Masterarbeit

Julia Hosieny

Ein Framework zur automatisierten Fehlererkennung und -lokalisierung für Java-Anwendungen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Julia Hosieny

Ein Framework zur automatisierten Fehlererkennung und -lokalisierung für Java-Anwendungen

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Olaf Zukunft
Zweitgutachter : Prof. Dr. Stefan Sarstedt

Abgegeben am 13. Mai 2011

Julia Hosieny

Thema der Masterarbeit

Ein Framework zur automatisierten Fehlererkennung und -lokalisierung für Java-Anwendungen

Stichworte

Framework, Monitoring, Debugging, automatisiert, Java

Kurzzusammenfassung

Das Ziel dieser Masterarbeit ist die Konzeption und die Realisierung eines Frameworks zur automatisierten Fehlererkennung und -lokalisierung für Java-Anwendungen, einschließlich der Hilfeleistungen in der Fehlerbehebung. Das intendierte Framework bietet eine umfangreiche Unterstützung für das Monitoring und Debugging. Insbesondere dient es der Qualitätssicherung für Anwendungen im laufenden Betrieb. Für das Erreichen der Zielsetzung werden in dieser Masterarbeit einige essentielle Schritte durchlaufen. Der Aufbau der Arbeit gestaltet sich folgendermaßen: Einführung in die Literatur (Framework, Monitoring, Debugging), Definition der genauen Anforderungen, eine Darstellung vorhandener Lösungsansätze und Systeme, die Erstellung eines Konzepts und einer geeigneten Architektur sowie die Realisierung und Bewertung des Prototypen.

Julia Hosieny

Title of the paper

A framework that serves to automatically detect and localise defects in Java applications

Keywords

Framework, Monitoring, Debugging, automated, Java

Abstract

The aim of this master thesis is to work up a concept and to put to practice a framework that serves as a means to automatically detect and localise defects relating to Java applications, including tools that help disposing of the defects. The resulting framework is to provide a comprehensive support of the monitoring and debugging features. In particular, this framework is to serve the quality assurance for applications that are in actual use. To that end, a number of steps will be presented in the course of this thesis. These consist in an introduction into the specific literature (framework, monitoring, debugging), a definition of the claims, a presentation of the different approaches to solutions and related systems, the provision of a concept, and a suitable architecture, as well as the realisation and an assessment of the prototypes.

Ich widme diese Arbeit
ganz besonderen Menschen;

meiner ehrenhaften Mutter,
meinem verstorbenen Vater
und verstorbenen Schwiegervater
und meinem lieben Ehemann.

Danksagung

Ein großer Dank geht an meinen betreuenden Prüfer Herr Prof. Dr. Olaf Zukunft für seine bemerkenswerte Geduld und konstruktive Kritik, die diese Masterarbeit wissenschaftlich sehr gefördert hat. Ein weiterer Dank wird an meinen Zweitgutachter Prof. Dr. Stefan Sarstedt ausgesprochen, der mit Hilfe seines Feedback zur letzten Abrundung dieser Arbeit beigetrug.

Vielen Dank an meine Familie und Freunde, die zur Motivation beitrugen und mich moralisch unterstützten. Mein besonderer Dank geht an meinen Ehemann Arazm Hosieny, der zur selben Zeit ebenfalls seine Masterarbeit verfasste und jeden Tag Seite an Seite mit mir die Hürden überwunden hat.

Inhaltsverzeichnis

Tabellenverzeichnis X

Abbildungsverzeichnis XI

1. Einführung 1
1.1. Problemstellung 1
1.2. Aufgabenstellung und Ziel 2
1.3. Gliederung der Arbeit 2

2. Grundlagen 4
2.1. Framework 4
2.1.1. Framework: Definition & Struktur 4
2.1.1.1. Definition 4
2.1.1.2. Abgrenzung 5
2.1.1.3. Hollywood-Prinzip 7
2.1.1.4. Bestandteile eines Frameworks 7
2.1.2. Anforderungen an Frameworks 8
2.1.3. Klassifikationen 10
2.1.3.1. Klassifizierung nach Spezifizier- bzw. Erweiterbarkeit 10
2.1.3.2. Klassifizierung nach Ausdehnungsebene 11
2.1.3.3. Weitere Klassifizierungen 11
2.1.4. Entwicklung & Vorgehensweise 11
2.2. Monitoring 13
2.2.1. Definition 13
2.2.2. Klassifikation 14
2.2.2.1. Spezifikationssprache 14
2.2.2.2. Monitor 16
2.2.2.3. Behandeln von Ereignissen 17
2.2.2.4. Operationale Fragen 17
2.3. Debugging 18
2.3.1. Definition 18
2.3.2. Fehlerbezeichnungen 19
2.3.3. Klassifikationen von Fehlern 20

2.3.3.1. Klassifizierung nach Fehlertypen . . . 20
2.3.3.2. Klassifizierung nach Reproduzierbarkeit . . . 21
2.3.3.3. Klassifizierung nach zeitlichem Aspekt . . . 22
2.3.4. Methodiken . . . 22
2.3.4.1. Trace Debugging . . . 23
2.3.4.2. Slice Debugging . . . 23
2.3.4.3. Delta Debugging . . . 24
2.3.4.4. Deklaratives (Algorithmisches) Debugging . . . 24

3. Anforderungen **26**
3.1. Systemidee . . . 26
3.2. Fachliche Anforderungen . . . 27
3.3. Technische Anforderungen . . . 33

4. Existierende Ansätze und Marktanalyse **35**
4.1. Vorgehen . . . 35
4.1.1. Vorgehen zur Recherche existierender Ansätze . . . 36
4.1.2. Vorgehen zur Marktanalyse . . . 36
4.2. Monitoring . . . 37
4.2.1. Existierende Ansätze: Monitoring . . . 37
4.2.1.1. Aspektorientiertes Programmieren (AOP) . . . 38
4.2.1.2. DynaMICs . . . 38
4.2.1.3. Java with Assertions (Jass) . . . 38
4.2.1.4. Java PathExplorer (JPaX) . . . 39
4.2.1.5. Java Runtime Timing-Constraint Monitor (JRTM) . . . 39
4.2.1.6. Monitoring and Checking (MaC) . . . 40
4.2.1.7. Monitoring-Oriented Programming (MoP) . . . 40
4.2.1.8. Runtime Assertion Checker for the Java Modeling Language (RAC) . . . 41
4.2.2. Marktanalyse: Monitoring . . . 41
4.2.2.1. Java Application Monitor (JAMon) . . . 41
4.2.2.2. JConsole . . . 42
4.2.3. Vergleich . . . 42
4.2.4. Evaluation Monitoring . . . 44
4.3. Debugging . . . 44
4.3.1. Existierende Ansätze: Debugging . . . 44
4.3.1.1. Debugging strategy based on Requirements of Testing (DRT) 45
4.3.1.2. Java Diagnosis Experiments (JADE) . . . 45
4.3.1.3. Java Debugger using Aspect and Slicing (JDAS) . . . 46
4.3.1.4. Java Interactive Visualization Environment (JIVE) . . . 47
4.3.1.5. Java Platform Debugger Architecture (JPDA) . . . 48

4.3.2. Marktanalyse: Debugging . . . 49
4.3.2.1. jBixbe . . . 49
4.3.2.2. The Java Debugger (JDB) . . . 50
4.3.2.3. JSwat . . . 50
4.3.3. Evaluation Debugging . . . 51

5. Konzept und Architektur **52**
5.1. Konzeptidee . . . 52
5.1.1. Laufzeitfehler in Java . . . 52
5.1.2. Gruppierung der Laufzeitfehler . . . 53
5.1.3. Exceptions erkennen und protokollieren . . . 55
5.1.4. Laufzeitfehler reproduzieren . . . 55
5.1.5. Debugging Funktionalitäten . . . 56
5.1.6. Versionsverwaltung . . . 57
5.1.7. Webbrowser . . . 57
5.2. Architektur . . . 57
5.2.1. Komponentenübersicht . . . 58
5.2.2. Komponentenverteilung . . . 60
5.3. Interne Aktivitäten . . . 65
5.3.1. Exceptionfilterung . . . 65
5.3.2. Fehlererkennung . . . 68
5.3.3. Fehlerlokalisierung und -behebung . . . 69
5.4. Konkretes Systemmodell . . . 71
5.4.1. Clientadapter . . . 71
5.4.2. Server-Webanwendung . . . 74
5.4.2.1. Gesamtübersicht . . . 75
5.4.2.2. Detaillierter Auszug . . . 77

6. Realisierung **81**
6.1. Angewandte Technologien . . . 81
6.1.1. Basis Technologien . . . 81
6.1.1.1. Entwicklungssprache . . . 82
6.1.1.2. Entwicklungsumgebung . . . 82
6.1.1.3. Webserver . . . 82
6.1.1.4. Versionsverwaltungssystem . . . 83
6.1.2. Präsentation . . . 83
6.1.2.1. Web-Framework . . . 83
6.1.3. Logik . . . 84
6.1.3.1. Applikations-Framework . . . 84
6.1.3.2. Reflection . . . 85
6.1.3.3. AOP . . . 85

6.1.4. Datenbank . . . 86
6.1.4.1. ORM-Framework . . . 86
6.1.4.2. Datenbanksystem . . . 86
6.1.5. Kommunikation . . . 87
6.1.5.1. Webservices . . . 87
6.1.6. Zusammenfassung der angewandten Technologien . . . 87
6.2. Abweichungen vom Konzept . . . 88
6.2.1. Parsen der Exceptions . . . 89
6.2.2. Abspeichern der Methodenaufrufe . . . 89
6.2.3. Laden der Objekte . . . 90
6.2.4. Darstellung der Methodenaufrufe . . . 90
6.2.5. Inkonsistenz der Daten . . . 91
6.3. Tests . . . 92
6.4. System-Präsentation . . . 94
6.4.1. Exceptionparsing und -einstellungen . . . 94
6.4.2. Ansicht über aufgetretene Exceptions inklusive historisches Debugging 97

7. Fazit und Ausblick **102**
7.1. Fazit . . . 102
7.2. Ausblick . . . 104

Literaturverzeichnis **106**

A. Anwendungsfälle **114**

B. Inhalt der beigefügten CD-ROM **116**

Tabellenverzeichnis

2.1. Zusammenfassung der Fehlerbezeichnungen 20

3.1. UC1: Framework integrieren 29
3.2. UC2: Fehlereinstellungen durchführen 29
3.3. UC3: aufgetretene Fehler erkennen und speichern 30
3.4. UC4: Fehler verwalten 30
3.5. UC5: Automatische Lokalisierung eines Fehlers ansehen 31
3.6. UC6: Fehlerfall im Debuggingmodus verfolgen 32
3.7. UC7: Vergleich durchführen 32

4.1. Vergleich der Monitoringsysteme und -ansätze 43
4.2. Evaluation der Monitoringsysteme und -ansätze 44
4.3. Evaluation der Debuggingsysteme und -ansätze 51

6.1. Zusammenfassung der angewandten Technologien 88
6.2. Evaluation des entwickelten Frameworks (Prototyp) 93

A.1. UC0: Ein- und Ausloggen 114
A.2. UC8: Sourcecodemodifikation durchführen 114
A.3. UC9: Korrektur testen 115
A.4. UC10: Modifikation aufheben 115

Abbildungsverzeichnis

2.1. Evolutionäres Vorgehensmodell [Schmitz (2004)] 12
2.2. Monitoring Klassifikation 14
2.3. Deklaratives Debugging: Top-Down Verfahren 25

3.1. Anwendungsfall-Diagramm 28

4.1. JPDA 48

5.1. Exception-Hierarchie 53
5.2. Fachliche Komponenten 58
5.3. Überblick über mögliche Architekturstile 61
5.4. Zwischenschritt bei der Architekturerstellung 63
5.5. Geeignete Architektur 66
5.6. Aktivitätsdiagramm: Exceptionfilterung 67
5.7. Aktivitätsdiagramm: Fehlererkennung 69
5.8. Aktivitätsdiagramm: Fehlerlokalisierung und -behebung 70
5.9. Klassendiagramm: Client 72
5.10. Klassendiagramm: Server 76
5.11. Auszug aus dem Klassendiagramm: Server 79

6.1. Screenshot: Exception-Hierarchie 95
6.2. Screenshot: Kontaktgruppe erstellen 96
6.3. Screenshot: Kontaktprofil ansehen und bearbeiten 97
6.4. Screenshot: Alle aufgetretenen Exceptions 99
6.5. Screenshot: Exceptiondetails 100
6.6. Screenshot: Abhängige Exceptions 101

1. Einführung

Heutzutage werden im Alltag, ob im Berufs- oder im Privatleben, viele Aktivitäten unter Verwendung von Computern jeglicher Art erledigt. Für die Nutzung dieser Geräte werden Softwaresysteme benötigt. In der Regel enthalten Softwaresysteme Fehler. Aus diesem Grund ist das Erkennen sowie Lokalisieren und Beheben von Fehlern eine Notwendigkeit. Diese Tätigkeiten sind kontinuierliche Aktivitäten und umfassen den gesamten Lebenszyklus eines Softwaresystems, einschließlich der Softwareentwicklungsphase sowie der „Go-Life“ Phase.

1.1. Problemstellung

Aufgrund der steigenden Komplexität von Softwaresystemen erhöht sich zum einen die Anzahl der Fehler und zum anderen gestaltet sich das Erkennen, Lokalisieren und Beheben von Fehlern komplizierter und beschwerlicher.
In vielen Fällen treten Fehler erst nach der Inbetriebnahme eines Softwaresystems auf, da zum einen nicht alle Testfälle abgebildet werden können, und weil zum anderen die Testphase in der Praxis oftmals zu kurz ausfällt. Gründe dafür: ein nicht eingehaltener Zeitplan, hervorgerufen durch z.B. ein schlechtes Zeitmanagement oder unvorhersehbare Probleme.
Der Umgang mit Fehlern, die während der Go-Life Phase auftreten, gestaltet sich aus den folgenden zwei Gründen komplexer:

1. Zum einen ist es schwierig, Kenntnis über einen aufgetretenen Fehler zu erhalten, denn die Benutzer sind nicht verpflichtet, aufgetretene Fehler dem Wartungspersonal zu signalisieren. Ein Anwender, der z.B. im Internet Reiseangebote vergleicht, wird bei einer fehlerhaften Seite sich kaum die Mühe machen, den Anbieter ausfindig zu machen, um ihn über diesen Fehler in Kenntnis zu setzen. In den meisten Fällen wendet sich der Benutzer von dem betreffenden System ab. Das Resultat ist der Verlust eines Kunden. Selbstverständlich besteht ebenso die Möglichkeit, dass der Benutzer bei nicht gravierenden Fehlern diese umgeht, indem ein „Workaround“ praktiziert wird. Dies zählt jedoch nicht zu den anzustrebenden Zielen eines guten Softwaresystems.

2. Ein weiteres Problemfeld ergibt sich, wenn der Benutzer sich entschließt, den aufgetretenen Fehler bekannt zu geben. Die übermittelten Informationen vom Benutzer

bezüglich des aufgetreten Fehlers sind nahezu unbrauchbar, weil sie unprofessionell beschrieben werden. Für die Lokalisierung des Fehlers sind lückenhafte und unklare Informationen ungeeignet. Des Weiteren können Fehler aufgrund anderer Einflussfaktoren auftreten, z.B. Problemen mit der Datenbankverbindung. Diese externen Einflüsse können nicht ausgeschlossen werden und erschweren die Fehlerlokalisierung zusätzlich.

1.2. Aufgabenstellung und Ziel

Die oben angeführten Problematiken bilden den Kern dieser Masterarbeit. Das Erkennen von Fehlern (erste aufgeführte Problematik) fällt unter das Themengebiet Monitoring und die Lokalisierung sowie Behebung von Fehlern (zweite aufgeführte Problematik) umfasst das Debugging. Das Ziel dieser Arbeit ist die Konzeption und die anschließende Realisierung eines Frameworks, basierend auf den beiden Themengebieten Monitoring und Debugging.
Die Hauptaufgaben des Frameworks umfassen zum einen das eigenständige und automatisierte Erkennen und Erfassen von Fehlern, die in einer Anwendung (Zielanwendung) zur Laufzeit auftreten. Zum anderen bildet die automatisierte Lokalisierung von Fehlern eine weitere wichtige Aufgabe des Frameworks. Diese beiden Hauptaufgaben ergeben sich aus den beiden oben vorgestellten Problematiken. Die Fehlererkennung gestaltet sich unabhängig vom Benutzer. Fehler werden ohne Einwirkung des Zielanwendungsbenutzer erkannt. Des Weiteren wird durch die automatisierte Lokalisierung der Fehler, die Erhebung der Informationen vom Benutzer bezüglich des aufgetretenen Fehlers umgangen.
Das zu entwickelnde Framework stellt ebenso eine Unterstützung für die Fehlerbehebung, einschließlich der Fehlerverwaltung und der aktiven Beseitigung von Fehlern, zur Verfügung, um den gesamten Monitoring- und Debugging-Prozess in einem Framework zu vereinen. Für eine schnellstmögliche Reaktion auf schwerwiegende Fehler wird unmittelbar nach dem Auftreten eines Fehlers das Wartungspersonal mit Hilfe des Frameworks informiert. Zudem werden beliebige Java-Anwendungen als Zielanwendungen akzeptiert. Darüber hinaus werden ebenso bereits existierende Java-Anwendungen vom Framework unterstützt. Dementsprechend kann das Framework in vorhandene Java-Anwendungen integriert werden.

1.3. Gliederung der Arbeit

Die Arbeit ist in sieben Kapitel gegliedert:

- Im Anschluss an die Einleitung folgen in Kapitel 2 die Grundlagen. Diese umfassen die relevanten Themengebiete dieser Masterarbeit; Framework 2.1, Monitoring 2.2 und

Debugging 2.3. Jedes Themengebiet beinhaltet die Erläuterung von Begrifflichkeiten, die Präsentation von Basiswissen und Zusatzinformationen.

- Kapitel 3 beschäftigt sich mit der detaillierten Definition der Anforderungen an das Framework. Zu diesem Zweck wird als erstes eine Systemidee 3.1 entwickelt, gefolgt von den fachlichen 3.2, sowie den technischen 3.3 Anforderungen.
- Anschließend wird in Kapitel 4 auf die Marktanalyse und die Recherche existierender Ansätze eingegangen. An erster Stelle werden die Vorgehensweisen 4.1 beider Nachforschungsarten erläutert und die wesentlichen Unterschiede dargestellt. Das Nachforschungsgebiet wird in zwei Bereiche eingeteilt: Monitoring4.2 und Debugging 4.3. Für beide Bereiche wird jeweils eine Marktanalyse und eine Recherche existierender Ansätze durchgeführt und abschließend bewertet.
- Kapitel 5 konzentriert sich auf die Konzeption des zu entwickelnden Frameworks. Zu Beginn wird eine Konzeptidee 5.1 entwickelt. Basierend auf der Konzeptidee und den Anforderungen wird anschließend die Erstellung einer geeigneten Architektur 5.2 vollzogen. Im Anschluss folgt die Beschreibung der internen Aktivitäten 5.3, die abhängig von der Architektur sind. Abgeschlossen wird das Kapitel mit der Betrachtung und der Erläuterung des konkreten Systemmodells 5.4.
- Kapitel 6 befasst sich mit der Realisierung, also mit der Umsetzung des Konzepts in die Praxis. Die angewandten Technologien 6.1, die zur Realisierung des Frameworks zum Einsatz kommen, werden dargestellt und deren Auswahl begründet. Anschließend werden die Realisierungsproblematiken diskutiert und die Abweichungen vom Konzept 6.2 präsentiert. Daraufhin wird zusammenfassend auf die Testergebnisse eingegangen. Zuletzt folgt eine Systempräsentation 6.4 unter Verwendung von Screenshots.
- Das letzte Kapitel 7 fasst die Masterarbeit zusammen 7.1 und gewährt einen Ausblick 7.2.

2. Grundlagen

Das Ziel dieser Masterarbeit ist die Ausarbeitung eines Frameworks. Der Schwerpunkt des Frameworks bezieht sich auf das automatisierte Erkennen und Lokalisieren von Softwarefehlern in Java-Anwendungen. Das Erkennen bzw. Entdecken von Softwarefehlern wird unter dem Begriff Monitoring und das Lokalisieren unter dem Begriff Debugging geführt. Debugging umfasst zudem das Beheben von Fehlern. Das Framework soll ebenfalls dem Anwender die Möglichkeit zur Verfügung stellen, den Fehler beheben zu können. Aufgrund der Zielsetzung befasst sich das Kapitel Grundlagen mit den Begriffen Framework, Monitoring und Debugging.

2.1. Framework

Die Framework-Grundlagen befassen sich zu Beginn mit der Definitionsklärung und Frameworkstruktur. Im Anschluss folgt eine Auflistung wichtiger Anforderungen an das Framework. Es existiert eine Vielzahl von Framework-Klassifikationen; die Autorin stellt die bekanntesten vor und verweist auf weiterführende Literatur. Das Kapitel endet mit einer kurzen Übersicht über die Frameworkentwicklung.

2.1.1. Framework: Definition & Struktur

Dieser Abschnitt umfasst die Definitionsklärung, eine Abgrenzung des Begriffs Framework, die wichtigsten Frameworkeigenschaften, das Hollywood-Prinzip und die Erläuterung der Frameworkbestandteile.

2.1.1.1. Definition

Die ersten Frameworkansätze erschienen in der Informatik bereits vor mehreren Jahrzehnten. Aus diesem Grund wurde ebenso die Definition „Framework“ bereits in der Anfangszeit festgelegt. Da der Begriff länger existiert und sich zudem sehr gut etabliert hat, sind aktuelle Definitionen für diesen Begriff schwer oder gar nicht zu finden. Es ist lediglich eine Vielzahl

an Erläuterungen von speziellen Frameworks vorhanden. Infolgedessen wird auf ältere Definitionen zurückgegriffen, die aber nach wie vor Gültigkeit haben. Nachfolgend sei eine weit verbreitete und oft zitierte Definition, gefolgt von zwei weiteren, genannt:

„A framework is a set of classes that embodies an abstract design for solutions to a family of related problems.“ [Johnson und Foote (1988)]

„A framework consists of a large structure that can be reused as a whole for the construction of a new system.“ [Bosch u. a. (1997)]

„Ein Rahmenwerk (engl. Framework) ist ein objektorientiertes abstraktes Entwurfsmuster für die generische Lösung einer Problemfamilie aus einem bestimmten Kontext. Es besteht aus mehreren kollaborierenden Klassen und bildet ein nahezu vollständiges Programm. Zusammen stellen die Klassen einen wiederverwendbaren Entwurf für eine bestimmte Art von Software dar.“ [Gamma u. a. (2001)]

Die Autorin fasst die Frameworkdefinition (die in dieser Arbeit auch so zur Anwendung kommt), mit Hilfe der oben erwähnten Erläuterungen wie folgt zusammen:
Frameworks dienen im Bereich der Informatik zur Unterstützung der Wiederverwendung. Ein Framework besteht aus einer wiederverwendbaren Architektur und Implementierung, welche kooperativ ein Grundgerüst für Anwendungen bildet, die gemeinsame Problematiken lösen. Durch die Erweiterung bzw. Spezifizierung wird aus dem Grundgerüst eine individuelle abgeschlossene Anwendung.

2.1.1.2. Abgrenzung

Es existieren in der Informatik einige Begriffe, die ähnliche Merkmale wie das Framework aufweisen und oftmals genauso definiert werden wie der Begriff Framework. Aus diesem Grund soll an dieser Stelle eine Abgrenzung von Frameworks zu Komponenten, Entwurfsmustern (Patterns) und Klassenbibliotheken erfolgen.

Frameworks vs. Komponenten
Laut [Schmitz (2004)] ist eine Komponente lediglich ein Teil einer Anwendung, die in den meisten Fällen ausschließlich einen fachlichen Aspekt abbildet. Zudem ist im Gegensatz zu Frameworks keine Architektur vorgeschrieben. Ein Framework umfasst, wie in der Definition bereits erwähnt, eine adäquate Architektur, die bei der Frameworkerweiterung eingehalten bzw. gepflegt werden sollte.
[Johnson (1997)] grenzt Frameworks zu Komponenten zusätzlich in weiteren Bereichen ein. Frameworks sind mit einer höheren Anzahl an Schnittstellen ausgestattet, die dadurch eine hohe Flexibilität mit sich bringen. Durch die hohe Anpassungsfähigkeit des Frameworks kann besser auf die Wünsche und Bedürfnisse der Nutzer eingegangen werden. [Johnson

(1997)] ist ebenso wie [Schmitz (2004)] der Meinung, dass Komponenten lediglich einen fachlichen Aspekt abbilden. Des Weiteren sind lose gekoppelte und kohärente Komponenten Bestandteil eines Frameworks und das eigentliche Framework ist zuständig für die individuellen Anpassungen sowie Erweiterungen.

Frameworks vs. Entwurfsmuster (Patterns)

Die wesentlichen Unterschiede zwischen Frameworks und Entwurfsmustern können in drei Bereiche unterteilt werden [Schmid (1997)] [Mayer (1999)] [Johnson (1997)]:

1. Abstraktion: Entwurfsmuster sind im Gegensatz zu Frameworks abstrakt. Die Wiederverwendung eines Entwurfsmusters bezieht sich lediglich auf die Wiederverwendung einer Idee. Ersteres bei jeder Verwendung erneut implementiert werden. Im Vergleich zum Framework ist nicht ausschließlich die Idee Bestandteil der Wiederverwendung sondern ebenfalls die Implementierung.

2. Granularität: Ein Entwurfsmuster ist ein Lösungsansatz für genau eine Problematik. Ein Framework befasst sich hingegen mit einer Vielzahl von Problematiken. Es wird davon gesprochen, dass ein Framework viele Entwurfsmuster enthält. Demzufolge kann konstatiert werden, dass ein Entwurfsmuster eine Mikro-Architektur besitzt, die die Struktur der Architektur eines Framework beeinflusst. Dies bedeutet zusammen gefasst, dass die Problematiken eines Frameworks in Teilprobleme unterteilt werden können und Mithilfe von Entwurfsmustern gelöst werden.

3. Spezifizierung: Entwurfsmuster können in nahezu allen Anwendungsbereichen angewandt werden. Der Einsatz von Frameworks ist hingegen in den meisten Fällen auf bestimmte Anwendungsbereiche begrenzt.

Frameworks vs. Klassenbibliotheken

Der wesentliche Unterschied zwischen Frameworks und Klassenbibliotheken liegt darin, dass Klassenbibliotheken passiv sind. Das bedeutet, dass Sie keinen Kontrollfluss besitzen. Des Weiteren verfügen Klassenbibliotheken über kein Standardverhalten und es existiert keine Definition über (Object-) Kooperation von Funktionen, Objekten oder Paketen. Eine Anpassung der Klassenbibliotheken in die Anwendung erfolgt lediglich über die Klasseninstanziierung oder über den Aufruf von Funktionen. Daraus leitet sich ab, dass ein Framework einen Kontrollfluss besitzt; ein Standard- oder Musterverhalten, die (Objekt-)Kooperation von Funktionen, Objekten und Paketen ist definiert, und für die Anpassung sind mehrere Möglichkeiten vorhanden [Mayer (1999)]. Weitere Anpassungs- bzw. Erweiterungsmöglichkeiten werden in den nachfolgenden Kapiteln genauer erläutert.

2.1.1.3. Hollywood-Prinzip

Bei der Frameworkabgrenzung haben sich einige Eigenschaften von Frameworks bereits herauskristallisiert. An dieser Stelle wird erneut auf den Kontrollfluss eingegangen. Ein Framework beinhaltet eine vordefinierte Architektur, die die Umkehr des Kontrollflusses beinhaltet. Der Kontrollfluss wird vom Framework übernommen im Gegensatz zu Klassenbibliotheken und Komponenten, die bei Bedarf explizit aufgerufen werden müssen. Die Eigenschaft ist unter dem Namen „Hollywood-Prinzip“ geläufig. Die Namensgebung leitet sich aus dem in Hollywood üblichen Slogan „Don’t call us, we’ll call you“ ab. Dieses Grundprinzip bringt den entscheidenden Aspekt auf den Punkt. Durch die Umkehr des Kontrollflusses ist das Framework die aufrufende Instanz [Schmitz (2004)] [Fayad (2000)].
Durch den Kontrollfluss auf der Frameworkseite tritt eine wesentliche Problematik auf, die an dieser Stelle lediglich nur angerissen werden kann. Lösungsansätze oder ähnliches werden in dieser Masterarbeit nicht behandelt. Heutzutage wird mehr als ein Framework zu einer Anwendung gebündelt. Die Folge der Frameworkzusammenführung ist, dass jedes Framework einen Kontrollfluss besitzt, und diese in Konflikt zueinander geraten können. Demzufolge können erheblich Probleme auftreten.
Zusammengefasst ist eine der Haupteigenschaften eines Frameworks das Verkörpern des Hollywood-Prinzips. [Sparks u. a. (1996)] vertritt jedoch einen anderen Standpunkt. Er ist der Meinung, dass ein Framework nicht zwangsläufig das Hollywood-Prinzip befolgen muss. Diese Frameworkvariante wird als passiv bezeichnet und mit dem Namen „called framework“ versehen. In dieser Masterarbeit wird nicht die Meinung von [Sparks u. a. (1996)] vertreten, sondern Frameworks werden als eine aktive Instanz gesehen, die gemäß dem Hollywood-Prinzip arbeiten.

2.1.1.4. Bestandteile eines Frameworks

Ein Framework besteht aus genau zwei Bestandteilen, den Hot- und Frozen-Spots.
Die flexiblen Bereiche und die dynamischen Anpassungspunkte werden als Hot-Spots bezeichnet, die statischen Bereiche dagegen als Frozen-Spots. Wie bereits in der Frameworkdefinition und Frameworkabgrenzung ausgeführt, sind Frameworks nicht abgeschlossen. Infolgedessen müssen Frameworks um spezifische Anwendungsdetails erweitert werden, um ausführbar zu sein. Die Hot-Spots bilden diese Stellen im Framework ab, an denen die Anpassungen und Erweiterungen realisiert werden können. Es existiert eine Vielzahl an Möglichkeiten, Anpassungen und Erweiterungen durchzuführen. Eine Variante davon ist die Benutzung vordefinierter parametrisierter Klassen oder Komponenten. Eine weitere Variante ist die Neuimplementierung von Funktionen, die durch abstrakte Klassen oder Interfaces vorgegeben sind [van Grup und Bosch (2001)]. Aktuell existieren noch weiterte Ansätze für die Anpassung eines Frameworks. Anpassungen können z.B. Mithilfe von konfigurierbaren

Dateien getätigt werden. Exemplarisch sei Spring [Spring] genannt; bei diesem Applikationsframework bilden XML-Dateien die Hot-Spots. Jede beliebige Erweiterung und Anpassung von Spring ist lediglich durch XML-Elemente konfigurierbar.

2.1.2. Anforderungen an Frameworks

Es existiert eine Reihe guter und schlechter Softwaresysteme, und dieser Sachverhalt spiegelt sich auch bei den Frameworks wieder. Um eine qualitativ hochwertige Software zu erzielen, werden viele Anforderungen an das System gestellt. Die Anforderungen beziehen sich z.B. auf Zuverlässigkeit, Benutzerfreundlichkeit, Wartbarkeit, Effizienz usw. und ihrem Zusammenspiel. Da ein Framework ebenfalls ein Softwaresystem ist, lediglich ein spezielles, sollte es die allgemeinen Software-Qualitäts-Anforderungen bzw. Kriterien erfüllen. Diese Anforderungen sind die Basis eines guten Frameworks. Für das spezielle Softwaresystem Framework existieren noch einige zusätzliche Anforderungen. In diesem Unterkapitel wird auf einige wesentliche eingegangen. Es werden ebenso Anforderungen vorgestellt, die bei Softwaresystemen grundsätzlich eingehalten werden sollten, die jedoch im Zusammenhang mit einem Framework noch weitere Hintergründe haben.

Hot-Spots-Definition
Die richtige Anzahl an Hot-Spots macht ein qualitativ hochwertiges Framework aus. Besitzt ein Framework eine zu große Menge an Hot-Spots, besteht die Gefahr, dass das Framework zu komplex wird, und die Entwicklungs- sowie Wartungskosten eine akzeptable Summe übersteigen. Eine zu stark eingegrenzte Hot-Spots-Definition kann aufgrund fehlender Flexibilität zur Unbrauchbarkeit eines Frameworks führen. Nicht nur die Anzahl an Hot-Spots ist entscheidend sondern auch die Wahl der richtigen Anknüpfungspunkte. Eine Spezifizierung an ungeeigneten Stellen ist ineffektiv.

Verständlichkeit
Frameworks dienen hauptsächlich der Wiederverwendung. Naturgemäß kann nur etwas wiederverwendet werden, wenn es durch Verständlichkeit zur Wiederverwendung einlädt. Aus diesem Grund ist eine wesentliche Anforderung an ein Framework seine Verständlichkeit. Die Verständlichkeit eines Systems ist nicht nur der Dokumentation zu verdanken, wobei diese ein wichtiger Bestandteil ist und als nächstes behandelt wird. Weitere Möglichkeiten, ein System bzw. ein Framework begreiflich zu gestalten, ist die Verwendung von klaren Konzepten, welche konsequent verfolgt werden. Der Einsatz von Entwurfsmustern, die in der Praxis bewährte Lösungen vertreten, kann ebenfalls einem besseren Verständnis dienen. Im Grunde können alle Methodiken und Techniken eingesetzt werden, die die Verständlichkeit eines System fördern.

Dokumentation
Die Dokumentation innerhalb eines Frameworks spielt eine sehr wichtige Rolle, wie grundsätzlich bei jedem Softwaresystem. In der Realität sieht das leider anders aus, durch Zeit- und Geldmangel wird in den meisten Fällen die Dokumentation vernachlässigt. Diese Problematik sollte sich jedoch auf keinen Fall bei der Dokumentation von Frameworks widerspiegeln.
Eine ungenügende Dokumentation hat zur Folge, dass ein Framework nicht zweckmäßig oder gar nicht verwendet werden kann, falls nicht die entsprechenden Informationen über Bedienbarkeit etc. vorhanden sind. Laut [Schmitz (2004)] sollte eine Dokumentation eines Frameworks drei wesentliche Bereiche abdecken:

1. Funktionalität: Diese Dokumentation sollte alle notwendigen Informationen über Funktionalitäten und Schnittstellen sowie aussagekräftige Beispiele bereitstellen, um den Frameworkanwendern ausreichende Unterstützung zu bieten.

2. Qualität: Eine Qualitätsbeschreibung beinhaltet Angaben über Testergebnisse, Testmethoden und Kriterien, die auf die Tests angewendet werden.

3. Technische Anforderungen: Der technische Dokumentationsbereich liefert alle Informationen über Anforderungen, die an die Hardware gestellt werden. Zudem wird beschrieben, ob weitere Softwaresysteme oder Subsysteme für die Verwendung des Frameworks benötigt werden oder vielleicht eine gute Ergänzung bieten.

Entscheiden sich Frameworkentwickler für anderweitige Dokumentationsvarianten, für den Informationsaustausch, sollte ungeachtet dessen jede Dokumentation immer **vollständig** und **konsistent** sein. Eine Dokumentation muss vollständig sein, um den gesamten Frameworkumfang nutzen zu können. Die Konsistenz einer Dokumentation spielt ebenfalls eine sehr wichtige Rolle. Durch den Evolutionären Ansatz bei der Frameworkentwicklung, unterliegen Frameworks Modifikationen. Es ist sehr wichtig, dass Änderungen und Erweiterungen auch in der Dokumentation konsistent vermerkt werden.

Effizienz
Die Performance und der Speicherverbrauch sind nicht zu unterschätzende Anforderungen, insbesondere bei Frameworks. Wie bereits erwähnt, werden heutzutage mehrere Frameworks in eine Anwendung integriert. Demzufolge steigt nicht nur die Wiederverwendbarkeit, sondern auch die Performance- und Speicherauslastung. Aus diesem Grund sollte immer die Effizienz im Hintergrund stehen. Hierbei sind hauptsächlich Frameworks in den Bereichen Persistenz, Kommunikation usw. betroffen.

Angemessenheit
Bei der Entwicklung eines Softwaresystem muss überlegt werden, welcher Umfang realisiert werden soll. Diese Überlegung bei der Entwicklung eines Frameworks spielt eine extrem

große Rolle. Der Einsatz einer umfangreicheren Software, die nicht den vollen Funktionsbereich ausschöpft, kann durchaus auch für trivialere Aufgaben genutzt werden. Jedoch wird kein komplexes Framework für die Erstellung einer einfachen Anwendung in Frage kommen. Aus diesem Grund muss im voraus geklärt werden, welchen Umfang das Framework annehmen soll.

Die oben vorgestellten Anforderungen sind kein Geheimrezept für die Entwicklung eines qualitativ hochwertigen Framework, allerdings dienen sie als ein praktikabler Wegweiser [Maier].

2.1.3. Klassifikationen

Frameworks können unterschiedlichen Klassifikationen zugeordnet werden. Die Klassifikationen werden anhand von Unterscheidungsmerkmalen strukturiert. Nachfolgend werden die geläufigsten Klassifikationen kurz angeschnitten.

2.1.3.1. Klassifizierung nach Spezifizier- bzw. Erweiterbarkeit

Wie bereits erwähnt, wird durch die Erweiterung eines Frameworks eine Anwendung realisiert. Die Erweiterungen können auf unterschiedlicher Art und Weise erfolgen. Aus diesem Grund existiert eine Klassifikation in diesem Bereich laut [Birrer u. a. (1995)] [Schmitz (2004)].

White-Box Framework:
Bei einem White-Box Framework wird die Spezifizierung durch das Überschreiben von Methoden in Form von Interfaces oder abstrakten Klassen durchgeführt. Diese Erweiterungsmöglichkeit bietet eine sehr hohe Flexibilität. Jedoch wird für den Gebrauch dieser Methodik ein umfangreiches Wissen über das interne Verhalten und Zusammenspiel vorausgesetzt.

Black-Box Framework:
Wie der Name bereits andeutet, ist bei dieser Variante eine Kenntnis des internen Verhaltens und des Zusammenspiels nicht notwendig. Im Gegensatz zum White-Box Konzept gestaltet sich die Bedienbarkeit einfacher, jedoch auch unflexibler.

Gray-Box Framework:
Generell handelt es sich in der frühen Entwicklungsphase um ein White-Box Framework, das anschießend durch die Evolutionäre Entwicklung zu einem Black-Box Framework mutiert. In dieser Zwischenphase wird von einem Gray-Box Framework gesprochen, das beide Arten

vereint. In einigen Fällen verfügt ein Framework dauerhaft über beide Erweiterungsmöglichkeiten und ist dementsprechend nicht nur in der Entwicklungsphase ein Gray-Box Framework. Derartige Frameworks genießen somit die Vorteile beider Variationen: die Flexibilität bei ausreichenden Kenntnissen und die Einfachheit der schnellen Nutzung.

2.1.3.2. Klassifizierung nach Ausdehnungsebene

Frameworks können spezifisch auf bestimmte Fachgebiete ausgelegt sein oder allgemeine Probleme lösen. Dementsprechend wird ebenso laut [Mayer (1999)] gemäß der Ausdehnungsebene eine Unterteilung vorgenommen.

Horizontales Framework:
Diese Arten von Frameworks sind auf allgemeinere Fachbereiche ausgelegt. Sie stellen ein breites Spektrum an Funktionalitäten zur Verfügung und können dementsprechend in verschiedenen Bereichen eingesetzt werden.

Vertikales Framework:
Diese Klassifikation begrenzt sich auf ein spezielles Problemgebiet. Sie bietet Funktionalitäten für gezielte Domänen, wie z.B. das Finanzwesen, an.

2.1.3.3. Weitere Klassifizierungen

Es existiert eine Reihe weiterer Klassifizierungen. [Fayad (2000)] gliedert nach „System infrastructure frameworks“, „Middleware integration frameworks“ und „Object-oriented enterprise frameworks“. [van Grup und Bosch (2001)] dagegen nach „Application frameworks“, „Domain frameworks“ und „Support frameworks“. Die Beschreibung aller Klassifizierungen würde den Rahmen dieser Masterarbeit sprengen. Aus diesem Grund erfolgte lediglich eine Auflistung, für weitere Informationen wird auf die einzelnen Veröffentlichungen verwiesen.

2.1.4. Entwicklung & Vorgehensweise

Es ist um ein vielfaches komplexer, ein qualitativ hochwertiges Framework zu realisieren als ein komplexes Softwaresystem.
Nachfolgend werden einige Punkte angeschnitten, die den Unterschied und die zusätzliche Problematik einer Frameworkentwicklung verdeutlichen soll.

- Ein Framework dient als Gerüst vieler Anwendungen. Aus diesem Grund bildet bei der Frameworkentwicklung eine Hauptaufgabe die Modellierung der Abstraktion. Diese Aufgabe erfordert ein profundes Fachwissen und sollte nur von qualifizierten Personen durchgeführt werden. Einige Unternehmen realisieren Ihre Frameworks basierend auf vorhandenen Anwendungen. Dies stellt eine wesentliche Hilfe bei der Abstraktionsmodellierung dar.

- Wie bereits erwähnt, ist die Definition der Hot-Spots problematisch; die richtigen Anknüpfpunkte und die geeignete Anzahl zu bestimmen, stellt eine Herausforderung an die Entwickler dar, die bei der gewöhnlichen Softwareentwicklung nicht zu beobachten ist.

- Für die Entwicklung eines Frameworks ist ein profundes Fachwissen erforderlich. Die Realisierung eines herkömmlichen Softwaresystems setzt ebenfalls ein ausreichendes Fachwissen voraus. Jedoch muss bedacht werden, dass ein Framework der Erstellung einer Vielzahl von Anwendungen dienen soll.

Durch die oben erwähnte Komplexität wird die evolutionäre Vorgehensweise bevorzugt. Es stellt sich als nahezu unmöglich dar, alle Anforderungen und Funktionalitäten fehlerfrei nach nur einem Durchlauf zu erfüllen. [Martin u. a. (1997)] stellen dazu treffend fest: *„People develop abstractions by generalizing from concrete examples. Every attempt to determine the correct abstractions on paper without actually developing a running system is doomed to failure. No one is that smart“.*
[Schmitz (2004)] unterteilt das evolutionäre Vorgehensmodell in drei Phasen.

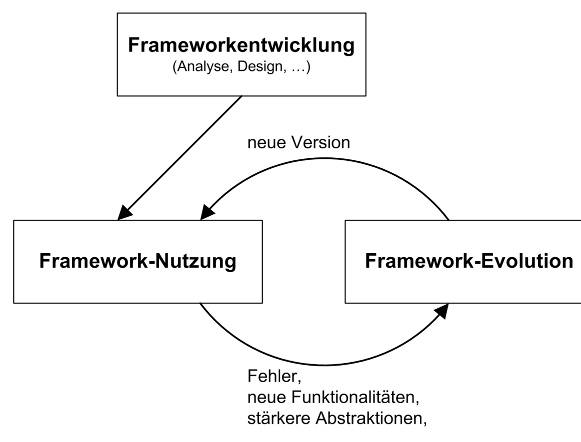

Abbildung 2.1.: Evolutionäres Vorgehensmodell [Schmitz (2004)]

Die erste Phase bildet die gesamte Entwicklung ab. Die Verwendung des Frameworks charakterisiert die zweite Phase. Im Anschluss folgt die Wartung und Entwicklung des Frameworks, Phase drei. An dieser Stelle ist die Entwicklung jedoch nicht abgeschlossen. Es exis-

tiert ein Zyklus, ein kontinuierlicher und reproduktiver Kreislauf.
[Maier] und [Bosch u. a. (1997)] sind ebenfalls der Meinung, dass das evolutionäre Vorgehensmodell sich am besten für die Frameworkentwicklung eignet. Im Gegensatz zu [Schmitz (2004)] gliedern Sie den gesamten Prozess in fünf Phasen. Die Aufteilung der fünf Phasen erfolgt bei [Maier] abweichend von [Bosch u. a. (1997)]. Das Prinzip des evolutionären Ansatzes und des konventionellen Entwicklungshergangs ist jedoch bei allen drei Beschreibungen ähnlich.

2.2. Monitoring

In den Grundlagen zum Monitoring wird zu Beginn eine Definition gegeben, gefolgt von einer Monitoring Klassifikation.

2.2.1. Definition

In dieser Masterarbeit wird das Erkennen bzw. Entdecken von Softwarefehlern in Java- Anwendungen als Monitoring bezeichnet. Der Begriff „Monitoring“ beschränkt sich nicht auf diese spezifische Aufgabenstellung und wird aus diesem Grund an dieser Stelle abstrakter definiert. In der IT-Welt ist der Begriff „Monitoring“ weit verbreitet, und es ist keine Seltenheit gebräuchliche Termini homonym zu verwenden. Aus diesem Grund werden drei Definitionen dargestellt, die auch als solche in dieser Masterarbeit angewendet werden. Als erstes wird die Definition aus Wikipedia vorgestellt, die für die Autorin eine der am besten zutreffenden Erläuterungen ist.

„Monitoring ist ein Oberbegriff für alle Arten der unmittelbaren systematischen Erfassung (Protokollierung), Beobachtung oder Überwachung eines Vorgangs oder Prozesses mittels technischer Hilfsmittel oder anderer Beobachtungssysteme. “ [Enzyklopädie (2010)]

„Softwaremonitore sind Programme die in regelmäßigen Zeitabständen wie Benutzerprogramme auf der zu beobachtenden Rechenanlage ausgeführt werden und dabei deren Verhalten protokollieren.“ [Claus und Schwill (1993)]

„A functional unit that observes and records selected activities within a data processing system for analysis.“ [Csikai (1985)]

2.2.2. Klassifikation

Es existieren unterschiedliche Monitoring Klassifikationen, die in der Regel auf sehr spezielle Monitoringsysteme ausgelegt sind. [Delgado u. a. (2004)] stellen eine Klassifikation für Software-Fault Monitoringsysteme vor. Anhand dieser Klassifikation können Monitoringsysteme zur Fehlererkennung, die Bestandteil dieser Masterarbeit sind, sehr detailliert beschrieben werden. An dieser Stelle wird im Detail die Klassifikation von [Delgado u. a. (2004)] erläutert. Als Erstes folgt eine Übersicht über alle Kategorien in Form eines Baums, gefolgt von einer schriftlichen Beschreibung jeder Verzweigung bis hin zum Blatt.

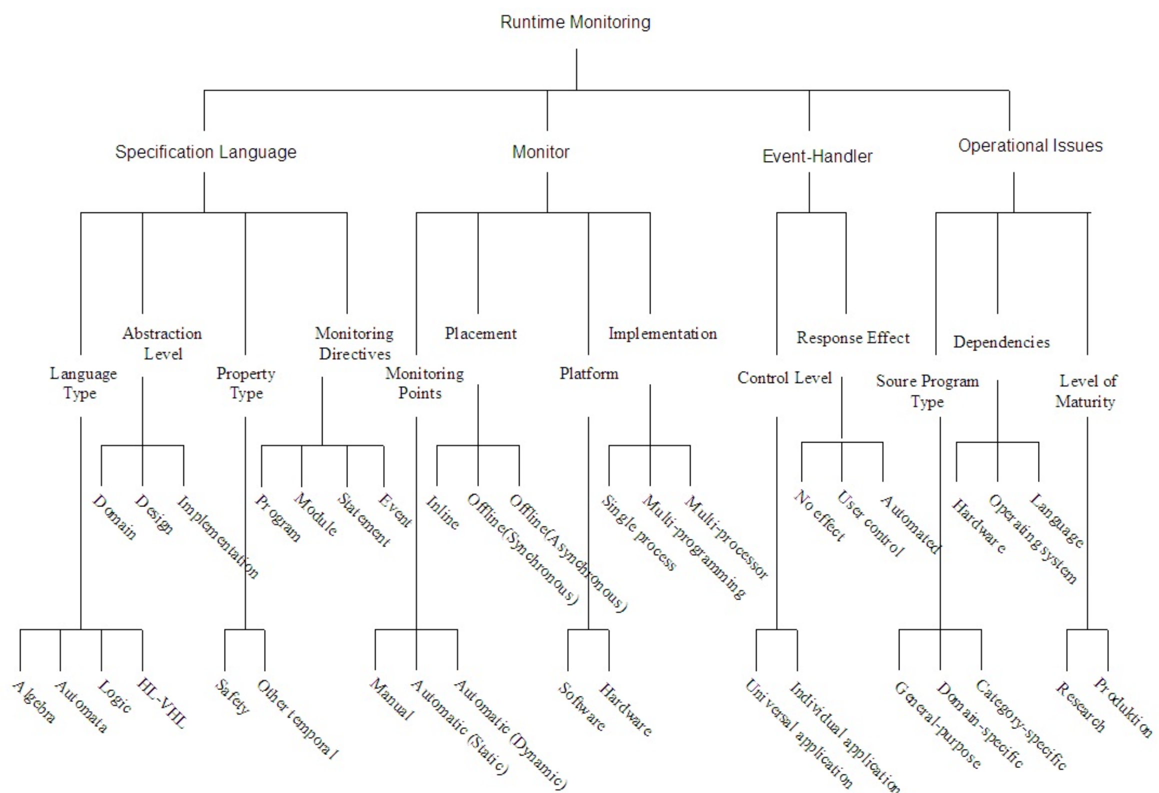

Abbildung 2.2.: Monitoring Klassifikation

2.2.2.1. Spezifikationssprache

Die Verzweigung Specification Language stellt die Klassifizierung der Sprachebene dar. Mit Hilfe der Spezifikationssprache können Eigenschaften der Zielanwendungen für die Laufzeitmonitore spezifiziert werden.

Language Type (Sprachtypen):
Es wird zwischen vier verschiedenen Sprachtypen unterschieden, die zur Spezifikation von Eigenschaften verwendet werden können. Als erstes können Eigenschaften mittels Algebra definiert werden, die einen fundamentalen Teilbereich der Mathematik darstellt. Des Weiteren können Automaten eingesetzt werden, mit denen unter anderem Verhalten sowie Zustände dargestellt werden. Eine weitere Alternative stellt die Logik dar, die die Temporallogik, Prädikatenlogik etc. umfasst. Bei dem letztgenannten Sprachtyp handelt es sich um HL(High-Level)- VHL(Very-High-Level) Programmiersprachen.

Abstraction Level (Abstraktionsebene):
Es besteht die Möglichkeit, auf unterschiedlichen Abstraktionsebenen Eigenschaften zu spezifizieren. Aus diesem Grund existiert die Klassifikation Abstraktionsebene. Sprachen, die Domän- bzw. Fachbereich-spezifische Eigenschaften oder Informationen beschreiben, werden in der Ebene Domäne gegliedert, Design-bezogene Spezifikationen bezüglich des Softwaredesign in der Design-Ebene. Die letzte Abstraktionsebene ist die Implementierungsebene, die die Spezifikation von Implementierungsabhängigen Eigenschaften darstellt.

Property Type (Eigenschaftstypen):
Die Klassifikation Eigenschaftstypen bestehen aus zwei konkreten Klassen. Zum einen existieren Eigenschaften vom Type Safety, zum anderen solche vom Type Other Temporal. Eigenschaften vom Typ Safety drücken explizit aus, welche Zustände oder welches Verhalten nicht auftreten darf bzw. welche Zustände und welches Verhalten vorkommen dürfen. Dieses kann unter anderem mit Hilfe von Invarianten dargestellt werden sowie mit Hilfe einer Definition von Prüfwerten. Eigenschaften vom Typ Other Temporal beziehen sich eher auf zeitliche Aspekte, wie zum Beispiel eine Eigenschaft, die ausdrückt, dass ein Ereignis innerhalb von zehn Sekunden auftreten muss. Beide Eigenschaftstypen stehen nicht disjunkt zueinander.

Monitoring Directives (Überwachungsrichtlinien):
Eigenschaften können auf unterschiedlichen Ebenen innerhalb eines Programms festgelegt werden: Programm, Modul, Anweisung, Ereignis. Eigenschaften auf der Programmebene werden für das gesamte Programm definiert. Ein Beispiel ist eine Programminvariante, die für alle Zustandsänderungen in einem Programm gelten muss. Modulbasierte Eigenschaften spezifizieren Funktionen, Prozeduren, Methoden oder Komponenten, wie abstrakte Datentypen oder Klassen. Beispiele für Eigenschaften der Klasse Modulebene beinhalten Klasseninvarianten und Vor- und Nachbedingungen von Prozeduren und Methoden. Anweisungseigenschaften (Statement Properties) sind Eigenschaften, die Anweisungen spezifizieren. Die Eigenschaften der Ereignisebene basieren auf Zustandsänderungen oder Folgen von Zustandsänderungen.

2.2.2.2. Monitor

Der Monitor-Zweig repräsentiert die Klassifizierung des Monitorbereichs. Mithilfe dieser können Eigenschaften bezüglich der Überwachung spezifiziert werden.

Monitoring Points (Überwachungsstellen):
Die Monitoring Points bilden die Stellen im Code, die überwacht werden. Sie können manuell oder automatisiert gesetzt werden. Des Weiteren existiert bei der automatisierten Monitoring Pointsetzung eine Unterteilung in statisch und dynamisch. Werden die Monitoring Points vor der Programmausführung automatisiert bestimmt, handelt es sich um statische, automatisierte Monitoring Points. Erfolgt jedoch die Monitoring Points Wahl zur Laufzeit, werden diese als dynamische, automatisierte Monitoring Points bezeichnet.

Placement (Platzierung):
Bei einem Monitoring Systems kann unterschieden werden, an welcher Stelle der Monitoring-Code ausgeführt werden soll. Einerseits kann der Monitoring-Code in dem Ziel-Code eingebettet sein und somit im überwachendem System ausgeführt werden, bzw. nutzt dessen Ressourcen. Andererseits kann ein Monitoring-System „offline“ überwachen, das bedeutet, es verwendet einen eigenen Thread, Prozess oder sogar eine andere Maschine. Ein offline laufendes Monitoring kann zudem asynchron oder synchron überwachen. Bei der synchronen Überwachung wartet das laufende Programm, bis der Monitoring Check beendet ist, anders als es bei der asynchronen Überwachung der Fall ist.

Platform (Plattform):
Bei der Plattform wird unterschieden zwischen dem Überwachen einer Hard- oder Software.

Implementation (Implementierung):
Für die Implementierung werden drei Klassen vorgestellt:

1. Single-Prozess; das Überwachen wird im selben Prozess wie das Zielprogramm ausgeführt.
2. Multiprogramming; das Überwachen verwendet einen eigenen Prozess oder Thread, wobei der gleiche Prozessor verwendet wird.
3. Mulitprocessor; es existiert ein separater Prozessor für das Überwachen und ein anderer für die Zielanwendung.

Die beiden letztgenannten wurden bereits als offline in Placement klassifiziert.

2.2.2.3. Behandeln von Ereignissen

Die Art und Weise, wie ein Monitoringsystem auf erkannte Fehler reagiert, wird in dieser Klassifikation (engl.: Event-Handler) abgedeckt.

Control Level (Kontrollebene):
Die Kontrollebene unterscheidet zwischen der individuellen und der universellen Reaktion. Monitoringsysteme, die eine universelle Reaktion unterstützen, reagieren auf einen Fehler immer mit derselben Reaktion. Ein Beispiel für eine universelle Reaktion ist ein Logger, der jeden erkannten Fehler protokolliert. Im Gegensatz dazu steht die individuelle Reaktion, die es möglich macht, individuelle Reaktionen auf unterschiedliche Ereignisse zu spezifizieren.

Response Effect: (Reaktionswirkung):
Es existieren drei Klassen dieser Kategorie, die zum Ausdruck bringen, welche Möglichkeiten bestehen, auf erkannte Fehler zu reagieren, und wie das Verhalten beeinflusst werden kann. Ist die einzige Reaktion auf einen auftretenden Fehler die Meldung oder die Erstellung eines Traces, ist von der Klasse No effect die Rede. Beispiele für Systeme, die als User control klassifiziert werden, sind Systeme, die eine technische Möglichkeit bieten, erforderliche Benutzerinteraktionen mit dem System durchzuführen, wenn ein Fehler entdeckt wird. Bei der letztgenannten Klassifikation handelt es sich um Automated. In diese Kategorie werden Reaktionen, wie saubere Degradierungen, Speicher-Rollbacks, Exception Handling, automatisierte Durchführung der Programmsteuerung, etc. beschrieben.

2.2.2.4. Operationale Fragen

Diese Kategorie befasst sich nicht direkt mit dem Monitoringsystem, sondern mit den externen Einflüssen.

Source Program Type (Programmarten):
Monitoringsysteme sind für unterschiedliche Arten von Programmen ausgelegt, die in dieser Kategorie beschrieben werden. Die Klassifikation beinhaltet Allgemeinzweck-, Domän-, und Kategoriespezifische Monitoringsysteme. Allgemeinzweck-spezifische Monitoringsysteme sind für Zielprogramme ausgelegt, die allgemeine Zwecke erfüllen. Diese Art von Monitoringsystemen verwendet keine fachspezifischen Informationen. Domänspezifische Monitoringsysteme beziehen sich auf ein bestimmtes Umfeld und enthalten Informationen über die entsprechende Domäne, die das Monitoringsystem für die Erfüllung seiner Dienste nutzt. Exemplarisch seien Monitoringsysteme für Kernkraftwerke oder Flugverkehrskontrollen zu nennen. Kategoriespezifische Monitoringsysteme sind ausgerichtet auf Programme, die einer speziellen Klasse angehören, wie z.B. Echtzeitprogramme oder verteilte Systeme.

Dependencies (Abhängigkeiten):
Die Abhängigkeiten werden in Hardware, Betriebssystem und Sprache eingeteilt. Einige Monitoringsysteme setzen eine spezielle Hardware voraus, können nur auf bestimmten Betriebssystemen operieren und/oder können nur Zielprogramme überwachen, die in einer vordefinierten Sprache geschrieben sind.

Level of Maturity (Reifegrad):
Der Reifegrad definiert den Produktionsstatus eines Monitoringsystems. Ein Monitoringsystem, welches sich in einem sehr frühen Realisierungsstadium befindet, wird in der Kategorie Research zusammengefasst. Im besten Fall existiert ein Prototyp, andernfalls sind nur Ansätze und Konzepte vorhanden. Monitoringsysteme der Kategorie Produktion umfassen ausgereifte Systeme.

2.3. Debugging

Ein Teil des Titels dieser Masterarbeit ist das Lokalisieren und indirekt das Beheben von Fehlern, der im Begriff Debugging zusammen gefasst wird. Aus diesem Grund beschäftigt sich dieses Kapitel mit den Debugging Grundlagen, beginnend mit der Definition. Anschließend wird näher auf das Themengebiet Fehler eingegangen, das einen wichtigen Bestandteil des Debuggings darstellt. Zum Abschluss wird dem Leser ein Einblick in Debugging Methodiken gegeben, indem auf vier populäre Methoden kurz eingegangen wird.

2.3.1. Definition

Zur Begriffserläuterung werden zwei Debuggingdefinitionen zitiert.

„Debugging is the process of removing defects from computer programs.“ [Chmiel und Loui (2004)]

„Debugging is centered on two main aspects, namely the observation of the computation state, and the need to exercise some modification and control upon the computation, to identify and correct program bugs.“ [Cunha u. a. (1998)]

Die Autorin fasst die Debuggingdefinition folgendermaßen zusammen: Der Begriff „Debugging“ umfasst den gesamten Prozess der Identifizierung bzw. Lokalisierung und Behebung von Programmfehlern.

Der Begriff „Debugging“ hat seinen Ursprung im englischen Begriff „Bug“. Als ein Bug wird im Bereich der Informatik ein Programmfehler verstanden. Diese Namensgebung prägte Prof.

Grace Hopper [Cleve und (Hrsg.)]. Heutzutage werden Bugs unter weiteren Synonymen geführt; im Anschluss wird auf diese Thematik näher eingegangen.

2.3.2. Fehlerbezeichnungen

In der Informatik sorgt der Begriff Fehler oder Programmfehler für Uneindeutigkeiten. Dies gilt ebenfalls für das oben erwähnte Synonym Bug. Es kommt häufig vor, dass ein Begriff synonym verwendet wird oder viele Begriffe homonym. Um dieser Problematik entgegen zu wirken, werden nachfolgend einige Termini näher erläutert.
Laut [Zeller (2009)] können Programmfehler in drei Stufen auftreten. Ein inkorrekter Code kann zu einem inkorrekten Programmzustand führen und dieser wiederum ein inkorrektes Programmverhalten hervorrufen.
Defekt ist der inkorrekte Code (engl. defect)
Ein Defekt liegt bei einer falschen Implementierung vor; bzw. dem Fall, dass ein Code zwar richtig implementiert ist, den Anforderungen jedoch nicht genügt.
Infektion bezeichnet den inkorrekten Programmzustand (engl. infection)
Das Ausführen eines Programms mit einem Defekt kann zu einer Infektion führen. Dabei ist zu beachten, dass ein Defekt nicht immer eine Infektion zur Folge hat. Eine Infektion kommt erst zustande, wenn bestimmte Bedingungen erfüllt sind. Eine Infektion führt zu einem vom Anwender/Entwickler unerwarteten Zustand.
Störfall vermerkt das inkorrekte Programmverhalten (engl. failure)
Die Infektion verursacht einen failure. Der Störfall macht sich durch einen Fehler im Programmverhalten bemerkbar. Dies ist ein von außen sichtbarer Fehler z.B., eine fehlerhafte Ausgabe für den Anwender.

Ein weiterer häufig verwendeter Begriff in der Literatur ist fault. Fault entspricht nach [IEEE (1990)] dem defect von [Zeller (2009)]. [Zeller (2009)] benutzt noch einen weiteren Begriff, den der **flaws**. Flaws sind architektonische Programmfehler bzw. Programmfehler, die durch die Laufzeitumgebung verursacht werden. Ein weiterer Begriff, **error**, der den Unterschied zwischen dem gemessenen und dem zu erwartenden Wert bzw. dem Zustand definiert, [IEEE (1990)] entspricht dem von [Zeller (2009)] definierten Begriff der Infektion.
Im Anschluss folgt eine tabellarische Zusammenfassung der Fehlerbezeichnungen einschließlich der Referenzen auf die Synonyme.

In dieser Masterarbeit werden die Begriffsdefinitionen laut Zeller angewandt. Weitere Begriffe, die nicht im Definitionsumfang von Zeller auftreten, beispielsweise Fehler, error oder bug, werden in dieser Masterarbeit als defect (laut Zeller) definiert.

Id	Begriff	Autor	Synonym (Id)
1	defect	Zeller (2009)	4,7,8
2	infection	Zeller (2009)	6
3	failure	Zeller (2009)	—
4	fault	IEEE (1990)	1,7,8
5	flaw	Zeller (2009)	—
6	error	IEEE (1990)	2
7	bug	—	1,4,8
8	Fehler	—	1,4,7

Tabelle 2.1.: Zusammenfassung der Fehlerbezeichnungen

2.3.3. Klassifikationen von Fehlern

Fehler können nach unterschiedlichen Gesichtspunkten klassifiziert werden. Nachfolgend werden drei Klassifikationsmöglichkeiten vorgestellt, die in der Praxis Gebrauch finden.

2.3.3.1. Klassifizierung nach Fehlertypen

Häufig werden Fehler anhand der Typen klassifiziert und demzufolge ebenfalls von [Chmiel und Loui (2004)] in Syntax-, Semantik- und Logikfehler gegliedert. [Sender (2008)] fügt zu den drei erwähnten Fehlerkategorien eine weitere hinzu, den Designfehler.

Syntaxfehler:
ede natürliche Sprache sowie jede Programmiersprache unterliegt einer vordefinierten Syntax, die für das richtige Verständnis eingehalten werden muss. Wird die Syntax einer Programmiersprache nicht eingehalten, handelt es sich um Syntaxfehler, die vom Compiler erkannt werden.

Semantikfehler:
Liegt ein Fehler in der semantischen Bedeutung vor, ist von Semantikfehlern die Rede. In der Regel werden Semantikfehler nicht vom Compiler erfasst, Ausnahmen sind jedoch nicht ausgeschlossen. Einige Compiler sind fähig, bestimmte Semantikfehler zu erfassen. In Java wird beispielsweise der Vergleich zwischen einem Zahlenstring „9999“ und einem Integer 9999 , das einen semantischen Fehler repräsentiert, vom Compiler entdeckt. Ein weiteres Beispiel, die `IndexOutOfBoundsException`, stellt einen Semantikfehler dar, der vom Compiler nicht erfasst wird. Das Auftreten der `IndexOutOfBoundsException` ist die Folge eines fehlerhaften Elementenzugriffs. Ein Array besitzt ohne Ausnahme eine festgelegte Länge. Der Zugriff eines Elements außerhalb des Arrays ist nicht erlaubt. Der letztge-

nannte Fehler wird erst zur Ausführungszeit sichtbar und demzufolge ebenfalls der Kategorie Laufzeitfehler (siehe Klassifizierung nach zeitlichem Aspekt) zugeordnet.

Logikfehler:
Alle Fehler, die den Bereich Logik betreffen, entstehen in den meisten Fällen durch einen Denkfehler. Eine mathematische Formel für eine Kreisfläche $\pi * r^2$ wird mit der Formel für den Kreisring verwechselt und als $2 * \pi * r$ implementiert.

Designfehler:
Fehler, die auf ein mangelhaftes Softwaredesign oder eine mangelhafte Softwarearchitektur zurück zu führen sind, werden als Designfehler bezeichnet. Die zuvor erwähnten flaws entsprechen dieser Kategorie.

2.3.3.2. Klassifizierung nach Reproduzierbarkeit

Die nächste vorgestellte Fehlerklassifikationen ist die Klassifizierung nach ihrer Reproduzierbarkeit. Bei den zwei Extremen handelt es sich um den Bohrbug und um den Heisenbug, auch unter dem Namen Mandelbug bekannt, hinzu kommt der Aging-related Bug.

Bohrbug:
Fehler, die sehr einfach reproduzierbar, sind werden als Bohrbugs bezeichnet. Diese Art von Fehlern können mit Leichtigkeit kontinuierlich nachgestellt werden, und im Resultat wird fortlaufend ein identischer Fehler dargestellt [Madsen u. a. (2006)].

Heisenbug (Synonym: Mandelbug):
Das Gegenteil des Bohrbug ist der Heisenbug, ebenfalls unter dem Synonym Mandelbug bekannt. Heisenbugs sind Fehler, die entweder sehr schwer oder andernfalls nicht reproduzierbar sind. Hinzu kommt, dass es sich bei den Heisenbugs um die am häufigsten auftretenden Fehler handelt, die in laufenden Systemen zu beobachten sind. Die Reproduzierbarkeit von Heisenbugs gestaltet sich sehr kompliziert, weil derartige Fehler durch die Systemumgebung beeinflusst werden, wie z.B. durch Datenbanken über fehlerhafte Datenbankaufrufe, falsche Benutzerinteraktionen aufgrund von Unsicherheit und mangelnder Erfahrung des Benutzers mit dem System, durch Zugriff auf gesperrte Systemressourcen und viele weitere Faktoren [Vaidyanathan und Trivedi (2001)].

Aging-related Bug:
Eine dritte Kategorie stellen [Vaidyanathan und Trivedi (2001)] und [Grottke und Trivedi (2007)] zu den bereits erwähnten Heisenbugs bzw. Mandelbugs und den Bohrbugs vor; die Aging-related Bugs. Aging-related Bugs sind Fehler, hervorgerufen durch die Alterung von Softwaresystemen. Ältere Systeme können z.B. inkonsistente, undichte und/oder beschädigte Daten enthalten, die Fehler verursachen. Bezüglich des Aging-related Bug erweitern [Grottke und Trivedi (2007)] am Ende der Literatur ihre Sichtweise. Die Spezifizierung sagt

aus, dass ein Aging-related Bug ein spezielles Mandelbug bzw. eine Unterkategorie ist. Die Autorin vertritt ebenso diese Meinung, da ein Aging-related Bug das Merkmal der komplizierten oder nicht möglichen Reproduzierbarkeit entsprechend dem Mandelbug aufweist.

Definitionsabweichungen:
In der Literatur existieren einige Abweichungen zu den Begriffen Heisenbug und Mandelbug. Nicht ausnahmslos wird Mandelbug als Synonym für Heisenbug verwendet. [Madsen u. a. (2006)] vertritt die Meinung, dass ein Mandelbug ein spezieller Heisenbug ist. Im Gegensatz dazu behaupten [Grottke und Trivedi (2007)], dass ein Heisenbug ein gesonderter Mandelbug ist. Der einfacheren Zuordnung halber wird in dieser Masterarbeit Heisenbug als Synonym für Mandelbug gesehen.

2.3.3.3. Klassifizierung nach zeitlichem Aspekt

Es besteht die Möglichkeit, Fehler ebenso nach dem zeitlichem Aspekt zu klassifizieren. Diese Klassifizierung umfasst zwei Klassen; Kompilierungsfehler und Laufzeitfehler.

Kompilierungsfehler:
Kompilierungsfehler treten während der Kompilierungphase auf, falls die Interpretation des Codes nicht möglich ist. Infolgedessen wird der Kompilierungprozess unterbunden und die Ausführung des Programms verhindert. Heutzutage stellt ein Auftreten von Kompilierungsfehlern kein gravierendes Problem mehr dar. Entwicklungsumgebungen, wie z.B. Eclipse [Eclipse] und NetBeans [NetBeans], beugen Kompilierungsfehlern vor, indem Kompilierungsfehler unmittelbar erkannt (während des Schreibens rot markiert) und Verbesserungen vorgeschlagen werden.

Laufzeitfehler:
Unter Laufzeitfehlern werden Fehler verstanden, die zur Laufzeit auftreten. Laufzeitfehler decken den größten Teil der Fehler in der Informatik ab und umfassen aus den oben erwähnten Klassifikationen u.a. Semantikfehler, Logikfehler, Bohrbugs, Heisenbugs (Mandelbugs) und Aging-related Bugs. Dementsprechend beschäftigt sich die Informatik am häufigsten mit derartigen Fehlern im Bereich der Fehlererkennung, -lokalisierung und -behebung. Aufgrund des Schwerpunktes dieser Masterarbeit; der Erkennung und Lokalisierung von Fehlern, die in einer laufenden Anwendung auftreten, beschäftigt sich diese Masterarbeit ebenso mit Laufzeitfehlern.

2.3.4. Methodiken

In diesem Kapitel werden vier geläufige Debugging-Methodiken vorgestellt.

2.3.4.1. Trace Debugging

Das Trace-Debugging besteht hauptsächlich aus dem manuellen Betrachten der Programmzustände, die zur Zeit des Fehlers eingetreten waren. Das Zielprogramm erzeugt während der Ausführung Output-Nachrichten. Jede Output-Nachricht repräsentiert ein ausgeführtes Statement, wobei diese nacheinander abgespeichert den Ausführungsablauf darstellen. Die abgespeicherten Daten werden aktuell als Log-Dateien bezeichnet. Der Entwickler kann anhand der Log-Dateien schrittweise den Programmablauf verfolgen, um den Fehler zu finden [Sender (2008)]. Bei dem Trace-Debugging handelt es sich um eine ältere Debugging-Methodik, die auch heute noch in der Praxis Verwendung findet. Aufgrund des mühsamen, zeitaufwändigen und unübersichtlichen Betrachtens der Log-Dateien, wird das Trace-Debugging heutzutage allerdings in Kombination mit anderen Methodiken und/oder Technologien eingesetzt.

2.3.4.2. Slice Debugging

Das Slice-Debugging macht es sich zur Aufgabe, nur den direkt vom Fehler betroffenen Programmteil aus dem Gesamten zu extrahieren. Das Slice-Verfahren berechnet alle Statements die im Kontroll- und Datenfluss in Abhängigkeit zu dem Fehler stehen. Infolgedessen werden lediglich Statements in die Slice (Betrachtungsmenge) aufgenommen, die möglicherweise Einfluss auf den Fehler (Slice-Kriterium) nehmen. Dadurch wird erreicht, dass der Debugging-Prozess sich ausschließlich auf den entscheidenden Programmteil konzentriert, irrelevante Statements werden ausgeblendet. Nachfolgend wird zum besseren Verständnis ein Beispiel mit dem Slice-Kriterium (5,i) präsentiert, d.h., ein Fehler wurde in Zeile 5 Variable i entdeckt [Krinke (2003)].

```
1.int i = 10;               1.int i = 10;
2.int j = 20;
3.int k = i + j;
4.berechne(k);
5.berechne(i);               5.berechne(i);
```

Beispielprogramm — Slice zum Kriterium (5,i)

Der Pionier des Slice-Debugging ist Mark Weiser, der als erstes diese Methodik veröffentlichte [Weiser (1979)]. Bis heute, also seit über 30 Jahren, werden diese Slicing-Ansätze im Debugging-Bereich angewendet. Durch neue Erkenntnisse und Ideen wurden jedoch einige Grundlagen optimiert, wie z.B. das Berechnungsverfahren zur Slice-Bestimmung.

2.3.4.3. Delta Debugging

Delta Debugging stellt laut [Zeller und Hildebrandt (2002)] die Minimierung des Testfalls in den Vordergrund, um den Debugging-Prozess auf den relevanten Teil des Testfalls zu beschränken. Der Delta Debugging Algorithmus ist zuständig für die Verfeinerung und die Verallgemeinerung des Testfalls. Ein minimaler Testfall ist eine Reduktion des ursprünglichen Testfalls auf die notwendigsten Statements, wobei der ursprüngliche Fehler weiterhin produziert werden muss. Zudem müssen alle Statements des minimalen Testfalls tatsächlich in Abhängigkeit zum Fehler stehen. Das bedeutet, dass jede Entfernung eines weiteren beliebigen Statements den Fehler nicht mehr auslöst. Zur Erstellung des minimalen Testfalls führt der Delta Debugging Algorithmus aufeinander folgende Tests durch mit dem Ziel der Entfernung nicht relevanter Statements, bis der minimale Testfall erreicht ist.
Es soll zur Veranschaulichung des Prinzips ein Beispiel aus dem Alltag verwendet werden: ein Testflug. Ein Flugzeug stürzt wenige Sekunden nach dem Start ab. Durch die kontinuierliche Wiederholung der Situation unter veränderten Umständen kann herausgefunden werden, was genau den Absturz herbei geführt hat. Wenn trotz des Entfernen der Passagiersitze oder der Kaffeemaschine weiterhin ein Absturz erfolgt, stellen diese also keinen Bestandteil des minimalen Testfalls dar. Schließlich verbleibt das relevante Gerüst, einschließlich eines Testpiloten, des Kotflügels, der Start- und Landebahn, des Brennstoffs und der Motoren. Jedes Bestandteil ist relevant für das Reproduzieren des Absturzes und jede weitere Entfernung eines Bestandteils würde den Start und aufgrund dessen den Absturz verhindern. Analog zum Programm ist das der minimale Testfall.

2.3.4.4. Deklaratives (Algorithmisches) Debugging

Das deklarative Debugging ist eine semi-automatische Debugging Methodik zur Lokalisierung von Fehlern und wird laut [Sender (2008)] ebenfalls als algorithmisches Debugging bezeichnet. Die Funktionsweise wird in zwei Schritte gegliedert:
1. Erzeugen des Ausführungsbaums: Zu Beginn wird das komplette Zielprogramm durchlaufen, auf diese Weise erzeugt der deklarative Debugger einen Aufrufbaum.
2. Finden des fehlerhaften Knotens durch Befragung eines Orakels: Anhand des Aufrufbaums wird der Fehler gefunden, indem über den Aufrufbaum iteriert wird und ein Orakel (meist Entwickler) über die Richtigkeit der Knoten befragt wird. Jeder Knoten repräsentiert einen Berechnungsschritt oder einen Funktionsaufruf, und das Orakel muss über die Richtigkeit des Knotens bestimmen (true oder false). Es existieren einige Iterationsverfahren; an dieser Stelle wird das Top-Down-Verfahren betrachtet. Das Top-Down Verfahren beginnt an der Wurzel, das Orakel nach der Richtigkeit des Knotens zu befragen. Stellt sich der Knoten als richtig dar, wird die Befragung am Nachbarknoten weitergeführt. Bei einem fehlerhaften Knoten wird die Befragung an den Kinderknoten fortgeführt. Die Iteration wird beendet,

sobald ein fehlerhafter Knoten keine Kinderknoten mehr oder keinen einzigen fehlerhaften Kinderknoten besitzt. Die folgende Abbildung stellt eine Top-Down Fehlersuche dar. Diese

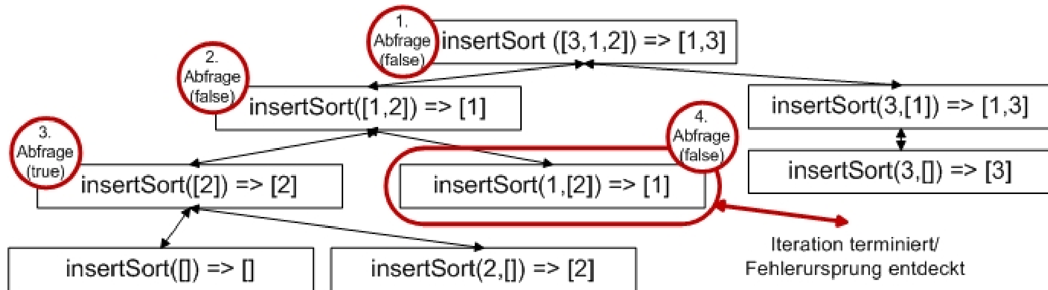

Abbildung 2.3.: Deklaratives Debugging: Top-Down Verfahren

Debugging Methodik wurde erstmals Anfang der Achtziger von [Shapiro (1983)] veröffentlicht. Aufgrund der Vielzahl an gestellten Fragen, die vom Entwickler beantwortet werden müssen, tendiert diese Methodik zu den zeitaufwändigen und monotonen Alternativen, Fehler zu lokalisieren, insbesondere in größeren Programmen.

3. Anforderungen

Der nächste Schritt dieser Masterarbeit ist die Anforderungsanalyse. Bestandteil einer Anforderungsanalyse sind die fachlichen und technischen Anforderungen [Starke (2002)]. Vor der Aufstellung der fachlichen und technischen Anforderungen wird die Systemidee definiert. Die Systemidee dient als erste konkrete Spezifizierung des Systems.

3.1. Systemidee

Ein viel versprechender Ansatz, um eine Systemidee zu entwickeln, ist die Auseinandersetzung mit den folgenden Fragen [Starke (2002)]:
Was ist die Kernaufgabe des Systems?
Die Kernaufgabe des Frameworks ist die automatisierte Fehlererkennung und -lokalisierung für Java-Anwendungen. In dieser kurzen Zusammenfassung wird eine Vielzahl von Aufgaben und Funktionen zusammen gefasst. Als erstes dient das Framework zur automatisierten Erkennung von Laufzeitfehlern. Das bedeutet, dass aufgetretene Laufzeitfehler in einer Zielanwendung (Java-Anwendung) bemerkt werden, und alle notwendigen und mit dem Fehler verbundenen Informationen gespeichert werden. Des Weiteren soll das Framework automatisiert aufgetretene Fehler lokalisieren können. Zu seinem Funktionsumfang gehört außerdem die Unterstützung des Wartungspersonals bei der Behebung der aufgetretenen Laufzeitfehler. Eine weitere Eigenschaft des Framework ist die unmittelbare Benachrichtigung der verantwortlichen Instanz über einen aufgetretenen Fehler. Um eine schnelle Benachrichtigung zu gewährleisten, soll die Versendung von SMS etc. unterstützt werden. Das Framework soll ebenfalls eine Fehlerverwaltung umfassen, über die auch Projektmanager zu jeder Zeit und an jedem Ort einen Überblick über den Fehlerstatus der Zielanwendung erhalten können. Das Framework soll neben dem Erkennen eines aufgetretenen Fehlers ebenso die Benutzer-bezogenen Daten erfassen. Benutzer der Zielanwendung, die zur Zeit des Auftretens des Fehlers authentifiziert waren und in Verbindung mit dem Fehler standen, können infolgedessen ermittelt werden. Anhand dieser Informationen kann auf die Benutzer (z.B. Kunden) eingegangen werden.
Zusammengefasst handelt es sich bei dem zu entwickelnden System um ein vertikales Framework (siehe Klassifizierung nach Ausdehnungsebene), dessen Fokus auf den Monitoring- und Debuggingbereich ausgerichtet ist.

Von wem wird das System genutzt?
Das System wird von Entwicklern bzw. von für die Wartung zuständigen Personen genutzt, deren Aufgabe die Fehlerbehebung ist. Weitere Benutzer können Projektverantwortliche sein, die sich über den aktuellen Fehlerstand des Systems informieren möchten. Infolgedessen sollte das System auf erfahrene Anwender sowie Laien ausgerichtet werden.
Welche Art Benutzeroberfläche hat das System?
Für den Bereich des Fehlermonitoring und die inbegriffene Fehlerverwaltung ist eine Benutzeroberfläche zu bevorzugen, die in kritischen Situationen eine Einsicht über aufgetretene Fehler an jedem Ort und zu jeder Zeit ermöglicht. Für die Debugging-Unterstützung kann die Benutzeroberfläche durch eine andere Alternative ersetzt werden.
Über welche Schnittstellen zu anderen Systemen verfügt das System?
Das Framework soll das zu beobachtende System (die Zielanwendung) überwachen; die Interaktion beschränkt sich auf die Zielanwendung. Zudem soll das Framework mit Benachrichtigungssystemen, wie Systemen zur SMS Versendung oder anderen Alternativen, kooperieren.
Wie werden die vom System bearbeiteten Daten verwaltet? Welche Art von Datenzugriffen ist notwendig?
Das Framework benötigt einen persistenten Speicher. Sollte sich in der Architekturerstellung herauskristallisieren, dass für die Zielanwendung ein persistenter Datenspeicher benötigt wird, sollte kein allzu komplexes Datenbankmanagementsystem verwendet werden, um die Zielanwendung nicht zusätzlich zu belasten.
Wie wird das System gesteuert?
Das Framework wird ereignisgetrieben gesteuert. Falls Fehler in der Zielanwendung auftreten, werden diese persistiert, und die zuständige Instanz wird kontaktiert. Ebenso wird im Bereich der Debugging-Unterstützung das System durch Ereignisse gelenkt. Der Benutzer steuert den Prozess zur Fehlerbehebung an.

3.2. Fachliche Anforderungen

Die fachlichen Anforderungen werden mit Hilfe von Anwendungsfällen (engl.: Use Cases) definiert. Nachfolgend werden lediglich die wichtigsten Anwendungsfallbeschreibungen aufgeführt. Zur Vervollständigung des Anwendungsfall-Diagramms befinden sich die restlichen Anwendungsfallbeschreibungen im Anhang A.

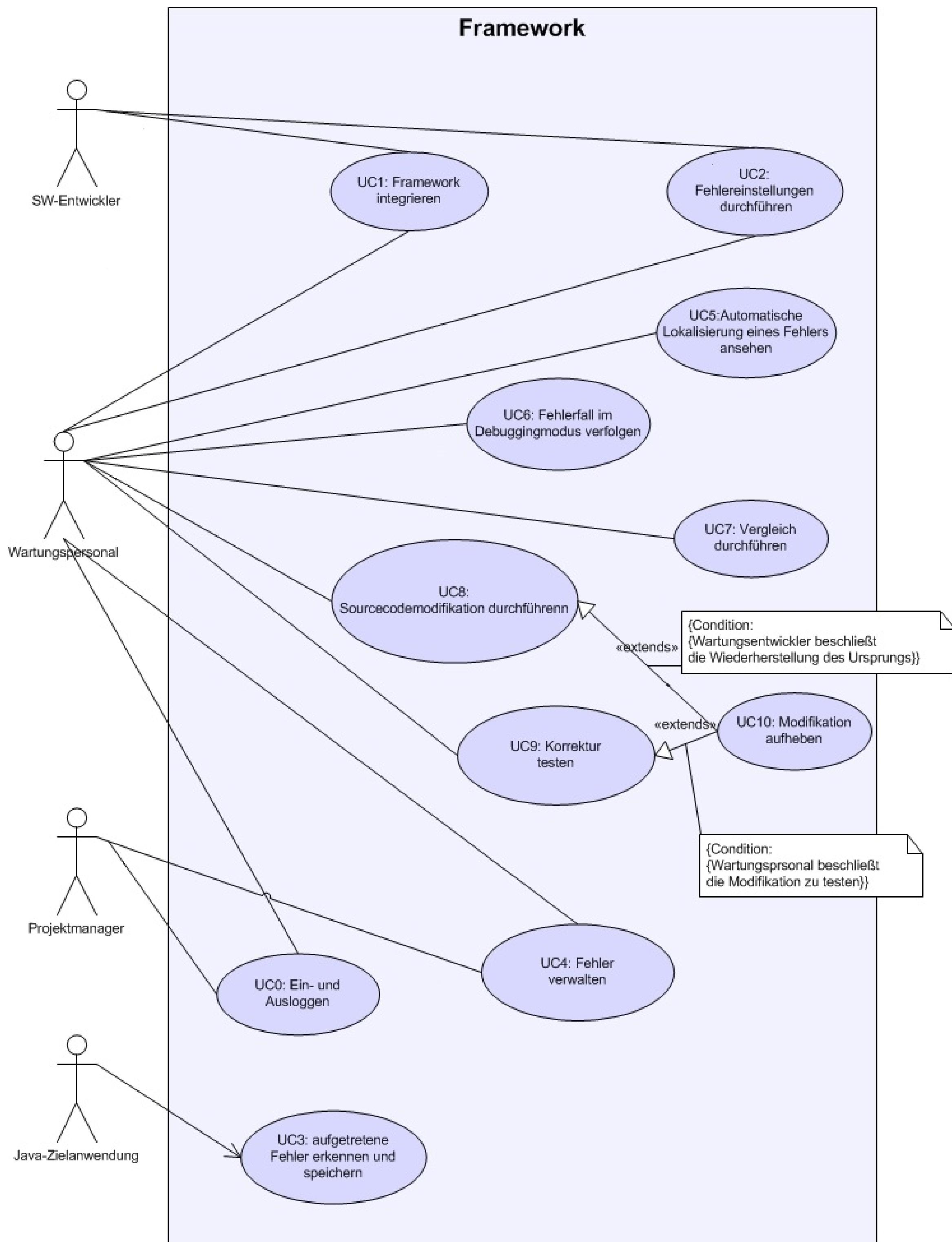

Abbildung 3.1.: Anwendungsfall-Diagramm

ID	1
Name	Framework integrieren
Kurzbeschreibung	Das Framework wird in die Java-Zielanwendung integriert.
Akteur	SW-Entwickler
Vorbedingung	Das Framework kann nur in eine Java-Anwendung integriert werden.
Nachbedingung	Das Framework ist vollständig integriert.
Hauptszenario	1. Hot-Spots anpassen, die für die komfortable Integration zur Verfügung gestellt werden.
Alternativabläufe	Hot-Spots anpassen, die für die flexible Integration zur Verfügung gestellt werden.
Fehlersituationen	—
Bemerkung	—

Tabelle 3.1.: UC1: Framework integrieren

ID	2
Name	Fehlereinstellungen durchführen
Kurzbeschreibung	Einstellungen für die Laufzeitfehler müssen definiert werden, um auf auftretende Fehler entsprechend reagieren zu können. Zu diesem Zweck werden Einstellungen für Gruppen vorgenommen, wobei jeder auftretende Laufzeitfehler einer Gruppe angehört.
Akteur	SW-Entwickler, Wartungspersonal
Vorbedingung	Der Akteur ist erfolgreich eingeloggt (UC0).
Nachbedingung	Die Gruppen der Laufzeitfehler müssen bekannt sein.
Hauptszenario	1. Bei der Integration können SW-Entwickler oder zu einem späteren Zeitpunkt das Wartungspersonal Fehlergruppen aktivieren oder deaktivieren, um zu steuern, ob auf einen aufgetretenen Fehler reagiert werden soll. 2. Bei der Integration können SW-Entwickler oder zu einem späteren Zeitpunkt das Wartungspersonal für jede Fehlergruppe Prioritäten festlegen. 3. Bei der Integration können SW-Entwickler oder zu einem späteren Zeitpunkt das Wartungspersonal einen Verantwortlichen oder eine Gruppe von Verantwortlichen einer Fehlergruppe zuweisen. 4. Bei der Integration können SW-Entwickler oder zu einem späteren Zeitpunkt das Wartungspersonal für jeden Fehler definieren, auf welche Art und Weise der Verantwortliche informiert wird (z.B. E-mail, SMS), falls dieser Fehler auftritt.
Alternativabläufe	—
Fehlersituationen	Eine Fehlergruppe kann nicht auf Aktiv gesetzt werden, wenn kein Verantwortlicher und keine Benachrichtigungsmethode festgelegt wurde.
Bemerkung	—

Tabelle 3.2.: UC2: Fehlereinstellungen durchführen

ID	3
Name	aufgetretene Fehler erkennen und speichern
Kurzbeschreibung	Das Framework muss aufgetretene Fehler, die in der Java-Zielanwendung ausgelöst wurden, erkennen und persistieren.
Akteur	Java-Zielanwendung
Vorbedingung	Ein Fehler muss in der Java-Zielanwendung auftreten und die entsprechende Fehlergruppe als aktiv deklariert sein.
Nachbedingung	Der Verantwortliche oder eine Gruppe von Verantwortlichen sind in Kenntnis gesetzt worden über den aufgetretenen Fehler. Alle notwendigen Informationen sind persistiert.
Hauptszenario	1. Automatisch einen aufgetretenen Laufzeitfehler erkennen. 2. Prüfen ob die entsprechende Fehlergruppe aktiv ist. 3. Falls Fehlergruppe aktiv ist, die verantwortliche Instanz informieren. 4. Alle relevanten Informationen über den Auftritt des Fehlers persistieren.
Alternativabläufe	Wenn die entsprechende Fehlergruppe des aufgetretenen Fehlers inaktiv ist, erfolgt keine Reaktion.
Fehlersituationen	—
Bemerkung	—

Tabelle 3.3.: UC3: aufgetretene Fehler erkennen und speichern

ID	4
Name	Fehler verwalten
Kurzbeschreibung	Das Framework bietet eine Fehlerverwaltung, mit Hilfe derer alle in der Java-Zielanwendung aufgetretenen Fehler verwaltet und Informationen eingesehen werden können.
Akteur	Wartungspersonal, Projektmanager
Vorbedingung	Der Akteur ist erfolgreich eingeloggt (UC0).
Nachbedingung	—
Hauptszenario	1. Aufgetretene Fehler verwalten, z.B. Bearbeitungsstaus setzen. 2. Alle relevanten Informationen über den Auftritt des Fehlers ansehen.
Alternativabläufe	—
Fehlersituationen	Fehlerverhalten loggen.
Bemerkung	Das Framework soll innerhalb der Fehlerverwaltung gleiche oder ähnliche Fehler erkennen können und in der Lage sein, dem Wartungspersonal sogenannte Geschwister-Fehler anzuzeigen. Infolgedessen können Geschwister-Fehler mitbehoben werden bzw. der Fehlerverlauf besser dargestellt werden. Diese sehr nützliche Anforderung wird in dieser Masterarbeit abgegrenzt.

Tabelle 3.4.: UC4: Fehler verwalten

ID	5
Name	Automatische Lokalisierung eines Fehlers ansehen
Kurzbeschreibung	Das Wartungspersonal soll die Möglichkeit erhalten, die Fehlerstelle zu betrachten und zudem den Progammverlauf, während der Fehler aufgetreten ist, zu verfolgen.
Akteur	Wartungspersonal
Vorbedingung	Der Akteur ist erfolgreich eingeloggt (UC0).
Nachbedingung	—
Hauptszenario	1. Der Akteur wählt den Fehler, der betrachtet werden soll. 2. Dem Wartungspersonal wird die Lokalisation (Stelle im Sourcecode), an der dieser Laufzeitfehler aufgetreten ist, präsentiert. 3. Des Weiteren kann das Wartungspersonal Informationen über Methoden, die vor und nach dem Fehlerauftritt aufgerufen wurden, einsehen. 4. Es kann der gesamte Verlauf mit Hilfe von gewöhnlichen Debugging- Funktionalitäten (z.B. step-in, step-over) verfolgt werden, der tatsächlich vorhanden war, als der Fehler aufgetreten ist. 5. Zudem sollen alle Parameter und Rückgabewerte (der aufgerufenen Methoden) bis ins kleinste Detail veranschaulicht werden können.
Alternativabläufe	—
Fehlersituationen	Meldung über Fehlerverhalten anzeigen.
Bemerkung	Dieser Anwendungsfall kann als eine Art historisches Debugging angesehen werden.

Tabelle 3.5.: UC5: Automatische Lokalisierung eines Fehlers ansehen

ID	6
Name	Fehlerfall im Debuggingmodus verfolgen
Kurzbeschreibung	Das Wartungspersonal kann den Sourcecode im Debuggingmodus laufen lassen. Es handelt sich hierbei um den Sourcecode, der ursprünglich den Fehler produziert hat. Im Debuggingmodus kann anschließend die Betrachtung des Sourcecodes durchgeführt werden.
Akteur	Wartungspersonal
Vorbedingung	Der Akteur ist erfolgreich eingeloggt (UC0).
Nachbedingung	—
Hauptszenario	1. Der Akteur wählt den Fehler, der betrachtet werden soll. 2. Der Akteur wählt die Version der Java-Zielanwendung, die ausgeführt werden soll. Standardmäßig wird die Version geladen, in der der Fehler ursprünglich aufgetretenen ist. 3. Des Weiteren besteht optional die Möglichkeit, einen Bereich im Sourcecode zu definieren, der ausgeführt werden soll (Testfallautomatisierung). 4. Verlauf im Debuggingmodus verfolgen. Der Debuggingfunktionsumfang soll den Funktionsumfang von Eclipse [Eclipse] ähneln.
Alternativabläufe	—
Fehlersituationen	Meldung über Fehlerverhalten anzeigen.
Bemerkung	Dieser Anwendungsfall repräsentiert das herkömmliche Debugging mit zusätzlichen Funktionalitäten.

Tabelle 3.6.: UC6: Fehlerfall im Debuggingmodus verfolgen

ID	7
Name	Vergleich durchführen
Kurzbeschreibung	Das Framework präsentiert eine Gegenüberstellung. Der ursprüngliche Verlauf (UC5) wird dem Verlauf des Debuggingmodus (UC6) gegenübergestellt. Der ursprüngliche Verlauf beschreibt die Zustände, die beobachtet wurden, als der Fehler aufgetreten ist. Der Verlauf des Debuggingmodus, die Zustände, die aktuell produziert werden.
Akteur	Wartungspersonal
Vorbedingung	Der Akteur ist erfolgreich eingeloggt (UC0).
Nachbedingung	—
Hauptszenario	1. Der Akteur wählt den Fehler, der beobachtet werden soll. 2. Der Akteur verfolgt den Fehlerfall im Debuggingmodus und zugleich wird ebenso der ursprüngliche Verlauf gegenübergestellt. 3. Abweichungen beider Verläufe werden deutlich markiert.
Alternativabläufe	—
Fehlersituationen	Meldung über Fehlerverhalten anzeigen.
Bemerkung	—

Tabelle 3.7.: UC7: Vergleich durchführen

3.3. Technische Anforderungen

Im Gegensatz zu den fachlichen Anforderungen werden an dieser Stelle die nicht-fachlichen Anforderungen beschrieben, die das Framework erfüllen soll.

Java Kompatibel:
Als Zielanwendungen sollen alle Java-Anwendungen ab Java 1.4 akzeptiert werden.

Zwei Arten von Hot-Spots:
Das Framework soll die Merkmale eines Gray-Box Frameworks (siehe Klassifizierung nach Spezifizier- bzw. Erweiterbarkeit) aufweisen. Zum einen soll die Integration des Frameworks in eine Java-Anwendung schnell durchführbar sein, um eine komfortable Nutzung zu ermöglichen. Zum anderen soll es möglich sein, das Framework sehr flexibel nutzen zu können. Aus diesem Grund sollen einerseits wenige präzise Hot-Spots zur schnellen Integration des Framework definiert werden. Andererseits sollen zudem weitere Hot-Spots zu Verfügung gestellt werden, die den flexiblen Einsatz des Frameworks ermöglicht.

Modell- und Spezifikationsunabhängig:
Ein Bestandteil des Frameworks ist das Monitoring. In den meisten Monitoringintegrationen wird ein Modell der Zielanwendung oder andere Spezifikationen, die das korrekte Verhalten der Zielanwendung beschreiben, benötigt. Diese Modell- oder Spezifikationserstellung ist zeitaufwändig und bedarf eines detaillierten Wissens über die Zielanwendung. Da dies nicht in allen Fällen gegeben ist, soll eine schnelle und einfache Integration des Frameworks gewährleistet werden. Das Ziel ist, formale Spezifikationen, formale Logik, eine Verifikations-Technik, ein Wissen über das Verhalten der Zielanwendung und ein Modellbild der Zielanwendung als Grundvoraussetzung nicht mehr erforderlich zu machen.

Erreichbarkeit:
Eine zur jeder Zeit verfügbare und leicht erreichbare Fehlerverwaltung bietet die Möglichkeit, in Extremsituationen einen schnellen Überblick über die Sachlage zu verschaffen, daher wird diese technische Anforderung gestellt. Der Zugriff auf die Fehlerverwaltung soll von jedem Ort und zu jeder Zeit leicht möglich sein, um aktuell aufgetretene und vorhandene Fehler einsehen zu können.

Konsistenz:
Das Framework ist in zwei logische Teile gegliedert; das Monitoring und das Debugging. Die Fehlerverwaltung bildet die Brücke zwischen beiden Bereichen und umfasst beide Teile.

Es ist festgelegt, dass die Fehlerverwaltung Ort- und zeitunabhängig erreichbar sein soll. Die beiden anderen Teile müssen nicht zwingend Bestandteil dieser Fehlerverwaltung sein. Unabhängig von der Architektur und der physikalischen Verteilung müssen jedoch die Daten (Informationen über aufgetretene Fehler) konsistent zueinander gehalten werden. Das erfüllt u.a. den Zweck, dass aktuell aufgetretene Fehler sofort in der Fehlerverwaltung eingesehen werden können, und behobene Fehler (durch die Debuggingunterstützung) umgehend in der Fehlerverwaltung als behoben markiert sind.

Signalisierung:
Erkannte und relevante Fehler der Zielanwendung müssen innerhalb kürzester Zeit dem Verantwortlichen mitgeteilt werden. Aus diesem Grund muss das Framework die Fähigkeit besitzen, SMS verschicken oder andere Benachrichtigungsalternativen unterstützen zu können, die eine schnelle Benachrichtigung ermöglichen.

Versionsabgleichung:
Das Framework muss in der Lage sein, Versionen der Zielanwendung zu verwalten. Zu Debugging-Zwecken ist es wichtig, es dem Wartungspersonal zu ermöglichen, die entsprechende Version (der Zielanwendung) für die Ausführung des Debugging Vorgangs wählen zu können.

Performance:
Zwei Performanceziele sollen durch die technischen Anforderungen erreicht werden.

1. Aufgetretene relevante Fehler sollen innerhalb von 10 Sekunden dem Verantwortlichen mitgeteilt werden.
2. Die Performance der Zielanwendung soll nicht stark beeinträchtigt werden. Eine maximale Verzögerung von 10% soll nicht überschritten werden.

4. Existierende Ansätze und Marktanalyse

Diese Masterarbeit verfolgt zwei Arten von Forschungsansätzen.

1. Zum einen handelt es sich um die Recherche existierender Ansätze,
2. zum anderen um eine Marktanalyse.

Der wesentliche Unterschied zwischen diesen beiden Ansätzen ist der wissenschaftliche bzw. der kommerzielle Aspekt. Bei der Recherche existierender Ansätze werden lediglich Veröffentlichungen und Forschungen aus der Fachliteratur (z.B. IEEE, ACM) verwendet. Diese Quellen sind nicht profitgebunden, der Schwerpunkt liegt in den neusten Innovationen, Erkenntnissen, Errungenschaften, Forschungen und vielen anderen wissenschaftlichen Aspekten. Die Marktanalyse weitet dagegen ihre Suche auf Produkte aus, die einen kommerziellen Hintergrund haben. In den meisten Fällen handelt es sich um fertige Lösungen, die auf dem Markt angeboten werden.
Bei der Recherche existierender Ansätze sowie bei der Marktanalyse werden die Bereiche Monitoring und Debugging separat behandelt, da andernfalls kaum oder keine Ergebnisse erzielbar sind. Vorab folgt ein Einblick in die Vorgehensweise einer Recherche existierender Ansätze und der Durchführung einer Marktanalyse.

4.1. Vorgehen

Nachfolgend werden die Vorgehensweisen der Recherche existierender Ansätze und der Marktanalyse vorgestellt. Die Autorin konnte in Erfahrung bringen, dass Ansätze zur Durchführung einer Marktanalyse geläufiger ist im Gegensatz zur Recherche existierender Ansätze.

4.1.1. Vorgehen zur Recherche existierender Ansätze

Bei den existierenden Ansätzen wird nicht auf alle Anforderungen wert gelegt, im Gegensatz zur Marktanalyse. Es ist viel wichtiger, Lösungs- oder Forschungsansätze, die ähnliche Probleme behandeln, zu finden. Aus diesem Grund ist der Umfang an Informationsquellen beträchtlich. Literatur, die sich vordergründig mit einer anderen Problematik befasst, kann sich im Nachhinein als nützlich erweisen. Die Autorin fand keine Literatur, die sich mit dem Vorgehen einer Recherche existierender Ansätze befasst. Ein wichtiges Grundprinzip bei der Recherche existierender Ansätze ist, stets Fachliteratur zu verwenden.

4.1.2. Vorgehen zur Marktanalyse

[Schütte und Vering (2004)] stellen eine Möglichkeit zur Durchführung einer Marktanalyse vor. Es findet eine Aufteilung der Analyse in zwei Phasen statt, in die Grob- und Feinanalyse. Die **Grobanalyse**, wie der Name besagt, beinhaltet eine Auflistung der Alternativen, die über einen groben Filter gesucht werden. Dies soll kostensparend wirken, da ein feiner Filter über eine Vielzahl an Softwaresystemen sich als zu zeitaufwändig gestalten würde. Jedoch werden die Ausschluss-Kritierien bereits in der Grobanalyse berücksichtigt, um unpassende Software vorweg zu verwerfen.
Die **Feinanalyse** beinhaltet eine Vielzahl an Kriterien. In der zweiten und letzten Phase wird nach einer Software gesucht, die allen Kriterien entspricht.
Diese Vorgehensweise der Grobanalyse sowie die Feinanalyse ist sehr zeitaufwändig. Zudem wird keine Aussage getroffen, wie im Detail die Kriterien dargestellt werden sollen.

[Vlachakis u. a. (2005)] gehen einen anderen Weg, um eine Marktanalyse durchzuführen. Es wurde eine Marktanalyse für Portal Software für Business-, Enterprise und E-Collaboration mit Hilfe eines Fragebogens durchgeführt. Das Fraunhofer-Institut hat einen Fragebogen, bestehend aus Multiple Choice und Textfragen mit vordefinierter Antwortlänge, an Softwarehersteller bzw. Vertreiber versendet. Anhand der Auswertung dieser Fragebögen wird eine Marktanalyse erstellt. Selbstverständlich ist vorab eine Grobauswahl notwendig, um nicht alle Softwarefirmen kontaktieren zu müssen.
Dieses Vorgehen wirkt auf den ersten Blick effizient, birgt aber auch einige wesentliche Nachteile: Die Fragebögen werden nicht immer ausgefüllt und zurück geschickt. Bei Fraunhofer wurde ein Rücklauf von nur 44 von 72 Fragebögen beobachtet. Bei kleineren Unternehmen oder Einzelpersonen ist mit einer weiteren Senkung der Rate zu rechnen. Des Weiteren müssen die Antworten präzise sein, um inkorrekten Aussagen Vorzubeugen, und es ist zu bedanken, dass Antwortzeiten zu erheblichen Verzögerungen in der Untersuchung führen können.

Es existieren noch weitere Vorgehensweisen, die meisten Vorgehensweisen ähneln jedoch dem eingangs vorgestellten Verfahren. Jedoch wird in den meisten Fällen die Marktanalyse in mehr als zwei Phasen spezifiziert, wie z.B. das Vorgehen der Marktanalyse laut [Veiser (2007)].

Die Autorin fasst, basierend auf [Schütte und Vering (2004)] und [Veiser (2007)], die Marktanalyse in vier Phasen plus eine Phase zusammen. Die letzte Phase ist eine Entwicklung der Autorin dieser Arbeit.

1. Kriterien aufstellen (Anforderungen an die Software).
2. Durchführung der Grobsuche; alle findbaren Softwaresysteme auflisten, die den wichtigsten Kriterien entsprechen, z.B. DBMS mit einfacher Anbindung an Java.
3. Aussortierung nach den restlichen Kriterien.
4. Genauere Betrachtung und, wenn möglich, ein Testen der verbleibenden Alternativen (Schritt 3 und 4 können zusammengefasst werden).
5. Falls die Suche zu keinem Ergebnis führt, können die Anforderungen bzw. Kriterien extrahiert und gruppiert werden. Anschließend wird die Marktanalyse auf die Kriterien beschränkt, die einer Gruppe zuzuordnen sind. Jedoch ist Vorsicht geboten, einige Kriterien können nicht extrahiert werden und sollten für alle Systeme gelten, wie z.B. Open Source. Das Spalten der Kriterien gilt eher für den Funktionsumfang.

4.2. Monitoring

An dieser Stelle werden die Recherchen für das Monitoring durchgeführt. Als erstes werden existierende Ansätze recherchiert und im Anschluss der Markt analysiert. Die Recherchen schließen mit einem Vergleich und einer Evaluation der ermittelten Ergebnisse ab.

4.2.1. Existierende Ansätze: Monitoring

Die nachfolgend vorgestellten existierenden Ansätze von Monitoringsystemen sind Java kompatibel.

4.2.1.1. Aspektorientiertes Programmieren (AOP)

[Nusayr und Cook (2009)] verfolgen einen interessanten Ansatz im Bezug auf das Laufzeit-Monitoring. Sie sind der Meinung, dass mit Hilfe der AOP Technologie das Laufzeit-Monitoring unkompliziert umgesetzt werden kann. Laut [Richter (2005)] ist AOP eine weit verbreitete und oft zum Einsatz kommende Technologie, mit deren Unterstützung logisch abgegrenzte Aspekte konzeptionell von einem domänenspezifischen Programm separiert werden können. AOP ist in der Realität keine eigene Programmiersprache, sondern ein Konzept, das zur Laufzeit bzw. Kompilierzeit das Einweben der Aspekte (als Sourcecode) in den vorhandenen Sourcecode ermöglicht. Am Ende ist das Resultat reiner Sourcecode. Die AOP Technologie ist ebenfalls im Javaumfeld verbreitet: Ein Beispiel für die AOP Implementierung in Java ist AspectJ. AspectJ wird mit Hilfe von Reflections umgesetzt, die das Eingreifen in den Programmablauf ermöglichen, um zur Laufzeit Klassenstrukturen auszulesen und zu verändern.

4.2.1.2. DynaMICs

[Gates und Teller (1999)] [Gates u. a. (2001)] [Delgado u. a. (2004)] Bei DynaMICs handelt es sich um einen Monitoring Ansatz, der auf Constraints (zu Deutsch: Bedingungen) basiert. Er ist so ausgelegt, dass nicht nur Programmierer, sondern auch Designer sowie Fachexperten in der Lage sind, Spezifikationen zu definieren. Die Eigenschaften werden mit Hilfe der Vienna Develpment Method (VDM) formuliert. VDM basiert auf Mengen und Funktionen, die das Formulieren präziser Spezifikationen ermöglicht. DynaMICs basiert auf dem event-conditions-action Paradigma. Das Paradigma besagt, dass ein Ereignis (event) die Auswertung der Bedingung (condition) auslöst und, falls diese verletzt wird, eine Aktion (action) ausgeführt wird. Bei einer Verletzung loggt DynaMICs die Ausführungsstelle, an der die Verletzung stattfand, die Identität der verletzten Bedingung und den aktuellen Zustand. DynaMICs unterscheidet zwischen zwei Verletzungsarten, der kritischen und der nicht kritischen Verletzung. Tritt eine kritische Verletzung ein, wird das Programm beendet oder eine andere Terminierungsaktion ausgeführt. Eine unkritische Verletzung führt zu keiner Programmterminierung. DynaMICs ist nicht nur für Java-Programme, sondern ebenso für C und C++ ausgelegt.

4.2.1.3. Java with Assertions (Jass)

[Bartetzko u. a. (2001)] [Delgado u. a. (2004)] Jass ist ein in Java geschriebener Pre- Compiler für Java-Anwendungen. Die Hauptaufgabe von Jass besteht darin, Annotationen in reinen Java-Code zu übersetzen. Die Annotationen dienen zur Spezifizierung von Eigenschaften.

Mit Hilfe von Annotationen ist es möglich, Vor- und Nachbedingungen für Methoden, Klasseninvarianten und Schleifeninvarianten zu modellieren, sowie anderweitige Prüfungen, die in Teile des Programmcodes eingefügt werden. Zur Laufzeit werden die Spezifikationen überprüft; sollte eine Verletzung vorliegen, kann ebenfalls zur Laufzeit mittels Exception-Handling reagiert werden.

4.2.1.4. Java PathExplorer (JPaX)

[Havelund und Rosu (2001)] [Havelund und Rosu (2004)] Java PathExplorer ist ein Laufzeit Verifikationstool für Java-Anwendungen. Es besteht die Möglichkeit, benutzerdefinierte Eigenschaften eines Java Programms mit Hilfe der Temporallogik zu formulieren, um zu überprüfen, ob diese konform mit dem laufenden Java Programm sind. Temporallogik ist eine formale Spezifikationssprache für die Beschreibung und Analyse von zeit- und verhaltensabhängigen Aspekten bei Systemen [Manna und Pnueli (1992)]. JPaX kann ebenso analysieren, ob das Java Programm Nebenläufigkeitsfehler, wie Deadlocks, aufweist. Die Nebenläufigkeitsanalyse bedarf keiner bereitgestellten Spezifikation des Benutzers. Die Analyse wird mit Hilfe von klassischen Mustern für Nebenläufigkeitsfehler durchgeführt. Um noch einmal auf die Temporallogik zurück zu kommen: Die Temporallogik kann vom Anwender in Maude formuliert werden. Maude ist eine high-level Sprache und ein High-Performance-System, welches ausführbare Spezifikationen und eine deklarative Programmierung in Temporallogik unterstützt. Für weitere Informationen wird auf [Clavel u. a. (2010)] verwiesen. Alternativ können ebenso temporale Spezifikationen in Automaten oder Algorithmen überführt werden [Delgado u. a. (2004)].

4.2.1.5. Java Runtime Timing-Constraint Monitor (JRTM)

[Mok und Liu (1997b)] [Mok und Liu (1997a)] [Delgado u. a. (2004)] Java Runtime Timing-Constraint Monitor ist für Echtzeitsysteme ausgelegt. Aus diesem Grund sind in diesem Zusammenhang die zeitlichen Bedingungen (timing-constraints) von großer Wichtigkeit. JRTM ist in Java geschrieben und stellt Dienste für das Überwachen von timing-constraint für Java-Anwendungen bereit. JRTM beinhaltet einen Algorithmus namens „ESJicient“, der Verletzungen von timing-contraints zum frühestmöglichen Zeitpunkt erfasst. Für die Definition von timing-constraints wird eine Spezifikationssprache, basierend auf Real Time Logic (RTL), verwendet. Beim Auftritt einer Verletzung der Spezifikation wird der Benutzer benachrichtigt. Zudem kann eine Nachricht zu der betreffenden Java-Anwendung geschickt werden, um eine Behebungsmaßnahme aufzurufen.
Der Monitor ist fähig, auf der gleichen Maschine, die den Zielprozess bedient, oder auf einer separaten Maschine zu agieren.

4.2.1.6. Monitoring and Checking (MaC)

[Kim u. a. (1999)] [Kim (2001)] [Kim u. a. (2004)] [Delgado u. a. (2004)] Monitoring and Checking ist ein in Java implementiertes Laufzeit Monitoring Framework. MaC besitzt die Fähigkeit, Sicherheit, begrenzte Lebendigkeit und andere zeitliche Eigenschaften zu überwachen.
Das Laufzeit Monitoring Framework (MaC) besteht aus den folgenden drei Komponenten:

- Der **Filter** setzt sich zusammen aus einer Reihe von Programm-Fragmenten, die in die Implementierung integriert werden. Die Hauptfunktion eines Filters besteht darin, Veränderungen von Objekten zu verfolgen und festzuhalten, um Objekte zu überwachen. Die relevanten Zustandsinformationen werden an den event recognizer gesendet.
- Der **event recognizer** empfängt vom Filter ein Monitoring Skript, anhand dieses erkennt der event recognizer Ereignisse, die überwachte Variablen enthalten. Die Ereignisse inklusive der veränderten Variablen werden an den runtime-checker gesendet. Zudem werden ebenfalls die entsprechenden Anforderungen für das auftretende Ereignis an den runtime-checker übermittelt.
- Der **runtime checker** führt im Anschluss eine Prüfung, die an die Anforderungen angelehnt ist, durch.

MaC verwendet zwei Sprachen, Primitive Event Denition Language (PEDL) und Meta Event Defnition Language (MEDL). Die Sprache PEDL wird für das Monitoring Skipt verwendet, die unter anderem spezifiziert, welche Informationen vom Filter zum event recognizer übersendet werden. MEDL ist dagegen für die Spezifizierung der Anforderungen zuständig.

4.2.1.7. Monitoring-Oriented Programming (MoP)

[Chen und Rosu (2003)] [Chen u. a. (2004)] [Chen und Rosu (2007)] [Delgado u. a. (2004)] MoP basiert auf Annotations. Der Benutzer hat die Möglichkeit, mit Hilfe der Annotations formal Spezifikationen zu definieren. Die Annotations können an jeder Stelle im Programmcode vom Benutzer platziert werden. Im Anschluss folgt die sogenannte Vorkompilierungsphase, in der aus den formalen Spezifikationen (Annotations) ein Monitoringcode erzeugt wird. Der erzeuge Monitoringcode wird in derselben Sprache erzeugt wie das Zielprogramm. MoP unterstützt mehrere Formalismen, zum einen die lineare Temporallogik, zum anderen erweiterte reguläre Ausdrücke. MoP ist ein neuer Ansatz, macht jedoch mit dem ersten Prototypen namens Java-MoP auf sich aufmerksam. Wie der Name Java-MoP vermuten lässt, erfolgte die Implementierung in Java und wurde als eine Client-Server Applikation umgesetzt. Für die Java-MoP Realisierung wird AspektJ verwendet. In diesem Fall wird die Spezifikation nicht mittels Annotations realisiert, sondern als ein Aspekt. Bei einer näheren Betrachtung erweist

sich Java-MoP nicht nur als eine Realisierung des MoP-Ansatzes, es ist ebenso eine Realisierung für einen Monitor, basierend auf dem AOP Ansatz, der bereits in 4.2.1.1 vorgestellt wurde.

4.2.1.8. Runtime Assertion Checker for the Java Modeling Language (RAC)

[Cheon und Leavens (2002)] [Burdy u. a. (2004)] [Leavens u. a. (1998)] [Delgado u. a. (2004)] Dieser Ansatz basiert auf der Java Modeling Language (JML). Mit JML besteht die Möglichkeit, Java-Klassen oder Java-Interfaces zu spezifizieren. Spezifikationsformalien basieren auf der Syntax der Java Deklaration. Diese können in Form von Vor- und Nachbedingungen sowie Klasseninvarianten deklariert werden. Bei einer Verletzung wird eine Exception geworfen. Vorbedingungen werden als requires und Nachbedingungen als ensures deklariert. Die Invarianten beschreiben Eigenschaften, die kontinuierlich erfüllt sein müssen. Die Spezifikationen werden mit Hilfe von Annotations definiert, die in den Java-Sourcecode als Java-Sourcecode eingefügt und zur Laufzeit geprüft werden. Zum besseren Verständnis wird ein kleines Beispiel dargestellt mit zwei Vor- und Nachbedingungen sowie Invarianten:

```
/*@behavior
@ requires wB != null && ! wB.equlas("");
@ ensures wB.equals(winkelBezeichnung)
@ && winkel == 90;
@ signals_only NullPointerException;
@ signals (NullPointerException) wB == null;
@*/
public Rechterwinkel(String wB){
     winkelBezeichnung=n;
     winkel=90;
}
```

4.2.2. Marktanalyse: Monitoring

Äquivalent zur Recherche existierender Ansätze werden ebenso Java kompatible Monitoringsysteme aufgelistet.

4.2.2.1. Java Application Monitor (JAMon)

[Souza (2007)] JAMon stellt kein eigentliches Monitoring-Tool dar, es ist eine kleine Java-Bibliothek zur Unterstützung des Monitorings von Java-Anwendungen. Die Version 1.0 hat lediglich einen kleinen Funktionsumfang, mit dessen Hilfe die Ausführungszeiten des Codes programmatisch gemessen und darauf basierende Statistiken erzeugt werden konnten.

Diese Klassenbibliothek befindet sich jedoch in einem ständigen Prozess der Weiterentwicklung. Sie nimmt von Version zu Version an Funktionsumfang zu. Mittlerweile können ebenso Aufrufe an externe Systeme, wie z.B. Webservices, Datenbankaufrufe, gemessen werden. Version 2.2 besitzt die Fähigkeit, JDBC und SQL-Exceptions mit Hilfe des neuen JAMon-JDBC-Driver zu verfolgen. Die aktuelle Version 2.7 ermöglicht zusätzlich die Überwachung von Tomcat, Jetty, JBoss und anderen Webcontainern.

4.2.2.2. JConsole

[Chung (2004)] JConsole-Tool ist eine in Java JDK 5.0 mitgelieferte Swing-Applikation. Sie enthält eine Monitoring- und Management Console zur Überwachung von Java Prozessen via Java Management Extension (JMX). Über JMX besteht die Möglichkeit, von außen auf eine Java-Anwendung über managed Beans (MBean) zuzugreifen. Für jede Zielanwendung können MBeans erstellt und bei der MBean Server Plattform registriert werden. Der JMX kompatible Client JConsole kann sich im Nachhinein mit der MBean Server Plattform verbinden, um die Zielanwendung zu verwalten.

4.2.3. Vergleich

Der Vergleich fasst alle Resultate existierender Ansätze und Marktanalyse für Monitoringsysteme tabellarisch anhand der in 2.2 vorgestellten Klassifikationen zusammen.

	Eigenschaften/Monitoring	AOP	DynaMICs	Jass	JPaX	JRTM	MaC	MoP	RAC	JAMon	JConsole
Specification Language											
Lang.T.	Algebra			X	X			X			
	Automata							X			
	Logic		X		X	X	X	X	X		
	HL/VHL	X		X			X	X	X	X	X
Abs.L.	Domain			X		X	X	X			
	Design			X				X	X	X	
	Implementation	X	X	X	X	X	X	X	X	X	X
Pro.T.	Safety	X	X	X	X		X	X	X	X	X
	Other Temporal	X		X	X	X	X	X		X	X

	Eigenschaften/Monitoring	AOP	DynaMICs	Jass	JPaX	JRTM	MaC	MoP	RAC	JAMon	JConsole
M.Direct.	Program	X		X				X		X	X
	Module	X		X			X	X	X	X	X
	Statement	X		X	X		X	X	X		X
	Event	X	X	X	X	X	X	X			X
	Monitor										
M.Points	Manual	X		X		X		X			X
	Automatic-Static	X	X	X	X	X	X	X	X	X	X
	Automatic-Dynamic						X				
Placem.	Inline	X	X	X			X	X	X		X
	Offline-Synchronous		X			X	X	X			X
	Offline-Asynchronous				X		X	X		X	X
Platf.	Software	X	X	X	X	X	X	X	X	X	X
	Hardware										
Impl.	Single Processor	X	X	X					X		X
	Multi-Programming				X	X	X	X			X
	Multi-Processor		X		X	X	X			X	X
	Event-Handler										
C.L.	Universal Application				X				X	X	
	Individual Applikation	X	X	X		X	X	X			X
Resp. E.	No Effect				X	X				X	
	User Controll		X								X
	Automated	X	X	X			X	X	X		
	Operational Issues										
SourcePT	General-Purpose	X	X	X	X		X	X	X	X	X
	Domain-Specific										
	Category-Specific					X					
Depend	Hardware										
	Operating System										
	Language	X	X	X	X	X	X	X	X	X	X
L.ofM.	Research	X	X		X	X		X	X		
	Production			X			X			X	X

Tabelle 4.1.: Vergleich der Monitoringsysteme und -ansätze

4.2.4. Evaluation Monitoring

Nachfolgend werden zusammenfassend alle Monitoringsysteme resultierend aus den existierenden Ansätzen und der Marktanalyse bewertet. Die Bewertungskriterien stützen sich auf die Anforderungen, die in Kapitel 3 ausgeführt wurden.

Bewertungskriterien/Monitoring	AOP	DynaMICs	Jass	JPaX	JRTM	MaC	MoP	RAC	JAMon	JConsole
Anwendbar für alle Java-Anwendungen	++	++	++	++	++	++	++	++	++	++
Anwendbar für vorhandene Java-Anw.	++	++	+	+	+	++	+	+	+	++
Automat. Erkennung von Laufzeitfehlern	-	+	+	+	+-	+-	+	+	-	- -
Modell- bzw. Spezifikationsunabhängig	-	- -	- -	-	- -	- -	- -	- -	+	+
Einfache Integrierbarkeit	+	-	-	-	-	-	-	-	+	+
Erfassung aufgetretener Fehler	-	++	+	+	+	+	++	+-	+-	-
Erfassung benutzerbezogener Daten	-	- -	- -	- -	- -	- -	- -	- -	- -	- -
Sofortige Benachrichtigung über Fehler	-	- -	- -	- -	- -	- -	+-	- -	- -	- -

Tabelle 4.2.: Evaluation der Monitoringsysteme und -ansätze

4.3. Debugging

Die Nachforschungen im Debugging Bereich beginnen ebenfalls mit der Recherche existierender Ansätze. Als nächster Schritt folgt die Marktanalyse. Das Kapitel endet mit der Evaluation der gewonnenen Erkenntnisse.

4.3.1. Existierende Ansätze: Debugging

Es besteht eine Vielzahl an existierenden Ansätzen für das Debugging. Um den Fokus auf relevante Ansätze für die Masterarbeit zu setzen, ist eine Abgrenzung notwendig. Die im Anschluss vorgestellten existierenden Ansätze in Bezug auf das Debugging umfassen Veröffentlichungen und Forschungen, die ab dem Jahr 2000 publiziert wurden. Dies bedeutet nicht, dass ältere existierende Ansätze keine Relevanz haben, im Gegenteil; existierende Ansätze ab dem Jahr 2000 beinhalten oftmals ältere Ansätze und werden zudem miteinander oder mit andren Technologien kombiniert, um bessere Ergebnisse zu erzielen. Des

Weiteren besteht die Abgrenzung darin, existierende Ansätze gezielt zu betrachten, die zum Debuggen von Java-Anwendungen verwendet werden können.

4.3.1.1. Debugging strategy based on Requirements of Testing (DRT)

[Chaim u. a. (2003)] DRT stellt eine Strategie zur Unterstützung der Lokalisierung von Fehlern vor. Die Basis bilden die Untersuchungen der Indikatoren bzw. Hinweise, die zur Laufzeit durch die „data-flow testing requirement“ (dua) geliefert werden. Der Hintergrund dieser Strategie ist der, den Code mit Test-Artefakten zu assoziieren, um Fehler besser lokalisieren zu können. Dieser Ansatz basiert auf der Erkenntnis, dass während der Testphase die gesammelten Artefakte nützliche Informationen zur Fehlersuche liefern.
DRT beinhaltet:

- die Verwendung von Heuristiken zur Identifizierung einer Gruppe von „data-flow testing requirement“, diese sind Kandidaten für die Fehlerlokalisierung.
- zwei Mechanismen zur Unterstützung des Benutzers bei der Identifizierung der „dua’s“, bis der Fehler lokalisiert ist.

DRT besitzt drei wesentliche Eigenschaften:

1. konzentriert sich auf dynamische Informationen im Zusammenhang mit Testdaten,
2. kann mit wenig Aufwand in die Praxis umgesetzt werden und
3. verwendet Algorithmen, die einen konstanten Speicher verbrauchen.

4.3.1.2. Java Diagnosis Experiments (JADE)

[Mateis u. a. (2000)] JADE ist ein Prototyp, der einen benutzerfreundlichen und intuitiven Debugger darstellt, um den Benutzer bei der Lokalisierung von Fehlern in Programmen zu unterstützen. Die Debugging Software basiert auf der modellbasierten Diagnose mit dem Akronym MBD. MBD ist eine erfolgreiche und bewährte Technik zur Lokalisierung und Identifizierung von Fehlern. Das Grundprinzip der MBD basiert auf dem Vergleich eines Modells, das das korrekte Verhalten eines Systems beschreibt, mit dem beobachteten Verhalten eines Systems. Die Differenz kennzeichnet diejenigen Komponenten, die von ihrem normalen bzw. gewünschten Verhalten abweichen und bildet den Grundstein für die Lokalisierung der Fehler.
Der Ablauf beginnt mit der vollautomatisierten Partitionierung und Kompilierung des Programms in Modellkomponenten. Jede Modellkomponente entspricht einem Fragment aus dem Progammsourcecode, so dass zu einem späteren Zeitpunkt Differenzen im Sourcecode

direkt referenziert werden können. Unter Verwendung dieser Bestandteile erstellt die Diagnostic Engine eine Diagnose, die zum Finden von Bug Kandidaten verwendet wird. Kandidaten können durch zusätzliche Informationen eingegrenzt werden. Dazu müssen erwartete Werte von Variablen an bestimmten Stellen im Programm vom Benutzer (oder anderen Orakeln) geliefert werden. Die Wahl der Variablen und Stellen im Programm berechnet der „Measurement Selection“ Algorithmus. Das Resultat umfasst lediglich die Stellen im Programm, die einen Fehler beinhalten können. Aufgrund des Einschränkens des Fehlerbereichs gestaltet sich die Durchführung des Debuggins-Prozesses effizienter. Der Debuggingvorgang wird über die JADE-Benutzerschnittstelle getätigt.

4.3.1.3. Java Debugger using Aspect and Slicing (JDAS)

[Rakesh (2010)] JDAS ist ein leichtgewichtiges Tool, welches den Debugging Prozess unterstützt. Das Tool wurde basierend auf zwei Grundlagen entwickelt. Zum Einen wird Slicing Verwendet, diese Debugging Methode wurde bereits im Kapitel 2.3.4.2 vorgestellt, sie ist für die Erstellung der Abhängigkeitskette (Slice) zuständig. Die Abhängigkeitskette umfasst ein Trace-Statement, das im Kontroll- und Datenfluss in Abhängigkeit zu dem beobachteten Fehler steht, die restlichen Statements werden von JDAS ignoriert. Zum Anderen werden die Fähigkeiten der AOP Technologie eingesetzt, um Informationen zur Laufzeit zu gewinnen, die den Debugging Prozess effektiver gestalten. Zusammen gefasst: Slice Ergebnisse und dynamische Resultate werden während der Programmausführung gesammelt und kombiniert angezeigt, um dem Benutzer einen Fokus auf den relevanten Teil der Anwendung zu ermöglichen.
JDAS besteht aus den vier folgenden Komponenten:

1. Criteria Generator; wandelt den vom Benutzer erfassten Programmfehler in ein Slice Kriterium um.
2. Static Slicing Engine; ist verantwortlich für die Erstellung des Slice (Betrachtungsmenge).
3. Aspect Based Tracer; ist verantwortlich dafür, das Slice mit den dynamischen Informationen dem Benutzer zur Laufzeit kombiniert anzuzeigen.
4. JDAS GUI; ist die Benutzerschnittstelle und zeigt die Ergebnisse der drei eben vorgestellten Komponenten.

Nachfolgend wird schrittweise ein regulärer Ablauf dargestellt:

- Die Anwendung wird unter Verwendung von JDAS ausgeführt.
- Der Benutzer erfasst den Programmfehler.

- Die Criteria Generator Komponente wandelt dies in ein Slice Kriterium (eine Datei, bestehend aus Klassenname, Zeilennummer, Methodensignatur) um.
- Die Criteria Generator Komponente wird aktiv und erstellt den Slice, basierend auf den Kriterien der Criteria Generator Komponente.
- Die Aspect Based Tracer Komponente ist abschließend dafür zuständig, dem Benutzer ein effektives Debugging der Anwendung zu ermöglichen, indem der Benutzer den Debugging-Vorgang auf dem Slice durchführt, der zusätzlich mit weiteren Informationen angereichert ist.

4.3.1.4. Java Interactive Visualization Environment (JIVE)

[Czyz und Jayaraman (2007)] JIVE ist eine deklarative und visuelle Debugging Umgebung für Eclipse, die auf der Java Platform Debugger Architecture (JPDA) basiert. Die Umgebung hilft dem Benutzer ein besseres Laufzeitverständnis zu bekommen und unterstützt ihn beim Debugging Prozess. JIVE umfasst drei Bestandteile:

1. Interaktive Visualisierung: Während der Programmausführung werden Laufzeitinformationen in einer visuellen Darstellung, als Diagramme, veranschaulicht. Zwei Diagrammarten werden parallel verwendet. Zum einen wird ein Objektdiagramm dargestellt, das den aktuellen Ausführungszustand repräsentiert, indem es Objekte und ihre Strukturen sowie Methoden-Aktivierungen anzeigt. Zum anderen handelt es sich um ein Sequenzdiagramm, welches zur Visualisierung der Aufruf-Historie eingesetzt wird. Das JIVE Sequenzdiagramm wird interaktiv zur Laufzeit aufgebaut und reflektiert die aktuelle Abfolge der Objekt-Interaktion zur Ausführungszeit.
2. Deklaratives (query-based) Debugging: JIVE unterstützt Abfragebasiertes (querybased) Debugging. Der Benutzer hat somit die Möglichkeit, nicht traditionell Schritt für Schritt oder Objekt für Objekt den Debugging-Prozess zu durchlaufen, sondern präziser durch die Formulierung von Abfragen vorzugehen. Ein Benutzer kann z.B. Abfragen für Variablenänderungen, Zeilenausführungen, Objekterstellung, Exceptionauffangen definieren. Zudem wird mit Hilfe der Abfragen das Objekt- und Sequenzdiagramm besser skaliert, denn es werden lediglich Teile visualisiert, die den Abfragen entsprechen. Die Abfragen werden als Diagramm Anmerkungen in den Diagrammen aufgeführt und ebenso in einer Tabelle oder als Baum dargestellt.
3. Interaktive Ausführung: JIVE unterstützt vorwärts sowie rückwärts „Stepping“ von Java-Programmen und bietet auch die Möglichkeit, direkt auf alle vorherigen Stellen in der Aufruf-Historie zu springen, um das Objektdiagramm an dieser Stelle zu beobachten.

4.3.1.5. Java Platform Debugger Architecture (JPDA)

[Oracle (2010)] JPDA ist eine mehrstufige Debugging-Architektur und besteht aus einer Sammlung von APIs. Tool-Entwickler können unter Verwendung der JPDA auf JVM und JDK Versionen portierbare Debugging-Anwendungen erstellen.

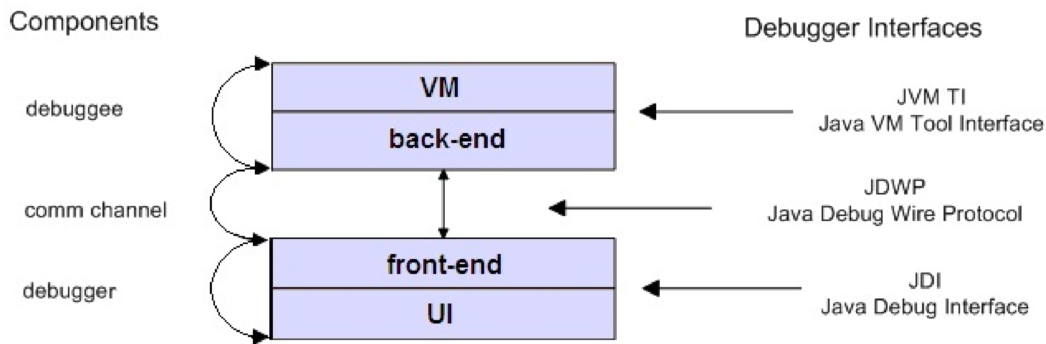

Abbildung 4.1.: JPDA

JPDA besteht aus drei Schichten:

- JVM TI - Java VM Tool Interface: Seit J2SE 5.0 wird JVM TI eingesetzt und ersetzt JVMDI. JVM TI bietet Dienste an, die für das Debugging erforderlich sind. Dienste umfassen z.B. das Einholen von Auskünften über das aktuelle Stack-Frame, Aktionen auf einem Breakpoint oder die Mitteilung über das Erreichen eines Breakpoints.
- JDWP - Java Debug Wire Protocol: Das JDWP legt das Protokoll zur Kommunikation zwischen dem gebuggten Prozess und dem Debugger-Front-End fest. Inklusive des Formats der transferierten Informationen und Anfragen. Ausgeschlossen von der Definition sind die Transport-Mechanismen, wie Socket, serielle Schnittstellen, Shared Memory etc. Die Spezifikation des Protokolls ermöglicht dem Debuggee und dem Debugger-Front-End, auf separaten JVM-Implementierungen und/oder auf getrennten Plattformen zu operieren.
- JDI - Java Debug Interface: Ist eine 100%-ige Java-Schnittstelle, die es Tool- Entwicklern ermöglicht, unter Verwendung dieser, einfach eine Debugger-Anwendung zu erstellen. Die JDI definiert Informationen und Anfragen auf der Benutzer-Codeebene. Es besteht ebenso die Möglichkeit, für die Implementierung einer Debugger-Anwendung direkt auf JDWP oder JVM TI zuzugreifen. Es ist jedoch empfehlenswert, die JDI-Schicht für die Entwicklung zu verwenden, weil sie die Integration von Debugging-Fähigkeiten in Entwicklungsumgebungen erleichtert.

JDPA umfasst folgende Komponenten:

- Debuggee: Bei dem Debuggee handelt es sich um den Prozess, der gebuggt wird, er besteht aus:
  - der Zielanwendung,
  - der JVM, auf der die Zielanwendung läuft. Die VM implementiert das Java Virtual Machine Debug Interface (JVM TI).
  - und dem Back-End. Das Back-End ist verantwortlich für die Kommunikation zwischen dem Debugger-Front-End und der Debuggee-VM. Die Anfragen werden an die VM übermittelt und die Antworten an das Debugger-Front-End zurückgesendet.
- Debugger: Der Debugger dagegen beinhaltet:
  - das Front-End: Das Front-End ist die Implementierung des High-Level Java Debug Interface (JDI) unter Verwendung der Informationen aus dem Low-Level Java Debug Wire Protocol (JDWP).
  - und das User Interface (UI): Die Benutzeroberfläche ist absichtlich nicht spezifiziert, damit Tool-Entwickler bzw. Tool-Anbieter eine mehrwertige Implementierung realisieren. JPDA bietet jedoch ein einfaches Benutzeroberflächen-Beispiel, welches als Testumgebung und als Ausgangspunkt für die Entwicklung komplexer Benutzeroberflächen dient.

4.3.2. Marktanalyse: Debugging

Wie bereits erwähnt, folgt an dieser Stelle eine Auflistung aller Debuggingsysteme resultierend aus der Marktanalyse. Die nachfolgenden Debuggingsysteme unterstützen das Debugging von Java-Anwendungen.

4.3.2.1. jBixbe

jBixbe [jBixbe] ist ein kommerzieller stand-alone Debugger für Java Programme, basierend auf dem Java Virtual Machine Tool Interface (JVM TI, siehe 4.3.1.5). jBixbe bietet zu dem traditionellen Quellcode-Debugging die Veranschaulichung von objektorientierten Aspekten unter Verwendung von UML-Diagrammen (Unified Modeling Language-Diagrammen). Durch die objektorientierte Darstellung soll eine enge Verbindung zur Entwurfsphase des Programms erreicht werden, um die Lokalisierung von Fehlverhalten in komplexen Systemen zu beschleunigen. jBixbe besteht aus drei Bestandteilen:

1. Quellcode Debugging: Der jBixbe Quellcode Debugger ist vergleichbar mit einem traditionellen Quellcode Debugger, der es ermöglicht, die Zielanwendung Schritt für Schritt („step in2, „step over“ etc.) zu verfolgen. Zudem unterstützt der Quellcode Debugger das Debuggen von Multi-Threading, die separat überprüft werden.
2. Visualisierung der Objektbeziehungen: Das UML-Klassendiagramm wird verwendet, um Objekt- und Datenstrukturen zu visualisieren. jBixbe erkennt Beziehungen zwischen Objekten, Null-Referenzen, beobachtet Wertänderungen, etc. und stellt diese in Form eines Klassendiagramms dar.
3. Visualisierung des Nachrichtenaustauschs zwischen Objekten: Für diese Visualisierung des Aufrufstack eines Threads erstellt jBixbe ein UML-Sequenzdiagramm. Dieses Sequenzdiagramm stellt u.a. Methodenaufrufe inklusive Parameter und Rückgabewerten, sowie geworfene Exceptions dar.

4.3.2.2. The Java Debugger (JDB)

JDB [JDB] ist ein Kommandozeilen-Debugger für Java-Anwendungen, basierend auf der Java Platform Debugger Architecture (JPDA). Der Debugging-Prozess wird interaktiv über eine Kommandozeile durchgeführt. Es existieren mehrere Alternativen, eine JDB-Sitzung zu starten. Für die einzelnen Alternativen wird auf die Quelle [JDB] verwiesen. Im Anschluss an das Starten der JDB-Sitzung kann unter Verwendung der verfügbaren Befehle der Debugging-Prozess durchgeführt werden. Es werden Befehle bereitgestellt, um Breakpoints zusetzen, Debugging-Prozess im Einzelschrittmodus auszuführen oder die Inhalte von Variablen und Objekten genauer zu untersuchen. Die Durchführung des Debugging-Prozesses unter Verwendung von JDB ist im Gegensatz zu anderen Debuggern komplizierter, da JDB lediglich das Debuggen unter Verwendung der Kommandozeile zulässt.

4.3.2.3. JSwat

JSwat [JSwat] ist ein grafisches Java Debugger Front-End, gestützt auf der Java Platform Debugger Architecture (JPDA) und basierend auf der NetBeans Plattform. Es handelt sich um eine Open-Source Software, die in binärer Form und als Quellcode frei verfügbar ist. Die Installation gestaltet sich aufgrund des vorhandenen Installations-Managers sehr einfach. Der Installations-Manager entpackt die JSwat-Dateien (.jar-Datei) in einen vom Benutzer gewählten Ordner und führt automatisiert einige Anpassungen durch, um die Konfigurationsdatei auf die Umgebung abzustimmen. Die Durchführung des Debugging-Prozesses erfolgt über einen Code-Navigator, der das aktuelle Statement im Quellcode farbig markiert. Über Breakpoints oder Einzelschrittfunktionen besteht die Möglichkeit zu navigieren, um den Quellcode

zu inspizieren. Der Byte-Code-Viewer und der Aufruf-Stack bieten dem Benutzer weitere Informationsquellen zur Fehlerlokalisierung, wie das Anzeigen von aktuell geladenen Klassen und Variablen.

4.3.3. Evaluation Debugging

Anschließend folgt eine zusammenfassende Bewertung aller Debuggingsysteme, gewonnen aus den existierenden Ansätzen und der Marktanalyse. Die Bewertungskriterien basieren auf den Anforderungen, die in Kapitel 3 dargestellt wurden.

Bewertungskriterien/Debugging	DRT	JADE	JDAS	JIVE	JPDA	jBixbe	JDB	JSwat
Anwendbar für alle Java-Applikationen	++	++	++	++	++	++	++	++
Einfache Integrierbarkeit	+-	- -	+-	+-	+-	+	++	++
Automatische Fehlerlokalisierung	+	+	-	-	-	- -	- -	- -
Debugging auf dem ursprünglichem Stand	- -	- -	- -	- -	- -	- -	- -	- -
Herkömmliches Debugging	- -	+	+	++	+	++	+-	+
Gegenüberstellung der Historie zum aktuellen Stand	- -	- -	- -	- -	- -	- -	- -	- -

Tabelle 4.3.: Evaluation der Debuggingsysteme und -ansätze

5. Konzept und Architektur

Als erstes entwickelt die Autorin eine Konzeptidee, basierend auf den Anforderungen im Kapitel 3. Im Anschluss folgt aus der Konzeptidee das Extrahieren aller internen Komponenten. Durch die Diskussion der Verteilung aller Komponenten wird die endgültige Architektur erarbeitet. Für ein besseres Verständnis werden anschließend die internen Abläufe erläutert. Das Kapitel schließt mit der Darstellung des konkreten Systemmodells ab.

5.1. Konzeptidee

Die Konzeptidee beschäftigt sich mit allgemeinen Konzeptfragen, die Auswirkungen auf das konkrete Konzept und die Architektur hat.

5.1.1. Laufzeitfehler in Java

In den Java-Anwendungen sollen Laufzeitfehler erkannt und lokalisiert werden. Mit dieser Problemstellung beschäftigt sich die erste Konzeptfrage. Es stellt sich die Frage, in welcher Art und Weise das Framework die Laufzeitfehler erkennen soll, ohne dass vorab eine Spezifikation oder ähnliches vom Benutzer definiert wurde.
Das Java-Sprachkonzept beinhaltet ein fest-integriertes Fehlerkonzept, das unter dem Namen „Ausnahmesituation“ oder im Englischen „Exception“ geführt wird. Alle Ereignisse, die zur Laufzeit im Java-Programm den ursprünglichen Ablauf sabotieren, werden als Ausnahmesituation bezeichnet. Die Ursachen einer Ausnahmesituation können sehr unterschiedlicher Natur sein, wie z.B. fehlerhafte Benutzereingaben oder beliebige Programmierfehler.
Das Fehlerkonzept in Java kann wie folgt zusammengefasst werden: Wenn Ausnahmesituationen auftreten, erzeugt die betroffene Methode je nach Fehlerart eine Instanz der Unterklasse `Throwable`. `Throwable` verfügt über genau zwei Unterklassen; `Error` und `Exception`, die wiederum weitere Unterklassen besitzen [Gosling u. a. (2005)] (siehe Abbildung 5.1).

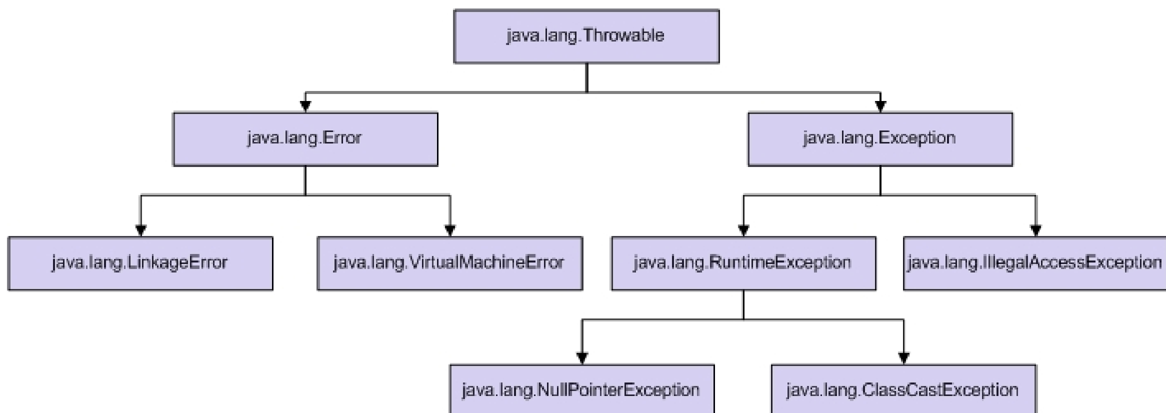

Abbildung 5.1.: Exception-Hierarchie

Error
Verursachte Fehler durch das Laufzeitsystem, wie z.B. ein fehlerhaftes Einbinden einer Klassenbibliothek, werden unter der Klasse `Error` zusammengefasst. Instanzen der Klasse `Error` und Instanzen der Unterklasse von `Error` werden im Java-Programm weder behandelt, abgefangen noch geworfen. Entwickler sind nicht befugt, eigene Fehler zu entwickeln, die von der Klasse `Error` erben (im Gegensatz zu Exceptions) [Gosling u. a. (2005)].

Exception
`Exception` ist die Basisklasse aller anderen zur Laufzeit auftretenden Fehler. Zudem wird Entwicklern die Möglichkeit gegeben, eigene `Exception`-Klassen zu definieren, diese müssen jedoch immer von der Klasse `Exception` erben.

Konzeptidee
Das in der Masterarbeit erarbeitete Framework wird basierend auf dem Fehlerkonzept von Java realisiert. Laufzeitfehler werden als Unterklassen von `Throwable` abgebildet. In dieser Masterarbeit umfasst der Begriff Exception nicht nur die Java `Exception` und deren Unterklassen, sondern ebenfalls das Java `Error` und deren Unterklassen.

5.1.2. Gruppierung der Laufzeitfehler

Im Anwendungsfall 3.2 setzt die dargestellte Fehlerverwaltung unter anderem eine Gruppierung der Laufzeitfehler voraus. Die Gruppierung der Laufzeitfehler wird benötigt, um Prioritäten aufgetretener Laufzeitfehler bestimmen zu können und einen Fehler der entsprechenden verantwortliche Instanz zuweisen zu können. An dieser Stelle tritt die nächste wichtige Frage auf. Wie können die unvorhersehbaren Laufzeitfehler gruppiert werden?

Die Anlehnung der Laufzeitfehlererkennung an das Fehlerkonzept von Java kann weitere Vorteile hervor rufen. Es besteht die Möglichkeit, die Klassenhierarchie der Ausnahmesituationen (Exceptions) als Gruppierung der Laufzeitfehler zu übernehmen. Infolgedessen könnte z.B. definiert werden, dass alle auftretenden Fehler der Unterklasse `RuntimeException` höchste Priorität haben, und die Instanz XY über diesen Fehler informiert wird.
Die Klassenhierarchie für alle Ausnahmesituationen muss bekannt sein, bevor das Framework die ersten Laufzeitfehler erkennt. Die Ausnahmeklassen sind in der Zielanwendung enthalten. Zudem sind Exceptionklassen in den eingebundenen Klassenbibliotheken oder in integrierten Frameworks der Zielanwendung vorhanden. Um alle Ausnahmeklassen in eine Klassenhierarchie (eine Gruppierung) einordnen zu können, müssen alle Exceptions bekannt sein. Vor diesem Hintergrund ergibt sich eine weitere Anforderung: das Filtern aller Exceptionklassen, die auftreten können, und die Anordnung dieser Exceptions in eine Klassenhierarchie.

Konzeptidee
Mit Hilfe von Reflections besteht die Möglichkeit, zur Laufzeit Modifikationen an Klassen oder deren Instanzen vorzunehmen. Zudem können Klassen zur Laufzeit nachgeladen werden. Dieses Feature kann verwendet werden, um alle Klassen der Zielanwendung zu laden und, gestützt darauf, die Filterung der Exceptionklassen (Unterklasse von `Throwable`)) durchzuführen. Des Weiteren können Klassenbibliotheken oder Frameworks mit Hilfe von Reflections geladen werden, ungeachtet dessen, ob sie in kompilierter Form vorliegen. Die Konzeptidee stellt sich wie folgt dar;

- Als erstes müssen die Speicherorte der Zielanwendung, der eingebundenen Klassenbibliotheken und integrierten Frameworks bekannt sein.
- Im Anschluss muss eine Iteration über alle Speicherorte erfolgen und Dateien mit dem Präfix „.java“ und „.class“ gefiltert werden.
- Jede unkompilierte (*.java) und kompilierte(*.class) Javaklasse wird mit Hilfe von Reflections geladen.
- Daraufhin muss geprüft werden, ob es sich bei der geladenen Klasse um eine Unterklasse von `Throwable` handelt und somit um eine Exception (oder um einen Error).
- Abschließend werden die gefilterten Exceptionklassen in eine Klassenhierarchie eingeordnet.

5.1.3. Exceptions erkennen und protokollieren

Im Anschluss an eine Klärung der Frage, wie Laufzeitfehler abgebildet und gruppiert werden können, entsteht eine weitere Frage: Wie werden Exceptions programmatisch erkannt und erfasst?
Einen Lösungsansatz bietet das AOP 4.2.1.1, welches bereits in den existierenden Ansätzen von Monitoringsystemen vorgestellt wurde. Das Prinzip des aspektorientierten Programmierens ist das Einfügen eines Programmteils an beliebigen Stellen in den Quellcode. Dies hat den Vorteil, die Funktionalitäten unabhängig von der Fachdomäne konzeptionell zu trennen, wie z.B. das Protokollieren von Exceptions. Des Weiteren unterstützt AOP das Einfügen bzw. Ausführen von Funktionen unmittelbar nach dem Auftreten einer Exception. Insofern besteht die Möglichkeit, mittels AOP nach jeder aufgetretenen Exception, diese programmatisch zu erfassen und weitere Aufgaben, wie z.B. das Benachrichtigten der verantwortlichen Instanz über den aufgetretenen Fehler, anzusteuern.

Konzeptidee
Das Framework wird in der Architektur das AOP-Konzept beinhalten, um auftretende Exceptions programmatisch zu erfassen und alle mit der Exception abhängigen Aktionen anzusteuern.

5.1.4. Laufzeitfehler reproduzieren

Wie bereits in den Grundlagen des Debugging erwähnt, treten Fehler in einer laufenden Anwendung am häufigsten auf. Zudem gestaltet sich die Reproduzierbarkeit dieser Fehler nicht einfach. Aus diesem Grund stellt sich die folgende Frage: Wie kann die Reproduktion von Laufzeitfehlern besser unterstützt werden?
Die meisten Debugging Ansätze konzentrieren sich darauf; ein fehlerhaftes Programmstück erneut laufen zu lassen mit den gleichen Parametern, die zum Fehler geführt haben. Bedauerlicherweise ist in vielen Fällen die Ursache des Fehlers anderer Natur, und ein Laufzeitfehler könnte auf dieser Art und Weise schwer reproduziert werden. Aus diesem Grund ist die Überlegung, so viele Informationen wie möglich, bereits, während die Exception auftritt, zu sammeln, um zu einem späteren Zeitpunkt auf dem ursprünglichen Zustand zu debuggen. Für diesen Ansatz ist das Protokollieren der aufgerufenen Methoden vor und nach der ausgelösten Exception von großer Bedeutung. Die Methodeninformationen sollten detaillierte Informationen umfassen, wie z.B. welche Parameter und welcher Rückgabewert verwendet wurden. Die Objekte sollten zudem bis ins Detail angesehen werden können.
Das Loggen von aufgerufenen Methoden ist performance- und speicherlastig, jedoch wird das triviale Log-Verfahren (z.B. log4 [log4j]) zum Protokollieren von Methodennamen und weiteren Informationen oft verwendet. Daraus schließt die Autorin, dass der produzierte Overhead akzeptabel ist. Es handelt sich hierbei allerdings um eine flache Speicherung

der Daten. In dieser Masterarbeit ist jedoch die Abspeicherung von komplexen Objekten von großem Interesse. Mit Hilfe der abgespeicherten Objekte soll das historische Debugging äquivalent zum herkömmlichen Debugging durchgeführt werden können. Somit sollen Objekte, die während der tatsächlichen Fehlersituation vorhanden waren, bis ins Detail einsehbar sein.

Konzeptidee
Zu diesem Zweck kommen die Ansätze der Trace-Debugging-Methodik 2.3.4.1 zum Einsatz. Alle aufgerufenen Methoden werden zur Laufzeit abgespeichert. Zur Performance- und Speicherlastminimierung wird lediglich für jeden Methodenaufruf der Methodenname, die Methodensignatur, die Parameter und die Rückgabewerte abgespeichert. Lokale Variable innerhalb der Methode werden ignoriert. Bei den Parametern und den Rückgabewerten handelt es sich fast immer um Objekte (mit Ausnahmen der primitiven Datentypen), die zum späteren Zeitpunkt vom Wartungspersonal bis ins Detail einsehbar sein sollen. Mit Hilfe von Reflections besteht die Möglichkeit, Objekte zu laden und zu betrachten. Da die Parameter und die Rückgabewerte vor der Speicherung in die Datenbank als Objekt verfügbar sind und nach dem Abruf aus der Datenbank wieder als Objekt benötigt werden, folgt die Abspeicherung dieser Daten als Objekte. Infolgedessen entfällt das Zerlegen und Rekonstruieren der Objekte.
Es wird zudem ein Mechanismus eingeführt, der veraltete Methoden löscht, die keinen Bezug zu einer aufgetretenen Exception haben.

5.1.5. Debugging Funktionalitäten

Für die Umsetzung einiger Anwendungsfälle, z.B. A.2 werden die Funktionalitäten eines herkömmlichen Debuggers für Java-Anwendungen benötigt. Es stellt sich die Frage, wie dies umgesetzt werden kann. Im Zusammenhang mit den existierenden Ansätzen wurde bereits im Abschnitt 4.3.1.5 JPDA vorgestellt. Die alleinige Nutzung von JPDA für die gesamte Problemlösung dieser Masterarbeit erwies sich nicht als geeignet, wie in der Tabelle 4.3 ausgeführt. Jedoch besteht die Möglichkeit, JPDA als einen Teil der Problemlösung zu betrachten. JPDA bietet eine schnelle und flexible Möglichkeit, eine Debugger-Anwendung zu erstellen. Durch die vielen in JPDA mitgelieferten APIs und der Referenzimplementierung gestaltet sich die Realisierung einzelner herkömmlicher Debugging-Funktionalitäten einfach.

Konzeptidee
In die Architektur des Frameworks wird JPDA integriert, um benötigte Debugging-Funktionalitäten zu realisieren und um eine Java-Anwendung im Debugmodus ausführen zu können.

5.1.6. Versionsverwaltung

Informationen über eine auftretende Exception umfassen u.a. die Version der Java-Zielanwendung, in der die Exception aufgetreten ist. Mit Hilfe der Versionsinformationen ist es dem Wartungspersonal möglich, die entsprechende Version der Java-Zielanwendung für den Debuggingvorgang zu wählen, um das Problem gezielt beseitigen zu können. Zudem soll es dem Wartungspersonal ebenso möglich sein, die aufgetretene Exception auch auf anderen Versionen beobachten zu können, um zu entscheiden, inwieweit die auftretende Exception versionsabhängig ist. Für diesen Funktionsumfang ist eine Verwaltung der Versionen notwendig. Ebenso besteht die Möglichkeit, unter Verwendung einer Versionsverwaltung die Codemodifikation zu unterstützen. Das Wartungspersonal kann Codes jeder Version modifizieren und bei Bedarf den alten Zustand wieder herstellen. Die Notwendigkeit einer Versionsverwaltung ist nicht verwunderlich; heute ist in vielen Bereichen der Softwareentwicklung die Verwaltung von Versionen unabdingbar. Aus diesem Grund existieren bereits einige bewährte und frei verfügbare Versionsverwaltungssysteme.

Konzeptidee
Für die Verwaltung von Versionen wird auf ein bereits existierendes System zurückgegriffen (siehe 6.1.1.4).

5.1.7. Webbrowser

Der Zugriff auf die Fehlerverwaltung muss zeit- und ortsunabhängig für das Wartungspersonal möglich sein. Dies kann durch das Realisieren der Fehlerverwaltung über eine Webanwendung erreicht werden. Mit Hilfe dieser Eigenschaft ist es dem Wartungspersonal möglich, sich bei einer Benachrichtigung über eine auftretende Exception sofort einen Einblick in die Problematik zu verschaffen. Es ist lediglich ein Internetzugang einschließlich eines Internetbrowsers notwendig, der heutzutage auf jedem internetfähigem Gerät gegeben ist.

Konzeptidee
Aus diesem Grund muss in der Architektur berücksichtigt werden, dass mindestens dieser Teil des Frameworks als Webanwendung fungiert.

5.2. Architektur

In diesem Abschnitt wird die endgültige Architektur für die Problemstellung dieser Masterarbeit erstellt. Als erstes werden alle internen Komponenten extrahiert und präsentiert. Der zweite Teil bezieht sich auf die physikalische Verteilung der Komponenten und somit auf die

Prägung des Architekturstils.

5.2.1. Komponentenübersicht

Basierend auf den Anforderungen und der Konzeptidee werden nachfolgend Komponenten extrahiert und kurz erläutert. In diesem Abschnitt werden lediglich die internen fachlichen Komponenten aufgeführt. Die externen Komponenten, Systeme und Technologien, wie z.B. AOP, werden in den Architekturstilen 5.2.2 wieder aufgenommen. Vorab folgt in Abbildung 5.2 ein Überblick über alle fachlichen Komponenten.

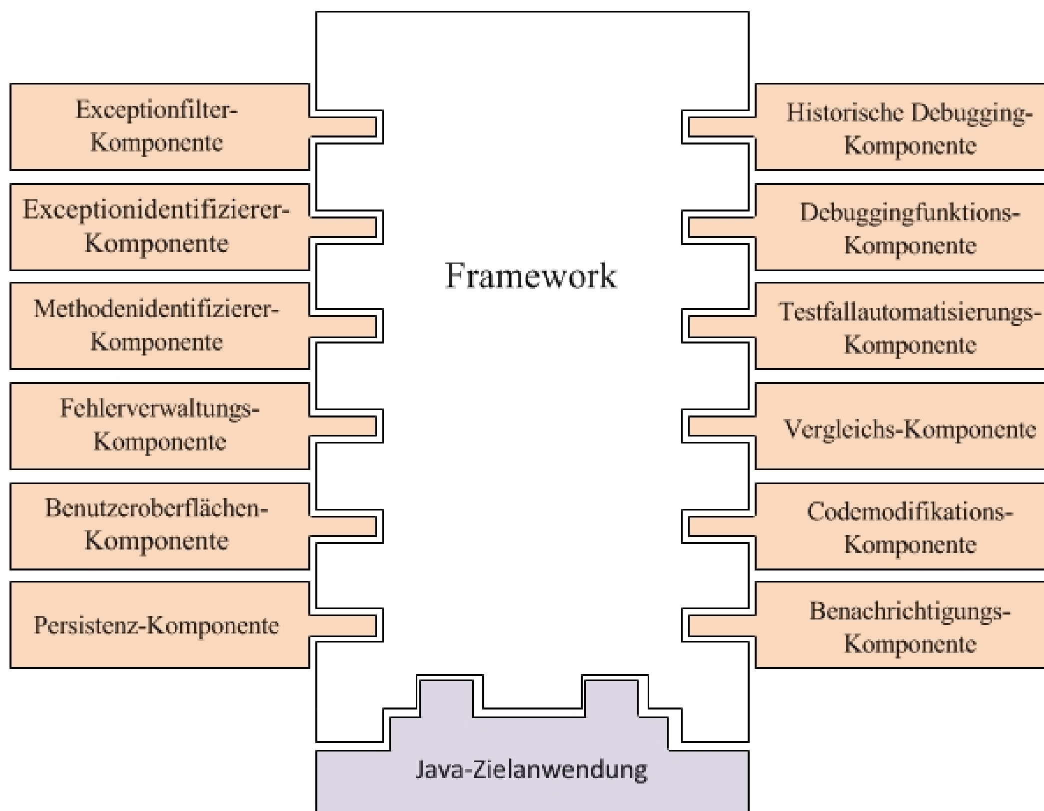

Abbildung 5.2.: Fachliche Komponenten

Exceptionfilter-Komponente
Die Exceptionfilter-Komponente ist dafür zuständig, alle Exceptions, die in einer Zielanwendung auftreten können, zu filtern. Des Weiteren müssen die gefilterten Exceptions in eine gemeinsame Klassenhierarchie eingruppiert werden.

Exceptionidentifizierer-Komponente
Diese Komponente muss alle in der Zielanwendung auftretenden Exceptions erkennen. Mithilfe von AOP wird das Identifizieren der aufgetretenen Exceptions durchgeführt. Zudem müssen weitere Aktionen initiiert werden. Die Aktionen umfassen das Persistieren der Informationen bezüglich der aufgetretenen Exception, sowie die Benachrichtigung der verantwortlichen Instanzen über den aufgetretenen Fehler.

Methodenidentifizierer-Komponente
Das Protokollieren aller aufgerufenen Methoden ist Teil dieser Komponente, wie auch das Löschen von protokollierten Methoden, die nicht länger relevant sind und infolgedessen nicht in Abhängigkeit zu einer aufgetretenen Exception stehen.

Fehlerverwaltungs-Komponente
Diese Komponente umfasst alle Funktionalitäten zur Fehlerverwaltung einschließlich der Einstellungen zu den Fehlergruppen. Die Funktionalitäten umfassen die Priorisierung von Exceptions, die Definition und Zuweisung verantwortlicher Instanzen, die Wahl und Definition der Benachrichtigungsalternativen über aufgetretene Exceptions, die De- und Aktivierung der Protokollierung von Exceptions und die Verwaltung aller aufgetretenen Exceptions (umfasst Anwendungsfall 3.2 und 3.4).

Benutzeroberflächen-Komponente
Die Benutzeroberfläche ist verantwortlich für die Interaktion zwischen Benutzer und System. Alle Benutzerinteraktionen werden über die Benutzeroberfläche angesprochen. Es sei erwähnt, dass der Fehlerverwaltungsbereich als Weboberfläche implementiert werden muss. In der Komponentenverteilung 5.2.2 wird die Frage beantwortet, ob die weiteren Benutzerinteraktionen ebenso über eine Weboberfläche geregelt werden.

Persistenz-Komponente
Die Persistenz-Komponente kapselt den direkten Datenbankzugriff. Vor der Datenbank wird eine Persistenz-Komponente platziert, um das darunter liegende relationale DBMS austauschbar zu konzipieren.

Historische Debgging-Komponente
Diese Komponente bietet Funktionen, um ein Debugging auf dem tatsächlichen Stand zum Zeitpunkt des Auftretens der Exception durchzuführen. Das bedeutet, es werden alle Informationen über die aufgetretene Exception sowie alle in der Realität aufgerufenen Methoden vor und nach dem Auftreten der Exception präsentiert. Mit Hilfe dieser Komponente kann das Wartungspersonal genau verfolgen, was zur Laufzeit passiert ist, als die Exception auftrat.

Debuggingfunktions-Komponente
Diese Komponente implementiert die JDI Schnittstelle der JPDA. Sie umfasst alle Funktionalitäten, die für das Ausführen der Zielanwendung im Debugmodus notwendig sind.

Testfallautomatisierungs-Komponente
Das Wartungspersonal soll die Möglichkeit bekommen, einen Testfall an der Stelle des Auftretens der Exception zu simulieren. Das bedeutet, dass das Wartungspersonal den Bereich in der Java-Zielanwendung festlegt, der im Debuggingmodus ausgeführt werden soll. Die Startwerte umfassen die Werte aus der Historie, die ursprünglich beim Auftreten des Fehlers vorhanden waren. Die Testfallautomatisierungs-Komponente umfasst alle Funktionalitäten, die für die Erfüllung dieser Anforderung notwendig sind.

Vergleichs-Komponente
Diese Komponente veranschaulicht den Vergleich eines Ablaufs im Debuggingmodus mit den ursprünglichen Informationen der aufgetretenen Exception. Diese Komponente beinhaltet alle erforderlichen Vergleichsfunktionen. Damit ist ein Abgleich eines vom Benutzer ausgewählten Testfalls mit dem historischen Methodenverlauf, dem Zeitpunkt, als die Exception aufgetreten ist, möglich.

Codemodifikations-Komponente
Die Codemodifikations-Komponente ist dafür zuständig, dem Wartungspersonal einen direkten Zugriff auf den Sourcecode zu ermöglichen. Über den direkten Zugriff kann der Fehler vom Wartungspersonal behoben werden. Diese Komponente ist ebenso verantwortlich dafür, bei Bedarf den ursprünglichen Zustand bzw. Sourcecode wiederherzustellen.

Benachrichtigungs-Komponente
Auftretende Fehler werden an die verantwortliche Instanz signalisiert. Die Aufgabe der Benachrichtigungs-Komponente ist das Informieren der verantwortlichen Instanz über den aufgetretenen Fehler durch das vordefinierte Benachrichtigungsmedium (z.B.: E-mail, SMS).

5.2.2. Komponentenverteilung

Die Definition der internen Komponenten für die Architektur ist erfolgt. Als nächster Schritt wird die Verteilung der Komponenten vorgenommen. Diese ist Bestandteil der Erarbeitung der geeigneten Architektur. Werden beispielsweise alle Komponenten eines Systems zusammen in einer Einheit gebündelt, wird eine Stand-alone Architektur erzielt. Bei einer Verteilung auf zwei separate Teilsysteme ist von einer Client-Server Architektur die Rede, falls eine Seite Dienste anbietet und die andere Seite diese nutzt, bzw. von einer nicht geschichteten Client-Server Architektur die Rede, falls beide Seiten die gegenseitigen Dienste nutzen, siehe Abbildung 5.3. Alle Architekturvarianten besitzen zudem eine Datenbank, die ebenfalls Bestandteil des Frameworks ist. Bevor die Überlegungen und Diskussionen über die Verteilung präsentiert werden, muss berücksichtigt werden, dass das Framework oder zumindest ein Teil des Frameworks auf die Java-Zielanwendung aufgesetzt werden muss. Die Protokollierung der Methodenaufrufe sowie der aufgetretenen Exceptions setzt

den direkten Zugriff auf die Java-Zielanwendung voraus. Vor diesem Hintergrund ergibt sich ein wichtiges Ziel, welches bei der Erstellung der Architektur berücksichtigt werden muss. Auf die Java-Zielanwendung wird eine Last aufgesetzt, die den gewöhnlichen Arbeitsablauf der Zielanwendung bis zu einem bestimmten Punkt beeinträchtigen darf. Das Framework muss gegenüber der Java-Zielanwendung transparent gehalten werden. Nachfolgend wird auf die Verteilung bzw. Positionierung jeder einzelnen Komponente eingegangen und die Auswirkung auf bzw. die Wechselwirkung mit anderen Komponenten diskutiert. Nach der Verteilungsdiskussion jeder einzelnen Komponente wird ein kurzes Fazit gezogen, und es wird definiert, an welcher Stelle die Komponente letztendlich positioniert wird.

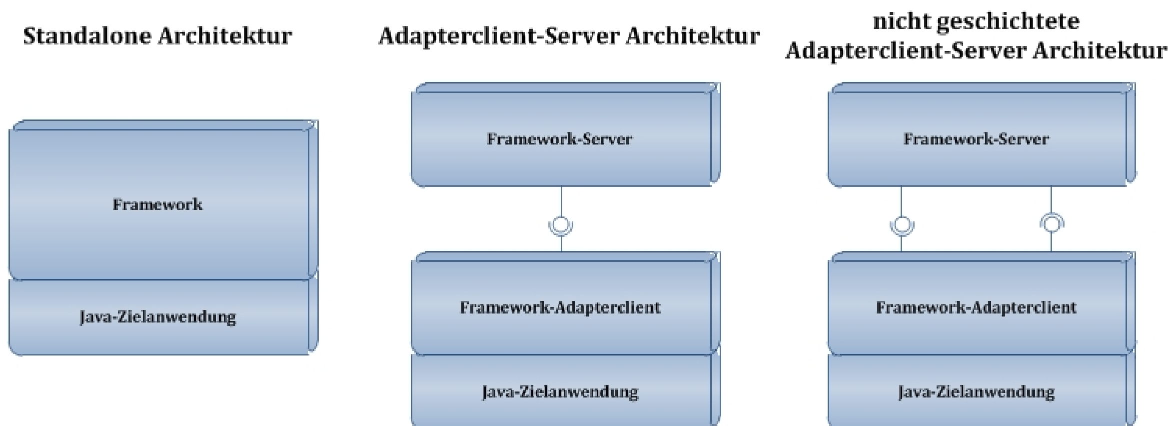

Abbildung 5.3.: Überblick über mögliche Architekturstile

Monitoring

Die Komponenten Methodenidentifizierer und Exceptionidentifizierer werden in einer Komponente Namens Monitoring zusammengefasst. Die gesamte Oberkomponente Monitoring muss auf der Adapterseite positioniert sein, da der direkte Zugriff auf die Java- Zielanwendung vorausgesetzt wird, um auftretende Exceptions und aufgerufene Methoden zu identifizieren. Die zweite Frage in diesem Zusammenhang stellt sich bezüglich der persistenten Speicherung dieser Informationen. An welcher Stelle sollen die Daten abgespeichert werden? In Anbetracht des sehr hohen Datenvolumens (u.a. alle Methodenaufrufe) sollte die Datenbank in der Nähe des Adapters positioniert werden, um die Streckendistanz der Übertragung zu minimieren. Aufgrund der enormen Datenmenge besteht die Gefahr eines Flaschenhals; diese Gefahr wird verstärkt durch große Netzstrecken. In diesem Zusammenhang könnte der Verdacht entstehen, das Problem nur umgeleitet, es jedoch nicht beseitigt zu haben. Sollten Komponenten, die die abgespeicherten Informationen verwenden, in großer Distanz zu dem Adapter und somit zur Datenbank stehen, würde die Übertragung der großen Datenmenge an eine andere Stelle versetzt. Bei dieser Vermutung sollte jedoch auch berücksichtigt werden, dass viele Informationen (z.B. Methodenaufrufe) irrelevant sind und nachträglich gelöscht werden. Infolgedessen werden sie also nicht verwendet und somit

auch nicht übertragen. Aufgrund dessen bestätigt sich der erwähnte Verdacht nicht, und die Positionierung der Datenbank in kurzer Distanz zu dem Adapter wird bestärkt.
Bei einem verteilten Framework bestünde die Möglichkeit ebenso, zwei Datenbanken in Betracht zu ziehen: eine Datenbank in der Nähe des Adapters und eine weitere in der Nähe des separaten Frameworkteils. Diese Überlegung wird aus mehreren Gründen verworfen; zum einen aufgrund der großen Datenschnittmenge, die eine konsistente Datenhaltung zur Folge hat. Zum anderen ist die tatsächliche Distanz zwischen Adapter und dem separatem Teil nicht bekannt, und demgemäß ist das Betreiben zweier Datenbanken nebeneinander nicht ausgeschlossen.

Fazit über die Verteilung der Monitoring-Komponente

- Monitoring auf dem Adapter
- Eine Datenbank, die sich in der Nähe des Adapters befindet

Benutzeroberfläche

Die Nutzung der Fehlerverwaltung und die Durchführung des Debugging erfordert eine Benutzeroberfläche. Die technischen Anforderungen 3.3 besagen, dass der Zugriff auf die Fehlerverwaltung zeit- und ortsunabhängig durchgeführt werden muss. Infolgedessen wurde in der Konzeptidee 5.1.7 das Ergebnis erzielt, den Zugriff über einen Webbrowser zu realisieren. Das Erreichen dieses Ziel setzt eine Webanwendung voraus. Das Aufsetzen einer Webanwendung inklusive Webserver etc. auf die Java-Zielanwendung würde diese zu sehr Belasten, und die Transparenz des Frameworks würde leiden. Aus diesem Grund wird das Framework verteilt und eine Stand-alone Architektur für diese Problemstellung verworfen. Zusammengefasst wird die geeignete Architektur aus drei Bestandteilen bestehen: dem Framework-Adapter, der Datenbank und dem separaten Frameworkteil, der als Webanwendung realisiert wird, siehe Abbildung 5.4.

Fazit über die Verteilung der Benuzteroberflächen-Komponente

- Benutzeroberfläche über die Webanwendung (separater Teil)

Exceptionfilter

Die Exceptionfilter-Komponente benötigt für die Filterung der Exceptions, die auftreten können, die Java-Zielanwendung. Für dieses Bestreben wird eine zweite separate Instanz der Java-Zielanwendung verwendet, die der Version im laufenden Betrieb entspricht. Es besteht die Möglichkeit, die Exceptionfilter-Komponente auf dem Adapter oder separat auf der Webanwendung zu platzieren. Die Ausführung der Funktionen dieser Komponente kommt selten zum Einsatz, allerdings ist sie sehr rechenintensiv. Aus diesem Grund steht einer Positionierung der Komponente auf der Webanwendung und der damit verbundenen Entlastung der Zielanwendung nichts entgegen. Die Speicherung der Daten weist ebenfalls keine erkennbare Problematik auf. Die Datenmenge ist überschaubar, und auch bei langer Netzstrecke zwischen der Datenbank und der Webanwendung würde keine Erschwernis

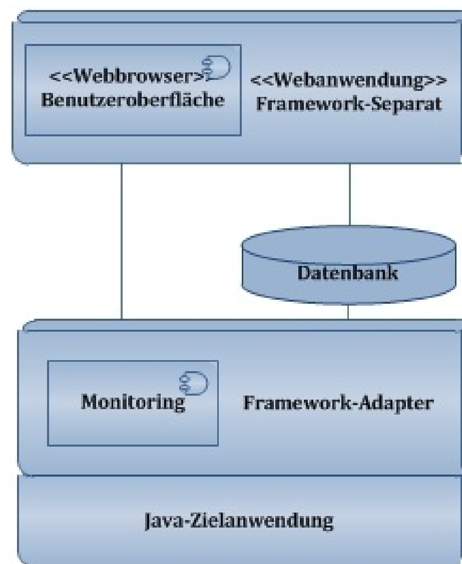

Abbildung 5.4.: Zwischenschritt bei der Architekturerstellung

hervor gerufen werden.
Fazit über die Verteilung der Exceptionfilter-Komponente
- Exceptionfilter auf der Webanwendung

Fehlerverwaltung
Wie bereits erwähnt, wird die Fehlerverwaltung über einen Webbrowser ansprechbar sein. Aus diesem Grund befindet sich die Fehlerverwaltungs-Komponente auf der Webanwendung.
Fazit über die Verteilung der Fehlerverwaltungs-Komponente
- Fehlerverwaltung auf der Webanwendung

Benachrichtigung
Die Benachrichtigungs-Komponente wird durch eine auftretende Exception angetriggert, welche vom Monitoring erkannt wird. Für die Ausführung ihrer Aufgabe benötigt die Benachrichtigungs-Komponente Informationen über die auftretende Exception und deren Einstellungen. Die Informationen bezüglich der auftretenden Exception werden direkt von der Monitoring-Komponente geliefert. Die Exceptioneinstellungen sind in der Datenbank abgespeichert und können von dort abgerufen werden. Die Benachrichtigungs-Komponente arbeitet mit einem externen Nachrichtenvermittlungssystem zusammen. Unter Berücksichtigung der oben erwähnten Eigenschaften ist die Platzierung der Benachrichtigungs-Komponente auf der Seite der Webanwendung sinnvoll. Die Benachrichtigungs-Komponente und infolgedessen ebenfalls die Kooperation mit einem externen Nachrichtenvermittlungs-

system wird von der Java-Zielanwendung separiert und stellt eine Entlastung der Zielanwendung dar. Die geringen Datenmengen bezüglich der Exceptioneinstellungen können direkt von der Datenbank abgerufen werden. Informationen über die aufgetretene Exception werden vom Adapter übertragen, sobald eine auftretende Exception erkannt wurde.
Fazit über die Verteilung der Benachrichtigungs-Komponente

- Benachrichtigung auf der Webanwendung

Debugging
Die Komponenten historisches Debugging, Debuggingfunktion, Codemodifikation, Vergleich und Testfallautomatisierung werden in der Komponente Debugging zusammengefasst. Um den gesamten Funktionsumfang der Debugging Komponente realisieren zu können, wird u.a. JPDA verwendet. JPDA greift auf die virtuelle Maschine der Java-Zielanwendung über JVM TI zu, um über JDI nützliche Funktionen für das Debugging anbieten zu können. Mit Hilfe eines Versionsverwaltungssystem besitzt das Wartungspersonal die Möglichkeit, die benötigte Version der Java-Zielanwendung auszuwählen, auf der das Debugging ausgeführt werden soll.
Die Debugging-Komponente könnte zum einen auf dem Adapter und zum anderen auf der Webanwendung positioniert werden. Die Platzierung der gesamten Komponente auf die Webanwendung bringt einige Vorteile. Die Transparenz des Frameworks gegenüber der Java-Zielanwendung wird nicht gefährdet. Der Ausmaß der Debugging Komponente ist angemessen groß, und zudem arbeitet die Debugging Komponente mit zwei Fremdsystemen (JPDA und Versionsverwaltung) zusammen. Der zweite Vorteil liegt darin, die Vorzüge der Webanwendung auf das Debugging zu übertragen. Es besteht die Möglichkeit, die gesamte Benutzeroberfläche für die Debugging-Komponente als Weboberfläche zu realisieren und den Funktionsumfang somit Zeit- und ortsunabhängig zu gestalten.
Fazit über die Verteilung der Debugging-Komponente

- Debugging auf der Webanwendung

Zusätzliche Komponenten
Unabhängig vom Funktionsumfang werden weitere Komponenten benötigt. Der Framework-Adapter sowie die Webanwendung greifen auf die Datenbank zu. Aus diesem Grund verfügen beide Frameworkbestandteile über eine Persistenz-Komponente. Die Persistenz-Komponente ist für die Interaktion mit der Datenbank zuständig. Zudem benötigen beide jeweils eine Kommunikations-Komponente, die den Kommunikationsaustausch innerhalb des Frameworks regelt.
Fazit über die Verteilung weiterer Komponenten

- Persistenz-Client-Komponente auf dem Adapter
- Persistenz-Server-Komponente auf der Webanwendung
- Kommunikations-Client-Komponente auf dem Adapter

- Kommunikations-Server-Komponente auf der Webanwendung

Der separate Frameworkteil bietet für die Webanwendung Dienste an, die vom Framework-Adapter genutzt werden. Umgekehrt werden jedoch keine Dienste angeboten und genutzt. Aus diesem Grund wird von einer Client-Server Architektur gesprochen. Des Weiteren handelt es sich um keine gewöhnliche Client-Server Architektur, weil die Clientseite als Adapter auf eine Java-Zielanwendung gesetzt wird. Infolgedessen erweiterte die Autorin die Architekturbezeichnung auf den Begriff der Adapterclient-Server Architektur. Zudem handelt es sich um eine mehrstufige Client-Server Architektur, da der Server seine Dienste mit Hilfe von anderen externen Diensten anbietet. Die Erarbeitung einer geeigneten Architektur für die Problemstellung dieser Masterarbeit ergab eine mehrstufige Adapterclient-Server Architektur, die nachfolgend mit Hilfe eines Komponentendiagramms in Abbildung 5.5 dargestellt wird.

5.3. Interne Aktivitäten

In diesem Abschnitt wird auf die erarbeitete Architektur eingegangen, und die internen Aktivitäten werden detailliert betrachtet. Die internen Aktivitäten des Frameworks können in drei separate Abläufe eingeteilt werden:

1. Exceptionfilterung: bezieht sich ausschließlich auf den Server
2. Fehlererkennung; bezieht sich auf den Client und den Serve
3. Fehlerlokalisierung und -behebung: bezieht sich ausschließlich auf den Server

Die Exceptionfilterung ist der statische Teil des Framework. Die Durchführung der Exceptionfilterung muss als erstes getätigt werden. Bei der Fehlererkennung sowie der Fehlerlokalisierung und -behebung handelt es sich um dynamische Abläufe. Nachfolgend wird auf jede Aktivität eingegangen und mit Hilfe je eines Aktivitätsdiagramms jeder Ablauf verdeutlicht.

5.3.1. Exceptionfilterung

Das Exceptionkonzept in Java ist sehr verbreitet und wird in der Praxis oft genutzt. Aus diesem Grund können in einer Zielanwendung Exceptions, die keinen realen Fehler darstellen, permanent auftreten. Als Beispiel sei eine vom Entwickler implementierte Exception erwähnt, die das Unterbinden von zu langen Wörtern in der Eingabe ausdrückt. Infolgedessen ist es sehr wichtig, im Vorfeld zu definieren, welche Exceptions relevant sind und welche nicht, um einen unnötigen Overhead zu minimieren. Die Problemlösung verfolgt den Ansatz, im Vorfeld

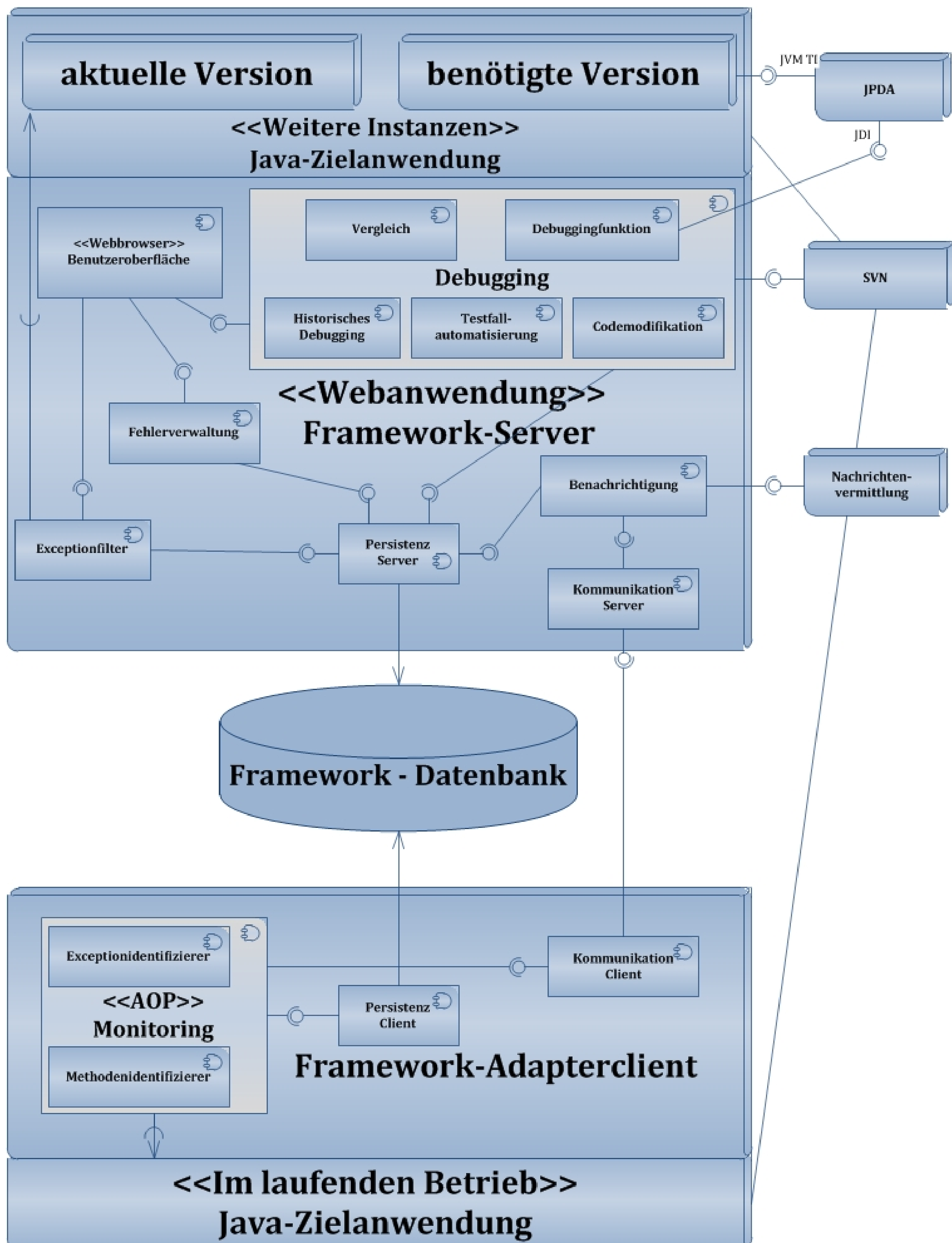

Abbildung 5.5.: Geeignete Architektur

Kenntnisse über alle Exceptions, die in der Zielanwendung auftreten können, zu erlangen, um zu definieren, welche relevant sind, und welche außer Acht gelassen werden können. Aus diesem Grund ist eine Filterung aller Exceptions aus der Zielanwendung ein wichtiger Teilbereich des Frameworks.

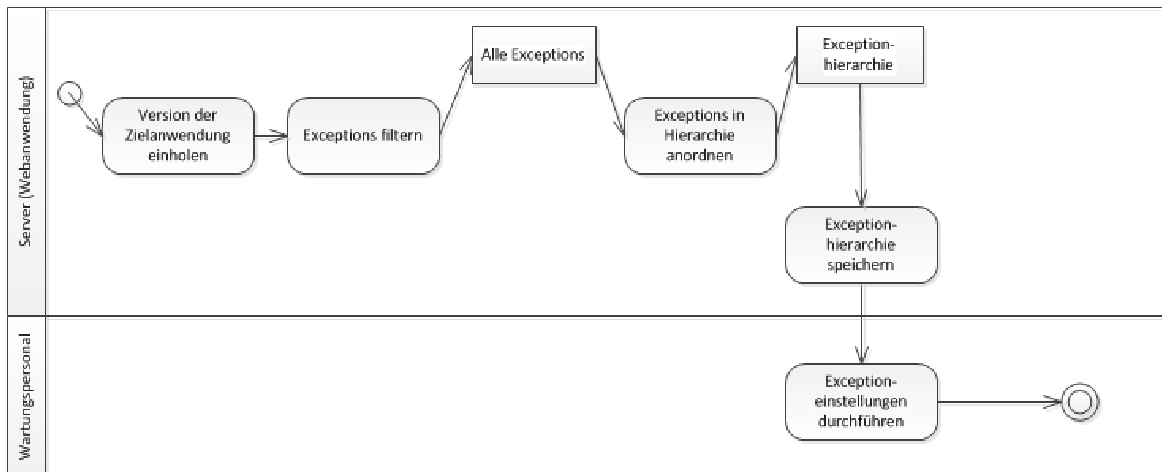

Abbildung 5.6.: Aktivitätsdiagramm: Exceptionfilterung

Vom Ablauf ist primär die Komponente Exceptionfilter betroffen, zudem werden Dienste der Fehlerverwaltung und der Benutzeroberfläche für die Aufgabe hinzugezogen. Wie in Abbildung 5.6 zusehen ist, beginnt der Filtervorgang mit dem Einholen der entsprechenden Instanz der Java-Zielanwendung. Die Instanz der Java-Zielanwendung muss der Version entsprechen, die sich im laufendem Betrieb befindet bzw. auf der das Framework aufgesetzt wird. Die Versionsverwaltung wird mit Hilfe eines Versionsverwaltungssystems durchgeführt. Im Anschluss werden alle Exceptions von der Exceptionfilter-Komponente aus der Zielanwendung gefiltert und in eine Klassenhierarchie eingeordnet. Die Klassenhierarchie dient zur Gruppierung der Exceptions. Anhand der Gruppierung können Exceptions spezifiziert werden. Nach der Exceptionfilterung und Einordnung werden diese Daten in der Datenbank gespeichert. Über die Benutzeroberfläche der Fehlerverwaltung erhält das Wartungspersonal die Möglichkeit, im Anschluss einzelne Exceptions oder Exceptiongruppen zu spezifizieren. Die Spezifikationen enthalten u.a.

- Aktivität der Exception; gibt Informationen, ob die aufgetretene Exception berücksichtigt werden soll.
- Priorität der Exception; gibt Auskunft über die Wichtigkeit der aufgetretenen Exception.
- verantwortliche Instanz; stellt die verantwortliche Instanz des Wartungspersonals dar, die kontaktiert werden muss.

- Kontaktdaten der verantwortlichen Instanz; gibt Auskunft über die Art und Weise der Kontaktierung, z.B. per SMS oder E-Mail, und die entsprechenden Kontaktdaten.

Es können Einstellungen einer Exception für alle Unterklassen dieser Exception übernommen werden. Anhand dieser Möglichkeit können Einstellungen für Exceptiongruppen vorgenommen werden. Das Wartungspersonal definiert z.B., dass die `java.sql.SQLException` und deren Unterklassen dem Datenbankpersonal zugewiesen werden sollen.

Die Exceptionfilterung muss mindestens einmal vor dem Einsatz des Frameworks durchgeführt werden bzw., bevor der Adapter anfängt, auftretende Exceptions zu erkennen. Die Einstellungen bezüglich der Exceptions müssen nicht unbedingt sofort bearbeitet werden. Bis zur Einstellung über eine Exception oder eine Exceptiongruppe sind alle Exceptions inaktiv und werde außer Acht gelassen. Zu bedenken ist, dass mindestens nach jedem Release eine Filterung durchgeführt werden muss, um hinzugekommene Exceptions aufnehmen und veraltete entnehmen zu können.

5.3.2. Fehlererkennung

Beim Ablauf der Fehlererkennung handelt es sich um die Identifizierung einer auftretenden Exception bis hin zur Benachrichtigung der entsprechenden Instanz über den registrierten Fehler. Der Fehlererkennungsvorgang umfasst die Monitoring-Komponente sowie die Benachrichtigungs-Komponente, zudem ist die direkte Kommunikation zwischen Client und Server ausschließlich für die Mitteilung der auftretenden Exception vorgesehen. Weitere Interaktionen zwischen Client und Server sind passiv über die Datenbank geregelt, d.h. der Client platziert Informationen, die vom Server eingeholt werden, in der Datenbank.

Abbildung 5.7 verdeutlicht den Ablauf der Fehlererkennung. Die Komponente Methodenidentifizierer ist dafür zuständig, aufgerufene Methoden in der Zielanwendung zu erkennen. Anschließend werden diese Informationen an die Persistenz-Komponente delegiert und in der Datenbank abgespeichert. Aufgetretene Exceptions werden durch die Exceptionidentifizierer-Komponente erkannt. Falls diese als aktiv gekennzeichnet sind, werden die Informationen bezüglich der aufgetretenen Exception zum einen in der Datenbank abgespeichert und zum anderen an den Server gesendet. Sobald der Server eine Nachricht über eine auftretende Exception empfängt, werden weitere notwendige Informationen über die Exception bzw. die Exceptioneinstellungen aus der Datenbank eingeholt und die verantwortliche Instanz über die aufgetretene Exception informiert.

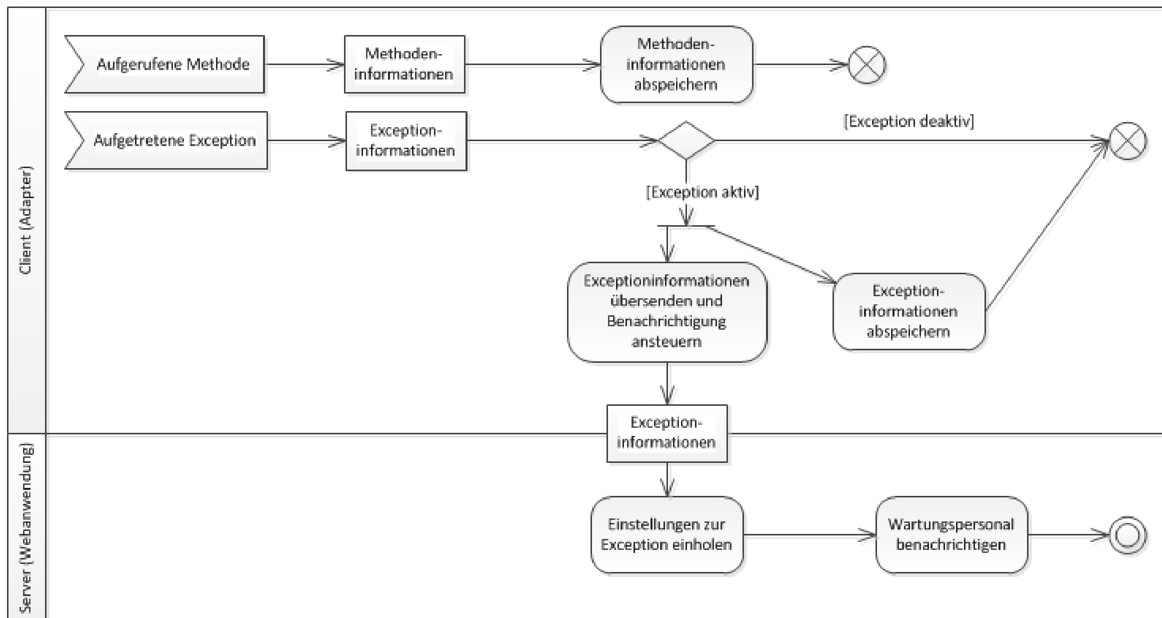

Abbildung 5.7.: Aktivitätsdiagramm: Fehlererkennung

5.3.3. Fehlerlokalisierung und -behebung

Im Bereich der Fehlerlokalisierung und -behebung spielen die Oberkomponente Debugging und teilweise die Benutzeroberflächen-Komponente, die Fehlerverwaltungs-Komponente und die Persistenz-Komponente eine wichtige Rolle, ebenfalls in Abhängigkeit von JPDA und dem Versionsverwaltungssystem. Dieser Ablauf konzentriert sich in der Hauptsache auf die Lokalisierung der Exception bzw. auf die Lokalisierung der Herkunft. Der Lokalisierungsvorgang beschränkt sich nicht auf eine Herangehensweise. Es werden drei Möglichkeiten für die Arbeitsweise zur Verfügung gestellt.

1. herkömmliches Debugging
2. Historisches Debugging
3. Vergleich (zwischen herkömmlichem und historischem Debugging)

Abbildung 5.8 zeigt mit Hilfe eines Aktivitätendiagramms die einzelnen Arbeitswege, gefolgt von einer Erläuterung. Es sei zu erwähnen, dass das Aktivitätsdiagramm einen „happy path“ darstellt, auf weitere Verzweigungen und Aktivitäten wird bewusst verzichtet, um die Übersichtlichkeit zu gewährleisten.

Wie in Abbildung 5.8 zu sehen ist, beginnt der Prozess mit der Auswahl der zu behandelnden Exception. Das Wartungspersonal hat die Möglichkeit, über die Benutzeroberfläche in

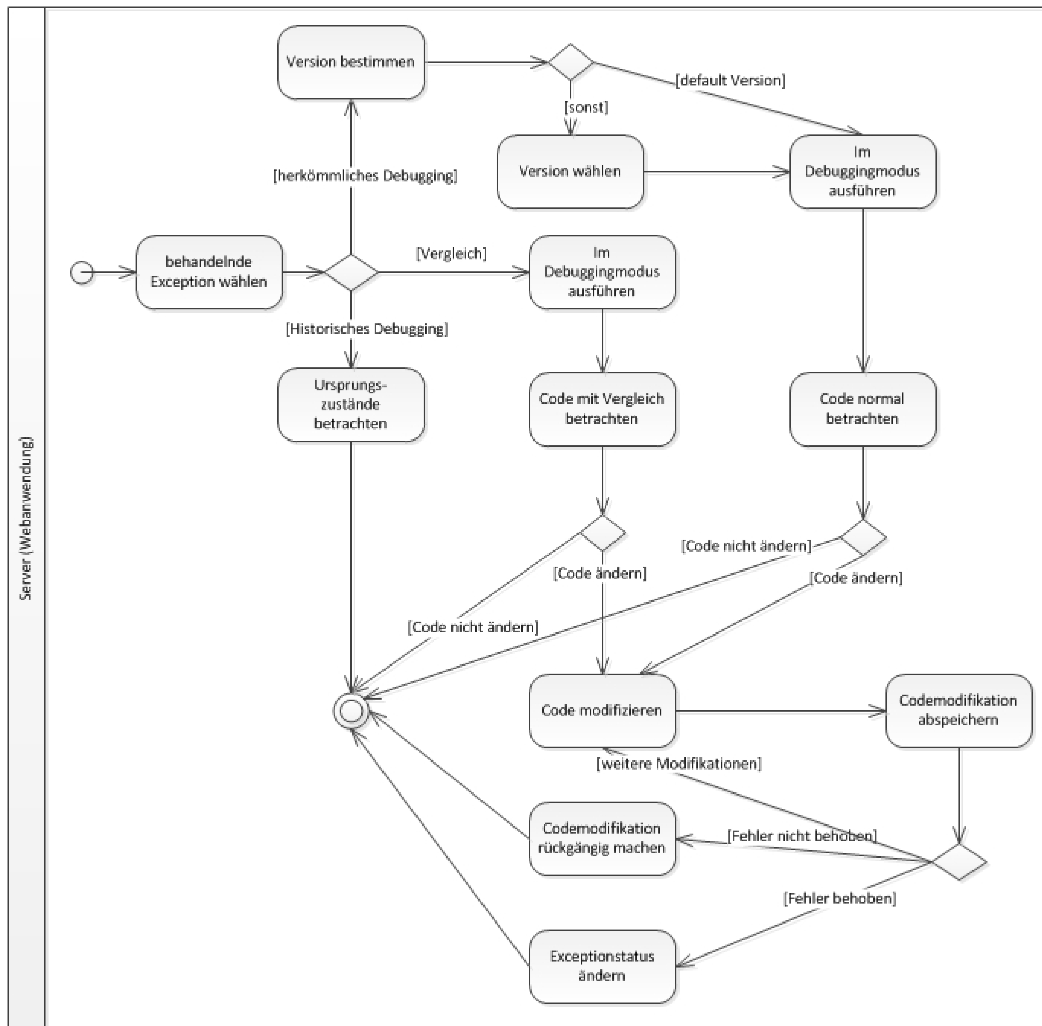

Abbildung 5.8.: Aktivitätsdiagramm: Fehlerlokalisierung und -behebung

der Fehlerverwaltung die aufgetretene Exception auszuwählen, die behandelt bzw. behoben werden soll. Im Anschluss kann das Wartungspersonal entscheiden, auf welche Art und Weise der Debuggingvorgang vollzogen werden soll. Wie erwähnt, stehen hierfür drei Alternativen zur Verfügung.
Das herkömmliche Debugging repräsentiert den normalen Debuggingvorgang. Die Anwendung wird in Gang gesetzt, um Schritt für Schritt den Code und die entsprechenden Zustände auf Ihre Richtigkeit zu überprüfen. Vorab hat das Wartungspersonal optional die Möglichkeit, eine Version der Zielanwendung auszuwählen. Zu diesem Zweck kommt das Versionsverwaltungssystem zum Einsatz. Standardmäßig wird die Version, in der die gewählte Exception aufgetreten ist, verwendet. Im Anschluss kann das Wartungspersonal wie gewohnt den Code unter Verwendung herkömmlicher Debugging Funktionalitäten, wie z.B. „step in“, „step

over“ betrachten. Das Wartungspersonal ist befähigt, den Code aktiv zu modifizieren. Die Codemodifikation kann im Anschluss gespeichert und bei Bedarf wieder storniert werden.
Das historische Debugging ist kein reales Debugging, es handelt sich hierbei um ein simuliertes Debugging. Der Vorgang ähnelt einem herkömmlichen Debuggingvorgang, sodass das Wartungspersonal herkömmliche Debugging Funktionen nutzen kann, um sich im Programmablauf bewegen und das Programm betrachten zu können. Der Unterschied besteht darin, dass hierbei keine Instanz der Zielanwendung aktiv ist, sondern, dass auf den historischen Zuständen gearbeitet wird. Beim Debuggen werden die Zustände angezeigt, die in der Realität aufgetreten sind, als die Exception im laufendem Betrieb ausgelöst wurde. In diesem Zusammenhang kann sich das Wartungspersonal einen genauen Einblick in die tatsächlichen Geschehnisse verschaffen.
Der Vergleich gibt dem Wartungspersonal die Möglichkeit, einen direkten Vergleich zwischen dem historischem Programmablauf und dem Resultat eines aktuellen Ablaufs im Debuggingmodus zu ziehen. Abweichungen werden deutlich erkennbar dargestellt. In diesem Zusammenhang sei erwähnt, dass das Wartungspersonal zu einer aktuell ausgeführten Methode den ursprünglichen Zustand betrachten kann, einschließlich der Parameter und Rückgabewerte, die vorhanden waren, als die Exception in der Vergangenheit aufgetreten ist. Beim Vergleich kann auf den Sourcecode zugegriffen werden, um Modifikationen durchführen zu können.

5.4. Konkretes Systemmodell

Dieses Unterkapitel präsentiert das konkrete Systemmodell. Zu diesem Zweck werden zwei Klassendiagramme vorgestellt. Das erste Klassendiagramm umfasst die Adapterseite des Frameworks, das zweite Klassendiagramm bezieht sich auf die Serverseite (Webanwendung). Keins der beiden Klassendiagramme bezieht sich konkret auf einen der drei vorgestellten Abläufe. Das Adapterclient-Klassendiagramm deckt einen Teil der Fehlererkennung ab. Der Server hingegen enthält den anderen Teil der Fehlererkennung und zudem die zwei weiteren Abläufe. Diese Aufteilung kann aus den Aktivitätsdiagrammen 5.6, 5.7 und 5.8 entnommen werden.

5.4.1. Clientadapter

Wie bereits erwähnt, deckt die Clientseite einen Teil der Fehlererkennung ab. Die Abbildung 5.9 beschreibt die gesamte Clientadapterseite und infolgedessen einen Teil der Fehlererkennung. Im Anschluss folgt eine Beschreibung des Klassendiagramms.

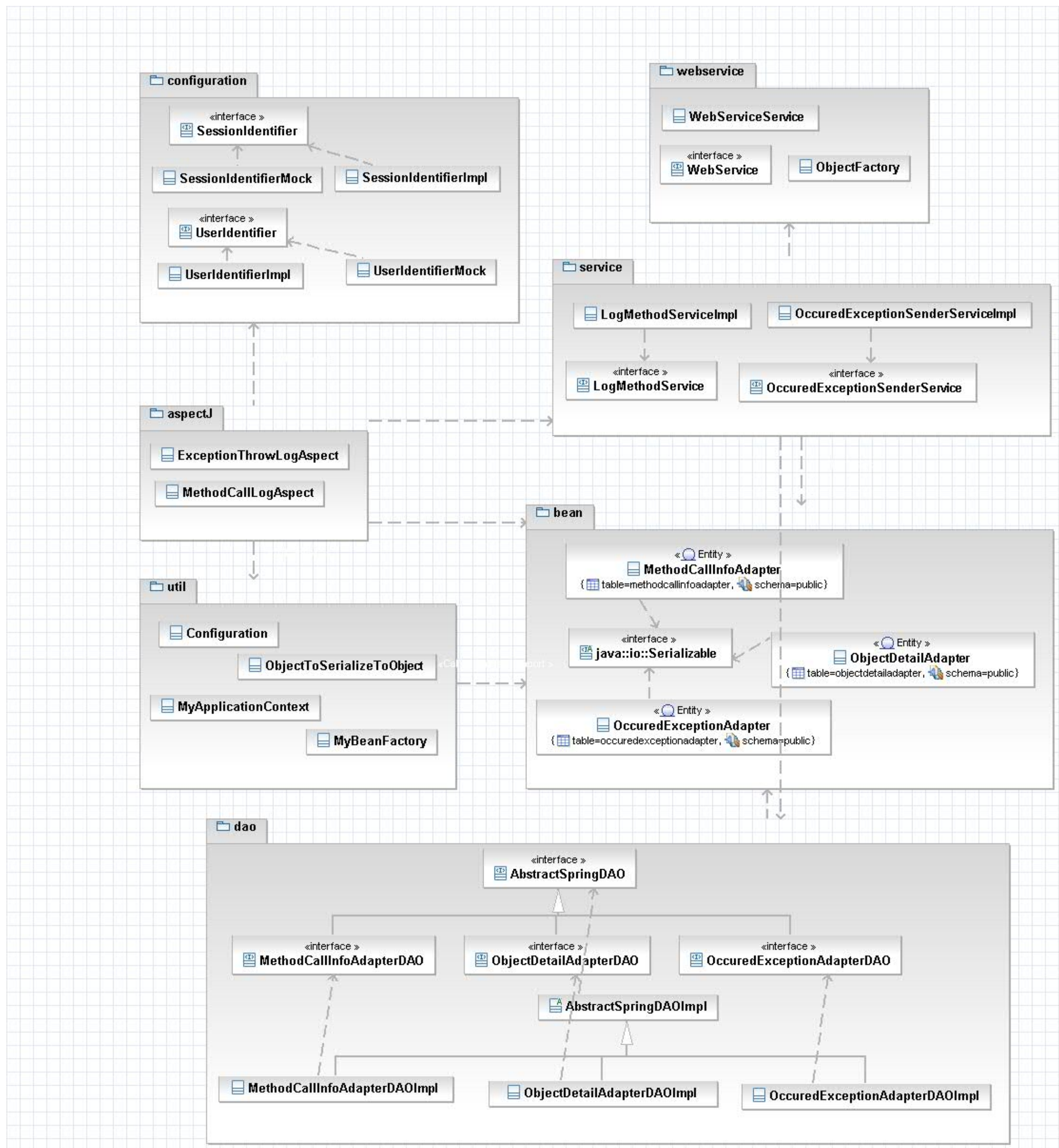

Abbildung 5.9.: Klassendiagramm: Client

aspectJ
Der Package aspectJ bezieht sich auf zwei Aspekte. Der Aspekt `MethodCallLogAspect` ist zuständig für die Identifikation aufgerufener Methoden und wird vor jedem Methodenaufruf aufgerufen. Für die Erkennung auftretender Exceptions ist der Aspekt `ExceptionThrowLogAspect` verantwortlich.

service
Die Klassen im Package service implementieren die Logik. Die Logik besteht u.a. aus der Protokollierung der gewonnen Daten durch die Aspekte.

bean
Das Package bean enthält drei Klassen. Jede Klasse repräsentiert eine Entität, die direkt auf den Datenbanktabellen abgebildet sind. Die Entität `MethodCallInfoAdapter` repräsentiert einen Methodenaufruf. Die Entität `OccuredExceptionAdapter` stellt eine aufgetretene Exception dar. Der `ObjectDetailAdapter` ist Bestandteil des `MethodCallInfoAdapter` und repräsentiert ein Java-Objekt. Mithilfe des `ObejctDetailAdapter`s können Parameter und Rückgabewerte als Objekte in der Datenbank abgespeichert und bei Bedarf wieder als Objekte abgerufen werden.

util
Das Package util enthält wiederverwendbare Funktionalitäten, die nicht direkt die fachlichen Anforderungen abbilden.

dao
Data access objects bieten CRUD-Operationen (Create, Read, Update, Delete) für Entitäten in der Datenbank. Aufgrund dessen wird eine Abstraktion des zugrunde liegenden Persistenz-Mechanismus erreicht. DAOs werden lediglich von den Services angesprochen.

configuration
Package configuration repräsentiert die Hot-Spots. Für die Integration des Frameworks ist die Konkretisierung der Klassen im Package configuration notwendig. Dies erfordert die Spezifizierung zweier Klassen; `SessionIdentifierImpl` und `UserIdentifierImpl`. Beide Klassen verfügen über das entsprechende Interface und jeweils eine Mock-Klasse. Die Mock-Klassen agieren als Default-Klassen, die durch die Implementierungsklassen ersetzt werden sollen.
Die Konkretisierung der Klasse `SessionIdentifier` ist für für die Identifizierung eines Prozesses unbedingt erforderlich, um Methodenaufrufe und auftretende Exceptions zuordnen zu können. Prozesse können auf unterschiedliche Weisen abhängig von der Zielanwendung identifiziert werden. Als Beispiele seien hier Threads oder Sessions zu nennen.
Die Konkretisierung der `UserIdentifier` Klasse ist ausschlaggebend für den User-Support. Besteht die Möglichkeit, einen Benutzer während eines Prozesses zu identifizieren,

ob über einen Login oder ähnliches, sollte dies in der Klasse `UserIdentifier` erfolgen. Infolgedessen ist das Framework in der Lage, bei auftretenden Exceptions ebenso den Auslöser zu identifizieren. Infolgedessen kann der Benutzer, der mit dem Fehlverhalten des Systems konfrontiert war, konkret angesprochen werden.

webservice
Jede auftretende Exception wird an die Serverseite kommuniziert. In diesem Package sind alle notwendigen Klassen für die Kommunikation zwischen dem Client und dem Server enthalten.

5.4.2. Server-Webanwendung

Die Serverseite fungiert als Webanwendung. Webanwendungen werden standardmäßig in drei Schichten unterteilt und als 3-Tier Architektur bezeichnet [(Dunkel und Holitschke, 2003, Seite 26)]. Die Autorin greift diese Unterteilung auf und strukturiert die Serverseite dementsprechend nach der 3-Tier Architektur. Nachfolgend werden die drei Schichten benannt und den fachlichen Komponenten zugewiesen.

1. Benutzeroberfläche; die Schicht, die die Benutzeroberfläche repräsentiert, der Begriff bezieht sich auf die gesamte Benutzeroberflächen-Komponente
2. Persistenz; die Schicht, die die Persistenz repräsentiert, bezieht sich auf die gesamte Persistenz-Komponente
3. Business; diese Schicht repräsentiert die gesamte Logik und umfasst folgende Komponenten:
   - Debugging
   - Fehlerverwaltung
   - Exceptionfilterung
   - Benachrichtigung
   - Kommunikation

Ein weit verbreitetes und oft zum Einsatz gelangendes Framework für Webanwendungen ist JSF [JSF]. JSF steht für JavaServer Faces (siehe 6.1.2.1) und unterstützt die Entwicklung von Webanwendungen in Bezug auf die Benutzeroberfläche. Auf Grund dessen wird JSF ebenfalls für die Serverseite des Frameworks (Webanwendung) verwendet.

Die Serverseite umfasst gegenüber der Clientseite eine höhere Anzahl an Klassen, deren Auflistung jedoch zu einer unübersichtlichen Darstellung führen würde. Aus diesem Grund

werden zwei Klassendiagramme dargestellt. Das erste Klassendiagramm umfasst die gesamte Serverseite, die auf der Package-Ebene abgebildet wird. Das zweite Diagramm zeigt auf der Klassenebene einen detaillierten Ausschnitt der Serverseite. Der Ausschnitt bezieht sich auf alle Schichten und bildet einen Anwendungsfall ab. Mit Hilfe dieser beiden Darstellungen wird zum einen die Serverseite komplett abgebildet und zum anderen ein detaillierter Einblick verschafft.

5.4.2.1. Gesamtübersicht

Das in Abbildung 5.10 vorgestellte Klassendiagramm ist generisch für Webanwendungen aufgebaut. Neue Funktionalitäten oder neue fachliche Anforderungen werden in den meisten Fällen aus einem Teil Präsentation, aus einem Teil Logik und einem Teil Datenhaltung bestehen. Die grobe Struktur generisch für Webanwendungen aufzubauen, ermöglicht es dem Entwickler, Erweiterungen einfacher und sauberer in der Webanwendung zu realisieren. Unabhängig von den fachlichen Anforderungen ist fest gelegt, wie der Ablauf in der Webanwendung regelhaft erfolgt; dieser kann in der Folge vom Entwickler für die Erweiterung übernommen werden.
Im Anschluss folgt in Abbildung 5.10 das generische Klassendiagramm, welches grob die gesamte Serverseite repräsentiert.

Nachfolgend wird kurz auf die Packages des generischen Klassendiagramms für die Webanwendung eingegangen.

JSP
Die JSP-Seiten repräsentieren die Benutzeroberfläche. Sie enthalten JSF-Komponenten und haben eine Verbindung zum Controller über die Konfigurationsschnittstelle (facesconfig.xml).

Validatoren
Die Validatoren sind für den Gültigkeitsbereich zuständig. Sie werden in die JSP-Seiten integriert und in den Beans bzw. Entitäten deklariert.

PageCode (RequestScope)
Der PageCode repräsentiert die JSF-Seiten-Information und behandelt die GUI- Kontroll- Logik. Demzufolge sind sie für das Handling innerhalb der Benutzerinteraktion zuständig. In der Regel interagiert eine JSP mit einer PageCode-Klasse. Bei stark voneinander abhängigen JSPs kann eine PageCode-Klasse die Kontroll-Logik für mehrere JSP übernehmen.

ViewBean
Eine ViewBean beinhaltet alle relevanten GUI-Informationen, die für die GUI-Anzeige not-

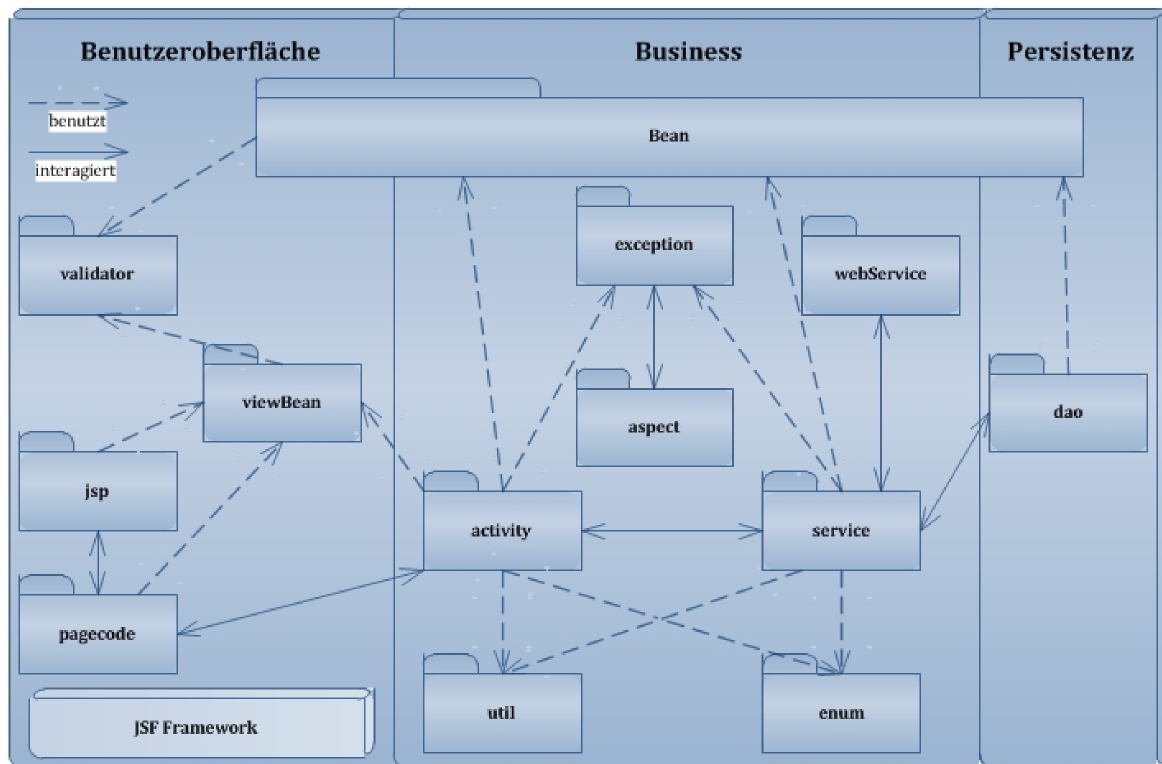

Abbildung 5.10.: Klassendiagramm: Server

wendig sind. Darüber hinaus können Informationen aus mehreren Einheiten für die GUIAnzeige zusammengeführt werden.

Activity (Session Scope)
Eine Aktivität ist ein Business-Anwendungsfall. Größere Business-Anwendungsfälle können ebenfalls in mehrere Aktivitäten aufgeteilt werden, wobei eine klare Abgrenzung der Teile des Business-Anwendungsfalls eingehalten werden sollte. Eine Aktivität ruft die Geschäftslogik von Services auf und bündelt den gesamten relevanten Business-Code, der für die Ausführung des Anwendungsfalls notwendig ist. Eine Aktivität weist Sitzungsinformationen auf und existiert nur einmal pro aktiver Sitzung.

Service
Services sind zustandslose Business-Objekte, sie implementieren die Business-Logik und können wiederverwendet werden. Zudem können Services gegenseitig Ihre Dienste nutzen.

Webservice
WebServices bilden die Kommunikations-Komponente ab und regeln den Informationsaustausch zwischen dem Adapterclient und der Serverseite (Webanwendung).

Bean (Enitity)
Entitäten sind Datenhalter-Objekte, die in allen Schichten verwendet und über alle Schichten weitergeleitet werden können. Entitäten können demzufolge ebenfalls angepasst direkt an die GUI weitergegeben werden. Ebenso ist die direkte Abbildung von Entitäten in Datenbanktabellen möglich, da es sich hierbei um POJOs (Plain Old Java Object) handelt.

Util
Utils enthalten Funktionen, die einen hohen Wiederverwendungsgrad haben, und die sich von Services abkapseln, indem sie keine fachlichen Anforderungen abbilden.

Enum
Enumarations Akronym Enums sind Objekte, bestehend aus einer Ansammlung von Konstanten.

Exception
Es existieren zwei Oberklassen: `BusinessException` und `SystemException`. Alle vom Entwickler implementierten Exceptions erben von einer dieser Oberklassen. Exceptions, die logische Fehler abhandeln, werden als `BusinessException`s klassifiziert. Schwerwiegendere Fehler gehören der Oberklasse `SystemException` an.

DAOs
Data access objects bieten CRUD-Operationen (Create, Read, Update, Delete) für Entitäten in der Datenbank. Mit ihrer Hilfe wird eine Abstraktion des zugrunde liegenden Persistenz-Mechanismus erreicht. DAOs werden lediglich von den Services angesprochen.

Aspect
Aspects separieren logische Aspekte, die an unterschiedlichen Stellen in der Anwendung eingebunden werden sollen. Als Beispiel sei ein Aspekt erwähnt, der alle `SystemException`s abfängt, um den Benutzer zu einer benutzerfreundlichen Fehlerseite zu leiten.

5.4.2.2. Detaillierter Auszug

Der Gesamtüberblick bietet dem Leser einen ersten Eindruck der konkreten Implementierungsstruktur der Webanwendung. Um dem Leser einen detaillierteren Einblick zu verschaffen, wird das Klassendiagramm konkretisiert und ein Teil des Frameworks auf der Klassenebene dargestellt, eine Art Auszug des generischen Klassendiagramms (Abbildung 5.11), bezogen auf einen Teil eines konkreten Anwendungsfalls.
Das Beispiel bezieht sich auf einen Teil der Fehlerverwaltung. Die Fehlerverwaltung umfasst die Erstellung und die Übersicht der verantwortlichen Instanzen. Die verantwortlichen

Instanzen sind die Parteien, die kontaktiert werden, falls eine Exception auftritt. Eine verantwortliche Instanz kann aus einer Person „ConnectionProfile“ oder einer Gruppe von Personen „ConnectionProfileGroup“ bestehen. Zur einer Person sind Informationen wie Vorname, Nachname, Personenbeschreibung (z.B. Beruf oder Position), Email Adresse, Liste der Telefonnummern notwendig. Die Personengruppe enthält den Gruppennamen, die Gruppenbeschreibung sowie eine Liste aller Gruppenangehörigen. Das nachfolgende Klassendiagramm (Abbildung 5.11) weist alle notwendigen Klassen, sowie Assoziationen auf, die nötig sind, um den Anwendungsfall zu realisieren.

JSP

Für die Erstellung eines ConnectionProfile und einer ConnectionProfilGroup wird jeweils eine JSP-Seite sowie jeweils eine Seite für die Ansicht der erstellten verantwortlichen Instanzen benötigt. Des Weiteren wird in jeder JSP-Seite eine Menuleiste sowie eine Kopf- und Fußleiste integriert. Daraus ergeben sich insgesamt sieben JSP-Seiten.

Validatoren

Er werden drei Validatoren benötigt: jeweils ein Validator zur Beschreibung des Formats von Email Adressen und Telefonnummern und ein weiterer Validator, der den Gültigkeitsbereich von Längen für Eingabeparameter definiert.

PageCode

Für die Durchführung der GUI-Kontroll-Logik wird jeder JSP eine PageCode-Klasse mit ähnlichen Namen zugewiesen (`create_connection_profile_page.jsp => CreateConnectionProfilePagecode`). Jede Pagecode-Klasse erbt von der abstrakten Klasse `AbstractPagecode`, die Methoden für alle Pagecode-Klassen zur Verfügung stellt. Als Beispiel sei die Methode `callErrorMessage` erwähnt, die während der Pagecode-Ausführung Informationen über alle Fehler sammelt und diese dem Benutzer gebündelt darstellt.

ViewBean

Wie es zu jeder JSP eine Pagecode-Klasse gibt, existiert für jedes JSP-Pagecode Paar eine ViewBean. Jede ViewBean enthält die erforderlichen Informationen für die GUI.

Activity

Jede Aktivität besteht aus einer Klasse (Endung Impl) und dem dazugehörigen Interface. Alle Aktivitätenklassen erben von der abstrakten Klasse `AbstractActivityImpl` und alle Aktivitäteninterfaces von der Interface `AbstractActivity`. Für diesen Anwendungsfall existiert eine Aktivitätenklasse mit dem Namen `ManageConnectionActivityImpl` und das dazugehörige Interface `ManageConnectionActivity` . Diese Aktivität ist dafür zuständig, alle notwendigen Dienste aus den Services zu bündeln, um die Erstellung und die Ansicht von verantwortlichen Instanzen (ConnectionProfile und ConnectionProfileGroup) zu ermöglichen.

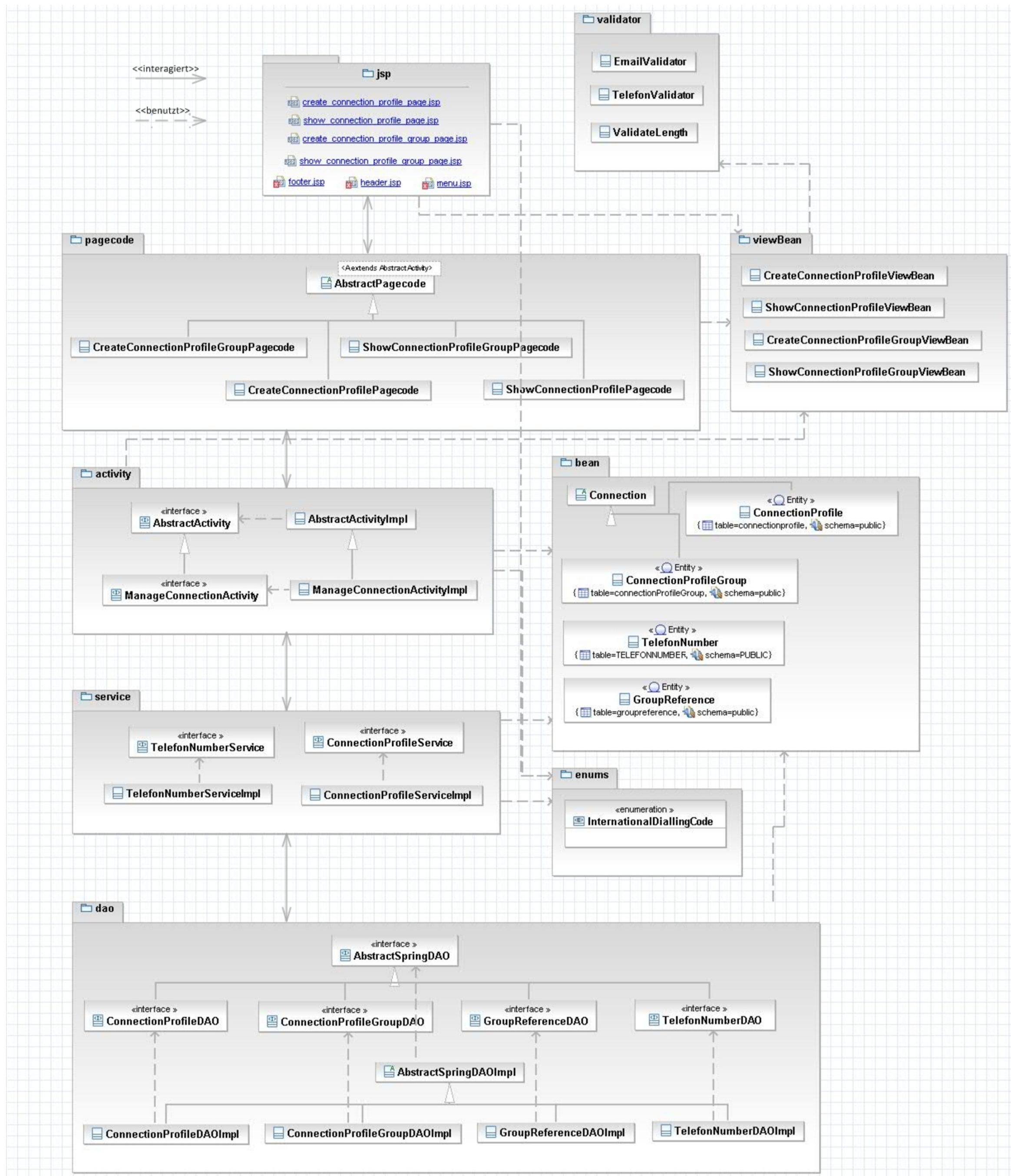

Abbildung 5.11.: Auszug aus dem Klassendiagramm: Server

Service
Die Services bilden die gesamte Logik ab, die u.a. für die Realisierung des Anwendungsfalls notwendig ist. Es werden zu diesem Zweck zwei Service-Klassen benötigt. Eine Service-Klasse für die Abbildung der Dienste bezüglich der verantwortlichen Instanz und eine weitere für die Telefonnummern.

Bean (Enitity)
Es besteht ein Bedarf an insgesamt vier Entitäten. Eine Entität für die verantwortliche Instanz Person, eine weitere für die verantwortliche Instanz Gruppe. Für die Gruppenzugehörigkeit wird ebenfalls eine Entität benötigt und für Telefonnummern die vierte und letzte.

Enum
Die Telefonnummern bestehen aus Landesvorwahlen, die als Konstanten in der Enum `InternationalDiallingCode` gebündelt werden.

DAOs
Zu jeder Entität gehört ein DAO, der die Operationen auf der Datenbank zur Verfügung stellt. Zu jeder DAO-Klasse existiert das entsprechende Interface. Ausnahmslos erben alle DAO-Klassen von der Oberklasse `AbstractSpringDAOImpl`, die wiederum von der Oberklassen `HibernateDaoSupport` erbt. Die Interfaces erben von dem Interface `AbstractSpringDAO`. Die Datenbankoperationen, die von den DAOs zur Verfügung gestellt werden, sollten nur von den Service-Klassen aufgerufen werden dürfen.

Eine zu hohe Anzahl an Anwendungsfällen könnte dazu führen, dass diese Klassenstruktur unübersichtlich wird. Aus diesem Grund sollte bei einer hohen Anzahl an Klassen, Packages untergliedert werden. Die Package activity kann aufgesplittet werden, siehe nachfolgend aufgeführtes Beispiel.

Packagestruktur einer überschaubaren Anzahl an Aktivitäten
`com.companyname.project.activity`

Packagestruktur einer zu hohen Anzahl an Aktivitäten
`com.companyname.project.activity` => abstrakte Activity-Klassen
`.....activity.errorManagement` => Activity-Klassen der Fehlerverwaltung
`.....activity.debugging` => Activity-Klassen des Debuggings
`...`

6. Realisierung

In diesem Kapitel wird als erstes auf die Technologien, die zur Realisierung des Frameworks angewandt wurden, eingegangen. Das zweite Unterkapitel beschäftigt sich mit den wesentlichen Problematiken, die sich während der Realisierung zeigten, und die zur Abweichung des Konzepts und/oder der Architektur führten. Im Anschluss werden die Tests des Prototypen vorgestellt. Abschließend werden einige Screenshots der Webbenutzeroberfläche einschließlich einer Erläuterung präsentiert.

6.1. Angewandte Technologien

Heutzutage existiert eine Vielzahl an Technologien, die zur erfolgreichen Entwicklung eines Systems beitragen. Die Autorin hat für jede Schicht (Präsentation, Logik, Datenbank, Kommunikation) auf Basis Ihrer Erfahrungen und Recherchen eine Reihe unterstützender Hilfen (Frameworks, Klassenbibliotheken etc.) ausgewählt. Die angewandten Technologien beschränken sich auf OpenSource Produkte. Nachfolgend werden diese Technologien kategorisiert nach den Schichten aufgelistet, zuzüglich der Basis Technologien. Zu jeder vorgestellten Technologie wird kurz auf die entscheidenden Auswahlkriterien eingegangen, die Kriterien-erfüllenden Alternativen aufgelistet, die Entscheidung begründet und der Einsatz der Technologie bewertet. Es sei erwähnt, dass lediglich auf die wichtigsten angewandten Technologien eingegangen wird, jedoch kam eine Vielzahl weiterer Technologien zum Einsatz.

6.1.1. Basis Technologien

An dieser Stelle wird auf die wesentlichen angewandten Basis-Technologien eingegangen.

6.1.1.1. Entwicklungssprache

Kriterien: Die Entwicklungssprache sollte zum einen ein objektorientiertes Konzept verfolgen, um die Entwicklung einer flexiblen und erweiterungsfähigen Anwendung zu unterstützen. Ein weiteres Kriterium ist die Plattformunabhängigkeit der Programmiersprache. Ein Vorteil ist die weite Verbreitung der eingesetzten Sprache, die infolgedessen in den meisten Fällen aussagekräftige Dokumentationen und wiederverwendbare Komponenten besitzt.
Alternativen: C# [C-Sharp], C++ [C++], Java [Java]
Entscheidung: Aufgrund der Erfahrungen, über die die Autorin im Java-Umfeld verfügt, fiel die Wahl auf Java.
Bewertung: ie hohe Beliebtheit von Java führt dazu, dass derzeit viele Erweiterungen, auf Java zugeschnittene Entwicklungsumgebungen, unzählige Beispiele und weitere sehr hilfreiche Unterstützungen existieren. Die zahlreichen Unterstützungen führten zu einer komfortablen Implementierung und einem verringertem Aufwand bei der Realisierung.

6.1.1.2. Entwicklungsumgebung

Kriterien: Die Entwicklungsumgebung soll den gesamten Software-Entwicklungszyklus unterstützen und insbesondere den Schwerpunkt auf die Entwicklung von JEE-Anwendungen, Web-Applikationen und Webservices legen.
Alternativen: Eclipse [Eclipse], NetBeans [NetBeans]
Entscheidung: Beide Entwicklungsumgebungen bieten eine hervorragende Unterstützung bei der Entwicklung von JEE-Anwendungen, Web-Applikationen und Webservices. Die Autorin entschied sich jedoch für Eclipse. Eclipse sowie NetBeans unterstützen das Plugin-Konzept, allerdings ist das Plugin-Konzept bei Eclipse ausgereifter, und es steht eine größere Auswahl an Plugins zur Verfügung. Des Weiteren verfügt die Autorin über mehr Erfahrungen mit Eclipse.
Bewertung: Der Schwerpunkt, der auf der Erstellung von JEE-Anwendungen, Webservices und Web-Applikationen liegt, ist bemerkbar. Es existieren für diesen Bereich zahlreiche Plugins und Features, die über den Software Update Manager leicht nachinstalliert werden können. Zudem bewährt sich Eclipse in punkto Schnelligkeit.

6.1.1.3. Webserver

Kriterien: Der Webserver soll zum einen eine HTTP Web-Server-Umgebung zur Ausführung von Java-Code zur Verfügung stellen, zum anderen muss die Java-Servlet und JavaServer Pages (JSP) Spezifikationen von Sun Microsystems (fusioniert mit Oracle) implementiert sein.
Alternativen: Apache Tomcat (ebenfall Jakarta Tomcat genannt) [Tomcat]. Es existieren

ebenso andere Alternativen. Einige herkömmliche Applikations Server können um die Java-Servlet- und JSP-Spezifikationen erweitert werden und ebenso als Webserver bzw. Servlet-Container agieren. JBoss [JBoss] stellt ein Beispiel dar,in dem die Implementierungserweiterung basierend auf Tomcat zur Verfügung gestellt wird.
Entscheidung: Apache Tomcat ist ein vollständiger Servlet-Container, was ihn dazu prädestiniert, als Webserver zum Einsatz zu gelangen.
Bewertung: Der Umgang mit dem Servlet-Container gestaltet sich aufgrund der hohen Eclipse-Unterstützung sehr komfortabel. Eclipse bietet eine ausgeweitete Schnittstelle zu Apache Tomcat, die das Starten, Stoppen, Cleanen, Deployen usw. unterstützt. Des Weiteren agierte Apache Tomcat sehr stabil, keine Server-Abstürze wurden registriert.

6.1.1.4. Versionsverwaltungssystem

Kriterien: Das Versionsverwaltungssystem soll die Protokollierung von Änderungen an Dokumenten, insbesondere Javaklassen (Sourcecode), durchführen. Zudem ist die komplette Wiederherstellung beliebiger Versionen ein weiteres Kriterium. Zusätzlich soll die Benutzerkennung und der Zeitstempel erfasst werden, um den Verlauf von Modifikationen besser nachvollziehen zu können.
Alternativen: CVS (Concurrent Version System) [CVS], SVN (Apache Subversion) [SVN]
Entscheidung: SVN wird als Nachfolger von CVS gesehen und bietet viele nützliche Erweiterungen gegenüber dem Vorgänger. In CVS werden verschobene Dateien im ursprünglichen Verzeichnis als gelöscht und im aktuellen Verzeichnis als neu markiert dargestellt. Ein Verlust der Historie ist die Folge. SVN hingegen überträgt die Historie. Aufgrund der zusätzlichen Erweiterungen wird die Verwendung von SVN bevorzugt.
Bewertung: Die Entwicklungsumgebung Eclipse unterstützt SVN sehr stark. Aus diesem Grund gestaltete sich der Einsatz von SVN sehr komfortabel.

6.1.2. Präsentation

An dieser Stelle wird auf die wesentlichen angewandten Technologien für die Realisierung der Weboberfläche eingegangen.

6.1.2.1. Web-Framework

Kriterien: Das Web-Framework soll eine Unterstützung bei der Entwicklung von Webbenutzeroberflächen darstellen. Zu den geforderten Hilfeleistungen zählen u.a. triviale Aufgaben,

die sehr aufwändig sind. Zu nennen sei das Binden von Request- und Responseparametern, das Validieren von Eingaben u.v.m. Des Weiteren soll auf einfachem Wege die Erstellung von einheitlichen Erscheinungsbildern und wiederverwendbaren View- Komponenten für Webanwendungen möglich sein, um die Entwicklung eines konsistenten „Look and Feel“ zu vereinfachen, die sich auf die gesamte Webanwendung ausdehnen soll.
Alternativen: JSF [JSF] steht für JavaServer Faces und ist eine Webframework Standard Spezifikation (keine konkrete Implementierung) zur Unterstützung der Entwicklung von Webbenutzeroberflächen. Apache Software Foundation stellt mehrere JSF Implementierungen unter dem Namen MyFaces bereit. Die Implementierungen beinhalten Unterstützungen in vielen Bereichen der Webbenutzeroberflächen-Entwicklung. Das Binden von Request- und Responseparametern, das Validieren von Eingaben sind u.a. ebenfalls abgedeckt. Tiles [Tiles] ist die zweite Alternative und ein Templating Framework (und Bestandteil von Struts). Mit Hilfe von Tiles besteht die Möglichkeit, Seitenfragmente zu definieren, welche zur Laufzeit gebündelt eine komplette Seite darstellen. Infolgedessen kann die Vervielfältigung von wiederverwendbaren Fragmenten verhindert und wiederverwendbare Vorlagen genutzt werden.
Entscheidung: JSF ist im Gegensatz zu Tiles ein umfangreiches Framework. Aufgrund der unterschiedlichen Schwerpunkte überschneiden sich jedoch ihre bereitgestellten Hilfeleistungen kaum. Beide Frameworks ergänzen sich gut und decken mehrere Bereiche ab. Infolgedessen werden beide Frameworks verwendet.
Bewertung JSF: Für den erfolgreichen Einsatz von JSF ist eine Einarbeitungsphase notwendig. Infolgedessen eignet sich das Verwenden von JSF in kleinen Projekten nur, wenn bereits Erfahrungen in diesem Bereich vorhanden sind. Ansonsten verkürzt JSF den Entwicklungsprozess sehr. Jedoch gestaltet sich das Debugging innerhalb der sechs Phasen von JSF nicht ohne Probleme. Hilfen, wie z.B. Facetrace und Bugzilla unterstützen diesen Debugging Vorgang.
Bewertung Tiles: Mit geringem Aufwand ist Tiles in eine Webanwendung integrierbar. Die Aufteilung der Fenster gestaltet sich problemlos und wird für alle Seiten übernommen.

6.1.3. Logik

Für die Umsetzung der Logikfunktionalitäten wurden verschiedene Technologien verwendet. Auf die wichtigsten wird nachfolgend eingegangen.

6.1.3.1. Applikations-Framework

Kriterien: Es existieren zahlreiche neue Konzepte, Ansätze, Innovationen usw., die eingesetzt werden, um eine bessere Anwendung zu realisieren. Aus diesem Grund soll ein Applikations-Framework verwendet werden, das eine umfangreiche Unterstützung vieler dieser Bereiche bietet. Das erste Kriterium bezieht sich auf die Java-Kompatibilität, das nächste

auf die Unterstützung von Dependency Injection (DI). DI lagert die Erzeugung, Instanziierung und die Verwaltung von Objekte an einer Stelle aus. Infolgedessen werden starke Abhängigkeiten minimiert. Es sind weitere Kriterien an das Applikations-Framework vorhanden, die an dieser Stelle nicht weiter erwähnt werden. Als abstraktes Kriterium sei genannt, ein rundum JEE-Applikations-Framework, welches unterschiedliche Bereiche abdeckt.
Alternativen: Spring [Spring]
Entscheidung: Als alleinige Alternative wurde Spring aufgeführt, es ist dementsprechend der Favorit.
Bewertung: Spring ist ein sehr umfangreiches Applikations-Framework; unter Verwendung von Spring können sehr viele unterschiedlichen Konzepte, Ansätze, Innovationen in die zu entwickelnde JEE-Anwendung problemlos einbezogen werden. Die Autorin empfand zudem die Spring-Dokumentationen als sehr ausführlich und hilfreich.

6.1.3.2. Reflection

Kriterien: In einigen Bereichen des Frameworks werden Reflections benötigt. Aus diesem Grund wird nach einer Reflection Implementierung gesucht. Es soll sich um eine schmale Implementierung handeln, die problemlos in Java integriert werden kann, und die einfach handhabbar ist. Der Funktionsumfang sollte die Definition, Modifikation und das Laden von Java-Objekten zur Laufzeit umfassen.
Alternativen: Javassist (Java programming assistant) [Javassist]
Entscheidung: Javassist wurde als einzige Alternative gefunden, die alle Kriterien erfüllte.
Bewertung: Das Integrieren bzw. Einbinden von Javassist gestaltet sich, da es sich um eine Klassenbibliothek handelt, sehr einfach. Die Verwendung ist ebenso unkompliziert.

6.1.3.3. AOP

Kriterien: Die AOP-Implementierung muss Java kompatibel sein. Zudem muss das Einbinden bzw. Einweben von Aspekten nach einer auftretenden Exception unterstützt werden. Des Weiteren ist der Bekanntheits- und Verwendungsgrad ausschlaggebend; zu bekannten und häufig verwendeten Technologien existieren in den meisten Fällen ausreichend viele Dokumentationen, Beispiele etc.
Alternativen: AspectJ [AspectJ], AspectWerkz [AspectWerkz], DynamicAspects [DynamicAspects]
Entscheidung: AspectJ erweist sich als der Favorit unter den Alternativen. Im Konkurrenzumfeld wird AspectJ als AOP- Implementierung im Java Umfeld am häufigsten verwendet. Aufgrund dessen stehen zahlreiche Dokumentationen, Beispiele usw. zur Verfügung. Des Weiteren unterstützt Eclipse (die eingesetzte Entwicklungsumgebung) die Verwendung von AspectJ. Ebenso bedeutsam sind die Erfahrungen, die die Autorin mit AspectJ gemacht hat.

Bewertung: Für den Einsatz von AspectJ ist eine Einarbeitungszeit erforderlich. Der Grund ist nicht AspectJ, sondern die Einarbeitung in das Konzept des Aspektorientierten Programmierens. Die zahlreich vorhandenen Dokumentationen, Tutorien und Beispiele waren bei der Einarbeitung sehr hilfreich.

6.1.4. Datenbank

Es folgt eine Darstellung der angewandten Technologien im Bereich der persistenten Datenhaltung.

6.1.4.1. ORM-Framework

Kriterien: Das Framework soll Object-Relational Mapping (ORM) für Java unterstützen, d.h., das bidirektionale Zuordnen bzw. Abbilden von Java-Objekten zu Datenbank-Relationen. Zudem soll das ORM-Framework kompatibel mit unterschiedlichen Datenbanken sein, um einen unkomplizierten Datenbankaustausch zu ermöglichen. Weitere nützliche Dienste im Bereich der persistenten Datenhaltung sind ebenfalls erwünscht.
Alternativen: Hibernate [Hibernate], Ujorm [Ujorm]
Entscheidung: Hibernate ist ein bewährtes Framework mit einem hohen Beliebtheits- und Reifegrad. Ujorm dagegen ist ein Neuling. Aus diesem Grund wird auf Hibernate zurückgegriffen.
Bewertung: Wie im Zusammenhang mit der Entscheidung bereits ausgeführt, sticht Hibernate durch einen sehr hohen Beliebtheits- und Reifegrad hervor; infolgedessen sind zahlreiche Dokumentation, Tutorien, Beispiele, Plugins für Eclipse usw. vorhanden, die den Umgang mit Hibernate unterstützen.

6.1.4.2. Datenbanksystem

Kriterien: Hibernate entkoppelt die Anwendung von der Datenbank und ist zudem kompatibel zu vielen Datenbanken; infolgedessen ist der Datenbankaustausch unkompliziert. Aus diesem Grund werden für die Datenbankauswahl des Prototypen lediglich die wesentlichen Kriterien definiert. Bei der Datenbank soll es sich um ein relationales Datenbanksystem mit vollständiger ACID (atomicity, consistency, isolation, durability)-Kompatibilität und der Unterstützung der SQL-Datentypen handeln.
Alternativen: Es existiert eine Vielzahl an Datenbanksystemen, die die Kriterien erfüllen. Zwei sehr bekannte sind MySQL [MySQL] und PostgreSQL [PostgreSQL].
Entscheidung: Im Grunde sind beide Alternativen sehr gute Datenbanksysteme. Da die jetzige Wahl nur den Prototypen betrifft und jeder Zeit ein Austausch möglich ist, fiel die Wahl

auf PostgreSQL aufgrund der verbesserten Schnelligkeit in den letzten Releases.
Bewertung: Im Allgemeinen lief PostgreSQL in der Entwicklungsphase bis zur Versionsumstellung stabil. Bei dem Versuch, von Version 8.3 zu Version 9 zu wechseln, traten gravierende Probleme im Bereich der serialisierten Objekte ein. In PostgreSQL eingetragene serialisierte Objekte erhielten beim Wiederaufrufen andere Werte. Infolgedessen konnte das Objekt nicht deserialisiert werden.

6.1.5. Kommunikation

Es wurde für die Kommunikation die Webservices Technologie angewandt, auf die nachfolgend eingegangen wird.

6.1.5.1. Webservices

Kriterien: Für die Kommunikation zwischen Client und Server werden Webservices verwendet. Aus diesem Grund wird für diese Problemstellung ein Framework, Tool oder eine andere Alternative gesucht. Diese Alternative soll im Bereich Java eine einfache Möglichkeit zur Erstellung und zum Deployen von Webservice bereit stellen. Es soll zudem das Code-First Prinzip unterstützt werden, d.h. aus Javaklassen und -methoden sollen die entsprechende WSDL und die Deskriptoren erzeugt werden. Zusätzlich sollen Webservice mit Hilfe von Annotations entwickelt und deployed werden können.
Alternativen: Apache Axis2 [Axis2], JAX-WS (Java API for XML - Web Services) [JAX-WS].
Entscheidung: JAX-WS ist bereits seit Java 6 im Java SE-Umfang enthalten und wurde infolgedessen favorisiert.
Bewertung: Die Erstellung von Webservices mit Hilfe von JAX-WS unter Verwendung von Annotations stellte sich als sehr unkompliziert heraus.

6.1.6. Zusammenfassung der angewandten Technologien

Technologienart	Name	Herausgeber	Version
Basis Technologien			
Entwicklungssprache	Java	Sun Microsystems (Oracle)	1.6.0_23
Entwicklungsumgebung	Eclipse	Eclipse Foundation	3.4.2 Ganymede
Servlet-Container	Apache Tomcat	Apache Software Foundation	6.0
Versionsverwaltungssystem	SVN	CollabNet	1.3.1

Technologienart	**Name**	**Herausgeber**	**Version**
		Präsentation	
Web-Framework	JSF	Apache Software Foundation	1.1.6 MyFaces Tomahawk
Web-Framework	Tiles	Apache Software Foundation	1.2.9 (Struts)
		Logik	
Applikations-Framework	Spring	SpringSource	2.5.6
Java-API für Reflection	Javassist	JBoss	3.8.0.GA
AOP	AspectJ	Eclipse Foundation	1.5.4
		Datenbank	
ORM-Framework	Hibernate	JBoss	3.4.0.GA
Datenbanksystem	PostgreSQL	PostgreSQL	8.3
		Kommunikation	
Java-API für Webservices	JAX-WS	Sun Microsystems (Oracle)	2.0

Tabelle 6.1.: Zusammenfassung der angewandten Technologien

6.2. Abweichungen vom Konzept

Konzepte und Architekturen verändern sich während des gesamten Entwicklungsprozesses. Diese Änderungen werden hervorgerufen durch bewegliche Ziele, geänderte Einflussfaktoren, suboptimale Entwurfsentscheidungen, ausgelöst durch fehlende Detailinformationen etc. [(Starke, 2002, Seite 26,30)]. Das in dieser Masterarbeit zu entwickelnde System unterliegt ebenfalls Abweichungen. Dieses Unterkapitel befasst sich mit den Problematiken, die während der Realisierungs- und Testphase auftraten und zu Abweichungen bzw. Änderungen des Konzepts und/oder der Architektur führten.

Die Testphase lief parallel zur Realisierungsphase und darüber hinaus. Während dieser vermischten Realisierungs- und Testphase sind einige Problematiken zum Vorschein gekommen, auf die nachfolgend näher eingegangen wird. Die aufgetretenen Problematiken werden nicht danach unterteilt, in welcher Phase sie auftraten, da durch die Vermischung der Realisierungs- und Testphase keine klare Abgrenzung möglich ist. Zudem soll erwähnt werden, dass lediglich auf die wesentlichen und interessanten Problematiken und Abweichungen eingegangen wird, um den Rahmen dieser Masterarbeit nicht zu sprengen. Zu jedem Problem bzw. jeder Abweichung wird zunächst die Problematik erläutert, und im Anschluss werden mögliche Lösungsansätze vorgestellt.

6.2.1. Parsen der Exceptions

Problematik: Im Wesentlichen gestaltet sich die Realisierung des Exception-Parsens nach dem Konzept 5.1.3. Es ist jedoch im Konzept ein wichtiger Aspekt außer Acht gelassen worden. Es reicht nicht aus, alle Exceptions aus der Zielanwendung, den eingebundenen Klassenbibliotheken und den integrierten Frameworks zu filtern. Die JRE umfasst ebenso Exceptions, die in der Zielanwendung auftreten können. Es sei ein einfaches Beispiel zu nennen: die Verwendung einer `ArrayList` kann zu einer `NoSuchElementException` führen. Beide `ArrayList` und `NoSuchElementException` sind Klassen des Package `java.util`, die sich in rt.jar befinden. Die rt.jar ist eingeschlossen in den JRE System Klassenbibliotheken und nicht im Exceptionparsvorgang berücksichtigt worden. Die Verwendung von JDK oder JEE vergrößert das Ausmaß des Problems. JDK und JEE beinhalten weitere potenzielle, auftretende Exceptions.

Lösungsansätze: Es existieren mehrere Möglichkeiten diese Problematik zu beheben. Eine Alternative besteht darin, den Parsvorgang um die Java-Laufzeitumgebung auszudehnen unter Erweiterung der Angaben bezüglich der Speicherorte, die geparst werden. Eine weitere Alternative ist die Erkundung, welche Exceptions die verwendeten Java-Laufzeitumgebung umfasst, und diese in der Datenbank nachträglich einzutragen.

6.2.2. Abspeichern der Methodenaufrufe

Problematik: Wie bereits erwähnt, ist die Protokollierung der Methoden, die vor und nach einer aufgetretenen Exception aufgerufen wurden, für das historische Debugging sehr wichtig. Es werden zu der vollständigen Methodensignatur (Klassenname, Methodenname, Typen der Rückgabwerte und Parameter) ebenso die Rückgabewerte und Parameter als vollständige Objekte abgespeichert. Zu diesem Zweck kann der Anwender beim historischen Debugging den Exceptionverlauf besser verfolgen, indem ein tieferer Einblick in die Objekte ermöglicht wird. Im Konzept 5.1.4 wurde entschieden, die Abspeicherung dieser Objekte (Rückgabewerte und Parameter) in serialisierter Form in der Datenbank durchzuführen und beim Abrufen aus der Datenbank diese zu deserialisieren. Auf den ersten Blick schien alles unproblematisch, die Objekte wurden serialisiert, im Anschluss in der Datenbank in Form eines Byte Arrays abgespeichert und bei Bedarf aus der Datenbank aufgerufen und deserialisiert. Das kontinuierliche Testen parallel zur Realisierung wies keine Schwierigkeiten auf. Erst die komplexeren Testfälle brachten das Problem zum Vorschein. Objekte können in Java nur dann de-/serialisiert werden, wenn das Objekt das leere Interface `java.io.Serializable` implementiert. Es kann nicht garantiert werden, dass in der Zielanwendung und in den eingebundenen Klassenbibliotheken und integrierten Frameworks alle Rückgabe- und Parameterobjekte das Interface `java.io.Serializable`

implementieren und infolgedessen serialisierbar sind.

Lösungsansätze: Ein möglicher Lösungsansatz ist die Verwendung von Reflections. Wie bereits erwähnt, können Klassen zur Laufzeit mit Hilfe von Reflections modifiziert werden. Jede Klasse, die de-/serialisiert werden soll und nicht das Interface `java.io.Serializable` implementiert, kann zur Laufzeit um `implements java.io.Serializable` erweitert werden. Bei `java.io.Serializable` handelt es sich um ein leeres Interface, und infolgedessen müssen keine weiteren Methoden der modifizierten Klasse hinzugefügt werden.

6.2.3. Laden der Objekte

Problematik: In Bezug auf die zuvor behandelte Problematik tritt eine weitere Schwierigkeit auf. Nachdem die Objekte der Datenbank entnommen und deserialisiert wurden, müssen sie mit Hilfe von Reflections geladen werden. Die Objekte werden geladen, damit der Anwender über die Weboberfläche die Objekte (Rückgabewerte und Parameter von Methodenaufrufen) bis in die tiefste Ebene betrachten kann. Bis zu dieser Stelle ist keine ersichtliche Schwierigkeit erkennbar. In Anbetracht des Standpunktes der Reflectionausführung, kommt jedoch das Problem zum Vorschein. Die Objekte werden unter Verwendung von Reflections auf der Seite der Webanwendung geladen. Die Objekte sind jedoch Erzeugnisse der Java-Zielanwendung und deren Umgebung. Für das Laden der Objekte muss die entsprechende Klasse bekannt sein, die logischerweise der Webanwendung nicht bekannt ist. Infolgedessen scheitert das Laden der Objekt für unbekannte Klassen (ausgenommen Standardklassen, die auf beiden Seiten bekannt sind).

Lösungsansätze: Objekte, die ursprünglich aus der Zielanwendung stammen, sollen in der Webanwendung über Reflections geladen werden. Dies ist möglich, sobald der Webanwendung die Klassen der Zielanwendung bekannt sind. Dieses kann erreicht werden durch Hinzufügen der Zielanwendung inklusive Klassenbibliotheken und Frameworks in die Webanwendung. Nach diesem Vorgang sind die Klassen der Zielanwendung ebenso in der Webanwendung Bekannt, und das Laden der Objekte kann durchgeführt werden.

6.2.4. Darstellung der Methodenaufrufe

Problematik: In der Webanwendung werden Methoden, die vor und nach einer aufgetretenen Exception aufgerufen wurden, in einer Baumstruktur angezeigt, siehe Abbildung 6.5. Eine Methode, die innerhalb ihrer Methode vier andere Methoden aufruft, hat in der Baumstruktur vier Äste und diese Methoden wiederum so viele Äste, wie sie Methoden aufgerufen

hat. Der Grundgedanke hierbei war, dem Anwender eine bessere Sicht der Abhängigkeiten zwischen den Methoden zu ermöglichen. Auf den ersten Blick scheint diese Anordnung sehr geordnet und übersichtlich. Für kleine Methodenaufrufbäume trifft diese Aussage auch zu. Große Methodenaufrufbäume, die in der Regel in der Realität vorkommen werden, reduzieren dagegen die Übersichtlichkeit gravierend. Jede tiefere Methodenebene ist in einer Baumstruktur um eine Einheit nach rechts versetzt. Durch die große Anzahl an Methoden entsteht eine Baumstruktur, die sich nicht nur nach unten, sondern extrem nach rechts ausweitet. Das zusätzliche Öffnen und Schließen der einzelnen Äste in dem „versetzenden“ Baum kann zur Beeinträchtigung bei der Betrachtung der Methodenabläufe führen. Die Anwender reagieren eventuell mit Verwirrung auf die Informationen.

Lösungsansätze: Ein einfacher Lösungsansatz ist die Auflistung aller aufgerufenen Methoden untereinander (nicht versetzt wie in der Baumstruktur). Das Öffnen und Schließen einzelner Methodenabläufe soll darüber hinaus unterstützt werden, um weiterhin unnötige Informationen ausblenden zu können.

Es waren und sind noch weitere Usability-Verbesserungen notwendig, auf die an dieser Stelle nicht weiter eingegangen wird. Es soll jedoch erwähnt werden, dass die Usability ausschlaggebend für die Akzeptanz oder das Verwerfen eines Systems ist.

6.2.5. Inkonsistenz der Daten

Problematik: Die Webanwendung (Framework-Server) ist ein interaktives System. Zum einen arbeitet es mit Anwendern, zum anderen mit Systemen (Zielanwendungen). Beide interaktive Partnergruppen wirken direkt auf die abgespeicherten Daten ein. In diesem Zusammenhang können Inkonsistenzen innerhalb der Daten auftreten. Es sei ein Beispiel zu nennen, das beim Testen aufgefallen ist: Der Anwender betrachtet alle aufgetretenen Exceptions, die aus der Datenbank entnommen wurden und in der Session gehalten werden. Zwischenzeitlich greift der Anwender auf die Details einer Exception zu, und parallel dazu tritt eine weitere Exception in der Zielanwendung auf. Zurück navigiert in die Übersicht über alle aufgetretenen Exceptions werden diese aus der Session entnommen. Infolgedessen wird die kurz zuvor ausgetretene Exception nicht ausgelistet. Dieses triviale Problem ist mit Hilfe einer Sessionsäuberung einfach zu beheben; diese wird bereits an vielen anderen Stellen eingesetzt. Diese Problematik muss jedoch in verfeinerter Form betrachtet werden. Nicht an jeder Stelle ist eine Sessionsäuberung erwünscht. Viele Seiten sind voneinander abhängig und die Weitergabe von Daten erwünscht. In Anbetracht dessen, dass beim Testen lediglich ein aktiver Anwender beteiligt war, und die Exceptions in der Zielanwendung kontrolliert aufgetreten sind, weitet sich die Problematik unter realen Bedingungen noch weiter aus. Es existiert keine allgemeine Regel, die z.B. definiert, Daten immer direkt aus der Datenbank zu

holen oder zu Beginn einer Session alle Daten zu laden. In Abhängigkeit von dem Arbeitsschritt innerhalb der Webanwendung ist der Bedarf an aktuellen Daten unterschiedlich.

Lösungsansätze: Diese Problematik bedarf eines konkreten Konzepts zur inkonsistenten Datenhaltung. Unter anderem sollte im Konzept genau ausgearbeitet werden, wann und in welchem Ausmaß eine Sessionsäuberung durchzuführen ist. Ebenso sollten in das Konzept Überlegungen einfließen, ob der Anwender von außen die Möglichkeit erhält, aktuelle Daten auf der Datenbank upzudaten, z.B. über einen Refresh-Button. Das Thema der inkonsistenten Datenhaltung ist bekannt und weit verbreitet. An dieser Stelle soll nicht weiter auf diese Thematik eingegangen werden.

6.3. Tests

In diesem Abschnitt wird kurz erwähnt, auf welche Art und Weise das zu entwickelnde Framework getestet wurde. Des Weiteren wird mittels der Testresultate bewertet, inwieweit der Prototyp den gestellten Anforderungen entspricht.

Wie bereits erwähnt, verlief die Testphase parallel zur Realisierungsphase und darüber hinaus.
Während der Realisierungsphase wurden kontinuierlich funktionale Tests durchgeführt. Die funktionalen Tests umfassten unter anderem die Überprüfung einzelner Funktionen mittels JUnit. Für weitere Informationen bezüglich JUnit wird auf die Publikationsseite [JUnit] verwiesen.
Im Anschluss an die Realisierung des Prototypen wurde das Framework weiteren Tests unterzogen. Zu diesem Zweck implementierte die Autorin zwei Test-Zielanwendungen. Das Framework wurde nacheinander in diese Test-Zielanwendungen integriert. Die Test-Zielanwendungen wurden ausgeführt und produzierten an verschiedenen Stellen, unterschiedliche Exceptions. Das Framework agierte in beiden integrierten Test-Zielanwendungen erwartungsgemäß. Die Exceptionfilterung parste alle in den Test-Zielanwendung vorhandenen Exceptions ordnungsgemäß. Die von der Test-Zielanwendungen geworfenen Exceptions wurden alle erkannt und die entsprechenden Informationen persistiert. Direkt im Anschluss wurden die verantwortlichen Instanzen per Email erfolgreich mit entsprechenden Informationen benachrichtigt. Nachfolgend standen alle Informationen bezüglich der aufgetretenen Exception über die Webbenutzeroberfläche bereit, einschließlich der Lokalisierung der Exception und aller vor und nach der Exception aufgerufenen Methoden. Im nächsten Abschnitt 6.4 wird das Framework basierend auf der ersten Test-Zielanwendung präsentiert.

Die Bewertung des entwickelten Frameworks (Prototypen) wird mit Hilfe der Bewertungskriterien aus der Marktanalyse und den existierenden Ansätzen 4 durchgeführt. Diese Bewertungskriterien basieren auf den für das Framework gestellten Anforderungen 3. Nachfolgend werden tabellarisch die Kriterien und die entsprechenden Bewertungen dargestellt. Es sei angemerkt, dass der erste Teil der aufgeführten Kriterien den Bereich des Monitoring und der zweite Teil den Bereich des Debuggings umfasst (äquivalent zur Marktanalyse und den existierenden Ansätzen). Des Weiteren werden in der Tabelle alle Resultate aus der Marktanalyse und den existierenden Ansätzen aufgelistet, um einen besseren Vergleich erzielen zu können.

Bewertungskriterien (Monitoring)	Prototyp	AOP	DynaMICs	Jass	JPaX	JRTM	MaC	MoP	RAC	JAMon	JConsole
Anwendbar für alle Java-Anwendungen	++	++	++	++	++	++	++	++	++	++	++
Anwendbar für vorhandene Java-Anw.	++	++	++	+	+	+	++	+	+	+	++
Automat. Erkennung von Laufzeitfehlern	++	-	+	+	+	+-	+-	+	+	-	- -
Modell- bzw. Spezifikationsunabhängig	++	-	- -	- -	-	- -	- -	- -	- -	+	+
Einfache Integrierbarkeit	+	+	-	-	-	-	-	-	-	+	+
Erfassung aufgetretener Fehler	++	-	++	+	+	+	+	++	+-	+-	-
Erfassung benutzerbezogener Daten	+	-	- -	- -	- -	- -	- -	- -	- -	- -	- -
Sofortiges benachrichten über Fehler	++	-	- -	- -	- -	- -	- -	+-	- -	- -	- -
Bewertungskriterien (Debugging)	**Prototyp**	**DRT**	**JADE**	**JDAS**	**JIVE**	**JPDA**	**jBixbe**	**JDB**	**JSwat**		
Anwendbar für alle Java-Applikationen	++	++	++	++	++	++	++	++	++		
Einfache Integrierbarkeit	+	+-	- -	+-	+-	+-	+	++	++		
Automatische Fehlerlokalisierung	+	+	+	-	-	-	- -	- -	- -		
Debugging auf dem ursprünglichem Stand	+	- -	- -	- -	- -	- -	- -	- -	- -		
Herkömmliches Debugging	+	- -	+	+	++	+	++	+-	+		
Gegenüberstellung der Historie zum aktuellen Stand	+	- -	- -	- -	- -	- -	- -	- -	- -		

Tabelle 6.2.: Evaluation des entwickelten Frameworks (Prototyp)

Wie aus der Tabelle ersichtlich, erfüllt das Framework die aufgestellten Anforderungen. Diese Bewertung sollte aufgrund von zwei wesentlichen Aspekten jedoch kritisch betrachtet werden. Zum einen ist der Prototyp nicht vollständig implementiert und ausgereift, infolgedessen

können Anforderungen, die zu diesem Zeitpunkt als erfüllt eingestuft werden, sich zu einem späteren Zeitpunkt als nur eingeschränkt erfüllt herauskristallisieren. Zum anderen wurden aus zeitlichen Gründen keine Performancetests durchgeführt, die insbesondere bei großen Zielanwendungen von Relevanz sind, um zu überprüfen, ob der vom Framework produzierte Overhead die gewöhnlichen Aktivitäten der Zielanwendung nicht beeinträchtigt.

6.4. System-Präsentation

Im Abschluss des Kapitels Realisierung werden einige Screenshots des realisierten Systems gezeigt. Die Screenshots beziehen sich auf die grafische Benutzersicht des Anwenders (Wartungspersonals). Um den Rahmen dieser Masterarbeit nicht zu sprengen, beschränken sich die Screenshots auf zwei Abläufe; Exceptionparsing und -einstellungen sowie die Ansicht über aufgetretene Exceptions inklusive Teile des historischen Debuggings.
Vorab sei zu erwähnen, dass jedes Benutzerfenster in vier Fragmente unterteilt ist. Die Aufteilung der Benutzerfenster wird, wie bereits erwähnt, mit Hilfe von Tiles [Tiles] erzielt. Der obere Bereich der Seite deckt die Kopfzeile (Header), der untere die Fußzeile (Footer) ab. Der mittlere Bereich besteht aus zwei Teilen; links und rechts. Der linke Teil im mittleren Bereich repräsentiert das Menu, mit Hilfe dessen der Anwender über die Benutzeroberfläche navigieren kann. Der rechte Teil im mittleren Fensterbereich beinhaltet den konkreten Kontext.

6.4.1. Exceptionparsing und -einstellungen

Nach erfolgreicher Integration des gesamten Frameworks in die Zielanwendung müssen vom Anwender einige Aktionen über die Benutzeroberfläche durchgeführt werden. Die Aktivitäten umfassen das Ansteuern des Exceptionparsvorgangs sowie die Einstellungen der Exceptions.
In Abbildung 6.1 sind die bereits geparsten Exceptions der Zielanwendung zu sehen. Dieser Vorgang wurde hervorgerufen durch das Betätigen des Buttons „Exception-Klassen neu parsen“ **(1)**. Dieser Button wird bestätigt, falls die Exception-Klassen noch nicht geparst wurden oder Änderungen an der Zielanwendung erfolgt sind, sodass möglicherweise neue Exceptionklassen hinzugekommen sind.
Alle geparsten Exceptions sind in einer Baumstruktur basierend auf der Vererbungshierarchie angeordnet. Wie graphisch dargestellt, besteht die Möglichkeit, für den Anwender die Einstellungen für jede einzelne Exception separat zu gestalten. Anschließend an den Exceptionnamen besteht die Möglichkeit, eine verantwortliche Instanz **(2)** für diese Exception zu selektieren. Eine verantwortliche Instanz repräsentiert eine einzelne Person (Profil) oder

mehrere Personen (Gruppe), die benachrichtigt werden, falls diese Exception in der Zielanwendung eintritt. Die verantwortlichen Instanzen müssen ebenfalls vom Anwender definiert werden, wie in Abbildung 6.2 und 6.3 dargestellt. Nach der Selektion der verantwortlichen Instanz kann die Priorität **(3)** der Exception definiert werden. Mit Hilfe der Priorisierung wird die Dringlichkeit der Fehlerbehebung ausgedrückt. Unterhalb des Exceptionnamen kann per Radio Button **(4)** definiert werden, auf welche Art und Weise die verantwortliche Instanz beim Auftreten eines Fehlers kontaktiert werden soll. Eine entscheidende Rolle spielt ebenfalls die Aktivierung und Deaktivierung einer Exception, die über die jeweilige Check-Box **(5)** getätigt wird. Ist eine Exception deaktiviert, wird das Eintreten dieser Exception völlig ignoriert, infolgedessen wird weder eine verantwortliche Instanz kontaktiert noch werden Informationen über die aufgetretene Exception ausgegeben. Wie bereits erwähnt, ist dieser Aktivierungsmechanismus sehr wichtig. In Java können Exceptionklassen, die z.B. Businesslogikfehler abbilden und durch einen catch-Block vollständig behoben werden und zudem das Auftreten irrelevant ist, selbst definiert werden. Diese Arten von Exceptions sollten deaktiviert bleiben, um einen unnötigen Overhead zu vermeiden. In Abhängigkeit zur Zielanwendungsgröße ist in der Regel ebenfalls die Anzahl der Exceptionklassen steigend. Um Einstellungen für andere Exception-Klassen übernehmen zu können, besteht die Möglichkeit, Einstellungen einer Exception unter Betätigung des Icon-Button **(6)** für alle Unterklassen der Exception zu übertragen.

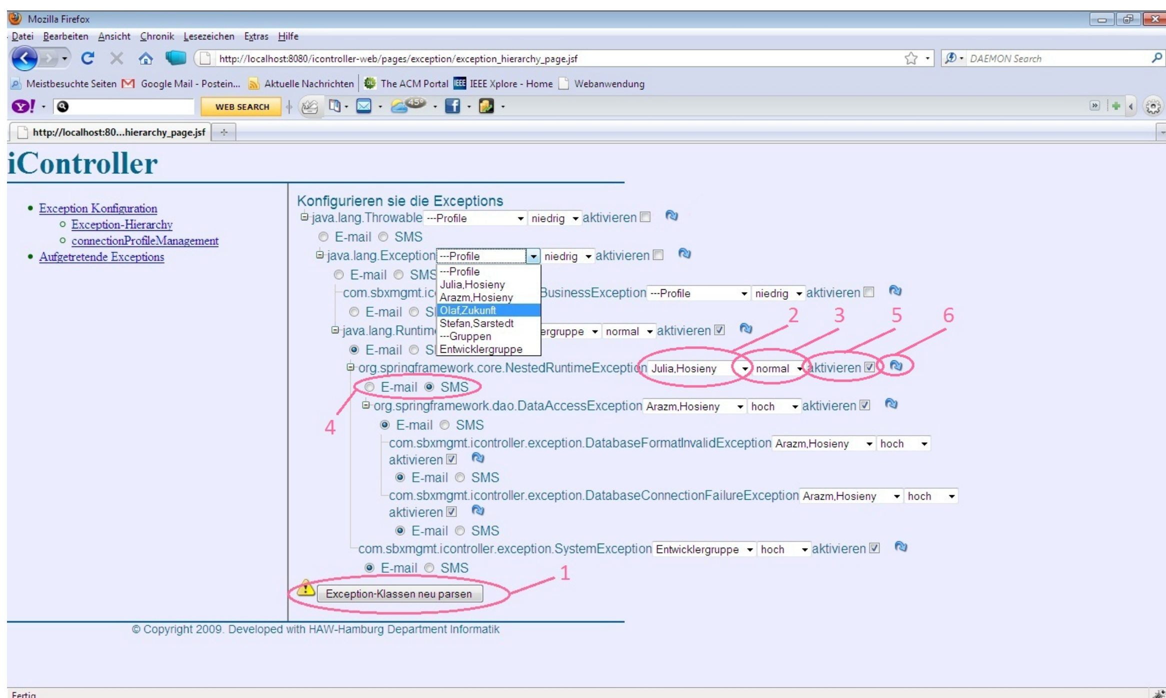

Abbildung 6.1.: Screenshot: Exception-Hierarchie

Wie bereits beim vorherigen Screenshot erwähnt, müssen ebenfalls vom Anwender verantwortliche Instanzen erstellt werden. Die verantwortlichen Instanzen sind zuständig für das Beheben von auftretenden Exceptions, und sie werden vom System kontaktiert, falls eine zugewiesene Exception in der Zielanwendung auftritt. Eine verantwortliche Instanz kann aus einer einzelnen Person (ConnectionProfile) bestehen oder aus mehreren Personen, die eine Gruppe(ConnectionProfileGroup) bilden.
Wie in Abbildung 6.2 zu sehen ist, befinden sich unter dem Menulink „Connection Profile Managment“ **(1)** vier Kontexte zur Verwaltung aller verantwortlichen Instanzen. Es werden an dieser Stelle zwei Screenshots vorgestellt; „Kontaktgruppe erstellen“ (ConnectionProfileGroup) und „Kontakte anzeigen oder editieren“ (ConnectionProfile). Auf die Darstellung der parallelen Seiten “Kontakt erstellen“ und „Kontaktgruppen anzeigen oder editieren“ wird verzichtet.

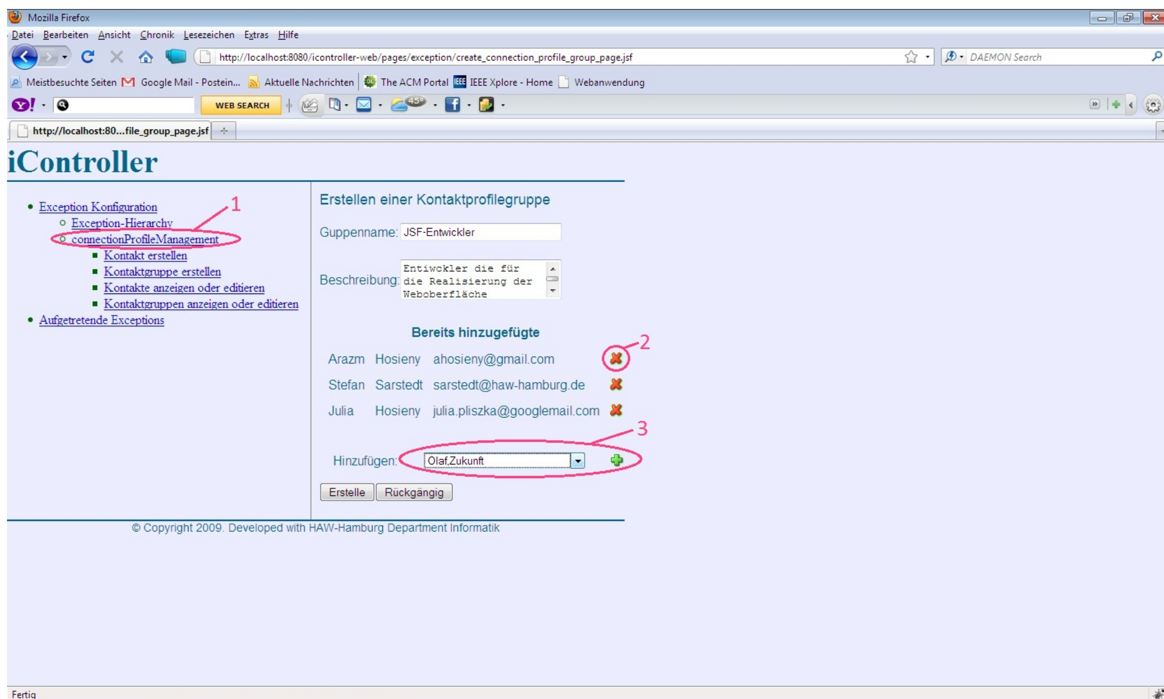

Abbildung 6.2.: Screenshot: Kontaktgruppe erstellen

In Abbildung 6.2 ist ein Screenshot zur Erstellung einer Kontaktgruppe (ConnectionProfileGroup) zu erkennen. Zur Erstellung einer Gruppe bedarf es des Gruppennamens, optional einer Gruppenbeschreibung und mindestens eines Gruppenmitglieds (einer verantwortlichen Person). In der Abbildung 6.2 ist hinter jedem bereits hinzugefügten Gruppenmitglied ein Delete-Icon **(2)** zu sehen, der das Löschen der jeweiligen Person aus der Gruppe ermöglicht. Unter der Auflistung der bereits hinzugefügten Gruppenmitglieder befindet sich eine Selektliste **(3)**. Diese enthält alle potenziellen Mitglieder, die zur Gruppe hinzugefügt

werden können.

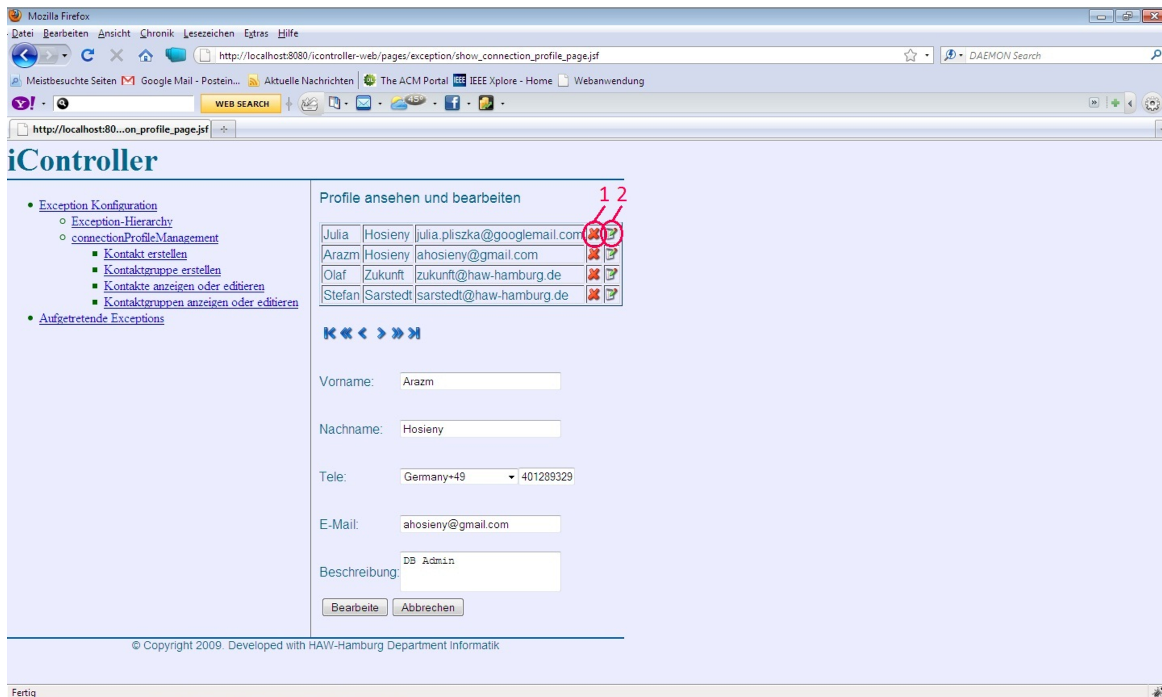

Abbildung 6.3.: Screenshot: Kontaktprofil ansehen und bearbeiten

Abbildung 6.3 repräsentiert die Seite, die zur Ansicht und zur Bearbeitung der bereits existierenden Profile (ConnectionProfiles) dient. Im Kontext Fragment oben werden tabellarisch alle existierenden Kontaktprofile angezeigt. Hinter jedem Kontaktprofil sind zwei Icons platziert. Das erste Icon **(1)** löscht bei Betätigung das entsprechende Kontaktprofil. Die Betätigung des zweiten Icons **(2)** blendet alle Details des entsprechenden Kontaktprofils unter der Tabelle ein und steht zur Modifikation bereit.

Mit Abschluss dieser Aktivitäten kann der Ablauf der Fehlererkennung vollständig durchgeführt werden. Auftretende Exceptions in der Zielanwendung werden erkannt, und bei Bedarf wird die verantwortliche Instanz kontaktiert.

6.4.2. Ansicht über aufgetretene Exceptions inklusive historisches Debugging

Nachdem eine aktive Exception vom Framework erkannt und die entsprechende verantwortliche Instanz über diesen Vorfall informiert wurde, hat der Anwender die Möglichkeit,

diese Exception einzusehen. In der Benachrichtigungsmitteilung ist unter anderem eine ExceptionID enthalten, die jede aufgetretene Exception eindeutig identifiziert. Mit Hilfe dieser Identifikationsnummer kann die entsprechende Exception ausfindig gemacht werden. Abbildung 6.4 stellt die erste Anlaufstelle aller aufgetretenen Exceptions dar. Die Exceptions werden tabellarisch zuzüglich weiterer Informationen aufgelistet. Bei den weiteren Informationen handelt es sich um den vollständigen Exceptionklassennamen der verantwortlichen Instanz dieser Exception, Datum und Uhrzeit, zu der die Exception aufgetreten ist, die Priorität dieser Exception (laut Einstellungen), der aktuelle Status (offen, in Bearbeitung, geschlossen, geschlossen durch Verwerfen der Exception), sowie der verursachende User, falls vorhanden und ermittelbar. Bei dem verursachenden User handelt es sich um die Person, die aktiv an der Produktion des Fehlers beteiligt war und durch ihre Interaktion mit der Zielanwendung den Fehler verursacht oder angesteuert hat.
Für eine performante Suche nach der gewünschten Exception oder nach gewünschten Exceptions steht eine Vielzahl an Filtern zur Verfügung. Die erste Filteralternative ist die Suche nach der ExceptionId, die die Eingabe der ersten Ziffern voraussetzt. Weitere Filtervorgänge sind anhand des Exceptionklassennamens, der verantwortlichen Instanz, der Priorität oder dem Status durchführbar. Des Weiteren wird eine Sortierung, basierend auf Zeit und Datum, aufsteigend oder abfallend, angeboten. Selbstverständlich ist der kombinierte Einsatz der Filter möglich.
Hinter jeder aufgelisteten Exception sind jeweils zwei Icons zu sehen. Durch Betätigung des ersten Icons **(1)** können Details über die aufgetretene Exception eingesehen werden 6.5. Der zweite Icon **(2)** führt zur Ansicht der Abhängigkeit zu anderen Exceptions 6.6.

Abbildung 6.5 zeigt, in welcher Form Informationen über eine ausgewählte aufgetretene Exception sowie deren Verlauf der Exceptionbehandlung dargestellt werden. Der obere Bereich des Kontext konzentriert sich auf die Informationen bezüglich der Exceptionbehandlung und der untere Bereich auf das historische Debugging, inklusive der Informationen die gesammelt wurden, während die Exception aufgetreten ist.
Der Verlauf der Exceptionbehandlung wird unter Verwendung von zwei Aktivitäten protokolliert. Zum einen besteht die Möglichkeit, Kommentare assoziiert an eine aufgetretene Exception zu hinterlassen **(1)**, und zum anderen, den Status zu ändern **(2)**. Am Anfang ist der Status jeder aufgetretenen Exception „offen“. Alle hinzugefügten Kommentare sowie Statusänderungen werden im Verlauf der Exceptionbehandlung **(3)** angezeigt.
Der untere Bereich des Kontext präsentiert das historische Debugging. Als erstes wird die eindeutige ExceptionId und der Exceptionklassenname angezeigt. Darunter befindet sich ein Methodenaufrufbaum. Dieser Baum enthält Methoden, die vor und nach der aufgetretenen Exception aufgerufen wurden. Jede aufgerufene Methode wird zusammen mit dem Rückgabewert, dem Klassennamen, dem Methodennamen, den Parametern, der Uhrzeit und dem Datum (wann die Methode aufgerufen wurde) und der Codezeile angezeigt. Die aufgetretene Exception ist ebenfalls im Methodenbaum in roter Schrift zu sehen, sie be-

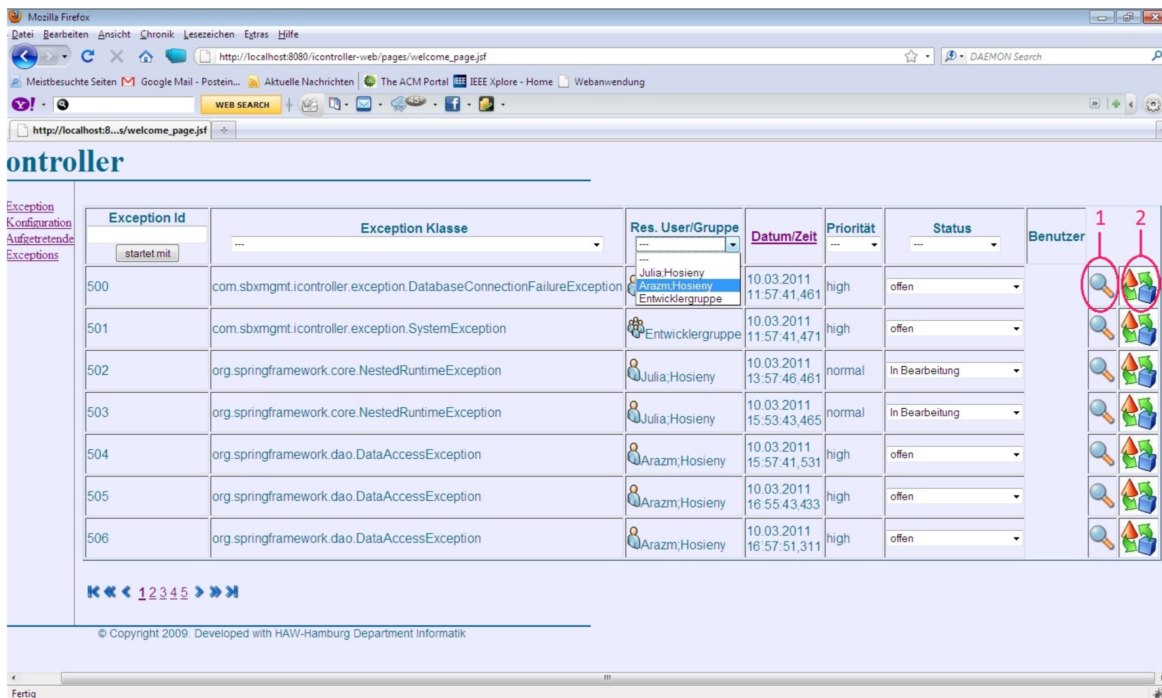

Abbildung 6.4.: Screenshot: Alle aufgetretenen Exceptions

findet sich innerhalb der Methode, in der sie aufgetreten ist. Die Anzahl der präsentierten Methoden im Methodenbaum, die vor der aufgetretenen Exception aufgerufen wurden, kann vom Anwender selber bestimmt werden. Diesbezüglich wird die Summe der gewünschten Methodenaufrufe eingetragen und mit Betätigung des Icons **(4)** durchgeführt. Der Benutzer kann ebenso die Anzahl der aufgerufenen Methoden nach der aufgetretenen Exception definieren **(5)**. Es dient der Performance, dass nur Methoden angezeigt werden, die der Anwender tatsächlich sehen möchte. Für einen kleinen Einblick in die Methodenaufrufe müssen nicht alle abhängigen Methoden geladen werden. Wie bereits erwähnt, handelt es sich beim zweiten unteren Teil des Kontextes um das historische Debugging. Wie auch beim ursprünglichem Debugging, sollen ebenfalls beim historischem Debugging alle Parameter und Rückgabewerte bis ins Detail einsehbar sein. Der Unterschied besteht darin, dass es sich bei den Parametern und Rückgabewerten um diejenigen Objekte handelt, mit denen auch in der Realität gearbeitet wurde, als die Exception erstmalig aufgetreten ist. Im Methodenbaum wird der Typ des Rückgabewerts oder des Parameters angezeigt; sobald der Mauszeiger über diesen Typen gelangt, ist das Objekt **(6)** inspizierbar. Wie auch im Falle des integrierten Debuggers können in Eclipse die Objekte bis ins Detail angesehen werden, unter Betätigung des Plus- Zeichens kann immer tiefergehend in das Objekt hinein geschaut werden.

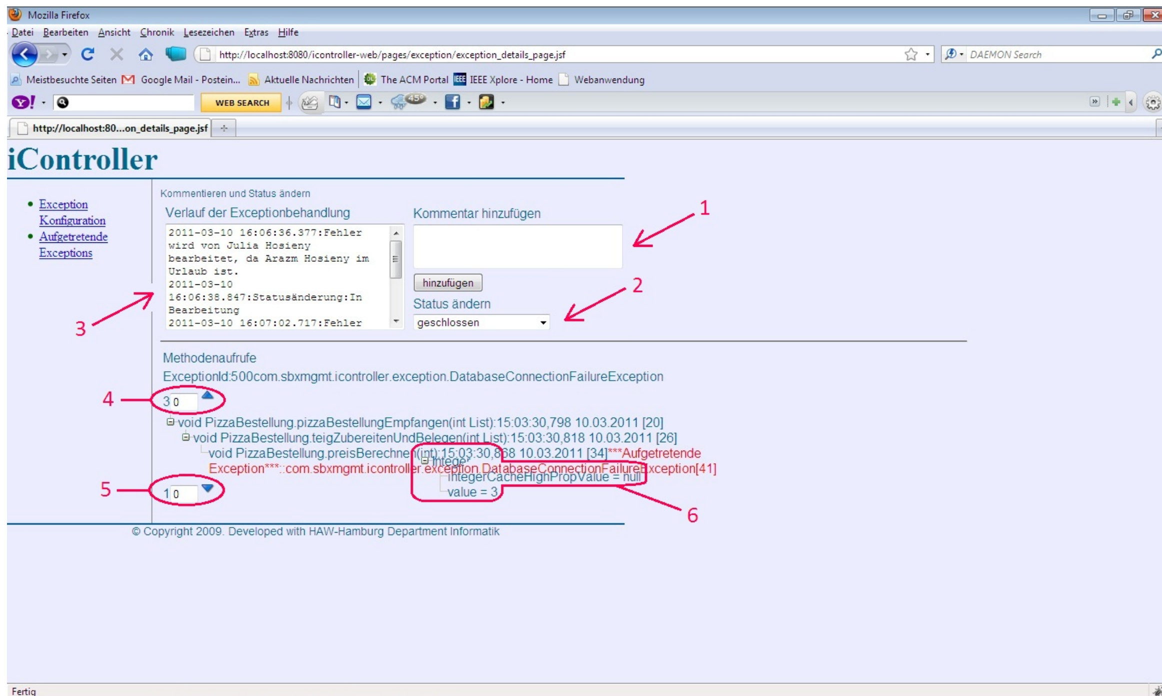

Abbildung 6.5.: Screenshot: Exceptiondetails

Exceptions stehen oftmals in direkter Abhängigkeit zu anderen Exceptions. Das bedeutet, dass eine aufgetretene Exception zum Auftreten einer anderen Exception führen kann. Abbildung 6.6 stellt ein Beispiel von aufgetretenen Exceptions die in Abhängigkeit zueinander stehen, dar. Der Anwender kann sich zu jeder aufgelisteten aufgetretenen Exception 6.4 die abhängigen Exceptions anzeigen lassen 6.6. Die Exceptions werden tabellarisch aufgeführt, zu jeder Exception ist die ExceptionId, der Exceptionklassennamen, das Datum und die Uhrzeit, sowie der aktuelle Status aufgeführt. Aus zwei wesentlichen Gründen ist die Möglichkeit der Betrachtung der Abhängigkeiten von Vorteil. Zum einen dient sie als Informationszusatz bei der Behebung der Exception. Zum anderen kann der Anwender eine Statusänderung für alle abhängigen Exceptions übernehmen. Zum Beispiel können bei einer behobenen Exception alle abhängigen Exceptions ebenfalls den Status „geschlossen“ erhalten, falls diese ebenfalls mit behoben wurden.

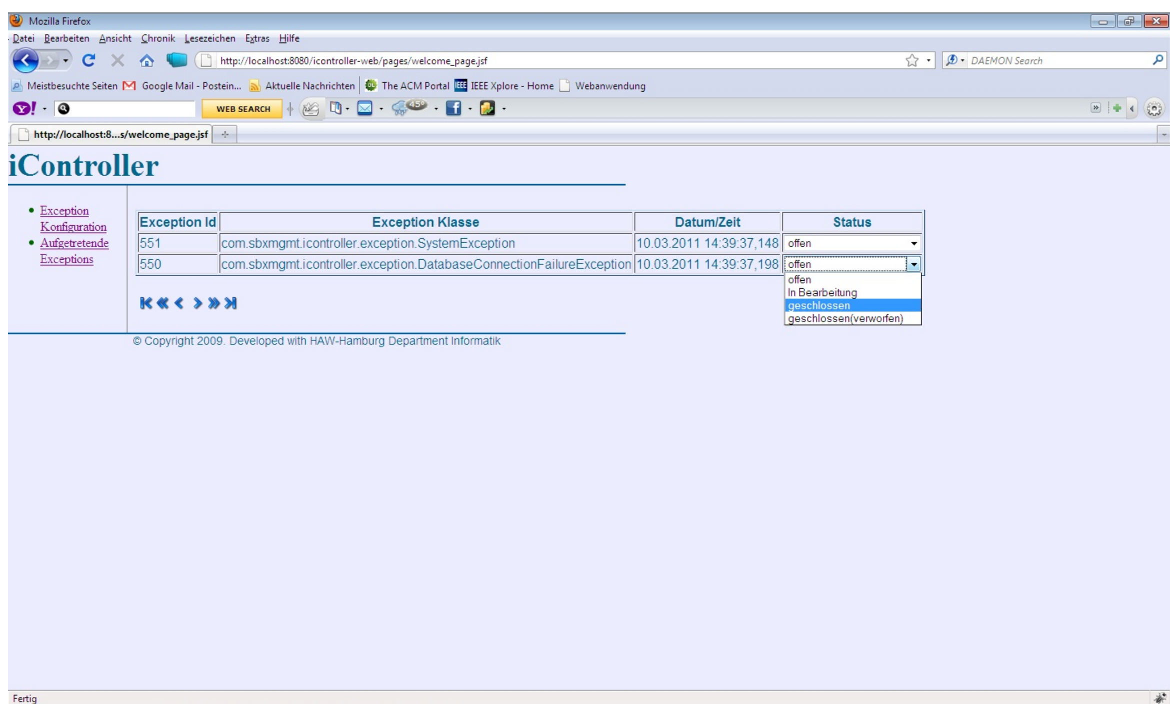

Abbildung 6.6.: Screenshot: Abhängige Exceptions

7. Fazit und Ausblick

Im letzten Kapitel wird ein Fazit der Masterarbeit gezogen und dem Leser ein Ausblick gewährt.

7.1. Fazit

Das Ziel dieser Masterarbeit war die Realisierung eines Frameworks zur automatisierten Fehlererkennung und -lokalisierung für Java-Anwendungen. Zu Beginn und vor der Spezifikation der Anforderungen wurden die Grundlagen behandelt. Das Kapitel Grundlagen dient mehreren Zwecken. Zum einen soll der Leser einen gewissen Wissensstands erreichen, um nachfolgende Themen und Zusammenhänge verstehen und nachvollziehen zu können. Dazu ist die Autorin unter anderem auf einige Debugging-Methodiken eingegangen, die zum besseren Verständnis der existierenden Ansätze und der Resultate der Marktanalyse beitrugen. Zum anderen dienen die Grundlagen als Informationsquelle zum Themengebiet der Masterarbeit. Aus diesem Grund wurden ebenso die Entwicklung und die Vorgehensweisen von Frameworks eingeführt, die zusätzliche Problematiken gegenüber einer herkömmlichen Softwareentwicklung hervor rufen können. Die Abgrenzung einiger Begrifflichkeiten voneinander, um Homonymitäten vorzubeugen, wurde als ein Teil der Grundlagen behandelt. Eine Begriffsabgrenzung wurde für das Framework durchgeführt, da der Begriff der Frameworks häufig synonym für Komponenten, Entwurfsmuster und Klassenbibliotheken verwendet wird. Es treten bezüglich des Begriffs „Fehler“ große Missverständnisse auf, auf die ebenfalls in den Grundlagen eingegangen wurde.

Im Anschluss an die Grundlagenpräsentation beschäftigte sich die Autorin mit der Definition der Anforderungen. Die Aufstellung der Anforderungen umfasst eine Systemidee sowie die fachlichen und technischen Anforderungen. Die Darstellung der Systemidee diente als erste konkrete Spezifikation des Frameworks. Die fachlichen Anforderungen wurden mittels UseCases modelliert, die technischen schriftlich festgehalten. Diese bildeten die Basis für das Design des Frameworks.

Vor der Erstellung des Konzepts und der Architektur erforschte die Autorin die vorhandenen Systeme und die aktuell existierenden Ansätze. Die Anforderungen an das System (Framework) wurden in zwei Bereiche gegliedert, um bessere Resultate zu erzielen. Der erste Bereich umfasst das Monitoring, der zweite das Debugging.
Die Forschungsaktivitäten im Monitoring Bereich ergaben zahlreiche existierende Ansätze und vollständige Monitoringsysteme, die Java kompatibel sind. Keiner dieser Resultate erfüllt jedoch alle Anforderungen. Dies trifft auch auf die Forschungsergebnisse im Debugging Bereich zu. Hier sind ebenfalls existierende Ansätze und Fertiglösungen, die Java kompatibel sind, vorhanden, jedoch werden sie den Ansprüchen nicht komplett gerecht.
Wie oben ausgeführt, ist die Autorin in keinem der beiden Bereiche zu Resultaten gekommen, die allen Anforderungen gerecht werden. Ungeachtet dessen stand der kombinierten Verwendung einiger Ansätze nichts entgegen. In die Konzepterstellung sind unter anderem die existierenden Ansätze Aspektorientiertes Programmieren (AOP) und Java Platform Debugger Architecture (JPDA) eingeflossen.

Die Konzeption und Architekturerstellung des Frameworks wurde in mehreren Schritten durchgeführt. Zu Beginn hat sich die Autorin mit Konzeptfragen, basierend auf den Anforderungen, die an das Framework gestellt werden, beschäftigt. Die Beantwortung der Fragen floss in die Entwicklung des konkreten Konzepts und der Architektur ein. Auf der Basis der identifizierten Anforderungen, der Forschungsergebnisse sowie der Auseinandersetzung mit einigen Konzeptfragen wurde die Architektur konzipiert. Die Architekturerstellung bestand in der Bestimmung der einzelnen Komponenten; es wurden insgesamt 12 Komponenten identifiziert. Im nächsten Schritt fand eine Verteilung der Komponenten statt. Die Autorin diskutierte die Verteilung jeder Komponente und erläuterte deren Auswirkung und Wechselwirkung mit anderen Komponenten. Das Resultat ist eine Adapterclient-Server Architektur. Der Adapterclient wird auf die Zielanwendungen aufgesetzt, infolgedessen erhielt er den Präfix „Adapter". Der Server agiert hier als Webanwendung. Anschließend wurde ausführlicher auf Details der Architektur eingegangen. Die Autorin identifizierte und erläuterte im Detail die Abläufe innerhalb des Frameworks, um dem Leser ein besseres Verständnis der Architektur zu ermöglichen. Die Architektur umfasst drei Abläufe: Exceptionfilterung, Fehlererkennung und Fehlerlokalisierung und -behebung. Zum Abschluss der Konzeption wurde je ein internes Systemmodell für die Adapterclient- und Serverseite Anhand von Klassendiagrammen modelliert.

Das Konzept wurde mittels eines Prototypen in die Praxis umgesetzt. Für die Realisierung des Frameworks wurden unterschiedliche Technologien verwendet. Die Autorin ging auf die essenziellen angewandten Technologien ein und bewertete kurz deren Einsatz. Im Wesentlichen erwiesen sich die verwendeten Technologien als zufriedenstellend. Die Realisierung des Prototypen, basierend auf dem erarbeiteten Konzept, kann grundsätzlich als erfolgreich

angesehen werden. Wie erwartet traten einige Probleme auf, die, obwohl sie keine gravierenden Ausmaße annahmen, auf Basis der vorgeschlagenen Lösungsansätze zu kleinen Konzeptänderungen führten. Die durchgeführten Tests führten im Großen und Ganzen zu den erwarteten Resultaten.

7.2. Ausblick

Das entwickelte Framework stellt einen Prototypen dar. Die wichtigsten Aspekte wurden realisiert, um das Framework auf seine Machbarkeit zu testen. Für die Vollendung des Frameworks ist jedoch ein weiterer Entwicklungsaufwand erforderlich. Das System ist überaus umfangreich, und eine Vollständigkeit geltend machende Entwicklung würde den Rahmen dieser Masterarbeit bei weitem überschreiten. Aus diesem Grund stellt die Weiterentwicklung des Frameworks einen wichtigen Ausblick dieser Arbeit dar. Es existiert eine Reihe von Erweiterungsmöglichkeiten, die einen Mehrwert für das Framework bringen würden. Nachfolgend werden drei Erweiterungen vorgeschlagen.

Wie bereits erwähnt, ist nach der Integration des Frameworks das Parsen der Exceptions erforderlich. Im Anschluss müssen die Einstellungen für die jeweiligen Exceptions durchgeführt werden. Jede Änderung an der Zielanwendung führt zu einer Änderung in der Exceptionhierarchie. Codeänderungen, Integrationen neuer Frameworks oder das Hinzufügen von Klassenbibliotheken haben eine modifizierte Exceptionstruktur zur Folge. Aus diesem Grund ist nach jeder Änderung der Zielanwendung eine erneute Durchführung des Exceptionparsvorgangs erforderlich. Dieser Vorgang hat den Verlust aller vorherigen Exceptioneinstellungen zur Folge und macht durch die erneute Durchführung der Einstellungen einen zusätzlichen Aufwand erforderlich. Aufgrund dieses negativen Nebeneffekts ergibt sich ein weiterer Ausblick. Ein erneut durchgeführter Parsvorgang soll einen Abgleich der alten und neuen Exceptionhierarchie durchführen und Exceptioneinstellungen übernehmen. Dadurch müssen lediglich Einstellungen für neu hinzugekommende Exceptions durchgeführt werden.

Der Protokollierungsauswand für aufgerufene Methoden produziert einen sehr hohen Overhead. Jede Methode wird in der Datenbank abgespeichert. Mit Hilfe von Hibernate besteht die Möglichkeit, sogenannte Verbindungspools zu konfigurieren. Diese Pools sind zuständig für eine dauerhaft gehaltene Verbindung zur Datenbank. In der Konsequenz entfällt für jeden Datenbankzugriff der aufwendige Verbindungsaufbau (inklusive der Erstellung des physikalischen Kanals und die Durchführung des Handshakes) und -abbau, wodurch ein unnötiger Aufwand vermieden wird. Dennoch ist der Overhead weiterhin sehr hoch. Eine

weitere Zielvorgabe lautet aus diesem Grund die Reduzierung dieses Overheads. Ein möglicher Ansatz besteht darin, die Informationen über die aufgerufenen Methoden bis zu einer gewissen Anzahl temporär in der Anwendung zu halten und im Anschluss bei Bedarf diese Ansammlung gebündelt in die Datenbank zu übertragen. Der Bedarf an einer Abspeicherung der Daten in der Datenbank erfolgt, wenn eine Exception innerhalb der Methodenkette auftritt.

Als letzter Ausblick sei die Erweiterung des Frameworks um weitere Zielanwendungen genannt. Derzeit ist es vorgesehen, einen Adapterclient auf eine Zielanwendung aufzusetzen, wobei eine Webanwendung (Server) für die Fehlerverwaltung und -behebung dieser Zielanwendung verantwortlich ist. Dies hat zur Folge, dass mehrere zu kontrollierende Zielanwendungen mehrere Adapterclients und ebenso viele Webanwendungen (Server) erforderlich machen. Es bestünde die Möglichkeit, das Framework dergestalt zu erweitern, dass eine Webanwendung (Server) für mehrere Zielanwendungen zuständig ist. Auf jede Zielanwendung wird ein Adapterclient aufgesetzt, und alle diese Adapterclients kooperieren mit dem gleichen Server (Webanwendung). Dementsprechend können unter Einsatz nur einer einzigen Webanwendung alle Fehler aller Zielanwendungen behoben und verwaltet werden.

Literaturverzeichnis

[Apache 2011] APACHE: *Apache Subversion - Enterprise-class centralized version control for the masses*. Apache - http://subversion.apache.org/. 2011

[AspectJ] ASPECTJ: *Publikationsseite*. Eclipse Foundation - http://www.eclipse.org/aspectj/

[AspectWerkz] ASPECTWERKZ: *Publikationsseite*. http://aspectwerkz.codehaus.org/

[Axis2] AXIS2: *Publikationsseite*. Apache Software Foundation - http://axis.apache.org/axis2/java/core/

[Bartetzko u. a. 2001] BARTETZKO, Detlef ; FISCHER, Clemens ; MÖLLER, Michael ; WEHRHEIM, Heike: *Jass - Java with Assertions*. CiteSeerX. 2001

[Bauer 2000] BAUER, Günther: *Bausteinbasierte Software*. Vieweg, 2000. – ISBN 3-528-05722-X

[Birrer u. a. 1995] BIRRER, Andi ; BISCHOFBERGER, Dr. Walter R. ; EGGENSCHWILER, Thomas: Wiederverwendung durch Frameworktechnik - vom Mythos zur Realität. In: *OBJEKTspektrum* (1995)

[Bischofs 2000] BISCHOFS, Ludger: *Grundlagen der komponentenbasierten Softwareentwicklung*, Universität Oldenburg, Diplomarbeit, 2000

[Bosch u. a. 1997] BOSCH, Jan ; MOLIN, Peter ; MATTSSON, Michael ; BENGTSSON, PerOlof: *Object-Oriented Frameworks-Problems and Experiences*, University of Karlskrona/Ronneby, Research Report, 1997

[Burdy u. a. 2004] BURDY, Lilian ; CHEON, Yoonsik ; COK, DavidR. ; ERNST, Michael D. ; KINIRY, Joseph R. ; LEAVENS, Gary T. ; LEINO, K. Rustan M. ; POLL, Erik: *An overview of JML tools and applications*. Springer-Verlag. 2004

[C++] C++: *Publikationsseite*. Bjarne Stroustrup - http://www2.research.att.com/~bs/C++.html

[C-Sharp] C-SHARP: *Publikationsseite*. Microsoft - http://msdn.microsoft.com/en-us/vcsharp/default.aspx

[Chaim u. a. 2003] CHAIM, Marcos L. ; MALDONADO, Jose C. ; JINO, Mario: *A Debugging Strategy Based on Requirements of Testing*. IEEE - http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1192424. 2003

[Chen u. a. 2004] CHEN, Feng ; D'AMORIM, Marcelo ; ROSU, Grigore: *Monitoring-Oriented Programming: A Tool-Supported Methodology for Higher Quality Object-Oriented Software*. University of Illinois at Urbana-Champaign - Department of Computer Science. 2004

[Chen und Rosu 2003] CHEN, Feng ; ROSU, Grigore: *Towards Monitoring-Oriented Programming:A Paradigm Combining Specification and Implementation*. CiteSeerX. 2003

[Chen und Rosu 2007] CHEN, Feng ; ROSU, Grigore: *MOP: An Efficient and Generic Runtime Verification Framework*. ACM - http://portal.acm.org/citation.cfm?id=1297027.1297069. 2007

[Cheon und Leavens 2002] CHEON, Yoonsik ; LEAVENS, Gary T.: *A Runtime Assertion Checker for the Java Modeling Language (JML)*. CiteSeerX - http://archives.cs.iastate.edu/documents/disk0/00/00/02/74/00000274-00/jmlrac.pdf. 2002

[Chmiel und Loui 2004] CHMIEL, Ryan ; LOUI, Michael C.: *Debugging: From Novice to Expert*. ACM - http://portal.acm.org/citation.cfm?id=971300.971310. 2004

[Chung 2004] CHUNG, Mandy: *Using JConsole to Monitor Applications*. http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html. 2004

[Claus und Schwill 1993] CLAUS, Prof. Dr. V. ; SCHWILL, Dr. A.: *Duden - Informatik*. Dudenverlag Mannheim, 1993. – ISBN 3-411-05232-5

[Clavel u. a. 2010] CLAVEL, M. ; DURÁN, F. ; EKER, S. ; LINCOLN, P. ; MARTÍ-OLIET, N. ; MESEGUER, J. ; QUESADA, J. F.: *Theoretical Computer Science - Maude: specification and programming in rewriting logic*. 2010. – 187–243 S

[Cleve und (Hrsg.) 2001] CLEVE, Holger ; (HRSG.), Andreas Z.: *Automatisches Debugging*, Universität Passau - Lehrstuhl für Software-Systeme, Diplomarbeit, 2001

[Csikai 1985] CSIKAI, Redaktion K.: *Fachausdrücke der Informationsverarbeitung - Wörterbuch und Glossar*. IBM Deutschland GmbH, 1985

[Cunha u. a. 1998] CUNHA, José C. ; LOURENÇO, João ; VIEIRA, João ; MOSCÃO, Bruno ; PEREIRA, Daniel: *High-Performance Computing and Networking*. Springer Berlin / Heidelberg, 1998. – 708 S

[CVS] CVS: *Publikationsseite*. CVS Team - http://savannah.nongnu.org/projects/cvs

[Czyz und Jayaraman 2007] CZYZ, Jeffrey K. ; JAYARAMAN, Bharat: *Declarative and Visual Debugging in Eclipse*. ACM - http://portal.acm.org/citation.cfm?id=1328279.1328286&coll=DL&dl=GUIDE&CFID=111567748&CFTOKEN=45609344. 2007

[Delgado u. a. 2004] DELGADO, Nelly ; GATES, Ann Q. ; ROACH, Steve: *A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools*. IEEE - http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1377185&tag=1. 2004

[Dunkel und Holitschke 2003] DUNKEL, Jürgen ; HOLITSCHKE, Andreas: *Softwarearchitektur für die Praxis*. Springer-Verlag Berlin Heidelberg, 2003. – ISBN 3-540-00221-9

[DynamicAspects] DYNAMICASPECTS: *Publikationsseite*. http://dynamicaspects.sourceforge.net/

[Eclipse] ECLIPSE: *Publikationsseite*. Eclipse Foundation - http://www.eclipse.org/

[Enzyklopädie 2010] ENZYKLOPÄDIE, Wikipedia: *Monitoring*. Wikipedia - http://de.wikipedia.org/wiki/Monitoring. 2010

[Fayad 2000] FAYAD, Mohamed E.: *Introduction to the Computing Surveys' Electronic Symposium on Object-Oriented Application Frameworks*. ACM - http://portal.acm.org/citation.cfm?id=351937. 2000

[Gamma u. a. 2001] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 2001. – ISBN 3-89319-950-0

[Gates u. a. 2001] GATES, A. Q. ; ROACH, S. ; MONDRAGON, O. ; DELGADO, N.: *DynaMICs: Comprehensive Support for Run-Time Monitoring*. CiteSeerX - http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.7.8483. 2001

[Gates und Teller 1999] GATES, Ann Q. ; TELLER, Patricia J.: *DynaMICs: An Automated and Independent Software-Fault Detection Approach*. IEEE - http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=809470&tag=1. 1999

[Gosling u. a. 2005] GOSLING, James ; JOY, Bill ; STEELE, Guy ; BRACHA, Gilad: *The Java Language Specification - Third Edition*. Addison-Wesley, 2005. – ISBN 0-321-24678-0

[Grässle u. a. 2007] GRÄSSLE, Patrick ; BAUMANN, Henriette ; BAUMANN, Philippe: *UML 2 - projektorientiert*. Galileo Computing, 2007. – ISBN 978-3-8362-1014-0

[Grottke und Trivedi 2007] GROTTKE, Michael ; TRIVEDI, Kishor S.: *Fighting Bugs:Remove, Retry, Replicate, and Rejuvenate*. IEEE - http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4085640. 2007

[van Grup und Bosch 2001] GRUP, J. van ; BOSCH, J.: *Design,implementation and evolution of object oriented frameworks: concepts and guidelines*. Software-Practice & Experience - http://portal.acm.org/citation.cfm?id=370152. 2001

[Havelund und Rosu 2001] HAVELUND, Klaus ; ROSU, Grigore: *Monitoring Java Programs with Java PathExplorer*. ACM - http://portal.acm.org/citation.cfm?id=891177. 2001

[Havelund und Rosu 2004] HAVELUND, Klaus ; ROSU, Grigore: *An Overview of the Runtime Verification Tool Java PathExplorer - Formal Methods in System Design*. 2004

[Hibernate] HIBERNATE: *Publikationsseite*. JBoss - http://www.hibernate.org/

[IEEE 1990] IEEE: *Standard Glossary of Software Engineering Terminology*. Std 610.12-1990. 1990

[Java] JAVA: *Publikationsseite*. Oracle - http://www.oracle.com/technetwork/java/javase/downloads/index.html

[Javassist] JAVASSIST: *Publikationsseite*. JBoss - http://www.jboss.org/javassist

[JAX-WS] JAX-WS: *Publikationsseite*. Oracle - http://java.sun.com/developer/technicalArticles/J2SE/jax_ws_2/

[jBixbe] JBIXBE: *Publikationsseite*. http://www.jbixbe.com/

[JBoss] JBOSS: *Publikationsseite*. JBoss (zu Red Hat gehörendes Unternehmen) - http://www.jboss.org/jbossweb

[JDB] JDB: *Publikationsseite*. Oracle - http://download.oracle.com/javase/1.3/docs/tooldocs/solaris/jdb.html

[Johnson 1997] JOHNSON, Ralph E.: *Frameworks=(Components+Patterns)*. ACM - http://portal.acm.org/citation.cfm?id=262799. 1997

[Johnson und Foote 1988] JOHNSON, Ralph E. ; FOOTE, Brian: Designing Reuseable Classes. In: *Journal of Object-Oriented Programming* (1988), Nr. Volume 1, Number 2 / Juni 1988

[JPDA] JPDA: *Publikationsseite*. Oracle - http://java.sun.com/javase/technologies/core/toolsapis/jpda/

[JSF] JSF: *Publikationsseite*. Apache Software Foundation - http://myfaces.apache.org/

[JSwat] JSWAT: *Publikationsseite*. http://code.google.com/p/jswat/

[JUnit] JUNIT: *Publikationsseite*. http://www.junit.org/

[Kappel und Hitz 1999] KAPPEL, Gerti ; HITZ, Martin: *UML @ Work - Von der Analyse zur Realisierung*. dpunkt.verlag, 1999. – ISBN 3-932588-38-X

[Kecher 2006] KECHER, Christoph: *UML 2.0 - Das umfassende Handbuch*. Galileo Computing, 2006. – ISBN 3-89842-738-2

[Kim 2001] KIM, Moonjoo: *Information Extraction for Run-time Formal Analysis*, University of Pennsylvania, Doktorarbeit, 2001

[Kim u. a. 1999] KIM, Moonjoo ; VISWANATHAN, Mahesh ; BEN-ABDALLAHY, Hanene ; KANNAN, Sampath ; LEE, Insup ; SOKOLSKY, Oleg: *Formally Specified Monitoring of Temporal Properties*. IEEE - http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=777457. 1999

[Kim u. a. 2004] KIM, Moonjoo ; VISWANATHAN, Mahesh ; KANNAN, Sampath ; LEE, Insup ; SOKOLSKY, Oleg: *Java-MaC: A Run-Time Assurance Approach for Java Programs*. SpringerLink - http://www.springerlink.com/content/wg1t2628pl24548q/. 2004

[Krinke 2003] KRINKE, Jens: *Advanced Slicing of Sequential and Concurrent Programs*, Universität Passau, Dissertation, 2003

[Krümberg 2006] KRÜMBERG, Andreas: *Isolierung und Vereinfachung von fehlerauslösenden Testfällen mit Hilfe des Delta-Debugging-Algorithmus*. Westfälische Wilhelms-Universität Münster. 2006

[Leavens u. a. 1998] LEAVENS, Gary T. ; BAKER, Albert L. ; RUBY, Clyde: *JML: a Java Modeling Language*. Iowa State University - http://www.csg.is.titech.ac.jp/~muga/paper/others/oopsla/1998/jml.pdf. 1998

[Liang und Xu 2005] LIANG, Donglin ; XU, Kai: *Debugging ObjectOriented Programs with Behavior Views*. ACM - http://portal.acm.org/citation.cfm?id=1085148. 2005

[log4j] LOG4J: *Publikationsseite*. Apache Software Foundation - http://axis.apache.org/axis2/java/core/

[Madsen u. a. 2006] MADSEN, H. ; THYREGOD, P. ; BURTSCHY, B. ; ALBEANU, G. ; POPENTIU, F.: *A fuzzy logic approach to software testing and debugging*. http://www2.imm.dtu.dk/~popentiu/MADFLA.pdf. 2006

[Maier] MAIER, Peter: *Kritische Erfolgsfaktoren objektorientierter Frameworks*

[Manna und Pnueli 1992] MANNA, Zohar ; PNUELI, Amir: *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992. – ISBN 3-540-97644-7

[Markiewicz und Lucena 2001] MARKIEWICZ, Marcus E. ; LUCENA, Carlos J.: *Object Oriented Framework Development*. ACM - http://portal.acm.org/citation.cfm?id=372765.372771. 2001

[Martin u. a. 1997] MARTIN, Robert C. ; RIEHLE, Dirk ; BUSCHMANN, Frank: *Pattern languages of program design 3*. Addison-Wesley, 1997. – 656 S

[Mateis u. a. 2000] MATEIS, Cnstinel ; STUMPTNER, Markus ; WIELAND, Dominik ; WOTAWAT, Franz: *JADE - AI Support for Debugging Java Programs*. IEEE - http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00889847. 2000

[Mayer 1999] MAYER, Wolfram: *Wiederverwendung von Softwarekomponenten und Rahmenwerken*, Universität Karlsruhe, Diplomarbeit, 1999

[Mayr 2005] MAYR, Herwig: *Projekt Engineering - Ingenieurmäßige Softwareentwicklung in Projektgruppen*. Fachbuchverlag Leipzig, 2005. – ISBN 3-446-40070-2

[Mok und Liu 1997a] MOK, Aloysius K. ; LIU, Guangtian: *Early Detection of Timing Constraint Violation at Runtime*. IEEE - http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=641280. 1997

[Mok und Liu 1997b] MOK, Aloysius K. ; LIU, Guangtian: *Efficient Run-Time Monitoring of Timing Constraints*. IEEE - http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=601363&userType=&tag=1. 1997

[MySQL] MYSQL: *Publikationsseite*. Sun Microsystems (Fusionierung mit Oracle) - http://www.mysql.de/

[NetBeans] NETBEANS: *Publikationsseite*. Sun Microsystems (Fusionierung mit Oracle) - http://netbeans.org/

[Neumann 2005] NEUMANN, Christian: *Bewertung von Open Source Frameworks als Ansatz zur Wiederverwendung*, Wirtschaftsuniversität, Diplomarbeit, 2005

[Nusayr und Cook 2009] NUSAYR, Amjad ; COOK, Jonathan: *AOP for the domain of runtime monitoring: breaking out of the code-based model*. ACM - http://portal.acm.org/citation.cfm?id=1509310&coll=GUIDE&dl=ACM&CFID=33304032&CFTOKEN=99858450&ret=1#Fulltext. 2009

[Oracle 2010] ORACLE: *JPDA Java SE Documentation*. Oracle - http://download.oracle.com/javase/6/docs/technotes/guides/jpda/jpda.html. 2010

[Polke 2004] POLKE, Oliver: *Vorteile der Softwareentwicklung mit Framework-Technologie*, Bayerische Julius-Maximilians-Universität Würzburg, Diplomarbeit, 2004

[PostgreSQL] POSTGRESQL: *Publikationsseite*. PostgreSQL Global Development Group - http://www.postgresql.org/

[Rakesh 2010] RAKESH, M.G.: *A Lightweight Approach for Program Analysis and Debugging*. ACM - http://portal.acm.org/citation.cfm?id=1730874.1730880&coll=DL&dl=GUIDE&CFID=111567748&CFTOKEN=45609344. 2010

[Richter 2005] RICHTER, Jan: *Einführung in AspectJ - Konzepte und Werkzeuge für die Softwareentwicklung mit Java*, Universität Leipzig - Institut für Informatik, Diplomarbeit, 2005

[Robinson 2002] ROBINSON, William N.: *Monitoring Software Requirements using Instrumented Code*. IEEE - http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=994468&tag=1. 2002

[Robinson 2005] ROBINSON, William N.: *Implementing Rule-based Monitors within a Framework for Continuous Requirements Monitoring*. IEEE - http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1385616. 2005

[Rutar u. a. 2004] RUTAR, Nick ; ALMAZAN, Christian B. ; FOSTER, Jeffrey S.: *A Comparison of Bug Finding Tools for Java*. IEEE Computer Society, 2004. – ISBN 0-7695-2215-7

[Schmid 1997] SCHMID, Dr. Hans A.: Objektorientierte Entwurfsmuster und Frameworks in der Informatik-Ausbildung an der Fachhochschule Konstanz. In: *Informatik-Spektrum;Springer-Verlag* (1997), Nr. Volume 20, Number 6 / Dezember 1997, S. 364–371

[Schmitz 2004] SCHMITZ, Alexander: *Automatisierung des Testens objektorientierter Frameworks*, Universität Dortmund, Diplomarbeit, 2004

[Schütte und Vering 2004] SCHÜTTE, R. ; VERING, O.: *Erfolgreiche Geschäftsprozesse durch standardisierte Warenwirtschaftssysteme - Marktanalyse, Produktübersicht, Auswahlprozess*. Springer-Verlag, 2004. – Kapitel 2.5 S

[Sender 2008] SENDER, Tim: *Deklaratives Debugging*. Westfälische Wilhelms-Universität Münster. 2008

[Shapiro 1983] SHAPIRO, Ehud Y.: *Algorithmic Program Debugging*. MIT Press. 1983

[Souza 2007] SOUZA, Steve: *Java Application Monitoring*. http://jamonapi.sourceforge.net/. 2007

[Sparks u. a. 1996] SPARKS, Steve ; BENNER, Kevin ; FARIS, Chris ; CONSULTING, Andersen: *Managing Object-Oriented Framework Reuse*. IEEE - http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=536784. 1996

[Spring] SPRING: *Publikationsseite*. SpringSource - http://www.springsource.org/

[Starke 2002] STARKE, Gernot: *Effektive Software-Architekturen:Ein praktischer Leitfaden*. Hanser, 2002. – ISBN 3-446-21998-6

[SVN] SVN: *Publikationsseite*. Apache Software Foundation - http://subversion.apache.org/

[Tiles] TILES, Struts: *Publikationsseite*. Apache Software Foundation - http://struts.apache.org/1.x/struts-tiles/index.html

[Tomcat] TOMCAT, Apache: *Publikationsseite*. Apache Software Foundation - http://tomcat.apache.org/

[Tutzschke 2008] TUTZSCHKE, Jan-Peter: *Reduktion der Komplexität in pervasiven Spielen - Konzeption und Realisierung eines Rahmenwerks*, HAW-Hamburg, Masterarbeit, 2008

[Ujorm] UJORM: *Publikationsseite*. Pavel Ponec - http://ujoframework.org/

[Vaidyanathan und Trivedi 2001] VAIDYANATHAN, Kalyanaraman ; TRIVEDI, Kishor S.: *Extended Classification of Software Faults Based on Aging*. Dept. of ECE, Duke University Durham, NC 27708, USA - ISSRE and Chillarege Corp. 2001

[Veiser 2007] VEISER, Simon: *Projektvorgehen bei der Einführung eines elektronischen Datenarchivs*, Fachhochschule Düsseldorf - University of Applied Sciences, Diplomarbeit, 2007

[Vlachakis u. a. 2005] VLACHAKIS, Joannis ; KIRCHHOF, Anja ; GURZKI, Thorsten: *Marktübersicht - Portal Software - für Business-, Enterprise-Portale und E-Collaboration*. Fraunhofer IRB Verlag, Stuttgart, 2005. – 10–13 S

[Weiser 1979] WEISER, Mark: *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. ACM - http://portal.acm.org/citation.cfm?id=909356. 1979

[Zeller 2009] ZELLER, Andreas: *Why Programs Fail*. Elsevier, 2009. – ISBN 978-0-12-374515-6

[Zeller und Hildebrandt 2002] ZELLER, Andreas ; HILDEBRANDT, Ralf: *Simplifying and Isolating Failure-Inducing Input*. IEEE - http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=988498. 2002

A. Anwendungsfälle

ID	0
Name	Ein- und Ausloggen
Kurzbeschreibung	Ein- und Ausloggen
Akteur	Wartungspersonal, Projektmanagement
Vorbedingung	—
Nachbedingung	Der Akteur ist eingeloggt und kann den Funktionsumfang, der für seine Rolle definiert ist, nutzen.
Hauptszenario	1. Einloggen 2. Ausloggen, falls eingeloggt ist.
Alternativabläufe	—
Fehlersituationen	Meldung über Fehlerverhalten anzeigen.

Tabelle A.1.: UC0: Ein- und Ausloggen

ID	8
Name	Sourcecodemodifikation durchführen
Kurzbeschreibung	Der Akteur kann den Sourcecode der Java-Zielanwendung modifizieren.
Akteur	Wartungspersonal
Vorbedingung	Das Wartungspersonal muss erfolgreich eingeloggt sein.
Nachbedingung	Der Sourcecode ist modifiziert und die Änderungen aktiv.
Hauptszenario	1. Der Akteur hat direkten Zugriff auf den Sourcecode der Java-Zielanwendung. 2. Der Akteur kann Modifikationen durchführen und die Änderungen werden für die Java-Zielanwendung übernommen.
Alternativabläufe	—
Fehlersituationen	Meldung über Fehlerverhalten anzeigen.

Tabelle A.2.: UC8: Sourcecodemodifikation durchführen

ID	9
Name	Korrektur testen
Kurzbeschreibung	Der modifizierte Sourcecode wird getestet.
Akteur	Wartungspersonal
Vorbedingung	Wartungspersonal muss erfolgreich eingeloggt sein (UC0). Sourcecode wurde modifiziert.
Nachbedingung	—
Hauptszenario	1. Der modifizierte Sourcecode wird ausgeführt. 2. Das Wartungspersonal kann diesen Prozess verfolgen und sehen, ob der ursprüngliche Fehler ausgelöst wird.
Alternativabläufe	Sollte kein Sourcecode vorab modifiziert worden sein, wird diese Information dem Wartungspersonal mitgeteilt.
Fehlersituationen	Meldung über Fehlerverhalten anzeigen.

Tabelle A.3.: UC9: Korrektur testen

ID	10
Name	Modifikation aufheben
Kurzbeschreibung	Der ursprüngliche Sourcecode wird wiederhergestellt.
Akteur	Wartungspersonal
Vorbedingung	Das Wartungspersonal muss erfolgreich eingeloggt sein (UC0). Eine Modifikation muss erfolgt sein (UC9).
Nachbedingung	Der ursprüngliche Sourcecode ist wiederhergestellt.
Hauptszenario	1. Der ursprüngliche Sourcecode wird wiederhergestellt.
Alternativabläufe	Sollte kein Sourcecode vorab modifiziert worden sein, wird diese Information dem Wartungspersonal mitgeteilt.
Fehlersituationen	Meldung über Fehlerverhalten anzeigen.

Tabelle A.4.: UC10: Modifikation aufheben

B. Inhalt der beigefügten CD-ROM

Die beigefügte CD-ROM enthält folgende Dateien:

- `\thesis.pdf`: Die schriftliche Ausarbeitung der Masterthesis.
- `\workspace`: Der gesamte Workspace, der den Sourcecode umfasst.
  - `\icontroller-adapter`: Der Sourcecode des Adapterclients.
  - `\icontroller-web`: Der Sourcecode des Servers (Webanwendung).
- `\.m2\`: Die verwendeten Java-Klassenbibliotheken, die mit Hilfe von Maven verwaltet wurden.
- `Dokumentation`: Dokumentationen des Frameworks.
- `Software`: Einige verwendete Open Source Programme.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 13. Mai 2011

Ort, Datum	Unterschrift