Softwaresystem nicht anzutreffen sind (z.B. Vererbungseigenschaften in einer nicht-objektorierentierten Programmiersprache), sind für die Analyse nicht relevant. Des Weiteren können keine Eigenschaften verwendet werden, für die die Eigenschaften nicht erhoben werden können, da die notwendigen Dokumente nicht bzw. nicht mehr zur Verfügung stehen. Auf der anderen Seite ist es wichtig, eine Kosten-Nutzen-Betrachtung durchzuführen. Einige Eigenschaften lassen sich sehr leicht automatisch aus existierenden Entwicklungsartefakten (z.B. dem Quelltext) extrahieren. Andere Eigenschaften sind nur mit sehr viel Aufwand zu erheben. Häufig sind dies Eigenschaften, die nicht automatisch erhoben werden können, sondern manuell ermittelt werden müssen. In den Fallstudien in 5.4 wurden Abhängigkeitseigenschaften ausgewählt, die sich mit Hilfe von statischen Analysewerkzeugen automatisch aus dem Quelltext extrahieren lassen. In den Fallstudien wurde im ersten Schritt überprüft, welche Entwicklungsartefakte zur Verfügung stehen und wie leicht sich daraus Abhängigkeitseigenschaften extrahieren lassen.
Nachdem die zu erhebenden Eigenschaften festgelegt worden sind, können die Vorbereitungen für die Analyse einzelner Versionen beginnen. Für jede zu untersuchende Version müssen die Schritte 2.1-2.4 aus Abbildung 16 ausgeführt werden. Im ersten Schritt (2.1) werden die Fehler der Bausteine der untersuchten Version ermittelt. Eine Möglichkeit zur Ermittlung dieser Fehleranzahl kann in Entwicklungsprojekten gefunden werden, die in einem Fehlermanagementsystem nicht nur die Informationen über den Fehler dokumentieren, sondern auch, welche Bausteine geändert werden mussten, um diesen Fehler zu beseitigen. Mit diesen Informationen ist eine Zuordnung von Fehlern zu Bausteinen sehr leicht möglich. Werden im Fehlermanagementsystem nur Informationen über Fehler gespeichert (z.B. Fehler-ID, Beschreibung, Schwere usw.) müssen die fehlerverursachenden Bausteine nachträglich identifiziert werden. Ein möglicher Ansatz hierfür wird in [Zi06], [ZPZ07] und [RSG08] vorgestellt. Die Autoren verwenden zusätzlich zu den Informationen aus dem Fehlermanagementsystem die Informationen aus einem Versionsverwaltungssystem (z.B. CVS [CV09], SVN [Su09]). In [Zi06] und [ZPZ07] werden die Identifikationsnummern der dokumentierten Fehler im Fehlermanagementsystem mit den Einträgen im Versionsverwaltungssystem verglichen. Bei jedem „Einchecken“ des Quelltextes in das Versionsverwaltungssystem hat der Entwickler die Möglichkeit, einen Kommentar einzufügen, in dem er beschreibt, welche Änderungen er vorgenommen hat und warum. Wenn er einen Fehler beseitigt hat, kann er die Identifikationsnummer aus dem Fehlermanagementsystem dort eintragen, mit dem Vermerk, dass dieser Fehler behoben wurde. Der Ansatz von [Zi06] und [ZPZ07] durchsucht systematisch die Versionshistorie des Softwaresystems nach Identifikationsnummern von Fehlern. Wird eine Identifikationsnummer in einem Kommentar gefunden, wird dieser Fehler allen Bausteinen zugeordnet, die mit dem betreffenden „Check-In“ des Kommentars in das Versionsverwaltungssystem geschrieben wurden. Auf diese Weise kann für jeden Baustein im Versionsverwaltungssystem eine Fehleranzahl zugeordnet werden. In [RSG08] wurde dieser Ansatz erweitert und zusätzlich um eine Schlüsselwortsuche im Versionsverwaltungssystem ergänzt. Hierbei wird nicht nur nach der Identifikationsnummer eines Fehler gesucht, sondern zusätzlich nach bestimmten Schlüsselwörtern, die auf die Beseitigung von Fehlern hindeuten, z.B. „bug fixed“ oder „bug solved“. Beide Verfahren ermöglichen es, die gefundenen Fehler zu bestimmten Softwaresystemversionen und Bausteinen zuzuordnen. Alle Fehler, die innerhalb eines definierten Zeitraums nach dem Erscheinen einer Version gefunden und beseitigt wurden, werden dieser Version zugeordnet. Nachdem die Fehleranzahl pro Baustein der untersuchten Version ermittelt worden ist, können im zweiten Schritt (2.2) die Abhängigkeiten und ihre Eigenschaften identifiziert werden. Eine Abhängigkeit besteht dabei aus einem abhängigen Baustein und einem unabhängigen Baustein sowie einer Reihe von Eigenschaften, die diese Abhängigkeit charakterisieren. Die Informationen über die

Abhängigkeit werden um die Fehleranzahl des abhängigen und des unabhängigen Bausteins ergänzt. Das Ergebnis dieses Schrittes ist eine Tabelle, die alle Informationen über die Abhängigkeiten und ihre Eigenschaften enthält. Jede Zeile der Tabelle stellt eine Abhängigkeit dar. Die Spalten repräsentieren die Eigenschaften der Abhängigkeiten. Zwei der Spalten enthalten dabei den eindeutigen Namen des abhängigen und des unabhängigen Bausteins. Jede Konstellation von abhängigen und unabhängigen Bausteinen darf dabei nur einmal auftreten. Die weiteren Spalten der Tabelle enthalten die Informationen über die Abhängigkeitseigenschaften. Für jede Abhängigkeit werden diese Eigenschaften erhoben. Besitzt eine Abhängigkeit eine untersuchte Eigenschaft nicht, so wird dieser Wert vorerst mit einem „Fehlwert“ (z.B. „-111“) versehen. Die Definition der Fehlwerte sorgt während der Korrelationsanalyse für einen Ausschluss dieser Abhängigkeit innerhalb der Untersuchung der entsprechenden Eigenschaft.
Das statistische Verfahren, das für die Korrelationsanalyse zum Einsatz kommt, setzt eine Gruppierung der Abhängigkeiten nach Eigenschaften voraus (Schritt 2.3), da die unabhängige Variable (Abhängigkeitseigenschaft) ordinalskaliert ist (vgl. 5.1). Die Einteilung der Abhängigkeiten in Gruppen wird anhand der Eigenschaften vorgenommen. Für jede Eigenschaft werden die Abhängigkeiten aufsteigend sortiert. Das Sortierkriterium wird durch die Eigenschaft vorgegeben. Ein Beispiel: Die Anzahl der aufgerufenen Services zwischen zwei Bausteinen ist die untersuchte Eigenschaft. Die Abhängigkeiten werden nun nach der Anzahl der aufgerufenen Services aufsteigend sortiert. Auf Basis dieser Sortierung werden die Abhängigkeiten in Gruppen eingeteilt, wobei folgende Bedingungen gelten müssen:

a) Eine Abhängigkeit darf nur einer Gruppe zugeordnet werden.
b) Alle Abhängigkeiten in einer Gruppe müssen in Bezug auf die Eigenschaft kleiner sein als die Abhängigkeiten in den nachfolgenden Gruppen und größer als die Abhängigkeiten in den vorangegangenen Gruppen.
c) Die Gruppen sollen in etwa gleich groß sein.

Die in Punkt a) gestellte Forderung ist notwendig, da sonst die Ergebnisse verfälscht werden. Punkt b) fordert, dass in zwei Gruppen keine Abhängigkeiten enthalten sind, die die gleiche Ausprägung einer Eigenschaft besitzen. Beispielsweise müssen zwei Abhängigkeiten, die die gleiche Anzahl Services aufrufen, in der gleichen Gruppe sein. Die letzte Forderung c) ermöglicht es, die Fehleranzahl der Abhängigkeiten in den Gruppen besser vergleichen zu können. Für die Einteilung der Abhängigkeiten in Gruppen in Bezug auf eine Eigenschaft werden Quantile (bzw. Quantilgruppen) verwendet, da diese die drei Forderungen a) bis c) erfüllen. In wie viele Gruppen die Eigenschaften eingeteilt werden müssen, hängt von der Größe des untersuchten Softwaresystems und von der erzielten Aussagekraft der zu gewinnenden Ergebnisse ab. Je mehr Gruppen gebildet werden, desto feiner kann die Unterscheidung zwischen den Abhängigkeiten gemäß der betrachteten Eigenschaft gemacht werden. Je größer jedoch die Anzahl der Gruppen, desto weniger Abhängigkeiten stehen für die Korrelationsanalyse innerhalb dieser Gruppe zur Verfügung und umso weniger statistische Aussagekraft haben die Ergebnisse. D.h. desto weniger wahrscheinlich ist es, statistisch signifikante Ergebnisse zu erzielen. In einem sehr großen Softwaresystem kann eine große Anzahl an Gruppen gewählt werden. In der Fallstudie des Softwaresystems Eclipse (vgl. Kapitel 5.4.1) standen zwischen 5.000 und 10.000 Bausteine für die Untersuchung zur Verfügung. Aus diesem Grund wurden Dezilgruppen gebildet. In viel kleineren Systemen bieten sich Quartilgruppen an. In viel größeren Systemen als das Eclipse System können auch eine größere Anzahl Gruppen erstellt werden.
Eine Abhängigkeit kann sich in Bezug auf eine Eigenschaft nur in einer Gruppe befinden. Da der Ansatz aber vorsieht, dass eine Abhängigkeit mehrere Eigenschaften besitzen kann, muss die Abhängigkeit für jede ihrer Eigenschaften in Gruppen eingeteilt werden. Für

unterschiedliche Eigenschaften kann eine Abhängigkeit somit in unterschiedlichen Gruppen sein. Zum Beispiel kann sich eine Abhängigkeit hinsichtlich der Eigenschaft „Anzahl aufgerufener Services“ in der Gruppe mit den höchsten Ausprägungen befinden und hinsichtlich der Eigenschaft „Anzahl direkter Attributzugriffe“ nur in der dritten Gruppe.

5.3.1.2 Durchführung der Korrelationsanalyse

Nachdem für eine Version eines Softwaresystems die Fehleranzahl ermittelt, die Abhängigkeiten identifiziert und ihre Eigenschaften erhoben sowie die Abhängigkeiten gruppiert worden sind, kann die Korrelationsanalyse durchgeführt werden (Schritt 2.4). Es werden Zusammenhänge zwischen einer Abhängigkeitseigenschaft und der Fehleranzahl des abhängigen Bausteins und/oder mit der Fehleranzahl des unabhängigen Bausteins aufgedeckt.
Definition: Eine Korrelation zwischen einer Abhängigkeitseigenschaft und der Fehleranzahl liegt vor, wenn (1) es eine Menge von Abhängigkeiten (mit der gleichen Eigenschaft) gibt, deren Fehleranzahl höher ist als die Fehleranzahl in allen anderen Abhängigkeiten und (2) sie sich statistisch signifikant (vgl. Kapitel 5.1, Seite 72) von allen anderen Abhängigkeiten bezüglich der Fehleranzahl unterscheidet.

Für die Korrelationsanalyse können verschiedene Verfahren verwendet werden. Um die besten Verfahren für die Identifikation von Korrelationen, wie sie oben definiert worden sind, herauszufiltern, müssen folgende Kriterien erfüllt werden:

Erkennen von statistisch signifikanten Zusammenhängen
Für die Testfokusauswahl sind nur die Eigenschaften interessant, die einen Zusammenhang mit der Fehleranzahl aufweisen. Diese Zusammenhänge müssen aber statistisch abgesichert sein, d.h. sie müssen auf einem vorgegebenen Signifikanzniveau (z.B. 5%) Gültigkeit haben.

Richtung und Stärke des Zusammenhangs
Um die aufgedeckten Zusammenhänge für die Testfokusauswahl nutzen zu können, muss die Richtung der Zusammenhänge bekannt sein. Die Richtung gibt einen positiven oder einen negativen Zusammenhang zwischen abhängiger und unabhängiger Variable an. Bei einem positiven Zusammenhang steigt der Wert der abhängigen Variablen, wenn der Wert der unabhängigen Variablen steigt. Bei einem negativen Zusammenhang ist es umgekehrt.
Die Stärke eines Zusammenhangs beschreibt, wie eng die abhängige Variable von der unabhängigen Variablen abhängt, d.h. in wie vielen Fällen der Stichprobe der beobachtete Zusammenhang existiert.

Unregelmäßige Zusammenhänge
Ein Zusammenhang zwischen der abhängigen und der unabhängigen Variablen kann regelmäßig und unregelmäßig sein. Ein Zusammenhang ist regelmäßig, wenn mit steigender Ausprägung der unabhängigen Variablen auch die abhängige Variable steigt. Ein Spezialfall des regelmäßigen Zusammenhangs ist der lineare Zusammenhang, d.h. die abhängige Variable steigt oder sinkt immer im gleichen Maß wie die unabhängige Variable. Ein Beispiel für einen regelmäßigen Zusammenhang ist in Abbildung 17 (Seite 85) dargestellt.
Ein unregelmäßiger Zusammenhang existiert, wenn eine Menge von Abhängigkeiten mit der gleichen Ausprägung der betrachteten Eigenschaft die höchste Fehleranzahl aufweist, obwohl die Werte dieser Eigenschaft nicht zu den höchsten Werten dieser Eigenschaft gezählt werden können. Abbildung 18 (Seite 85) zeigt ein Beispiel eines unregelmäßigen Zusammenhangs.

Ein statistisches Verfahren muss in der Lage sein, auch unregelmäßige Zusammenhänge zu erkennen und sie für die Testfokusauswahl zur Verfügung zu stellen.

Verteilungsfreiheit des verwendeten statistischen Verfahrens
Einige der vorgestellten statistischen Verfahren können nur verwendet werden, wenn bestimmte Annahmen über die Verteilung der Werte der abhängigen und/oder unabhängigen Variablen eingehalten werden. Nach [FO00] und [OW07] befinden sich 80% der Fehler in ca. 20% der Bausteine, wodurch eine Gleich-, aber auch eine Normalverteilung ausgeschlossen werden können. Somit muss das verwendete statistische Verfahren verteilungsfrei sein.

Überzeugungskraft der Ergebnisse
In der Statistik ist es schwierig, die Ergebnisse einer Analyse zu interpretieren. Noch schwerer ist es, sie anderen überzeugend darzulegen. Idealerweise können die Ergebnisse in Form von Grafiken oder Diagrammen dargestellt werden, die sich auf einfache Weise lesen und interpretieren lassen. Diese können verwendet werden, um sie dem Projektmanager, dem Kunden oder den Entwicklern vorzulegen.

Einfache Verwendbarkeit für die Testfokusauswahl
Die Ergebnisse der Korrelationsanalyse für frühere Versionen müssen sich leicht auf die Testfokusauswahl der aktuellen Version übertragen lassen. Dies ist dann der Fall, wenn keine größeren Berechnungen für die aktuelle Version mehr notwendig sind, um den Testfokus festzulegen. Idealerweise liefert das Ergebnis des eingesetzten statistischen Verfahrens eine einfache Berechnungsvorschrift, um die für den Testfokus auszuwählenden Abhängigkeiten festzulegen.

Mit Hilfe der definierten Kriterien können die in 5.1 genannten statistischen Verfahren auf ihre Eignung zur Testfokusauswahl bewertet werden. Das Ergebnis dieser Bewertung ist in Tabelle 4 zusammengefasst. Für jedes Verfahren (Zeilen) wird angegeben, wie gut es die einzelnen Kriterien (Spalten) erfüllt. Dabei wurden als mögliche Bewertungen „**++**“, „**+**“, „**-**“ und „**--**“, wobei „**++**“ für „*erfüllt das Kriterium sehr gut*“ und „**--**" für „*erfüllt das Kriterium überhaupt nicht*“ steht. Da alle genannten statistischen Verfahren das erste Kriterium erfüllen (Erkennen von statistisch signifikanten Zusammenhängen), wurde es nicht mit in die Tabelle aufgenommen.

Tabelle 4: Bewertung der statistischen Verfahren für die Eignung zur Testfokusauswahl

	Richtung und Stärke	Verteilungs-freiheit	Unregelmäßige Zusammenhänge	Überzeugungs-kraft	Verwend-barkeit
Chi-Quadrat-Test	**--**	**++**	**++**	**--**	**--**
Kruskal und Goodmann tau	**--**	**++**	**++**	**--**	**--**
Rangkorrelation nach Spearman	**++**	**++**	**--**	**-**	**-**
Korrelationskoeffizient nach Pearson	**++**	**--**	**--**	**-**	**-**
Univariate lineare Regression	**+**	**++**	**--**	**-**	**+**
ANOVA	**+**	**--**	**++**	**++**	**+**
Kruskal-Wallis-H Test	**+**	**++**	**++**	**+**	**+**

Die Richtung des Zusammenhangs wird vom Chi-Quadrat-Test und vom Test von Kruskal und Goodman tau nicht angegeben. Ersterer gibt nur an, ob es ein Zusammenhang gibt. Da

letzterer nur auf nominalskalierte Variablen angewendet werden kann, gibt er nur Auskunft über die Stärke des Zusammenhangs. Die Richtung des Zusammenhangs wird durch die verbleibenden Verfahren in Tabelle 4 angegeben. Eine exakte Angabe der Stärke des Zusammenhangs ist jedoch nur durch den Rangkorrelation nach Spearman und den Korrelationskoeffizient nach Pearson möglich. Das Ergebnis beider Tests ist eine Zahl zwischen „-1“ und „1“. Je stärker sich der ermittelte Wert an „-1“ bzw. „1“ annähert, desto stärker ist der Zusammenhang zwischen der abhängigen und der unabhängigen Variable. Die Regressionen und die Mittelwertvergleiche können solche Informationen nicht liefern.
Eine der Eigenschaften, die das ausgewählte statistische Verfahren erfüllen muss, ist die Verteilungsfreiheit. Da die Fehler nicht normal- oder gleichverteilt vorliegen, können statistischen Verfahren, die eine solche Verteilung voraussetzen, nicht eingesetzt werden. Davon sind der Korrelationskoeffizient nach Pearson und die einfaktorielle Varianzanalyse (ANOVA) betroffen, d.h. sie setzen eine normal bzw. Gleichverteilung voraus. Die verbleibenden Tests sind parameterfrei und können somit zum Einsatz kommen.
Das ausgewählte statistische Verfahren muss darüber hinaus in der Lage sein, auch unregelmäßige Zusammenhänge aufzudecken. Dies ermöglichen der Chi-Quadrat-Test, der Test von Kruskal und Goodmann tau, die einfaktorielle Varianzanalyse (ANOVA) und der Kruskal-Wallis-H Test. Die drei verbleibenden Tests sind dafür konzipiert, lineare Zusammenhänge zwischen der abhängigen und der unabhängigen Variable aufzudecken.
Die Ergebnisse der Korrelationsanalyse müssen überzeugend sein, d.h. sie müssen in einer einfachen Form präsentiert und schnell vom Betrachter verstanden werden können. Hierfür bietet sich die einfaktorielle Varianzanalyse (ANOVA) an, da sie auf den Mittelwerten der Fehleranzahl beruht und der Mittelwert eine statistische Maßzahl ist, die von der Mehrzahl der Personen verstanden wird. Die Ergebnisse der ANOVA können sehr einfach als Diagramm dargestellt werden, wobei die Abszisse die (gruppierten) Werte der unabhängigen Variablen und die Ordinate den Mittelwert anzeigt. Aus diesem Diagramm kann sehr schnell herausgelesen werden, welche Gruppe den höchsten Fehlerdurchschnitt aufweist und um wie viel höher dieser Durchschnitt im Vergleich zu anderen Gruppen ist. Da der Kruskal-Wallis-H Test mit den mittleren Rängen und nicht mit dem Mittelwert arbeitet, kann er nur Informationen über die Gruppe mit dem durchschnittlich höchsten Rang liefern. Er kann jedoch keine Aussagen über die Höhe der Unterschiede zwischen den Gruppen machen. Die verbleibenden statistischen Verfahren sind weniger überzeugend, da sie häufig nur eine oder zwei Maßzahlen liefern, die die Stärke des Zusammenhanges angeben. Diese Zahl ist für Betrachter, die mit den statistischen Verfahren nicht sehr vertraut sind, nicht intuitiv verständlich. Weiterhin lassen sich diese Maßzahlen nicht grafisch aufbereiten.
Das letzte Kriterium für die Auswahl eines geeigneten statistischen Verfahrens ist seine Eignung für die spätere Testfokusauswahl. Ziel muss es sein, ein Verfahren zu verwenden, das es ermöglicht, aus den gewonnenen Ergebnissen die Abhängigkeiten der zu testenden Version eines Softwaresystems auszuwählen. Die ersten beiden Verfahren in der Tabelle sind hierfür nicht geeignet, da sie nur angeben, ob es einen Zusammenhang gibt. Der Rangkorrelationskoeffizient nach Spearman und der Korrelationskoeffizient nach Pearson geben an, ob ein linearer Zusammenhang vorliegt. Insofern ein entsprechend starker Zusammenhang vorliegt, müssen für die zu testende Version des Softwaresystems die Abhängigkeiten ausgewählt werden, die hohe Werte in der untersuchten unabhängigen Variablen aufweisen. Die Schwierigkeit stellt hierbei die Definition eines Schwellwertes dar. Zum einen ist ein Schwellwert notwendig, um einen starken Zusammenhang zu definieren. Zum anderen muss ein Schwellwert definiert werden, der angibt, ab welchem Wert der unabhängigen Variable eine Abhängigkeit als Testfokus ausgewählt wird. Daher eignen sich die statistischen Verfahren der Regression am besten für die Testfokusauswahl. Die ermittelte lineare Gleichung erlaubt es, die Fehleranzahl in den beteiligten Bausteinen anhand der unabhängigen Variablen vorherzusagen. Eine Abhängigkeit für die zu testende

Version wird zum Testfokus hinzugefügt, wenn die vorhergesagte Fehleranzahl einen bestimmten Schwellwert überschreitet. Die Definition des Schwellwerts stellt hierbei eine Herausforderung dar. Die Ergebnisse des Mittelwertvergleiches (ANOVA, Kruskal-Wallis-H Test) bieten die beste Möglichkeit, die Ergebnisse für die Testfokusauswahl zu verwenden. Beide Verfahren setzen voraus, dass die Abhängigkeiten nach der unabhängigen Variablen gruppiert werden. Diese Gruppen können anschließend für die zu testende Version gebildet werden. Es wird die Gruppe ausgewählt, die in den früheren Versionen den höchsten Mittelwert bzw. den höchsten mittleren Rang aufgewiesen hat.

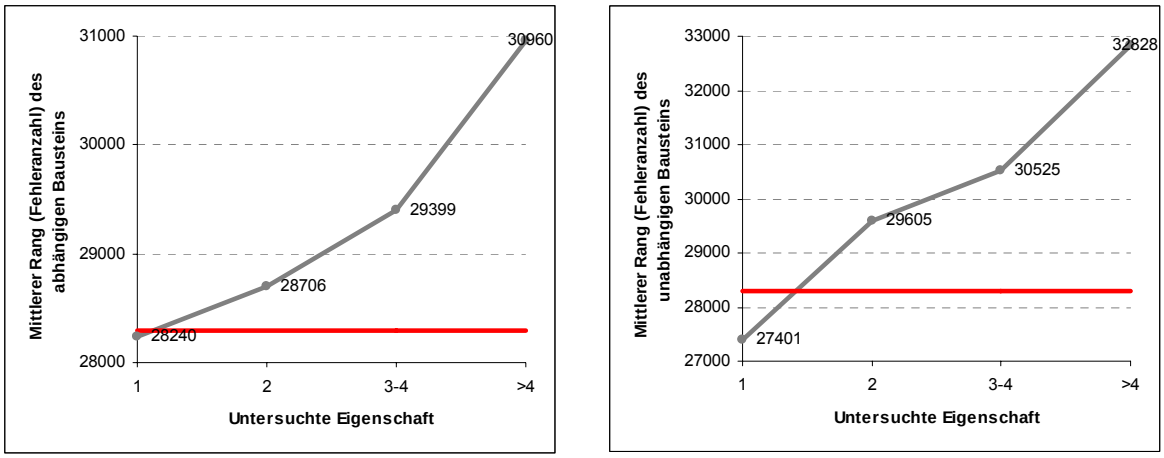


Abbildung 17: Beispiel einer regelmäßigen Korrelation in Eclipse 2.0

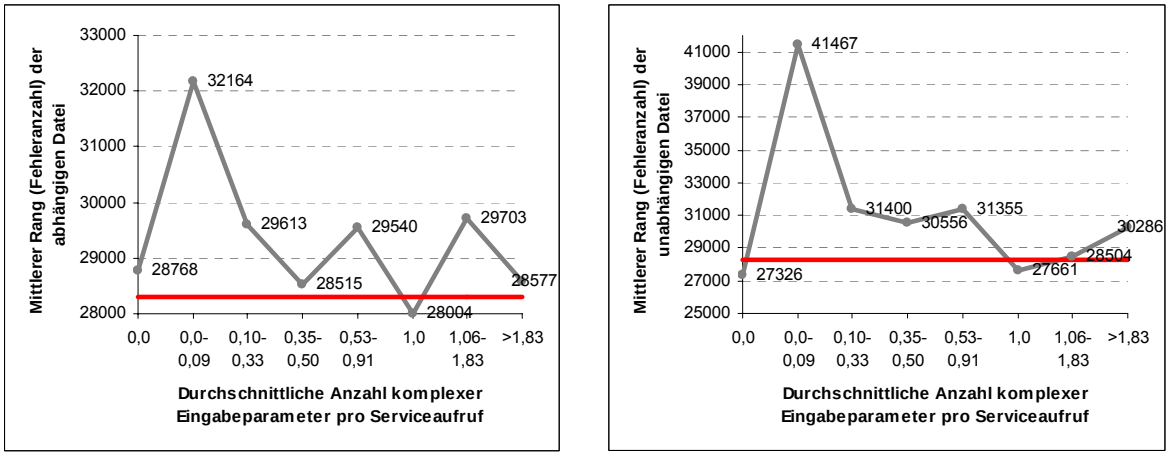


Abbildung 18: Beispiel einer unregelmäßigen Korrelation in Eclipse 2.0

Aus den Bewertungen der verschiedenen statistischen Verfahren in Tabelle 4 kann entnommen werden, dass der Kruskal-Wallis-H Test am besten für das Identifizieren von Zusammenhängen zwischen Abhängigkeitseigenschaften und der Fehleranzahl in den beteiligten Bausteinen verwendet werden kann. Dieser Test wird verwendet, um den Zusammenhang zwischen **einer** Abhängigkeitseigenschaft und der Fehleranzahl im abhängigen bzw. der Fehleranzahl im unabhängigen Baustein zu untersuchen. Dies hat zur Folge, dass für jede zu untersuchende Abhängigkeitseigenschaft der Kruskal-Wallis-H Test **zweimal** durchgeführt werden muss. Die Abhängigkeiten werden nach der zu untersuchenden Eigenschaft gruppiert. Anschließend wird für jede Gruppe der mittlere Rang ermittelt. Die Ergebnisse des Kruskal-Wallis-H Tests lassen sich mit Hilfe von Diagrammen graphisch darstellen. Beispiele für Ergebnisse der Korrelationsanalyse zwischen einer Abhängigkeitseigenschaft und der Fehleranzahl im abhängigen bzw. im unabhängigen Baustein sind in Abbildung 17 und Abbildung 18 dargestellt. Die linke Seite der Abbildungen

enthält jeweils die Ergebnisse der Korrelationsanalyse zwischen Eigenschaft und abhängigem Baustein und die rechte Seite die Ergebnisse für die Eigenschaft und dem unabhängigen Baustein. In den Diagrammen stellt die Abszisse die Gruppen der Eigenschaft und deren Wertebereiche dar. Im Beispiel in Abbildung 17 enthält die erste Gruppe alle Abhängigkeiten, deren Wert der Eigenschaft gleich „1“ ist und die zweite Gruppe den Wert gleich „2“. In der dritten Gruppe befinden sich alle Abhängigkeiten deren Ausprägung der Eigenschaft gleich „3“ oder „4“ haben. In beiden Diagrammen repräsentiert die Ordinate den mittleren Rang (nach der Fehleranzahl). In beiden Diagrammen sind jeweils zwei Linien eingezeichnet. Die graue Linie stellt den mittleren Rang der einzelnen Gruppen dar. Für jede Gruppe sind die jeweiligen Werte der Gruppe an den Datenpunkten gekennzeichnet. Im Beispiel für die Untersuchung des Zusammenhangs zwischen Eigenschaft und der Fehleranzahl im abhängigen Baustein (linkes Diagramm) ist der mittlere Rang für die letzte Gruppe 30960. Die rote Linie in den Diagrammen repräsentiert den mittleren Rang aller Abhängigkeiten über alle Gruppen hinweg. Es ist deutlich zu erkennen, dass in beiden Diagrammen die letzten Gruppen deutlich über dem Fehlerdurchschnitt aller Bausteine liegen.
Neben der Ermittlung des mittleren Rangs für die einzelnen Gruppen berechnet der Kruskal-Wallis-H Test einen Signifikanzwert. Mit Hilfe dieses Signifikanzwerts wird ermittelt, ob sich mindestens zwei Gruppen in Bezug auf ihren mittleren Rang signifikant unterscheiden. Der Test gibt keine Auskunft darüber, welche Gruppen sich signifikant voneinander unterscheiden. Um diese Gruppen zu ermitteln, muss der Mann-Whitney-U Test verwendet werden. Für je zwei Gruppen wird geprüft, ob sie sich signifikant voneinander unterscheiden. Hierfür wird ein angestrebtes Signifikanzniveau (z.B. 5%) festgelegt und anschließend geprüft ob der errechnete Signifikanzwert kleiner als das angestrebte Niveau ist.
Sowohl der Kruskal-Wallis-H Test als auch der Mann-Whitney-U Test können mit Hilfe des Statistikwerkzeugs SPSS [SP09] durchgeführt werden.
Nachdem beschrieben worden ist, welche statistischen Verfahren für die Korrelationsanalyse verwendet werden, wird nachfolgend die Durchführung der Korrelationsanalyse für eine zu untersuchende Version des Softwaresystems dargestellt. (Abbildung 19).

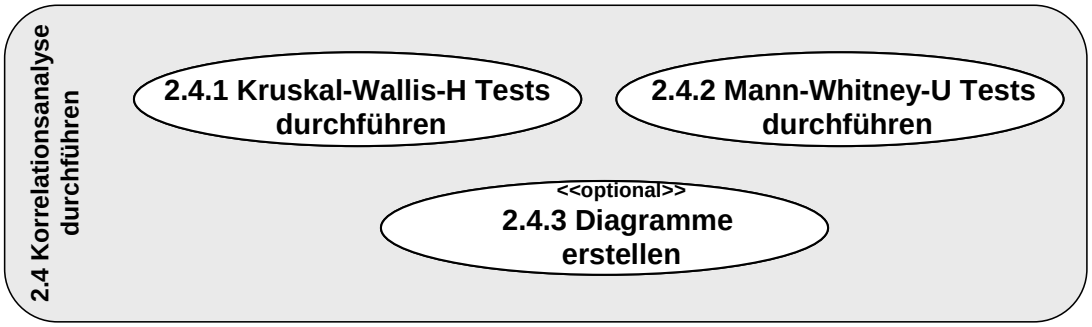


Abbildung 19: Teilschritte zur Durchführung der Korrelationsanalyse

Ausgangspunkt ist die in den Schritten 2.1.-2.3. (vgl. Abbildung 16) erstellte Tabelle, welche die identifizierten Abhängigkeiten, ihre Eigenschaften und die Fehleranzahl der beteiligten Bausteine pro Abhängigkeit enthält. Aufbauend auf dieser Tabelle wird für jede Eigenschaft der Kruskal-Wallis-H Test durchgeführt. Die unabhängige Variable stellt dabei die aktuell untersuchte Eigenschaft dar. Die Abhängigkeiten wurden zuvor in Schritt 2.3. nach dieser Eigenschaft gruppiert, so dass die Eigenschaft als ordinalskalierte Variable vorliegt. Die abhängige Variable stellt die Fehleranzahl der beteiligten Bausteine dar. Hierbei wird unterschieden, ob ein Zusammenhang mit der Fehleranzahl des abhängigen oder der Fehleranzahl des unabhängigen Bausteins untersucht wird. Da das Vorgehen für beide Fälle identisch ist, wird nachfolgend die Analyse für den Zusammenhang zwischen der

Eigenschaft und der Fehleranzahl anhand des abhängigen Bausteins erklärt. Die Analysen selbst müssen jedoch sowohl für den abhängigen als auch für den unabhängigen Baustein durchgeführt werden.

Als Eingabe für den Kruskal-Wallis-H Tests dienen die gruppierten Abhängigkeiten nach der untersuchten Eigenschaft (als unabhängige Variable) und die Fehleranzahl des abhängigen Bausteins (als abhängige Variable). Der Kruskal-Wallis-H Test liefert als wichtigstes Ergebnis einen Signifikanzwert. Dieser gibt an, ob sich mindestens zwei Gruppen der unabhängigen Variable signifikant voneinander unterscheiden. Ist dies nicht der Fall, können keine statistisch abgesicherten Aussagen über mögliche Zusammenhänge getroffen werden. Ist der ermittelte Signifikanzwert des Kruskal-Wallis-H Tests kleiner als das zuvor festgelegt Signifikanzniveau wird im Schritt 2.4.2 der Mann-Whitney-U Test durchgeführt. Er dient zum Identifizieren derjenigen Gruppen, die sich bezüglich der Fehleranzahl voneinander unterscheiden. Der Mann-Whitney-U Test wird auf zwei Gruppen angewendet und ermittelt einen Signifikanzwert. Dieser Wert kann verwendet werden, um festzustellen, ob die Unterschiede zwischen beiden Gruppen bezüglich der Fehleranzahl statistisch signifikant sind. Sie sind statistisch signifikant, wenn sie unterhalb eines zu erreichenden Signifikanzniveaus liegen.
Der Mann-Whitney-U Test muss für jede paarweise Kombination der Gruppen durchgeführt werden. Dies wären bei **n** Gruppen **n*(n-1)** Mann-Whitney-U Tests, die durchgeführt werden müssten. Da der ermittelte Signifikanzwert des Mann-Whitney-U Tests jedoch zweiseitig ist, d.h. wenn ein Zusammenhang zwischen Gruppe A und B besteht, dann besteht auch ein Zusammenhang zwischen Gruppe B und Gruppe A, kann die Anzahl der notwendigen Test halbiert werden: **n*(n-1)/2**. Dies wären bei n=10 (Dezilgruppen) 45 notwendige Mann-Whitney-U Tests. Werden dieses Tests per Hand ausgeführt (d.h. sie müssen in einem Statistik-Werkzeug per Hand angestoßen werden) entstünde ein sehr hoher Aufwand. Für die spätere Testfokusauswahl sind jedoch nur die Gruppen von Interesse, die den höchsten mittleren Rang aufweisen, da dies die Gruppen sind, die auf eine hohe Fehleranzahl hinweisen. Aus diesem Grund kann die Anzahl der notwendigen Mann-Whitney-U Tests durch folgende zwei Regeln eingeschränkt werden:

Maximumregel
Bei der Untersuchung des Zusammenhangs einer Eigenschaft mit der Fehleranzahl ist nur die Gruppe interessant, die den maximalen mittleren Rang aufweist. In dieser Gruppe ist die Wahrscheinlichkeit am größten, einen Fehler im abhängigen bzw. unabhängigen Baustein und somit in der Abhängigkeit zu finden.

Überdurchschnittlichkeitsregel
Bei der Untersuchung des Zusammenhangs einer Eigenschaft mit der Fehleranzahl werden die Gruppen berücksichtigt, die über dem durchschnittlichen mittleren Rang aller Abhängigkeiten liegen. Dies sind alle Gruppen, deren mittlerer Rang oberhalb der roten Linie in Abbildung 17 liegt.

Durch beide Regeln kann die Anzahl der notwendigen Mann-Whitney-U Tests weiter reduziert werden. Greift die Überdurchschnittlichkeitsregel nicht, d.h. der mittlere Rang aller Gruppen liegt unter dem mittleren Rang aller Abhängigkeiten, dann sind keine weiteren Tests notwendig, da diese Eigenschaft nicht für die Testfokusauswahl herangezogen wird. Gibt es hingegen Gruppen, deren mittlerer Rang über dem mittleren Rang aller Abhängigkeiten liegt, dann wird nur die Gruppe ausgewählt, deren mittlerer Rang die größte Ausprägung aufweist (Maximumregel). Nur für diese Gruppe wird mit Hilfe des Mann-

Whitney-U Tests überprüft, ob sie sich signifikant von den anderen Gruppen unterscheidet. Somit sind nur noch (n-1) Tests durchzuführen.
Zur besseren Veranschaulichung der Ergebnisse des Kruskal-Wallis-H Tests können im optionalen Schritt 2.4.3 die mittleren Ränge der Gruppen in ein Diagramm eingetragen werden (vgl. Abbildung 17 und Abbildung 18). Dort kann abgelesen werden, ob es einen Zusammenhang zwischen der unabhängigen und der abhängigen Variable gibt. In Abbildung 17 ist zu erkennen, dass die untersuchte Eigenschaft mit steigendem Wert der unabhängigen Variablen einen höheren mittleren Rang aufweist. Der Kruskal-Wallis-H Test hat einen Signifikanzwert von 0,00 für beide Untersuchungen (abhängiger und unabhängiger Baustein) ergeben, so dass sich mindestens zwei Gruppen in Abbildung 17 signifikant voneinander unterscheiden (da der Signifikanzwert deutlich unter den angestrebten 5% liegt). Für die Beispieluntersuchung in Abbildung 17 wurde der Mann-Whitney-U Test für alle paarweisen Kombinationen durchgeführt. Die Ergebnisse sind in Tabelle 5 dargestellt.
Die ersten beiden Spalten enthalten die Beschriftung der untersuchten Gruppen. Jeweils in einer Zeile kann anhand der ersten zwei Spalten erkannt werden, für welche zwei Gruppen der Mann-Whitney-U Test durchgeführt worden ist. Die dritte Spalte enthält den ermittelten Signifikanzwert des Mann-Whitney-U Tests für die Fehleranzahl des abhängigen Bausteins, die letzte Spalte für den unabhängigen Baustein. Es ist zu erkennen, dass sich alle Gruppen signifikant voneinander unterscheiden, da der ermittelte Signifikanzwert deutlich unterhalb der festgelegten 5% (=0,05) liegt. Die Maximumregel hätte die Anzahl der notwendigen Tests auf drei reduzieren können. Die Tests wären dann nur noch für die Paarungen „1“ und „>4“, „2“ und „>4“ sowie „3-4“ und „>4“ notwendig gewesen. Für alle drei Paare zeigt sich, dass die Gruppe „>4“ mit dem größten mittleren Rang sich signifikant von den anderen Gruppen unterscheidet.

Tabelle 5: Ergebnis der Signifikanztests für die Unterschiede zwischen den Gruppen

Gruppe 1	Gruppe 2	Mann-Whitney-U Test abhängiger Baustein	Mann-Whitney-U Test unabhängiger Baustein
1	2	0,011982	0,0
1	3-4	0,0	0,0
1	>4	0,0	0,0
2	3-4	0,005662	0,000076
2	>4	0,0	0,0
3-4	>4	0,000001	0,0

Die Korrelationsanalysen werden für die ausgewählten Eigenschaften sowohl für die Fehleranzahl des abhängigen als auch für die Fehleranzahl des unabhängigen Bausteins durchgeführt. Für die spätere Testfokusauswahl ist es wichtig, mehrere Versionen eines Systems zu untersuchen, um Zusammenhänge aufzudecken, die über mehrere Versionen Bestand haben.

Anmerkung zur statistischen Gültigkeit
(1) Für die Überprüfung der statistischen Signifikanz der Unterschiede zwischen zwei Gruppen wird der Mann-Whitney-U Test verwendet. Durch die Maximum- und die Überdurchschnittlichkeitsregel müssen (n-1) Tests durchgeführt werden. In [Ab07] wird dargestellt, dass das mehrfache Anwenden von Paartests, wie die Mann-Whitney-U Tests die Gefahr von Typ I Fehlern (vgl. [Ab07]) steigt. Um die Gefahr dieser Typ I Fehler zu reduzieren muss die Bonnferroni-Korrektur (vgl. [Ab07]) angewendet werden. Hierzu wird der zu erreichende Signifikanzwert durch die Anzahl auszuführender Mann-Whitney-U Tests geteilt. Da für eine Eigenschaft n-1 Mann-Whitney-U Tests durchgeführt werden, wenn die Maximum- und Überdurchschnittlichkeitsregeln greifen, muss der zu erreichende

Signifikanzwert durch n-1 geteilt werden. Wird zum Beispiel ein Signifikanzwert von 0,05 angestrebt, so muss dieser Wert bei zehn Gruppen durch neun geteilt werden. Dies hat zur Folge, dass der Signifikanzwert, der durch den Mann-Whitney-U Test berechnet wird, den Wert 0,0055 (`= 0,5/(10-1)`) nicht überschreiten darf.
(2) Bei der Korrelationsanalyse zwischen einer Eigenschaft und der Fehleranzahl im abhängigen[1] Baustein ist zu bedenken, dass ein Baustein in mehreren Abhängigkeiten als abhängiger Baustein beteiligt sein kann. Dies hat zur Folge, dass die Fehleranzahl dieses Bausteins mehrfach in die Korrelationsanalyse auftaucht. Dies hat insbesondere auf die Berechnung der Fehlermittelwerte innerhalb der Gruppen Einfluss. Dieses Problem tauchte erstmals bei Versuchen mit der einfaktoriellen Varianzanalyse (ANOVA) auf. Die berechneten Mittelwerte der Bausteine lagen deutlich höher als die Fehlermittelwerte aller Bausteine. Bei den Kruskal-Wallis-H und den Mann-Whitney-U Tests wird dieses Problem umgangen, da nur die Rangwerte verwendet werden. Darüber hinaus werden die, durch die Tests ermittelten mittleren Ränge nur verwendet, um eine Tendenz abzuleiten, d.h. ob bestimmte Eigenschaften auf eine erhöhte Anzahl Fehler im abhängigen Baustein hindeuten. Sie werden nicht verwendet, um sie mit Mittelwerten aller Bausteine zu vergleichen.

5.3.2. Festlegung des Testfokus

Nachdem für mehrere Versionen des Softwaresystems die Korrelationsanalysen durchgeführt worden sind, kann der Testfokus für die zu testenden Version ausgewählt werden. Das Ziel der Korrelationsanalyse ist es, Eigenschaften zu identifizieren, die mit der Fehleranzahl in den beteiligten Bausteinen korrelieren. Dabei kann unterschieden werden, ob eine Eigenschaft mit der Fehleranzahl im abhängigen, mit der Fehleranzahl im unabhängigen oder mit der Fehleranzahl in beiden Bausteinen korreliert. Ein Beispiel für eine beidseitige Korrelation ist in Abbildung 17 zu sehen. Mit steigenden Werten der untersuchten Eigenschaft steigt auch der mittlere Rang der Fehleranzahl des abhängigen und des unabhängigen Bausteins. Diese Informationen werden für die Testfokusauswahl ausgenutzt. Von besonderem Interesse sind die Gruppen der Eigenschaften, die den höchsten mittleren Rang aufweisen und sich signifikant von allen anderen Gruppen unterscheiden.
Im ersten Schritt müssen die Abhängigkeiten und ihre Eigenschaften für die zu testende Version identifiziert werden (vgl. 3.1 in Abbildung 16). Im Anschluss werden die Eigenschaften nach dem gleichen Schema wie in Schritt 2.3 gruppiert, d.h. wenn die Eigenschaften für die Korrelationsanalyse in Dezile (10er Gruppen) eingeteilt worden sind, müssen die Eigenschaften der zu testenden Version ebenfalls in Dezile eingeteilt werden (3.2). Abschließend kann der Testfokus für die zu testende Version ausgewählt werden.
Eine Abhängigkeit wird als Testfokus ausgewählt, wenn sie 1) eine Eigenschaft besitzt, die mit der Fehleranzahl korreliert und 2) der Wert der Eigenschaft in die Gruppe mit dem höchsten mittleren Rang fällt (ermittelt aus den früheren Versionen). Auf diese Weise kann eine Einteilung vorgenommen werden, dass eine Abhängigkeit getestet werden muss oder nicht. In [BP09a] wird darüber hinaus der Ansatz erweitert, indem ein zusätzlicher Schwellwert eingefügt wird. Dieser Schwellwert wird so gewählt, dass er über dem Mittleren Rang aller Abhängigkeiten liegt. Er dient dazu, Eigenschaften, die zwar mit der Fehleranzahl korrelieren, aber deren mittlerer Rang nur knapp über dem mittleren Rang aller Abhängigkeiten liegt, herauszufiltern. Es werden nur die Eigenschaften für die Testfokusauswahl verwendet, die über dem festgelegten Schwellwert liegen.
Wie im vorangegangenen Kapitel beschrieben, wird mit der Korrelationsanalyse untersucht, ob eine Eigenschaft mit der Fehleranzahl im abhängigen und/oder im unabhängigen Baustein korreliert. Diese Einteilung kann für die Testfokusauswahl ausgenutzt werden und

[1] Nachfolgende Überlegungen sind identisch zur Betrachtung der Fehleranzahl im unabhängigen Baustein

fließt in die Festlegung einer Testpriorität für jede Abhängigkeit mit ein. Aus den Informationen lassen sich vier Kategorien für die Testpriorität festlegen:

Keine Testpriorität: Eine Abhängigkeit bekommt *keine Testpriorität*, wenn sie keine Eigenschaft besitzt, die mit der Fehleranzahl des abhängigen oder des unabhängigen Bausteins korreliert oder Eigenschaften besitzt, die zwar mit der Fehleranzahl korrelieren, aber nicht in die Gruppen mit den höchsten mittleren Rang eingeordnet werden können.

Testpriorität-abhängiger Baustein: Einer Abhängigkeit wird die Kategorie *Testpriorität-abhängiger Baustein* zugeordnet, wenn sie mindestens eine Eigenschaft besitzt, die mit der Fehleranzahl des abhängigen Bausteins korreliert und sich die Abhängigkeit für diese Eigenschaft in die Gruppe mit dem höchsten mittleren Rang einordnen lässt. Darüber hinaus besitzt diese Abhängigkeit keine Eigenschaften, die mit der Fehleranzahl des unabhängigen Bausteins korreliert bzw. wenn sie korrelierende Eigenschaften (in Bezug auf den unabhängigen Baustein) besitzt, lässt sie sich in Bezug auf diese Eigenschaften nicht in die Gruppen mit den höchsten mittleren Rang einordnen.

Testpriorität-unabhängiger Baustein: Diese Testpriorität wird allen Abhängigkeiten zugeordnet, die mindestens eine Eigenschaft besitzen, die mit der Fehleranzahl des unabhängigen Bausteins korreliert und sich in Bezug auf diese Eigenschaft in die Gruppe mit dem höchsten mittleren Rang einordnen lassen. Darüber hinaus besitzen sie keine Eigenschaften, die mit der Fehleranzahl im abhängigen Baustein korrelieren bzw. lassen sich in Bezug auf eine korrelierende Eigenschaft nicht in die Gruppe mit dem höchsten mittleren Rang einordnen.

Testpriorität-beide Bausteine: Diese Testpriorität wird allen Abhängigkeiten zugeordnet, die mindestens eine Eigenschaft besitzen, die mit der Fehleranzahl des abhängigen und des unabhängigen Bausteins korrelieren und sich in Bezug auf diese Eigenschaften in die Gruppe mit dem höchsten mittleren Rang eingruppieren lassen.

Diese Einteilung der Abhängigkeit nach der Testpriorität ermöglicht es, die vorhandenen Ressourcen auf eine ausgewählte Menge von Abhängigkeiten zu verteilen. Abhängigkeiten, mit der *Testpriorität-beide Bausteine* sollten intensiver getestet werden als Abhängigkeiten mit *Testpriorität-unabhängiger Baustein* bzw. *Testpriorität-abhängiger Baustein*.
Darüber hinaus geben die Prioritäten Hinweise darauf, wo sich die Fehler voraussichtlich befinden werden (im abhängigen oder im unabhängigen Baustein.

5.4. Fallstudien

Das in Kapitel 5.3 vorgestellte Vorgehen zur Testfokusauswahl wird in zwei Fallstudien angewendet, um den Testfokus festzulegen. Ziel der Fallstudien ist zu zeigen, dass der zuvor vorgestellte Ansatz tragfähig ist. Für die Fallstudien werden je zwei frühere Versionen eines in Java realisierten Softwaresystems verwendet, um die fehleranfälligen Eigenschaften zu identifizieren. Der Testfokus wird anschließend für eine dritte Version ermittelt. Folgende Ziele werden durch die Fallstudien verfolgt:

Richtigkeit der Ergebnisse

Von besonderem Interesse ist der Nachweis, dass das vorgeschlagene Vorgehen zur Testfokusauswahl auch die Abhängigkeiten auswählt, die auf eine erhöhte Fehleranzahl in mindestens einem der beteiligten Bausteine hinweisen. Dies geschieht durch den Vergleich des ausgewählten Testfokus der dritten Version des Softwaresystems mit der tatsächlichen Fehleranzahl der beteiligten Bausteine. Für die dritte Version liegt ebenfalls die Fehleranzahl

der Bausteine vor. Diese Informationen wurden jedoch nicht für die Auswahl des Testfokus herangezogen. Wenn das Vorgehen richtig funktioniert, müssen die Abhängigkeiten, die als Testfokus ausgewählt worden sind, durchschnittlich mehr Fehler enthalten (gezeigt durch den mittleren Rang) als die Abhängigkeiten, die nicht als Testfokus ausgewählt worden sind.

Aufwand des Vorgehens

Für den Einsatz des Vorgehens in der Praxis sind Informationen über den Zeit- und Ressourcenaufwand des Vorgehens von Interesse. Aus diesem Grund wird für die durchgeführten Analysen ermittelt, wie viel Zeit für das Analysieren einer früheren Version und für die Testfokusauswahl der zu testenden Version benötigt wird.

Anwendbarkeit

Ein weiteres Ziel der Fallstudien ist zu zeigen, dass das Vorgehen zur Testfokusauswahl für Softwaresysteme unterschiedlicher Größe anwendbar ist.

Wie im oben beschriebenen Vorgehen müssen im ersten Schritt die zu untersuchenden Eigenschaften festgelegt werden. Für die Testfokusauswahl sollen Eigenschaften verwendet werden, die sich leicht aus existierenden Entwicklungsartefakten ermitteln lassen. Hierfür bietet sich der Quelltext der Softwaresysteme an. Die Entscheidung wird durch Jiang et al. unterstützt, die in [JCM+08] schreiben: „*It is not unusal to derive design metrics from code*“ (Seite 12). Der Quelltext kann durch statische Analysewerkzeuge ausgewertet werden, um die Abhängigkeiten und ihre Eigenschaften (in [JCM+08] die Designmetriken) automatisch zu extrahieren.

Bevor die Entscheidung getroffen werden kann, welche Abhängigkeitseigenschaften Anwendung finden, muss entschieden werden, zwischen welchen Arten von Bausteinen (Klassen, Quelltextdateien, Pakete, Komponenten, Schichten) die Abhängigkeiten untersucht werden sollen. In den Fallstudien werden Abhängigkeiten zwischen Quelltextdateien untersucht, da für die einzelnen Quelltextdateien die Fehleranzahl für die zu untersuchenden Versionen des Softwaresystems bereits vorliegen (vgl. Fallstudie Eclipse in Kapitel 5.4.1) oder sich leicht aus bestehenden Artefakten ermittelt lassen (vgl. Fallstudie Fördergeldverwaltung in Kapitel 5.4.2). Durch die Einschränkung auf Quelltextdateien stehen nicht alle in Kapitel 4.1 vorgestellten Abhängigkeitseigenschaften zur Verfügung. Aus diesem Grund konzentrieren sich die nachfolgenden Fallstudien auf 13 Eigenschaften, die mit geringem Aufwand automatisch aus dem Quelltext extrahieren werden. Diese Eigenschaften sind in Tabelle 6 dargestellt und werden später näher erläutert.

Tabelle 6: Übersicht über die erhobenen Abhängigkeitseigenschaften

Quelltexteigenschaft	Wertebereich
Anzahl direkter Attributzugriffe	{0, ∞}
Anzahl direkt aufgerufener Services[1]	{0, ∞}
Anzahl Services mit mindestens einen Eingabeparameter	{0, ∞}
Anzahl Services mit komplexe Ausgabe	{0, ∞}
Anzahl Services mit mindestens zwei Parameter des gleichen Typs	{0, ∞}
Anzahl Eingabeparameter in allen Services	{0, ∞}
Anzahl komplexer Eingabeparameter in allen Services	{0, ∞}
Anzahl neuer Attribute	{0, ∞}
Anzahl neuer Services	{0, ∞}
Anzahl überschriebener Services	{0, ∞}
Durchschnittliche Anzahl komplexer Eingabeparameter pro Serviceaufruf	{0, ∞}
Durchschnittliche Anzahl Eingabeparameter pro Serviceaufruf	{0, ∞}
Relative Häufigkeit der Services mit komplexer Ausgabe	{0, ∞}

[1] In der Fallstudie wird der Begriff Service verwendet, auch wenn im untersuchten Softwaresystem Eclipse der Aufruf von Methoden stattfindet. Ziel ist es, die Terminologie konsistent zu den Begriffen in Kapitel 4 zu halten.

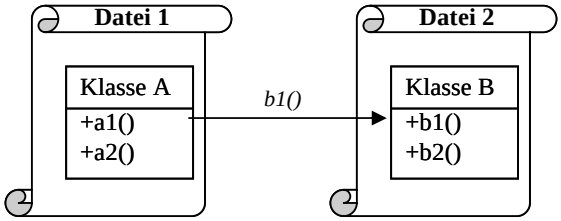
Die Eigenschaften in Tabelle 6 sind Eigenschaften für Abhängigkeiten zwischen Klassen eines objektorientierten Softwaresystems. Alle untersuchten Softwaresysteme sind in der objektorientierten Programmiersprache Java [Ja09] realisiert, wodurch die festgelegten Eigenschaften für diese Systeme angewendet werden können. Durch die vorherige Festlegung, dass nur Abhängigkeiten zwischen Quelltextdateien untersucht werden, lassen sich jedoch nicht alle Eigenschaften in Tabelle 6 uneingeschränkt für die Testfokusauswahl verwendet (z.B. *Anzahl neuer Attribute*, *Anzahl neuer Services*, *Anzahl überschriebener Services*), weil eine Quelltextdatei mehr als eine Klasse beinhalten kann. Es ist daher notwendig, die Eigenschaften von Abhängigkeiten zwischen Klassen durch nachfolgende Regeln auf Eigenschaften von Abhängigkeiten zwischen Quelltextdateien abzubilden:

Regel 1 - Vererbung
*Eine Vererbung zwischen zwei Dateien liegt vor, wenn mindestens eine Klasse aus einer Datei die Unterklasse (erbende Klasse) einer zweiten Klasse in der zweiten Datei ist. Sollten mehrere Klassen der ersten Datei von einer/oder mehrerer Klassen der zweiten Datei erben, so wird dies dennoch nur als **eine** Vererbungsbeziehung zwischen der ersten Datei und der zweiten Datei gewertet.*

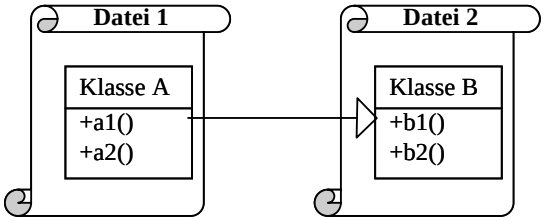
Regel 2 - Serviceaufrufe
Eine Datei ruft an einer zweiten Datei Services auf, wenn mindestens eine Klasse der ersten Datei mindestens einen Service einer Klasse in der zweiten Datei aufruft. Hierbei spielt es keine Rolle, welche Klasse innerhalb der ersten Datei eine Methode bei einer anderen Klasse der zweiten Datei aufruft.

Regel 3 - Attributzugriffe
Eine Datei greift auf Attribute einer zweiten Datei zu, wenn mindestens eine Klasse der ersten Datei auf mindestens ein Attribut einer Klasse in der zweiten Datei zugreift. Hierbei spielt es keine Rolle, welche Klasse innerhalb der ersten Datei auf ein Attribut einer Klasse in der zweiten Datei zugreift.

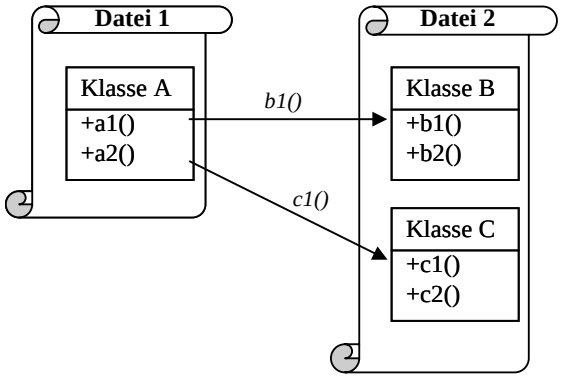
Die Anwendung der Regeln wird in Abbildung 20 näher erläutert. Sie enthält Beispiele von Abhängigkeiten zwischen Klassen und ihre Abbildung auf Abhängigkeiten zwischen Dateien. Im Beispiel „a)“ ruft die `Klasse A` in der `Datei 1` genau einen Service der `Klasse B` in der `Datei 2` auf. Es entsteht eine Aufrufbeziehung zwischen `Datei 1` und `Datei 2`. Ähnliches trifft für Beispiel „b)“ zu. Die Vererbungsbeziehung zwischen der `Klasse A` und der `Klasse B` lässt sich auf `Datei 1` und `Datei 2` übertragen, wobei nicht mehr von Ober- und Unterklasse gesprochen werden kann, sondern dass es eine Abhängigkeit zwischen `Datei 1` und `Datei 2` gibt, bei der Vererbungseigenschaften vorhanden sind.
Im Beispiel „c)“ ruft die `Klasse A` an der `Klasse B` den Service `b1()` und an der `Klasse C` den Service `c1()` auf. Auf Klassenebene handelt es sich um zwei unterschiedliche Abhängigkeiten. Übertragen auf die Abhängigkeit zwischen `Datei 1` und `Datei 2` wird daraus **eine** Abhängigkeit. Ähnliches trifft auch für das Beispiel „d)“ zu. Die drei Abhängigkeiten zwischen den vier Klassen werden auf **eine** Abhängigkeit zwischen zwei Dateien übertragen. Auch im Beispiel „e)“ wird aus den drei Abhängigkeiten zwischen den vier Klassen **eine** Abhängigkeit zwischen Datei 1 und Datei 2. Als Besonderheit enthält diese Abhängigkeit sowohl Eigenschaften der Klasse Vererbung als auch der Klasse Client/Server (vgl. Kapitel 4.1).



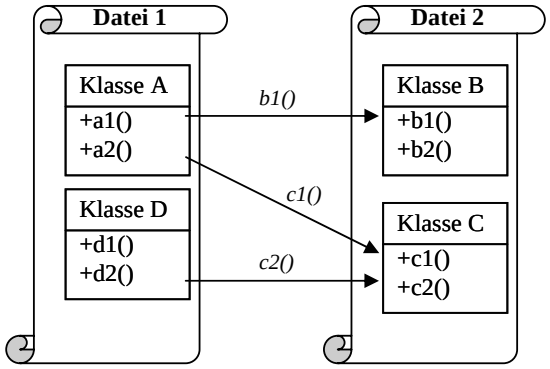
a.) Eine Abhängigkeit mit Aufrufeigenschaften zwischen Datei 1 und Datei 2



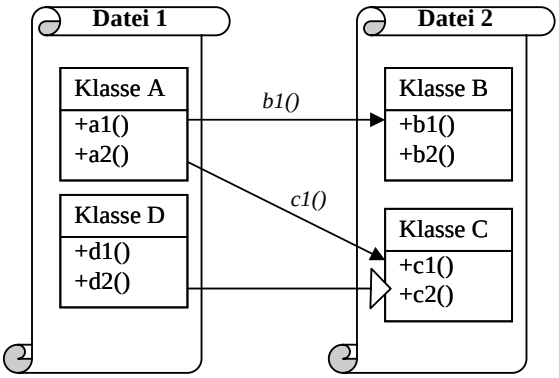
b.) Eine Abhängigkeit mit Vererbungseigenschaften zwischen Datei 1 und Datei 2



c.) **Eine** Abhängigkeit mit Aufrufeigenschaften zwischen Datei 1 und Datei 2



d.) **Eine** Abhängigkeit mit Aufrufeigenschaften zwischen Datei 1 und Datei 2



e.) **Eine** Abhängigkeit mit Aufruf- und Vererbungseigenschaften zwischen Datei 1 und Datei 2

Abbildung 20: Mögliche Abhängigkeiten zwischen Dateien

Die zu untersuchenden Abhängigkeitseigenschaften aus Tabelle 6 werden nachfolgend vorgestellt. Es wird beschrieben, wie sich die Abhängigkeitseigenschaften zwischen Klassen in der Fallstudie auf Abhängigkeitseigenschaften zwischen Quelltextdateien übertragen lassen.

Anzahl direkter Attributzugriffe

Diese Eigenschaft beschreibt, auf wie viele unterschiedliche Attribute Klassen der abhängigen Datei in Klassen der unabhängigen Datei zugreifen. Der Zugriff kann nur auf nicht private Attribute erfolgen. Jedes Attribut wird hierbei nur einmal gezählt, auch wenn mehrfache Attributzugriffe auf das gleiche Attribut stattfinden können.

Anzahl direkt aufgerufener Services
Die Klassen der abhängigen Datei rufen Services (Methoden) an Klassen der unabhängigen Datei auf. Die Anzahl beschreibt, wie viele unterschiedliche Services an Klassen in der unabhängigen Datei aufgerufen werden. Jeder Service wird dabei nur einmal gezählt, auch wenn dieser Service mehrmals aufgerufen wird.

Anzahl Services mit mindestens einem Eingabeparameter
Der Aufruf eines Services kann mit der Übergabe von Inputparametern verbunden sein. Diese Eigenschaft zählt die Anzahl der unterschiedlich aufgerufenen Services, bei denen mindestens ein Parameter mit übergeben wird.

Anzahl Services mit komplexer Ausgabe
Das Ergebnis (der Rückgabewert) eines Services kann ein einfacher oder ein komplexer Datentyp sein. Diese Eigenschaft gibt an, wie viele aufgerufene Services einen komplexen Datentyp als Rückgabewert besitzen.

Anzahl Services mit mindestens zwei Eingabeparameter des gleichen Typs
Ein Aufruf eines Services, bei dem mindestens zwei Inputparameter gleichen Typs übergeben werden, kann zu Fehlern führen. Diese Eigenschaft zählt die Anzahl der unterschiedlich aufgerufenen Services, bei denen mindestens zwei Parameter gleichem Typ beim Aufruf übergeben werden.

Anzahl Eingabeparameter in allen Services
Diese Eigenschaft summiert die Anzahl der Inputparameter pro Service auf. Sie gibt die Gesamtanzahl der Inputparameter, die in der Abhängigkeit zwischen abhängiger und unabhängiger Datei bei Serviceaufrufen übergeben werden. Hierbei werden die unterschiedlich aufgerufenen Services nur einmal betrachtet, d.h. auch wenn ein Service in der unabhängigen Datei durch die abhängige Datei mehrfach aufgerufen wird, werden seine Inputparameter nur einmal berücksichtigt.

Anzahl komplexer Eingabeparametern in allen Services
Diese Eigenschaft gibt die Summe aller komplexen Inputparametern an, die bei allen Serviceaufrufen zwischen abhängiger Datei und unabhängiger Datei übergeben werden. Dieser Wert ist immer kleiner gleich der *Anzahl der Inputparameter*.

Anzahl neuer Attribute
Die Eigenschaft gibt an, wie viele Attribute neu in Unterklassen in der abhängigen Datei definiert worden sind. Hierbei werden alle Vererbungsbeziehungen zwischen den Klassen der abhängigen Datei (Datei mit den Unterklassen) mit den Klassen der unabhängigen Datei betrachtet. Für jede Vererbungsbeziehung zwischen zwei Klassen wird untersucht, wie viele neue Attribute in der Unterklasse hinzugefügt worden sind. Die Summe aller neuen Attribute wird unter der Eigenschaft „Anzahl neuer Attribute“ zusammengefasst.

Anzahl neuer Services
Die Eigenschaft gibt an, wie viele Services in der Abhängigkeit zwischen der abhängigen Datei und der unabhängigen im Rahmen von Vererbungen zwischen Klassen neu in den erbenden Klassen hinzugefügt wurden. Wie bei der Anzahl der neuen Attribute bezieht sich diese Eigenschaft auf die Summe der neuen Services aller Vererbungsbeziehungen von Klassen in der abhängigen Datei und der unabhängigen Datei.

Anzahl überschriebener Services
Die Eigenschaft gibt an, wie viele Services durch die erbenden Klassen der abhängigen Datei überschrieben werden. Sie bezieht sich auf die Summe der überschriebenen Services aller Vererbungsbeziehungen von Unterklassen in der abhängigen Datei und den Oberklassen in der unabhängigen Datei.

Durchschnittliche Anzahl Eingabeparameter pro Serviceaufruf
Diese Eigenschaft beschreibt, wie viele Inputparameter im Durchschnitt bei jedem Serviceaufruf zwischen abhängiger und unabhängiger Datei übergeben werden. Sie lässt sich als Quotient aus der *Anzahl der Inputparameter* und der *Anzahl direkt aufgerufener Services* bestimmen.

Durchschnittliche Anzahl komplexer Eingabeparameter pro Serviceaufruf
Diese Eigenschaft beschreibt die Anzahl der komplexen Inputparameter, die im Durchschnitt bei jedem Serviceaufruf übergeben werden. Dieser Durchschnitt ist der Quotient aus der *Anzahl komplexer Inputparameter* und der *Anzahl direkt aufgerufener Services*.

Relative Häufigkeit der Services mit komplexer Ausgabe
Die relative Häufigkeit bezeichnet den prozentualen Anteil der aufgerufenen Services, bei denen ein komplexer Output zurückgeliefert wird. Sie ist der Quotient aus der *Anzahl Services mit komplexer Ausgabe* und der *Anzahl direkt aufgerufener Services*. Dieser Wert liegt immer zwischen 0 und 1 und kann als Prozentzahl interpretiert werden.

Alle Abhängigkeiten und ihre Eigenschaften können mit Hilfe von statischen Werkzeugen erhoben werden (vgl. Kapitel 1). Das Werkzeug Sissy [Si09] analysiert eine Version eines Softwaresystems und exportiert eine abstrakte Darstellung des Quelltextes in eine Datenbank. Mit Hilfe des in dieser Arbeit erstellten Werkzeugs MetrikAnalyzer wurden die relevanten Informationen über Abhängigkeiten und ihre Eigenschaften aus der abstrakten Darstellung des Quelltextes extrahiert. Das Ergebnis ist eine Tabelle, wie sie in Tabelle 7 beispielhaft dargestellt ist. Sie stellt einen Ausschnitt aus der Ergebnistabelle für die Abhängigkeiten für die Version 2.0 des Werkzeugs Eclipse dar. Jede Zeile der Tabelle repräsentiert eine Abhängigkeit und enthält alle ihre Eigenschaften. Die ersten zwei Spalten enthalten die Bezeichnung der beiden Dateien, die an der Abhängigkeit beteiligt sind[1]. Die erste Spalte repräsentiert den Namen der abhängigen und die zweite Spalte den Namen der unabhängigen Datei. In Tabelle 7 hängt die Datei `CodeAttribute.java` von der Datei `IBytecodeVisitor.java` (Zeile 1) ab. Im Beispiel ist zu erkennen, dass eine Datei für mehrere abhängige Dateien als unabhängige Datei auftreten kann. In Zeile 2 und 3 ist zu erkennen, dass sowohl die Datei `GC.java` als auch die Datei `Decorations.java` von der Datei `OS.java` abhängen.
Die Spalten 3 bis 16 enthalten die oben beschriebenen Abhängigkeitseigenschaften. In Tabelle 7 sind nur vier Eigenschaften (*Anzahl Attributzugriffe*, *Anzahl Serviceaufrufe*, *Summe der Inputparameter* und *Vererbung*) eingetragen. Auf die übrigen wurden zur besseren Übersichtlichkeit im Beispiel verzichtet. Mit Hilfe von Sortier- und Filterfunktionen können aus der Tabelle schnell Abhängigkeiten mit bestimmten Eigenschaften identifiziert werden. In Tabelle 7 sind die Abhängigkeiten nach der Anzahl der Serviceaufrufe absteigend sortiert. Es kann schnell erkannt werden, welche Abhängigkeiten die größten Ausprägungen in dieser

[1] Für eine bessere Übersicht wurden die Dateinamen im Beispiel gekürzt. In der ursprünglichen Ergebnistabelle ist für jede Datei nicht nur der Name, sondern auch der Ordnerpfad der Datei gegeben (z.B. `Eclipse_2_0/plugins/org.eclipse.jdt.core/model/org/eclipse/jdt/core/util/IBytecodeVisitor.java`). Auf diese Weise kann die Eindeutigkeit der Dateien gewährleistet werden.

Eigenschaft aufweisen. Im Beispiel von Eclipse (Version 2.0) ruft die Datei `CodeAttribute.java` 205 unterschiedliche Services an der Datei `IBytecodeVisitor.java` auf.

Tabelle 7: Ausschnitt der Ergebnistabelle für die Abhängigkeitseigenschaften für Eclipse 2.0

Abhängige Datei	Unabhängige Datei	# Attributzugriffe	# Serviceaufrufe	# Inputparameter	Vererbung	...
CodeAttribute.java	IBytecodeVisitor.java	0	205	15	0	...
GC.java	OS.java	37	75	56	0	...
Decorations.java	OS.java	79	72	31	0	...
Control.java	OS.java	164	71	42	0	...
BinaryExpression.java	CodeStream.java	1	64	6	0	...
ASTConverter.java	AST.java	0	63	3	0	...
AstMatchingNodeFinder.java	ASTVisitor.java	0	62	1	1	...
Display.java	OS.java	80	52	33	0	...
CodeStream.java	ConstantPool.java	4	51	3	0	...
Shell.java	OS.java	49	47	29	0	...
⋮	⋮	⋮	⋮	⋮	⋮	...

Die Ergebnistabelle des MetricAnalyzers wird um die Fehleranzahl der einzelnen Bausteine ergänzt. Sie dient als Eingabe für die Korrelationsanalyse, um fehleranfällige Abhängigkeitseigenschaften zu identifizieren. Nach der Identifikation der fehleranfälligen Abhängigkeitseigenschaft wird der Testfokus für eine dritte Version festgelegt. Abschließend wird überprüft, wie gut die Auswahl des Testfokus ist, indem die Fehleranzahl der dritten Version verwendet wird.

5.4.1. Eclipse

Die erste Fallstudie wird am Open-Source Werkzeug Eclipse [Ec09] durchgeführt. Eclipse stellt eine umfangreiche Umgebung für Softwareentwickler zur Verfügung und „*… kann in den meisten Phasen der Softwareentwicklung eingesetzt werden, vor allen in denen, die sich mit Quelltexten beschäftigen*“([ZK04] Seite 344). Es wurde für die Fallstudie ausgewählt, da für drei Versionen dieses Werkzeug die Fehleranzahl pro Datei in [ZPZ07] bereits von Zimmermann, Premraj und Zeller ermittelt worden waren. Diese konnten in der Fallstudie verwendet werden.

Für die Identifikation der Eigenschaften, die auf eine hohe Fehleranzahl hindeuten, werden die Eclipse Versionen 2.0 und 2.1 verwendet. Die Version 2.0 besteht aus 6747 Quelltextdateien. Diese 6747 Dateien enthalten in der Summe 1361739 Quelltextzeilen, das sind im Durchschnitt ca. 202 Quelltextzeilen pro Datei. Zwischen den 6747 Quelltextdateien existieren 56765 Abhängigkeiten, was einer durchschnittlichen Anzahl von 8 Abhängigkeiten pro Datei entspricht. Von der Gesamtzahl der Abhängigkeiten sind 5,25% Abhängigkeiten, die nur Eigenschaften einer Vererbungsbeziehung aufweisen. 5,91% enthalten sowohl Vererbungs- als auch Client/Server-Eigenschaften. Die größte Gruppe (88,84%) der Abhängigkeiten bilden die Abhängigkeiten, die nur Client/Server-Eigenschaften aufweisen,

d.h. mindestens ein Attributzugriff oder einen Serviceaufruf an der unabhängigen Datei durchführen. Die größte Anzahl Attributzugriffe von Klassen in der abhängigen Datei zu Klassen in der unabhängigen Datei ist 240. Die größte Anzahl Serviceaufrufe zwischen zwei Dateien ist 205. Die Eclipse Version 2.1 besteht aus 7908 Quelltextdateien. Auf diese verteilen sich 1678952 Quelltextzeilen, was zu einer durchschnittlich 211 Quelltextzeilen pro Datei führt. Es existieren 71182 Abhängigkeiten zwischen den Quelltextdateien, was einer durchschnittlichen Anzahl Abhängigkeiten pro Datei von 9 entspricht. Von der Gesamtzahl der Abhängigkeiten sind 5,05% Abhängigkeiten, die nur Eigenschaften einer Vererbungsbeziehung aufweisen. 5,62% enthalten sowohl Vererbungs- als auch Client/Server-Eigenschaften. 88,84% der Abhängigkeiten weisen nur Abhängigkeiten mit Client/Server-Eigenschaften auf. Die größte Anzahl Attributzugriffe 262 und die größte Anzahl Serviceaufrufe ist unverändert 205.
Die Fehleranzahl der einzelnen Quelltextdateien für beide Versionen kann aus der Arbeit von Zimmermann, Premraj und Zeller in [ZPZ07] übernommen werden. [ZPZ07] stellt sowohl Pre- als auch Postrelease Fehler zur Verfügung. Für die nachfolgenden Analysen wurde für jede Quelltextdatei die Summe der gefundenen Fehler vor dem Release und nach dem Release gebildet, um die Gesamtanzahl von Fehlern pro Datei für ein Release (eine Version) zu ermitteln. Von den 6747 Quelltextdateien der Version 2.0 enthalten 2891 Dateien (42,85%) mindestens einen Fehler. In der Version 2.1 enthalten „nur“ 30,68% (2426) der Dateien mindestens einen Fehler. Die durchschnittliche Fehleranzahl pro Datei für die Version 2.0 ist 1,38 und für die Version 2.1 liegt sie bei 0,78 Fehlern pro Datei.
Die Fehleranzahl der Quelltextdateien und die Informationen über die Abhängigkeiten und ihre Eigenschaften werden verwendet, um signifikante Zusammenhänge zwischen den Eigenschaften und der Fehleranzahl aufzudecken. Für jede Eigenschaft in Tabelle 6 wird der Kruskal-Wallis-H Test zweimal ausgeführt. Zum einen, um Zusammenhänge zwischen der Eigenschaft und der Fehleranzahl der abhängigen Datei aufzudecken und zum anderen um Zusammenhänge zwischen der Eigenschaft und der Fehleranzahl der unabhängigen Datei aufzudecken. Zuvor werden die Abhängigkeiten nach ihren Werten der Eigenschaften gruppiert. Für die Gruppierung werden Dezile (10er Gruppen) verwendet.
Die Ergebnisse des Kruskal-Wallis-H Tests für die 13 Eigenschaften sind grafisch in Anhang A: *Statistiken Eclipse* dargestellt. Für jede untersuchte Eigenschaft wurden vier Diagramme erstellt. Die Diagramme wurden zur besseren Übersicht in eine vierspaltige Tabelle eingeordnet. Die ersten zwei Spalten enthalten die Diagramme, die den (möglichen) Zusammenhang zwischen den gruppierten Eigenschaftswerten und der Fehleranzahl im abhängigen Baustein darstellen. Die erste Spalte stellt diesen Zusammenhang für Version 2.0 und die zweite Spalte für Version 2.1 dar. Die Spalten 3 und 4 enthalten diese Informationen für den (möglichen) Zusammenhang mit der Fehleranzahl des unabhängigen Bausteins. Die grau hinterlegten Diagramme signalisieren, dass der Kruskal-Wallis-H einen Signifikanzwert ermittelt hat, der größer als 0,05 ist. D.h. dass sich keine der dort aufgezeigten Gruppen signifikant voneinander unterscheiden. In der Fallstudie betrifft das die folgenden Eigenschaften:

Anzahl neuer Attribute
Für diese Eigenschaft hat sich in der Version 2.0 gezeigt, dass es bei der Untersuchung des Zusammenhangs zwischen dieser Eigenschaft und der Fehleranzahl in der unabhängigen Datei keinen signifikanten Zusammenhang gibt, d.h. es gibt keine Gruppe, in der sich der mittlere Rang signifikant von den anderen Gruppen unterscheidet.

Anzahl Services mit mindestens zwei Eingabeparametern gleichen Typs
Für diese Eigenschaft ermittelte der Kruskal-Wallis-H Test für die Version 2.0, dass es keinen signifikanten Zusammenhang mit der Fehleranzahl in der unabhängigen Datei gibt.

Darüber hinaus zeigt die Untersuchung des Zusammenhangs dieser Eigenschaft mit der Fehleranzahl der abhängigen Datei, dass es für die Version 2.1 keinen signifikanten Zusammenhang gibt.
Für die Eigenschaften, für die der Kruskal-Wallis-H Test einen signifikanten Zusammenhang zwischen mindestens zwei Gruppen ermittelt hat, werden die Mann-Whitney-U Tests für die Gruppen angewendet, die jeweils den höchsten mittleren Rang aufweisen. Es wird überprüft, ob diese Gruppe sich signifikant von den verbleibenden Gruppen unterscheidet. Hierbei wurden nur Gruppen betrachtet, deren mittlerer Rang über dem durchschnittlichen Rang aller Abhängigkeiten der untersuchten Version lag. Das Ergebnis ist für Version 2.0 in Tabelle 8 dargestellt. Die erste Spalte enthält die Bezeichnung der untersuchten Eigenschaften. Die jeweils signifikanten Gruppen sind in Spalte 2 und Spalte 4 dargestellt. Spalte 2 enthält die Gruppen die den höchsten mittleren Rang für die Fehleranzahl der abhängigen Datei aufweisen, Spalte 4 entsprechend die Gruppen für die unabhängige Datei. Der Wert „-1“ signalisiert, dass es für die entsprechende Eigenschaft keinen Zusammenhang mit der Fehleranzahl der bestreffenden Datei gibt. Dies kann zwei Gründe haben. 1) Keine der Gruppen weist einen mittleren Rang auf, der größer dem mittleren Rang aller Abhängigkeiten ist oder 2) die Gruppe mit dem größten mittleren Rang unterscheidet sich nicht signifikant (ermittelt mit dem Mann-Whitney-U Test) von mindestens einer weiteren Gruppe dieser Eigenschaft.

Tabelle 8: Signifikante Gruppen für Eclipse 2.0

	Abhängige Datei		**Unabhängige Datei**	
Eigenschaften	Gruppen	Werte	Gruppen	Werte
Anzahl überschriebener Services	10	>10	-1	-1
Anzahl neuer Services	10	>17	-1	-1
Anzahl neuer Attribute	10	>11	-1	-1
Anzahl direkter Attributzugriffe	10	>5	10	>5
Anzahl direkter Serviceaufrufe	10	>4	10	>4
Anzahl komplexer Eingabeparameter in allen Services	10	>2	9	2
Anzahl Eingabeparameter in allen Services	10	>3	-1	-1
Anzahl Services mit komplexer Ausgabe	10	>2	10	>2
Anzahl Services mit mindestens einem Eingabeparameter	10	>2	10	>2
Anzahl Services mit mindestens zwei Eingabeparameter gleichen Typs	10	>0	-1	-1
Durchschnittliche Anzahl komplexer Eingabeparameter pro Serviceaufruf	4	0,0-0,09	4	0,0-0,09
Durchschnittliche Anzahl Eingabeparameter pro Serviceaufruf	-1	-1	8	1,07-1,20
Relative Häufigkeit der Services mit komplexer Ausgabe	-1	-1	5	5-25%

Das Interpretieren der Tabelle 8 wird am Beispiel der ersten und letzten Eigenschaft erläutert. In der ersten Zeile sind die signifikanten Gruppen für die Eigenschaft *Anzahl überschriebener Services* aufgelistet. Die Gruppe 10 (vgl. Spalte 2) beinhaltet alle Abhängigkeiten, bei denen in einer Vererbungsbeziehung die Anzahl der überschriebenen Services größer 10 ist (vgl. Spalte 3). Der mittlere Rang (nach Fehleranzahl der abhängigen Datei) der Gruppe 10 ist größer als der mittlere Rang in den jeweiligen verbleibenden Gruppen und liegt über dem mittleren Rang aller Abhängigkeiten (auch zu erkennen im Diagramm im Anhang A). Die Spalten 4 und 5 zeigen für die Eigenschaften *Anzahl*

überschriebener Services an, dass es keine Gruppe gibt, die die Voraussetzung für die Signifikanz (mittlerer Rang der Gruppe über mittlerem Rang aller Abhängigkeiten, signifikanter Unterschied zu anderen Gruppen) erfüllt. Die letzte Zeile der Tabelle beschreibt die signifikanten Gruppen für die Eigenschaft *Relative Häufigkeit der Services mit komplexer Ausgabe*. Diese Eigenschaft weist keine Gruppe auf, die sich signifikant bezüglich des mittleren Rangs der Fehleranzahl der abhängigen Datei von den anderen Gruppen unterscheidet (gegenzeichnet durch „-1“ in der Tabelle). Jedoch gibt es einen signifikanten Zusammenhang zwischen der Gruppe 5 und der Fehleranzahl der unabhängigen Datei. Diese Gruppe beinhaltet alle Abhängigkeiten bei denen 5-25% der aufgerufenen Services einen komplexen Rückgabewert haben.

Die Ergebnisse für die Korrelationsanalyse für Eclipse 2.1 ist in Tabelle 9 dargestellt. Sie kann ebenso wie Tabelle 8 interpretiert und gelesen werden. Es ist zu erkennen, dass es ebenfalls Eigenschaften gibt, die einen Zusammenhang zwischen der Fehleranzahl der abhängigen und/oder unabhängigen Datei aufweisen.

Tabelle 9: Signifikante Gruppen für Eclipse 2.1

	Abhängige Datei		**Unabhängige Datei**	
Eigenschaften	Gruppen	Werte	Gruppen	Werte
Anzahl überschriebener Services	10	>11	-1	-1
Anzahl neuer Services	10	>17	-1	-1
Anzahl neuer Attribute	10	>7	-1	-1
Anzahl direkter Attributzugriffe	10	>5	-1	-1
Anzahl direkter Serviceaufrufe	10	>4	10	>4
Anzahl komplexer Eingabeparameter in allen Services	10	>2	-1	-1
Anzahl Eingabeparameter in allen Services	-1	-1	10	>3
Anzahl Services mit komplexer Ausgabe	10	>2	10	>2
Anzahl Services mit mindestens einem Eingabeparameter	10	>2	10	>2
Anzahl Services mit mindestens zwei Eingabeparameter gleichen Typs	-1	-1	5	0
Durchschnittliche Anzahl komplexer Eingabeparameter pro Serviceaufruf	4	0,0-0,08	4	0,0-0,08
Durchschnittliche Anzahl Eingabeparameter pro Serviceaufruf	-1	-1	-1	-1
Relative Häufigekeit der Services mit komplexer Ausgabe	7	52-94%	5	5-32%

Die Ergebnisse der Korrelationsanalyse der Versionen 2.0 und 2.1 werden anschließend verwendet, um den Testfokus für die Version 3.0 festzulegen. Ziel ist es, die Abhängigkeiten auszuwählen, die auf eine hohe Fehleranzahl im abhängigen und/oder unabhängigen Baustein hindeuten. Bevor dies geschehen kann, müssen in einem weiteren Schritt alle Eigenschaften identifiziert werden, die Gruppen aufweisen, die in den Versionen 2.0 und 2.1 auf einen Zusammenhang zwischen der Fehleranzahl des unabhängigen bzw. abhängigen Bausteins hinweisen. Hierzu werden die Tabelle 8 und Tabelle 9 „übereinander“ gelegt. Befindet sich für eine Eigenschaft in der 2. Spalte die gleiche Gruppenzahl, so wird ein versionsübergreifender Zusammenhang zwischen der Eigenschaft und der Fehleranzahl des abhängigen Bausteins angenommen. Für die Version 2.0 und 2.1 ist dies in der ersten Zeile der Tabellen (Eigenschaft *Anzahl überschriebener Services)* der Fall. Befindet sich für eine

Eigenschaft in der 4. Spalte die gleiche Gruppenzahl, dann deutet dies auf einen versionsübergreifenden Zusammenhang zwischen der Eigenschaft und der Fehleranzahl der unabhängigen Datei hin. Dies ist für Version 2.0 und 2.1 der Fall für die Eigenschaft *Relative Häufigkeit der Services mit komplexer Ausgabe*.
Alle Eigenschaften, die versionsübergreifende Zusammenhänge mit der Fehleranzahl der abhängigen und/oder der unabhängigen Datei aufweisen, werden zur Testfokusauswahl für die Version 3.0 verwendet. Diese Eigenschaften sind in Tabelle 10 aufgelistet. Diese Tabelle beschreibt, welche Eigenschaften für die Testfokusauswahl verwendet werden. Im Detail gibt sie an, welche Gruppe der jeweiligen Eigenschaften auszuwählen ist, um den Testfokus festzulegen. Hierbei wird unterschieden, ob die Gruppe mit der Fehleranzahl der abhängigen Datei, der unabhängigen Datei oder beiden Dateien korreliert.

Tabelle 10: Auszuwählende Gruppen für die Testfokusauswahl der Version 3.0

Eigenschaften	Abhängige Datei	Unabhängige Datei	Beide Dateien
Anzahl überschriebener Services	10	-1	-1
Anzahl neuer Services	10	-1	-1
Anzahl neuer Attribute	10	-1	-1
Anzahl direkter Attributzugriffe	10	-1	-1
Anzahl direkter Serviceaufrufe	10	10	10
Anzahl komplexer Eingabeparameter in allen Services	10	-1	-1
Anzahl Eingabeparameter in allen Services	-1	-1	-1
Anzahl Services mit komplexer Ausgabe	10	10	10
Anzahl Services mit mindestens einem Eingabeparameter	10	10	10
Anzahl Services mit mindestens zwei Eingabeparameter gleichen Typs	-1	-1	-1
Durchschnittliche Anzahl komplexer Eingabeparameter pro Serviceaufruf	4	4	4
Durchschnittliche Anzahl Eingabeparameter pro Serviceaufruf	-1	-1	-1
Relative Häufigkeit der Services mit komplexer Ausgabe	-1	5	-1

Es ist zu erkennen, dass für die Testfokusauswahl der Version 3.0 vier Eigenschaften existieren (*Anzahl direkter Serviceaufrufe*, *Anzahl Services mit komplexer Ausgabe*, *Anzahl Services mit mindestens einem Eingabeparameter*, *Durchschnittliche Anzahl Eingabeparameter pro Serviceaufruf*) die versionsübergreifende Zusammenhänge sowohl mit der Fehleranzahl der abhängigen als auch der unabhängigen Datei aufweisen. Diese Abhängigkeiten erhalten in der Version 3.0 die *Testpriorität-beide Bausteine*. Darüber hinaus gibt es fünf Eigenschaften (*Anzahl überschriebener Services*, *Anzahl neuer Services*, *Anzahl neuer Attribute*, *Anzahl direkter Attributzugriffe*, *Anzahl komplexer Eingabeparameter in allen Services*), die mit der Fehleranzahl der abhängigen Datei zusammenhängen. Ein Zusammenhang zwischen Eigenschaft und der Fehleranzahl der unabhängigen Datei existiert nur für die Eigenschaft *Relative Häufigkeit der Services mit komplexer Ausgabe*. Diese Eigenschaften werden bei der Testfokusauswahl für die Version 3.0 verwendet, um den entsprechenden Abhängigkeiten die *Testpriorität-unabhängiger Baustein* bzw. *Testpriorität-abhängiger Baustein* zuzuweisen. Alle verbleibenden Abhängigkeiten bekommen die Kennzeichnung *keine Testpriorität*.
Angewendet auf die Version 3.0 ergibt die Testfokusauswahl, dass von den 96476 Abhängigkeiten zwischen den Quelltextdateien 14319 Abhängigkeiten (ca. 14%) eine Testpriorität zugeordnet wurden. Von den 14319 Abhängigkeiten waren 1352

Abhängigkeiten (1,4%) mit der *Testpriorität-abhängiger Baustein* und 7737 Abhängigkeiten (8,02%) mit der *Testpriorität-unabhängiger Baustein*. Die *Testpriorität-beide Bausteine* bekamen 5230 Abhängigkeiten (5,42%). 82157 Abhängigkeiten (85,16%) wurden mit *Keine Testpriorität* gekennzeichnet.

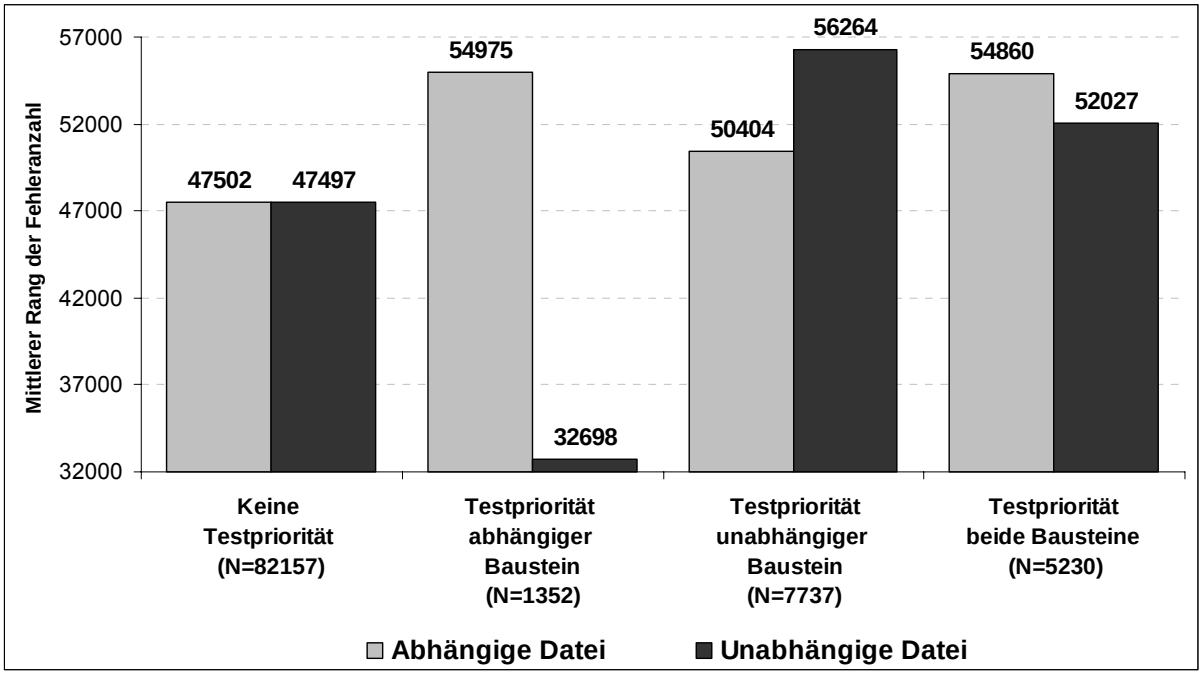


Abbildung 21: Mittlere Ränge (Fehleranzahl) der Abhängigkeiten, eingeteilt nach Testpriorität für Eclipse 3.0

Zur Überprüfung, ob Abhängigkeiten ausgewählt worden sind, die fehleranfälliger sind als die nicht-ausgewählten Abhängigkeiten, wurden mit Hilfe des Kruskal-Wallis-H Tests die mittleren Ränge der einzelnen Gruppen ermittelt. Das Ergebnis dieses Tests ist in Abbildung 21 dargestellt. Der hellgraue Balken beschreibt den mittleren Rang der Fehleranzahl der abhängigen Datei und die dunkelgraue Balken den mittleren Rang der Fehleranzahl der unabhängigen Datei. In einem ersten Schritt betrachten wir die hellgrauen Balken. Es ist zu erkennen, dass alle Abhängigkeiten, für die eine Testpriorität zugeordnet ist, einen höheren mittleren Rang aufweisen, als die nicht-ausgewählten Abhängigkeiten. Dabei ist, wie zu erwarten, der mittlere Rang für die Abhängigkeiten mit *Testpriorität-abhängiger Baustein* und *Testpriorität-beide Bausteine* signifikant höher als der mittlere Rang der nicht-ausgewählten Abhängigkeiten. Für die dunkelgrauen Balken sind die gleichen Ergebnisse zu erkennen. Die Abhängigkeiten, die Eigenschaften besitzen, die mit der Fehleranzahl in den unabhängigen Bausteinen korrelieren (*Testpriorität-unabhängiger Baustein* und *Testpriorität-beide Bausteine*) weisen einen signifikant höheren mittleren Rang auf, als die nicht-ausgewählten Abhängigkeiten.
Das Diagramm in Abbildung 21 bestätigt, dass für Eclipse 3.0 Abhängigkeiten ausgewählt worden sind, die eine größere Fehleranzahl in den abhängigen bzw. in den unabhängigen Dateien besitzen.

5.4.2. Werkzeug zur Fördergeldverwaltung

Die zweite Fallstudie wird an einer kommerziellen Software durchgeführt. Die Software wurde entwickelt, um öffentliche Behörden bei der Verwaltung und Begutachtung von Fördergeldanträgen zu unterstützen. Antragsteller haben die Möglichkeit, Anträge auf Fördergelder der EU (Europäische Union) zu stellen. Diese Anträge werden von den Mitarbeitern der Behörden bearbeitet und es wird geprüft, ob die Anträge angenommen oder

abgelehnt werden. Bei dieser Entscheidung werden die Behörden von der Software unterstützt, in dem die Software automatisch prüft, ob die Vorbedingungen für die Vergabe der Fördergelder erfüllt sind und ob alle gesetzlichen Rahmenbedingungen eingehalten werden. Werden die Anträge angenommen, unterstützt das System bei der Zuteilung der Fördergelder, führt die Überweisungen der Fördergelder durch und prüft in regelmäßigen Abständen, ob die Bedingungen für die Fördergeldvergabe weiterhin erfüllt sind.
Die analysierte Software besteht aus über vier Millionen Quelltextzeilen, die sich auf über 23000 Quelltextdateien verteilen und kann somit als realistisch großes Softwaresystem betrachtet werden. Bei der Software handelt es sich um eine Client-Server-Anwendung.
Für die Software sind keine Versionen fest definiert. Vielmehr wird die Software an sich ständig wechselnde gesetzliche Bedingungen angepasst. Dies erschwert die Auswahl von zu untersuchenden Versionen für die Fallstudie. Um die Ergebnisse der Fallstudie *Werkzeugunterstützung zur Fördergeldverwaltung* mit den Ergebnisse der Fallstudie *Eclipse* vergleichen zu können, werden drei Versionen der Software ausgewählt. Zwei Versionen dienen zur Identifikation von Korrelationen. Für die dritte Version wird der Testfokus festgelegt. Wie die drei Versionen für die Fallstudie festgelegt worden sind, wird im Unterkapitel 5.4.2.1 erläutert.
Zur Durchführung der Fallstudie müssen Informationen über die Fehleranzahl der Quelltextdateien vorliegen. Diese werden in einem ersten Schritt aus vorhandenen Artefakten des Entwicklungsprozesses gewonnen. Hierzu wird ein Vorgehen verwendet, das sich an das Vorgehen von [ZPZ07] anlehnt. Es werden Informationen über die dokumentierten Fehler verwendet und mit Hilfe der Versionshistorie des Versionsverwaltungssystems Subversion [Su09] die Fehleranzahl der Bausteine aufgedeckt. Das detaillierte Vorgehen wird in Kapitel 5.4.2.1 näher vorgestellt. Im Anschluss werden die Ergebnisse der Fallstudie, d.h. die korrelierenden Eigenschaften und der ausgewählte Testfokus beschrieben (Kapitel 5.4.2.2).

5.4.2.1 Fehleridentifikation

Die entwickelnde Firma der Software für das Werkzeug zur Fördergeldverwaltung verwendet zur Dokumentation von Fehlern die Software Lotus Notes von IBM [LN09]. Jeder gefundene Fehler wird in Lotus Notes dokumentiert und bekommt automatisch eine eindeutige ID zugeordnet (z.B. TM-2974). Dabei wird unterschieden, ob es sich um einen internen Fehler oder einen externen Fehler handelt. Ein interner Fehler ist ein Fehler, der während des internen Tests aufgedeckt wird. Ein externer Fehler ist ein Fehler, der von Benutzern während des Produktivbetriebs gefunden und gemeldet wird. Wenn Entwickler einen Fehler beseitigen, ändern ´sie die betroffenen Quelltextdateien und schreiben die Änderungen zurück in das Versionsverwaltungssystem Subversion [Su09]. Das Versionsverwaltungssystem vergibt automatisch für diese Dateien eine neue Revisionsnummer. Diese Nummer lesen die Entwickler aus und tragen sie händisch in Lotus Notes für den entsprechenden dokumentierten Fehler ein. Für jeden beseitigten Fehler kann somit erkannt werden, welche Revision neu erzeugt wurde, um den Fehler zu beseitigen. Diese Informationen können verwendet werden, um die geänderten Dateien zu identifizieren.
Mit Hilfe des Werkzeugs SVNReport [SR09] kann aus Subversion die vollständige Historie herausgelesen werden. Als Ergebnis erhält man eine Tabelle, die Informationen über jede Revision enthält. Für jede Revision kann herausgelesen werden, welcher Entwickler die Revision erstellt hat, welche Dateien während dieser Revision verändert worden sind und wann die Revision erstellt wurde. Diese Informationen und die Informationen der dokumentierten Fehler werden verwendet, um die Dateien zu identifizieren, die für die Beseitigung eines Fehlers verändert worden sind.
Im nächsten Schritt müssen Versionen ausgewählt werden, die für die spätere Fallstudie verwendet werden können. In Absprache mit der entwickelnden Firma legen wir fest, dass es

sich um drei Versionen aus dem Jahr 2008 handeln soll. Dies liegt zum einen darin begründet, dass die Menge der Daten aus der Versionshistorie reduziert werden kann, da nur noch die Einträge von 2008 untersucht werden müssen. Zum anderen sollen es aktuelle Versionen sein. Die Frage ist nun, wie die Versionen aus dem Jahr 2008 ausgewählt werden können. Da keine fest definierten Versionen, wie in der Fallstudie Eclipse existieren, müssen sie künstlich eingeführt werden. Es werden drei Tage aus dem Jahr 2008 ausgewählt. Es wurde definiert, dass die Revision, die als letztes an einem Tag erzeugt wurde eine Version darstellt. Die drei Tage werden mit dem Ziel ausgewählt, möglichst Versionen zu verwenden, die eine hohe Fehleranzahl aufweisen. Hierzu wird für jeden Tag des Jahres 2008 die Anzahl Fehler berechnet, die sich zu diesem Zeitpunkt in der Software befanden. Zur Berechnung der Fehleranzahl in der Software zu einem bestimmten Tag werden die Informationen aus der Versionshistorietabelle und die dokumentierten Fehler verwendet. Folgende Schritte müssen durchgeführt werden, um die Anzahl Fehler pro Version zu ermitteln:

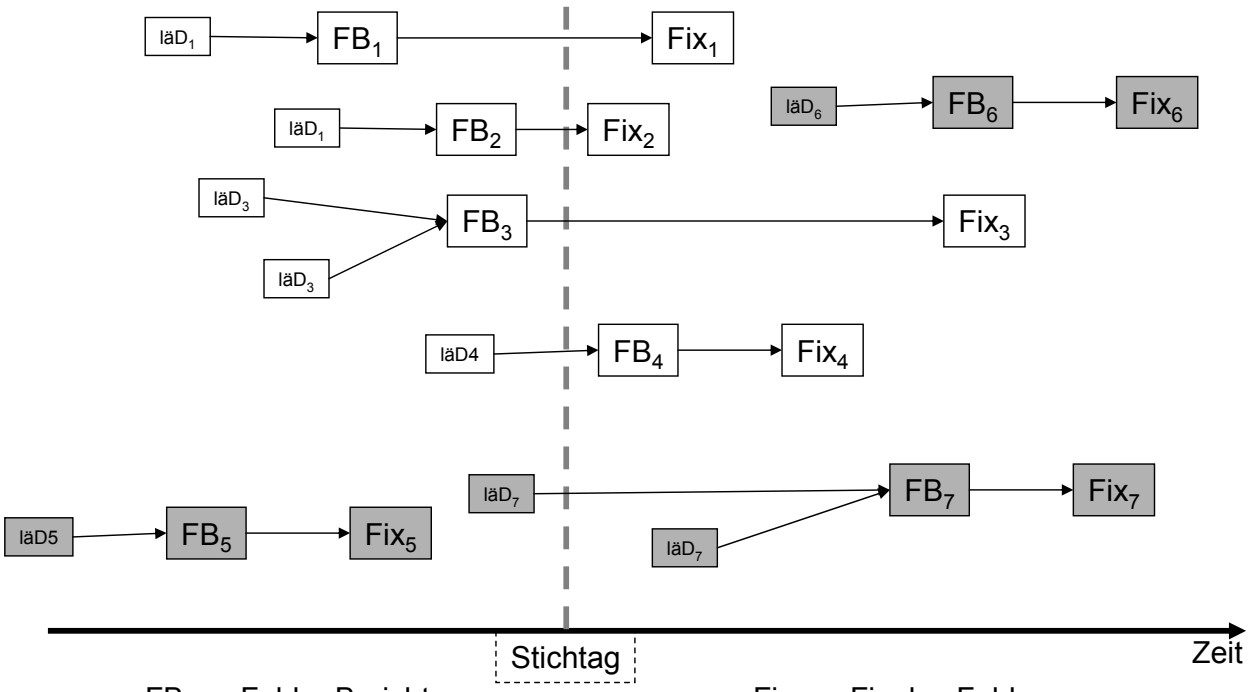
Schritt 1: Für jeden dokumentierten Fehler werden die Revisionen identifiziert, die erzeugt wurden, um den Fehler zu beseitigen. Diese Informationen sind in den Informationen über den Fehler zu finden.

Schritt 2: Mit Hilfe der Revisionen werden die Dateien identifiziert, die geändert werden mussten, um den Fehler zu beseitigen. Diese Informationen finden sich in der Versionshistorietabelle.

Schritt 3: Für jeden Fehler und die zugeordneten Dateien wird bestimmt, wann die Dateien vor der Dokumentation des Fehlers das letzte Mal verändert worden sind. Wann ein Fehler dokumentiert und wann er beseitigt wurde, lässt sich aus den Informationen über den Fehler entnehmen. Wann die betroffenen Dateien das letzte Mal geändert wurden, ist aus der Versionshistorie ablesbar.

Schritt 4: Es wird ein Stichtag definiert, für den die Anzahl Fehler innerhalb des Softwaresystems ermittelt wird. Dies geschieht mit Hilfe der Informationen aus Schritt 3. Ein dokumentierter Fehler ist zu dem ausgewählten Tag im System, wenn (1) das Datum (Fixdatum), an dem der Fehler beseitigt worden ist, nach dem Stichtag liegt und (2) die letzten Änderungen aller Dateien, die zur Beseitigung des Fehlers verändert wurden, vor dem Stichtag liegen. Die Zuordnung der Fehler zu einer Version (definiert durch den Stichtag) wird an einem Beispiel in Abbildung 22 veranschaulicht. In der Abbildung sind beispielhaft sieben dokumentierte Fehler zu erkennen. Ein Fehler ist durch FB_n in der Abbildung gekennzeichnet. Die Position innerhalb der Abbildung spiegelt die die Zeit wieder. Je weiter links sich ein Fehlerbericht auf der x-Achse befindet, desto früher wurde er erstellt. Zu jedem Fehler ist jeweils ein Zeitpunkt zugeordnet, der angibt, wann der Fehler beseitigt worden ist (Fix_n). Dieser befindet sich immer zeitlich nach dem Erstellungszeitpunkt des dokumentierten Fehlers. Für das Beseitigen eines Fehlers müssen eine oder mehrere Dateien geändert werden. Die letzte Änderung der betroffenen Dateien ist in der Abbildung durch $läD_n$ (letzte Änderung der Datei) gekennzeichnet und ihre Position in der Abbildung spiegelt den Zeitpunkt der letzten Änderung der Datei vor der Fehlermeldung wider. Die senkrechte gestrichelte Linie repräsentiert den Stichtag. Alle weißen Elemente innerhalb der Abbildung deuten auf Fehler hin, die sich zum betrachteten Zeitpunkt im System befinden. Für den Fehler FB_1 beispielsweise zeigt sich, dass der Fehler nach dem Stichtag beseitigt wurde und die letzte Änderung der Datei ($läD_1$) vor dem Stichtag liegt. Somit befindet sich der Fehler zum betrachteten Zeitpunkt im System, weil er erst später beseitigt worden ist. Die grauen Elemente in der Abbildung stellen dokumentierte Fehler dar, die nicht dem Stichtag zugeordnet werden können. FB_6 kann nicht zugeordnet werden, da sich die letzte

Änderung der betroffenen Datei ($läD_6$) nach dem Stichtag befindet und der Fehler FB_6 durch diese Änderung in das System eingefügt sein kann. Der Fehler FB_5 wird ebenfalls nicht dem Stichtag zugeordnet, da er bereits vor dem Stichtag beseitigt wurde und nicht mehr in der Software enthalten ist. Der Fehler FB_7 wird dem Stichtag nicht zugeordnet, da nicht alle von dem Fehler betroffenen Dateien vor dem Stichtag verändert worden sind. Eine Änderung wurde erst nach dem Stichtag durchgeführt, wodurch der Fehler nicht eindeutig dem Stichtag zugeordnet werden kann. Im Beispiel würde die Software zum Stichtag vier Fehler besitzen.



FB_n … Fehler Bericht *n* Fix_n … Fix des Fehlers *n*

$läD_n$ … Letzte Änderung der fehlerhaften Datei vor der Fehlermeldung

Abbildung 22: Zuordnung der Fehler zu einer ausgewählten Version

Für jeden Tag des Jahres 2008 wird die Fehleranzahl der Software ermittelt. Es werden die drei Tage ausgewählt, die die höchste Fehleranzahl aufweisen. Die Revision, die als letztes an diesem Tag im Versionsverwaltungssystem erzeugt wurde, wird als Version für die Fallstudie ausgewählt. Die Tage mit den meisten Fehlern sind für das untersuchte Softwaresystem der 01.07.2008, der 31.07.2008 und der 18.09.2008. Die Ursache liegt nach Aussagen von Firmenmitarbeitern darin begründet, dass im Juli ein Großteil der Entwickler im Urlaub war und die gefundenen Fehler nicht sofort beseitigt werden konnten.

Mit Hilfe dieser drei Versionen wird die Fallstudie durchgeführt, wobei die Versionen vom 01.07. und vom 31.07.2008 verwendet werden, um die Korrelationen aufzudecken. Der Testfokus wird für die Version vom 18.09.2008 bestimmt.

5.4.2.2 Ergebnisse der Fallstudie

Die Version vom 01.07.2008 besteht aus 22987 Quelltextdateien auf die sich 4,02 Millionen Zeilen Quelltext verteilen. Für diese Version können 158628 Abhängigkeiten zwischen Quelltextdateien identifiziert werden. Von den 22987 Quelltextdateien enthalten nur 602 Dateien (2,62%) mindestens einen bekannten Fehler. Die zweite Version vom 31.07.2008 enthält 4,05 Millionen Quelltextzeilen in 23216 Quelltextdateien. 643 Dateien (2,77%) enthalten mindestens einen bekannten Fehler. Die Anzahl Abhängigkeiten stieg nur leicht auf 158637.

Tabelle 11: Signifikante Gruppen für die Version vom 01.07.2008

	Abhängige Datei		**Unabhängige Datei**	
Eigenschaften	Gruppen	Werte	Gruppen	Werte
Anzahl überschriebener Services	10	>9	4	1
Anzahl neuer Services	10	>15	-1	-1
Anzahl neuer Attribute	-1	-1	-1	-1
Anzahl direkter Attributzugriffe	-1	-1	10	>5
Anzahl direkter Serviceaufrufe	3	1	-1	-1
Anzahl komplexer Eingabeparameter in allen Services	7	1	10	>2
Anzahl Eingabeparameter in allen Services	9	2	10	>2
Anzahl Services mit komplexer Ausgabe	2	0	-1	-1
Anzahl Services mit mindestens einem Eingabeparameter	6	1	10	>2
Anzahl Services mit mindestens zwei Eingabeparameter gleichen Typs	5	0	-1	-1
Durchschnittliche Anzahl komplexer Eingabeparameter pro Serviceaufruf	8	1,00	7	0,53-0,94
Durchschnittliche Anzahl Eingabeparameter pro Serviceaufruf	10	>1,50	-1	-1
Relative Häufigkeit der Services mit komplexer Ausgabe	2	0%	4	4-49%

Bei der Ermittlung der Korrelationen zwischen den Eigenschaften und der Fehleranzahl in den beteiligten Dateien werden die in Kapitel 5.3 beschriebenen Schritte durchgeführt. Es werden mit Hilfe des statischen Analysewerkzeugs die Abhängigkeiten und ihrer Eigenschaften erhoben und anschließend die Korrelationen mit SPSS ermittelt. Die graphische Darstellung der mittleren Ränge ist im Anhang B: *Statistiken Werkzeug zur Fördergeldverwaltung* zu finden.

Eine übersichtliche Darstellung der Korrelationen für die Version vom 01.07.2008 enthält Tabelle 11. In dieser Version können 11 Eigenschaften identifiziert werden, die mit der Fehleranzahl der abhängigen Datei korrelieren. In der Tabelle ist zu erkennen, dass es sich in den meisten Fällen (Ausnahme *Anzahl überschriebener Services, Anzahl neuer Services, Durchschnittliche Anzahl Eingabeparameter*) um unregelmäßige Zusammenhänge handelt. Die Korrelationen zwischen Eigenschaften und der Fehleranzahl der unabhängigen Datei sind in den rechten zwei Spalten der Tabelle zu erkennen. Hier zeigen sieben Eigenschaften eine Korrelation mit der Fehleranzahl der unabhängigen Datei.

Die Ergebnisse für die Version vom 31.07.2008 sind in Tabelle 12 dargestellt. Für diese Version konnten nur sechs Eigenschaften identifiziert werden, die mit der Fehleranzahl in der abhängigen Datei korrelieren. Für die unabhängige Datei konnten hingegen acht korrelierende Eigenschaften identifiziert werden.

Aufbauend auf diesen zwei Tabellen können die Eigenschaften identifiziert werden, die in beiden Versionen mit der Fehleranzahl der abhängigen bzw. der unabhängigen Datei korrelieren. Hierbei zeigt sich, dass es nur zwei Eigenschaften (*Anzahl überschriebener Services* und *Anzahl neuer Services*) in beiden Versionen gibt, die mit der Fehleranzahl der abhängigen Datei korrelieren. Für die unabhängige Datei können hingegen fünf Eigenschaften (*Anzahl komplexer Eingabeparameter in allen Services, Anzahl aller Eingabeparameter in allen Services, Anzahl Services mit mindestens einem Eingabeparameter, Durchschnittliche Anzahl komplexer Eingabeparameter pro*

Serviceaufruf, Relative Häufigkeit der Services mit komplexer Ausgabe) für die Testfokusauswahl verwendet werden, da sie in beiden Versionen mit der Fehleranzahl der unabhängigen Datei korrelieren.
Es können keine Eigenschaften gefunden werden, die sowohl mit der Fehleranzahl der abhängigen als auch mit der Fehleranzahl der unabhängigen Datei in beiden Versionen korrelieren.

Tabelle 12: Signifikante Gruppen für die Version vom 31.07.2008

	Abhängige Datei		**Unabhängige Datei**	
Eigenschaften	Gruppen	Werte	Gruppen	Werte
Anzahl überschriebener Services	10	>9	-1	-1
Anzahl neuer Services	10	>15	-1	-1
Anzahl neuer Attribute	-1	-1	-1	-1
Anzahl direkter Attributzugriffe	-1	-1	8	3
Anzahl direkter Serviceaufrufe	10	>4	-1	-1
Anzahl komplexer Eingabeparameter in allen Services	-1	-1	10	>2
Anzahl Eingabeparameter in allen Services	-1	-1	10	>2
Anzahl Services mit komplexer Ausgabe	10	>2	2	0
Anzahl Services mit mindestens einem Eingabeparameter	10	>2	10	>2
Anzahl Services mit mindestens zwei Eingabeparameter gleichen Typs	-1	-1	10	>1
Durchschnittliche Anzahl komplexer Eingabeparameter pro Serviceaufruf	-1	-1	7	0,53-0,94
Durchschnittliche Anzahl Eingabeparameter pro Serviceaufruf	4	0,00-0,19	-1	-1
Relative Häufigekeit der Services mit komplexer Ausgabe	-1	-1	4	4-49%

Basierend auf den gefundenen Korrelationen in den zwei früheren Versionen wird der Testfokus für die Version vom 18.09.2009 festgelegt. Die Ergebnisse der Testfokusauswahl für diese Version sind in Abbildung 23 dargestellt. Von den 161421 Abhängigkeiten, die für diese Version identifiziert werden können, werden 31536 Abhängigkeiten (19,54%) eine Testpriorität zugeordnet. Die verbleibenden 80,46% wird *Keine Testpriorität* zugeordnet. Von den ausgewählten Abhängigkeiten erhalten 2778 Abhängigkeiten die *Testpriorität-abhängiger Baustein* und 27534 Abhängigkeiten die *Testpriorität-unabhängiger Baustein*. Obwohl es keine Eigenschaft die sowohl mit der Fehleranzahl des abhängigen als auch mit der Fehleranzahl des unabhängigen Bausteins korreliert, lassen sich zu 1224 Abhängigkeiten die *Testpriorität-beide Bausteine* zuordnen. Dies liegt darin begründet, dass diese Abhängigkeiten mindestens eine Eigenschaft besitzen, die mit der Fehleranzahl des abhängigen Bausteins und mindestens eine Eigenschaft, die mit der Fehleranzahl des unabhängigen Bausteins korreliert.
Abbildung 23 zeigt die mittleren Ränge der Fehleranzahl der abhängigen Datei (hellgrauer Balken) bzw. der unabhängigen Datei (dunkelgrauer Balken). Es ist zu erkennen, dass alle Abhängigkeiten, die die *Testpriorität-abhängiger Baustein* oder *Testpriorität-beide Bausteine* besitzen einen höheren mittleren Rang der Fehleranzahl des abhängigen Bausteins aufweisen. Für ausgewählte Abhängigkeiten mit der *Testpriorität-unabhängiger Baustein* bzw. *Testpriorität-beide Bausteine* gilt ähnliches für die mittleren Ränge der Fehleranzahl in der unabhängigen Datei.

Auch die zweite Fallstudie zeigt auf, dass die Abhängigkeiten, die eine Testpriorität zugeordnet bekommen haben, eine höhere Fehleranzahl in der abhängigen bzw. in der unabhängigen Datei (repräsentiert durch den mittleren Rang) besitzen. Somit sind dies Abhängigkeiten, die im Integrationstestprozess getestet werden müssen.

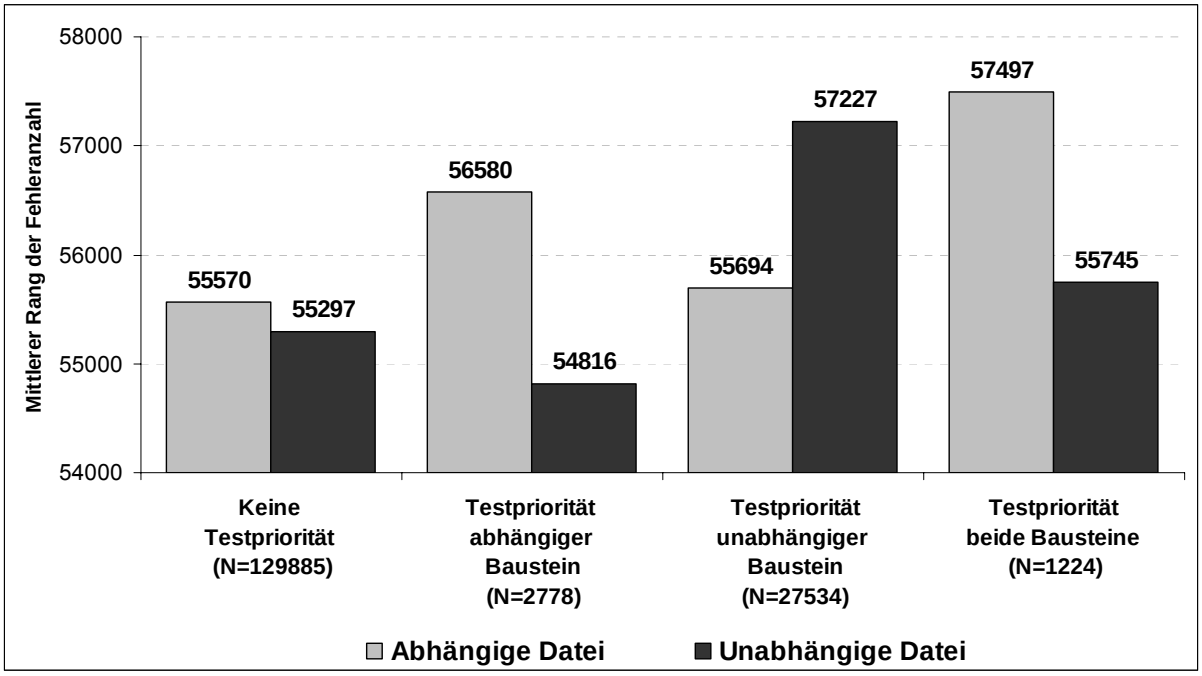


Abbildung 23: Mittlere Ränge (Fehleranzahl) der Abhängigkeiten, eingeteilt nach Testpriorität für Fördergeldverwaltungssoftware Version 18.09.2008

In beiden Fallstudien zeigt sich, dass durch die Anwendung des vorgestellten Ansatzes Abhängigkeiten ausgewählt werden, die einen höheren mittleren Rang bezüglich der Fehleranzahl aufweisen als nicht-ausgewählte Abhängigkeiten. Da der mittlere Rang auf Basis der Fehleranzahl ermittelt wird, bedeutet dies, dass ausgewählte Abhängigkeiten fehleranfälliger sind, als nicht-ausgewählte. Somit hat sich der Ansatz für die zwei Fallstudien als tragbar erwiesen.

6. Integrationsreihenfolge

Eine Integrationsreihenfolge für den Integrationstest beschreibt, wie das System zusammengesetzt werden kann, um die Abhängigkeiten zwischen den Bausteinen zu testen. Verschieden Ansätze für das Festlegen einer Integrationsreihenfolge können in der Literatur gefunden werden. Sie reichen von Ansätzen, die alle Bausteine des Softwaresystems in einem Schritt integrieren und das Zusammenspiel testen, über das Zusammensetzen des Systems in mehreren großen Schritten mit vielen Einzelbausteinen gleichzeitig, bis hin zum schrittweisen Zusammensetzen des Systems Einzelbaustein für Einzelbaustein. Ersteres benötigt für das Zusammensetzen nur einen Integrationsschritt mit einer Integrationsschrittgröße von *N*, wobei *N* die Anzahl der Bausteine des Systems sind. Letzteres benötigt *N-1* Integrationsschritte mit einer Integrationsschrittgröße von eins. Alle anderen Ansätze benötigen zwischen einem und *N-1* Integrationsschritte.
Binder argumentiert in [Bi00] „*A key lesson learned is that incremental integration is the most effective technique: add components a few at a time and then test their interoperability. Trying to integrate most or all of the components in a system at the same time is usually problematic.*" (Seite 630). Die Probleme auf die Binder anspielt sind zum einen das schwierige Identifizieren der Fehlerursache, da die Ursache sich in jedem der beteiligten Bausteine befinden kann. Zum anderen führt das das nicht-systematische Testen zu Problemen, d.h. die Testfälle können nicht systematisch hergeleitet werden, um einzelne Abhängigkeiten gezielt testen zu können. Der inkrementelle Integrationstest hingegen zeigt eine Reihe von Vorteilen auf (vgl. [Bi00]):

- Schnittstellen und Abhängigkeiten werden systematisch getestet. Fehler können somit leichter identifiziert werden.
- Das Auffinden der Fehlerursache wird erleichtert, da sich die Fehlerursache mit großer Wahrscheinlichkeit in einer Abhängigkeit befindet, deren beteiligte Bausteine als letztes integriert wurden.
- Schnittstellen und Abhängigkeiten werden getestet, Fehler identifiziert und beseitigt, bevor neue Bausteine und somit nicht-getestete Abhängigkeiten zum System hinzugefügt werden.

Den Vorteilen der inkrementellen Integration steht ein Nachteil gegenüber, der durch das einmalige Zusammensetzen des Systems in einem Schritt nicht auftreten kann. Bereits integrierte Bausteine können Abhängigkeiten zu Bausteinen aufweisen, die noch nicht integriert und getestet worden sind, wodurch der Aufbau der Testumgebung aufwändiger wird. In jedem Schritt der inkrementellen Integration müssen Bausteine und Abhängigkeiten simuliert werden, die noch nicht integriert wurden. Das Simulieren der Bausteine wird durch Stubs (Platzhalter) übernommen.
Dem Nachteil zum Trotz überwiegen die Vorteile der inkrementellen Integration [Bi00]. Im Fall einer idealen Integration „… *werden die Einzelbausteine schrittweise zu größeren Einheiten zusammengesetzt*" ([SL05] S. 52). Die Integration beginnt mit zwei Bausteinen, um die Abhängigkeiten zwischen diesen zu testen. Anschließend wird der nächste Baustein zu den beiden bereits integrierten Bausteinen hinzugefügt und überprüft, ob die Abhängigkeiten korrekt realisiert sind. Dieser Vorgang wiederholt sich solange, bis alle Bausteine nacheinander zum System hinzugefügt worden sind und alle Abhängigkeiten getestet sind [Me79].
Eine große Herausforderung bei der Erstellung der optimalen Integrationsreihenfolge stellen Softwaresysteme dar, die Zyklen enthalten. Existierende Ansätze zur Ermittlung der

Reihenfolge unterscheiden sich häufig nur darin, wie sie mit Zyklen im System umgehen. Ein Zyklus in einem Softwaresystem existiert, wenn ein Baustein des Systems indirekt von sich selbst abhängt. Ein Baustein hängt indirekt von sich selbst ab, wenn es eine Folge von Bausteinen gibt, in denen ein Baustein als unabhängiger Baustein seines Vorgängers und als abhängiger Baustein seines Nachfolgers fungiert und der Startbaustein gleichzeitig auch der Endbaustein der Folge ist. Zur Veranschaulichung eines Zyklus wird das Beispielsystem aus [BLW01] verwendet, das in Abbildung 24 dargestellt ist. Das Beispielsystem besteht aus acht Bausteinen, die voneinander abhängen. Die Abhängigkeiten sind als Pfeile zwischen den Bausteinen dargestellt. Die Beschriftung der Pfeile liefert Informationen darüber, welcher Art die Abhängigkeit ist. „I“ kennzeichnet eine Vererbungsbeziehung, „As“ eine Assoziationsbeziehung und „Ag“ eine Aggregationsbeziehung. Existierende Ansätze unterscheiden sich darin, wie die unterschiedlichen Arten von Abhängigkeiten behandelt werden. Im vorliegenden Beispiel sind mehrere Zyklen zu finden. Ein einfacher Zyklus tritt beispielsweise zwischen Baustein **H** und **C** auf, da Baustein **H** von **C** und Baustein **C** von Baustein **H** abhängt.

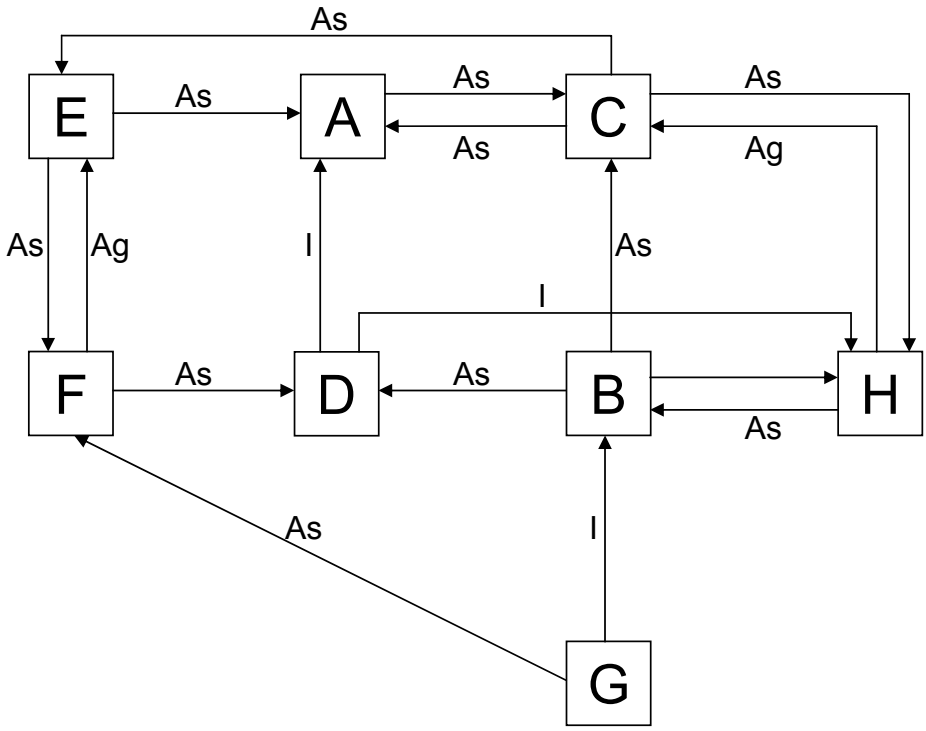


Abbildung 24: Beispielsystem dargestellt als Objektrelationsdiagramm (ORD) nach [BLW01]

Da in einem Graphen sehr viele Zyklen auftreten können und ein Baustein an einer Vielzahl von Zyklen gleichzeitig beteiligt sein kann, wurde der Begriff der *Strongly Connected Components* (kurz SCC) eingeführt. Le Traon et al. definiert einen SCC als einen Teilgraph des Originalgraphs, so dass es für jedes Knotenpaar **x** und **y** innerhalb des Teilgraphs ein Pfad von **x** zu **y** und von **y** zu **x** gibt [TJJ+00]. Alle Bausteine außerhalb des SCC bilden mit keinem der Bausteine innerhalb des SCC einen Zyklus. In Abbildung 24 bilden die Bausteine **A**, **B**, **C**, **D**, **E**, **F**, **H** einen SCC. Ein Algorithmus zur Identifikation von SCCs in einem Graphen wurde von Robert Tarjan bereits 1972 entwickelt [Ta72][1].

[1] Der Pseudocode für den Algorithmus von Tarjan ist im Internet zu finden: http://de.wikipedia.org/wiki/Algorithmus_von_Tarjan_zur_Bestimmung_starker_Zusammenhangskomponenten (03.09.2009)

Viele der existierenden Ansätze verwenden die topologische Sortierung[1], um eine Integrationsreihenfolge zu ermitteln. Der Algorithmus für die topologische Sortierung kann auf Graphen angewendet werden und nutzt die Informationen über die Abhängigkeiten zwischen den Knoten, um eine Reihenfolge zu ermitteln, mit dem Ziel, keine Bausteine simulieren zu müssen. Jedoch kann dieser Algorithmus nur auf azyklische Graphen, d.h. auf Graphen ohne Zyklen, angewendet werden. Somit kann die topologische Sortierung nicht direkt auf den Graph in Abbildung 24 angewendet werden. Zuvor müssen die Zyklen des Graphen entfernt werden. Dies gelingt durch das Entfernen von Kanten. Das Entfernen einer Kante hat zur Folge, dass der unabhängige Baustein einer entfernten Kante durch einen Stub simuliert werden muss. Der Zyklus **C**, **H** aus Abbildung 24 könnte durch die Beseitigung der *As* Kante zwischen **C** und **H** aufgelöst werden. Dies führt im Integrationsschritt der **C** integriert zur Erstellung eines Stubs für **H**.

In der Literatur können viele Ansätze zur Ermittlung der Integrationsreihenfolge gefunden werden. Im nachfolgenden Kapitel 6.1 State of the Art werden existierende Ansätze zur Ermittlung einer Integrationsreihenfolge vorgestellt und deren Vor- und Nachteile aufgezählt. Die Ansätze können in zwei große Klassen eingeteilt werden: Standardstrategien und individuelle Strategien. Standardstrategien beschreiben, wie die Integrationsreihenfolge ermittelt werden kann, ohne bestimmte Optimierungskriterien zu berücksichtigen. Individuelle Strategien hingegen stellen Algorithmen zur Verfügung, die die Integrationsreihenfolge hinsichtlich bestimmter Kriterien optimieren. Diese Kriterien beschäftigen sich ausschließlich mit der Reduzierung des Testaufwands, d.h. sie konzentrieren sich auf die Minimierung der Aufwandserstellung für die Stubs.
Keine der existierenden Ansätze berücksichtigen bei der Integrationsreihenfolgeermittlung die Testfokusauswahl. Kapitel 6.2 stellt einen Ansatz vor, wie die Testfokusauswahl für ein Softwaresystem in die Integrationsreihenfolgeermittlung einfließen kann.

6.1. State of the Art – Integrationsreihenfolgeermittlung

Die Ansätze zur Ermittlung der Integrationsreihenfolge lassen sich in die zwei großen Klassen Standardstrategien und individuelle Strategien einordnen. Standardstrategien geben generische Ansätze vor, wie ein System integriert werden kann ohne die Berücksichtung von Optimierungskriterien. Sie schlagen Vorgehen vor, wie eine Reihenfolge ermittelt werden kann, nennen Vor- und Nachteile, liefern aber keine exakte algorithmische Vorgehensbeschreibung für die Reihenfolgeermittlung. Individuelle Strategien liefern genaue Vorgehensbeschreibungen und Algorithmen, um eine Integrationsreihenfolge zu ermitteln. Tabelle 13 enthält eine kurze Beschreibung für die beiden Klassen, gibt einen Überblick über ihre Eigenschaften sowie Voraussetzung für ihren Einsatz. Der Vorteil der Standardstrategien ist ihre einfache Anwendung und der geringe Aufwand für die Ermittlung der Reihenfolge. Nachteilig ist, dass die ermittelte Reihenfolge nicht immer „optimal“ in Hinblick auf bestimmte Optimierungskriterien, wie z.B. die der Anzahl zu erstellender Stubs ist. Die Anwendung der individuellen Strategien hingegen ist häufig kompliziert, da sie die Anwendung komplexer Algorithmen erfordert. Sie können aber im Gegensatz zu den Standardstrategien, eine „optimale“ Reihenfolge hinsichtlich wichtiger Kriterien ermitteln, wodurch die Reihenfolge besser an die Bedürfnisse des aktuellen Testprozesses angepasst werden kann. (Fast) alle Strategien beider Klassen liefern eine Reihenfolge, in der das System schrittweise zusammengesetzt werden kann. Die Ausnahme bildet die Big-Bang Integration [Bi00].

[1] Der Algorithmus zur Topologischen Sortierung ist in [Ka62] dargestellt. Ein übersichtlicher Pseudocode ist unter: http://en.wikipedia.org/wiki/Topological_sort zu finden (03.09.2009)

Tabelle 13: Gegenüberstellung der Standardstrategien und der Individuelle Strategien

	Standardstrategien	Individuelle Strategien
Kurzbeschreibung	*Die Standardstrategien fassen Strategien zusammen, die nicht hinsichtlich bestimmter Kriterien optimiert wurden. Sie bieten nur generische Vorschläge für eine Reihenfolge an.*	*Die individuellen Strategien ermitteln eine Integrationsreihenfolge, die hinsichtlich gegebener Kriterien optimal ist. Hierbei werden die inneren Strukturen des Systems genau analysiert.*
Eigenschaften	• Schnell anwendbar auf eine gegebene Systemstruktur • Generisch anwendbar auf alle Systemarchitekturen und -strukturen • Liefern keine optimalen Reihenfolgen hinsichtlich Optimierungskriterien	• Ermitteln Reihenfolge anhand Optimierungskriterien • Anwendbar auf alle Systemarchitekturen und -strukturen • Aufwändige Anwendung aufgrund komplexer Algorithmen
Voraussetzungen	• Benötigen ein ungefähres Modell des Systems • Detailgrad des Modells hat auf Reihenfolge keinen Einfluss	• Benötigen ein genaues Modell des Systems, je genauer das Modell • Detaillierungsgrad des Modells hat Einfluss auf Reihenfolge

Sowohl die Standardstrategien als auch die individuellen Strategien lassen sich weiter unterteilen, wie in Abbildung 25 zu sehen ist. Die in der Abbildung grau hinterlegten Strategien sind Strategien, die nicht weiter unterteilt werden können. Die Ansätze in der Literatur lässt sich einem dieser Blätter des Baumes zuordnen.

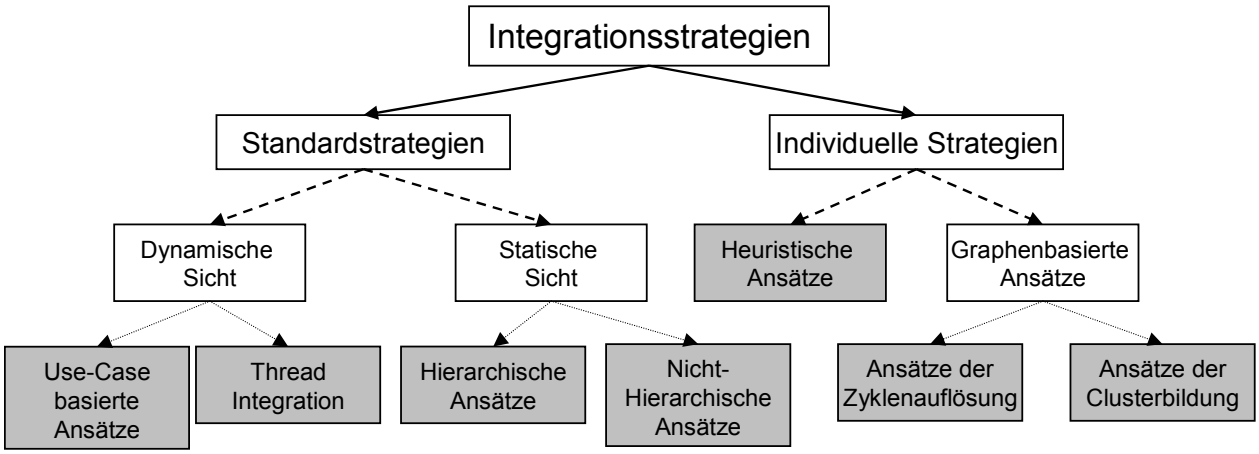


Abbildung 25: Übersicht der Integrationsstrategien

6.1.1. Standardstrategien

Die Standardstrategien lassen sich unter Berücksichtigung der Sicht auf das zu integrierende System in zwei Klassen unterteilen. Die Klasse der *statischen Sicht* betrachtet hauptsächlich die Struktur des Systems und berücksichtigt nicht die Interaktionen zur Laufzeit. Die *dynamische Sicht* betrachtet die Interaktionen der Bausteine zur Laufzeit. Die Struktur der Software ist dabei zweitrangig. Wie in Abbildung 25 zu sehen ist, lassen sich sowohl die Klasse der statischen Sicht als auch die Klasse der dynamischen Sicht erneut unterteilen.

6.1.1.1 Statische Sicht

Die statische Sicht kann in hierarchische und nicht-hierarchische Ansätze unterteilt werden. Hierarchische Ansätze setzen eine hierarchisch aufgebaute Softwarestruktur voraus, um sie einsetzen zu können. Zwei typische Vertreter der hierarchischen Ansätze sind die Top-Down [Bi00], [Be90] und die Bottom-Up Integration [Bi00], [Be90].
Eine weniger bekannte Strategie ist die Sandwich Integration [ER96], [SW02]. Sie stellt eine Kombination aus Top-Down und Bottom-Up Integration dar. Bei dieser Art der Integration integrieren zwei Testteams parallel, wobei unterschieden werden kann zwischen Outside-In-

Integration und Inside-Out-Integration. In Ersterer beginnt ein Team die Integration mit den Bausteinen die sich in der Hierarchie ganz unten befinden und bewegt sich langsam weiter in Richtung „Mitte“ der Hierarchie. Ein zweites Team beginnt mit den Bausteinen, die sich in der Hierarchie oben befinden und integriert nun von Oben zur „Mitte“ der Hierarchie. In einem abschließenden Schritt werden die zwei Teilsysteme der beiden Teams integriert.
Die Inside-Out-Strategie beginnt die Integration in der Mitte der Hierarchie und wird sowohl nach oben als nach unten fortgesetzt. Das Ermitteln der Integrationsreihenfolge erfolgt dabei häufig unter Verwendung eines Algorithmus zur topologischen Sortierung.
Die zweite Klasse in der statischen Sicht stellen die nicht-hierarchischen Ansätze dar. Die Anwendung dieser Strategien stellt keine besondere Anforderung an die Struktur. Vielmehr leitet sich die Integrationsreihenfolge aus Projektfaktoren bzw. Bausteinfaktoren, z.B. Kritikalität oder Fertigstellungstermin der Bausteine, ab. Zwei bekannte Vertreter diese Klasse sind die Ad-hoc-Integration [Bi00], [Be90] und die Integrationsstrategie kritischer Module [ER96], [Or98]. Im ersten Fall werden die Bausteine in der Reihenfolge integriert, in der sie fertig gestellt worden sind. Im letzteren Fall wird die Kritikalität der Bausteine verwendet, um die Reihenfolge festzulegen. Kritische Bausteine werden früher integriert als weniger kritische Bausteine. Jedoch geben die Autoren keine Informationen, wie diese Kritikalität berechnet werden kann.
In die nicht-hierarchischen Ansätze lässt sich auch die Strategie *Big Bang* einordnen, obwohl es sich bei dieser Strategie um keine wirkliche Integrationsstrategie handelt, da sie die Bausteine nicht schrittweise integriert, sondern vorschlägt, alle Bausteine auf einmal zusammenzufügen und dann das Zusammenspiel zu testen. Die Nachteile wurden bereits am Anfang des Kapitels besprochen.
Die Ansätze der statischen Sicht sind in Tabelle 14 zusammenfassend dargestellt.

Tabelle 14: Beschreibung der statischen Standardstrategien

	Hierarchische Ansätze	**Nicht-Hierarchische Ansätze**
Kurzbeschreibung	*Die Integrationsreihenfolge wird anhand einer durch die Systemstruktur vorgegebenen Bausteinhierarchie ermittelt.*	*Bei den Nicht-hierarchischen Ansätzen kann die Integrationsreihenfolge nur schwer geplant werden. Sie kann vom Fertigstellungszeitpunkt oder der Kritikalität der Bausteine abhängen.*
Eigenschaften	• Ermittlung der Reihenfolge anhand einer Bausteinhierarchie • Strategien sind leicht auf gegebene Hierarchien anwendbar	• Ermittlung der Reihenfolgen entweder zufällig (anhand Fertigstellungszeitpunkt) oder anhand der Kritikalität • Reihenfolge hängt von einzelnen Bausteinen und weniger von der Struktur des Gesamtsystem ab
Voraussetzungen	• Hierarchisches Modell des Systems	• Modell des Systems • Fertigstellungszeitpunkte oder Kritikalität der Bausteine
Vertreter	• Top-Down ([Bi00], [Be90]) • Bottom-Up ([Bi00], [Be90]) • Sandwich (Inside-Out, Outside-In) ([ER96], [SW02])	• Ad-Hoc ([Bi00], [Be90]) • Kritische Module ([ER96], [Or98]) • Big-Bang ([Bi00], [Be90], [Me79])

6.1.1.2 Dynamische Sicht

Die zweite Unterklasse der Standardstrategien sind Strategien, die sich mit der *dynamischen Sicht* eines Systems beschäftigen, d.h. bei der Ermittlung der Integrationsreihenfolge spielen die Interaktionen zwischen den Bausteinen eine entscheidende Rolle. Es lassen sich die *Use-Case-basierten* Ansätzen und die *Thread Integration* unterscheiden. Die Ansätze sind in Tabelle 15 zusammengefasst. Die *Use-Case-basierten* Ansätze nutzen die Use Cases, um die Reihenfolge zu ermitteln. Es werden alle Bausteine identifiziert, die zur Realisierung eines Use Cases benötigt werden. Diese Bausteine werden nun schrittweise zusammengesetzt, und es wird getestet, ob sie das im Use Case geforderte Verhalten zeigen. Nachdem alle Bausteine des Use Cases integriert und getestet worden sind, werden

die Bausteine des nächsten Use Cases identifiziert und anschließend zu den bereits integrierten Bausteinen hinzugefügt. Dies wird solange wiederholt, bis alle Bausteine und Use Cases integriert worden sind. Ein bekannter Vertreter der *Use-Case-basierten* Ansätze ist die Kollaborationsintegration (vgl. [Bi00], [Be90]).
Die Thread Integration [ER96], [Or98] gibt im eigentlichen Sinne keine Integrationsreihenfolge für Bausteine vor. Vielmehr beschreibt sie eine Integrationsreihenfolge von parallelen Kontrollprozessen (Threads). Sie eignet sich besonders für die Integration von Systemen mit einer großen Anzahl parallel laufender Kontrollprozesse. In einem ersten Schritt werden ein Kontrollprozess und die damit verbundenen Bausteine integriert und getestet. Im zweiten Schritt kommt ein weiterer Kontrollprozess und seine Bausteine hinzu und so weiter. Die Wechselwirkungen zwischen den Bausteinen und des Kontrollprozesses stehen hierbei im Vordergrund.

Tabelle 15: Beschreibung der dynamischen Standardstrategien

	Use-Case-basierte Ansätze	**Thread Integration**
Kurzbeschreibung	*Die Integrationsreihenfolge der Use-Case-basierten Ansätze wird mit Hilfe von Use Case Beschreibungen ermittelt. Für jeden Use Case werden die daran beteiligten Bausteine und die ausgeführten Interaktionen ermittelt. Die Integration findet pro Use Case statt, d.h. erst werden die Bausteine, die zu einem Use Case gehören schrittweise integriert, dann die Bausteine des nächsten Use Cases usw.*	*Eine Integrationsreihenfolge für parallele Kontrollprozesse. In einem ersten Schritt werden ein Kontrollprozess und die damit verbundenen Bausteine integriert und getestet und anschließend werden die verbleibenden Kontrollprozesse und Bausteine schrittweise hinzugefügt.*
Eigenschaften	• Ermittlung der Reihenfolge von Use Cases • Schrittweise Integration der Bausteine eines Use Cases • Schrittweise Integration der Use Cases	• Ermittlung der Reihenfolgen anhand der parallel laufenden Kontrollprozesse innerhalb eines Systems
Voraussetzungen	• Modell des Systems • Use Case Beschreibungen • Projektion der Use Cases auf die Bausteininteraktionen (Verfolgbarkeit)	• Modell des Systems • Modell der parallel laufenden Kontrollprozesse innerhalb eines Systems
Vertreter	• Kollaborationsintegration ([Bi00], [Be90])	• Thread - Integration ([ER96], [Or98])

6.1.2. Individuelle Strategien

Die individuellen Strategien sind weniger generisch als die Standardstrategien. Sie bieten Algorithmen an, um die Integrationsreihenfolge unter Berücksichtigung von Optimierungskriterien zu ermitteln. Die berücksichtigten Optimierungskriterien beschäftigen sich mit der Reduzierung des Aufwands zur Erstellung der Testumgebung. In erster Linie bedeutet das, dass die Anzahl der zu erstellenden Stubs minimiert wird, da die Erstellung von Stubs laut [BLW03] einen Großteil der Kosten bei der Erstellung der Testumgebung für den Integrationstest beansprucht.
Alle individuellen Strategien benötigen ein Modell des zu integrierenden Systems, das die innere Struktur widerspiegelt. Je genauer dieses Modell ist, desto optimaler wird die Integrationsreihenfolge in Bezug auf die Kriterien bestimmt. Als ein solches Modell können Testabhängigkeitsgraphen (TDG = Test Dependency Graph) oder Objektrelationsdiagramme (ORD) [LTW+00] verwendet werden. Diese Graphen werden entweder aus den Entwurfsartefakten, wie Klassen- oder Sequenzdiagramme, oder direkt aus dem Quelltext des Softwaresystems abgeleitet. Ersteres hat den Vorteil, dass bereits sehr früh begonnen werden kann, den Integrationstest zu planen. Wohingegen die zweite Variante die vollständige Struktur des Systems liefert und somit ein detailliertes Modell zur Verfügung stellen kann.

Innerhalb der Klasse der individuellen Strategien kann zwischen den *graphenbasierten Ansätzen* und den *heuristischen Ansätzen* unterschieden werden. Beide Klassen sind in Tabelle 16 zusammenfassend dargestellt.

Tabelle 16: Erläuterung der Individuellen Strategien

	Graphenbasierte Ansätze	**Heuristische Ansätze**
Kurzbeschreibung	*Graphenbasierte Ansätze benutzen Algorithmen der topologischen Suche, um die Integrationsreihenfolge zu bestimmen. Sollten Zyklen in einem Graph vorhanden sein, so werden diese Zyklen vor der Anwendung der topologischen Suche gesondert behandelt.*	*Das Prinzip der genetischen Algorithmen stammt aus der Biologie. Hierbei werden die verschiedenen Bausteine des Systems als Gene betrachtet. Die Gene werden zu Chromosomen zusammengesetzt. Ein Chromosom entspricht einer möglichen Integrationsreihenfolge. Durch Mutations- und Kreuzungsfunktionen können Chromosomen verändert werden. Mit Hilfe einer Kostenfunktion kann überprüft werden, ob ein Chromosom besser als ein anderes ist.*
Eigenschaften	• Ermittlung der Reihenfolge mit Hilfe von topologischer Suche • Für Topologische Suche muss Graph azyklisch sein • Aufbrechen von Zyklen notwendig	• Kreuzung und Mutationsfunktionen müssen verändern Chromosomen = Reihenfolge der Integration • Kostenfunktion überprüft, ob optimale Reihenfolge bereits gefunden wurde • Basiert eher auf dem Prinzip des Ausprobierens
Voraussetzungen	• Detailliertes Modell des Systems • Modell muss verschiedenen Arten (d.h. Assoziation, Vererbung Aggregation) der Beziehungen zwischen Bausteinen enthalten (z.B. ORD oder TDG)	• Modell des Systems • Verschiedene Parameter für die Kreuzungs- und Mutationsfunktionen • Eine Kostenfunktion
Vertreter	*keine*	• Genetische Algorithmen ([BFL02])

6.1.2.1 Heuristische Ansätze

Die heuristischen Ansätze ermitteln die Integrationsreihenfolge durch „probieren“ und „bewerten“. Es werden viele verschiedene Integrationsreihenfolgen zufällig erstellt und geprüft, welche Reihenfolge die beste ist. Das Ermitteln der besten Reihenfolge erfolgt anhand einer Kostenfunktion, die für jede Reihenfolge einen Zahlenwert ausgibt. Je kleiner der Zahlenwert ist, desto besser ist die Integrationsreihenfolge hinsichtlich der Kostenfunktion. Die verschiedenen Ansätze der Klasse der heuristischen Ansätze unterscheiden sich in der Vorgehensweise, wie die Integrationsreihenfolgen „zufällig“ ermittelt werden. Der einfachste Ansatz beschreibt, dass jede Reihenfolge unabhängig von allen anderen neu ermittelt wird. Es wird N-mal probiert, eine Integrationsreihenfolge zufällig zu erzeugen und die beste Reihenfolge wird anschließend verwendet.
Einen besser strukturierten Ansatz stellt Briand et al. in [BFL02] vor. Sie verwenden genetische Algorithmen, um die Reihenfolge zu ermitteln. Ausgehend von einer zufälligen Menge von Startreihenfolge werden in nachfolgenden Iterationen weitere Reihenfolgen erzeugt. Durch eine Kostenfunktion können die besten Reihenfolgen identifiziert werden, die für das Erzeugen weiterer Reihenfolgen verwendet werden. Nach eine endlichen Anzahl Schritten erhält man die beste Reihenfolge aus.

6.1.2.2 Graphenbasierte Ansätze

Die graphenbasierten Ansätze arbeiten auf einem Modell bzw. einem Graphen des Systems (z.B. ORD oder TDG) und leiten aus diesem Modell die Integrationsreihenfolge mit Hilfe der topologischen Suche ab. Die Graphenbasierten Ansätze lassen sich erneut unterteilen in Ansätze der *Clusterbildung* und Ansätze der *Zyklenauflösung*. Die Ansätze unterscheiden sich durch die Art und Weise, wie Zyklen behandelt werden. Die Ansätze der Clusterbildung fassen Bausteine, die an einem SCC beteiligt sind, zu Cluster zusammen und betrachten diese Ansammlung an Bausteinen als einen Baustein [Or98]. Auf diese Weise werden alle

Zyklen aus dem Graphen entfernt. Nachteil ist, dass Systeme mit wenigen Zyklen, in denen der Großteil der Bausteine enthalten ist, sich schwer integrieren lassen. In diesem Fall ergeben sich die gleichen Probleme wie bei der Big-Bang-Integration, d.h. Fehler lassen sich schwer finden und die Abhängigkeiten zwischen den Bausteinen lassen sich nur ungenügend testen.

Tabelle 17: Erläuterung der graphenbasierten Strategien

	Ansätze der Clusterbildung	**Ansätze der Zyklenauflösung**
Kurzbeschreibung	*Diese Ansätze versuchen Zyklen nicht aufzubrechen, sondern sobald ein Zyklus im Graph erkannt wurde, werden alle Bausteine, die am Zyklus beteiligt sind, als ein Baustein behandelt. Dieser „große" Baustein hat Eingabe- und Ausgabevariablen, möglicherweise auch Zustände usw.* *Sollten mehrere Zyklen vorhanden sein, so werden mehrere „große" Bausteine während der späteren Integration auftreten.*	*Diese Ansätze versuchen, die Zyklen innerhalb eines Graphen möglichst effizient aufzubrechen. Dies geschieht durch das Entfernen von Kanten zwischen Bausteinen. Durch das Entfernen müssen Bausteine durch Stubs ersetzt werden. Hierbei unterscheiden sich die Ansätze auf welcher Granularität die Kanten entfernt werden, entweder auf Bausteinebene oder auf Methodenebene. Auf Bausteinebene muss beim Entfernen einer Kante immer der ganze Baustein simuliert werden. Auf der Methodenebene ist das Simulieren von ganzen Bausteinen nur selten notwendig.*
Eigenschaften	• Keine Zyklenaufbrechung, d.h. Bausteine innerhalb eines Zyklus werden als ein Baustein behandelt • Anschließende topologische Sortierung	• Zyklenauflösung nach verschiedenen Ansätzen • Anschließende topologische Sortierung
Voraussetzungen	• Modell des Systems	• Modell des Systems • Modell muss verschiedene Arten (d.h. Assoziation, Vererbung Aggregation) der Beziehungen zwischen Bausteinen enthalten (z.B. ORD oder TDG)
Vertreter	• Alessandro Orso [Or98]	• Bausteinebene: o Kung et al. [KGH+95a], [KGH+95b] o Tai und Daniels [TD97] o Labiche et al. [LTW+00] o Le Traon et al. [TJJ+00] o Briand et al. [BLW03] • Methodenebene: o Winter [Wi98] o Badri et al. [BBB05]

Die Ansätze der Zyklenauflösung beschäftigen sich mit dem Aufbrechen der Zyklen durch das Entfernen von Abhängigkeitsbeziehungen[1]. Hierbei werden drei Arten von Abhängigkeiten unterschieden: Assoziationen, Aggregrationen und Vererbung (vgl. [BLW01]). Während frühe Ansätze (z.B. [KGH+95a], [KGH+95b]) zufällig Abhängigkeiten auswählen und diese entfernen, wählen neuere Ansätze die zu löschenden Abhängigkeiten zielgerichtet aus. So werden beispielsweise keine Vererbungs- und Aggregationsbeziehungen gelöscht, sondern nur noch Assoziationen. Auf diese Weise wird die Stuberstellung erleichtert, da das Simulieren einer Oberklasse schwieriger ist, als das Simulieren eines (oder mehrerer) Serviceaufrufe [BLW01]. Dabei werden nur die Abhängigkeiten zwischen Bausteinen auf Bausteinebene betrachtet, z.B. in [KGH+95a], [KGH+95b], [LTW+00] oder [BLW03]. Die Ansätze, die Abhängigkeiten auf Bausteinebene betrachten, unterscheiden sich in der Art und Weise, wie die zu löschenden Abhängigkeiten ausgewählt werden. Dies geschieht entweder zufällig [KGH+95a], [KGH+95b] oder durch die Verwendung von Gewichtungen, wobei sich die Ansätze in der Berechnung der Gewichtungen unterscheiden.

[1] Abhängigkeiten, die zum Zweck der Zyklenauflösung entfernt wurden müssen durch Stubs ersetzt werden, d.h. der unabhängige Baustein muss simuliert werden.

In [BBB05] und [Wi98] berücksichtigen die Autoren zusätzlich die Abhängigkeiten auf Serviceebene, d.h. wenn ein Service einen anderen Service aufruft. Durch diese Betrachtung können die Autoren zwei Arten von Zyklen unterschieden: Nicht-effektive Zyklen und effektive Zyklen. Nicht-effektive Zyklen sind Zyklen, die im ORD oder TDG zwischen zwei Klassen sichtbar sind, aber auf Serviceebene nicht zu einem Zyklus führen. Beispiel:
Eine Klasse **A** besitzt zwei Methoden **m1** und **m2**. Eine zweite Klasse **B** besitzt zwei Methoden **m3** und **m4**. Ein *nicht-effektiver Zyklus* entsteht, wenn in der Methode **m1** aus **A** die Methode **m3** aus **B** und in Methode **m4** aus **B** die Methode **m2** aus **A** aufgerufen wird. Somit wäre auf Klassenebene ein Zyklus vorhanden, aber nicht auf Methodenebene. Um diese beiden Klassen zu integrieren wird auf die „partielle“ Integration zurückgegriffen werden, d.h. nur Teile einer Klasse (die betroffenen Methoden) werden integriert. Nur für das Aufbrechen von effektiven Zyklen müssten Stubs erstellt werden. Ein *effektiver Zyklus* ist ein Zyklus der während der Interaktionen zwischen Klassen auftreten kann. Im oberen Beispiel wäre ein effektiver Zyklus, wenn Methode **m1** aus **A** die Methode **m2** aus **A** aufruft, Methode **m2** aus **A** dann Methode **m3** aus **B** aufruft und Methode **m3** aus **B** die Methode **m1** aus **A** wieder aufruft. Somit wäre ein „echter“ Zyklus vorhanden, der nur durch Stubs aufgebrochen werden könnte. Die Ansätze der graphenbasierten Strategien setzen Kenntnisse voraus, die nur im Quellcode oder in sehr detaillierten Entwurfsmodellen vorhanden sind.

Alle vorgestellten Strategien können verwendet werden, um eine Reihenfolge für das Zusammensetzen des Softwaresystems im Integrationstest zu ermitteln. Jedoch berücksichtigt keine Strategie den ausgewählten Testfokus, d.h. die Abhängigkeiten, die im Integrationstest getestet werden müssen. Eine Reihenfolge, die den Testfokus berücksichtigt, integriert die Bausteine des Systems in der Art, dass die Testfokus-Abhängigkeiten früh integriert werden.

6.2. Ansatz zur Testfokusberücksichtigung

Ein Ziel der vorliegenden Arbeit ist die Entwicklung eines Ansatzes zur Bestimmung einer Integrationsreihenfolge, wobei die Reihenfolge sowohl den Testfokus als auch den Simulationsaufwand berücksichtigen soll. In einem ersten Schritt ist es notwendig, ein Verfahren vorzustellen, welches für eine gegebene Integrationsreihenfolge die Berücksichtigung des Testfokus bestimmt (Kapitel 6.2.1). Für existierende Algorithmen wird anschließend untersucht, wie gut sie den Testfokus bei der Ermittlung der Integrationsreihenfolge berücksichtigen. Diese Algorithmen werden in Kapitel 6.2.2 kurz vorgestellt. Diese Algorithmen werden auf mehrere Softwaresysteme angewendet und bestimmt, wie gut sie den Testfokus in der ermittelten Integrationsreihenfolge berücksichtigen. Zusätzlich wird ein neuer Ansatz vorgestellt, der den Testfokus in einer Integrationsreihenfolge zu 100% berücksichtigt.
Kapitel 6.3 erweitert eine Teilmenge der in Kapitel 6.2.2 vorgestellten Algorithmen mit dem Ziel, sowohl den Testfokus als auch den Simulationsaufwand zu berücksichtigen. Diese angepassten Algorithmen werden erneut auf verschiedene Softwaresysteme angewendet, um zu überprüfen, wie gut sie Testfokus und Simulationsaufwand berücksichtigen. Die Ergebnisse dieser Anwendung werden ebenfalls in Kapitel 6.3 vorgestellt.

6.2.1. Messen der Testfokusberücksichtigung

Ausgehend von den Informationen über den Testfokus einzelner Abhängigkeiten soll eine Integrationsreihenfolge **IR** hinsichtlich der Berücksichtigung dieses Testfokus untersucht werden. Eine Integrationsreihenfolge berücksichtigt den Testfokus vollständig, wenn alle Abhängigkeiten, die als Testfokus ausgewählt worden sind, vor Abhängigkeiten, die nicht als Testfokus ausgewählt wurden, integriert und getestet werden.

Der Ansatz der Testfokusauswahl, wie er in Kapitel 5 vorgestellt wurde, ordnet jeder Abhängigkeit eine Testpriorität zu: *Testpriorität beide Bausteine*, *Testpriorität abhängiger Baustein*, *Testpriorität unabhängiger Baustein* und *keine Testpriorität*. Ausgehend von diesen Informationen kann bestimmt werden, welche Abhängigkeiten vor welchen Abhängigkeiten integriert werden müssen. Im Rahmen dieser Arbeit gilt, dass Abhängigkeiten, die eine Testpriorität (*Testpriorität beide Bausteine*, *Testpriorität abhängiger Baustein* oder *Testpriorität unabhängiger Baustein*) besitzen, vor Abhängigkeiten integriert werden müssen, die *keine Testpriorität* besitzen. Eine Reihenfolge innerhalb der Abhängigkeiten mit Testpriorität wird nicht verwendet. Eine Abhängigkeit gilt als integriert, wenn die zwei beteiligten Bausteine der Abhängigkeit integriert wurden.
In einem ersten Schritt müssen alle Bausteine identifiziert werden, die an Abhängigkeiten mit Testpriorität beteiligt sind. Diese Bausteine müssen als erstes integriert werden, um die Bedingung zu erfüllen, dass Abhängigkeiten mit Testpriorität früh integriert werden. Die Menge wird als $\mathbf{B_{TP}}$ (Bausteine mit Testpriorität) bezeichnet. Im nächsten Schritt werden für die zu untersuchende Integrationsreihenfolge **IR** die ersten *X* Bausteine dieser Reihenfolge identifiziert, wobei *X* die Anzahl der Bausteine in $\mathbf{B_{TP}}$ (d.h. $X = |\boldsymbol{B_{TP}}|$) ist. Anschließend wird für jede Abhängigkeit ***a*** mit einer Testpriorität überprüft, ob sich die beiden beteiligten Bausteine von ***a*** unter den ersten *X* Bausteine der Reihenfolge **IR** befinden. Ist dies der Fall, befinden sich die zwei Bausteine in der Menge der Bausteine, die früh integriert werden. Befindet sich einer oder beide Bausteine von ***a*** nicht unter den ersten X Bausteinen von **IR**, so kann die Abhängigkeit nicht rechtzeitig integriert und getestet werden. Die Abhängigkeit wird zu spät integriert. Zur Bestimmung, wie gut die Integrationsreihenfolge den Testfokus berücksichtigt, werden alle Abhängigkeiten gezählt, die zu spät integriert werden. Je größer die Anzahl der zu spät integrierten Abhängigkeiten ist, desto schlechter berücksichtigt die Reihenfolge den Testfokus.
Das Vorgehen soll am Beispiel in Abbildung 24 (Seite 110) veranschaulicht werden. Für die 17 Abhängigkeiten werden die Abhängigkeiten **E→A**, **E→F**, **C→H** und **H→C** willkürlich als Testfokus ausgewählt. Für die Reihenfolge **ACHGEDBF** soll nun geprüft werden, wie gut sie den Testfokus der Abhängigkeiten unterstützt. Im ersten Schritt werden die Bausteine ausgewählt, die an Abhängigkeiten mit Testfokus beteiligt sind. Dies sind die Bausteine **A**, **C**, **E**, **F** und **H**. Somit wurden fünf (*X=5*) Bausteine ausgewählt, die vor den anderen Bausteinen integriert werden müssen, um die Abhängigkeiten mit Testfokus früh zu integrieren. Für jede Abhängigkeit mit Testfokus wird nun geprüft, ob die beiden beteiligten Bausteine unter den ersten fünf Bausteinen der Reihenfolge zu finden sind. Dies trifft für die Abhängigkeiten **A→E**, **C→H** und **H→C** zu. Die Abhängigkeit **E→F** wird zu spät integriert, da der Baustein **F** erst als letztes integriert wird und somit die Integration der Abhängigkeit ebenfalls erst im letzten Schritt stattfindet. Dennoch würde in diesem Beispiel die ermittelte Integrationsreihenfolge den Testfokus sehr gut berücksichtigen, da 75% der Abhängigkeiten mit Testfokus frühzeitig integriert würde. Ein Beispiel für eine Reihenfolge, die den Testfokus nur sehr schlecht berücksichtigt, wäre die Reihenfolge **GBDFEACH**. In diesem Fall würde nur eine Abhängigkeit (**E→F**) frühzeitig integriert.

6.2.2. Testfokusberücksichtigung in existierenden Ansätzen

Der Ansatz zum Messen der Testfokusberücksichtigung für eine gegebene Integrationsreihenfolge wird in diesem Kapitel verwendet, um existierende Ansätze zur Integrationsreihenfolge zu untersuchen. Ziel ist es, zu überprüfen, inwieweit bereits existierende Ansätze die Testfokusauswahl berücksichtigen. Die ausgewählten Verfahren sind in erster Linie entwickelt worden, um den Simulationsaufwand für die Integration zu minimieren. Keines der existierenden Verfahren wurde konzipiert, um den Testfokus explizit zu berücksichtigen.

Zur Überprüfung der Ansätze werden diese auf neun verschiedene Softwaresysteme angewendet, um eine Integrationsreihenfolge zu ermitteln. Dies ist zum einen die Entwicklungsumgebung *Eclipse Version 3.0* als Beispiel für ein sehr großes Softwaresystem. Für dieses Softwaresystem wurde bereits der Testfokus in Kapitel 5.4.1 festgelegt. Zum anderen werden zusätzlich acht kleinere Softwaresysteme verwendet, um die Ergebnisse der Untersuchung zu unterstützen. Für diese Softwaresysteme wird der Testfokus unter Verwendung der bekannten Fehleranzahl der Bausteine ermittelt. Für jedes dieser Systeme ermittelte Timea Illes-Seifert im Rahmen ihrer Arbeiten in [IP08a], [IP08b], [IP09] die Fehleranzahl pro Quelltextdatei. Um eine Einteilung der Abhängigkeiten gemäß dem Testfokus durchzuführen, werden die Abhängigkeiten nach der Fehleranzahl im abhängigen und nach der Fehleranzahl im unabhängigen Baustein gruppiert. Hierzu werden wie auch bei der Testfokusauswahl Dezilgruppen verwendet. Einer Abhängigkeit wird die *Testpriorität beide Bausteine* zugeordnet, wenn die Abhängigkeit sich sowohl in die höchste Gruppe (10), eingeteilt nach Fehleranzahl des abhängigen Bausteins als auch in die höchste Gruppe (10), eingeteilt nach Fehleranzahl des unabhängigen Bausteins, einordnen lässt. Eine Einteilung in die *Testpriorität abhängiger Baustein* bekommt eine Abhängigkeit, wenn sie sich nur in die höchste Gruppe, eingeteilt nach Fehleranzahl im abhängigen Baustein, einordnen lässt. Gleiches gilt für die Einordnung in *Testpriorität unabhängiger Baustein*. Alle verbleibenden Abhängigkeiten erhalten die Einordnung *Keine Testpriorität*.
Die in der Untersuchung verwendeten Softwaresysteme sind Open Source Programme unterschiedlicher Größe. Dabei handelt es sich um die Programme *Apache ANT* [ANT09], *Apache FOP* [FOP09], *Apache Chemistry Development Kit* [CDK09], *Free Network Project* [FRE09], *Jetspeed* [JET09], *JMol* [JMO09], *OSCache* [OSC09] und *TVBrowser* [TVB09]. Eine Übersicht über die Größe der verwendeten Softwaresysteme ist in Tabelle 18 zu finden. Die Tabelle gibt an, aus wie vielen Quelltextdateien das betrachtete Softwaresystem besteht. Darüber hinaus gibt sie an, wie viele Vererbungsbeziehungen und wie viele Assoziationsbeziehungen im Softwaresystem enthalten sind. Eine Assoziationsbeziehung ist eine Client/Server-Abhängigkeit (vgl. Kapitel 4.1.1.1), in der die Klassen der abhängigen Datei Services und/oder Attribute an Klassen der unabhängigen Datei aufrufen bzw. zugreifen. Die Informationen über die Anzahl der Quelltextdateien und die Abhängigkeiten können mit Hilfe des Quelltextanalysators Sissy [Si09] identifiziert werden, da die untersuchten Softwaresysteme in der Programmiersprache Java realisiert sind.

Tabelle 18: Überblick über Größe der verwendeten Beispielsysteme

	Anzahl Quelltextdateien	**Anzahl Vererbungs-beziehungen**	**Anzahl Assoziations-beziehungen**
OSCache	**110**	**43**	**282**
JMol	**323**	**172**	**1455**
Freenet	**456**	**262**	**1852**
TVBrowser	**818**	**393**	**4053**
Apache FOP	**1006**	**773**	**4468**
Apache Chemistry Development Kit	**1022**	**751**	**5289**
Apache ANT	**1053**	**1027**	**4236**
Jetspeed	**1347**	**991**	**3733**
Eclipse	**10133**	**10375**	**91455**

Für die neun Softwaresysteme wird mit unterschiedlichen Ansätzen die Integrationsreihenfolge ermittelt. Verwendet werden zum einen die graphenbasierten Ansätze von Tai und Daniels [TD97], Le Traon et al. [TJJ+00] und Briand [BLW03], da sie bereits in [BLW03] hinsichtlich ihrer Berücksichtigung des Simulationsaufwands untersucht

wurden. Dies bietet eine Möglichkeit, die gewonnen Erkenntnisse mit den Fallstudien aus [BLW03] zu vergleichen. Zum anderen werden die heuristischen Ansätze des genetischen Algorithmus von Briand et al. [BFL02] und der zufallsbasierte Algorithmus verwendet. Durch ihre Kostenfunktion (vgl. Kapitel 6.2.2.4) können sie leicht um weitere Optimierungskriterien erweitert werden, was in Hinblick auf die Berücksichtigung des Testfokus eine entscheidende Rolle spielt.
Die Gruppe der heuristischen Ansätze wird um einen neuen Ansatz basierend auf den Konzepten der Simulated Annealing Algorithmen (z.B. in [BR84]) erweitert, da sie eine bisher noch nicht verwendete Möglichkeit für die Integrationsreihenfolgeermittlung darstellt. Ziel ist es zu untersuchen, inwieweit diese sechs Ansätze den Testfokus berücksichtigen. Ergänzt wird diese Untersuchung um die Betrachtung, wie gut die unterschiedlichen Ansätze den Simulationsaufwand optimieren. In den nachfolgenden Kapiteln 6.2.2.1 bis 6.2.2.6 werden die einzelnen Ansätze zur Integrationsreihenfolge und ihre Funktionsweise kurz vorgestellt und am Beispiel in Abbildung 24 erläutert. Im Anschluss (Kapitel 6.2.2.7) wird ein Ansatz vorgestellt, der die Testfokusauswahl zu 100% berücksichtigt, aber nur bedingt den Simulationsaufwand einschließt. Im abschließenden Kapitel 6.2.2.8 werden die Ergebnisse der Anwendungen der Algorithmen auf die neun Beispielsysteme vorgestellt und die Algorithmen anhand einer Liste von Vergleichsmetriken miteinander verglichen.

6.2.2.1 Algorithmus von Tai und Daniels

Der Algorithmus von Tai und Daniels [TD97] wird auf einen zyklischen Graphen angewandt und löst darin schrittweise Zyklen auf, indem er einzelne Kanten entfernt. Wie auch der in dieser Arbeit später vorgestellte Ansatz von Briand vermeidet der Algorithmus von Tai und Daniels das Aufbrechen von Vererbungsbeziehungen, da diese den größten Aufwand bei der Simulation von Bausteinen ausmachen. Der Algorithmus arbeitet auf einem Objektrelationsdiagramm, wie es in Abbildung 24 zu sehen ist, und unterscheidet zwischen drei Arten von Abhängigkeiten: Vererbungen, Aggregationen und Assoziationen, behandelt aber Vererbungen und Aggregationen gleich. Tai und Daniels nutzen dabei die Konzepte der objektorientierten Programmiersprachen aus, in denen an einem Zyklus immer mindestens eine Assoziationskante beteiligt ist und Vererbungs- und Aggregationskanten keine Zyklen bilden können.
Der Algorithmus arbeitet in zwei Schritten. Im ersten Schritt werden die so genannten Major-Level Nummern vergeben. Zur Berechnung der Major-Level Nummern werden die Assoziationskanten ausgeblendet. Da der so entstandene Graph aufgrund der Eigenschaften der objektorientierten Programmiersprachen keine Zyklen mehr besitzen kann, kann ein Algorithmus ähnlich der topologischen Suche ausgeführt werden. Dem ersten Major Level werden alle Bausteine zugeordnet, die keine Vererbungs- oder Aggregationskanten besitzen, in denen sie als abhängiger Baustein fungieren. Dem zweiten Major-Level werden die Bausteine zugeordnet, die nur Vererbungs- oder Aggregationskanten zu den Bausteinen des ersten Major-Levels besitzen, in denen Sie als unabhängiger Baustein auftreten. Vererbungs- und Aggregationskanten innerhalb eines Levels sind nicht gestattet. Auf diese Weise kann jedem Baustein eine eindeutige Major-Level Nummer zugeordnet werden. Für das Beispiel in Abbildung 24 befinden sich auf dem Major-Level 1 die Bausteine **A**, **E** und **C**. Der zweite Major-Level besteht aus den Bausteinen **F** und **H**, der dritte Level aus den Bausteinen **D** und **B** und das letzte Level besteht aus dem Baustein **G**. Die Major-Levels stellen eine erste Integrationsreihenfolge dar. Jeder Baustein des Major-Level x muss vor den Bausteinen des Major-Level x+1 integriert werden.
Im zweiten Schritt des Algorithmus werden jetzt die Assoziationskanten berücksichtigt. Durch die Assoziationskanten können innerhalb eines Major-Levels Zyklen entstehen, die schrittweise aufgelöst werden müssen. Um zu bestimmen, welche Assoziationskante zu löschen ist, wird für jede Assoziationskante im Zyklus ein Gewicht berechnet. Das Gewicht

ermittelt sich durch „*… the number of the incoming dependencies of the origin node […] plus the number of outgoing dependencies of the target node"* ([BLW03], Seite 596). Es werden jedoch nur Assoziationskanten betrachtet. Die Aggregations- und Vererbungskanten fließen bei der Gewichtsbestimmung nicht mit ein. Die Kante mit dem größten Gewicht wird entfernt, da Tai und Daniels davon ausgehen, dass durch das Löschen der Kante mit dem größten Gewicht auch die größte Anzahl an Zyklen aufgebrochen wird. Sollten zwei Kanten das gleiche Gewicht aufweisen, wird zufällig eine Kante zum Löschen ausgewählt. Für die Bausteine jedes Major-Levels werden solange die Assoziationskanten gelöscht, bis keine Zyklen innerhalb der Major-Levels zu finden sind. Befinden sich auf einem Major-Level keine Zyklen mehr, so kann mit Hilfe der topologischen Suche eine Reihenfolge für jeden Major-Level festgelegt werden. Jeder Baustein erhält auf diese Weise neben der Major-Level Nummer noch eine Minor-Level Nummer, die angibt, an welcher Position innerhalb des Major-Levels der Baustein integriert wird. Für das Beispiel in Abbildung 24 befindet sich nur für die Bausteine im ersten Major-Level ein Zyklus. **A→C**, **C→A**, **C→E** und **E→A** bilden den Zyklus. Für jede dieser Abhängigkeiten wird das Gewicht berechnet. Das Gewicht für **A→C** ist 4, da der Baustein **A** zwei eingehende (**E→A** und **C→A**) und Baustein **C** zwei ausgehende (**C→E** und **C→A**) Assoziationsbeziehungen innerhalb des betrachteten Zykluses besitzt. Für die verbleibenden drei Assoziationen ergeben sich folgende Gewichte: **C→A** (2), **C→E** (2) und **E→A** (2). Somit würde die Kante A→C aus dem Graphen entfernt und der Zyklus wäre somit aufgebrochen. Mit Hilfe der topologischen Sortierung werden den Bausteinen des ersten Major-Levels die folgenden Minor-Level Nummern zugeordnet: **A** (1), **E** (2) und **C** (3). Die Bausteine **F** und **H** des zweiten Major-Levels erhalten beide die Minor-Level Nummer 1, da zwischen ihnen kein Zyklus auf dem Level existiert. Das heißt, dass es für den Simulationsaufwand keine Rolle spielt, ob erst **F** und dann **H** oder umgekehrt integriert wird. Auf dem dritten Level gibt es die Assoziationsbeziehung **B→D**, d.h. der Baustein **D** muss vor dem Baustein **B** integriert werden. Aus diesem Grund wird dem Baustein **B** die Minor-Level Nummer 1 und Baustein **B** die Minor-Level Nummer 2 zugeordnet.
Die Major-Level und Minor-Level Nummern werden abschließend verwendet, um die finale Integrationsreihenfolge festzulegen. Im Beispiel ergibt sich die Reihenfolge **AECFHDBG**. Im ersten Integrationsschritt werden für die Integration von **A** und **E** die Stubs **C** und **F** benötigt. Im zweiten Schritt wird der Stub von **C** durch den echten Baustein **C** ersetzt. Dafür wird zusätzlich ein Stub für den Baustein **H** benötigt. Im dritten Schritt wird der Stub von **F** durch den echten Baustein **F** ersetzt. Dadurch muss der Baustein **D** simuliert werden. Im vierten Schritt wird Baustein **H** integriert, d.h. der echte Baustein **H** ersetzt den Stub **H**. Da **H** für seine korrekte Funktionsweise von **B** abhängt, muss auch der Baustein **B** simuliert werden. Die verbleibenden Bausteine können ohne zusätzliche Stubs integriert werden. Diese Integrationsreihenfolge verlangt somit die Realisierung von fünf Stubs. Darüber hinaus müssen bei der ermittelten Reihenfolge die Abhängigkeiten **A→C**, **E→F**, **C→H**, **F→D** und **H→B** simuliert werden. In diesem Fall wäre die Anzahl der zu simulierenden Bausteine gleich der Anzahl der zu simulierenden Abhängigkeiten.

6.2.2.2 Algorithmus von Le Traon

In [TJJ+00] stellen Le Traon, Jéron, Jézéquel und Morel einen Algorithmus vor, der den Algorithmus von Tarjan [Ta72] verwendet, um die Strongly Connected Components (SCC) zu identifizieren, d.h. alle Bausteine zu identifizieren, die zu einem Zyklus gehören (vgl. Einleitung zu Kapitel 1). Alle identifizierten SCC werden unabhängig voneinander betrachtet. Für jeden SCC werden Kanten identifiziert, die entfernt werden können, um den SCC aufzulösen. Der Algorithmus von Le Traon et al. sieht vor, dass die eingehenden Kanten des Bausteins, der das größte Gewicht hat, gelöscht werden. Das Gewicht des Bausteins ist die Summe aus den eingehenden und ausgehenden Frontkanten des Knotens. Eine Frontkante

ist eine Kante, die von einem Baustein zu einem seiner Vorfahren geht, wobei die Vorfahrenbeziehung durch die Anwendung des Algorithmus von Tarjan definiert wird. Ein Vorfahre eines Bausteins **A** ist ein Baustein, der vor dem Baustein **A** vom Algorithmus von Tarjan betrachtet wurde. Da der Algorithmus von Tarjan mit einem beliebigen Baustein beginnen kann, ist der Algorithmus von Le Traon et al. hochgradig nicht deterministisch, da die Definition, was eine Frontkante ist, von der Anwendung des Algorithmus abhängt (vgl. [BLW03]).
Der Algorithmus von Le Traon et al. löscht alle eingehenden Kanten des Bausteins mit dem höchsten Gewicht Dabei unterscheidet er nicht, ob es sich um Vererbungs-, Assoziations- oder Aggregationskanten handelt.
Um die Anwendung des Algorithmus von Le Traon am Beispiel aus Abbildung 24 et al. zu veranschaulichen, muss im ersten Schritt definiert, werden in welcher Reihenfolge der Algorithmus von Tarjan die Bausteine abarbeitet, um die Frontkanten bestimmen zu können. Im Beispiel werden die Bausteine in der Reihenfolge **G, F**, **D**, **H**, **B**, **C**, **E**, **A** (vgl. [BLW03] Seite 598) abgearbeitet. Es wird ein SCC identifiziert, bestehend aus den Bausteinen **F**, **D**, **H**, **B**, **C**, **A** und **E**. Der Baustein **G** ist an diesem SCC nicht beteiligt. Durch die Anwendung des Algorithmus von Tarjan in der oben beschriebenen Reihenfolge ergibt sich, dass die Kanten **E→F**, **B→D**, **A→C**, **C→H** und **B→H** als Frontkanten markiert werden, da jeweils der unabhängige Baustein vor dem abhängigen Baustein betrachtet wird und somit eine Kante von einem Baustein zu seinem Vorgänger existiert. Im nächsten Schritt wird für alle Bausteine innerhalb des SCC das Gewicht, d.h. die Summe der eingehenden und ausgehenden Frontkanten, berechnet. Die Gewichte sind **F**(1), **D**(1), **H**(2), **B**(2), **C**(2), **E**(1) und **A**(1). Anschließend wird der Baustein mit dem höchsten Gewicht ausgewählt. Da drei Bausteine **H**, **B**, **C** zur Auswahl stehen, wird aus diesen drei Bausteinen einer zufällig ausgewählt. Wie auch in [BLW03] fällt die Auswahl auf den Baustein **B**. Alle eingehenden Kanten dieses Bausteins werden gelöscht. Dies betrifft die Kante **H→B**. Auf diese Weise konnte der SCC aufgebrochen werden. Nun wird der Algorithmus von Tarjan wieder auf die Bausteine angewendet, wobei ein weiterer SCC identifiziert wird, bestehend aus **H**, **D**, **C**, **A**, **E** und **F**. Die Bausteine **G** und **B** sind an keinem SCC mehr beteiligt. Anschließend werden wieder die Frontkanten identifiziert und Bausteine ausgewählt, deren eingehende Kanten entfernt werden. Dieser Vorgang wird solange wiederholt, bis eine Anwendung des Algorithmus von Tarjan auf alle Bausteine keine SCC identifiziert. Eine Reihenfolge, die der Algorithmus liefert, wenn Tarjan immer mit dem Baustein G begonnen wird ist **AHDEFCBG** (entnommen aus [BLW03] Seite 598). Für diese Reihenfolge müssen drei Bausteine (**C**, **B**, **F**) und vier Abhängigkeiten (**A→C**, **H→C**, **H→B**, **E→F**) simuliert werden. Durch die gute Wahl von G als Startpunkt für den Algorithmus von Tarjan müssen weniger Bausteine und Abhängigkeiten als bei der Anwendung des Algorithmus von Tai und Daniels simuliert werden. Das Beispiel in [BLW03] zeigt auch, dass bei der „schlechten“ Wahl des Startpunkt **A** für den Algorithmus von Tarjan sieben Abhängigkeiten simuliert werden müssen.

6.2.2.3 Algorithmus von Briand

Der Algorithmus von Briand, Labiche und Wang, vorgeschlagen in [BLW03], basiert auf der Grundidee von Le Traon et al., unterscheidet sich aber in der Berechnung der Gewichtung und somit in der Auswahl der zu löschenden Abhängigkeiten. Briand et al. betrachten nur Assoziationskanten, da nach Aussage von Tai und Daniels [TD97] an einem Zyklus mindestens eine Assoziationskante beteiligt sein muss. Wie auch Le Traon et al. identifizieren Briand et al. die SCC mit Hilfe des Algorithmus von Tarjan. Anschließend wird für jede Assoziationskante innerhalb des SCC ein Gewicht berechnet. Das Gewicht errechnet sich aus der Anzahl eingehender Kanten des abhängigen Bausteins, multipliziert mit der Anzahl ausgehender Kanten des unabhängigen Bausteins. Briand et al. modifizieren somit die Gewichtsberechnung von Tai und Daniels. Tai und Daniels addieren die Anzahl der

eingehenden Kanten des abhängigen Bausteins mit der Anzahl der ausgehende Kanten des unabhängigen Bausteins wohingegen Briand et al. diese zwei Werte multiplizieren. Anschließend wird die Assoziationskante mit dem größten Gewicht gelöscht und der Algorithmus von Tarjan auf den neuen Graph angewendet, der die gelöschte Kante nicht mehr enthält. Anschließend wird wieder eine Assoziationskante ausgewählt, die gelöscht wird. Dieser Vorgang wird solange wiederholt, bis keine Zyklen mehr im Graphen enthalten sind. Der Algorithmus von Briand et al. löscht nur Assoziationskanten und keine Aggregations- oder Vererbungskanten.
Im Vergleich zum Algorithmus von Le Traon ist die Auswahl der zu löschenden Kanten nicht vom Startpunkt des Tarjan Algorithmus abhängig, wodurch der Nichtdeterminismus deutlich abgeschwächt wird. Nur in dem Fall, in welchem mehrere Assoziationskanten das gleiche Gewicht haben, muss zufällig ausgewählt werden, welche Kante zu löschen ist.
Angewendet auf das Beispiel in Abbildung 24 (Seite 110) wird in einem ersten Schritt der Algorithmus von Tarjan ausgeführt, der den gleichen SCC wie im Beispiel von Le Traon identifiziert: **F**, **D**, **H**, **B**, **C**, **A** und **E**. Für alle Assoziationskanten innerhalb des SCC wird nun das Gewicht, wie zuvor beschrieben, berechnet. Dabei ergeben sich die folgenden Gewichte: **H→C** (9), **B→C** (3), **C→A** (3), **E→A** (3), **F→D** (2), **B→D** (2), **A→C** (9), **C→E** (6), **E→F** (4) und **C→H** (6) (vgl. [BLW03]). Es stehen somit die Kante **H→C** und die Kante **A→C** zum Löschen zur Verfügung. Es wird die Kante **A→C** zufällig ausgewählt, gelöscht und anschließend der Algorithmus von Tarjan erneut ausgeführt um verbleibende SCC zu identifizieren. Dieser identifiziert einen SCC an dem die Bausteine **H**, **B**, **C**, **D**, **F** und **E** beteiligt sind. In Anlehnung an die Beschreibung in [BLW03] werden in den nachfolgenden Schritten die Kanten **H→B**, **E→F** und **C→H** gelöscht und somit alle Zyklen aus dem Graph entfernt. Die dabei entstehende Integrationsreihenfolge integriert die Bausteine in der Reihung **AECHDFBG**, wobei die vier Bausteine **B**, **C**, **F** und **H** und die vier gelöschten Abhängigkeiten **A→C**, **E→F**, **C→H** sowie **H→B** simuliert werden müssen.

6.2.2.4 Genetischer Algorithmus

Die Anwendung der genetischen Algorithmen zur Ermittlung der Integrationsreihenfolge wurde von Briand, Feng und Labiche in [BFL02] vorgeschlagen. Die grundlegende Idee dahinter ist, dass zufällige Reihenfolgen generiert werden, wobei neue Reihenfolgen durch Modifikation der älteren Reihenfolgen ermittelt werden. Anschließend wird überprüft, ob die zufälligen Reihenfolgen besser sind als die vorher ermittelten Reihenfolgen. Um festlegen zu können, ob eine Reihenfolge besser ist als eine andere, ist eine Kostenfunktion notwendig. Diese Kostenfunktion bestimmt einen Zahlenwert, genannt Fitness. Ist die Fitness einer Reihenfolge $\mathbf{IR_1}$ größer als die Fitness einer Reihenfolge $\mathbf{IR_2}$, so ist $\mathbf{IR_1}$ hinsichtlich der in der Kostenfunktion berücksichtigten Parameter schlechter als $\mathbf{IR_2}$. Allgemein ausgedrückt: Je höher die Fitness, desto schlechter die Reihenfolge.
Briand et al. schlagen in [BFL02] eine Kostenfunktion vor, die den Simulationsaufwand (Stubbing Complexity) für die zu simulierenden Abhängigkeiten berechnet. Ziel von Briand et al. ist es, eine Integrationsreihenfolge zu finden, die einen möglichst geringen Simulationsaufwand hat.
Für das Ermitteln der zufälligen Reihenfolgen verwenden Briand et al. die genetischen Algorithmen. Inspiriert werden genetische Algorithmen durch die Prinzipien der Natur [SS05]. Die Grundlegende Idee dahinter besteht darin, dass eine Menge von Chromosomen sich über einen Zeitraum verändern und dabei die besten Chromosomen überleben, während schwache Chromsomen aussterben. Ein Chromosom stellt dabei eine mögliche Lösung eines gegebenen Problems dar. Die Menge von Chromosomen zu einem bestimmten Zeitpunkt wird Population genannt. Genetische Algorithmen beginnen mit der Erzeugung einer zufälligen Startpopulation. Diese Startpopulation wird im nächsten Schritt verändert. Hierzu kommen die Operatoren *Mutation* und *Kreuzung* zum Einsatz. Bei der *Mutation* wird

ein Chromosom verändert, so dass ein neues Chromosom entsteht. Die Kreuzung nimmt zwei Chromosomen und *kreuzt* sie miteinander, d.h. bildet daraus zwei neue Chromosomen. Durch die beiden Operatoren werden neue Chromosomen erzeugt. Anschließend wird geprüft, welche Chromosomen in die nächste Population übernommen werden. Da die neue Population nicht größer sein darf als die alte Population, müssen die schlechten Chromosomen entfernt werden. Es „überleben" nur die besten Chromosomen. In mehreren Schritten werden immer wieder neue Populationen erzeugt, bis nach einer festgelegten Zeit die letzte Population die beste Lösung bzw. die besten Lösungen enthält.
Voraussetzung für die Anwendung von genetischen Algorithmen zur Problemlösung ist, dass sich die mögliche Lösung des Problems als endliche Zeichenkette fester Länge beschreiben lässt und dass es eine Kostenfunktion gibt, die das Vergleichen der Zeichenkette ermöglicht. Das Problem der Integrationsreihenfolgeermittlung erfüllt das erste Kriterium, da sich eine Integrationsreihenfolge als Folge von Bausteinen beschreiben lässt. Für die Kostenfunktion wird der Simulationsaufwand einer aufgebrochenen Abhängigkeit verwendet. Briand et al. verwenden dazu zwei Parameter: a*ttribute coupling* und *method coupling*. Für eine Abhängigkeit **A→B** wird die Anzahl der in **B** deklarierten Services ermittelt. Im Fall, dass **B** von anderen Bausteinen erbt, werden zusätzlich die deklarierten Services der vererbenden Bausteine mitgezählt. Die so ermittelte Anzahl repräsentiert das *method coupling*. Ebenso wird mit der Anzahl der deklarierten Attribute in **B** bzw. seiner vererbenden Bausteine verfahren. Diese Anzahl wird als *attribute coupling* bezeichnet. Da nach Aussage von Briand et al. die Anzahl der deklarierten Attribute und Methoden sich stark voneinander unterscheiden, müssen die Werte normalisiert werden. Hierzu werden für alle Abhängigkeiten die maximale Anzahl deklarierte Services $\mathbf{M_{Services}}$ und die maximale Anzahl der deklarierten Attribute $\mathbf{M_{Attributes}}$ bestimmt. Anschließend wird für jede Abhängigkeit **A→B** der normalisierte Wert für die Anzahl der deklarierten Services $\mathbf{S_{norm}(A,B)}$ und der normalisierte Wert für die Anzahl der deklarierten Attribute $\mathbf{A_{norm}(A,B)}$ ermittelt. $\mathbf{S_{norm}(A,B)}$ berechnet sich aus der Anzahl deklarierter Services in **B** geteilt durch $\mathbf{M_{Services}}$. Äquivalent wird $\mathbf{A_{norm}}$ ermittelt. Durch die Normalisierung liegt der Wertebereich von $\mathbf{A_{norm}}$ und $\mathbf{S_{norm}}$ zwischen 0 und 1. Mit Hilfe von $\mathbf{S_{norm}}$ und $\mathbf{A_{norm}}$ berechnen Briand et al. den Simulationsaufwand für einer Abhängigkeit als gewichtetes geometrisches Mittel. Die Gewichtung erlaubt es, die später ermittelte Fitness an den geschätzten Simulationsaufwand für Services und Attribute anzupassen, d.h. durch die Gewichtung kann bestimmt werden, dass das Simulieren eines Serviceaufrufs aufwändiger ist, als das Simulieren eines Attributzugriffs bzw. umgekehrt. Um den Simulationsaufwand einer Integrationsreihenfolge zu ermitteln wird der Simulationsaufwand aller aufgebrochenen Abhängigkeiten aufsummiert.
Die *Mutation* und die *Kreuzung* von Chromsomen werden im genetischen Algorithmus zur Veränderung der existierenden Integrationsreihenfolgen wie folgt definiert.
Eine *Mutation* eines Chromosoms ist das zufällige Vertauschen zweier Bausteine innerhalb eines Chromosoms. Die Reihenfolge **ABCDEFG** könnte durch das Vertauschen des ersten und letzten Bausteines in die Reihenfolge **GBCDEFA** mutiert werden.
Eine *Kreuzung* zwischen zwei Chromosomen ist der Austausch zweier zufälliger Bausteine in beiden Chromosomen, so dass zwei neue Chromosomen entstehen. Hierzu wird zufällig eine Position bestimmt. Im ersten Chromosom wird bestimmt, welcher Baustein sich an dieser Position befindet. Für diesen Baustein wird im zweiten Chromosom die Position bestimmt und der Baustein, der sich an dieser Position befindet. Diese beiden Bausteine werden anschließend im ersten und im zweiten Chromosom ersetzt. Beispielsweise könnte die Kreuzung der zwei Chromosomen **ABCDEFG** und **GFACDEB** bei einer gewählten zufälligen Position von 3 die Chromosomen **CBADEFGH** und **GFCADEB** zur Folge haben.
Für die Anwendung der genetischen Algorithmen für die Integrationsreihenfolgeermittlung haben Briand et al. folgende Parameter identifiziert, die auf den Algorithmus Einfluss haben:

Mutationsrate, *Kreuzungsrate*, *Populationsgröße*, *Anzahl Iterationen* und *Anzahl überlebender Chromosomen*.
Die *Mutationsrate* gibt an, wie viel Prozent der Chromosomen in der aktuellen Population mutiert werden. Die *Kreuzungsrate* gibt an, wie viel Prozent der Chromosomen der aktuellen Population zur Kreuzung herangezogen werden. Die Populationsgröße beschreibt, wie viele Chromosomen sich in einer Population befinden können. Die *Anzahl der Iteration* beschreibt, wie oft die Populationen durch Mutation und Kreuzung verändert wird und gibt somit das Abbruchkriterium für den Algorithmus an. Die *Anzahl überlebender Chromosomen* beschreibt, wie viele Prozent der besten Chromosomen der aktuellen Population automatisch in die neue Population übernommen werden. Briand et al. begründen die Übernahme der besten Chromosomen mit der Forderung, dass die besten Lösungen nicht verloren gehen sollen.
Für die Anwendung des genetischen Algorithmus zur Reihenfolgeermittlung auf die neun Beispielsysteme wird in der nachfolgenden Untersuchungen als Mutationsrate der Wert 25%, als Kreuzungsrate 50%, als Populationsgröße die Anzahl der Bausteine des Softwaresystems und als Anzahl Iterationsschritte der von Briand et al. vorgeschlagene Wert von 500 verwendet. Als Anzahl überlebender Chromosomen wird der Wert 0.5 gewählt, d.h. 50% der besten Chromosomen aus der früheren Population überleben in der neuen Population. Darüber hinaus muss in der Untersuchung eine Anpassung der Kostenfunktion vorgenommen werden, da die von Briand et al. vorgeschlagene Kostenfunktion das Aufbrechen von Vererbungsbeziehungen nicht berücksichtigt. Sie berücksichtigen die Vererbungen durch eine Prioritätstabelle, die sie dem verwendeten Werkzeug mit übergeben können. Durch diese Prioritätstabelle werden Integrationsreihenfolgen, die durch Kreuzung und Mutation entstanden sind und Vererbungsbeziehungen aufbrechen würden, nicht akzeptiert.
In der Untersuchung steht das von Briand et al. verwendete Werkzeug nicht zur Verfügung. Aus diesem Grund wird für die Untersuchung die Kostenfunktion um den zusätzlichen Parameter *Vererbung* erweitert. Dieser wird auf 1 gesetzt, wenn der Simulationsaufwand für eine Vererbungsbeziehung betrachtet wird, und auf 0, wenn es keine Vererbungsbeziehung ist. Darüber hinaus wird ein drittes Gewicht hinzugefügt, das den Simulationsaufwand von Vererbungsbeziehungen entsprechend anpassen kann. Im Gegensatz zu Briand et al. wird in der Untersuchung nicht das gewichtete geometrische, sondern das gewichtete arithmetische Mittel verwendet, um den Simulationsaufwand einer Abhängigkeit **A→B** zu berechnen (vgl. Formel 1). Die gewählten Gewichtungen sollten in der Summe 1 ergeben (vgl. [BFL02]). Mit Hilfe der Gewichtungen W_A, W_S und W_V kann der Anwender von außen definieren, welcher Simulationsaspekt wie schwierig ist. So kann er angeben, ob das Simulieren einer Vererbungsbeziehung (z.B. W_V=0.9) viel schwerer als das Simulieren von Serviceaufrufen und Attributzugriffen (W_S=W_A=0.05) ist und ob es genau so schwer ist, Serviceaufrufe zu simulieren (W_S=0.05) wie das Simulieren von Attributzugriffen (W_A=0.05). Die in den Klammern beispielhaft genannten Gewichtungen sind die Gewichtungen, die im Rahmen der Anwendung auf die neun Softwaresysteme verwendet wurden. Die Begründung dafür ist, dass deutlich mehr Assoziationsbeziehungen als Vererbungsbeziehungen aufgebrochen werden sollen, da Vererbungsbeziehungen schwerer zu simulieren sind (vgl. [BLW03]). Aus diesem Grund wird das Gewicht W_V auf einen Wert von 0.9 gesetzt und die Gewichte von W_S und W_A auf die sehr kleinen Werte von 0.05. Dadurch erhöht sich bei einer aufgebrochenen Vererbungsbeziehung der Wert, den die Kostenfunktion ermittelt, deutlich stärker als bei einer aufgebrochenen Assoziationsbeziehung. Die beiden Werte W_S und W_A wurden in Anlehnung an Untersuchungen in [BFL02] gleich groß gewählt.
Die Gesamtfitness bzw. der Gesamtsimulationsaufwand einer Reihenfolge wird als Summe der Simulationsaufwände aller aufgebrochenen Abhängigkeiten ermittelt [BFL02]. Dieser Gesamtsimulationsaufwand wird für den Vergleich zweier Integrationsreihenfolgen

herangezogen. Hierbei gilt, dass eine Integrationsreihenfolge IR_1 besser als eine Integrationsreihenfolge IR_2 hinsichtlich des Simulationsaufwandes ist, wenn der berechnete Zahlenwert für den Simulationsaufwand für IR_1 kleiner als für IR_2 ist.

Formel 1: Berechnung des Simulationsaufwandes einer Abhängigkeit

$$SA(AB) = (W_A * A_{Norm}(AB) + W_S * S_{Norm}(AB) + W_V * V(AB)) * 1/3$$

SA(AB) … Simulationsaufwand für die Abhängigkeit A→B
W_A ... Gewichtung für Attribute
W_S … Gewichtung für Services
W_V … Gewichtung für Vererbungen
$A_{Norm}(AB)$ … Normierte Anzahl zugegriffener Attribute in der Abhängigkeit A→B
$S_{Norm}(AB)$ … Normierte Anzahl aufgerufener Services in der Abhängigkeit A→B
V(AB) … hat den Wert 1, wenn A→B eine Vererbungsbeziehung, sonst den Wert 0

6.2.2.5 Algorithmus basierend auf Zufall

Als Vergleich zu den existierenden Algorithmen wird ein zusätzlicher Algorithmus verwendet, der die Integrationsreihenfolgen zufällig ermittelt. Die ermittelten Integrationsreihenfolgen werden anhand der Kostenfunktion des genetischen Algorithmus miteinander verglichen und nach einer vordefinierten Anzahl an Versuchen wird das beste Ergebnis ausgegeben. Dieser Algorithmus kann durch den Wert *Anzahl der Versuche ohne Erfolg* parametrisiert werden. Diese Anzahl gibt an, wie oft der Algorithmus versucht, zufällig neue Reihenfolgen zu ermitteln, ohne dabei eine neue bessere Reihenfolge gefunden zu haben. In der späteren Untersuchung wird dieser Wert auf die Anzahl der Bausteine innerhalb des zu untersuchenden Softwaresystems gesetzt, d.h. wenn das Softwaresystem aus 110 Bausteinen besteht, unternimmt der Algorithmus mindestens 111 Versuche eine Integrationsreihenfolge zu ermitteln. Sollte zwischendurch eine Reihenfolge besser sein als die aktuell beste Reihenfolge, so wird diese zwischengespeichert und der Zähler wird auf 0 zurückgesetzt, so dass der Algorithmus erneut 110 Versuche hat, um eine neue Reihenfolge zu ermitteln. Dieser Algorithmus arbeitet so lange, bis er nach der vorgegebenen Anzahl an Versuchen keine neue bessere Reihenfolge gefunden hat.

6.2.2.6 Simulated Annealing

Ein Algorithmus, der in der Optimierung Anwendung findet, ist der Algorithmus des „Simulated Annealing“ (simuliertes Abkühlen). Burkard und Rendl beschreiben in [BR84] die Anwendung dieses Algorithmus zur Lösung von Optimierungsproblemen. Das Simulated Annealing ist inspiriert durch physikalische Vorgänge. Es beschreibt das langsame Abkühlen einer Kristallstruktur und der darin enthaltenen Atome. Interessant für die Physiker ist die in dem Kristallgitter enthaltene Energie bei einer bestimmten Temperatur. Ziel ist es, dass ein Gleichgewicht der Energie in jedem einzelnen Atom erreicht wird. Je höher die Temperatur des Kristallgitters ist, desto mehr Energie ist in den Atomen und desto leichter ist es für die Atome eine Position, die sie eingenommen haben, zu verlassen. Durch das Verändern der Position der einzelnen Atome kann sich die im Kristallgitter enthaltene Energie verändern. Bei einer hohen Temperatur haben Atome die Möglichkeit, aus dem Gitter auszubrechen und an einer anderen Stelle des Gitters eine neue Position zu finden. Je höher die Temperatur ist, desto höher ist auch die Wahrscheinlichkeit, dass sich das Kristallgitter durch das Lösen von Atomen aus dem Gitter verändert. Je weiter die Temperatur sinkt, desto kleiner wird die Wahrscheinlichkeit, dass sich Atome von einer bereits eingenommenen Position entfernen. Bei genügend geringer Temperatur erreicht das Kristallgitter einen Zustand, in dem die

Atome ihre Position nicht mehr verlassen können. Dabei „strebt“ das Gitter einen Zustand an, in dem seine Gesamtenergie möglichst minimal ist. Ist dieser Zustand erreicht, ist das Gitter „eingefroren“.
Dieser Vorgang kann mit Hilfe des Simulated Annealing Algorithmus simuliert werden. Ausgangspunkt ist ein zufällig erzeugtes Startgitter und eine Starttemperatur. Aus dem Startgitter wird durch einfache Modifikationen ein neues Kristallgitter erzeugt. Für das neue Kristallgitter wird die Energie berechnet und mit der Energie des Startgitters verglichen. Ist die Energie des neuen Gitters kleiner als die Energie des Startgitters, so wird das neue Gitter als Gitter für weitere Modifikationen verwendet. Ist die Energie des neuen Gitters hingegen größer als die Energie des Startgitters, wird das neue Gitter nur mit einer gewissen Wahrscheinlichkeit für weitere Modifikationen verwendet. Dabei hängt die Wahrscheinlichkeit, dass das neue, energetisch schlechtere Gitter verwendet wird, von der aktuellen Temperatur ab. Für eine festgelegte Anzahl **N** von Schleifendurchläufen werden neue Gitter berechnet ausgehend von der jeweils zuletzt akzeptierten Lösung. Nach Ablauf der **N** Schleifendurchläufe wird die Temperatur gesenkt. Anschließend werden in den **N** Schleifendurchläufe neue Gitter erzeugt. Durch das Senken der Temperatur reduziert sich die Wahrscheinlichkeit, dass eine schlechtere Lösung akzeptiert wird. Das Senken der Temperatur und das damit verbundene Ausführen der **N** Schleifendurchläufe, bestimmen den Endpunkt des Algorithmus. Wenn eine bestimmte Temperatur erreicht ist, kann das Gitter als „eingefroren“ betrachtet werden. Das Akzeptieren schlechterer Lösungen ermöglicht es, ein erreichtes lokales Minimum wieder zu verlassen, um das globale Minimum zu finden.
Übertragen auf die Integrationsreihenfolgeermittlung stellt eine Integrationsreihenfolge das Gitter dar. Für die Ermittlung der Energie der Reihenfolge kann die angepasste Kostenfunktion von Briand et al. in [BFL02] aus Kapitel 6.2.2.4 verwendet werden. Der Pseudocode des Simulated Annealing für die Integrationsreihenfolge ist in der Formel 2 dargestellt. Wie daraus entnommen werden kann, muss die Startemperatur $\mathbf{T_S}$, die Endtemperatur $\mathbf{T_E}$, die Temperatursenkungsfunktion **senke**, die Anzahl der Versuche mit gleicher Temperatur **N** und eine Modifikationsfunktion **modifiziere** bestimmt werden. Zur Bestimmung der entsprechenden Parameter und Funktionen wurden im Rahmen dieser Arbeit verschiedene Versuche durchgeführt, um eine möglichst optimale Integrationsreihenfolge bestimmen zu können. Unterschiedliche Kombinationen wurden hierzu untersucht.

Modifikationsfunktion: Als Modifikationsfunktion wurden Funktionen untersucht, die aus einer gegebenen Reihenfolge durch eine möglichst kleine Änderung eine neue Reihenfolge erzeugen, um die Energiedifferenzen nicht zu groß werden zu lassen. Hierbei boten sich zwei Möglichkeiten an: *Nachbarschaftstausch* und *Mutation*.
Der *Nachbarschaftstausch* bestimmt eine Position *X* innerhalb der Reihenfolge zufällig und tauscht den Baustein an Position *X* mit dem Baustein an Position *X+1* aus. Die neu ermittelte Reihenfolge ist somit sehr ähnlich zur Ausgangsreihenfolge.
Die *Mutation* wählte zufällig zwei Positionen innerhalb der Reihenfolge aus und tauscht die Bausteine der entsprechenden Positionen miteinander.
Die Verwendung der Mutation wird durch die genetischen Algorithmen angeregt. In den Untersuchungen zeigte sich, dass die Mutation deutlich bessere Ergebnisse als der Nachbarschaftstausch erzeugt.

Temperatursenkungsfunktion: Die Temperatursenkungsfunktion beschreibt, wie stark sich die Temperatur von einem Schritt zum nächsten ändern kann. Für diese Funktion wurden zwei verschiedene Funktionen untersucht.

Die erste Funktion stellt eine einfache lineare Gleichung in der Form $\mathbf{T_n = T_a - a}$, wobei *a* einen positiven Wert darstellte. Die Funktion beschreibt eine lineare Senkung der aktuellen Temperatur um den Wert *a*. Die neue Temperatur T_n errechnet sich aus der alten Temperatur T_a minus *a*. Die Temperatur senkt sich in jedem Schritt im gleichen Maße.
Die zweite verwendete Funktion ist eine Potenzfunktion der Form $\mathbf{T_n = T_a - (T_a * a)}$. Die Funktion beschreibt eine prozentuale Senkung der aktuellen Temperatur um den Faktor *a*. Die Anwendung der ersten Funktion führte dazu, dass am Anfang sehr viele falsche Lösungen akzeptiert wurden. Gegen Ende des Algorithmus hingegen wurden zu wenige Versuche durchgeführt, um eine bereits gute Lösung noch zu verbessern.
Die zweite Formel hat den Vorteil, dass die Temperatur am Anfang sehr schnell fällt und gegen Ende mehr Versuche darauf verwendet werden, eine bereits gefundene gute Lösung noch zu verbessern. Aus diesem Grund wird in den Untersuchungen die zweite Funktion verwendet. Für die Werte von *a* wurden unterschiedliche Werte ausprobiert, wobei sich zeigt, dass die Laufzeit des Algorithmus von *a* abhängt, da ein kleineres a eine höhere Anzahl von späteren Schleifendurchläufen nach sich zieht. In den Untersuchungen hat sich ein *a* von 0,002 als sehr gut erwiesen.

Formel 2: Pseudocode des Simulated Annealing Algorithmus

```
Startemperatur T_S
Endtemperatur T_E
Anzahl Versuche pro gleicher Temperatur N
Aktuelle Temperatur T = T_S
Erzeuge aktuelle Reihenfolge IR_akt zufällig
Beste Reihenfolge IR_Best = IR_akt
Zufallszahl zf
Wahrscheinlichkeit w
Solange T_E < T
     Wiederhole N mal
         Erzeuge neue Reihenfolge: IR_neu = modifiziere(IR_akt)
         Wenn Energie(IR_neu) < Energie (IR_Best)
               IR_Best = IR_neu
               IR_akt = IR_neu
         Sonst
               Wenn Energie (IR_neu) < Energie (IR_akt)
                    IR_akt = IR_neu
               Sonst
                    zf = neue Zufallszahl zwischen 0 und 1
                    w = exp( (Energie(IR_neu) – Energie(IR_akt)) / T)
                    Wenn w < zf
                         IR_akt = IR_neu
     Senke Temperatur: T = senke(T)
```

Starttemperatur: Die Wahl der Starttemperatur hat Einfluss darauf, wie viele falsche Lösungen am Anfang akzeptiert werden. Je größer diese Temperatur gewählt wird, desto mehr falsche Lösungen werden akzeptiert. In einem ersten Schritt wurde für unterschiedliche Beispielsysteme stichprobenartig die Differenz zwischen zwei benachbarten Integrationsreihenfolgen ermittelt. Dabei konnten sehr kleine Differenzen (< 0,0001) und sehr große Differenzen (>20) gefunden werden. Die großen Differenzen waren jedoch sehr selten. Aufbauend auf diesen Ergebnissen wurde mit den Startemperaturen 50; 25; 10; 5; 1 und 0,1 experimentiert. Dabei zeigte sich, dass eine kleinere Starttemperatur (um 1) die besten Ergebnisse geliefert hat. Am Anfang werden ausreichend viele schlechtere Lösungen (ca.

20%) akzeptiert, da die Wahrscheinlichkeit, schlechtere Lösungen zu akzeptieren von der aktuellen Temperatur abhängt. Aber die Starttemperatur darf nicht zu klein gewählt werden, da sonst die Wahrscheinlichkeit, eine schlechte Lösung zu akzeptieren zu gering ist und die Gefahr besteht, dass der Algorithmus ein lokales Optimum nicht mehr verlassen kann.
Für die späteren Untersuchungen wurde daher die Startemperatur 1 verwendet.

Endtemperatur: Die Wahl der Endtemperatur beeinflusst die Anzahl der Schleifendurchläufe **N** und somit die Laufzeit des Algorithmus. Je kleiner die Endtemperatur gewählt wird, desto länger wird versucht aus einer bereits guten Lösung noch bessere Lösungen zu erzeugen. Dabei zeigt sich, dass eine sehr kleine Endtemperatur nur sehr selten zu besseren Ergebnissen führt. In den Voruntersuchungen wurden verschiedene Endtemperaturen evaluiert und eine Endtemperatur von 0,001 hat sich als ausreichend herausgestellt. Bei kleineren Werten zeigte sich, dass zwischen einer gewählten Endtemperatur von 0,001 und 0,0001 keine nennenswerten Verbesserungen der Integrationsreihenfolge erreicht werden konnten.

Anzahl Versuche mit gleicher Temperatur: Diese Zahl beschreibt, wie oft versucht wird, neue Lösungen zu erzeugen, bevor die Temperatur gesenkt wird. Die gewählte Anzahl hat sehr starken Einfluss auf die Laufzeit des Algorithmus. Sie sollte so gewählt werden, dass die Anzahl der Versuche groß genug ist, um mit einer hohen Wahrscheinlichkeit bessere Lösungen zu finden. Im Rahmen der Arbeit wurden verschiedene Zahlen ausprobiert, sowohl feste als auch an das System angepasste Zahlen. Die festen Zahlen hatten immer den Nachteil, dass sie für eine bestimmte Anzahl von Bausteinen des Softwaresystems optimal waren, sich aber nicht auf andere Softwaresysteme anwenden ließen. Daher wird die Anzahl Versuche mit gleicher Temperatur an die Anzahl Bausteine des zu untersuchenden Softwaresystems angepasst, um die Größe des Softwaresystems im Algorithmus zu berücksichtigen. Vor jeder Temperatursenkung werden daher *X* Versuche unternommen um neue Reihenfolgen zu finden, wobei *X* die Anzahl der Bausteine innerhalb des Softwaresystems ist.

6.2.2.7 Ideale Testfokusberücksichtigung

Die Berücksichtigung des Testfokus in einer Integrationsreihenfolge erfordert einen neuen Algorithmus, da bisherige Algorithmen sich nur auf die Optimierung des Simulationsaufwands konzentrieren. Dieser verwendet eine einfache Sortierfunktion und ermittelt basierend auf diesen Ergebnissen die Integrationsreihenfolge. Darüber hinaus kann diese Sortierfunktion mit einem existierenden Algorithmus verbunden werden, um den Simulationsaufwand zusätzlich zu berücksichtigen.
Der Sortieralgorithmus identifiziert in einem ersten Schritt alle Bausteine, die an Abhängigkeiten mit Testfokus beteiligt sind, und fügt sie in die Menge der Bausteine mit Testpriorität $\mathbf{B_{TP}}$ ein. Die verbleibenden Bausteine werden der Menge der Bausteine ohne Testpriorität zugeordnet $\mathbf{B_{OP}}$. Um eine Integrationsreihenfolge zu ermitteln, die die Testpriorität zu 100% berücksichtigt, müssen die Bausteine in $\mathbf{B_{TP}}$ vollständig vor den Bausteinen in $\mathbf{B_{OP}}$ integriert werden. Diese Zweiteilung führt dazu, dass eine Abhängigkeit **A→B** simuliert werden muss, wenn sich der abhängige Baustein in $\mathbf{B_{TP}}$ und der unabhängige Baustein in $\mathbf{B_{OP}}$ befindet.
In welcher Reihenfolge die Bausteine in $\mathbf{B_{TP}}$ bzw. in $\mathbf{B_{OP}}$ jedoch integriert werden, ist für die Testfokusberücksichtigung nicht von Belang. An dieser Stelle können Optimierungen hinsichtlich des Simulationsaufwandes mit einfließen. Hierfür wird der Algorithmus von Briand et al. (vgl. Kapitel 6.2.2.3) angewendet, um für die Menge $\mathbf{B_{TP}}$ und die Menge $\mathbf{B_{OP}}$ unabhängig voneinander die Integrationsreihenfolge zu ermitteln. Die Wahl fiel auf diesen Algorithmus, da er sich sowohl in den Untersuchungen dieser Arbeit als auch in [BLW03] als

der beste Algorithmen zur Ermittlung einer Integrationsreihenfolge, die den Simulationsaufwand am besten berücksichtigt, herausgestellt hat.
Die Integrationsreihenfolge $\mathbf{IR_{TP}}$ der Menge $\mathbf{B_{TP}}$ wird mit Hilfe des Algorithmus von Briand et al. ermittelt und stellt den ersten Teil der gesamten Integrationsreihenfolge dar. Anschießend wird die Integrationsreihenfolge $\mathbf{IR_{OP}}$ für die Menge $\mathbf{B_{OP}}$ mit Hilfe des Algorithmus von Briand et al. ermittelt. Sie stellt den zweiten Teil der gesamten Integrationsreihenfolge dar. Die gesamte Integrationsreihenfolge ist die Konkatenation von $\mathbf{IR_{TP}}$ und $\mathbf{IR_{OP}}$. Die Reihenfolge berücksichtigt zu 100% den Testfokus und berücksichtigt darüber hinaus den Simulationsaufwand innerhalb der Mengen $\mathbf{IR_{TP}}$ und $\mathbf{IR_{OP}}$. Es muss jedoch angemerkt werden, dass alle Abhängigkeiten, deren abhängiger Baustein sich in $\mathbf{IR_{TP}}$ und der unabhängige Baustein in $\mathbf{IR_{OP}}$ befindet, trotz der Optimierung von Briand et al. aufgebrochen werden.

6.2.2.8 Vergleich aller Algorithmen

Die sieben Algorithmen, die in den Kapiteln 6.2.2.1 bis 6.2.2.7 vorgestellt wurden, werden auf die in Kapitel 6.2.2 vorgestellten neun Beispielsysteme angewendet. Um die berechneten Integrationsreihenfolgen und somit die Algorithmen vergleichen zu können, werden verschiedene Maßzahlen herangezogen. Diese Maßzahlen messen verschieden Aspekte des *Simulationsaufwands*, der *Testfokusberücksichtigung* und die *Berechnungsdauer*. Sie werden nachfolgend kurz erläutert.

Simulationsaufwand:

- **Anzahl der zu simulierenden Dateien**
  Die Anzahl der zu simulierenden Dateien zählt die Anzahl der Dateien, die als Stub simuliert werden muss. Dabei wird jede Datei nur einmal gezählt, auch wenn sie in mehreren Integrationsschritten benötigt wird. Diese Maßzahl ist vergleichbar mit der Anzahl realistischer Stubs, wie sie in [BLW03] verwendet wird.

- **Anzahl zu simulierender Abhängigkeiten**
  Die Anzahl zu simulierender Abhängigkeiten beschreibt, wie viele Abhängigkeiten aufgrund der gewählten Integrationsreihenfolge aufgebrochen werden. Jede Abhängigkeit wird dabei nur einmal gezählt, auch wenn sie in mehreren Integrationsschritten simuliert werden muss. Diese Maßzahl ist mit der Anzahl spezifischer Stubs, wie sie in [BLW03] vorgeschlagen wird, gleichzusetzen.

- **Anzahl aufgebrochener Vererbungen**
  Die Maßzahl beschreibt die Anzahl von aufgebrochenen Vererbungsbeziehungen. Sie ist kleiner gleich der Anzahl der zu simulierenden Abhängigkeiten, da Vererbungsbeziehungen eine Teilmenge der Abhängigkeiten sind.

- **Anzahl zu simulierender Attributzugriffe**
  Ähnlich zur Anzahl der zu simulierenden Serviceaufrufe wird die Anzahl der zu simulierenden Attributzugriffe ermittelt. Es werden alle Attributzugriffe zwischen der abhängigen Datei **A** und der unabhängigen Datei **B** gezählt, wenn die Abhängigkeit **A→B** simuliert wird. Diese Maßzahl stellt eine Erweiterung der in [BLW03] vorgeschlagenen Maßzahl „attribute coupling“ dar.

- **Anzahl zu simulierender Serviceaufrufe**
  Die Anzahl der zu simulierenden Serviceaufrufe beschreibt die Anzahl der Services, die im unabhängigen Baustein simuliert werden müssen, wenn die entsprechende Abhängigkeit aufgebrochen wird. Ruft eine Datei **A** beispielsweise 10 unterschiedliche Services an der unabhängigen Datei **B** auf und wird die Abhängigkeit **A→B** simuliert, so müssen 10 Serviceaufrufe simuliert werden. Diese Maßzahl stellt eine Verfeinerung der von Briand et al. in [BLW03] vorgeschlagenen Maßzahl der Methodenkopplung („method coupling“) dar. Während Briand et al. alle innerhalb der unabhängigen Datei deklarierten Methoden berücksichtigen, erlaubt die von hier verwendete Maßzahl eine genauerer Aussage über die Anzahl der zu simulierenden Services.

Testfokus:

- **Anzahl richtig integrierter Abhängigkeiten**
  Die Anzahl richtig integrierter Abhängigkeiten zählt die Abhängigkeiten, deren Bausteine frühzeitig integriert wurden.

- **Anzahl zu spät integrierter Abhängigkeiten**
  Die Anzahl zu spät integrierter Abhängigkeiten zählt alle Abhängigkeiten, von denen mindestens ein beteiligter Baustein zu spät integriert wurde.

Berechnungsdauer:
Die Berechnungsdauer gibt die zur Berechnung der Integrationsreihenfolge durch den eingesetzten Computer benötigte Zeitspanne in Millisekunden an.

Diese Maßzahlen erlauben es Aussagen zu machen, wie gut die einzelnen Ansätze den Testfokus bzw. den Simulationsaufwand berücksichtigen. Darüber hinaus gibt die Berechnungsdauer eine ungefähre Einschätzung des Laufzeitverhaltens bei steigender Anzahl zu integrierender Bausteine[1].
Zur Durchführung der Untersuchung werden alle Algorithmen in Java realisiert und die entsprechenden Maßzahlen automatisch erhoben (vgl. Kapitel 7.2). Als Eingaben erhalten die realisierten Algorithmen eine Tabelle, die in jeder Zeile eine Abhängigkeit und ihre Eigenschaften spezifiziert. Diese Eigenschaften umfassen die *Anzahl der aufgerufenen Services*, die *Anzahl der zugegriffenen Attribute*, die Information, ob es sich um eine *Vererbungsbeziehung* handelt, die vollständigen *Namen der beteiligten Dateien* sowie der zugeordnete *Testfokus*. Die Tabelle ist vergleichbar mit der Eingangstabelle für die Korrelationsanalysen zur Testfokusauswahl (vgl. Tabelle 7).
Die erhoben Maßzahlen pro Anwendung der Algorithmen sind im Anhang C: *Ergebnisse der Algorithmenanwendung* in Tabelle 21 bis Tabelle 27 zu finden. Sie geben für jeden Algorithmus die entsprechenden Maßzahlen für die einzelnen Softwaresysteme wider. Um die Ergebnisse der einzelnen Algorithmen miteinander vergleichen zu können, wird den Algorithmen für jede untersuchte Maßzahl pro untersuchtes Softwaresystem eine Platzierung zugeordnet. Diese Platzierung beschreibt wie gut der Algorithmus im Vergleich zu den anderen Algorithmen in Bezug auf die Maßzahl abgeschnitten hat. Aus dieser Platzierung wird ein Mittelwert über alle Softwaresysteme gebildet um zu beschreiben, wie gut der Algorithmus im Durchschnitt bei der betrachtete Maßzahl im Vergleich zu den anderen

[1] Es ist zu berücksichtigen, dass die angegebene Berechnungsdauer nur einen ungefähren Richtwert angibt, da für die Ausführung der Algorithmen die eingesetzte Hardware von mehreren Benutzern verwendet werden konnte. Dadurch stand nicht immer die volle Rechenkapazität für die Ermittlung der Integrationsreihenfolge zur Verfügung.

Algorithmen abschneidet. Um das Konzept zu veranschaulichen, wird die Anzahl der zu simulierenden Bausteine verwendet (vgl. Spalte 3 in Tabelle 21 bis Tabelle 27 im Anhang C). Die Reihenfolge, berechnet mit dem Algorithmus von Briand et al. (vgl. Kapitel 6.2.2.3), liefert für 5 Softwaresysteme die kleinste Anzahl zu simulierender Bausteine (Platzierung 1) und für 4 Softwaresysteme die zweitgeringste Anzahl (Platzierung 2) an simulierenden Bausteinen. Daraus lässt sich eine durchschnittliche Platzierung von 1,44 ableiten. Eine solche durchschnittliche Platzierung wird für alle Algorithmen erstellt. Die durchschnittlichen Platzierungen werden für den Vergleich der Algorithmen verwendet und sind in Tabelle 19 zusammenfassend dargestellt. Jede Zeile stellt die durchschnittlichen Platzierungen eines Algorithmus dar. Die Spalten enthalten die Platzierungen nach den erhobenen Maßzahlen. Für jede Maßzahl ist die jeweils beste durchschnittliche Platzierung hervorgehoben. Die einzelnen Maßzahlen werden nachfolgend im Detail diskutiert.

Tabelle 19: Durchschnittliche Platzierung der Algorithmen

	Dauer	**Simulierte Dateien**	**Simulierte Abhängigkeiten**	**Simulierte Serviceaufrufe**	**Simulierte Attributzugriffe**	**Aufgebrochene Vererbungen**	**Richtig integrierte Abhängigkeiten**	**Zu spät integrierte Abhängigkeiten**
Ideale Testfokus-berücksichtigung	2,8	5,2	5,0	4,9	5,2	5,9	**1,0**	**1,0**
Tai und Daniels	4,3	4,6	5,4	5,3	4,2	**1,0**	5,4	5,4
Simulated Annealing	7,0	2,8	2,6	2,0	2,4	1,8	4,1	4,1
Genetischer Algorithmus	6,0	4,9	4,1	3,7	3,6	2,1	4,4	4,4
Briand et al.	3,1	**1,4**	**1,0**	**1,6**	**1,4**	**1,0**	6,4	6,4
Le Traon et al.	**1,1**	1,9	2,9	3,6	4,0	4,4	2,3	2,3
Zufallsbasierter Algorithmus	3,7	7,0	7,0	7,0	7,0	7,0	3,8	3,8

Richtig und zu spät integrierte Abhängigkeiten (nach Testfokus)

Bei der Untersuchung zeigt sich, dass keiner der existierenden Ansätze explizit oder implizit den Testfokus in der Integrationsreihenfolgeermittlung berücksichtigt. In den Tabellen im Anhang C ist dies zu erkennen. Der Algorithmus von Tai und Daniels integriert die Abhängigkeiten, die als Testfokus ausgewählt wurden, deutlich zu spät. Im Durchschnitt werden nur ca. 13% dieser Abhängigkeiten zum richtigen Zeitpunkt integriert. Mehr als 87% der Abhängigkeiten werden durch die ermittelte Integrationsreihenfolge zu spät geprüft. Der Algorithmus von Le Traon et al. integriert im Durchschnitt 34% der Testfokus-Abhängigkeiten zum richtigen Zeitpunkt. Der graphbasierte Ansatz von Briand et al. schneidet als schlechtester Algorithmus ab. Durch ihn werden nur ca. 10% der Testfokus-Abhängigkeiten frühzeitig integriert, d.h. dass ca. 90% der Abhängigkeiten durch die von Briand et al. ermittelte Integrationsreihenfolge zu spät integriert werden.

Am Besten schneidet der Algorithmus zur idealen Testfokusberücksichtigung ab. Die Integrationsreihenfolge wird so ermittelt, dass alle Abhängigkeiten, die als Testfokus ausgewählt wurden, frühzeitig integriert werden.

Berechnungsdauer

In den Untersuchungen zeigt sich, dass der Algorithmus von Le Traon et al. in acht von neun Softwaresystemen der schnellste und im verbleibenden System der zweitschnellste war. Er ist somit im Vergleich mit den anderen Algorithmen der schnellste Algorithmus in der Untersuchung. Der Algorithmus zur idealen Testfokusberücksichtigung, der Algorithmus von Briand et al. und der zufallsbasierte Algorithmus sind im Durchschnitt gleich schnell. Am

längsten haben die Berechnungen des Simulated Annealing Algorithmus benötigt. In allen Softwaresystemen ist er der langsamste Algorithmus und benötigte im besten Fall (Freenet, 24730ms) „nur“ 325mal mehr Zeit als der Algorithmus von Le Traon et al. und im schlechtesten Fall (Jetspeed, 190434ms) ca. 890mal mehr Zeit. Die längste Berechnungszeit benötigte der Algorithmus bei der Berechnung der Integrationsreihenfolge für Eclipse. Dort rechnete er 21,5 Stunden. Der genetische Algorithmus ist in allen neun Softwaresystemen der zweitlangsamste. Aber er ist im Durchschnitt 3,5mal schneller als der Simulated Annealing Algorithmus.

Anzahl simulierter Dateien
Es zeigt sich, dass die Integrationsreihenfolgen der beiden graphenbasierten Algorithmen Briand et al. und Le Traon et al. die geringste Anzahl an zu simulierenden Dateien erfordern. Briand et al. erreicht in 5 von 9 Systemen die geringste Anzahl und in den verbleibenden 4 Systemen die zweitgeringste Anzahl zu simulierender Bausteine. Le Traon et al. erreicht 4mal den ersten Platz, 2mal den zweiten Platz und 3mal den dritten Platz und somit eine durchschnittliche Platzierung von 1,9. Diesen zwei Algorithmen folgt der Simulated Annealing Algorithmus mit einer durchschnittlichen Platzierung von 2,6. Für die kleineren Softwaresysteme erreicht er Plätze von 1 bis drei, schneidet aber beim großen Softwaresystem Eclipse mit dem 6. Platz sehr schlecht ab. Durchweg die schlechtesten Ergebnisse lieferte der zufallsbasierte Algorithmus. Der genetische Algorithmus, der Algorithmus zur idealen Testfokusberücksichtigung und der Algorithmus von Tai und Daniels eignen sich weniger, um die Anzahl simulierender Dateien zu minimieren.

Anzahl simulierter Abhängigkeiten
Hier zeigt sich, dass in allen Softwaresystemen der Algorithmus von Briand et al. die beste Platzierung erreicht, d.h. die Anzahl der zu simulierenden Abhängigkeiten ist für alle Softwaresysteme die kleinste. Auf Platz zwei folgt der Simulated Annealing Algorithmus mit einer durchschnittlichen Platzierung von 2,6 knapp gefolgt vom Algorithmus von Le Traon mit der durchschnittlichen Platzierung 2,9. Wie auch bei der Anzahl simulierter Dateien zeigt der Simulated Annealing Algorithmus seine Schwäche bei der Berücksichtigung der Anzahl simulierter Abhängigkeiten in sehr großen Softwaresystemen. Für Eclipse erreichte er nur den sechsten Platz.

Anzahl simulierter Serviceaufrufe
Der Algorithmus von Briand et al. schneidet in dieser Kategorie ebenfalls am besten mit einer durchschnittlichen Platzierung von 1,6 ab. Ihm folgt der Simulated Annealing Algorithmus mit der durchschnittlichen Platzierung 2,0. Wie in den vorangegangenen zwei Maßzahlen zeigt der Simulated Annealing Algorithmus Schwäche beim Softwaresystem Eclipse für das er nur den 6. Platz erreicht. Der genetische Algorithmus und der Algorithmus von Le Traon liefern bei allen Softwaresystemen in etwa die gleichen konstanten Ergebnisse und erreichen somit Platz 3,6 bzw. 3,7. Die Algorithmen von Tai und Daniels sowie der Algorithmus zur idealen Testfokusberücksichtigung eignen sich hingegen weniger, um die Anzahl der zu simulierenden Serviceaufrufe zu minimieren. Der zufallsbasierte Algorithmus war noch schlechter.

Anzahl simulierter Attributzugriffe
Der Algorithmus von Briand et al. benötigt in 5 von 9 Softwaresystemen die geringste Anzahl zu simulierender Attributzugriffe. Für die verbleibenden 4 Systeme erreicht er immer den 2. Platz. Der Simulated Annealing Algorithmus erreicht mit einer durchschnittlichen Platzierung von 2,4 den 2. Platz. Er erreichte ebenfalls in 5 Softwaresystemen die geringste Anzahl simulierter Attributzugriffe, erreichte aber für das Softwaresystem CDK und Eclipse nur Platz

6, wodurch sich die schlechtere durchschnittliche Platzierung erklären lässt. Die verbleibenden Algorithmen zeigen, dass sie weniger gut geeignet sind, um die Anzahl simulierter Attributzugriffe zu minimieren.

Anzahl aufgebrochener Vererbungsbeziehungen
In allen Softwaresystemen ermitteln die Algorithmen von Briand et al. und Tai und Daniels Integrationsreihenfolgen, die keine Vererbungsbeziehungen aufzubrechen, womit beide Algorithmen sich den ersten Platz teilen. Aber auch der Simulated Annealing Algorithmus erzeugt für sieben Softwaresysteme eine Reihenfolge, die keine Vererbungsbeziehungen aufbricht. Für das Softwaresystem CDK und Eclipse landet er jedoch auf dem 5. bzw. 6. Platz. Dem genetischen Algorithmus gelingt es für 5 Softwaresysteme eine Reihenfolge zu finden, die keine Vererbungsbeziehungen aufbricht. Für Eclipse, Jetspeed, CDK und FOP ermittelt er eine Reihenfolge die nur knapp über der Anzahl aufgebrochener Vererbungsbeziehungen als Briand et al. und Tai und Daniels liegt.

Aus Tabelle 19 und den vorangegangenen Beschreibungen geht hervor, dass der Algorithmus von Briand et al. am besten geeignet ist, um den Simulationsaufwand einer Integrationsreihenfolge zu minimieren. Er minimiert sowohl die Anzahl zu simulierender Dateien und Abhängigkeiten, als auch die Anzahl aufgebrochener Vererbungsbeziehungen sowie die Anzahl zu simulierender Serviceaufrufe und Attributzugriffe. Auf der anderen Seite berücksichtigt dieser Algorithmus am schlechtesten den ausgewählten Testfokus für den Integrationstest. Der in dieser Arbeit entwickelte Algorithmus zur idealen Testfokusberücksichtigung zeigt jedoch Schwächen in der Berücksichtigung des Simulationsaufwandes.
Das Ziel muss es also sein, einen Algorithmus (oder eine Kombination aus existierenden Algorithmen) zu finden, um sowohl den Testfokus als auch den Simulationsaufwand zu berücksichtigen.

6.3. Testfokus und Simulationsaufwand

Das Ziel der Arbeit ist die Entwicklung eines neuen Ansatzes zur Ermittlung der Integrationsreihenfolge, der sowohl den Testfokus als auch den Simulationsaufwand berücksichtigt. Aus den Untersuchungen in den vorangegangenen Kapiteln geht hervor, das Reihenfolgen, optimiert für den Simulationsaufwand, den Testfokus vernachlässigen. Auf der anderen Seite vernachlässigt der Algorithmus zur idealen Testfokusberücksichtigung aus Kapitel 6.2.2.7 den Simulationsaufwand. Auch wenn dieser Ansatz innerhalb der eingeteilten Mengen $\mathbf{IR_{TP}}$ und $\mathbf{B_{TP}}$ den Simulationsaufwand berücksichtigt, indem er den Algorithmus von Briand et al. [BLW03] verwendet, werden zwischen den Mengen zu viele Abhängigkeiten aufgebrochen.
Um sowohl den Testfokus als auch den Simulationsaufwand zu berücksichtigen, eignen sich die graphenbasierten Ansätze weniger, da sie hinsichtlich ihrer Erweiterbarkeit eingeschränkt sind. Sie wurden entwickelt, um den Simulationsaufwand zu minimieren. Die heuristischen Ansätze können durch die verwendete Kostenfunktion leicht erweitert und parametrisiert werden. Aktuell berücksichtigt die Kostenfunktion nur den Simulationsaufwand. Diese Funktion kann angepasst werden, um zusätzlich den Testfokus in die Integrationsreihenfolgeermittlung einfließen zu lassen. Wie die Erweiterung der Kostenfunktion aussehen kann, wird im nachfolgenden Kapitel 6.3.1 vorgestellt. Die heuristischen Ansätze werden erneut auf die neun Softwaresysteme angewendet und dabei sowohl der Testfokus als auch der Simulationsaufwand berücksichtigt. Die Ergebnisse der Anwendung und der Vergleich mit den anderen Ansätzen ist in Kapitel 6.3.3 und in [BP09b] dargestellt.

Der Nachteil des Simulated Annealing Algorithmus ist, dass er sehr rechenintensiv ist und für die Ermittlung der Integrationsreihenfolge sehr viel Zeit in Anspruch nimmt. Im Kapitel 6.3.2 wird daher untersucht, ob die Kombination des Simulated Annealing mit existierenden Algorithmen zu schnelleren und vielleicht sogar besseren Ergebnissen führen kann.

6.3.1. Kostenfunktion

Die Kostenfunktion dient den heuristischen Verfahren zum Vergleich von zwei Integrationsreihenfolgen. Mit ihrer Hilfe kann entschieden werden, wann eine Reihenfolge besser als eine zweite in Bezug auf vordefinierte Parameter ist. Die in Kapitel 6.2.2.4, 6.2.2.5, und 6.2.2.6 vorgestellten Algorithmen verwendeten eine Kostenfunktion, die die Anzahl der zu simulierenden Serviceaufrufe, Attributzugriffe und Vererbungsbeziehungen berücksichtigt. Für jede aufgebrochene Abhängigkeit wird ein Zahlenwert ermittelt, der den Aufwand für das Simulieren dieser Abhängigkeit beschreibt (vgl. Kapitel 6.2.2.4). Der Wertebereich des Zahlenwerts liegt zwischen 0 und 1[1]. Die Fitness oder auch Simulationsaufwand **SA** einer Integrationsreihenfolge errechnet sich dann aus der Summe aller Aufwände der aufgebrochenen Abhängigkeiten.
Ähnlich zum Simulationsaufwand wird ein Wert gesucht, der die Berücksichtigung des Testfokus **TF** beschreibt. **TF** sollte mit **SA** vergleichbar sein, d.h. er sollte bei einer schlechten Reihenfolge in Bezug auf den Testfokus in etwa gleich hoch **SA** sein, wenn die Reihenfolge den Simulationsaufwand nicht berücksichtigt.
Die Werte **TF** und **SA** werden anschließend miteinander verrechnet (siehe Formel 3) um einen Wert zu ermitteln, der angibt, wie gut eine Reihenfolge den Testfokus und den Simulationsaufwand berücksichtigt. Die Formel zur Berücksichtigung beider Werte sollte möglichst einfach gehalten und durch den Anwender an seine Bedürfnisse angepasst werden können. Aus diesem Grund verwendet der vorgeschlagene Ansatz das gewichtete arithmetische Mittel (vgl. Formel 3). Die Gewichtungen W_{TF} und W_{SA} ermöglichen es, die Kostenfunktion an die Bedürfnisse der Anwender anzupassen. Zu beachten ist, dass die Summe aus W_{TF} und $W_{SA,}$ aufgrund der Verwendung des arithmetischen Mittels, den Wert 1ergeben sollte. In den späteren Untersuchungen werden die Gewichtungen jeweils gleich groß gewählt, um den Testfokus und den Simulationsaufwand in gleichem Maße zu berücksichtigen. Es ist aber denkbar eine Integrationsreihenfolge hinsichtlich des Simulationsaufwand besser zu optimieren (z.B. mit W_{SA} = 0,8) als hinsichtlich des Testfokus (z.B. mit W_{TF} = 0,2) oder umgekehrt.

Formel 3: Fitnessberechnung mit Berücksichtigung des Testfokus und des Simulationsaufwands

$$\mathbf{Fitness = (W_{TF} * TF * W_{SA} * SA) * 0,5}$$

W_{TF} … Gewichtung des Testfokus
W_{SA} … Gewichtung des Simulationsaufwands
TF … Testfokusberücksichtigung
SA … Simulationsaufwandsberücksichtigung

In ersten Versuchen wurde für **TF** die Anzahl der falsch integrierten Abhängigkeiten verwendet und mit dem Simulationsaufwand aus Formel 3 verrechnet. Dies führte dazu, dass die Anzahl der falsch integrierten Abhängigkeiten einen zu großen Einfluss auf die Integrationsreihenfolge hatten. Dem wurde in weiteren Experimenten entgegengesteuert, indem die Anzahl der falsch integrierten Abhängigkeiten durch die durchschnittliche Anzahl

[1] Dies gilt nur, wenn die Summer der Gewichte W_A, W_S und W_V genau 1 ergibt.

Abhängigkeiten pro Baustein dividiert wird (vgl. Formel 4). Dies liegt darin begründet, dass **TF** von der Anzahl Abhängigkeiten und Bausteine im Softwaresystem abhängig ist. Durch diese Anpassung von **TF** konnten in den Untersuchungen in etwa vergleichbare Werte für **SA** und **TF** ermittelt werden und als Grundlage für die Kostenberechnung einer Reihenfolge verwendet werden.

Formel 4: Berechnung der Testfokusberücksichtigung TF einer Integrationsreihenfolge

$$TF = FIA / DAB$$

FIA … Anzahl zu spät integrierter Abhängigkeiten
DAB … Durchschnittliche Anzahl Abhängigkeiten pro Baustein

Die vorgestellte Kostenfunktion kann in heuristischen Verfahren verwendet werden, um den Simulationsaufwand und die Testfokusauswahl bei der Integrationsreihenfolgeermittlung zu berücksichtigen. Die Ergebnisse der Anwendung der heuristischen Verfahren und der Vergleich mit den existierenden Algorithmen ist in Kapitel 6.3.3 zusammengefasst.

6.3.2. Kombination von Simulated Annealing mit existierenden Ansätzen

Der Nachteil des Simulated Annealing Algorithmus ist seine sehr lange Laufzeit, die bei der Parametrisierung, beschrieben in Kapitel 6.2.2.6, ein exponentielles Verhalten mit steigender Anzahl Bausteine aufweist.
Aus diesem Grund wird in diesem Kapitel untersucht, inwieweit eine Kombination des Simulated Annealing Algorithmus mit existierenden Algorithmen zu einer Verbesserung der Laufzeit führt. Die grundlegende Idee dahinter ist die Anpassung der zufälligen Startreihenfolge, um mit einer möglichst guten Lösung den Algorithmus zu starten. Durch das systematische Ausprobieren wird diese gute Lösung noch weiter verbessert. Dadurch kann die Anzahl der Versuche reduziert werden, die benötigt werden, um optimale Lösungen zu erhalten.
In vorangegangenen Untersuchungen hat sich gezeigt, dass es zwei Algorithmen gibt, die bei der Ermittlung der Integrationsreihenfolge die Testfokusauswahl (Algorithmus zur idealen Testfokusberücksichtigung, vgl. Kapitel 6.2.2.7) bzw. den Simulationsaufwand (Algorithmus von Briand, vgl. Kapitel 6.2.2.3) berücksichtigen. Für beide Algorithmen ist das Laufzeitverhalten in etwa gleich und deutlich kleiner als das Laufzeitverhalten des Simulated Annealing Algorithmus. Diese zwei Algorithmen können dem Simulated Annealing Algorithmus „vorgeschaltet“ werden, um eine bereits optimierte Startreihenfolge zu ermitteln. Auf der einen Seite kann der Simulated Annealing Algorithmus mit der von Briand et al. berechneten Reihenfolge starten und berücksichtigt bereits von Beginn an den Simulationsaufwand. Schrittweise wird diese Reihenfolge verändert, um möglichst den Testfokus in der Reihenfolge zu berücksichtigen. Dadurch wird der Simulationsaufwand verschlechtert, aber in stärkerem Maße die Testfokusberücksichtigung verbessert. Auf der anderen Seite kann der Algorithmus mit einer Reihenfolge starten, die bereits die Testfokusauswahl ideal unterstützt und verändert diese, indem der Simulationsaufwand berücksichtigt wird. Die Idee ist in Abbildung 26 veranschaulicht. Begonnen werden kann somit mit einer Reihenfolge, ermittelt durch den Algorithmus von Briand et al., oder mit einer Reihenfolge, ermittelt durch den Algorithmus zur idealen Testfokusberücksichtung. In beiden Fällen wird die Startreihenfolge verändert und eine Reihenfolge gesucht, die sowohl den Testfokus als auch den Simulationsaufwand minimiert. Auf diese Weise entstehen zwei Variationen des ursprünglichen Simulated Annealing Algorithmus.

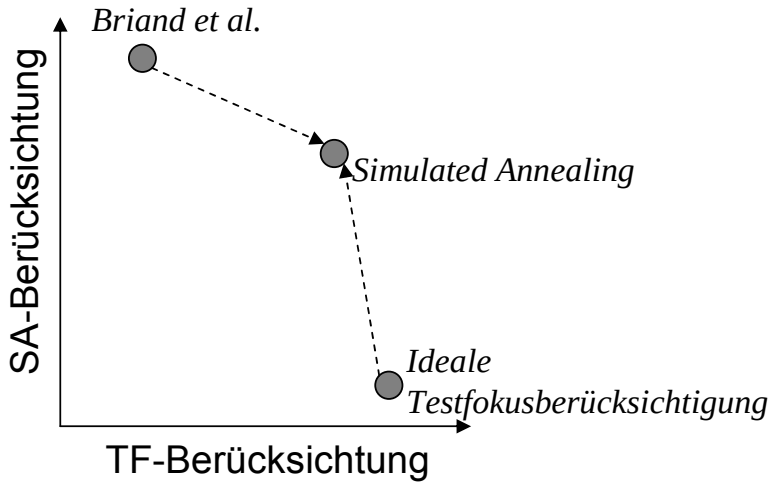


Abbildung 26: Mögliche Startreihenfolge für das Simulated Annealing

Durch das Starten mit einer bereits optimierten Reihenfolge kann die Anzahl der Versuche reduziert werden, die das Simulated Annealing durchführen muss, um eine bessere Reihenfolge zu finden. Das Reduzieren der Versuche wird durch die Parameter *Starttemperatur*, *Endtemperatur* und die *Anzahl Versuche mit gleicher Temperatur* erreicht. Da die Anzahl Versuche von der Anzahl der Bausteine im Softwaresystem abhängig gemacht wird, wird dieser Wert im Vergleich zum originalen Simulated Annealing auch nicht verändert. Da bereits eine gute Lösung verwendet wird, kann die Anzahl der akzeptierten schlechten Lösungen reduziert werden. Dies geschieht, indem die aktuelle Temperatur gesenkt wird. Im vorliegenden Fall bedeutet das, dass die Startemperatur des Simulated Annealing Algorithmus soweit gesenkt wird, dass nur wenige schlechte Lösungen akzeptiert werden. Bei Beibehaltung der Endtemperatur wird somit die Laufzeit gekürzt.
Die Ergebnisse der Anwendung dieser modifizierten Variante des Simulated Annealing Algorithmus ist im nachfolgenden Kapitel dargestellt. Zusätzlich werden die Ergebnisse des modifizierten Simulated Annealing mit den Ergebnissen der anderen Algorithmen verglichen.

6.3.3. Fallstudien

Die Untersuchungen aus Kapitel 6.2.2.8 werden in einer zweiten Untersuchung wiederholt. Es werden alle sieben Algorithmen auf die neun Beispielsysteme angewendet. Jedoch verwenden die heuristischen Algorithmen die angepasste Kostenfunktion, um sowohl die Testfokusauswahl als auch den Simulationsaufwand zu berücksichtigen. Ergänzend werden die beiden Variationen des Simulated Annealing Algorithmus verwendet. Zusätzlich zu den acht Maßzahlen, die bei der ersten Untersuchung erhoben wurden, werden die Fitness, ermittelt durch die Kostenfunktion, der berechnete Simulationsaufwand **SA** und die berechnete Testfokusberücksichtigung **TF** erhoben.
Für die Kostenfunktion werden die Gewichtungen $\mathbf{W_{TF}}$ = 0,5 und $\mathbf{W_{SA}}$ = 0,5 mit dem Ziel verwendet, Reihenfolgen zu erhalten, die sowohl den Testfokus als auch den Simulationsaufwand in gleichem Maße berücksichtigen. Die Parametrisierung des genetischen Algorithmus wird aus der ersten Untersuchung übernommen. Für den originalen Simulated Annealing Algorithmus wird zum einen die gleiche Konfiguration verwendet, wie in der ersten Untersuchung, d.h. zufällige Startreihenfolge, Startemperatur = 1,0, Endtemperatur = 0,001, Anzahl Versuche = Anzahl Bausteine im System. Zusätzlich werden zwei weitere Versuche durchgeführt, in denen mit einer optimierten Startreihenfolge begonnen wird. Zum einen mit der von Briand et al. ermittelten Reihenfolge, die den Simulationsaufwand sehr gut berücksichtigt, und zum anderen mit einer vom Algorithmus zur

idealen Testfokusberücksichtigung ermittelten Reihenfolge. Die Konfiguration ist dabei bis auf eine Ausnahme mit der oberen vergleichbar. Die Ausnahme betrifft den Parameter der Starttemperatur, der mit einem Wert von 0,01 initialisiert wird. Durch das Senken der Starttemperatur bei gleichzeitiger Beibehaltung der Endtemperatur wird die Anzahl der benötigten Iterationen reduziert. Gleichzeitig wird von Beginn an die Wahrscheinlichkeit gesenkt, schlechte Lösungen in Bezug auf die Fitness zu akzeptieren.
Das Ziel der Untersuchung ist es zu überprüfen, inwieweit die ermittelten Integrationsreihenfolgen sowohl den Testfokus als auch den Simulationsaufwand berücksichtigen. Idealerweise werden Integrationsreihenfolgen ermittelt, die einen geringen Simulationsaufwand haben **und** ein Großteil der Testfokus Abhängigkeiten früh integrieren.
Die Ergebnisse der einzelnen Algorithmen pro Softwaresystem sind im Anhang C in Tabelle 28 bis Tabelle 36 zu finden. Für den Vergleich der Algorithmen werden wie in Kapitel 6.2.2 die durchschnittlichen Platzierungen verwendet. Diese sind für die zweite Untersuchung in Tabelle 20 dargestellt.

Tabelle 20: Durchschnittliche Platzierung der Algorithmen mit Testfokusberücksichtigung

	Fitness	Simulations-aufwand	Testfokus-berücksichtigung	Dauer	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
Ideale Testfokus-berücksichtigung	5,4	7,9	**1,0**	2,8	6,7	6,6	6,7	6,9	7,9	**1,0**	**1,0**
Tai&Daniels	7,4	5,2	8,2	4,2	5,0	6,6	7,0	5,3	**1,0**	8,2	8,2
Original Simulated Annealing Startreihenfolge Zufall	**1,7**	3,3	2,6	9,0	5,0	4,2	3,7	3,7	2,3	2,6	2,6
Simulated Annealing Startreihenfolge mit idealer Testfokusberücksichtigung	1,9	4,4	2,6	7,1	4,9	4,6	3,9	3,3	4,4	2,6	2,6
Simulated Annealing Startreihenfolge nach Briand	3,0	2,7	4,2	7,7	3,7	2,9	2,8	3,0	2,7	4,2	4,2
Genetischer Algorithmus	4,2	5,9	4,8	6,2	7,7	7,1	6,9	6,8	4,3	4,8	4,8
Briand et al.	6,0	**1,0**	8,7	2,9	**1,4**	**1,0**	**1,1**	**1,6**	**1,0**	8,7	8,7
Le Traon et al.	6,3	5,6	6,2	**1,1**	**1,6**	3,1	4,1	5,2	5,9	6,2	6,2
Random	9,0	9,0	6,7	4,0	9,0	9,0	8,9	9,0	9,0	6,7	6,7

In der ersten Untersuchung wurde gezeigt, dass die graphenbasierten Algorithmen (Le Traon et al., Briand et al.) die Testfokusauswahl bei der Ermittlung der Integrationsreihenfolge nicht berücksichtigen. Die beste Berücksichtigung des Testfokus erreichte der Algorithmus von Le Traon et al., der im Durchschnitt ca. 33% der Testfokus-Abhängigkeiten frühzeitig integrierte. Durch die Anpassung der Kostenfunktion sind die heuristischen Ansätze in der Lage, diesen Wert zu erhöhen. Der genetische Algorithmus integriert im Durchschnitt ca. 67% der Testfokus-Abhängigkeiten frühzeitig. Der Simulated Annealing Algorithmus mit der Startreihenfolge von Briand et al. integriert ca. 75% der Testfokus Abhängigkeiten frühzeitig. Mit einer Startreihenfolge, ermittelt durch den Algorithmus zur idealen Testfokusberücksichtigung, erreicht der Simulated Annealing Algorithmus durchschnittlich 90% richtig integrierter Abhängigkeiten. Der Simulated Annealing Algorithmus, gestartet mit einer zufälligen Reihenfolge integriert durchschnittlich 88% der Abhängigkeiten zum richtigen Zeitpunkt. Nach wie vor berücksichtigt der selbst entwickelte Algorithmus zur idealen Berücksichtigung des Testfokus zu 100%. Der zufallsbasierte Ansatz integriert nur 27% der Testfokus Abhängigkeiten richtig.

Hinsichtlich der Minimierung der Kostenfunktion erreicht der Simulated Annealing Algorithmus mit der zufälligen Startreihenfolge in acht von neun Softwaresystemen die beste Platzierung, gefolgt von dem Simulated Annealing Algorithmus, gestartet mit der Reihenfolge mit der idealen Testfokusberücksichtigung bzw. gestartet mit der Reihenfolge nach Briand et al. Diese vordere Platzierung wird erreicht durch die Berücksichtigung des Simulationsaufwands und der Testfokusauswahl. Der Algorithmus zur idealen Testfokusberücksichtigung erreicht in allen neun Softwaresystemen die beste Platzierung hinsichtlich der Testfokusberücksichtigung. Die drei Simulated Annealing Varianten nehmen die drei nachfolgenden Platzierungen ein, gefolgt vom genetischen Algorithmus.
Bei der Berücksichtigung des Simulationsaufwands schneidet wie in der ersten Untersuchung der Algorithmus von Briand et al. am besten ab. Aber auch hier werden die folgenden Plätze durch die drei Varianten des Simulated Annealing Algorithmus eingenommen. Da sich der Simulationsaufwand aus der Anzahl simulierter Serviceaufrufe, Attributzugriffe und der Anzahl der aufgebrochenen Vererbungsbeziehungen ermittelt, schneiden die drei Varianten des Simulated Annealing Algorithmus auch in diesen drei Kategorien gut ab.
Da die Kostenfunktion die Anzahl zu simulierender Dateien und Abhängigkeiten nicht berücksichtigt, fließen diese nicht direkt in die ermittelte Integrationsreihenfolge, ermittelt durch die heuristischen Verfahren, mit ein. Der Algorithmus von Le Traon schneidet hier deutlich besser ab, als die drei Varianten des Simulated Annealing oder des genetischen Algorithmus.
Was die Laufzeit der Algorithmen angeht, so ist nach wie vor der Algorithmus von Le Traon der schnellste. In acht von neun Softwaresystemen errechnet er die Integrationsreihenfolge in der kürzesten Zeit. Für das Softwaresystem Eclipse ermittelt er eine Integrationsreihenfolge in 166 Sekunden. Am schlechtesten schneiden die drei Varianten des Simulated Annealing Algorithmus ab. Der Simulated Annealing Algorithmus, gestartet mit einer zufälligen Reihenfolge, ist in allen neun Softwaresystem am langsamsten. Seine längste Berechnungsdauer beträgt 3,4 Tage für Eclipse. Die zwei Varianten des Simulated Annealing Algorithmus sind in allen Fällen schneller als die originale Version des Algorithmus, benötigten aber für die Berechnung der Integrationsreihenfolge für Eclipse jeweils ca. einen Tag. Der Zeitgewinn durch die Anpassung der Startreihenfolge beläuft sich im Durchschnitt über alle Softwaresysteme auf den Faktor 3 im Vergleich zur zufälligen Startreihenfolge. Der genetische Algorithmus schaffte die Berechnung in 12,3 Stunden.

Die Untersuchungen zeigen, dass es mit Hilfe der heuristischen Ansätze des Simulated Annealing Algorithmus und der genetischen Algorithmen möglich ist, Integrationsreihenfolgen zu bestimmen, die sowohl den Testfokus als auch den Simulationsaufwand berücksichtigen. Derzeit sind sie die einzigen Ansätze, die dazu in der Lage sind. Der einzige Nachteil dieser Verfahren ist die lange Laufzeit, um eine Reihenfolge zu ermitteln. Insbesondere bei sehr großen Softwaresystemen nimmt die Berechnung mehrere Tage in Anspruch.
Weiterhin hat sich gezeigt, dass der Simulated Annealing Algorithmus, gestartet mit einer zufälligen Reihenfolge, bei der gegebenen Konfiguration Schwächen bei sehr großen Softwaresystemen zeigt. Sowohl in der ersten als auch in der zweiten Untersuchung versagte er bei dem Softwaresystem Eclipse. Die zwei Varianten des Simulated Annealing Algorithmus zeigten diese Schwankungen nicht und ermittelten zuverlässig sehr gute Ergebnisse. Der genetische Algorithmus lieferte ebenfalls gute Ergebnisse, wobei sein Vorteil die geringere Laufzeit im Vergleich der Simulated Annealing Algorithmen ist.

7. Werkzeugunterstützung

Die in den vorangegangenen Kapiteln vorgestellten Ansätze zur Testfokusauswahl und zur Integrationsreihenfolgermittlung werden durch verschiedene Werkzeuge unterstützt. Für die Testfokusauswahl werden existierende Werkzeuge in Kombination mit im Rahmen dieser Arbeit selbst entwickelten Werkzeugen verwendet. Hierbei kommen ein existierender Quelltextanalysator, ein Statistikwerkzeug und kleinere in Java realisierte Werkzeuge zum Einsatz. Die Ergebnisse der Testfokusauswahl können in einem Format exportiert werden, das von einem in Java realisierten Programm eingelesen werden kann. Diese Ergebnisse werden verwendet, um eine Integrationsreihenfolge zu ermitteln.
Die verwendeten Werkzeuge werden nachfolgend kurz vorgestellt und ihr Einsatzzweck sowie ihr Einsatzzeitpunkt in den vorgestellten Verfahren erläutert.

7.1. Werkzeuge zur Testfokusauswahl

Wichtig für die Anwendung dieses Verfahrens zur Testfokusauswahl ist das Vorhandensein von Informationen über die Abhängigkeiten und die Fehleranzahl der beteiligten Bausteine. Im Rahmen dieser Arbeit wurde eine Werkzeugunterstützung namens SWAN (**S**oft**W**are **AN**alyser) erarbeitet, die es ermöglicht, Softwaresysteme zu analysieren, um die Abhängigkeit zwischen Java-Quelltextdateien zu identifizieren und bis zu 13 Abhängigkeitseigenschaften zu erheben. Parallel werden die Informationen aus Fehlermanagementwerkzeugen verwendet, um die Fehleranzahl pro Datei zu ermitteln. Die Schritte, die dabei auszuführen sind, sind in Abbildung 27 dargestellt. Aus der Abbildung ist ersichtlich, dass ein Großteil der Schritte von SWAN unterstützt wird. Lediglich die Korrelationsanalyse muss durch ein Statistik-Werkzeug (z.B. das kommerzielle Werkzeug SPSS [SP09]) durchgeführt werden.
SWAN erlaubt es, verschiedene Softwaresysteme und ihre Informationen darüber zu verwalten. Für jedes Softwaresystem können frühere Versionen verwaltet werden. Für eine Version wird der zu analysierende Quelltext mit Hilfe des Open-Source-Werkzeugs SVNKit [SVK09] automatisch aus dem Versionsverwaltungssystem Subversion [Su09] ausgecheckt. Hierfür werden die Verbindungsdaten zum entsprechenden Subversion Server und die Authentifizierungsdaten benötigt, die über die SWAN-Oberfläche eingegeben und in SWAN gespeichert werden. Anschließend kann über die Oberfläche die Quelltextanalyse angestoßen werden. Es wird der Quelltextanalysator Sissy [Si09] im Hintergrund gestartet. Dieser liefert nach Abschluss der Analysen eine Erfolgsmeldung. Sissy benötigt eine Datenbank, um eine abstrakte Repräsentation des Quelltexts zu exportieren. Die Verbindungsdaten werden über die SWAN-Oberfläche beim Anlegen eines neu zu verwaltenden Softwaresystems spezifiziert. Nachdem die Version ausgecheckt und analysiert worden ist, können Informationen über die Fehler-IDs aus dem Fehlermanagementsystem importiert werden. Hierfür wird eine CSV[1]-Datei mit den Fehlerdaten (Fehler-ID, Reportdatum, Beschreibungstext) eingelesen und in die Datenbank exportiert. Eine erneute Verbindung zum Versionsverwaltungssystem mit dem SVNKit erlaubt es, die Versionshistorie des Softwaresystems auszulesen. Innerhalb dieser Versionshistorie werden die Fehler-IDs in den Kommentaren der Entwickler gesucht. Auf diese Weise wird die Fehleranzahl pro Datei ermittelt (vgl. Kapitel 5.4.1 und Kapitel 5.4.2.1). Für jede Datei wird die identifizierte Fehleranzahl in die Datenbank exportiert, die bereits das Modell des Quelltexts enthält. Liegen sowohl das Modell des Quelltexts als auch die

[1] CSV … comma separated value

Fehleranzahl pro Datei vor, kann der Benutzer auf der Oberfläche bis zu 13 Abhängigkeitseigenschaften auswählen, die im Anschluss erhoben werden. Das Ergebnis dieser Analyse ist eine CSV-Datei, die alle identifizierten Abhängigkeiten, die erhobenen Eigenschaften und die Fehleranzahl der beteiligten Bausteine enthält. Diese Datei kann anschließend in ein Statistikwerkzeug (z.B. SPSS [SP09]) eingelesen werden, um die Korrelationsanalysen durchzuführen.
Die Architektur der Werkzeugunterstützung ist in Abbildung 28 dargestellt. Es wurde dabei das Architekturmuster *Master-Slave* [HR02] verwendet, wobei die Anwendung SWAN als Master fungiert. In der Abbildung steht die Komponente SWAN für die realisierte Oberfläche. Neben der Erzeugung der Oberfläche und der Interaktion mit den Benutzern stößt sie alle notwendigen externen Programme an, kümmert sich darum, dass die externen Programme mit den richtigen Daten versorgt werden und realisiert gleichzeitig die Kommunikation mit den Benutzern. SWAN verwaltet darüber hinaus alle Daten, die für den reibungslosen Ablauf notwendig sind, d.h. Informationen über die Softwaresysteme und ihre Versionen, den Analysefortschritt für jede Version, die Verbindungsdaten zu den Versionsverwaltungssystemen, die Verbindungsdaten zu den Datenbanken und die eingelesenen Fehlerdaten. SWAN fängt Ausnahmen und Fehlermeldungen ab und speichert sie in die Datenbank. Die Benutzer haben so die Möglichkeit zu erkennen, welche Schritte bei der Analyse fehlgeschlagen sind. Diese Schritte können bei Bedarf wiederholt werden.

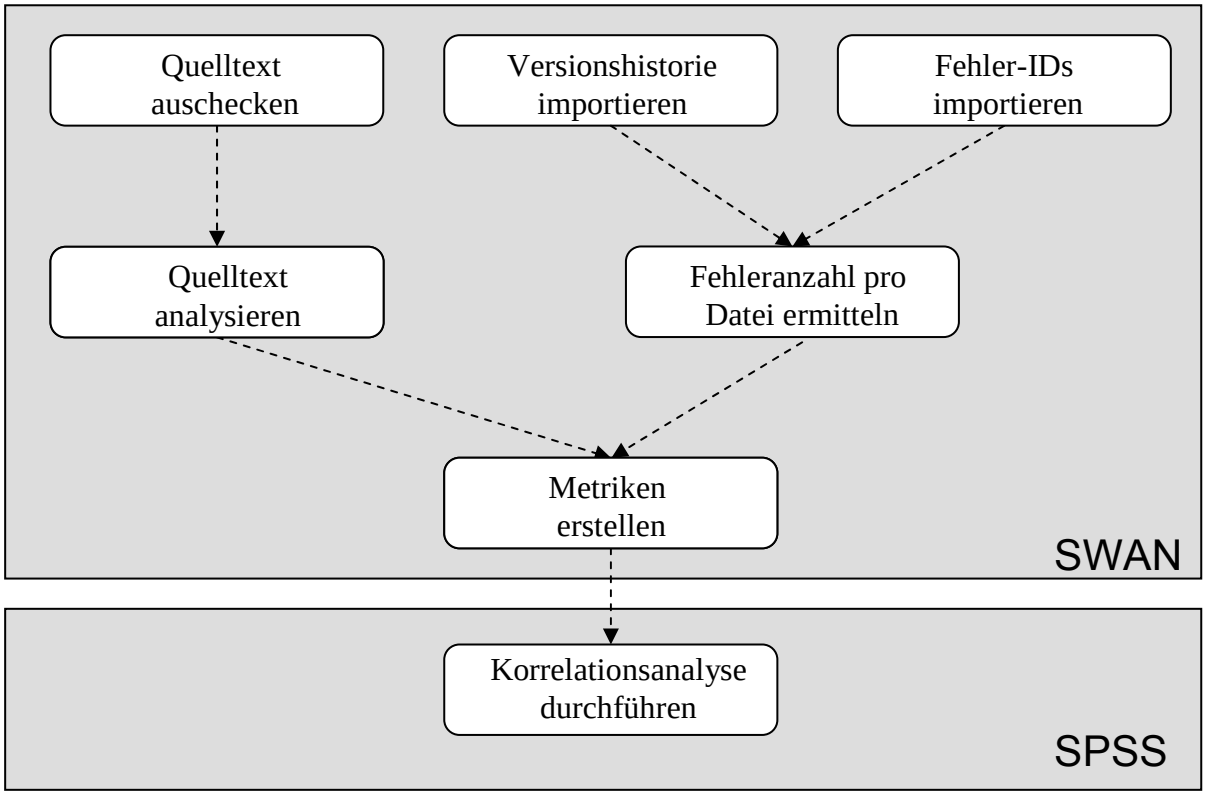


Abbildung 27: Werkzeugunterstützte Teilschritte der Testfokusauswahl

SWAN ist in Ruby on Rails [ROR09] realisiert. Ruby on Rails stellt ein Rahmenwerk zur Verfügung, um Software in Ruby [Ru09] zu realisieren. Ruby ist eine Skriptsprache, die von einem Interpreter ausgewertet und ausgeführt wird. Ruby on Rails stellt zudem eine umfassende Menge von Basisskripten zur Verfügung, die das einfache Erstellen von dynamischen Webseiten erlauben. Es stellt Funktionalität für das Lesen und Schreiben von Informationen in eine Datenbank bereit und ermöglicht das einfache Zusammenstellen der Webseiteninhalte. Ein Plugin zu Ruby on Rails stellt JRuby [JRu09] zur Verfügung, das es ermöglicht, in Java realisierte Softwaresysteme in der virtuellen Maschine von Java

auszuführen. Diese Kombination erlaubt es, die existierenden Softwaresysteme in SWAN einzubinden und somit ein Werkzeug bereit zu stellen, das die notwendigen Informationen für den Integrationstest über ein Softwaresystem und seine Versionen erstellen und verwalten kann.

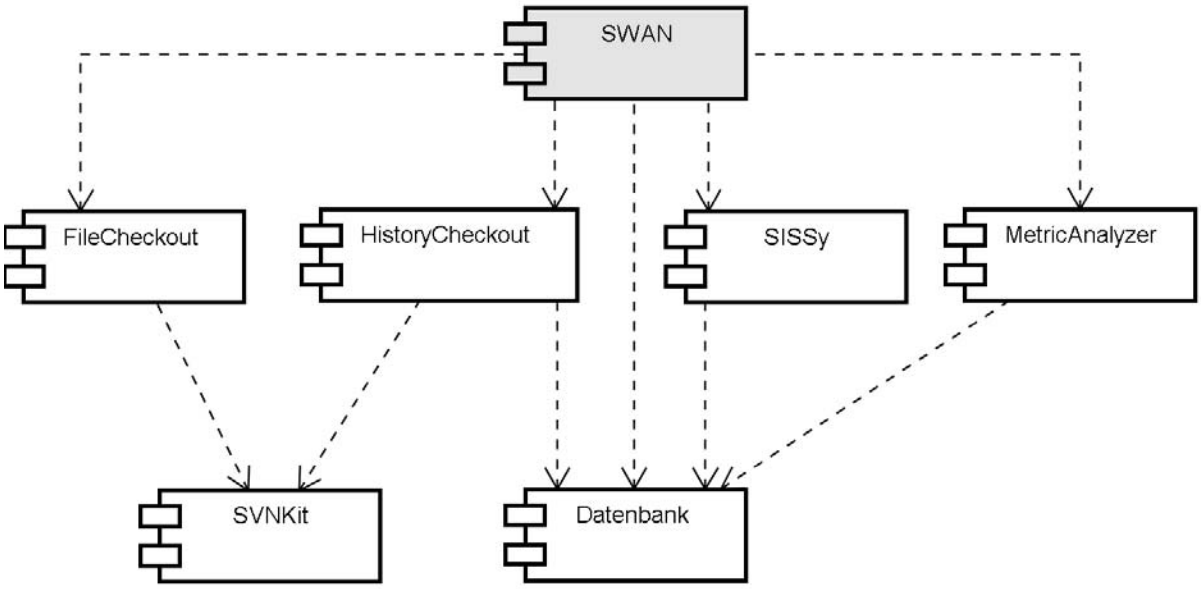


Abbildung 28: Architektur SWAN

SWAN liefert alle Ergebnisse, die für die Testfokusauswahl benötigt werden. Die Ergebnisse werden im CSV Format exportiert und werden anschließend von SPSS eingelesen. Die Korrelationsanalyse und die Testfokusauswahl selbst werden in SPSS durchgeführt.

7.2. Werkzeuge für die Integrationsreihenfolgeermittlung

Das Ergebnis der Testfokusauswahl ist eine Tabelle, die für jede Abhängigkeit der zu integrierenden Version eines Softwaresystems den Testfokus enthält. Darüber hinaus werden Informationen, die für die Ermittlung der Integrationsreihenfolge benötigt werden, z.B. die Anzahl Serviceaufrufe, Anzahl Attributzugriffe, Vererbungsbeziehung, Namen der Bausteine, ebenfalls in der Tabelle festgehalten. Diese Tabelle wird von dem, in dieser Arbeit entwickelten Java Programm *TOC* (**T**est **O**rder **C**alculator) eingelesen, das für die Berechnung der Integrationsreihenfolge verantwortlich ist.
TOC ist in Java realisiert und wird über eine Konsole gestartet. Über die Konsole wird ihm eine Konfigurationsdatei mitgegeben, die es erlaubt, die Algorithmen auszuwählen, die für die Integrationsreihenfolgeermittlung verwendet werden sollen. Die Parameter, mit den TOC konfiguriert werden kann, sind im Anhang D: *Konfigurationsparameter für TOC* in Tabellenform dargestellt. Für die heuristischen Algorithmen können spezielle Einstellungen vorgenommen werden, um die Ergebnisse an den Projektkontext anzupassen. Darüber hinaus können die Ergebnisse in eine CSV Datei exportiert werden.
TOC ist entwickelt worden, um sowohl die Praxis als auch die Forschung zu unterstützen. Für die Praxis liefert TOC eine Integrationsreihenfolge, die an die jeweiligen Projektkontexte angepasst werden kann. Es stehen unterschiedliche Algorithmen zur Verfügung, um nach Ermittlung der Reihenfolgen die beste auszuwählen. Für die Forschung stellt TOC ein Rahmenwerk zur Verfügung, um neue Algorithmen zur Ermittlung der Integrationsreihenfolge zu erproben.

Die Hauptklassen und Schnittstellen von TOC sind in Abbildung 29 dargestellt. Neue Algorithmen können schnell in die bestehe Architektur integriert werden und stehen TOC anschließend zur Verfügung. Hierzu muss nur die Schnittstelle *TestOrderCalculator* und die dazugehörige Operation *calculateIntegrationTestOrder* implementiert werden. In Abbildung 29 ist zu sehen, dass alle oben erläuterten Algorithmen durch jeweils eine Hauptklasse realisiert werden (z.B. Briand). Zusätzlich benötigen diese Klassen unter Umständen zusätzliche Hilfsklassen zur Berechnung. Um diese zusätzlichen Klassen besser strukturieren zu können, wird jeder realisierte Algorithmus in einem eigenen Java-Paket untergebracht. In diesem Paket werden alle zusätzlich benötigten Klassen zugeordnet. Klassen, die von allen Algorithmen benötigt werden, z.B. die Klasse *Item*[1] oder die Klasse *Dependency*[2], werden in einem gemeinsamen Paket *util* verwaltet.

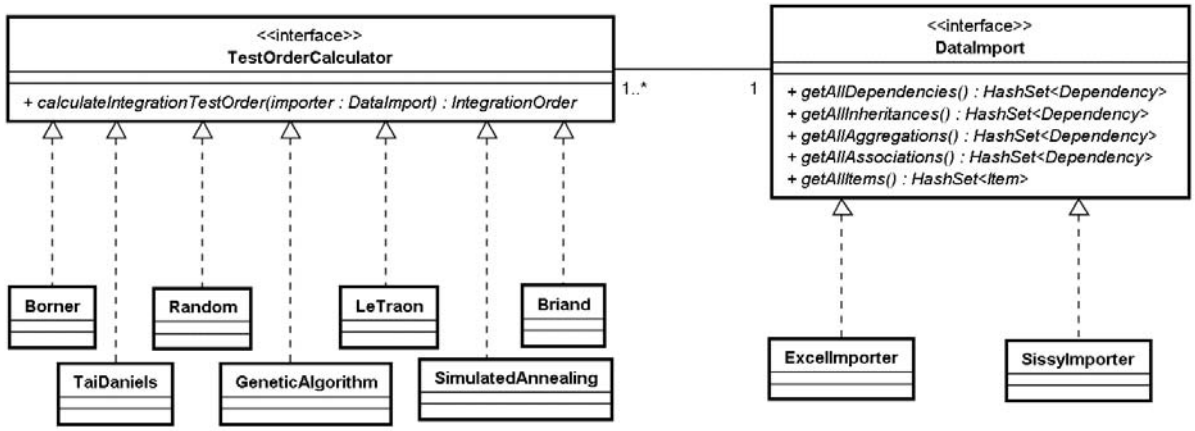


Abbildung 29: Hauptklassen und Schnittstellen von TOC

Zur Erweiterung können neben den bereits existierenden Import-Typen *ExcelFile* und *SissyDatabase* weitere Import-Typen leicht integriert werden. Eine gemeinsame Schnittstelle *(DataImport)* erlaubt es, schnell neue Importer zu realisieren und diese sofort in den existierenden Algorithmen einzusetzen. Somit können UML-Diagramme, die mit externen Werkzeugen modelliert werden, über die zusätzliche Schnittstelle mit eingebunden werden.

[1] Die Klasse *Item* repräsentiert einen Baustein des zu integrierenden Softwaresystems
[2] Die Klasse *Dependency* repräsentiert eine Abhängigkeit des zu integrierenden Softwaresystems

8. Zusammenfassung und Ausblick

Der Schwerpunkt dieser Arbeit lag in der Entwicklung neuer und innovativer Ansätze zur Unterstützung der beteiligten Rollen im Integrationstest.
Hierzu wurde zu Beginn der Promotionsarbeit ein Integrationstestprozess (Kapitel 3) definiert, der, im Gegensatz zu existierenden Testprozessbeschreibungen, sowohl die Eigenheiten des Integrationstests berücksichtigt als auch die neue Sichtweise der Entscheidungen mit einbringt. Dies ermöglicht es den auszuführenden Rollen, die Unterschiede des Integrationstestprozesses im Vergleich zu anderen Testprozessen (System- oder Unittest) zu verdeutlichen. Darüber hinaus erlaubt die neue Sichtweise, die Entscheidungen, die im Prozess getroffen werden müssen, explizit und bewusst zu treffen. Hierzu müssen beteiligten Rollen über Alternativen nachdenken, diese bewerten und letztendlich die beste Alternative auswählen und dokumentieren. Somit kann auch später noch nachvollzogen werden, warum der Integrationstestprozess so abgelaufen ist. Zusätzlich beschreibt diese Arbeit die Einflüsse der Entscheidungen untereinander. Es wird verdeutlicht, welche späteren Entscheidungen des Testprozesses von welchen abhängig sind. Somit wird klar, welche Auswirkungen das Treffen einer Entscheidung auf spätere Entscheidungen hat.
Im weiteren Verlauf der Arbeit wurden neue Ansätze vorgestellt, die das Treffen von zwei Entscheidungen innerhalb des Testprozesses unterstützen und vereinfachen. Dies sind zum einen die Entscheidung zum Testfokus und zum anderen die Entscheidung zur Integrationsreihenfolge.

Die Entscheidung der Testfokusauswahl legt fest, welche Abhängigkeiten des Softwaresystems zu testen sind. Der in dieser Arbeit neu entwickelte Ansatz zur Testfokusauswahl (Kapitel 5) verwendet Informationen früherer Versionen des zu integrierenden Softwaresystems. Hierzu wurden erstmals Abhängigkeitseigenschaften verwendet, um Zusammenhänge zwischen diesen und der Fehleranzahl der beteiligten Bausteine aufzudecken. Für die Entscheidung, welche Abhängigkeitseigenschaften für die Analyse der Zusammenhänge verwendet werden können, stellte die Promotionsarbeit einen Katalog von Abhängigkeitseigenschaften (Kapitel 4) zur Verfügung. Zu jeder Eigenschaft im Katalog wird erläutert, welche Fehler durch die Existenz dieser Eigenschaften auftreten können.
Zur Identifikation von Zusammenhängen zwischen den Abhängigkeitseigenschaften und der Fehleranzahl der Bausteine in früheren Versionen wurden spezielle statistische Tests verwendet. Mit diesen war es möglich, Eigenschaften zu identifizieren, die auf eine erhöhte Fehleranzahl in den Abhängigkeiten hindeuten. Diese Eigenschaften wurden in der aktuellen Version des zu integrierenden Systems verwendet, um die Abhängigkeiten auszuwählen, die im Integrationstestprozess getestet werden müssen. Der neue Ansatz wurde an zwei existierenden, realistisch großen Softwaresystemen evaluiert. In dieser Fallstudie zeigte sich, dass durch die Anwendung des Ansatzes Abhängigkeiten ausgewählt wurden, die eine erhöhte Fehleranzahl in den beteiligten Bausteinen aufwiesen.

Die Entscheidung zur Integrationsreihenfolge legt fest, in welcher Reihenfolge das Softwaresystem schrittweise zum Gesamtsystem zusammengesetzt wird, wobei zwei Optimierungskriterien von besonderem Interesse sind: der festgelegte Testfokus und der Simulationsaufwand für Bausteine. Derzeit existierende Ansätze berücksichtigen nur den Simulationsaufwand, ignorieren aber den festgelegten Testfokus. Der in dieser Arbeit vorgestellte Ansatz lässt als erster Ansatz sowohl den Testfokus als auch den

Simulationsaufwand in die Ermittlung der Integrationsreihenfolge einfließen. Hierzu kamen heuristische Algorithmen, wie z.B. der Simulated Annealing Algorithmus zum Einsatz. In den durchgeführten Fallstudien an neun real existierenden Software Systemen wurde evaluiert, dass der neue Ansatz Integrationsreihenfolgen ermitteln kann, die sowohl den Testfokus als auch die Simulationsreihenfolge berücksichtigen.

Sowohl der Ansatz zur Testfokusauswahl als auch die Algorithmen zur Integrationsreihenfolgeermittlung lassen sich sehr gut durch Werkzeuge automatisieren. Die in dieser Arbeit eingesetzten Werkzeuge wurden in Kapitel 7 beschrieben. Sie stellten eine Kombination aus existierenden und selbst entwickelten Werkzeugen dar. Zur Extraktion der Abhängigkeiten und ihrer Eigenschaften aus dem Quelltext sowie zur Ermittlung der Fehleranzahl pro Baustein wurde ein Werkzeug namens SWAN entwickelt. SWAN vereinigte in sich frei verfügbare Open Source Programme und selbst entwickelte Analysewerkzeuge.
Die Ergebnisse von SWAN wurden mit kommerziellen Werkzeugen für die statistische Analyse weiterverarbeitet, um den Testfokus für die aktuelle Version festzulegen.
Ein zusätzlich entwickeltes Werkzeug erlaubte es, die in der Testfokusauswahl erstellten Ergebnisse weiter zu verwenden, um eine Integrationsreihenfolge zu ermitteln.

Die im Rahmen dieser Arbeit neu entwickelten Ansätze tragen wesentlich zur Unterstützung des Integrationstests bei und eröffnen darüber hinaus neue Forschungsfelder im Bereich der Qualitätssicherung.
Der Integrationstestprozess mit den vorgestellten Entscheidungen muss in realen Projekten aber erst Anwendung finden, um seine Anwendbarkeit in der Praxis zu bestätigen. Dies kann am Besten erfolgen, wenn eine geeignete Werkzeugunterstützung vorliegt. Eine solche Werkzeugunterstützung würde es ermöglichen, die Entscheidungen explizit zu treffen und zu dokumentieren. Darüber hinaus muss dieses Werkzeug mit den weiteren Werkzeugen des Testprozesses verbunden werden, um die Verfolgbarkeit zwischen Entscheidungen, Artefakten und Aktivitäten zu gewährleisten. Auf diese Weise kann sichtbar gemacht werden, welche Artefakte für das Treffen von Entscheidungen notwendig sind und in welchen Aktivitäten diese getroffen werden.
Der vorgestellte Katalog von Abhängigkeitseigenschaften stellt keinen vollständigen Satz an Eigenschaften dar, die eine Abhängigkeit besitzen kann. Im Rahmen dieser Arbeit wurden nur die Eigenschaften identifiziert, die für den Integrationstest wichtig sind. Der Katalog kann aber auch außerhalb des Integrationstests eingesetzt werden. So können die Informationen über Abhängigkeiten frühzeitig definiert werden. Diese detaillierteren Informationen liefern eine präzise Vorlage für die Implementierung. Darüber hinaus können die Informationen zur Komplexitätsmessung von Software eingesetzt werden. Je mehr Informationen über eine Abhängigkeit verfügbar sind, desto besser lassen sich die Abhängigkeiten und somit bestimmte Teile einer Software miteinander vergleichen.
Der Ansatz der Testfokusauswahl konzentrierte sich auf das Aufdecken von Zusammenhängen zwischen Eigenschaften und der Fehleranzahl. Dabei wurden in dieser Arbeit jeweils nur eine einzelne Eigenschaft und ihre möglichen Korrelationen mit der Fehleranzahl untersucht. In zukünftigen Arbeiten ist zu untersuchen, inwieweit das Kombinieren von Eigenschaften zu besseren Vorhersagen führen kann. Es müssen geeignete Kombinationsmöglichkeiten für Eigenschaften gefunden werden, um die verschiedenen Gruppen der Abhängigkeiten miteinander zu verbinden. Das Ziel ist es, die Vorhersagen für die fehleranfälligen Abhängigkeiten noch präziser zu machen, um die wenigen Testressourcen besser auf die fehleranfälligen Ressourcen zu verteilen.
Die Algorithmen der Integrationsreihenfolgeermittlung müssen hinsichtlich ihrer Laufzeit stark verbessert werden. Die graphbasierten Algorithmen haben gegenüber den heuristischen

Algorithmen den Vorteil der geringeren Laufzeit. Die heuristischen Algorithmen versuchen durch „Probieren“ gute Lösungen zu finden, wobei sie sehr viele Versuche benötigen und deshalb viel Zeit benötigen. Das Ziel muss es sein, die Anzahl der Versuche zu verringern, ohne die Ergebnisse zu verschlechtern. Eine Möglichkeit wurde bereits in dieser Arbeit vorgestellt, indem versucht wurde, mit einer bereits „guten“ Lösung zu starten. Ein weiterer Ansatz für die Laufzeitoptimierung der heuristischen Algorithmen stellt die Parallelisierung dar. Innerhalb einer Iteration der heuristischen Ansätze können die verschiedenen Modifikationen der Reihenfolgen zur Ermittlung neuer Algorithmen parallel durchgeführt werden. Insbesondere der genetische Algorithmus bietet sich hier an. Die Chromosomen einer Population können parallel modifiziert werden, wodurch die neue Population schneller errechnet werden kann. Für den Simulated Annealing Algorithmus muss erarbeitet werden, auf welche Weise er erweitert oder angepasst werden muss, um durch Parallelisierung schneller Ergebnisse zu liefern.
Die aktuellen Algorithmen zu Integrationsreihenfolgeermittlung berücksichtigen derzeit nur eine Einteilung des Testfokus in zwei Gruppen: *Testen* und *Nicht-Testen*. In zukünftigen Arbeiten ist zu untersuchen, wie eine feinere Einteilung der Gruppen stattfinden kann. Der Ansatz zur Testfokusauswahl beispielsweise liefert die Einteilung in vier Gruppen. Derzeit können aber keine Aussagen gemacht werden, ob eine Reihenfolge den Testfokus hinsichtlich der vier Gruppen unterstützt. Es ist notwendig, den Ansatz entsprechend anzupassen, was durch die Verwendung der heuristischen Algorithmen möglich ist, da sie über ihre Kostenfunktion parametrisierbar sind.

Der Integrationstest wird in den kommenden Jahren und Jahrzehnten immer stärker an Bedeutung gewinnen, besonders durch die Verwendung neuer Technologien, die es erlauben, Bausteine lose miteinander zu verbinden, um hochgradig verteilte Softwaresysteme zu erstellen. Das Testen, ob diese Bausteine richtig miteinander interagieren, erlangt somit einen sehr hohen Stellenwert. Der Integrationstest wäre ein dafür geeigneter Test und die Inhalte dieser Arbeit tragen dazu bei, diesen besser zu verstehen und sie liefern Ansätze, wie Teilprobleme des Integrationstests bewältigt werden.

Abbildungsverzeichnis

Literaturverzeichnis

[Ab07] Abdi, H.: The Bonferonni and Šidák Corrections for Multiple Comparisons. In: Salkind, N. (Hrsg.): The encyclopedia of measurement and statistics. Sage Publications, 2007.

[BBB05] Badri, L.; Badri, M.; Ble, V.S.: A Method Level Based Approach for OO Integration Testing: An Experimental Study. Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks (SNPD-SAWN), Seiten 102-109, IEEE, 2005.

[BBM96] Basili, V.R.; Briand, L.C.; Melo, W.L.: A Validation of Object-Oriented Design Metrics as Qualità Indicators. IEEE Transactions on Software Engineering, Ausg. 22, Nr. 10, Seiten 751-761, 1996.

[BCI+00] Bertolino, A.; Corradini, F.; Inverardi, P.; Muccini, M.: Deriving Test Plans from Architectural Descriptions. 22nd International Conference on Software Engineering (ICSE), Seiten 220-229, ACM, 2000.

[BD04] Brügge, B.; Dutoit, A.H.: Object-Oriented Software Engineering – Using UML, Patterns and Java. Prentice Hall, 2004.

[Be90] Beizer, B.: Software Testing Techniques. International Thomson Computer Press, 1990.

[BEJ+08] Borner, L.; Ebrecht, L.; Jungmayr, S.; Hamburg, M.; Winter, M.: Suchen Sie die richtigen Fehler? – Warum es sich lohnt, Fehler zu kategorisieren. Objektspektrum, Ausg. 01/2008, Seiten 73-80, 2008.

[BFH+06] Borner, L.; Fraikin, F.; Hamburg, M.; Jungmayr, S.; Schönknecht, A.: Fehlerhäufigkeiten in objektorientierten Systemen: Basisauswertung einer Online-Umfrage. Technical Report SWEHD-TR-2006-01, 2006 http://www-swe.informatik.uni-heidelberg.de/research/publications/SWEHD_TR2006_01.pdf

[BFL02] Briand, L.C.; Feng, J.; Labiche, Y.: Using Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders. 14th International Conference on Software Engineering and Knowledge Engineering (SEKE), Seiten 43-50, ACM, 2002.

[Bi96] Binder, R.: Testing Object-Oriented Software: A Survey. Journal of Software Testing, Verification and Reliability, Ausg. 6, Nr. 3/4, Seiten 125-252, 1996.

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

[BIP07a] Borner, L.; Illes, T.; Paech, B.: Entscheidungen im Testprozess. In: Bleek, W.-G.; Raasch, J.; Züllighoven, H. (Hrsg.): Software Engineering 2007, LNI P-105, Seiten 247-248, Gesellschaft für Informatik, 2007.

[BIP07b] Borner, L.; Illes, T.; Paech, B.: The Testing Process - A Decision Based Approach. International Conference on Software Engineering Advances (ICSEA), Seite 41, IEEE, 2007.

[BLW01] Briand, L.C.; Labiche, Y.; Wang, Y.: Revisiting Strategies for Ordering Class Integration Testing in the presence of dependency cycles. 12th International Symposium on Software Reliability Engineering (ISSRE), Seiten 287-296, IEEE, 2001.

[BLW03] Briand, L.C.; Labiche, Y.; Wang, Y.: An Investigation of Graph-Based Class Integration Test Order. IEEE Transactions on Software Engineering, Ausg. 29, Nr. 7, Seiten 594-607, 2003.

[Bo79] Boehm, B. W.: Guidelines for Verifying and Validating Software Requirements and Design Specification. European Conference on Applied Information Technology (EÚRO IFIP), Seiten 711-719, North Holland, 1979.

[BP09a] Borner, L.; Paech, B.: Using Dependency Information to Select the Test Focus in the Integration Testing Process. Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART), Seiten 135-143, IEEE, 2009.

[BP09b] Borner, L.; Paech, B.: Integration Test Order Strategies to Consider Test Focus and Simulation Effort. First International Conference on Advances in System Testing and Validation Lifecycle (VALID), IEEE, 2009.

[BR84] Burkard, R.E.; Rendl, F.: A thermodynamically motivated simulation procedure for combinatorial optimization problems. European Journal of Operational Research, Ausg. 17, Nr. 2, Seiten 169-174, 1984.

[Br01] Browning, T.R.: Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions. IEEE Transactions on Engineering Management, Ausg. 48, Nr. 3, Seiten 292-306, 2001.

[BTS+03] Baudry, B.; Traon, Y.L.; Sunye, G.; Jézéquel, J.M.: Measuring and Improving Design patterns testability. 9th International Symposium on Software Metrics (METRICS), Seiten 50-59, IEEE, 2003.

[Bu03] Budgen, D.: Software Design. Addison-Wesley, 2003.

[CL04] Chen, Q.; Li, X.: An Order-Assigned Strategy of Classes Integration Testing Based on Test Level. 8th International Conference on Computer Supported Cooperative Work in Design (CSCWD), Seiten 653-657, IEEE, 2004.

[Co71] Constam, M.: FORTRAN für Anfänger, Springer Verlag, 1971.

[CR05] Chakraborty, K.; Remington, W.: "Offshoring" of IT services: the impact on the US economy. Journal of Computing Sciences in Colleges, Ausg. 20, Nr. 4, Seiten 112-125, 2005.

[Di02] Diekmann, A.: Empirische Sozialforschung – Grundlagen, Methoden, Anwendungen. Rowohlt Taschenbuch Verlag, 2002.

[ER96] Eickelmann, N.S.; Richardson, D.J.: What Makes One Software Architecture More Testable Than Another? Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops, Seiten 65-67, ACM, 1996.

[FG99] Fewster, M.; Graham, D.: Software Test Automation. Addison-Wesley, 1999.

[Fi92] Firesmith, D.G.: Object-oriented software. Technical Report, Ossian, Ind.: Advanced Technology Specialists, 1992.

[FO00] Fenton, N.E.; Ohlsson, N.: Quantitative analysis of faults and failures in a complex software system. IEEE Transactions on Software Engineering, Ausg. 26, Nr. 8, Seiten 797-814, 2000.

[GHJ+01] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley, 2001.

[GSW03] Gao, J.Z.; Tsao, H.S.J.; Wu, Y.: Testing and Quality Assurance for Component-Based Software. Artech House Publishers, 2003.

[HM92] Harrold, M. J.; McGregor, J.: Incremental Testing of Object-Oriented Class Structures. International Conference on Software Engineering (ICSE), Seiten 68-80, ACM, 1992.

[HR02] Horn, E.; Reinke, T.: Softwarearchitektur und Softwarebauelemente – Einführung für Softwarearchitekten. Hanser Verlag, 2002.

[IP08a] Illes-Seifert, T.; Paech, B.: Exploring the relationship of a file's history and its fault-proneness: An empirical study. Testing: Academic & Industrial Conference - Practice and Research Techniques (TAIC-PART), Seiten 13-22, IEEE, 2008.

[IP08b] Illes-Seifert, T.; Paech, B.: Exploring the relationship of history characteristics and defect count: an empirical study. 2008 Workshop on Defects in Large Software Systems (DEFECTS) held in conjunction with ISSTA, Seiten 11-15, ACM, 2008.

[IP09] Illes-Seifert, T.; Paech, B.: The vital few and trivial many: An empirical analysis of the Pareto Distribution of defects. Software Engineering 2009, LNI P-143, Seiten 151-162, Gesellschaft für Informatik, 2009.

[IHP+05] Illes, T.; Herrmann, A.; Paech, B.; Rückert, J.: Criteria for Software Testing Tool Evaluation. A Task Oriented View. 3rd World Congress for Software Quality, Seiten 213-222, 2005.

[IPR+06] Illes, T.; Pohlmann, H.; Roßner, T.; Schlatter, A.; Winter, M.: Software-Testmanagement Planung, Design, Durchführung und Auswertung von Tests - Methodenbericht und Analyse unterstützender Werkzeuge. IX Studie 01/2006 zum Thema "Software-Testmanagement", Heise Zeitschriften Verlag, 2006.

[JCM+08] Jiang, Y.; Cukic, B.; Menzies, T.; Bartlow, N.: Comparing Design and Code Metrics for Software Quality Prediction. 4th International Workshop on Predictor Models in Software Engineering (PROMISE), Seiten 11-18, ACM, 2008.

[JL03] Janssen, J.; Laatz, W.: Statistische Datenanalyse mit SPSS für Windows, Springer Verlag, 2003.

[Ka62] Kahn, A.B.: Topological sorting of large networks. Communications of the ACM, Ausg. 5, Nr. 11, Seiten 558-562, 1962.

[KFN99] Kaner, C.; Falk, J.; Nguyen, H.Q.: Testing Computer Software. John Wiley & Sons, 1999.

[KGH+95a] Kung, D.; Gao, J.; Hsia, P.; Lin, J.; Toyoshima, Y.: Class Firewall, test order, and regression testing of object-oriented programs. Journal of Object-Oriented Programming, Ausg. 8, Vo. 2, Seiten 51-65, 1995.

[KGH+95b] Kung, D.; Gao, J.; Hsia, P.; Toyoshima, Y.; Chen, C.; Kim, Y.-S.; Song, Y.-K.: Developing an Object-oriented Software Testing and Maintenance Environment. Communications of the ACM, Ausg. 38, Nr. 10, Seiten 75-87, 1995.

[KPB06] Knab, P.; Pinyger, M.; Bernstein, A.: Predicting Defect Densities in Source code Files with Decision Tree Learners. International Workshop on Mining Software Repositories (MSR), Seiten 119-125, ACM, 2006.

[Li90] Liggesmeyer, P.: Modultest und Modulverifikation - State of the Art. BI Wissenschaftsverlag, 1990.

[LTW+00] Labiche, Y.; Thévenod-Fosse, P.; Waeselynck, H.; Durand, M.: Testing Levels for Object-Oriented Software. 22nd International Conference on Software Engineering (ICSE), Seiten 136-145, ACM, 2000.

[Ma03] Mariani, L.: A Fault Taxonomy for Component-based Software. Electronic Notes in Theoretical Computer Science, Ausg. 82, Nr. 6, 2003.

[Mc76] McCabe, T.J.: A Complexity Measure. IEEE Transactions on Software Engineering, Ausg. SE-2, Seiten 308-320, 1976.

[MYB+91] MacLean, A.; Young, R.M.; Bellotti, V.; Moran, T.: Questions, options, and criteria: Elements of design space analysis. Human-Computer Interaction, Ausg. 6, Nr. 3, Seiten 201-250, 1991.

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

[MK92] Munson, J.C.; Khoshgoftaar, T.M.: The detection of fault-prone programs. IEEE Transactions on Software Engineering, Ausg. 18, Nr. 5, Seiten 423-433, 1992.

[MMS+06] Merdes, M.; Malaka, R.; Suliman, D.; Paech, B.; Brenner, D.; Atkinson, C.: Ubiquitous RATs: How Resource-Aware Run-Time Tests Can Improve Ubiquitous Software System. 6th International Workshop on Software Engineering and Middleware (SEM), Seiten 55-62, ACM, 2006.

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

[NB05] Nagappan, N.; Ball, T.: Static Analysis Tools as Early Indicators of Pre-Release Defect Density. 27th International Conference on Software Engineering (ICSE), Seiten 580-586, ACM, 2008.

[NBZ06] Nagappan, N.; Ball, T.; Zeller, A.: Mining metrics to predicts component failures. 28th International Conference on Software Engineering (ICSE), Seiten 452-461, ACM, 2006.

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

[Ov94] Overbeck, J.: Integration Testing for Object-Oriented Software. PhD Thesis, 1994.

[OW07] Ostrand, T.J.; Weyuker, E.J.: How to Measure Success of Fault Prediction Models. Fourth International Workshop on Software Quality Assurance (SOQUA), Seiten 25-30, ACM, 2007.

[PC89] Podgurski, A.; Clarke, L.A.: The Implications of Program Dependences for Software Testing, Debugging, and Maintenance. ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification, Seiten 68-178, ACM, 1989.

[PKS00] Pol, M.; Koomen, T.; Spillner, A.; Management und Optimierung des Testprozesses – Ein praktischer Leitfaden für erfolgreiches Testen von Software mit TPI und TMAP. Addison-Wesley, 2000.

[RHQ+05] Rupp, C.; Hahn, J.; Queins, S.; Jeckle, M.; Zengler, B.: UML 2 glasklar. Hanser-Verlag, 2005.

[RSG08] Ratzinger, J.; Sigmund, T.; Gall, H.C.: On the Relation of Refactoring and Software Defects. International Working Conference on Mining Software Repositories (MSR), Seiten 35-38, ACM, 2008.

[Sa07] Savernik, L.: Anatomie zyklischer Abhängigkeiten in Softwaresystemen. Technischer Bericht 1/2007, Institut für Systemsoftware, Johannes Kepler Universität Linz, 2007.

[SBM+06] Suliman, D.; Borner, L.; Merdes, M.; Brenner, D.: Laufzeittest mobiler und komponentenorientierter Software. In: Haasis, K.; Heinzl, A.; Klumpp, D. (Hrsg.): Aktuelle Trends in der Softwareforschung. Tagungsband zum doIT Forschungstag am 13. Juli 2006, Mannheim, Seiten 35-46, dpunkt.verlag GmbH, 2006.

[SBS08] Sneed, H.; Baumgartner, M.; Seidl, R.: Der Systemtest. Anforderungsbasiertes Testen von Software-Systemen. Hanser Fachbuchverlag, 2008.

[SJS+05] Sangal, N.; Jordan, E.; Sinha, V.; Jackson, D.: Using Dependency Models to Manage Complex Software Architecture. 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), Seiten 167-176, ACM, 2005.

[SK03] Subramanyam, R.; Krishnan, M. S.: Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. IEEE Transactions on Software Engineering, Ausg. 29, Nr. 4, Seiten 297-310, 2003.

[SL05] Spillner, A.; Linz, T.: Basiswissen Softwaretest – Aus- und Weiterbildung zum Certified Tester. dpunkt.verlag GmbH, 2005.

[So07] Sommerville, I.: Software Engineering. Addison-Wesley, 2007.

[Sp00] Spillner, A.: From V-Model to W-Model – Establishing the Whole Test Process. Conquest 2000 – 4th Conference on Quality Engineering in Software Technology, Seiten 222-231, 2000.

[Sp90] Spillner, A.: Dynamischer Integrationstest modularer Softwaresysteme. Dissertation, 1990.

[Sp01] Spillner, A.: Das W-Modell: Testen als paralleler Prozess zum Software-Entwicklungsprozess. Softwaretechnik-Trends, Ausg. 21, Nr. 1, Seiten 4-5, 2001.

[Sp02] Spillner, A.: The W-MODEL – Strengthening the Bond Between Development and Test. International Conference on Software Testing, Analysis & Review (STAReast), 2002.

[SS05] Saake, G.; Sattler, K.: Algorithmen und Datenstrukturen, dpunkt.verlag GmbH, 2005.

[SPB+06] Suliman, D.; Paech, B.; Borner, L.; Atkinson, C.; Brenner, D.; Merdes, M.; Malaka, R.: The MORABIT Approach to Runtime Component Testing. 30th Annual International Computer Software and Applications Conference (COMPSAC), Seiten 171-176, IEEE, 2006.

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

[Ta72] Tarjan, R.: Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing, Ausg. 1, Nr. 2, Seiten 146-160, 1972.

[Ta02] Tassey, G.: The Economic Impacts of Inadequate Infrastructure for Software Testing. Final Report Mai 2002, Research Triangle Institute, 2002. http://www.nist.gov/director/prog-ofc/report02-3.pdf

[TD97] Tai, K.; Daniels, F.: Test Order for Inter-Class Integrations Testing of Object-Oriented Software. 21st International Computer Software and Applications Conference (COMSAC), Seiten 602-607, IEEE, 1997.

[TJJ+00] Le Traon, Y. L.; Jéron, T.; Jézéquel, J.M.; Morel, P.: Efficient Object-Oriented Integration and Regression Testing. IEEE Transactions on Reliability, Ausg. 49, Nr. 1, Seiten 12-25, 2000.

[VR02] Vieira, M.; Richardson, V.: The Role of Dependencies in Component-based Systems Evolution. International Workshop on Principles of Software Evolution (IWPSE), Seiten 62-65, ACM, 2002.

[Wi98] Winter, M.: Managing Object-Oriented Integration and Regression Testing (without becoming drowned). Conference on Software Testing Analysis and Review (EuroSTAR), 1998.

[WC03] Wheeldon, R.; Counsell, S.: Power Law Distributions in Class Relationships. Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), Seiten 45-54, IEEE, 2003.

[WPC01] Wu, Y.; Pan, D.; Chen, M.-H.: Techniques for Testing Component-based Software. Seventh International Conference on Engineering of Complex Computer Systems (ICECCS), Seiten 222-232, IEEE, 2001.

[Zh08] Zhang, H.: An Initial Study of the Growth of Eclipse Defects. 2008 International Working Conference on Mining Software Repositories (MSR), Seiten 141-144, ACM, 2008.

[Zi06] Zimmermann, T.: Taking Lessons from History. 28th International Conference on Software Engineering (ICSE), Seiten 1001-1005, ACM, 2006.

[ZK04] Zeller, A.; Krinke, J.: Open-Source-Programmierwerkzeuge. dpunkt.verlag GmbH, 2004.

[ZN08] Zimmermann, T.; Nagappan, N.: Predicting defects using network analysis on dependency graphs. 30th International Conference on Software Engineering (ICSE), Seiten 531-540, ACM, 2008.

[ZPZ07] Zimmermann, T.; Premraj, R.; Zeller, A.: Predicting Defects for Eclipse. Third International Workshop on Predictor Models in Software Engineering (PROMISE), Seite 9, IEEE, 2007.

Web-Seiten

[ANT09] Apache ANT, http://ant.apache.org/, 16.09.2009

[CDK09] Chemistry Development Kit, http://sourceforge.net/projects/cdk/, 16.09.2009

[CV09] CVS, http://www.nongnu.org/cvs/, 16.09.2009

[Ec09] Eclipse, www.eclipse.org, 16.09.2009

[Ex09] Microsoft Excel, www.microsoft.com/excel/, 16.09.2009

[FOP09] Apache FOP, http://xmlgraphics.apache.org/fop/index.html, 16.09.2009

[FRE09] Free Network Project, http://freenetproject.org/whatis.html, 16.09.2009

[Ja09] Java, http://java.sun.com/, 16.09.2009

[JET09] Jetspeed 2, http://portals.apache.org/jetspeed-2/, 16.09.2009

[JMO09] JMol, http://jmol.sourceforge.net/, 16.09.2009

[JRu09] Jruby, http://jruby.codehaus.org/, 16.09.2009

[LN09] Lotus Notes, http://www-01.ibm.com/software/de/lotus/, 16.09.2009

[My09] MySQL, http://www.mysql.com/, 16.09.2009

[OSC09] OSCache, http://www.opensymphony.com/oscache/, 2009

[Po09] PostgrSQL, http://www.postgresql.org/, 16.09.2009

[ROR09] Ruby on Rails, http://rubyonrails.org/, 16.09.2009

[Ru09] Ruby, http://www.ruby-lang.org/, 16.09.2009

[Se09] Servlets, http://java.sun.com/products/servlet/, 16.09.2009

[Si09] Sissy, http://sissy.fzi.de/, 16.09.2009

[SP09] SPSS, http://www.spss.com/, 16.09.2009

[SR09] SVNReport, http://vcsreport.sourceforge.net/, 16.09.2009

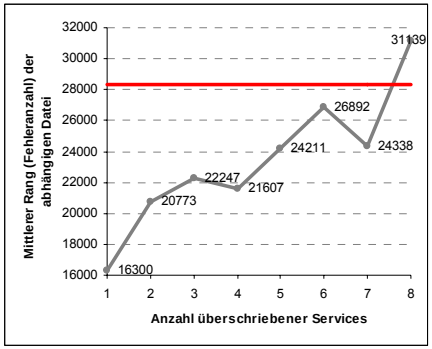
[SVK09] Subversion Kit, http://svnkit.com/, 2009

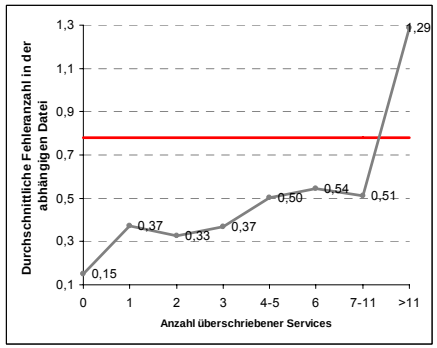
[Su09] Subversion, http://subversion.tigris.org/, 2009

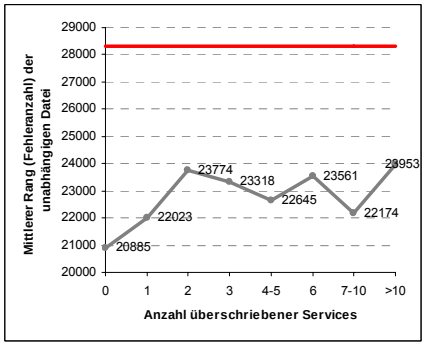
[Tom09] Apache Tomcat, http://tomcat.apache.org/, 16.09.2009
[Too09] Arbeitskreis „Testen objektorientierter Programme“ der Gesellschaft für Informatik, http://www1.gi-ev.de/fachbereiche/softwaretechnik/tav/toop/, 16.09.2009
[TVB09] TVBrowser, http://www.tvbrowser.org/, 16.09.2009
[UML09] UML, http://www.uml.org/, 16.09.2009

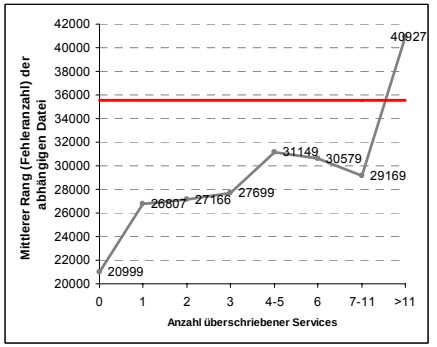
Anhang A: Statistiken Eclipse

Abhängige Datei		Unabhängige Datei	
Eclipse 2.0	Eclipse 2.1	Eclipse 2.0	Eclipse 2.1



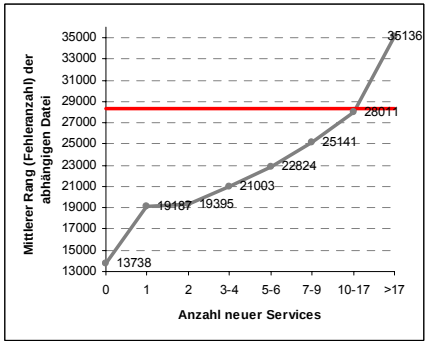


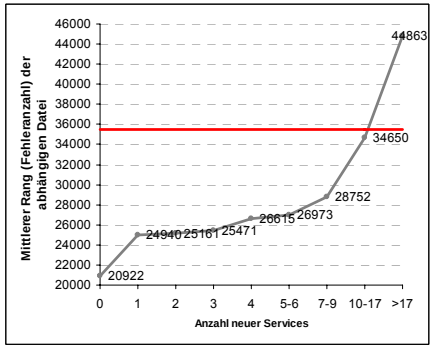


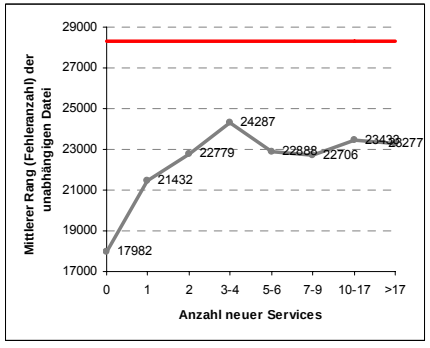


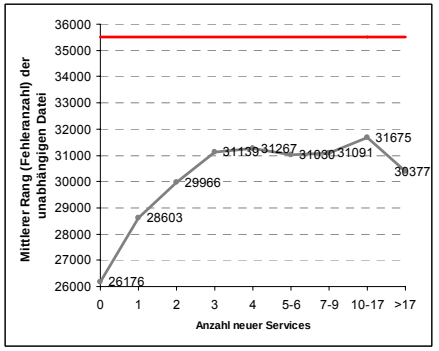
Anzahl überschriebener Services

Abhängige Datei		Unabhängige Datei	
Eclipse 2.0	Eclipse 2.1	Eclipse 2.0	Eclipse 2.1

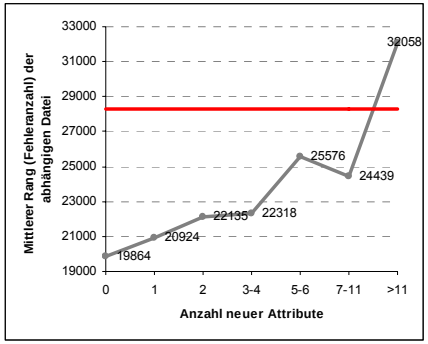
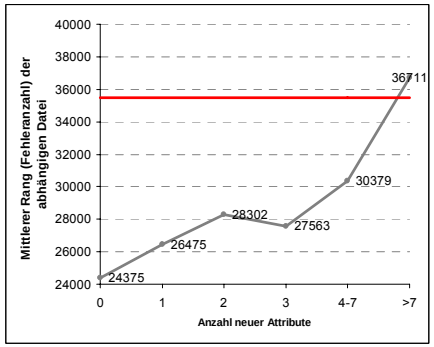
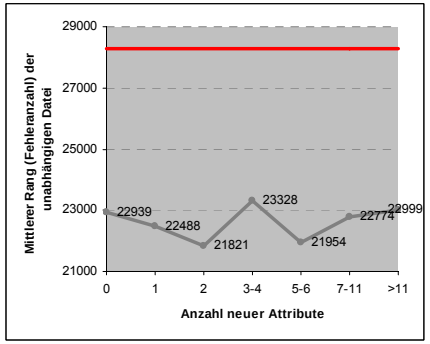
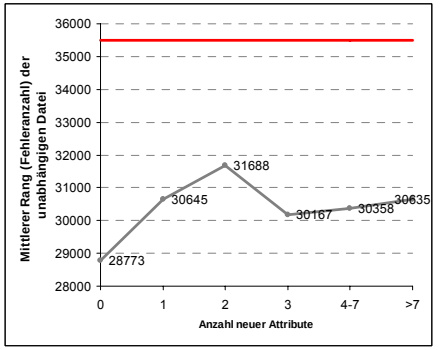




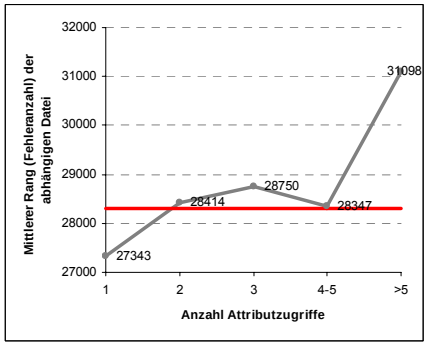
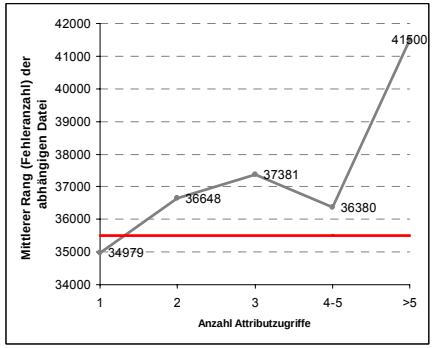
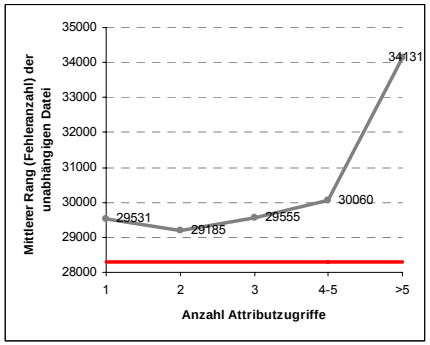
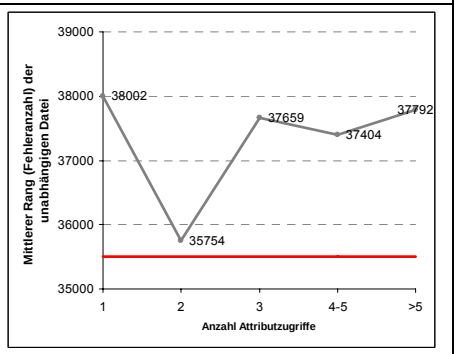




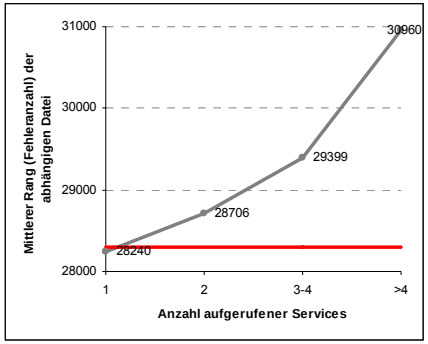
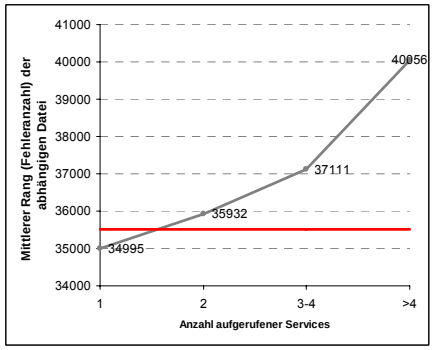
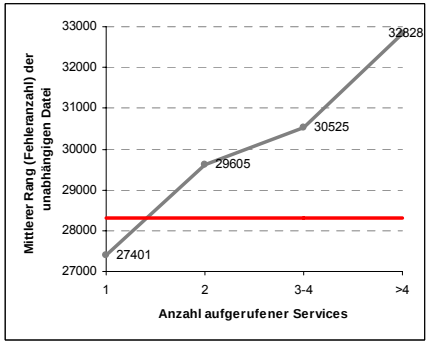
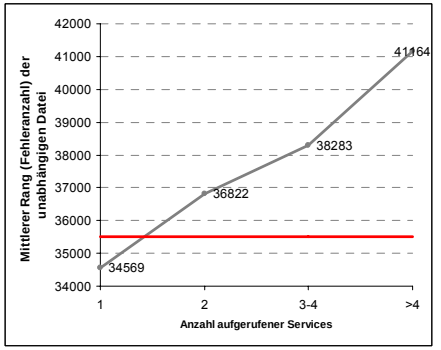
Anzahl neuer Services

Abhängige Datei		Unabhängige Datei	
Eclipse 2.0	**Eclipse 2.1**	**Eclipse 2.0**	**Eclipse 2.1**
			

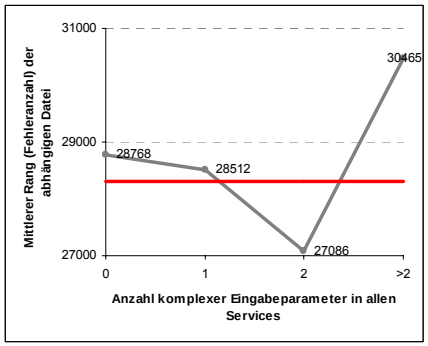
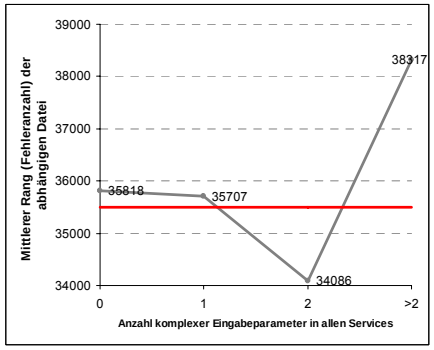
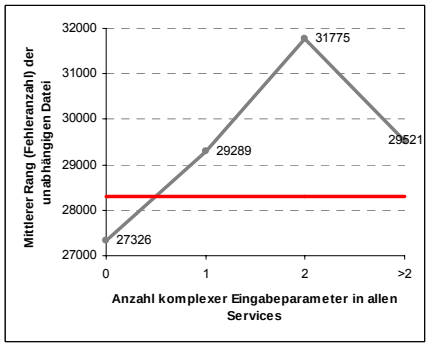
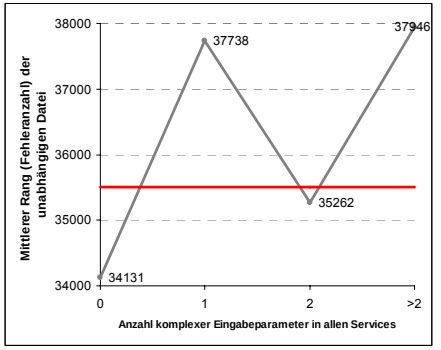
Anzahl neuer Attribute

Abhängige Datei		Unabhängige Datei	
Eclipse 2.0	**Eclipse 2.1**	**Eclipse 2.0**	**Eclipse 2.1**
			

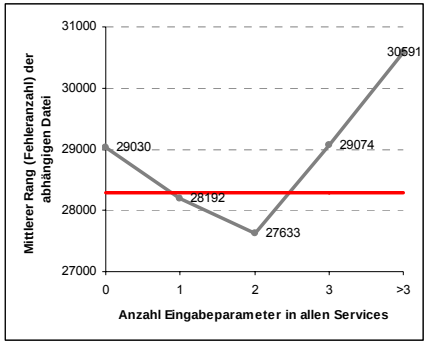
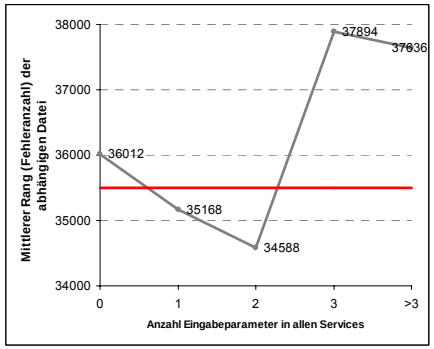
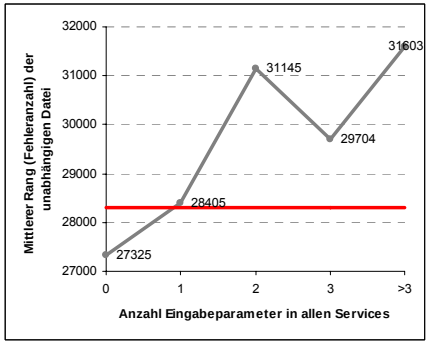
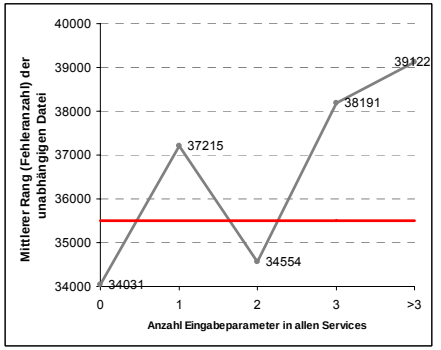
Anzahl Attributzugriffe

Abhängige Datei		Unabhängige Datei	
Eclipse 2.0	**Eclipse 2.1**	**Eclipse 2.0**	**Eclipse 2.1**
			

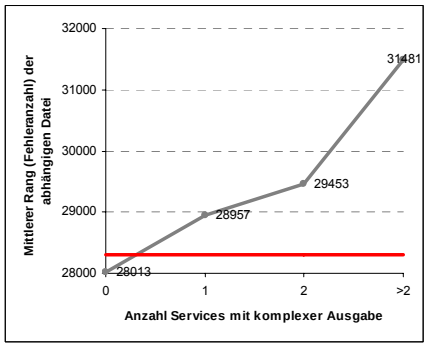
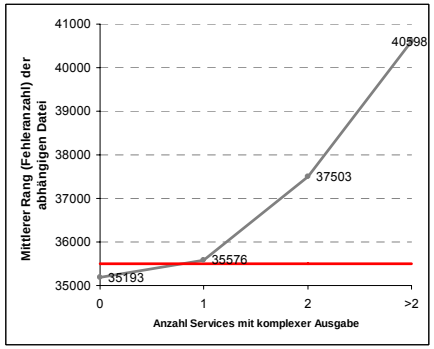
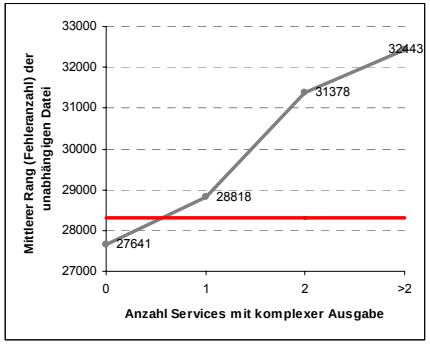
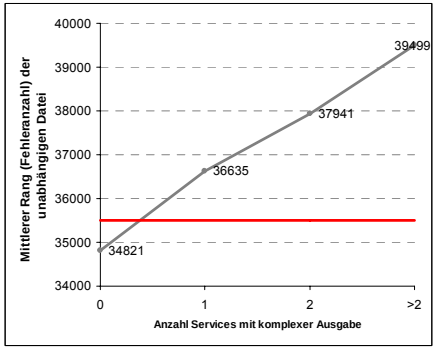
Anzahl Serviceaufrufe

Abhängige Datei		Unabhängige Datei	
Eclipse 2.0	**Eclipse 2.1**	**Eclipse 2.0**	**Eclipse 2.1**
			

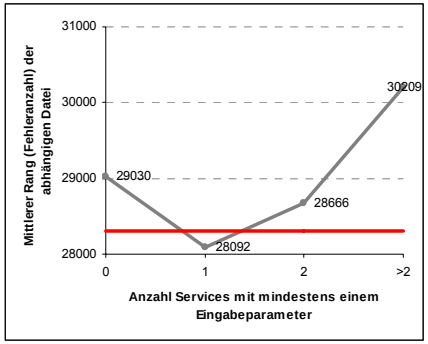
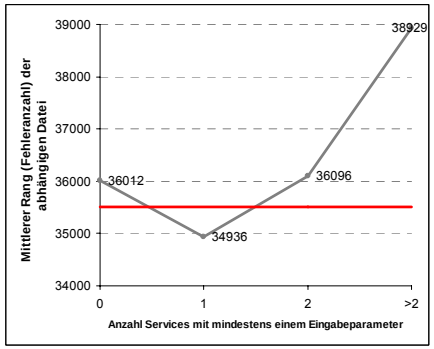
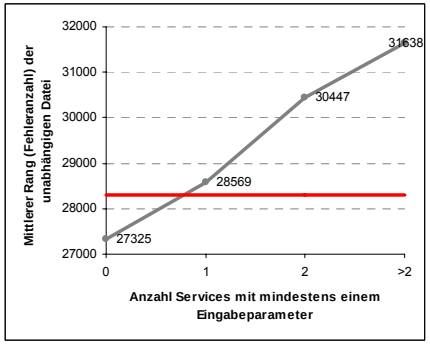
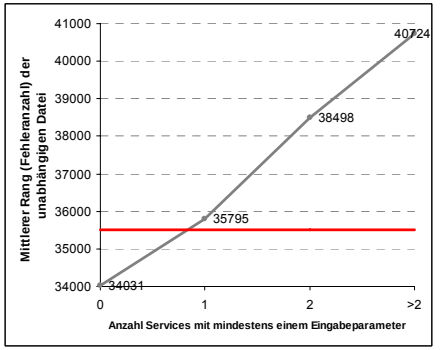
Anzahl komplexer Eingabeparameter in allen Services

Abhängige Datei		Unabhängige Datei	
Eclipse 2.0	**Eclipse 2.1**	**Eclipse 2.0**	**Eclipse 2.1**
			

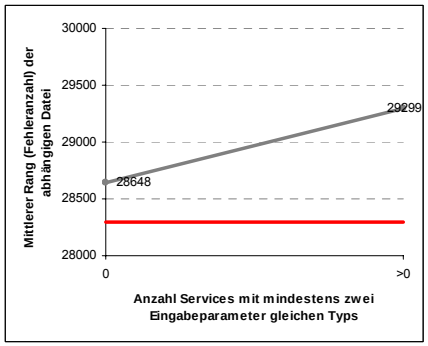
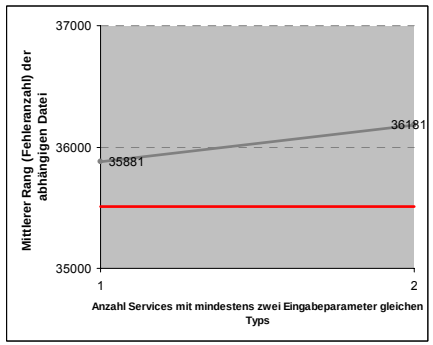
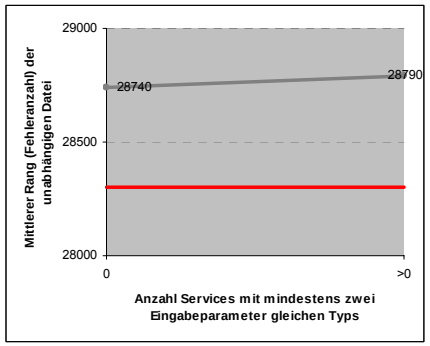
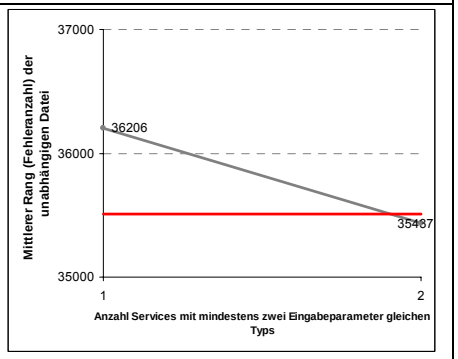
Anzahl Eingabeparameter in allen Services

Abhängige Datei		Unabhängige Datei	
Eclipse 2.0	**Eclipse 2.1**	**Eclipse 2.0**	**Eclipse 2.1**
			

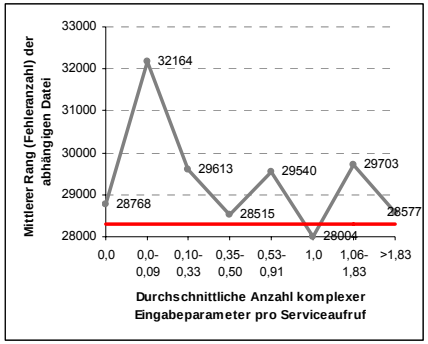
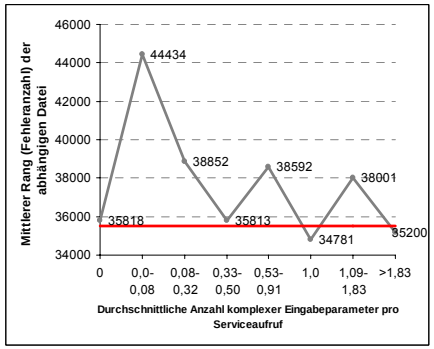
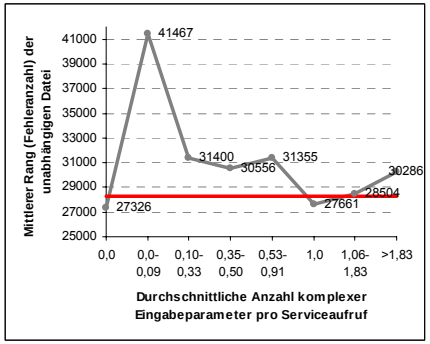
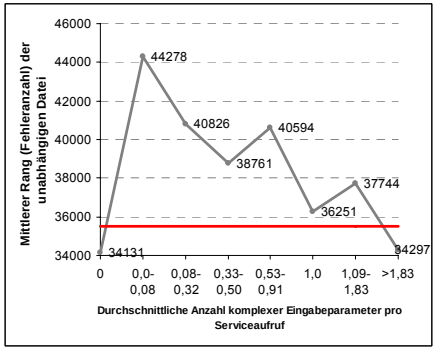
Anzahl Services mit komplexer Ausgabe

Abhängige Datei		Unabhängige Datei	
Eclipse 2.0	**Eclipse 2.1**	**Eclipse 2.0**	**Eclipse 2.1**
			

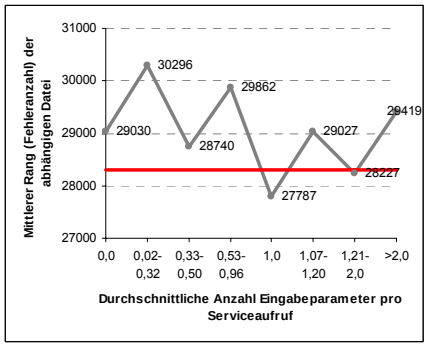
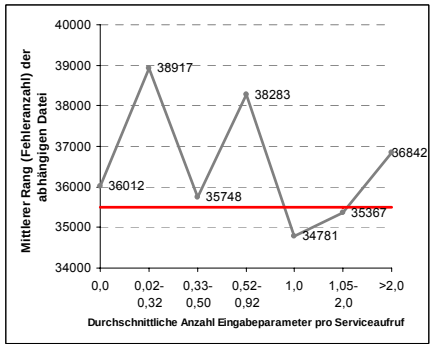
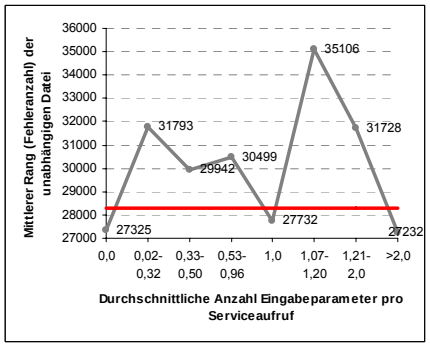
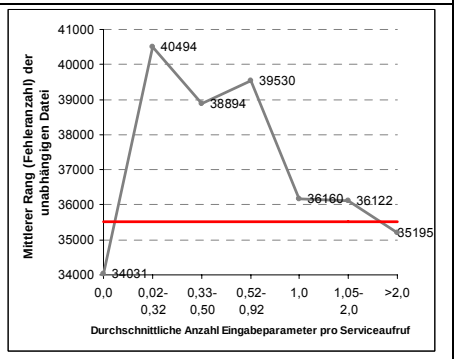
Anzahl Services mit mind. einem Eingabeparameter

Abhängige Datei		Unabhängige Datei	
Eclipse 2.0	**Eclipse 2.1**	**Eclipse 2.0**	**Eclipse 2.1**
			

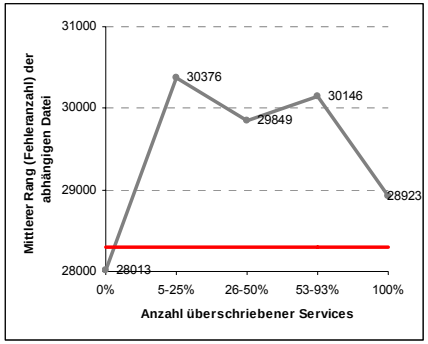
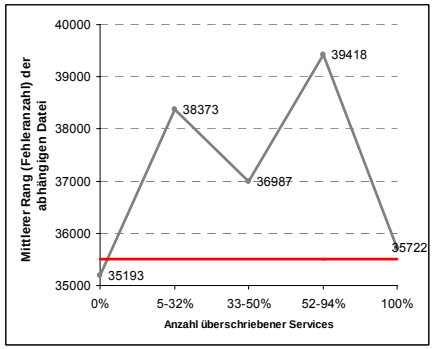
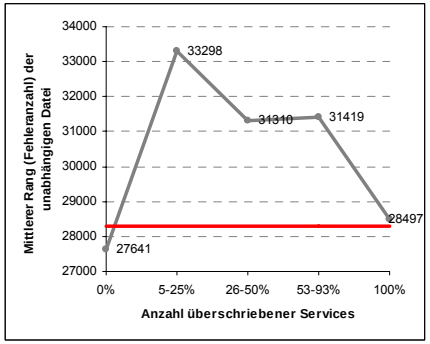
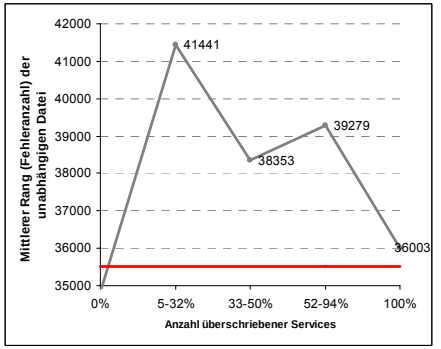
Anzahl Services mit mindestens zwei Eingabeparameter gleichen Typs

Abhängige Datei		Unabhängige Datei	
Eclipse 2.0	**Eclipse 2.1**	**Eclipse 2.0**	**Eclipse 2.1**
			

Durchschnittliche Anzahl komplexer Eingabeparameter pro Serviceaufruf

Abhängige Datei		Unabhängige Datei	
Eclipse 2.0	**Eclipse 2.1**	**Eclipse 2.0**	**Eclipse 2.1**
			

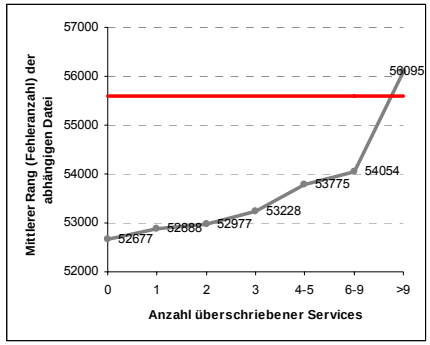
Durchschnittliche Anzahl Eingabeparameter pro Serviceaufruf

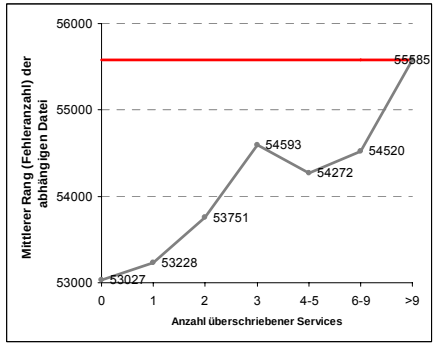
Abhängige Datei		Unabhängige Datei	
Eclipse 2.0	**Eclipse 2.1**	**Eclipse 2.0**	**Eclipse 2.1**
			

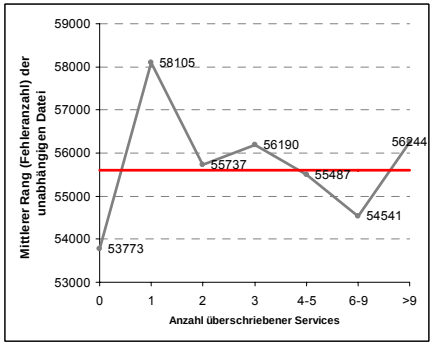
Relative Häufigkeit Services mit komplexer Ausgabe

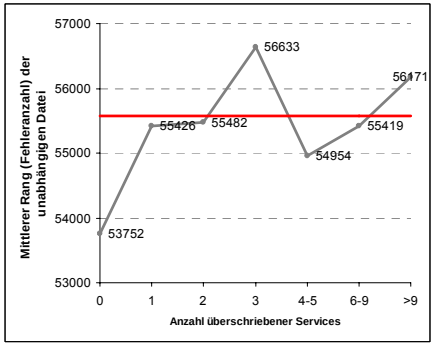
Anhang B: Statistiken Werkzeug zur Fördergeldverwaltung

Abhängige Datei		Unabhängige Datei	
01.07.2008	31.07.2008	01.07.2008	31.07.2008



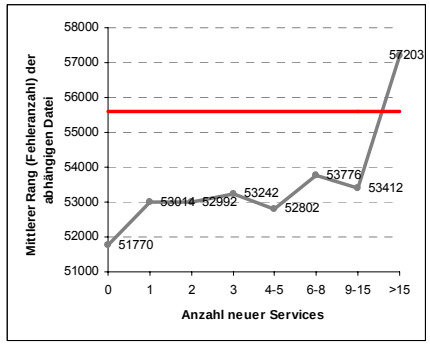


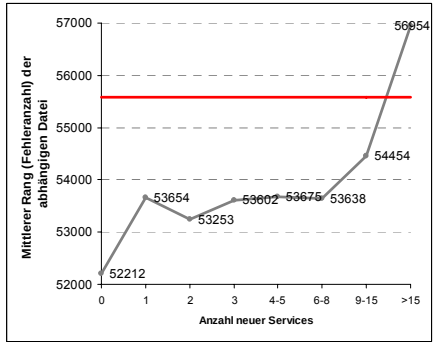


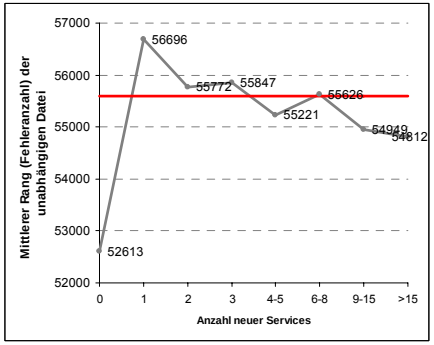


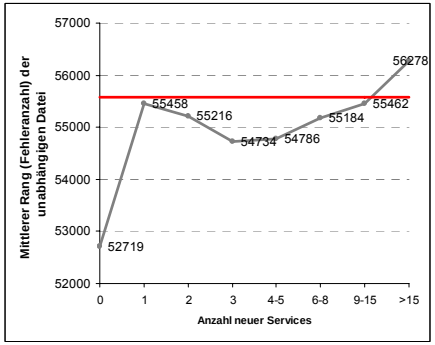
Anzahl überschriebener Services

Abhängige Datei		Unabhängige Datei	
01.07.2008	31.07.2008	01.07.2008	31.07.2008



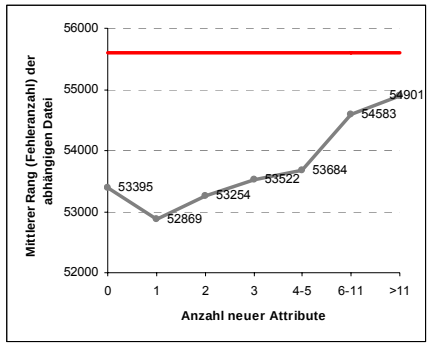


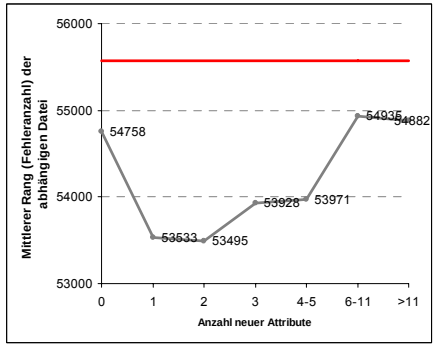


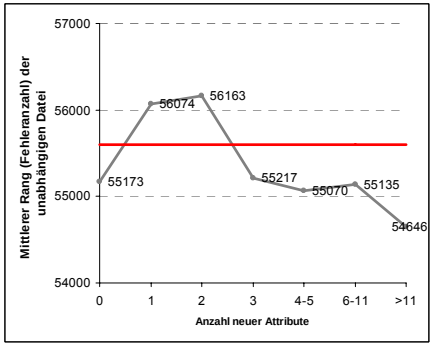


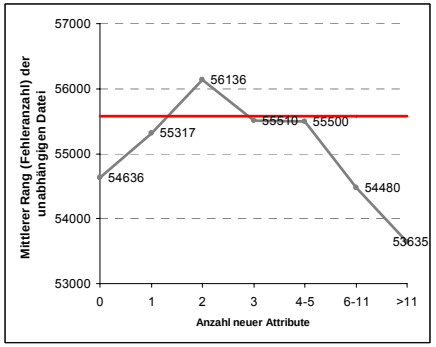
Anzahl neuer Services

Abhängige Datei		Unabhängige Datei	
01.07.2008	31.07.2008	01.07.2008	31.07.2008



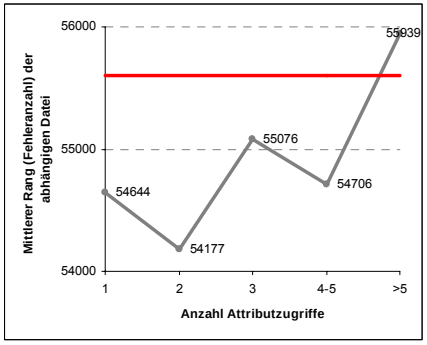


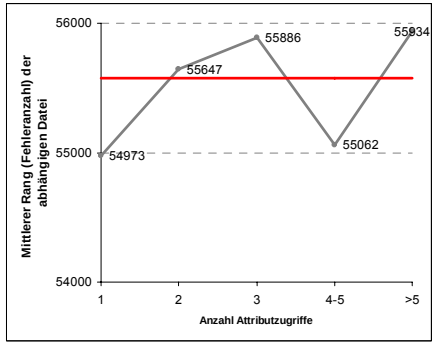


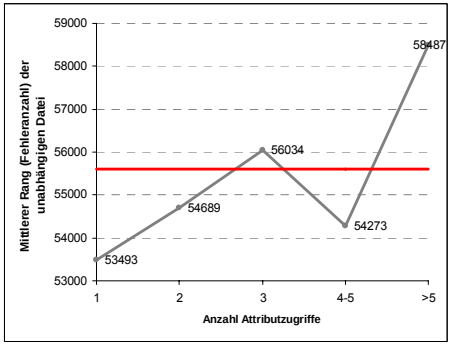


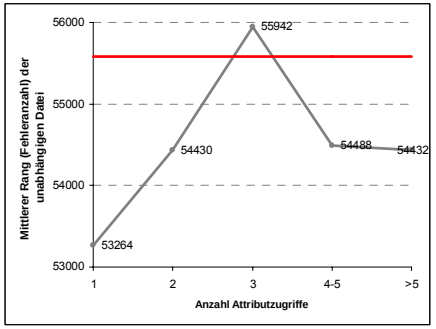
Anzahl neuer Attribute

Abhängige Datei		Unabhängige Datei	
01.07.2008	31.07.2008	01.07.2008	31.07.2008



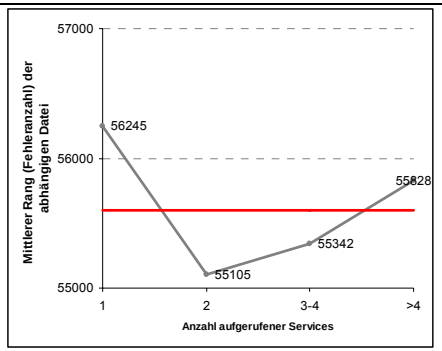


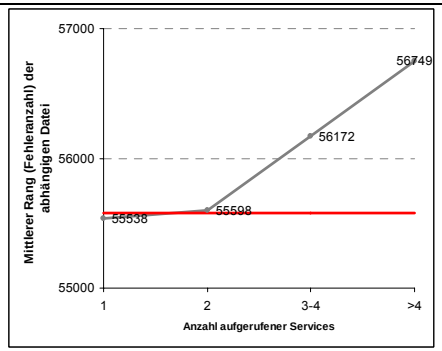


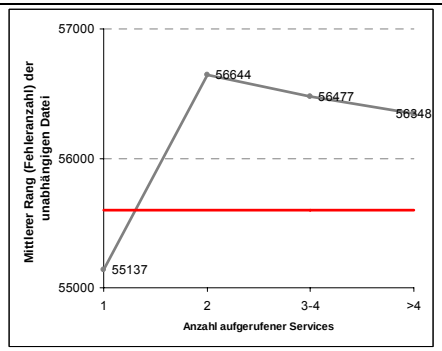


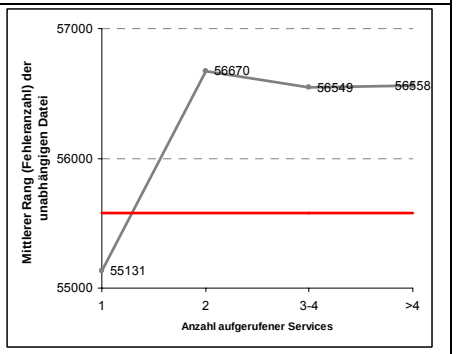
Anzahl Attributzugriffe

Abhängige Datei		Unabhängige Datei	
01.07.2008	31.07.2008	01.07.2008	31.07.2008



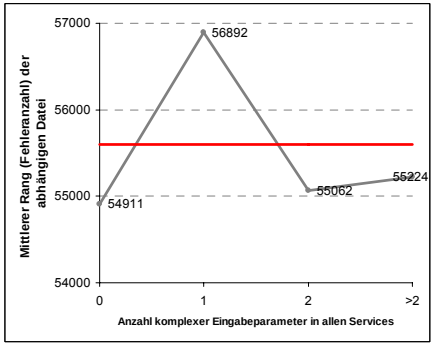


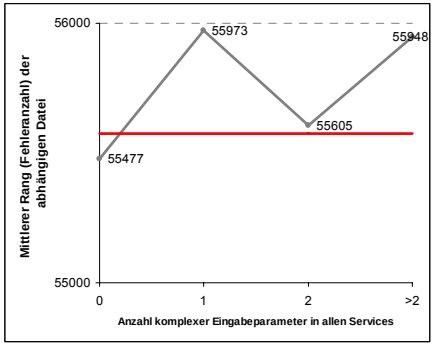


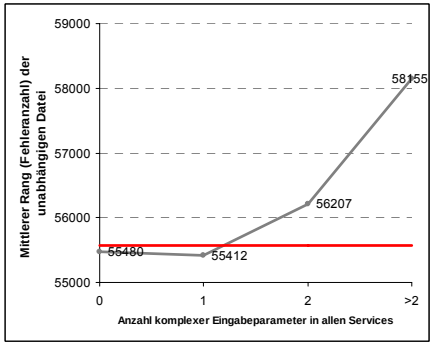


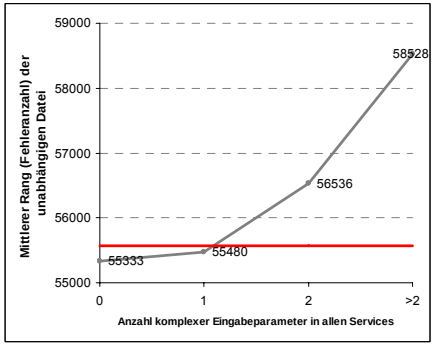
Anzahl Serviceaufrufe

Abhängige Datei		Unabhängige Datei	
01.07.2008	31.07.2008	01.07.2008	31.07.2008



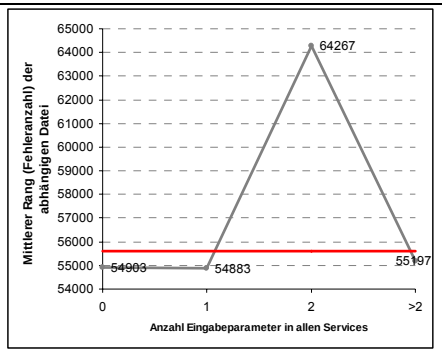


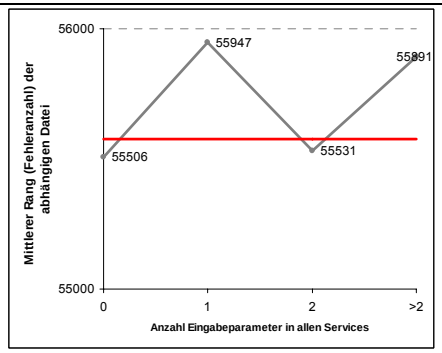


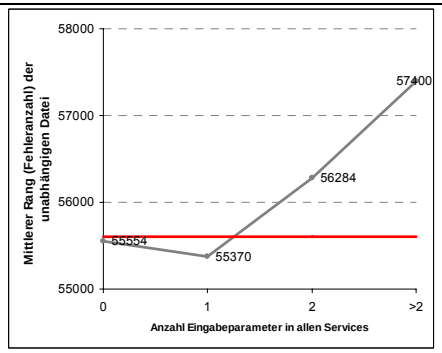


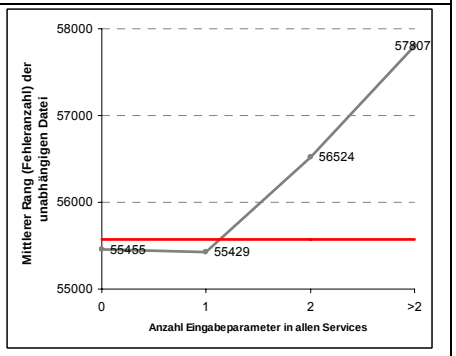
Anzahl komplexer Eingabeparameter in allen Services

Abhängige Datei		Unabhängige Datei	
01.07.2008	31.07.2008	01.07.2008	31.07.2008



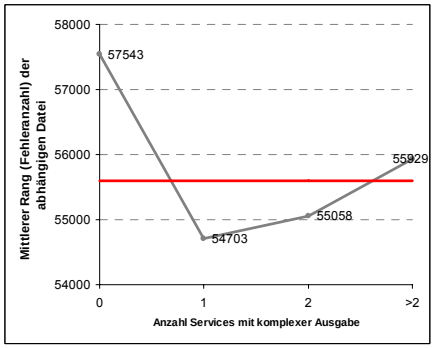


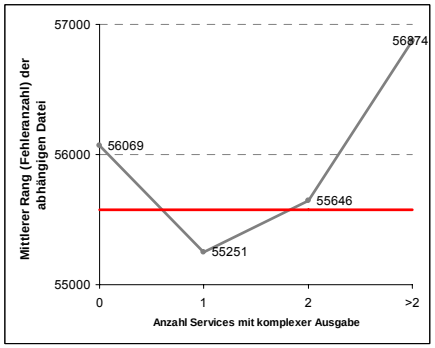


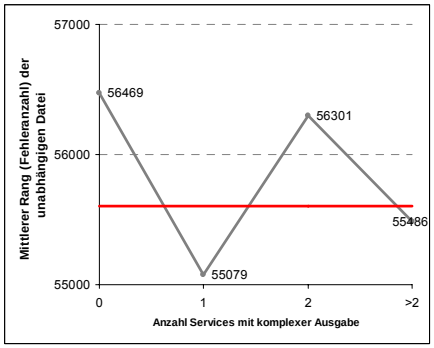


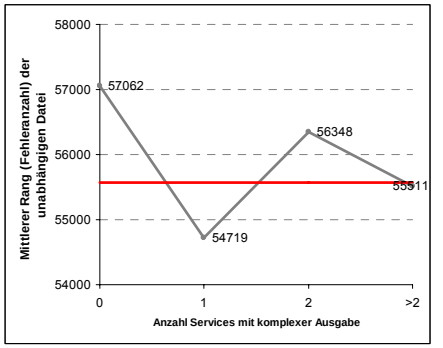
Anzahl Eingabeparameter in allen Services

Abhängige Datei		Unabhängige Datei	
01.07.2008	31.07.2008	01.07.2008	31.07.2008



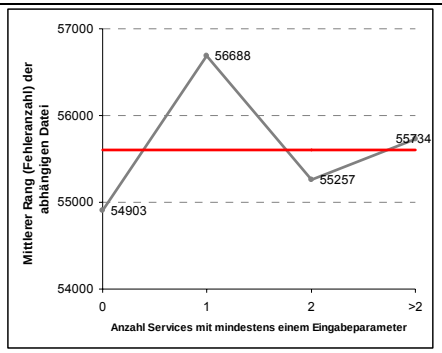


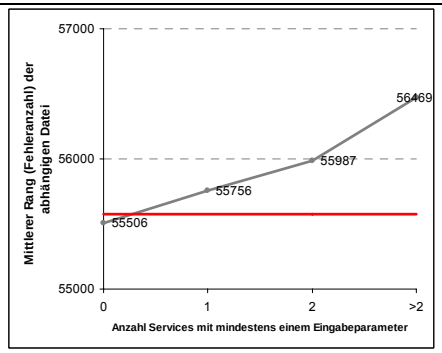


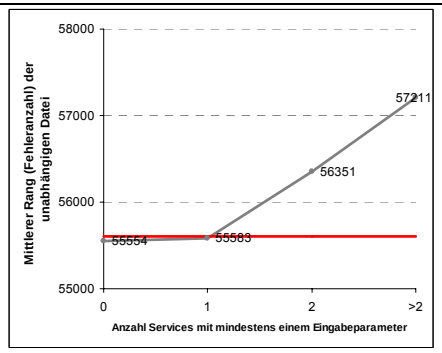


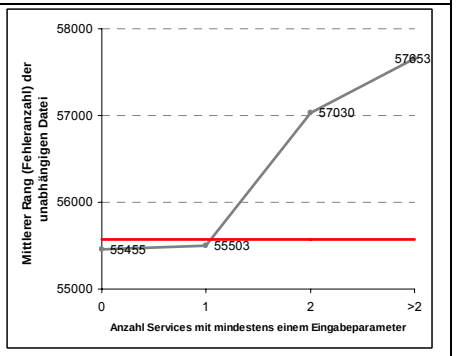
Anzahl Services mit komplexer Ausgabe

Abhängige Datei		Unabhängige Datei	
01.07.2008	31.07.2008	01.07.2008	31.07.2008



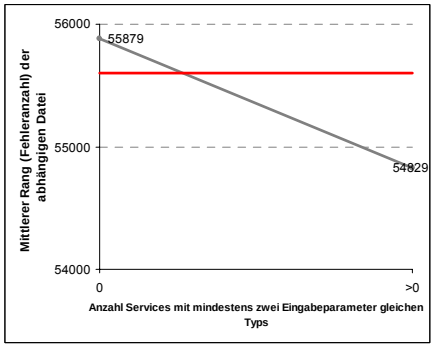


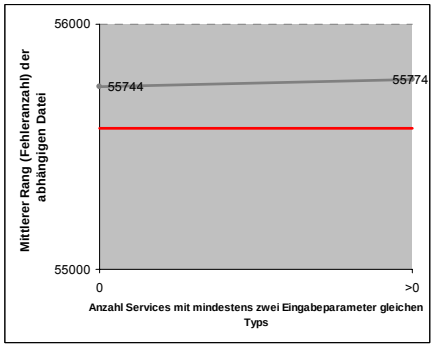


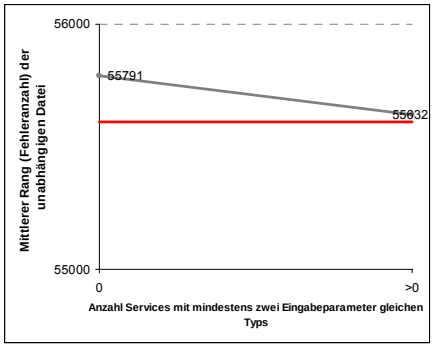


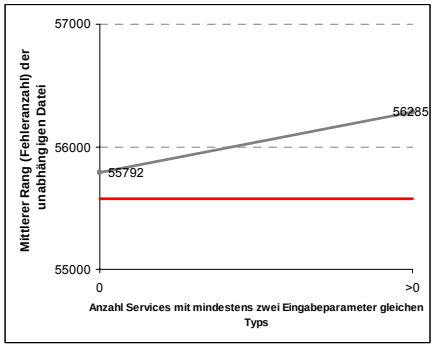
Anzahl Services mit mind. einem Eingabeparameter

Abhängige Datei		Unabhängige Datei	
01.07.2008	31.07.2008	01.07.2008	31.07.2008



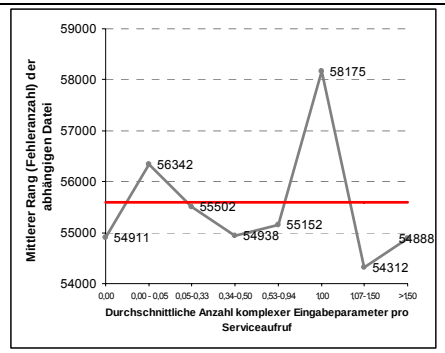


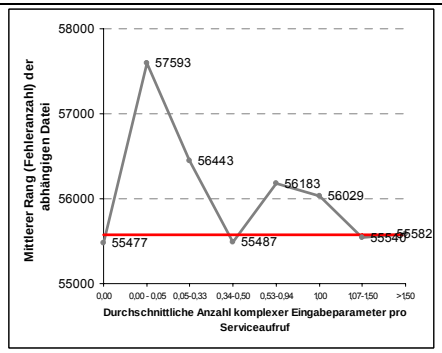


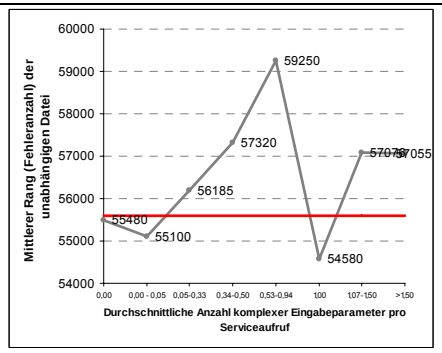


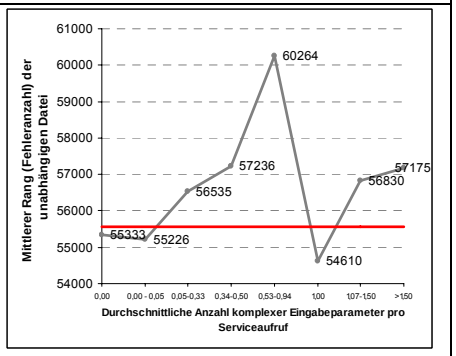
Anzahl Services mit mindestens zwei Eingabeparameter gleichen Typs

Abhängige Datei		Unabhängige Datei	
01.07.2008	31.07.2008	01.07.2008	31.07.2008



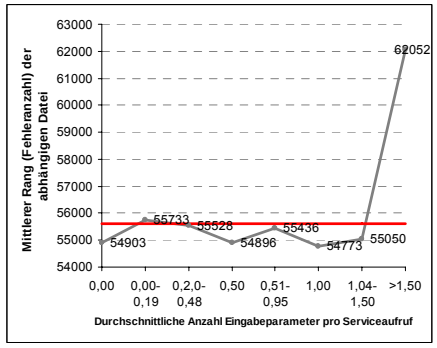


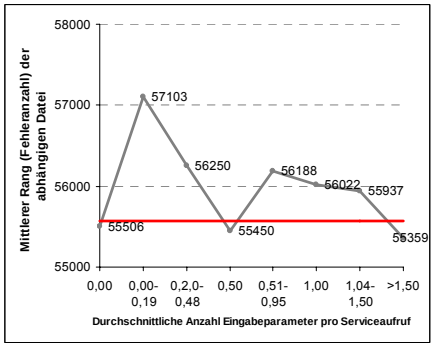


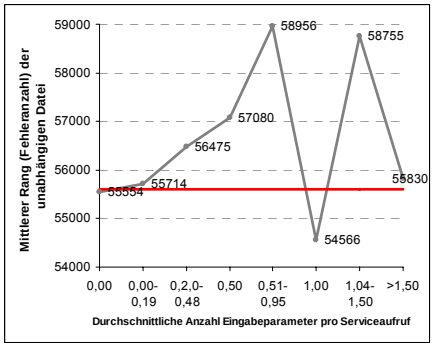


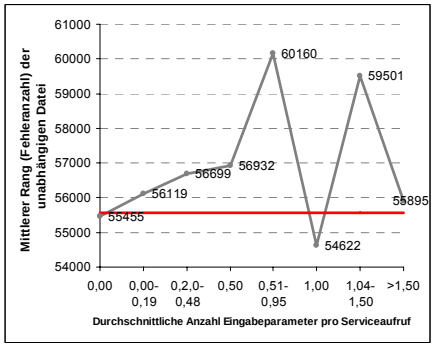
Durchschnittliche Anzahl komplexer Eingabeparameter pro Serviceaufruf

Abhängige Datei		Unabhängige Datei	
01.07.2008	31.07.2008	01.07.2008	31.07.2008



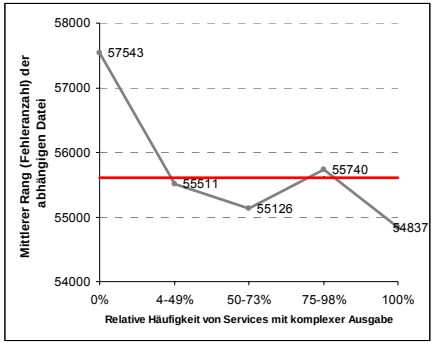


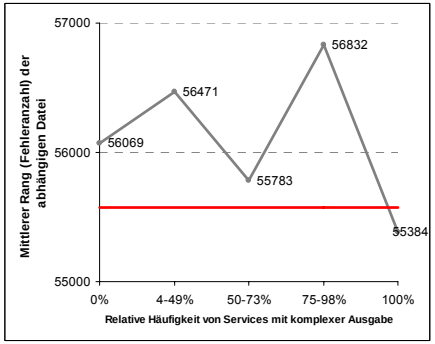


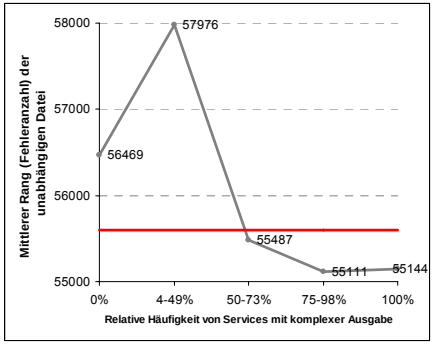


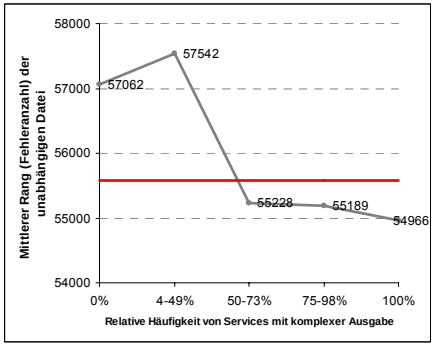
Durchschnittliche Anzahl Eingabeparameter pro Serviceaufruf

Abhängige Datei		Unabhängige Datei	
01.07.2008	31.07.2008	01.07.2008	31.07.2008









Relative Häufigkeit Services mit komplexer Ausgabe

Anhang C: Ergebnisse der Algorithmenanwendung

Tabelle 21: Ergebnisse des Algorithmus von Tai & Daniels ohne Berücksichtigung des Testfokus

	Dauer in Millisekunden	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**162**	**34**	**74**	**211**	**21**	**0**	**0**	**27**
JMol	**648**	**84**	**189**	**1068**	**158**	**0**	**45**	**260**
Freenet	**1228**	**145**	**311**	**616**	**163**	**0**	**7**	**351**
TVBrowser	**5735**	**258**	**577**	**1219**	**104**	**0**	**110**	**678**
FOP	**3159**	**387**	**818**	**1458**	**165**	**0**	**108**	**677**
CDK	**7440**	**312**	**1389**	**2853**	**17**	**0**	**97**	**707**
ANT	**2505**	**170**	**394**	**1052**	**113**	**0**	**166**	**715**
Jetspeed	**2183**	**219**	**535**	**1074**	**123**	**0**	**19**	**756**
Eclipse	**949035**	**3464**	**15513**	**28060**	**5625**	**0**	**8972**	**12623**

Tabelle 22: Ergebnisse des Algorithmus von Le Traon et al. ohne Berücksichtigung des Testfokus

	Dauer	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**16**	**6**	**10**	**23**	**8**	**0**	**1**	**26**
Jmol	**38**	**42**	**168**	**827**	**415**	**27**	**158**	**147**
Freenet	**76**	**59**	**234**	**380**	**236**	**39**	**23**	**335**
TVBrowser	**471**	**173**	**575**	**1313**	**178**	**21**	**305**	**483**
FOP	**462**	**147**	**506**	**1048**	**231**	**87**	**200**	**585**
CDK	**200**	**10**	**99**	**142**	**16**	**13**	**327**	**477**
ANT	**198**	**63**	**172**	**522**	**72**	**27**	**459**	**422**
Jetspeed	**214**	**19**	**27**	**59**	**3**	**1**	**138**	**637**
Eclipse	**166243**	**1706**	**7335**	**15244**	**5828**	**770**	**14536**	**7059**

Tabelle 23: Ergebnisse des Algorithmus von Briand et al. ohne Berücksichtigung des Testfokus

	Dauer	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**10**	**3**	**3**	**6**	**6**	**0**	**0**	**27**
Jmol	**114**	**59**	**95**	**546**	**104**	**0**	**57**	**248**
Freenet	**371**	**74**	**106**	**196**	**96**	**0**	**2**	**356**
TVBrowser	**13953**	**143**	**218**	**620**	**40**	**0**	**53**	**735**
FOP	**3297**	**101**	**138**	**262**	**33**	**0**	**45**	**740**
CDK	**593**	**38**	**69**	**111**	**2**	**0**	**0**	**804**
ANT	**603**	**51**	**71**	**161**	**37**	**0**	**126**	**755**
Jetspeed	**487**	**23**	**26**	**50**	**2**	**0**	**8**	**767**
Eclipse	**1261444**	**1266**	**1956**	**4185**	**1073**	**0**	**9534**	**12061**

Tabelle 24: Ergebnisse des Genetischen Algorithmus ohne Berücksichtigung des Testfokus

	Dauer	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**1430**	**53**	**67**	**105**	**15**	**0**	**3**	**24**
Jmol	**6373**	**111**	**188**	**550**	**150**	**0**	**73**	**232**
Freenet	**9930**	**159**	**278**	**436**	**152**	**0**	**4**	**354**
TVBrowser	**29431**	**284**	**561**	**976**	**85**	**0**	**121**	**667**
FOP	**41063**	**333**	**578**	**900**	**214**	**2**	**78**	**707**
CDK	**44385**	**319**	**602**	**1004**	**20**	**1**	**168**	**636**
ANT	**40320**	**164**	**261**	**520**	**80**	**0**	**193**	**688**
Jetspeed	**64891**	**156**	**216**	**316**	**19**	**1**	**28**	**747**
Eclipse	**14639624**	**3866**	**8983**	**14201**	**3138**	**8**	**9608**	**11987**

Tabelle 25: Ergebnisse des zufallsbasierten Algorithmus ohne Berücksichtigung des Testfokus

	Dauer	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**35**	**63**	**135**	**274**	**26**	**12**	**0**	**27**
Jmol	**139**	**163**	**707**	**2144**	**1047**	**57**	**82**	**223**
Freenet	**466**	**292**	**876**	**1647**	**637**	**91**	**88**	**270**
TVBrowser	**1504**	**468**	**1871**	**3626**	**882**	**129**	**87**	**701**
FOP	**2191**	**570**	**2193**	**4234**	**1164**	**309**	**134**	**651**
CDK	**1502**	**456**	**2317**	**5043**	**476**	**262**	**117**	**687**
ANT	**2924**	**368**	**1627**	**3691**	**681**	**339**	**204**	**667**
Jetspeed	**6633**	**652**	**2043**	**3667**	**672**	**402**	**122**	**653**
Eclipse	**1056448**	**6480**	**44866**	**77449**	**23772**	**4706**	**11298**	**10297**

Tabelle 26: Ergebnisse des Simulated Annealing Algorithmus ohne Berücksichtigung des Testfokus

	Dauer	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**7072**	**3**	**5**	**10**	**0**	**0**	**4**	**23**
Jmol	**12441**	**73**	**104**	**488**	**58**	**0**	**94**	**211**
Freenet	**24730**	**82**	**119**	**179**	**78**	**0**	**1**	**357**
TVBrowser	**90923**	**155**	**234**	**426**	**40**	**0**	**180**	**608**
FOP	**156616**	**142**	**196**	**237**	**67**	**0**	**72**	**713**
CDK	**147402**	**38**	**97**	**96**	**103**	**3**	**97**	**707**
ANT	**153989**	**71**	**102**	**187**	**22**	**0**	**242**	**639**
Jetspeed	**190434**	**48**	**53**	**62**	**5**	**0**	**57**	**718**
Eclipse	**77705856**	**4709**	**17542**	**29327**	**10785**	**805**	**10507**	**11088**

Tabelle 27: Ergebnisse des Algorithmus zur idealen Testfokusberücksichtigung ohne Berücksichtigung des Testfokus

	Dauer	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**36**	**33**	**47**	**74**	**12**	**8**	**27**	**0**
Jmol	**394**	**118**	**181**	**832**	**557**	**7**	**305**	**0**
Freenet	**409**	**235**	**555**	**863**	**340**	**88**	**358**	**0**
TVBrowser	**3174**	**405**	**819**	**1668**	**148**	**114**	**788**	**0**
FOP	**685**	**417**	**1131**	**2235**	**450**	**126**	**785**	**0**
CDK	**521**	**222**	**485**	**765**	**95**	**98**	**804**	**0**
ANT	**1331**	**174**	**331**	**642**	**247**	**87**	**881**	**0**
Jetspeed	**388**	**330**	**711**	**1214**	**397**	**178**	**775**	**0**
Eclipse	**486660**	**3404**	**11626**	**13774**	**3599**	**1259**	**21595**	**0**

Tabelle 28: Ergebnisse des Algorithmus von Tai und Daniels mit Berücksichtigung des Testfokus

	Dauer	Fitness	Simulations-aufwand	Test Fokus Berücksichtigung	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**162**	**10,52**	**7,54**	**13,50**	**34**	**74**	**211**	**21**	**0**	**0**	**27**
JMol	**648**	**34,71**	**4,43**	**65,00**	**84**	**189**	**1068**	**158**	**0**	**45**	**260**
Freenet	**1228**	**56,98**	**26,20**	**87,75**	**145**	**311**	**616**	**163**	**0**	**7**	**351**
TVBrowser	**5735**	**85,43**	**35,26**	**135,60**	**258**	**577**	**1219**	**104**	**0**	**110**	**678**
FOP	**3159**	**111,07**	**52,89**	**169,25**	**387**	**818**	**1458**	**165**	**0**	**108**	**677**
CDK	**7440**	**92,79**	**44,17**	**141,40**	**312**	**1389**	**2853**	**17**	**0**	**97**	**707**
ANT	**2505**	**101,46**	**24,18**	**178,75**	**170**	**394**	**1052**	**113**	**0**	**166**	**715**
Jetspeed	**2183**	**139,44**	**26,89**	**252,00**	**219**	**535**	**1074**	**123**	**0**	**19**	**756**
Eclipse	**949035**	**821,92**	**241,29**	**1402,56**	**3464**	**15513**	**28060**	**5625**	**0**	**8972**	**12623**

Tabelle 29: Ergebnisse des Algorithmus von Le Traon et al. mit Berücksichtigung des Testfokus

	Dauer	Fitness	Simulations-aufwand	Test Fokus Berücksichtigung	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**16**	**7,03**	**1,05**	**13,00**	**6**	**10**	**23**	**8**	**0**	**1**	**26**
Jmol	**38**	**33,15**	**29,56**	**36,75**	**42**	**168**	**827**	**415**	**27**	**158**	**147**
Freenet	**76**	**69,20**	**54,64**	**83,75**	**59**	**234**	**380**	**236**	**39**	**23**	**335**
TVBrowser	**471**	**76,58**	**56,56**	**96,60**	**173**	**575**	**1313**	**178**	**21**	**305**	**483**
FOP	**462**	**129,82**	**113,40**	**146,25**	**147**	**506**	**1048**	**231**	**87**	**200**	**585**
CDK	**200**	**54,96**	**14,52**	**95,40**	**10**	**99**	**142**	**16**	**13**	**327**	**477**
ANT	**198**	**70,37**	**35,23**	**105,50**	**63**	**172**	**522**	**72**	**27**	**459**	**422**
Jetspeed	**214**	**107,37**	**2,40**	**212,33**	**19**	**27**	**59**	**3**	**1**	**138**	**637**
Eclipse	**166243**	**819,14**	**853,94**	**784,33**	**1706**	**7335**	**15244**	**5828**	**770**	**14536**	**7059**

Tabelle 30: Ergebnisse des Algorithmus von Briand et al. mit Berücksichtigung des Testfokus

	Dauer	Fitness	Simulations-aufwand	Test Fokus Berücksichtigung	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**10**	**6,97**	**0,44**	**13,50**	**3**	**3**	**6**	**6**	**0**	**0**	**27**
Jmol	**114**	**32,12**	**2,24**	**62,00**	**59**	**95**	**546**	**104**	**0**	**57**	**248**
Freenet	**371**	**48,95**	**8,91**	**89,00**	**74**	**106**	**196**	**96**	**0**	**2**	**356**
TVBrowser	**13953**	**81,43**	**15,85**	**147,00**	**143**	**218**	**620**	**40**	**0**	**53**	**735**
FOP	**3297**	**96,96**	**8,91**	**185,00**	**101**	**138**	**262**	**33**	**0**	**45**	**740**
CDK	**593**	**81,35**	**1,90**	**160,80**	**38**	**69**	**111**	**2**	**0**	**0**	**804**
ANT	**603**	**96,45**	**4,15**	**188,75**	**51**	**71**	**161**	**37**	**0**	**126**	**755**
Jetspeed	**487**	**128,48**	**1,29**	**255,67**	**23**	**26**	**50**	**2**	**0**	**8**	**767**
Eclipse	**1261444**	**686,66**	**33,21**	**1340,11**	**1266**	**1956**	**4185**	**1073**	**0**	**9534**	**12061**

Tabelle 31: Ergebnisse des genetischen Algorithmus mit Berücksichtigung des Testfokus

	Dauer	Fitness	Simulations-aufwand	Test Fokus Berücksichtigung	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**1811**	**7,07**	**5,64**	**8,50**	**49**	**65**	**121**	**32**	**0**	**10**	**17**
Jmol	**11610**	**12,15**	**8,55**	**15,75**	**143**	**321**	**1247**	**587**	**2**	**242**	**63**
Freenet	**20566**	**30,47**	**31,94**	**29,00**	**221**	**390**	**668**	**174**	**2**	**242**	**116**
TVBrowser	**80171**	**44,28**	**46,16**	**42,40**	**413**	**802**	**1497**	**163**	**0**	**576**	**212**
FOP	**112002**	**65,01**	**55,77**	**74,25**	**399**	**837**	**1332**	**312**	**6**	**488**	**297**
CDK	**134825**	**37,92**	**28,45**	**47,40**	**414**	**849**	**1581**	**92**	**3**	**567**	**237**
ANT	**112310**	**42,65**	**26,55**	**58,75**	**240**	**410**	**995**	**73**	**4**	**646**	**235**
Jetspeed	**158453**	**69,59**	**38,52**	**100,67**	**397**	**643**	**1128**	**130**	**9**	**473**	**302**
Eclipse	**44404829**	**416,00**	**258,32**	**573,67**	**4906**	**14368**	**24416**	**5597**	**45**	**16432**	**5163**

Tabelle 32: Ergebnisse des zufallbasierten Algorithmus mit Berücksichtigung des Testfokus

	Dauer	Fitness	Simulations-aufwand	Test Fokus Berücksichtigung	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**33**	**15,67**	**19,83**	**11,50**	**61**	**115**	**207**	**54**	**11**	**4**	**23**
Jmol	**316**	**58,47**	**68,45**	**48,50**	**169**	**620**	**1843**	**1670**	**59**	**111**	**194**
Freenet	**673**	**114,05**	**160,84**	**67,25**	**288**	**933**	**1683**	**459**	**95**	**89**	**269**
TVBrowser	**2536**	**171,70**	**242,00**	**101,40**	**505**	**1869**	**3807**	**653**	**136**	**281**	**507**
FOP	**3298**	**291,95**	**428,90**	**155,00**	**616**	**2248**	**4181**	**1159**	**310**	**165**	**620**
CDK	**5102**	**239,82**	**342,23**	**137,40**	**469**	**2550**	**5640**	**554**	**275**	**117**	**687**
ANT	**3341**	**285,64**	**405,77**	**165,50**	**372**	**1861**	**4210**	**807**	**331**	**219**	**662**
Jetspeed	**4816**	**346,56**	**489,13**	**204,00**	**636**	**2093**	**4153**	**689**	**411**	**163**	**612**
Eclipse	**1491713**	**3124,67**	**5166,56**	**1082,78**	**6570**	**44820**	**77122**	**25944**	**4701**	**11850**	**9745**

Tabelle 33: Ergebnisse des Simulated Annealing Algorithmus mit Berücksichtigung des Testfokus (zufällige Startreihenfolge)

	Dauer	Fitness	Simulations-aufwand	Test Fokus Berücksichtigung	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**3865**	**2,52**	**2,54**	**2,50**	**28**	**34**	**45**	**13**	**0**	**22**	**5**
Jmol	**40166**	**2,66**	**3,57**	**1,75**	**117**	**193**	**607**	**245**	**0**	**298**	**7**
Freenet	**79413**	**15,42**	**15,35**	**15,50**	**159**	**214**	**323**	**86**	**0**	**296**	**62**
TVBrowser	**367612**	**17,88**	**22,36**	**13,40**	**281**	**395**	**726**	**64**	**0**	**721**	**67**
FOP	**494116**	**22,72**	**21,19**	**24,25**	**250**	**376**	**510**	**123**	**1**	**688**	**97**
CDK	**629377**	**9,51**	**9,43**	**9,60**	**214**	**310**	**499**	**17**	**1**	**756**	**48**
ANT	**510482**	**11,48**	**10,47**	**12,50**	**142**	**215**	**427**	**32**	**0**	**831**	**50**
Jetspeed	**713296**	**23,62**	**14,24**	**33,00**	**234**	**293**	**505**	**29**	**1**	**676**	**99**
Eclipse	**291584990**	**945,50**	**1316,67**	**574,33**	**5491**	**21783**	**37953**	**11012**	**1032**	**16426**	**5169**

Tabelle 34: Ergebnisse des Simulated Annealing Algorithmus mit Berücksichtigung des Testfokus (Startreihenfolge Algorithmus Borner)

	Dauer	Fitness	Simulations-aufwand	Test Fokus Berücksichtigung	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**1777**	**2,70**	**1,89**	**3,50**	**25**	**27**	**41**	**3**	**0**	**20**	**7**
Jmol	**15168**	**3,15**	**4,80**	**1,50**	**115**	**192**	**615**	**390**	**1**	**299**	**6**
Freenet	**27418**	**17,36**	**19,97**	**14,75**	**171**	**249**	**382**	**115**	**2**	**299**	**59**
TVBrowser	**119189**	**19,25**	**24,10**	**14,40**	**308**	**431**	**771**	**71**	**0**	**716**	**72**
FOP	**165696**	**25,23**	**27,97**	**22,50**	**274**	**425**	**637**	**112**	**4**	**695**	**90**
CDK	**199275**	**13,34**	**16,68**	**10,00**	**186**	**365**	**588**	**13**	**7**	**754**	**50**
ANT	**168281**	**14,96**	**16,43**	**13,50**	**150**	**228**	**396**	**51**	**6**	**827**	**54**
Jetspeed	**233882**	**28,16**	**21,33**	**35,00**	**217**	**302**	**519**	**39**	**8**	**670**	**105**
Eclipse	**93285070**	**351,55**	**617,10**	**86,00**	**3553**	**9200**	**12043**	**3390**	**519**	**20821**	**774**

Tabelle 35: Ergebnisse des Simulated Annealing Algorithmus mit Berücksichtigung des Testfokus (Startreihenfolge Algorithmus Briand)

	Dauer	Fitness	Simulations-aufwand	Test Fokus Berücksichtigung	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**1543**	**4,12**	**1,24**	**7,00**	**12**	**14**	**20**	**10**	**0**	**13**	**14**
Jmol	**15031**	**3,99**	**2,72**	**5,25**	**103**	**155**	**432**	**153**	**0**	**284**	**21**
Freenet	**28352**	**19,06**	**17,12**	**21,00**	**159**	**229**	**371**	**94**	**0**	**274**	**84**
TVBrowser	**132783**	**20,78**	**22,95**	**18,60**	**288**	**412**	**739**	**65**	**0**	**695**	**93**
FOP	**172531**	**33,36**	**22,72**	**44,00**	**237**	**344**	**537**	**162**	**3**	**609**	**176**
CDK	**204850**	**15,24**	**11,27**	**19,20**	**216**	**374**	**604**	**20**	**1**	**708**	**96**
ANT	**168671**	**19,60**	**10,21**	**29,00**	**123**	**184**	**378**	**32**	**1**	**765**	**116**
Jetspeed	**239631**	**53,12**	**12,56**	**93,67**	**173**	**216**	**373**	**26**	**3**	**494**	**281**
Eclipse	**99280619**	**567,15**	**157,74**	**976,56**	**2962**	**6964**	**13372**	**2552**	**49**	**12806**	**8789**

Tabelle 36: Ergebnisse des Algorithmus von Borner zur idealen Testfokusberücksichtigung

	Dauer	Fitness	Simulations-aufwand	Test Fokus Berücksichtigung	Simulierte Dateien	Simulierte Abhängigkeiten	Simulierte Serviceaufrufe	Simulierte Attributzugriffe	Aufgebrochene Vererbungen	Richtig integrierte Abhängigkeiten	Zu spät integrierte Abhängigkeiten
OSCache	**36**	**10,79**	**10,79**	**0,00**	**33**	**47**	**74**	**12**	**8**	**27**	**0**
Jmol	**394**	**11,00**	**11,00**	**0,00**	**118**	**181**	**832**	**557**	**7**	**305**	**0**
Freenet	**409**	**122,99**	**122,99**	**0,00**	**235**	**555**	**863**	**340**	**88**	**358**	**0**
TVBrowser	**3174**	**155,48**	**155,48**	**0,00**	**405**	**819**	**1668**	**148**	**114**	**788**	**0**
FOP	**685**	**191,34**	**191,34**	**0,00**	**417**	**1131**	**2235**	**450**	**126**	**785**	**0**
CDK	**521**	**106,27**	**106,27**	**0,00**	**222**	**485**	**765**	**95**	**98**	**804**	**0**
ANT	**1331**	**98,10**	**98,10**	**0,00**	**174**	**331**	**642**	**247**	**87**	**881**	**0**
Jetspeed	**388**	**200,28**	**200,28**	**0,00**	**330**	**711**	**1214**	**397**	**178**	**775**	**0**
Eclipse	**486660**	**1348,66**	**1348,66**	**0,00**	**3404**	**11626**	**13774**	**3599**	**1259**	**21595**	**0**

Anhang D: Konfigurationsparameter für TOC

Parameter	Wertebereich und Beschreibung
ALGORITHM	Wertebereich: LeTraon, Tai&Daniels, Briand, Genetic, SimulatedAnnealing, Random, Borner *Der Parameter ALGORITHM beschreibt, welche Algorithmen zur Bestimmung der Integrationsreihenfolge verwendet werden. Es kann ein oder mehrere Algorithmen ausgewählt werden.*
WEIGHT_SC_INHERITANCES	Wertebereich: reelle Zahlen zwischen 0 und 1 *Der Parameter WEIGHT_SC_INHERITANCES beschreibt den Einfluss einer aufgebrochenen Vererbungsbeziehung auf den Simulationsaufwand einer Integrationsreihenfolge.* *Die drei Werte WEIGHT_SC_INHERITANCES, WEIGHT_SC_SERVICES und WEIGHT_SC_ATTRIBUTES sollten in der Summe 1 ergeben.*
WEIGHT_SC_SERVICES	Wertebereich: reelle Zahlen zwischen 0 und 1 *Der Parameter WEIGHT_SC_SERVICES definiert den Einfluss der zu simulierenden Serviceaufrufe auf die Simulationskosten einer Integrationsreihenfolge.* *Die drei Werte WEIGHT_SC_INHERITANCES, WEIGHT_SC_SERVICES und WEIGHT_SC_ATTRIBUTES sollten in der Summe 1 ergeben.*
WEIGHT_SC_ATTRIBUTES	Wertebereich: reelle Zahlen zwischen 0 und 1 Der Parameter WEIGHT_SC_ATTRIBUTES definiert den Einfluss der zu simulierenden Attributzugriffe auf die Simulationskosten einer Integrationsreihenfolge. *Die drei Werte WEIGHT_SC_INHERITANCES, WEIGHT_SC_SERVICES und WEIGHT_SC_ATTRIBUTES sollten in der Summe 1 ergeben.*
WEIGHT_FITNESS_SC	Wertebereich: reelle Zahlen zwischen 0 und 1 *Der Parameter WEIGHT_FITNESS_SC definiert den Einfluss der Simulationskosten auf die Fitness einer Integrationsreihenfolge. Er beschreibt, wie stark der Einfluss der Simulationskosten auf die Integrationsreihenfolge ist.* *Die Werte WEIGHT_FITNESS_SC und WEIGHT_FITNESS_TP sollten in der Summe 1 ergeben.*
WEIGHT_FITNESS_TP	Wertebereich: reelle Zahl zwischen 0 und 1 *Der Parameter Weight_FITNESS_TP definiert den Einfluss der Testfokusauswahl auf die Fitness einer Integrationsreihenfolge. Er beschreibt, wie stark der Einfluss der Testfokusauswahl auf die Integrationsreihenfolge.* *Die Werte WEIGHT_FITNESS_SC und WEIGHT_FITNESS_TP sollten in der Summe 1 ergeben.*
EXPORT_RESULTS	Wertebereich: true, false *Dieser Parameter beschreibt, ob die Ergebnisse der Reihenfolgeermittlung exportiert werden sollen. Wenn der Parameter auf false gesetzt worden ist, so werden die Ergebnisse nur auf der Konsole ausgegeben.*
EXPORT_FILE	Wertebereich: Pfad zur Exportdatei *Mit diesem Parameter kann definiert werden, in welche Datei das Ergebnis der Integrationsreihenfolge exportiert wird.*
SA_PROCENT_VALUE	Wertebereich: reelle Zahl zwischen 0 und 1 *Dieser Parameter ist für die Einstellung des Simulated Annealing Algorithmus (SA). Mit ihm kann definiert werden, um welchen Faktor die aktuelle Temperatur gesenkt wird. Je größer die Zahl, desto schneller werden die Berechnungen abgeschlossen, aber desto schlechter die Ergebnisse.*

SA_START_TEMPERATURE	Wertebereich: reelle Zahl zwischen 0 und 1 *Dieser Parameter definiert die Starttemperatur für den Simulated Annealing Algorithmus. Je größer die Starttemperatur gewählt wird, desto mehr schlechte Reihenfolgen werden am Anfang akzeptiert.*
SA_END_TEMPERATURE	Wertebereich: reelle Zahl zwischen 0 und 1 *Dieser Parameter definiert die Abbruchbedingungen für den Simulated Annealing Algorithmus. Je geringer die Endtemperatur gewählt wird, desto länger rechnet der Algorithmus und desto weniger schlechte Reihenfolgen werden gegen Ende der Berechnung akzeptiert.*
SA_NUMBER_INNER_ITERATIONS	Wertebereich: positive ganze Zahl (oder -1) *Dieser Parameter definiert die Anzahl Versuche innerhalb einer Iteration, um neue Reihenfolgen zu ermitteln. Dieser Wert kann auf -1 gesetzt werden, um die Anzahl der Bausteine als Anzahl innerer Iterationen zu verwenden.*
GA_NUMBER_ITERATIONS	Wertebereich: positive ganze Zahl *Der Parameter GA_NUMBER_ITERATIONS definiert die Dauer des Genetischen Algorithmus. Er legt fest, wie oft eine neue Population errechnet wird.*
GA_POPULATION_SIZE	Wertebereich: positive ganze Zahl (oder -1) Der Parameter beschreibt, aus wie vielen Integrationsreihenfolgen eine Population besteht. Bei einem gewählten Wert von -1 wird die Anzahl der Bausteine zur Definition der Populationsgröße verwendet.
DATA_IMPORT	Wertebereich: SissyDatabase, ExcelFile *Der Parameter DATA_IMPORT beschreibt, woher die Informationen über die Abhängigkeiten zu bekommen sind. Standardmäßig werden Excel-Dateien im CSV Format importiert. Es können aber auch Informationen direkt aus einer Sissy-Datenbank gelesen werden.*
DATABASE_TYPE	Wertebereich: mysql, postgres *Beschreibt, ob sich TOC mit einer PostgreSQL [Po09] oder einer MySQL [My09] Datenbank verbinden soll, um die Abhängigkeitseigenschaften zu erheben.*
DATABASE_IP	Wertebereich: IP-Adresse *Beschreibt die IP-Adresse, unter der die Datenbank erreichbar ist.*
DATABASE_PORT	Wertebereich: positive ganze Zahl *Beschreibt den Port, der für die Kommunikation mit der Datenbank verwendet wird.*
DATABASE_LOGIN	*Login zur Authentifizierung an der Datenbank*
DATABASE_PASSWORD	Passwort für die Authentifizieren an der Datenbank
DATABASE_NAME	*Name der Datenbank, die das Quelltextmodell der SISSy Analyse enthält.*
EXCEL_SEPARATOR	*Ein oder mehrere Zeichen, die in der CSV Datei verwendet werden, um die Spalten zu trennen. Standardmäßig wird das Semikolon (;) verwendet.*
DEPENDENCY_ID_COLUMN	Wertebereich: positive ganze Zahl *Dieser Parameter beschreibt, in welcher Spalte der importieren Exceldatei sich die Identifikationsnummer der Abhängigkeiten befindet. Diese Identifikationsnummer setzt sich aus den SISSy Identifikationsnummern der beteiligten Quelltextdateien zusammen.*
DEPENDENCY_CLIENTNAME_COLUMN DEPENDENCY_SERVERNAME_COLUMN	Wertebereich: positive ganze Zahl *Diese zwei Parameter geben an, in welcher Spalte der Exceldatei sich der vollständige Name der abhängigen Datei (Client) bzw. der unabhängigen Datei (Server) befindet.*

DEPENDENCY_ATTRIBUTEACCESSES_COLUMN DEPENDENCY_SERVICECALL_COLUMN	Wertebereich: positive ganze Zahl *Diese Parameter beschreiben, in welcher Spalte sich die Anzahl der zugegriffenen Attribute bzw. die Anzahl der aufgerufenen Services befinden.*
DEPENDENCY_INHERITANCE_COLUMN	Wertebereich: positive ganze Zahl *Dieser Parameter definiert, in welcher Spalte sich die Information befindet, ob es sich um eine Vererbungsbeziehung.*
DEPENDENCY_TESTPRIORITY_COLUMN	Wertebereich: positive ganze Zahl *Dieser Parameter beschreibt, in welcher Spalte der Exceldatei die Testpriorität der Abhängigkeit definiert ist.*