Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Dennis Dedaj

Fehlerlokalisierung und Fehlereingrenzung in Java-basierten Programmen für Client-Server-Architekturen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Dennis Dedaj

Fehlerlokalisierung und Fehlereingrenzung in Java-basierten Programmen für Client-Server-Architekturen

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

Betreuender Prüfer: Prof. Dr. rer. nat. Bettina Buth
Zweitgutachter: Prof. Dr.-Ing. Martin Hübner

Abgegeben am 11. Juni 2012

Dennis Dedaj

Thema der Masterarbeit

Fehlerlokalisierung und Fehlereingrenzung in Java-basierten Programmen für Client-Server-Architekturen

Stichworte

Testen, Fehlerlokalisierung, Web Applications, Dynamic Slicing, Fault Containment, Fault Propagation

Kurzzusammenfassung

Fehler in komplexen Softwaresystemen zu Identifizieren ist eine aufwändige Aufgabe. Der Prozess des Debuggings teilt sich in das Finden und das Beheben des Fehlers auf. Das Beheben eines bereits entdeckten Fehlers ist nicht so aufwändig wie das Finden des Fehlers. Der Zustand eines Softwaresystems besteht aus einer Vielzahl von Variablen, welche sich zu jedem Zeitpunkt ändern können. Mit dieser Arbeit wird die Machbarkeit einer automatisierten Fehlerlokalisierung in Java-Programmen erforscht. Dazu werden vorerst die für die Lokalisierung von Fehlern in Java-Programmen relevanten Fehlerwirkungen identifiziert. Anschließend werden die in Client-Server-Architekturen möglichen Propagierungen der Fehler erarbeitet. Auf Basis der Fehlerwirkungen und der möglichen Propagierungen werden dann die für die angestrebte Fehlerlokalisierung denkbaren Softwarefehler identifiziert. Das Konzept beruht auf der Instrumentierung der Ausführung von erfolgreichen und fehlschlagenden Testfällen, um mit Hilfe von dynamischen Slices die Ausführungspfade zu extrahieren. Die Ausführungspfade werden wiederum mit ausgewählten Mengenoperationen zu einem resultierenden Slice vereinigt. Der resultierende Slice wird dem Softwareentwickler präsentiert. Die Präsentation des vereinigten Slices soll dazu führen, dass der Softwareentwickler nur eine reduzierte Menge von Anweisungen untersuchen muss, um den jeweiligen Fehler zu erkennen.

Title of the paper

Fault Localization and Fault Containment for Java-based Programs in Client-Server-Architectures

Keywords

Testing, Fault Localization, Web Applications, Dynamic Slicing, Fault Containment, Fault Propagation

Abstract

The identification of failures in complex software system is an elaborate task. The process of debugging consists of finding and solving a failure. Solving the failure is not as elaborate as finding it. The status of a software system depends on a great variety of variables, which can change at any time. This work understands itself as an proof of concept for an automated localization technique of failures in Java programs. At first, the failures will be identified in order to compile possible propagations of failures in client-server-architectures. Based on the identified failures and possible propagations a set of failure-causing software faults is generated. This concept consists of the instrumentation of successful and failing test case executions to extract execution traces by dynamic slices. The execution traces are joined by chosen set operations to resulting sets of slices. The resulting slice is presented to the software developer. The presentation should help the developer to identify the particular fault, because a reduced amount of possibly wrong instructions is given to him.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Kapitel 1

Einleitung

1.1 Motivation und Problemstellung

Die jährlichen durch inadäquate Softwaretest-Infrastruktur entstandenen Kosten in den Vereinigten Staaten von Amerika wurden 2002 von Tassey (2002) auf 22.2-59.5 Mrd. US-Dollar geschätzt. Über die Hälfte dieser Kosten wurden von Endbenutzern durch Fehlervermeidung und Aktivitäten zur Linderung der Auswirkungen verursacht. Die restlichen Kosten wurden von Softwareentwicklern verursacht und spiegeln die zusätzlichen Testressourcen, die durch inadäquate Testwerkzeuge und -methoden begründet sind, wider.
Für Software-Entwickler ist insbesondere das Finden von Fehlern ein zeitaufwändiger Prozess. Zhang u. a. (2006) schreiben dazu, dieser Prozess des Untersuchens sei oft mühsam und zeitraubend.
Insbesondere bei komplexen Systemen, fremden Quellcodes oder Nebeneffekt-behafteten Programmen kann die Fehlersuche sehr viel Zeit in Anspruch nehmen.
Aufgrund der Komplexität von Debugging und Fehlersuche soll mit dieser Arbeit ein Ansatz erarbeitet werden, diese Komplexität zu reduzieren.

Trotz des relativ viel erforschten Bereichs, von Failure Detection und Fault Localization, ist derzeit kein auf diesen Technologien basierter Ansatz für Java-basierte Programme im praktischen Einsatz. Eine Unterstützung des Entwicklers bzw. Softwaretesters durch ein entsprechendes Tool hat insbesondere für die weit verbreitete Programmiersprache Java[1] enormes Einsparpotenzial.

[1] Laut TIOBE (2012) ist Java mit 16,599% Marktanteil die zweithäufigste verwendete Programmiersprache.

1.2 Ziele

Die primäre Zielsetzung dieser Arbeit ist die Unterstützung von Softwareentwicklern und Softwaretestern indem die Komplexität und dadurch ebenso die Kosten von Softwarefehlern reduziert wird. Die Reduktion der Komplexität soll im Bereich des Debuggings bzw. Fehlerfindens von Java-Programmen erreicht werden.
Das konkrete Ziel dieser Arbeit ist die Erstellung eines Konzeptes zur Fehlerlokalisierung und Fehlereingrenzung. Dazu sollen verschiedene Möglichkeiten dieser Techniken vorgestellt und ein geeigneter Ansatz für Javaprogramme in Client-Server-Architekturen erarbeitet werden. Weiterhin ist die praktische Untersuchung des Verfahrens nötig, um den gewählten Ansatz zu verifizieren.

1.3 Abgrenzung

Diese Arbeit versucht herauszufinden, ob die Reduktion der Komplexität bei der Fehlersuche mit Hilfe des entwickelten Ansatzes möglich ist. Dabei wird kein Anspruch auf eine vollständige, in der Praxis einsetzbare Lösung erhoben. Diese Arbeit soll eher eine Machbarkeitsstudie als eine komplette Problemlösung sein. Die konkrete Implementierung und insbesondere die verwendeten Tools und Technologien sind dabei nur Mittel zum Zwecke der Demonstration.

1.4 Gliederung

Im Folgenden soll der Aufbau der Arbeit kurz erläutert werden. Nachdem in Kapitel 1 eine Einleitung mit Motivation und Problemstellung, Zielen, Abgrenzung und Gliederung in die Thematik einführt, werden in Kapitel 2 die fachlichen und technischen **Grundlagen** erarbeitet.
In Kapitel 2.1 wird das Anwendungsfeld dieser Arbeit, die Eigenschaften von Client-Server-Architekturen, die Netzwerkprotokollebene, die programmbezogene Ebene und das verwendete Beispiel der CoffeeMaker, beschrieben. Die Grundlagen zu dem Thema Fault Localization werden in Kapitel 2.2 erläutert. Dazu findet vorerst eine Begriffsklärung der verwendeten Terminologie statt. Weiterhin werden in Kapitel 2.2.4 mögliche Fault Containment und Propagation Typen in dem Anwendungsfeld identifiziert. Das Kapitel 2.2.5 kategorisiert die für die Fehlerlokalisierung und Fehlereingrenzung relevanten Failures. Im folgenden Kapitel 2.2.6 werden erforschte Fault Localization Methoden erläutert. Abschließend sind verwendete Tools und Technologien in Kapitel 2.3 beschrieben.

Kapitel 3 **Related Work** gibt einen Überblick über ähnliche wissenschaftliche Arbeiten aus dem Forschungsumfeld von Fault Localization.

Das eigentliche **Konzept** wird in Kapitel 4 erarbeitet. Dazu werden vorerst in Unterkapitel 4.1 relevante Fehlertypen für die Fehlerlokalisierung bzw. Fehlereingrenzung ausgewählt. Daraufhin wird in Kapitel 4.2 auf das Verfahren der Erstellung geeigneter Testfälle eingegangen, in Kapitel 4.3 die Art und Weise der Ausführung der Tests beschrieben und in Kapitel 4.4 deren Auswertung erläutert.

In Kapitel 5 findet die **Demonstration** des zuvor erarbeiteten Konzeptes anhand der in Kapitel 5.1 identifizierten Faulttypen statt. Die Ausführung und Auswertung der jeweiligen konkreten Fault Localization findet in den Kapiteln 5.2 bis 5.4 statt.

Der **Schluss** fasst die Erkenntnisse der Arbeit in Kapitel 6.1 zusammen, gibt einen Überblick über mögliche weiterführende Arbeiten und wertet den erarbeiteten Ansatz in Kapitel 6.3 aus.

Kapitel 2

Grundlagen

Im Folgenden wird das Anwendungsfeld dieser Arbeit definiert und erläutert, auf Theorien und Methoden von Fault Localization eingegangen und verwendete Tools und Technologien beschrieben.

2.1 Anwendungsfeld

Diese Arbeit fokussiert die Lokalisierung und Eingrenzung von Fehlern Java-basierter Programme. Aus den in 1.1 genannten Gründen soll in dieser Arbeit insbesondere das Verhalten von Programmen in Client-Server Architekturen untersucht werden.

2.1.1 Client-Server-Architekturen

Client-Server-Architekturen zeichnen sich nach Svobodova (1985) durch die klare hierarchische und im Folgenden beschriebene Rollenverteilung der beiden wesentlichen Komponenten, dem Client als Dienstkonsument und dem Server als Dienstanbieter, aus.

Der Server stellt einen definierten Dienst zur Verfügung. Ein Dienst ist als die Lösung einer vordefinierten Aufgabe zu verstehen.

Der Client erfragt diesen Dienst und erwartet eine korrekte Antwort und Lösung der definierten Aufgabe. Die Vorbedingungen hierfür sind die Einigkeit über die Korrektheit der Aufgabenlösung bzw. das Bereitstellen des korrekten Dienstes des Servers und das Wissen des Clients über den Standort bzw. der Erreichbarkeit des Servers.

2.1.1.1 Charakterisierung

Softwaresysteme, die nach dem Client-Server-Modell entworfen wurden, besitzen Eigenschaften, die für die Identifizierung und für das Eingrenzen der Fehlerfallerkennung bedeutend sind. Diese sollen im Folgenden zusammengefasst werden.

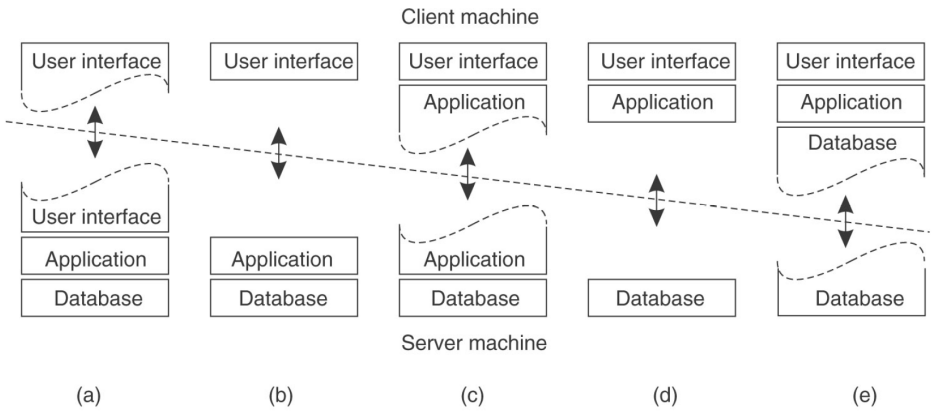

Abbildung 2.1: Alternative Client-Server-Anordnungen aus (Tanenbaum und Steen, 2008, S.60)

- Schnittstellen

  Die korrekte Beschreibung der aufzurufenden Schnittstelle des Servers ist eine elementare Aufgabe in Client-Server-Architekturen. Nur hierüber kann ein Vertrag zwischen Client und Server bezogen auf einen Dienst hergestellt werden und ein korrektes Verhalten sichergestellt werden. Zu der Schnittstellendefinition gehört der Name der Schnittstelle, der Methodenname, die nötigen und optionalen Methodenparameter sowie der Rückgabewert. Für Javaprogramme sind, durch die Eigenschaft der statischen Typisierung von Java, zusätzlich die Typen der Methodenparameter und der Typ des Rückgabewertes erforderlich.

- Verteilung

  Client und Server sind in der Regel physikalisch voneinander entfernt, d.h. sie befinden sich in anderen Netzwerken, auf anderen Computern oder zumindest in anderen Prozessen. Um eine Kommunikation zu gewährleisten, ist es nötig, dass der Client den Ort bzw. die Adresse des Servers kennt.

- Typ des Clients

  Client-Server-Architekturen lassen sich durch die Verteilung der Algorithmen auf Client und Server charakterisieren. Clients, bei denen der größte Teil der Algorithmen auf dem Server implementiert ist und kaum Logik im Client enthalten ist, werden Thin Clients genannt. Clients, die große Anteile der Algorithmen enthalten, werden Fat Clients genannt. Die Grenzen zwischen diesen Kategorien sind unscharf, und der Versuch eine fein-granulare Auflösung zu erarbeiten soll hier nicht stattfinden. Ein Überblick ist in Abbildung 2.1 dargestellt.

- Multi-Tier

  Client-Server-Architekturen enthalten oft Komponenten, die gleichzeitig Server für einen Dienst und Client eines anderen Dienstes sind. Solche mehrschichtigen Client-Server-Modelle werden Multi-Tier-Client-Server-Architekturen genannt. In mehrschichtigen/mehrstufigen Architekturen treten einzelne Subsysteme sowohl in der Rolle des Clients als auch in der Rolle des Servers auf.

2.1.1.2 Besonderheiten und Abgrenzung

Aus der in Kapitel 1.1 beschriebenen Motivation ergibt sich der Schwerpunkt dieser Arbeit, der auf Java-basierte E-Commerce Anwendungen mit Client-Server-Architekturen liegt. Weiterhin soll hier eine zusätzliche Eingrenzung des Zielsystems für die Fehlerlokalisierung und Fehlereingrenzung geschehen.

Die Verteilung von Client-Server-Architekturen in B2C[2] E-Commerce Anwendungen findet in der Regel Netzwerk-übergreifend im Internet statt. Um in Kapitel 2.2.5 auch die Fehlerzustände, die durch die Netzwerkschicht verursacht werden können, erfassen zu können, sollen in Kapitel 2.1.2 die notwendigen Charakteristika von Netzwerkprotokollen erfasst werden.

E-Commerce Anwendungen für B2C werden heutzutage zum großen Teil für Standard-Webbrowser entwickelt. Dadurch wird eine hohe Erreichbarkeit möglich gemacht, da nahezu jedes Endkunden-Betriebssystem einen Standard-Webbrowser enthält und auf mobilen Endgeräten Webbrowser vorinstalliert sind. Webbrowser sind, so lange sie nicht durch zusätzliche Technologien wie AJAX[3], Javascript- [4], Java-Applet-[5] oder Flashkomponenten[6] angereichert werden, Thin-Clients. Durch Erweiterungen mit Hilfe entsprechender Technologien können Webbrowser zu Fat-Clients werden. In dieser Arbeit soll der Webbrowser jedoch als Thin-Client betrachtet werden, da das Entdecken von Fehlerursachen in speziellen Webbrowser-Technologien einer anderen Zielrichtung entspräche. Außerdem ist hier die Möglichkeit einer isolierten Betrachtung verwendeter Frontendtechnologien zu empfehlen, indem eine Validierung der über das Netzwerk übertragenen Daten stattfindet.
Die Betrachtung von Multi-Tier-Client-Server-Architekturen führt zu einer Erhöhung der Komplexität und würde auf Grund der mehrfachen Rollenbelegung der einzelnen Subsysteme zu unverständlicheren Systemen führen ohne einen nennenswerten Vorteil zu schaffen.

[2] Business-to-Consumer

[3] AJAX = Asynchronous Javascript and XML, ein Akronym zur Bezeichnung von dynamischen Web2.0 Web-Applikationen (s.a. Garrett (2005))

[4] Eine aus ECMA (2011) Standard 262 hervorgegangene Programmiersprache

[5] http://java.sun.com/applets/

[6] http://www.adobe.com/de/flashplatform/

Diese Arbeit soll sich somit auf einfache Client-Server-Architekturen beschränken.

2.1.2 Netzwerkprotokollebene

„Das Standardtransferprotokoll im Web ist HTTP [7].“

(Tanenbaum, 2011, S. 706)

Weitergehend erwähnt Tanenbaum, dass die Verwendung von TCP [8] als Transportverbindung üblich sei, vom HTTP-Standard jedoch nicht formell gefordert ist. Dem Anwendungsfeld dieser Arbeit liegt ebenfalls das HTTP auf Ebene der Verarbeitungsschicht mit darunter liegendem TCP in der Transportschicht zu Grunde.

2.1.2.1 TCP

Das TCP wird in vier, dem OSI-Modell ähnlichen, Schichten aufgeteilt. Die Schichten sind Anwendungsschicht, Transportschicht, Internetschicht und Host-an-Netz-Schicht[9]. Auf der TCP-Protokollebene können diverse Fehlerzustände auftreten, welche dazu führen, dass Maschinen im Netzwerk nicht erreicht werden können. Das TCP-Protokoll enthält eine transparente Fehlerkorrektur, die eine korrekte Übertragung der einzelnen Pakete garantiert. Dadurch können Fehler auf Protokollebene, die durch fehlerhafte Datenübertragung entstanden sind, ausgeschlossen werden. Da sich die Fehlerkorrektur in der Transportschicht befindet, sind auch in den Schichten „Transport“, „Internet“ und „Host-an-Netz“ entstandene Fehler abgedeckt.

Weiterhin können jedoch Fehlerzustände entstehen, sollte gar keine Verbindung aufgebaut werden können. Diese Fehler entstehen in den Schichten unter der Anwendungsschicht und führen zu Fehlerzuständen in dieser. Fehler, die entstehen, weil eine Verbindung z.B. wegen nicht vorhandener DNS-Einträge, falscher Netzwerkadressen, falscher URLs oder defekten Nezwerkkomponenten nicht zustande kommt, sind für den Benutzer konkret sichtbar.

2.1.2.2 HTTP

Das HyperText Transfer Protocol ist nach Tanenbaum (2011) ein zur Übertragung von Daten im Web entwickeltes Protokoll. Es ist im RFC2616 von Fielding u. a. (1999) beschrieben. Es wird vornehmlich dazu verwendet, um Webseiten aus dem Internet in einen Browser zu

[7] HyperText Transfer Protocol

[8] Transmission Control Protocol

[9] Übersetzung nach (Tanenbaum, 2011, S. 60). Originalnamen aus Leiner u. a. (1985): Applications, Service Protocols, Internet and Networks

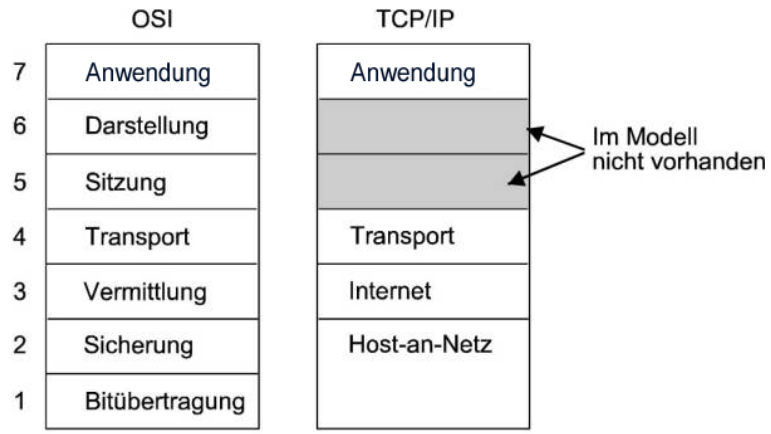

Abbildung 2.2: Das TCP/IP-Referenzmodell aus Tanenbaum (2011)

laden. Die Verwendung von HTTP ist weit verbreitet. Viele Anwendungen und Protokolle machen sich das zustandslose, eindeutig definierte und einfache Protokoll zu nutze, einige Beispiele aus der Praxis sind SOAP[10]-Webservices, REST[11]ful Webservices[12] oder das für Dateiübertragungen entwickelte WebDAV[13].

Von Fielding u. a. (1999) wurden die Methoden, die für das Protokoll implementiert sein müssen, definiert. Diese sind OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE und CONNECT. Im Kontext von E-Commerce Anwendungen über Standardbrowser werden in der Regel nur die Methoden GET und POST verwendet. Dabei dient GET dem Anfragen einer durch die URL definierten Ressource und POST ermöglicht das Anhängen von Daten an den gestellten Request.
Sämtliche Statuscodes des HTTP werden vom Server generiert, d.h. etwaige Fehler in unterliegenden Protokollen (wie TCP) werden nicht durch die Statuscodes abgebildet. Sollte eine Verbindung zustande gekommen sein und der Server beantwortet diese, wird in der im HTTP Header enthaltenen „Status Line“ (vgl. (Fielding u. a., 1999, S.39) ein Statuscode (maschinenlesbar) mit einer Reason Phrase (menschenlesbar) zurückgegeben. Die Statuscodes sind wie folgt klassifiziert:

- 1xx: Informational
- 2xx: Success

[10]Simple Object Access Protocol
[11]Representational State Transfer
[12]RESTful Webservices - Ein Programmierparadigma für zustandslose Webservices
[13]Web-based Distributed Authoring and Versioning - Ein offener Standard zur Übertragung von Dateien im Internet.

- 3xx: Redirection
- 4xx: Client Error
- 5xx: Server Error

Aus dieser Liste sind die beiden Statuscode-Klassen des HTTP „4xx Client-Error“ und „5xx Server-Error“ für diese Arbeit relevant, da sie direkte Fehlerzustände der Anwendung repräsentieren. Die weiteren Zustandscodes (vgl. Fielding u. a. (1999)) werden hier nicht näher erläutert, weil sie keine konkreten Fehlerzustände der Anwendung sondern des Netzwerkes oder der Konfiguration des Webservers repräsentieren. Der Vollständigkeit halber ist die Liste der HTTP-Statuscodes im Anhang A.1 ersichtlich. Abschließend sollen die drei für diese Arbeit relevanten Fehlercodes erläutert werden.
Der wohl bekannteste Client Error HTTP Statuscode ist 404 mit der Reason Phrase „Not Found“, welcher dem Client mitteilen soll, dass der Server die angefragte Ressource nicht finden konnte.
Weiterhin gehört der Fehler „500: Internal Server Error“ zu den bekanntesten im E-Commerce Bereich, da dieser angezeigt wird sobald ein Server-seitiger Softwarefehler aufgetreten ist, durch welchen das vollständige Verarbeiten der Anfrage unmöglich geworden ist.
Der Fehler „504: Gateway Timeout“ stellt ebenfalls einen häufig vorkommenden Fehler in Server-Applikationen dar. Er tritt auf, wenn ein als Proxy oder Gateway konfigurierter Webserver funktionsfähig ist, ein weiteres System z.B. der Applikationsserver jedoch nicht innerhalb der definierten Timeout-Zeiten antwortet.

2.1.3 Programmbezogene Ebene

In diesem Kapitel soll das Anwendungsfeld mit Blick auf die verwendete Programmiersprache Java betrachtet werden. Bevor auf die konkrete Ausnahmebehandlung der Programmiersprache Java (Exception Handling) eingegangen wird, sollen generelle Typen von Fehlern erläutert werden.
Oft verwendete Typen von Softwarefehlern sind Lexikalische Fehler, Syntaktische Fehler, Kontext-Semantische Fehler und Semantische Fehler.

Lexikalische Fehler sind Fehler, die entstehen, weil ein Rechtschreibfehler begannen wurde. Bspw. wurde ’wihle’ statt ’while’ getippt.

Syntaktische Fehler sind Fehler, bei denen die einzelnen Befehle zwar korrekt sind, in der gewählten Anordnung jedoch nicht der Sprachsyntax entsprechen.

Kontext-Semantische treten auf, wenn der Kontext nicht beachtet wurde oder sich verändert hat und es aufgrund dessen zu einem Semantischen Fehler kommt. Bspw. kann das dynamische Austauschen von Bibliotheken zur Laufzeit zu Kontext-Semantischen Fehlern führen, wenn die neu geladene Bibliothek eine andere Semantik erfordert.

Semantische Fehler bedeuten Fehler in dem vorliegenden Problemlösungsansatz oder in dem Algorithmus. Dies kann durch missverstandene Anforderungen oder durch Fehler beim Programmieren entstehen. Semantische Fehler sind Laufzeitfehler.
Lexikalische und Syntaktische Fehler sind für diese Arbeit nicht von Bedeutung, da diese Fehler bereits von den üblichen Entwicklungsumgebungen vor dem Kompilieren und spätestens durch das Ausführen des Compilers gefunden werden.
Kontext-Semantische Fehler sollen, auf Grund der deutlich erhöhten Komplexität von Systemen bei denen dynamisch Bibliotheken nachgeladen bzw. ausgetauscht werden, ebenfalls nicht relevant sein.
Im nächsten Kapitel wird auf das sog. Exception Handling, das Behandeln von Ausnahmen in der Programmiersprache Java eingegangen. Das ist insbesondere bei Java wichtig, da die Integration von Ausnahmen in die Syntax der Programmiersprache erhebliche Auswirkungen auf die Fehlereingrenzung und -lokalisierung hat.

2.1.3.1 Java Exception Handling

Die Programmiersprache Java[14] bietet im Gegensatz zu vielen anderen Sprachen (bspw. C) ein bereits in die Sprache integriertes Konzept, eine Objekthierarchie und eine dazugehörige Syntax zur Ausnahmebehandlung (Exception Handling). Sämtliche Ausnahmen (Exceptions) müssen mindestens indirekt von der Throwable-Klasse erben. Außerdem muss zwischen dem Auslösen einer Ausnahme und dem Behandeln differenziert werden.

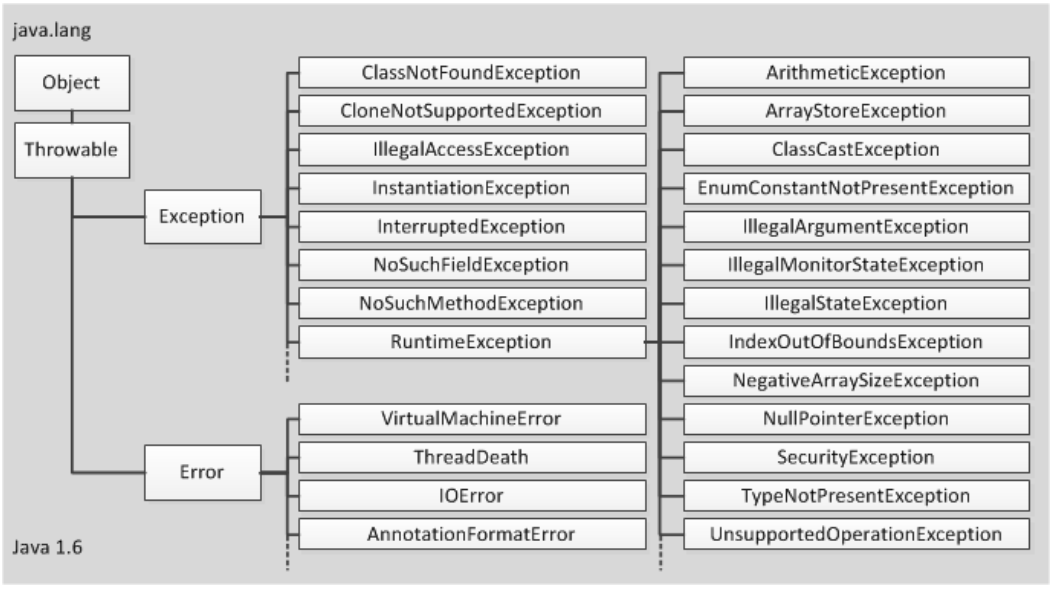

Abbildung 2.3: Auszug aus der Java Exception Hierarchie

[14]Die der Arbeit zu Grunde liegende Java Version ist 1.6.x.

Weiterhin ist zwischen Exceptions und Errors zu unterscheiden. Java Exceptions weisen auf ein nicht erwartetes Verhalten des Programms hin und dienen dem kontrollierten Umgang mit diesen Fehlerzuständen. Java Errors werden als nicht behandelbar angesehen und führen bspw. zu dem vollständigen Absturz der JVM[15]. Java Errors werden im Folgenden nicht weiter betrachtet, da der Programmierer in der Regel keinen Einfluss auf das Auftreten und keine Möglichkeit der Behandlung hat.

Eine Ausnahme kann entweder von der JVM oder durch den throw[16] Programmbefehl ausgelöst werden. Dabei kann weiterhin zwischen dem expliziten und dem impliziten Werfen einer Ausnahme unterschieden werden. Das explizite Werfen geschieht direkt im Programmcode mittels der bereits erwähnten throws-Klausel (s.a. Listing 2.1 Zeile 3). Das implizite Werfen von Ausnahmen kann bspw. durch eine Division durch Null geschehen. Im zweiten Fall wird die entsprechende Ausnahme von der JVM geworfen und erreicht den Programmablauf an entsprechender Stelle des fehlerhaften Divisionausdruckes.

Listing 2.1: Beispielcode Java throws-Klausel

```
1  public void startEngine(Engine engine) throws IllegalArgumentException{
2    if(null == engine){
3      throw new IllegalArgumentException("Engine darf nicht null sein.");
4    }
5    engine.start();
6  }
```

Das Behandeln einer Ausnahme geschieht mittels eines sog. try-catch-Blocks (s.a. Listing 2.2). Ein try-catch-Block besteht aus einem try-Block (Zeilen 1-3) und mindestens einer catch-Anweisung (Zeile 4). Der try-Block umgibt den potenziell Fehler-produzierenden Quellcode. Sollte innerhalb des try-Blockes eine Exception auftreten wird der reguläre Programmfluss unterbrochen und die Exception wird an die catch-Klausel gegeben. Die catch-Klausel ermöglicht es, die aufgetretene Exception zu behandeln. Dazu muss der in der catch-Klausel definierte Exception-Typ der geworfenen Exception entsprechen, mindestens jedoch in derselben Vererbungshierarchie liegen. Eine weitere Möglichkeit mit einer Exception umzugehen ist das Propagieren der Ausnahme an die aufrufende Methode. Dies geschieht mittels eines zusätzlichen Methodensignatur-Ausdrucks, welcher das Behandeln der Exception durch die aufrufende Klasse erlaubt (s.a. Listing 2.1 Zeile 1).

Listing 2.2: Beispielcode Java try-catch-Block

```
1  try {
2    Engine engine = null; // Fehler produzierender Quellcode
3    startEngine(engine);  // throws IllegalArgumentException
4    ...
5  } catch (IllegalArgumentException exception){
```

[15] Java Virtual Machine = Für die Ausführung von Bytecode verantwortliche Teil der Java-Laufzeitumgebung

[16] engl. throw = werfen. Beim Auftreten einer Ausnahme wird auch von dem Werfen einer Ausnahme gesprochen.

```
6     //Behandeln der Exception
7     ...
8     System.out.err("Es ist ein Fehler aufgetreten" + exception);
9   }
```

Es gibt Typen von Exceptions in Java, sogenannte checked Exceptions, die ein entsprechendes Behandeln der Ausnahme durch den Programmierer mittels try-catch-Block oder throws-Klausel bereits zur Compilezeit erzwingt. Checked Exceptions sind Unterklassen von Exception außer RuntimeException. Unchecked Exceptions, Ausnahmen die nicht zur Compilezeit beachtet werden können, sind alle Exceptions, die von Error und RuntimeException erben.

2.1.4 Beispiel: Coffee Maker

Der CoffeeMaker ist ein kleines Java-Programm des Software Engineering Departments der *North Carolina State University*. Die Anwendung besteht aus 4 Klassen und einer Main-Klasse, um das Programm zu starten. Weiterhin wurden zwei Exceptions (RecipeException und InventoryException) definiert. Diese erben direkt vom Supertyp Exception und sind somit als checked Exceptions zu betrachten. Der CoffeeMaker ist eine Single-Thread Javaanwendung.

2.2 Fault Localization Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen für das Eingrenzen und Lokalisieren von Fehlern erläutert.

2.2.1 Terminologie und Theoretische Grundlagen

Zur weiteren Definition und Eingrenzung der zu behandelnden Problematik dieser Arbeit ist es nötig, die möglichen Fehler, Fehlerarten und Fehlerzustände zu beschreiben und abzugrenzen.

2.2.2 Debugging

Programmierer müssen oft feststellen, dass ein Programm nicht das geforderte Verhalten zeigt. Durch hohe Komplexität, der großen Anzahl von Variablen und Prozessschritten eines Programmes ist es oft nicht möglich das Fehlverhalten allein mit Hilfe einer statischen Codeanalyse oder Fehlersuche zu identifizieren.
Der Debuggingprozess lässt sich in zwei Phasen teilen. Erstens wird mittels Fault Localization versucht die Quelle des Fehlers, die Fehlerursache, zu identifizieren. In der zweiten

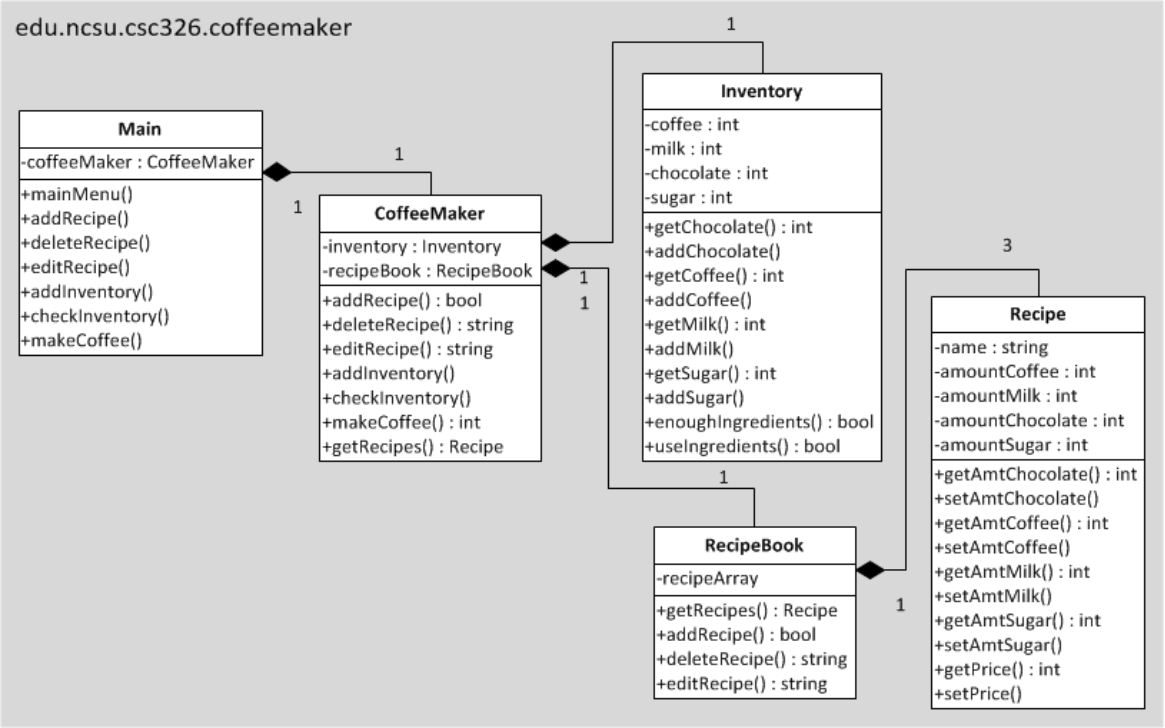

Abbildung 2.4: CoffeeMaker Klassendiagramm

Phase wird die Fehlerursache behoben, was nach Wong und Debroy (2009) Bugfixing oder Fault Fixing genannt wird.
Unterstützung bekommen Programmierer durch Werkzeuge, die ein Untersuchen des Programms zur Laufzeit ermöglicht. Der Prozess des Beobachtens von Programmen zu ihrer Laufzeit wird *Debbuging* genannt.
Ein Standard-Debbugingprozess besteht laut Zhang u. a. (2006) aus dem Setzen von Breakpoints, dem Wiederausführen des Programmes mit fehlerverursachenden Eingaben und dem Untersuchen des Programm-Zustandes (Variablenwerte, Call Stack, Prozessschritte und deren Reihenfolge, Zeitverhalten).

Bei dem Beobachten des dynamischen Verhaltens von fehlerhaften Programmen vergleicht der Programmierer stetig den Ist-Zustand mit dem erwarteten Soll-Zustand, um die fehlerhaften Zustandsübergänge identifizieren zu können (vgl. Liang und Xu (2005)).

2.2.3 Errors, Failures, Faults and Mistakes

Für das Verständnis und die Verdeutlichung des weiteren Vorgehens sollen im Folgenden die Begriffe Error, Failure, Fault und Mistake geklärt werden. Diese sind in dem Standard IEEE-24765 (2010) dem offiziellen Nachfolger der IEEE-610.12 (1990) definiert.

2.2.3.1 Errors

> „ (1)Eine menschliche Handlung, die ein falsches Ergebnis produziert, wie ein Softwarefehler.
> (2)Ein falscher Schritt, Prozess oder Datendefinition.
> (3)Ein falsches Ergebnis.
> (4)Ein Error (oder Fehler) ist der Unterschied zwischen einem berechneten, beobachteten oder gemessenen Wert oder einer Gegebenheit und dem echten, spezifizierten oder theoretisch korrekten Wert oder einer Gegebenheit.“
>
> (IEEE-24765, 2010, S.128)

Ein Error wird im IEEE Systems and software engineering - Vocabulary (IEEE-24765, 2010, S.128) durch die in den folgenden drei Kapiteln beschriebenen Begriffe Failures, Faults und Mistake definiert und gilt somit als deren Oberbegriff.

2.2.3.2 Failures

Ein Software-Fehlverhalten kann nur entdeckt werden, wenn eine Anforderung an ein System gestellt wurde und das aktuelle Verhalten nicht dem in den Anforderungen spezifizierten Verhalten entspricht.
Ein solches nicht erwartetes Verhalten eines Softwaresystems wird im Weiteren Failure oder Fehlerwirkung genannt.

2.2.3.3 Faults

Jede Fehlerwirkung hat auch eine Fehlerursache. Diese wird nach IEEE Fault genannt. Nach Spillner u. a. (2006) kann hier auch von einem Bug, einem Internen Fehler oder einem Defect gesprochen werden.
Fehlerursachen können beispielsweise durch falsche Programmierung oder vergessenen Programmcode entstehen.
Diese Arbeit konzentriert sich auf Softwarefehler. Darum wird die Betrachtung von Hardwarefehlern hier außer Acht gelassen.

Ein Fault kann nach IEEE-24765 (2010) ein inkorrekter Prozess, Instruktion oder Datendefinition sein.

2.2.3.4 Mistakes

Durch menschliche Handlungen (oder menschliches Versagen) verursachte Fehler werden nach IEEE 24765 Mistakes genannt. Mistakes können zu Faults führen.

2.2.4 Fault Containment und Fault Propagation

Bei der Betrachtung eines Softwarefehlers in Hinblick auf das Eingrenzen bzw. Lokalisieren der Fehlerquelle ist es wichtig das Entstehungsumfeld und die mögliche Ausbreitung des Fehlers zu untersuchen. In diesem Kapitel wird somit die Kapselung und Verbreitung von Laufzeitfehlern in Java-basierten Programmen in Client-Server-Architekturen erläutert. Weiterhin ist eine initiale Identifikation der Software-Systemkomponenten zu leisten, sowie deren Anordnung in der Softwarearchitektur. Auf Basis dieser Erkenntnisse wird anschließend analysiert, wie sich Fehler in entsprechender Architektur verbreiten und welche Komponenten betroffen sind. Das Ziel dieser Vorgehensweise ist die Verknüpfung der Fault Propagation Typen mit den Failure Modes aus Kapitel 2.2.5.2 in Kapitel 2.2.5.3.
Diese Arbeit bezieht sich, wie in 1.1 beschrieben, auf Java-Programme in Client-Server-Architekturen. Client-Server-Architekturen haben, wie in 2.1.1 erläutert, immer mindestens zwei Komponenten: Den Client und den Server. In unserem Anwendungsfeld ist der Client ein einfacher Webbrowser und auf Seiten des Servers kommt der CoffeeMaker zum Einsatz. Der CoffeeMaker wiederum ist ein Java-Programm, welches wie üblich eine JVM-Laufzeitumgebung benötigt. Beide Systeme basieren jeweils auf einem Betriebssystem (OS). Die Kommunikation zwischen Client und Server findet per HTTP over TCP statt.

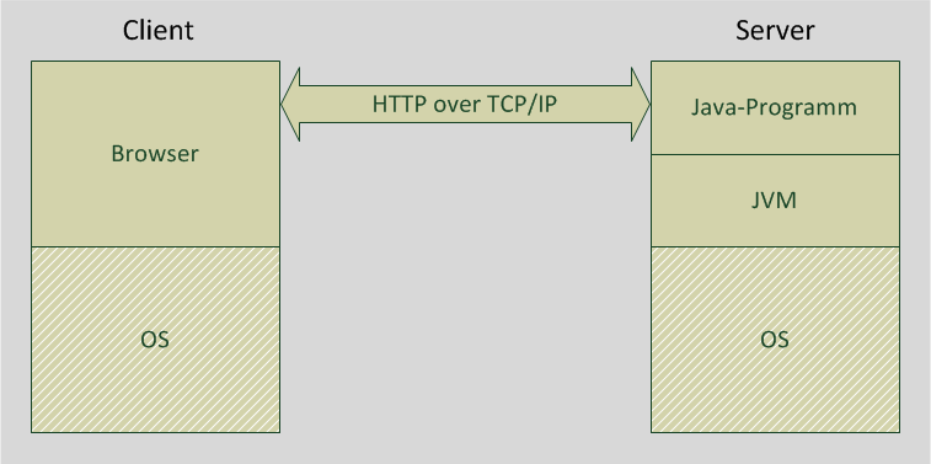

Abbildung 2.5: Die System-Architektur des CoffeeMaker

Zu der weiteren Betrachtung der Fehlerursachen und Fehlerpropagierung bleibt zu erläutern, dass die Analyse der auftretenden Fehler nur unidirektional geschieht, d.h. es werden keine Fehlerursachen erläutert, die zuerst in die eine Richtung und daraufhin entgegengesetzt propagiert werden[17]. Auf diese deutlich aufwändigere Sichtweise wird hier zu Guns-

[17] Beispielsweise kann ein im Java-Programm verursachter Fehler zu Auswirkungen im Betriebssystem führen, welche wiederum an den Benutzer propagiert werden würden

ten der Übersicht verzichtet. Außerdem werden keine weiteren Erkenntnisse hinsichtlich der Fehlerpropagierung erwartet.

Fehler können in dem jeweiligen Betriebssystem, in der JVM, im CoffeeMaker und im Browser auftreten. Außerdem können Fehler auf Netzwerkebene entstehen, sobald eine Verbindung nicht aufgebaut werden kann oder zusammenbricht.

Die im Folgenden beschriebenen Fehlerausbreitungen sind in Abbildung 2.6 dargestellt.

Der direkte Absturz eines Betriebssystems ist ein Fehler, der für diese Arbeit nicht weiter relevant sein soll. Komplette Systemabstürze haben keinen Bezug zu Fehlerlokalisierung in Java-Programmen. Sollte eine andere Fehlerquelle im **Betriebssystem** vorliegen, sind folgende Ausbreitungen möglich:
A) *Client-seitiger Betriebssystem-Fehler mit Auswirkungen im Browser:* Ein Fehler breitet sich vom Client-seitigen Betriebssystem zum Browser aus.
B) *Server-seitiger Betriebssystem-Fehler mit Auswirkungen in der JVM:* Ein im Server-seitigen Betriebssystem entstandener Fehler verursacht Fehlverhalten in der JVM, welcher die JVM zum Absturz/Stoppen bringt.
C) *Server-seitiger Betriebssystem-Fehler mit Auswirkungen im Java-Programm:* Ein im Server-seitigen Betriebssystem beherbergter Fehler breitet sich über die JVM bis hin zum Java-Programm aus und verursacht dort ein Fehlverhalten, welches weitere Verarbeitung im Java-Programm unmöglich macht (bspw. Absturz).
D) *Server-seitiger Betriebssystem-Fehler mit Auswirkungen im Browser:* Ein Server-seitiger Betriebssystem-Fehler breitet sich durch die JVM bis zum Java-Programm aus. Das Java-Programm gibt den Fehler bis zu dem Browser weiter.

Ist der Fehler in der **JVM** enthalten, sind diese Propagierungen des Fehlers denkbar:
E) *JVM-Fehler mit Auswirkungen im Java-Programm:* Eine in der JVM ansässige Fehlerquelle verursacht einen Fehler, der das Java-Programm unfähig für weitere Operationen macht (bspw. Absturz, Endlosschleife).
F) *JVM-Fehler mit Auswirkungen im Browser:* Ein in der JVM enthaltener Fehler verursacht eine Fehlerwirkung im Java-Programm, dieses gibt den Fehler an den Browser weiter.

Neben einem Absturz des **Java-Programms** kann es Fehler enthalten, die folgendermaßen propagiert werden würden:
G) *Java-Programmfehler mit Auswirkungen im Browser:* Ein Fehler im Java-Programm wird im Browser sichtbar.

Auf **Netzwerkprotokollebene** sind ebenfalls Fehlerquellen möglich:
H) *Netzwerkprotokollfehler mit Auswirkungen im Browser:* Ein Verbindungsabbruch bzw. ein Nichtzustandekommen ist die Fehlerquelle für den im Browser angezeigten Fehler.

Neben einem direkten Absturz kann der **Browser** Fehler enthalten:
I) *Browserfehler:* Der Browser kann einen Fehler in der Frontendtechnologie (bspw. Javascript Endlosschleife) oder eine unendliche Weiterleitung (die Fehlerquelle für diesen Fall könnte theoretisch jedoch auch in jeglicher Komponente des Servers liegen, wenn eine falsche Weiterleitung an den Browser gegeben wird) enthalten.

Der Browser kann einen Fehler in der Frontendtechnologie enthalten, wie bspw. eine Javascript Endlosschleife. Außerdem kann der Browser eine unendliche Weiterleitung enthalten, die Fehlerquelle hierfür könnte theoretisch jedoch auch in einer Serverkomponente liegen, wenn eine falsche Weiterleitung an den Browser gegeben wird.

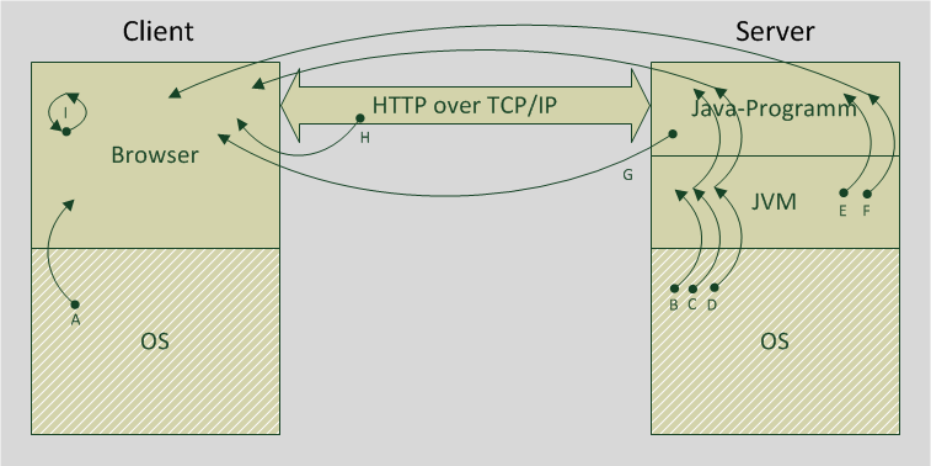

Abbildung 2.6: Die System-Architektur des CoffeeMaker mit möglichen Fehlerquellen und deren Ausbreitung

Zu Abbildung 2.6 ist anzumerken, dass durch die Änderung der Perspektive weitere Propagierungsmöglichkeiten bestehen. In dieser Arbeit wurde die Perspektive eines Browser-Benutzers gewählt. Fehler, die bspw. im Javaprogramm entstehen und auch nur dort Auswirkungen haben, werden entweder durch einen Fehler „500 - Internal Server Error“ gekapselt oder nicht von dem Benutzer wahrgenommen. Letzteres kann bspw. bei Konfigurationsfehlern ohne nennenswerte Auswirkungen auftreten.

2.2.5 Failure Modes

> „**failure mode**. The physical or functional manifestation of a failure. For example, a system in failure mode may be characterized by slow operation, incorrect outputs, or complete termination of execution.“
>
> (IEEE-24765, 2010, S.139)

2.2.5.1 Software Failure Modes in der Literatur

Ristord und Esmenjaud (2001) „General Failure Modes“ definieren generelle Failure Modes mit Blick auf die SPINLINE3[18] Software, die auf einem Betriebssystem verwendet wird und

[18] Ein „Instrumentation & Control System“ für die Steuerung verschiedener Atomkraftwerkkomponenten.

dabei entsprechende Fehlerzustände erreichen kann. Hier werden die Failure Modes aus Sicht eines Benutzers formuliert. Die Failure Modes reichen von dem Stoppen des gesamten Betriebssystems bis zum Anzeigen von augenscheinlich korrekten, aber eigentlich falschen, Ergebnissen.

Ristord und Esmenjaud (2001) general failure modes:

1. *the operating system stops*
2. *the program stops without clear message*
3. *the program stops with a clear message*
4. *the program runs, producing obviously wrong results*
5. *the program runs, producing apparently correct but in fact wrong results*

DeMillo u. a. (1997) erarbeiten eine Liste mit deutlichen Parallelen zu Ristord und Esmenjaud (2001):

1. *System failed* - Systemfehler wie bspw. Überlauf, Null-Division oder Speicher-Segmentierungsfehler.
2. *Nontermination* - Leistungskriterium einer Anwendung mit einer auf das Zeitverhalten bezogenen unteren und einer oberen Grenze. Die untere Grenze definiert ein gewünschtes bzw. erwartetes Zeitverhalten. Die obere Grenze definiert den maximalen Zeitrahmen, um ein Ergebnis zu erhalten.
3. *Unexpected output but in the legal output domain* - Das Ergebnis ist falsch, liegt aber in dem möglichen Wertebereich der Domäne.
4. *Unexpected and illegal output* - Das Ergebnis ist falsch und liegt außerhalb des erlaubten Wertebereichs.

Der Failure Mode *System failed* ist mit *the operating system stops* von Ristord und Esmenjaud (2001) gleichzusetzen. Weiterhin kann *Unexpected output but in the legal output domain* mit *the program runs, producing apparently correct but in fact wrong results* und *Unexpected and illegal output* mit *the program runs, producing obviously wrong results* verglichen werden.

Eine ebenfalls sinnvolle Liste von Failure Modes aus Lloyd und Lipow (1977) wird in Reifer (1979) verwendet. Sie ist deutlich von der zuvor genannten differenzierbar, da sie sich auf den Fehlerzustand als Programmfehler aus Programmierersicht bezieht.

1. *Computational* - Fehler, die durch fehlerhafte Berechnung auftreten.
2. *Logic* - Fehler, die durch falsche Programmsequenzen oder -abläufe entstehen.

3. *Data I/O* - Fehler, die durch falsche Ein- oder Ausgabe zu einer Komponente/Funktion hervorgerufen werden.

4. *Data Handling* - Fehler, die entstehen, weil eine Komponente die Daten falsch behandelt (bspw. fehlerhafte Bit-Operation).

5. *Interface* - Fehler, die durch eine fehlerhafte oder falsch beschriebene Schnittstelle bedingt sind.

6. *Data Definition* - Fehler, die durch fehlerbehaftete Daten entstehen, die aus falschen Datendefinitionen resultieren.

7. *Data Base* - Fehler, die durch falsche Daten entstehen.

8. *Other* - Jegliche weitere Fehler.

Eine weitere Sicht und Definition von Failure Modes leisten Lutz und Woodhouse (1999), dabei werden die Failure Modes in zwei Gruppen geteilt. Die erste Gruppe bezieht sich auf Daten:

1. *Verlorene Daten* (bspw. verloren gegangene Nachricht, Datenverlust aufgrund von Hardwareausfall)

2. *Inkorrekte Daten* (bspw. ungenaue Daten, verfälschte Daten)

3. *Inkorrektes Daten-Zeitverhalten* (bspw. veraltete Daten, Daten kommen zu früh an)

4. *Extra Daten* (bspw. Redundanz, Datenüberlauf).

Und die zweite Gruppe bezieht sich auf die Prozessschritte bzw. Ereignisse:

1. *Halt/Unerwartetes Prozessende* (bspw. eingefroren oder dead-Lock)

2. *Ausgelassenes Ereignis* (bspw. Erwartetes Ereignis findet nicht statt, Software läuft trotzdem weiter)

3. *Inkorrekte Logik* (bspw. ungenaue Vorbedingungen, Fallunterscheidungen)

4. *Zeitverhalten, Reihenfolge* (bspw. Ereignisse treten in falscher Reihenfolge auf; Ereignis tritt zu früh/zu spät auf).

2.2.5.2 Failure Modes für Java-Programme mit Client-Server-Architekturen

Das Erstellen von geeigneten Failure Modes ist stark von der Domäne abhängig, kann auf diversen Abstraktionsebenen und aus verschiedenen Perspektiven stattfinden.
Die Failure Modes aus Lloyd und Lipow (1977) sind für den Anwendungsfall dieser Arbeit geeignet, finden jedoch auf einer Programmiersprachen-nahen Abstraktionsebene statt, wodurch die Erfassung der Protokollebene von Client-Server-Architekturen nicht möglich ist.

Lutz und Woodhouse (1999) Failure Modes sind wegen der Schwerpunktlegung auf die harte Differenzierung zwischen Daten und Prozessschritten für das hier verwendete CoffeeMaker-Beispiel nicht geeignet. Eine solche harte Differenzierung ist bei der angestrebten Fehlerlokalisierung nicht nötig, weil ein Fehler nicht zwingend als Prozess- bzw. datenbezogen identifiziert werden muss, um den Entwickler auf die fehlerhafte Instruktion hinzuweisen.

Die Failure Modes nach Ristord und Esmenjaud (2001) sind aus der Sicht eines Benutzers formuliert, der die verschiedenen Fehlerzustände des Systems wahrnimmt. Die Perspektive des Nutzers kann auch in dem Anwendungsfeld dieser Arbeit verwendet werden, unterscheidet sich jedoch in der konkreten Ausprägung der Failure Modes durch die Eigenschaft des Verteiltseins der Client-Server-Architekturen, so dass jeder Fehlerzustand für Client und Server betrachtet werden muss.

Die Liste 2.2.5.1 genereller Failure Modes lassen sich auf den Anwendungsfall dieser Arbeit erweitern. Hierbei sollte jedoch nach Failure Modes, die bei einer Java-basierten Anwendung Client- sowie Server-seitig Auftreten können, gestaffelt werden.
Dazu soll zunächst eine Liste mit möglichen Failure Modes auf Basis der Failure Modes aus Ristord und Esmenjaud (2001) verfasst werden, um im folgenden Kapitel eine Zuordnung zu den verschiedenen Fault Propagation Typen zu erstellen.

1. *the operating system stops*

Das Abstürzen des kompletten Betriebssystems soll, wie bereits in Kapitel 2.2.4 erläutert, nicht weiter betrachtet werden.

2. *the program stops without a clear message*

Es ist ebenfalls möglich, dass die jeweiligen Prozesse stoppen und somit keine klare Fehlermeldung ausgeben können.

- the client stops without a clear message
- the server stops without a clear message

3. *the program stops with a clear message*

Der angenehmere Fall ist der, bei dem der jeweilige fehlerbehaftete Prozess eine klare Fehlermeldung ausgeben kann.

- the client stops with a clear message

- the server stops with a clear message

4. *the program runs, producing obviously wrong results* und 5. *the server process runs, producing apparently correct but in fact wrong results*
Diese beiden letzten Failure Modes brauchen in dem Anwendungsfall der vorliegenden Arbeit nur für den Server betrachtet werden, da der Client (Browser) die Ergebnisse des Servers nur empfängt und darstellt. Das Produzieren von Fehlern, die nicht in den technischen Schichten erkannt werden, ist Server-seitig zu betrachten, da solche Fehler nur dort entstehen können.

- the server process runs, producing obviously wrong results
- the server process runs, producing apparently correct but in fact wrong results.

2.2.5.3 Einteilung

In diesem Unterkapitel soll eine Einteilung der möglichen Fault Propagation Typen zu den zuvor erarbeiteten Failure Modes stattfinden. Das Ziel dieser Einteilung soll die Grundlage für die Identifikation relevanter Fehlerzustände in Kapitel 4.1 für die Demonstration der Fehlereingrenzung sein.

	Fault Containment/Propagation Type								
Failure Mode	A	B	C	D	E	F	G	H	I
Client stops without a clear message	X								
Server stops without a clear message		X	X		X				
Client stops with a clear message	X			X				X	X
Server stops with a clear message						X	X		
Server runs, producing obviously wrong results				X		X	X		
Server runs, producing apparently correct but in fact wrong results				X		X	X		

Tabelle 2.1: Zuordnung der Fault Propagation Typen zu Failure Modes

Dazu soll zunächst der Weg eines aufgetretenen Fehlers in Beziehung zu seinen Auswirkungen gesetzt werden. Als Ausgangspunkt werden hierzu die Fault Propagation Typen aus 2.2.4 verwendet und einzeln untersucht, um mögliche Verknüpfungen mit den generellen Failure Modes aus Kapitel 2.2.5.2 herzustellen.

A) *Client-seitiger Betriebssystem-Fehler mit Auswirkungen im Browser.*
Dieser Fault Propagation Typ kann viele Failure Modes auslösen. Ein Client-seitiger

Betriebssystem-Fehler kann dazu führen, dass der Browser ohne eine Fehlermeldung stoppt bzw. abstürzt (*Client stops without a clear message*) oder der Browser eine Fehlerauswirkung als Fehlermeldung ausgibt (*Client stops with a clear message*).
B) *Server-seitiger Betriebssystem-Fehler mit Auswirkungen in der JVM.*
Der Server-seitige Stopp/Absturz der JVM, verursacht durch einen Betriebssystem-Fehler, führt zu dem Failure Mode *Server stops without a clear message*.
C) *Server-seitiger Betriebssystem-Fehler mit Auswirkungen im Java-Programm.*
Ebenso wie im vorigen Fall verursacht der Absturz/das Stoppen des Java-Programms einen Failure Mode des Typs *Server stops without a clear message*.
D) *Server-seitiger Betriebssystem-Fehler mit Auswirkungen im Browser.*
Sollte im Server-seitigen Betriebssystem ein Fehler entstehen, welcher bis zum Browser propagiert würde, so könnten hierdurch die Failure Modes *Client stops with a clear message*, *Server runs, producing obviously wrong results* oder *Server runs, producing apparently correct but in fact wrong results* verursacht werden.
E) *JVM-Fehler mit Auswirkungen im Java-Programm.*
Der Absturz bzw. das Stoppen des Java-Programms führt zu dem Failure Mode *Server stops without a clear message*.
F) *JVM-Fehler mit Auswirkungen im Browser.*
Die Auswirkungen eines Fehlers der JVM, welcher bis zum Browser propagiert wird, prägt sich in den Failure Modes *Server stops with a clear message*, *Server runs, producing obviously wrong results* oder *Server runs, producing apparently correct but in fact wrong results* aus.
G) *Java-Programmfehler mit Auswirkungen im Browser.*
Die zuvor genannten Failure Modes gelten auch für diesen Fault Containment Typen.
H) *Netzwerkprotokollfehler mit Auswirkungen im Browser.*
Fehler im Netzwerkprotokoll zeigen sich durch den Failure Mode *Client stops with a clear message*.
I) *Browserfehler.*
Ebenso sind interne Browserfehler durch den Failure Mode *Client stops with a clear message* sichtbar.

2.2.6 Fault Localization Methode

Fault Localization ist ein Teilprozess des Debuggings, bei dem der Softwareentwickler versucht die Fehlerursache einer Fehlerwirkung zu finden (vgl. Wong und Debroy (2009)). Im Folgenden sollen mögliche Fault Localization Vorgehensweisen erläutert werden.

2.2.6.1 Intuitive Methoden

Eine viel verwendete Methode zum Fehlerfinden ist das Hinzufügen von Programmausgaben an den relevanten Codeteilen. Dabei wird versucht Programmausgaben zu erstellen, welche Hinweise auf den Zustand und mögliche inkorrekte Variablenwerte geben. Diese Ausgaben werden von dem Entwickler auf Basis von Vermutungen über fehlerbehaftete Anweisungen erstellt.

Diese Vorgehensweise ist sehr zeitaufwändig, da das Programm nach jeder zusätzlichen Ausgabe neu kompiliert, gestartet und mit den fehlerverursachenden Eingaben versorgt werden muss.

2.2.6.2 Just-In-Time Debugger

Ein Just-In-Time Debugger erlaubt dem Entwickler mit Hilfe sogenannter Breakpoints Anweisungen zu markieren. Der Just-In-Time Debugger führt daraufhin das Programm aus und hält vor der Ausführung der markierten Anweisung das Programm an. Der Entwickler hat somit zur Laufzeit die Möglichkeit sämtliche Variablen zu beobachten. Weiterhin wird dem Entwickler ermöglicht die weitere Ausführung des Programmes schrittweise zu kontrollieren.

Dadurch kann der Programmablauf sowie der Programmzustand zur Laufzeit beobachtet werden. Just-In-Time Debugger sind mächtige Werkzeuge, können jedoch nur dann helfen, wenn der Entwickler eine relativ konkrete Vorahnung von der Fehlerursache hat.

2.2.6.3 Program Slicing

Weiser hat in einer seiner Studien herausgefunden, dass erfahrene Programmierer nach dem Debuggen die relevanten Slices erinnern. Er ist der Meinung, dass die Programmierer ebenfalls 'Slicing' als Abstraktion nutzen und es aufgrund dessen eine nützliche Methode sein muss (vgl. Weiser (1981)).

Program Slicing ist ein durch Weiser (1979) vorgestelltes Verfahren, bei dem ein Programm in sogenannte *Slices* geteilt wird. Ein Slice ist hierbei eine minimale Teilmenge von Ausdrücken eines Programmes, die das selbe Verhalten produziert. Laut Tip (1995) enthält ein Program Slice die Teile eines Programmes, die eine Berechnung an einem Punkt von Interesse (*Slicing Criterion*) potentiell relevant sind. Ein Slicing Criterion besteht aus einer Menge von Werten für die eindeutige Identifikation des Kriteriums (bspw. ist ein Slicing Criterion C für statische Slices nach Weiser (1981) ein Tupel aus einem Programmknoten n und einer Menge von Variablen V). Daraus ergibt sich diese Formel für die Beschreibung eines Slicing Criterions: $C = \{n, V\}$. Ein Slice S ist nach Weiser (1981) eine Untermenge der Programmbefehle von Programm P für das folgende Eigenschaft erfüllt sein muss: Wenn P auf eine Eingabe wartet, muss S ebenfalls halten und die selben Werte für die Variablen V zum Zeitpunkt der Ausführung von n berechnen.

Weiser (1981) differenziert Program Slicing von anderen Methoden zur Reduzierung der Komplexität durch ihre Eigenschaft, erst nach dem Erstellen des Programms angewendet zu werden. Andere Dekompositionen (wie Information Hiding, Datenabstraktion) finden vorwiegend während des Designs von Programmen statt.
Tip (1995) weist darauf hin, dass eine wichtige Unterteilung etwaiger Program Slicing Algorithmen in statische, und dynamische, Program Slicing Methoden stattfinden muss.

Statisches Program Slicing wählt den jeweiligen Program Slice zu einem Slicing Criterion aus, ohne konkrete Werte oder Zustände des Programms in das Erstellen des Slices miteinzubeziehen.

Dynamisches Program Slicing, erstmals erwähnt von Korel und Laski (1988) und weiter beschrieben durch Agrawal und Horgan (1990), hingegen wertet einen speziellen Testfall mit entsprechenden Eingabevariablen und Programmzuständen aus und generiert auf Basis des jeweiligen Zustandes einen entsprechenden Program Slice. Auf Grund der dynamischen Berechnung von dynamischen Program Slices ergeben sich präzisere Slices, allerdings ist die Ausführung auch durch zeitliche und räumliche Faktoren begrenzt (vgl. Zhang u. a. (2006)).

Weiterhin muss nach Gold und Harman (2007) zwischen *Forward Slicing* und *Backward Slicing* unterschieden werden.

Beim *Forward Slicing* werden die durch das Slicing Criterion beeinflussten Zeilen zu einem Slice zusammengefasst. Mit dieser Technik kann herausgefunden werden, welchen Einfluss ein Statement auf die weitere Ausführung eines Programmes hat.

Das *Backward Slicing* hingegen betrachtet die bereits ausgeführten Zeilen des Codes. Mit Hilfe dieser Strategie kann ein Set von Statements in einem Slice zusammengefasst werden, um darzustellen welche Statements den Zustand zum Ausführungszeitpunkt des Slicing Criterions bedingt haben.

2.3 Tools & Technologien

Hier sind die in dieser Arbeit verwendeteten Tools und Technologien beschrieben.

2.3.1 Maven

Maven ist ein Build- und Konfigurationsmanagement-Tool, das dazu verwendet wird Software-Builds weitestgehend zu automatisieren. Maven erlaubt die Ausführung mit der Definition einer Phase (phase). Phasen in Maven sind: validate, compile, test, package, integration-test, verify, install und deploy. Die Ausführung mit dem Befehl deploy würde standardmäßig (die Rekonfiguration der Phasen ist ebenfalls möglich) einen Maven-Lebenszyklus starten, der in der genannten Reihenfolge alle Phasen ausführen würde.

Phasen bestehen wiederum aus Goals. Diese repräsentieren kleinere Tasks, welche in einer oder mehreren Phasen ausgeführt werden sollen. Die einzelnen $goals$ und ihre $phases$ sind in der $pom.xml$ definiert.

2.3.2 Slicing

Bei dem Erstellen von dynamischen Program Slices für Java-Programme muss jede Operation der JVM zur Laufzeit aufgezeichnet werden. Ein durch die Aufzeichnung der JVM-Operationen entstandenes Protokoll wird Trace genannt. Traces können sehr schnell sehr groß werden. Die Größe von Traces und die zwangsweise Erstellung dieser zur Laufzeit des Programms führen zu anspruchsvollen Leistungskriterien. Aufgrund dessen werden Traces in der Regel stark komprimiert und zu einem späteren Zeitpunkt analysiert, um dem Slicing Criterion entsprechende Slices zu extrahieren.

2.3.2.1 JSlice

JSlice ist ein dynamischer Slicer für Java-Programme. JSlice wurde von Wang und Roychoudhury (2004) vorgestellt. JSlice verwendet kaffe, eine modifizierte, alternative, Open-Source JVM-Implementierung, um Traces zu erstellen. Dadurch ist JSlice abhängig von aktuellen kaffe Versionen. Aufgrund dessen unterstützt JSlice Java bis einschließlich Version 1.4.

2.3.2.2 JavaSlicer

JavaSlicer ist ebenfalls ein dynamischer Slicer für Java-Programme. Der JavaSlicer wurde an der Universität Saarland von Hammacher (2008) entwickelt. JavaSlicer verwendet im Gegensatz zu JSlice einen Java Agenten[19] (vgl. Abbildung 2.7) statt die JVM direkt zu manipulieren. Somit ist JavaSlicer nicht abhängig von den Releasezyklen neuer Java Versionen und bleibt auch mit neueren Versionen kompatibel.

JavaSlicer besteht aus vier Komponenten.

Der *tracer* wird mit Hilfe der Java-Agenten Technologie in die JVM geladen und zeichnet während der Laufzeit des Programms einen Trace auf.

Der *traceReader* ist ein kleines Programm, das den im Binärformat gespeicherten Trace menschenlesbar darstellt.

Der *slicer* extrahiert aus einem Trace einen Backward Slice zu einem Slicing Criterion. Das entsprechende Slicing Criterion wird als Parameter übergeben. Ein Slicing Criterion C kann, wie in Kapitel 2.2.6.3 erläutert, durch ein Tupel bestehend aus einem Programmpunkt n und einer Menge von Variablen V beschrieben werden. Der Parameter für n sollte den

[19] Java Agenten sind ein Java-Konzept und eine Menge von Klassen, die es erlauben die durch einen Class-Loader in die JVM geladenen Klassen zu manipulieren.

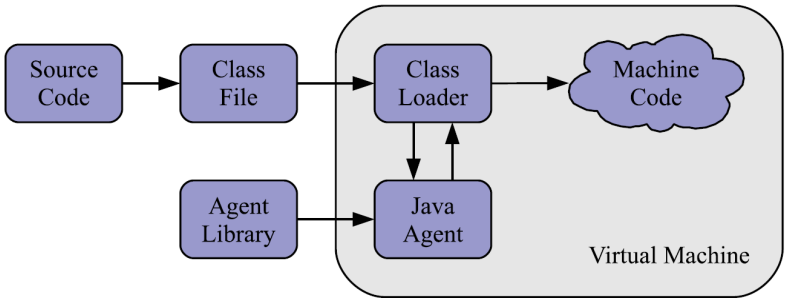

Abbildung 2.7: Java Agent Architektur aus Hammacher (2008)

Namen des Package, der Klasse und der Methode sowie die Zeilennummer enthalten. Die Definition der relevanten Variablen für den Backward Slice findet über den Parameter $\{var\ 1, var\ 2, ..., var\ i\}$ statt, wobei für alle zu beobachtenden Variablen auch ein $*$ verwendet werden kann.
Im Folgenden einige Beispiele zum Erstellen eines Backward Slices für die Zeile 53 der Main-Methode von der Main-Klasse des CoffeeMakers:
Parameter: **`edu.ncsu.csc326.coffeemaker.Main.main:53`**
Bedeutung: $C = \{n, V\}$ wobei $V = \emptyset$
Ohne Angabe von Variablen erstellt der Slicer den Slice anhand von Kontrollfluss-Abhängigkeiten.

Parameter: **`edu.ncsu.csc326.coffeemaker.Main.main:53:{var1,var2}`**
Bedeutung: $C = \{n, V\}$ wobei $V = \{var1, var2\}$
Mit der Angabe einer zu beobachtenden Variablen erstellt der Slicer den Slice auf Basis von Anweisungen, welche die Variable(n) beeinflussen.

Parameter: **`edu.ncsu.csc326.coffeemaker.Main.main:53:*`**
Bedeutung: $C = \{n, V\}$ wobei $V = \bigcup_{v \in n} v$

Außerdem bietet die Komponente *visualize* eine einfache Visualisierungsmöglichkeit des Slices. Diese ist jedoch selbst nach Meinung des Autors nicht besonders nützlich. Eine kurze Evaluierung hat die Verwendung dieser Komponente für diese Arbeit auf Grund der Unübersichtlichkeit ihrer Ausgabe ausgeschlossen.

Kapitel 3

Related Work

Dieses Kapitel beschreibt Projekte und Arbeiten der Wissenschaft, die in Bezug zu dem Thema dieser Arbeit stehen.

3.1 Fault Localization Methoden basierend auf Erfassung von Testdaten und deren Nutzung zur Fehlereingrenzung

In diesem Kapitel wird auf diverse Projekte eingegangen, die als Grundlage für die Fault Localization Testausführungen und Testauswertungen verwenden. Alle Verfahren (außer Cause Transitions) verwenden dazu Testergebnisse ausgeführter Testfälle in Verbindung mit Code Coverage Werkzeugen, um daraufhin die ausgeführten Programmbefehle der erfolgreichen Tests mit den fehlgeschlagenen zu vergleichen. Abschließend wird dem Programmierer eine Auswahl besonders verdächtiger Programmbefehle präsentiert. Das Ziel dieser Ansätze ist es, den Quellcode soweit zu minimieren, dass der Programmierer eine möglichst minimale Menge von Befehlen präsentiert bekommt, um die Fehlersuche entsprechend zu beschleunigen.

3.1.1 Set Union

Set Union ist eine einfache Methode, bei der die Menge der ausgeführten Programmbefehle aller erfolgreichen Tests B_e von der Menge der Programmbefehle eines fehlgeschlagenen Tests B_f abgezogen werden,diese Methode wird in Agrawal u. a. (1995) vorgestellt. Die resultierende Menge von Befehlen wird initial als verdächtig betrachtet und dient daraufhin dem Programmierer als Eingrenzung der Fehlerquelle. Die folgende mathematische Darstellung soll das Vorgehen verdeutlichen:
$B_{initial} = B_f - \bigcup_{e \in E} B_e$

3.1.2 Set Intersection

Set Intersection verfolgt einen umgekehrten Ansatz. Hier wird auf Grund der Annahme, ein nicht ausgeführter Befehl sei die tatsächliche Ursache für einen Fehler, eine Menge gebildet, die die ausgelassenen Befehle enthält. Dazu wird der Menge aller ausgeführten Befehle aller erfolgreichen Tests die Menge aller Befehle des fehlgeschlagenen Tests abgezogen.
$B_{initial} = \bigcup_{e \in E} B_e - B_f$

3.1.3 Nearest Neighbour

Renieris und Reiss (2003) verwenden eine Methode, bei der ein erfolgreicher Test für die Bildung einer Schnittmenge ausgewählt wird, um eine möglichst zielgerichtete Eingrenzung der fehlerhaften Befehle zu erreichen. Sie versuchen mittels *binary distancing* bzw. *permutation distancing* einen möglichst relevanten erfolgreichen Test zu dem fehlgeschlagenen Test auszuwählen. Am relevantesten ist hierbei derjenige erfolgreiche Testlauf, der dem fehlgeschlagenen Testlauf am ähnlichsten ist. Die Identifikation des ähnlichsten Testlaufs findet mittels folgender Distanzmethoden statt:

Beim *binary distancing* wird für alle erfolgreichen Testläufe die Mengendifferenz zwischen den ausgeführten Befehlen des jeweiligen erfolgreichen und des fehlgeschlagenen Testlaufs gebildet. Daraufhin wird der erfolgreiche Testlauf ausgewählt, der die kleinste Differenz zu dem zu untersuchenden fehlgeschlagenen Testlauf bildet.

Bei der Methode *permutation distancing* wird ebenfalls der Testlauf der erfolgreichen Tests analysiert. Hierbei wird die Anzahl der Ausführungen von Befehlen bzw. Blöcken ermittelt und nach deren Häufigkeit sortiert. Anschließend kann dann mit dem fehlgeschlagenen Testlauf verglichen werden. Ein Vergleich auf dieser Basis ist mit den Kosten einer Transformation von dem fehlgeschlagenen in den erfolgreichen Test gleichzusetzen.

Nach der Anwendung einer Distanzmethode auf alle erfolgreichen und einen fehlgeschlagenen Test wird der nächste erfolgreiche Test ausgewählt. Alle ausgeführten Befehle dieses erfolgreichen Tests B_{e_1} werden von den ausgeführten Befehlen des fehlgeschlagenen Tests B_f abgezogen.
$B_{initial} = B_f - B_{e_1}$

3.1.4 Cause Transitions

Von Cleve und Zeller (2005) wird ein weiterer Ansatz entwickelt, der auf der Annahme beruht, ein Fehler müsse immer unter Berücksichtigung von Raum und Zeit eines Programmes gesucht werden. Daraus resultieren zwei grundsätzliche Suchansätze:

Search in Space - Die Suche im Raum eines Programmes bedeutet das Beobachten des Programmzustandes mit dem Ziel eine fehlerverursachende Variable zu finden.

Search in Time - Die Suche in der Zeit muss außerdem durchgeführt werden, um den Zeitpunkt und das fehlerbehaftete Code-Fragment zu identifizieren, welches den inkorrekten Programmzustand hervorgerufen hat.

Hierzu haben Cleve und Zeller (2005) einen Ansatz entwickelt, bei dem ein erfolgreicher und ein fehlgeschlagener Test mit Hilfe eines Debuggers gestoppt werden und deren Zustände auf Basis des jeweiligen Speicherabbildes miteinander verglichen werden. Weiterhin ermöglicht das entwickelte System den Austausch von Speicherbereichen, wodurch das Vergleichen der beiden Testläufe zur Laufzeit ermöglicht wird. Weiterhin werden die Speicherbereiche, die scheinbar den Fault enthalten, iterativ eingegrenzt, so dass die kleinstmöglichen Zustandsänderungen identifiziert werden können. Die für diese Zustandsänderungen verantwortlichen Programmbefehle werden abschließend gesammelt und als verdächtige Befehle dem Programmierer angezeigt.

3.1.5 Tarantula

Die *Tarantula*-Anwendung ist entwickelt worden, um Entwicklern eine Möglichkeit der Darstellung von Softwarefehlern zu geben, sie wurde von Jones u. a. (2002) vorgestellt. Hierzu wird der gesamte Quellcode entsprechend einem berechneten Fehlerverdachtswert (sog. *suspiciousness score*) eingefärbt. Der Entwickler kann daraufhin die rot eingefärbten Codestellen begutachten und ggfs. korrigieren.
Jones u. a. (2001) haben besonderen Wert darauf gelegt, bereits bestehende Werkzeuge und Informationen zu verwenden. Bei Tarantula dienen die ausgeführten Testfälle, deren Ergebnisse, die ausgeführten Befehle (Code Coverage) und der Quellcode als einzige informelle Basis.
Die Basis für den Fehlerverdachtswert bietet die Annahme, dass Programmbefehle die von fehlgeschlagenen Tests ausgeführt wurden, eher fehlerbehaftet sind, als diejenigen, die primär bei erfolgreichen Testläufen ausgeführt wurden. Mit Hilfe von erfolgreichen und fehlgeschlagenen Testläufen und einer jeweils dazugehörigen Menge von ausgeführten Programmbefehlen wird der Verdachtswert mit folgender Formel berechnet:

$$Verdachtswert(b) = \frac{\frac{\sum erfolgreich(b)}{\sum erfolgreicheTests}}{\frac{\sum erfolgreich(b)}{\sum erfolgreicheTests} + \frac{\sum fehlgeschlagen(b)}{\sum fehlgeschlageneTests}}$$

Hierbei ist b der zu bewertende Programmbefehl, $erfolgreich(b)$ stellt somit einen erfolgreichen Test, bei dem Befehl b ausgeführt wurde, dar. Analog dazu stellt $fehlgeschlagen(b)$ einen Test dar, bei dem b ausgeführt wurde und der Test fehlgeschlagen ist. Weiterhin ist zu beachten, dass eine Nulldivision vermieden wird, indem der jeweilige Bruch auf Null gesetzt wird, sobald einer der Divisoren den Wert Null annimmt.

3.2 Weitere Fault Localization Ansätze

Insbesondere der konkrete Bezug zu Webapplikationen wird in diversen Arbeiten priorisiert. Artzi u. a. (2010) erarbeiten einen Ansatz zur Fehlerlokalisierung von dynamischen PHP-basierten Webapplikationen. Sie verwenden dazu einen HTML-Validator, der die vom Server erstellten HTML-Seiten auf eine korrekte Struktur prüft. Sollte invalides HTML erkannt werden, wird anhand des DOM[20]-Pfades das konkrete Element identifiziert und daraufhin das PHP-Serverprogramm mit Hilfe des DOM-Pfades mit dem fehlerhaften Element verglichen, um verdächtige Zeilen zu identifizieren. Dieser Ansatz wird wiederum mit der Tarantula-Technik kombiniert, wobei diese durch das intelligente Einbeziehen von if-else-Statements verbessert wird.

[20] Document Object Model - Eine vom W3C-Konsortium definierte Struktur eines HTML oder XML Dokumentes für die Ermöglichung von dynamischen Zugriffen und Aktualisierungen auf die Struktur, den Inhalt und den Stil von Dokumenten (vgl. dom).

Kapitel 4

Konzept

Das Ziel dieser Arbeit ist, wie in Kapitel 1.1 erläutert, die Kostenreduzierung für das Finden von Fehlern. Dazu soll der Entwickler bei dem Debugging-Prozess von Java-Programmen unterstützt werden. Dieses Ziel soll erreicht werden, indem dem Entwickler eine Menge von verdächtigen Programmzeilen präsentiert werden. Die Präsentation dieser Auswahl soll dazu führen, dass der Entwickler nicht den gesamten Programmcode begutachten muss, sondern seine Aufmerksamkeit auf die für die Fehlerursache verantwortlichen Code-Fragmente geleitet wird.

4.1 Auswahl von Fehlertypen für die Fehlerlokalisierung bzw. -eingrenzung

Die Anforderungen bezüglich der zu lokalisierenden Fehler werden im Folgenden definiert.

Um die Fehlerlokalisierung von Java-Programmen in Client-Server-Architekturen praktisch anwenden zu können, wird hier die in Kapitel 2.2.5.3 geleistete Einteilung und in Tabelle 2.1 dargestellte Zuordnung von Failure Modes zu Fault Propagation Typen verwendet, um irrelevante Kombinationen auszuschließen und schließlich die relevanten Kombinationen zu identifizieren. Zur Verdeutlichung wird in Abbildung 4.1 erneut die Zuordnung dargestellt.

4.1.1 Identifikation von Fault Propagation Typen für die Fehleranalyse

Das Betriebssystem soll, wie in Kapitel 2.2.4 erläutert, keine potentielle Fehlerquelle sein. Es wird davon ausgegangen, dass ein fehlerfreies Betriebssystem vorliegt. Das führt dazu, dass die Fault Containment/Propagation Typen *A-D* ausgeschlossen werden können.
Fault Propagation Typ *„E) JVM-Fehler mit Auswirkungen im Java-Programm“* wird ebenfalls von der Fehleranalyse ausgeschlossen, da der anschließende Absturz des Java-Programms

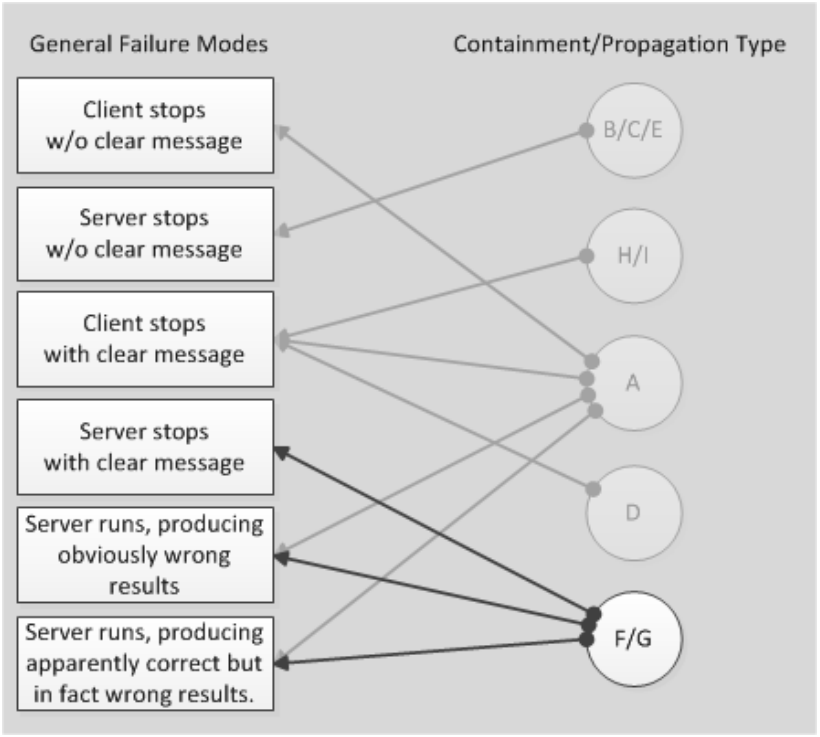

Abbildung 4.1: Die Zuordnung von Failure Modes und deren Fault Propagation Typen

einem Java-Error entspricht, welche aufgrund der Erläuterungen in Kapitel 2.1.3.1 nicht Teil der Betrachtungen dieser Arbeit sein sollen.
Fehlerursachen auf Netzwerkprotokollebene (vgl. Kapitel 2.1.2) sollen ebenfalls nicht beachtet werden, da eine Fehlerquellensuche im Netzwerk einer anderen Zielrichtung entsprächen. Fault Propagation Typ *„H) Netzwerkprotokollfehler mit Auswirkungen im Browser“* kann somit ebenfalls ausgeschlossen werden.
Der Fault Propagation Typ *„I) Browserfehler“* wird auch nicht weiter erörtert. Ein Webbrowser im Sinne eines Thin-Clients kann als abstrakte Instanz, welche nur HTTP Requests sendet und entsprechende HTTP Responses rendert, betrachtet werden. Da das Senden eines HTTP Requests eine triviale Aufgabe ist und das Rendern von HTTP Responses ein anderes Themengebiet -ohne direkten Bezug zu Java-Programmen- ist, beschränkt sich diese Arbeit auf das Server-seitige Lokalisieren und Eingrenzen von Fehlerursachen.
Aus den zuvor ausgeschlossenen Fault Propagation Typen ergeben sich die beiden Typen *„F) JVM-Fehler mit Auswirkungen im Browser“* und *„G) Java-Programmfehler mit Auswirkungen im Browser“*. Diese Zuordnung ist in Abbildung 4.1 dargestellt.

4.1.2 Abgrenzung der Analyse

Für eine zielgerichtete Fokussierung dieser Arbeit ist es nötig, dass unnötige komplexe Teile des Systems von der Analyse ausgeschlossen werden.
Der explizite Bezug auf Java-Programme in Client-Server-Architekturen führt dazu, dass ein Beispiel-Javaprogramm ausgewählt wurde. Das Beispielprogramm ist der in Kapitel 2.1.4 beschriebene CoffeeMaker. Der Quellcode des CoffeeMakers dient als einzige zu analysierende Befehlsmenge. Die konkrete Eingrenzung der möglichen Fehler auf den Quellcode des CoffeeMakers soll dazu führen, dass eine direkte Quellcode- bzw. Ausführungsanalyse zu einer erfolgreichen Fehlereingrenzung oder gar Fehlerlokalisierung führt. In dieser Arbeit sollen, wie in Kapitel 2.1.3 begründet, semantische Fehler von Java-Programmen untersucht werden.
Die Ausführungsumgebung des Javaprogrammes in Client-Server-Architekturen kann unter der Annahme, das Java-Programm biete eine Schnittstelle für die Anfragen des Clients, so weit reduziert werden, dass durch die ausschließliche Verwendung dieser Schnittstelle Zugriffe eines Web-Servers simuliert werden. Der CoffeeMaker ist für diese Annahme geeignet und bietet eine klare Schnittstelle, die durch die $CoffeeMaker$ Klasse zur Verfügung gestellt und durch die $Main$ Klasse verwendet wird.

4.1.3 Relevante Failure Modes und konkrete Failures

Im Folgenden wird eine Zuordnung von möglichen Failures zu den identifizierten Failure Modes geleistet und in Abbildung 4.2 dargestellt. Als Grundlage hierfür dient Abbildung 4.1, sie gibt durch einen Umkehrschluss von den relevanten Fault Propagation Typen Aufschluss über die drei möglichen Failure Modes *„Server stops with a clear message“*, *„Server runs, producing obviously wrong results“* und *„Server runs, producing apparently correct but in fact wrong results“*.

4.1.3.1 Server stops with a clear message

Für den Failure Mode *„Server stops with a clear message“* sind folgende Failures von Bedeutung:
Aus Kapitel 2.1.2.2 werden die relevanten Protokoll/Netzwerk-Failures (Fehlercodes 404, 500 und 504) erhoben. Diese werden, wie bereits erwähnt, von dem Webserver erzeugt und Client-seitig sichtbar. Der Fehlercode „404 - File Not Found“ soll für eine weitere Betrachtung ausgeschlossen werden, da er meistens aufgrund von fehlkonfigurierten Webservern bzw. gelöschter Dateien auftritt und die Fehlerursache i.d.R. eindeutig ist. Der Fehlercode „504 - Gateway Timeout“ wird ausgeschlossen, da er in mehrschichtigen Client-Server-Architekturen auftritt, welche in dieser Arbeit nicht als Objekt der Analyse vorgesehen sind,

aus Gründen die in Kapitel 2.1.1.2 erläutert sind.
Die Fehler, die durch den Fehlercode „500 - Internal Server Error“ repräsentiert werden, sind semantische Fehler. Sie treten zur Laufzeit auf und werden von der Programmiersprache Java als Fehler erkannt. Diese können ebenfalls von der Anwendung definierte Exception-Klassen sein, welche keine besondere Betrachtung erfordern, da der Mechanismus für das sog. Werfen und Fangen von Exceptions für anwendungsspezifische Ausnahmen identisch ist.

4.1.3.2 Server runs, producing obviously wrong results und Server runs, producing apparently correct but in fact wrong results

„Server runs, producing obviously wrong results“ und *„Server runs, producing apparently correct but in fact wrong results“* sind Failures, die jederzeit aufgrund fehlerhafter Semantik des Programmes auftreten können. Diese sehr vielfältige Kategorie kann von Berechnungs-, Daten-, oder Logikfehlern bis hin zu falsch verstandenen oder falsch formulierten Anforderungen reichen.
Im Unterschied zu dem vorhergehenden Failure Mode sind dies keine von der Programmiersprache zur Laufzeit erkannte Fehler. Diese Fehler müssen von dem Benutzer der Software oder durch semantische Tests entdeckt werden.

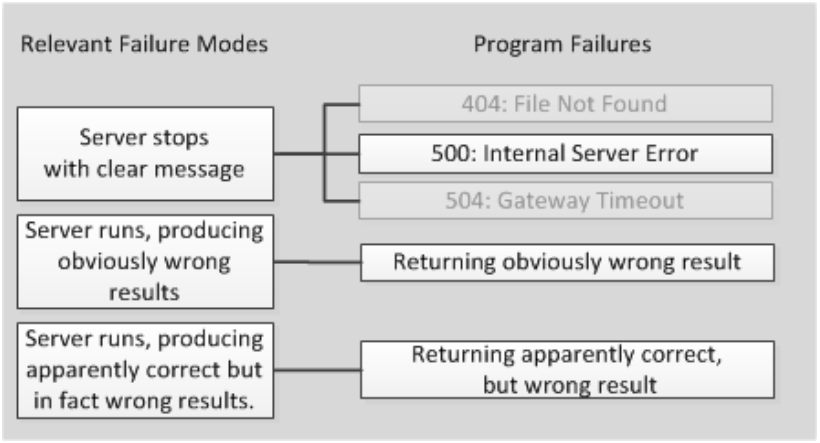

Abbildung 4.2: Die Zuordnung von Failure Modes zu entsprechenden Failures

4.1.4 Lösungsansatz für die Fehlereingrenzung und -lokalisierung

Die vorliegende Arbeit versucht auf Basis der in Kapitel 3 genannten Vorgehensweisen „Set Union“ und „Set Intersect“ einen erweiterten Ansatz zu generieren. Dabei soll eine Kombination aus beiden Ansätzen zum Einsatz kommen. Für die Erstellung der Mengen von

ausgeführten Befehlen eines Tests kommt der JavaSlicer (vgl. Kapitel 2.3.2.2) zum Einsatz. Weiterhin sollen anschließend Mengenoperationen auf die resultierenden Befehle angewendet werden. Abschließend werden die einzelnen Befehle entsprechend eines Verdachtswertes (vgl. Jones u. a. (2002)) bewertet und dem Entwickler zur Unterstützung beim Debugging oder für einen Code-Review dargeboten.

4.2 Testdesign

Die konkreten Tests, die für die Auswertung der Fehlerlokalisierung herangezogen werden, müssen zielgerichtet ausgewählt werden. Das konzipierte Vorgehen sieht die Ausführung von einem fehlgeschlagenen und einem erfolgreichen Test vor. Wie in diversen anderen Arbeiten (vgl. Jones u. a. (2002), Renieris und Reiss (2003) und Cleve und Zeller (2005)) wird auch hier der Ansatz verfolgt, ähnliche Programmabläufe zu identifizieren. Das soll dazu führen, dass die erzeugten Traces sehr ähnliche Mengen von Programmbefehlen enthalten, um durch Mengenoperationen eine möglichst kleine Menge vergleichen zu müssen und die resultierende Menge ebenfalls möglichst klein ist. Der Vergleich von minimalen Befehlsmengen führt zu einer schnellen Ausführungszeit der Analyse. Die kleine resultierende Menge von verdächtigen Befehlen führt zu dem wichtigen Designziel, dem Entwickler möglichst wenig Zeilen zu präsentieren und somit eine effektivere Untermenge anbieten zu können.

4.2.1 Ermittlung geeigneter manueller Testfälle

Die Auswahl der konkreten Testfälle soll für diese Arbeit manuell und auf Basis von Kenntnis über den Quellcodes bzw. der Schnittstelle des SUT[21] stattfinden. Eine automatisierte Auswahl von geeigneten Testfälle, kann in Betracht gezogen werden, soll hier jedoch vorerst nicht weiter betrachtet werden.

4.2.2 Ermittlung geeigneter JUnit-Testfälle

In Kapitel 5 wird gezeigt werden, dass die Ausführung von manuellen Testfällen aufgrund von Unzulänglichkeiten des verwendeten JavaSlicers nicht zu dem gewünschten Ergebnis führt. Deswegen wird hiermit die Ausführung von JUnit-Tests eingeführt, um Tests auf Komponentenebene analysieren zu können. Dieses Vorgehen widerspricht dem Designziel, eine Fehlerlokalisierung ohne tiefgreifendes Wissen über das SUT durchführen zu können, ist aber durch eine Erweiterung des JavaSlicers vermeidbar und soll hier nur dem Zweck dienen, das vorgeschlagene Konzept unter Beweis zu stellen.

[21] SUT = System under Test

Die Auswahl der entsprechenden JUnit-Testfälle geschieht auf Komponenten- bzw. Methodenbasis. D.h. es werden explizit erfolgreiche und fehlschlagende Testfälle für eine Komponente bzw. Methode erstellt.

4.3 Testausführung

Im Folgenden soll die Ausführung und Instrumentierung der Testfälle beschrieben werden.

4.3.1 Ausführung von manuellen Tests

Die manuellen Programmläufe werden durch das in Listing 4.1 ersichtliche Skript ausgeführt. Das Skript führt das Javaprogramm, in Verbindung mit besagtem Java Agenten für das Tracing, zweimalig aus. Außerdem erstellt das Skript nach dem jeweiligen Programmdurchlauf ebenfalls die Slices, entsprechend einem Slicing Criterion. Nach dem Starten des Skriptes können die einzelnen Testfälle manuell durchgeführt werden.

Listing 4.1: Startskript startSlicer.bat für die Ausführung und Instrumentierung von manuell durchzuführenden Testfällen

```
@ECHO OFF
REM This program calls sequentially a defined work flow to produce program slices of a
    simple java program
REM %1 should be the test name
REM %2 should be the slicing criterion
echo Building coffeemaker.jar
call mvn package -DskipTests
echo Starting Tracer
echo First Run (Success)
call java -javaagent:lib\tracer.jar=tracefile:reports\%1_success.trace
    -jar target\coffeemaker-0.1.jar
call java -Xmx2g -jar lib\slicer.jar -p reports\%1_success.trace \%2
    > reports\%1_success.slice
echo Second Run (Failing)
call java -javaagent:lib\tracer.jar=tracefile:reports\%1_fail.trace
    -jar target\coffeemaker-0.1.jar
call java -Xmx2g -jar lib\slicer.jar -p reports\%1_fail.trace \%2 > reports\%1_fail.slice
```

4.3.2 Ausführung von JUnit-Tests

In Kapitel 4.2.2 wurde bereits erläutert, dass die zusätzliche Ausführung von JUnit-Tests erforderlich geworden ist.
Die Ausführung der JUnit-Tests wird durch das in Listing 4.2 dargestellte Skript durchgeführt. Für die Ausführung von JUnit-Tests ist es wichtig, dass die Testklassen einer Namenskonvention entsprechen. Diese Konvention ist für die einfache Zuordnung und Ausführung eines erfolgreichen und eines fehlschlagenden Tests erforderlich. Hierbei sollten die Testklas-

sen `${testname}_success.java` bzw. `${testname}_fail.java` heißen und jeweils nur einen einzigen Test enthalten.

Listing 4.2: Startskript startSlicerJUnit3.bat für die Ausführung und Instrumentierung von JUnit-basierten Testfällen

```
1  @ECHO OFF
2  REM This program calls sequentially a defined work flow to produce program slices of a
       simple java program
3  REM %1 should be the successful test class name without .java/.class extension
4  REM %2 should be the failing test class name without .java/.class extension
5  REM %3 should be the slicing criterion
6  REM Example call: startSlicerJUnit3.bat FaultX_success FaultX_fail
7      edu.ncsu.csc326.coffeemaker.RecipeBook.addRecipe:37
8  echo Building coffeemaker.jar
9  call mvn package -DskipTests
10 echo Building coffeemaker-tests.jar
11 call mvn jar:test-jar
12 echo Starting Tracer
13 echo First Run (Success) %1
14 call java -javaagent:lib\tracer.jar=tracefile:reports\%1.trace
15     -cp lib\junit-3.8.2.jar;target\coffeemaker-0.1.jar;target\coffeemaker-0.1-tests.jar
16     junit.textui.TestRunner edu.ncsu.csc326.coffeemaker.%1
17 call java -Xmx2g -jar lib\slicer.jar -p reports\%1.trace %3 > reports\%1.slice
18 echo Second Run (Failing) %2
19 call java -javaagent:lib\tracer.jar=tracefile:reports\%2.trace
20     -cp lib\junit-3.8.2.jar;target\coffeemaker-0.1.jar;target\coffeemaker-0.1-tests.jar
21     junit.textui.TestRunner edu.ncsu.csc326.coffeemaker.%2
22 call java -Xmx2g -jar lib\slicer.jar -p reports\%2.trace %3
23     > reports\%2.slice
```

4.3.3 Instrumentierung

Während der CoffeeMaker gestartet wird, wird ein Java Agent (siehe Listing 4.1 Zeilen 9 und 14 bzw. Listing 4.2 Zeilen 14 und 19) geladen. Als Agent wird hier der Tracer des JavaSlicers aus Kapitel 2.3.2.2 verwendet. Er zeichnet zur Laufzeit des CoffeeMakers sämtliche ausgeführten Bytecode-Befehle auf und speichert diese in einem Trace-File `${testname}_success.trace` bzw. `${testname}_fail.trace`.

Eine Übersicht der Architektur ist in Abbildung 4.3 dargestellt.

4.4 Testauswertung

Nachdem mindestens zwei Testfälle durchgeführt und deren Traces aufgezeichnet worden sind, werden aus den Traces entsprechend eines Slicing Criterions die Slices generiert. Dazu wird das Slicer-Tool des Javaslicers verwendet. Die Erstellung der Slices aus den aufgezeichneten Traces findet bereits durch eines der in Kapitel 4.3 erwähnten Skripte statt.

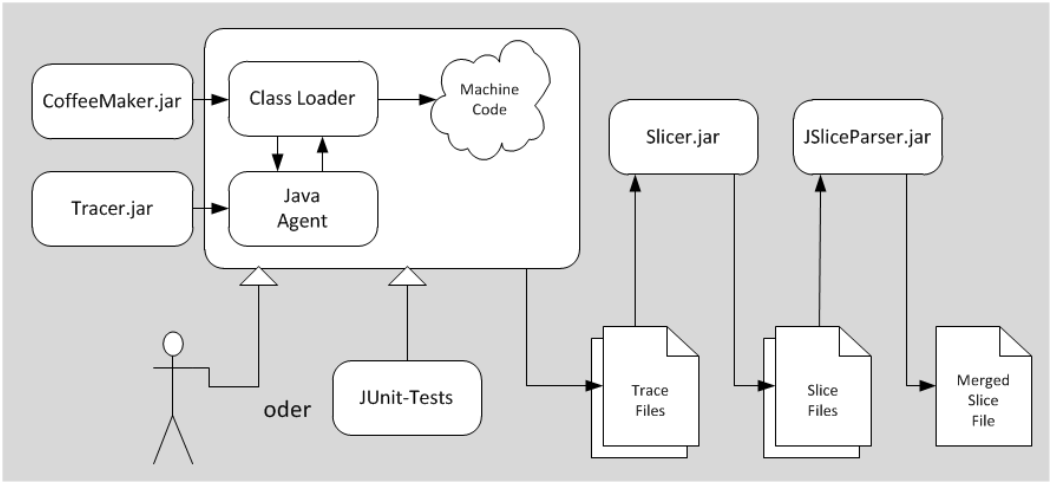

Abbildung 4.3: Die vollständige Architektur von der Ausführung des CoffeeMaker zur Erstellung der vereinigten Slices

Weiterhin dient ein kleines selbst geschriebenes Programm dazu, die Menge der Befehle zu aggregieren und Mengenoperationen darauf anwenden zu können.

4.4.1 Wahl des Slicing Criterions

Die Auswahl des richtigen Slicing Criterions ist eine nicht-triviale Aufgabe.
Die Komplexität entsteht durch die Tatsache, dass für die Auswahl eines Slicing Criterions bereits Wissen über das entsprechende Codefragment vorhanden sein muss.
Sollte eine Assertion[22] in einem JUnit-Test nicht erfüllt sein, ist die Wahl des Slicing Criterions relativ einfach. Aus der letzten Befehlszeile, die diesen Wert enthält, kann zusammen mit der Variablen (deren Wert inkorrekt ist) in der entsprechenden Zeile ein Slicing Criterion definiert werden.
Bei der manuellen Ausführung der Tests muss sich der Entwickler vorerst einen Überblick über die Schnittstelle und die Aufrufhierarchie verschaffen, um daraufhin die zuletzt ausgeführte Zeile mit einem Zugriff auf die Variable zu identifizieren.
Dieses Vorgehen bedeutet initialen Aufwand seitens des Entwicklers, welcher jedoch nicht besonders erheblich ist. Die Zeitersparnis, die durch eine potentielle Identifikation von verdächtigen Zeilen entstehen kann, bleibt nach Einschätzung des Autors weiterhin vorteilhaft.

[22] Zusicherung eines bestimmten Wertes einer Variablen.

4.4.2 Generierung der Slices

Die Generierung von Slices aus den aufgezeichneten Traces geschieht mit dem Slicer-Tool des Javaslicers. Diese Generierung ist aus dem Trace in Kombination mit dem gewählten Slicing Criterion per Kommandozeilenbefehl durchführbar. Ein beispielhafter Befehl für die Erstellung eines Slices ist in Skript 4.1 in Zeilen 11-12 und 16 sowie in Skript 4.2 in Zeilen 17 und 22 ersichtlich.

4.4.3 Darstellung und Auswertung der Slices

Für die Darstellung der relevanten Slices wird ein vom Autor selbst geschriebenes Javaprogramm „JSliceParser“ verwendet. In dem Flussdiagramm in Anhang A.2.1 ist der Algorithmus dargestellt. Das Programm bekommt mindestens zwei Slice-Files, eine Liste relevanter Packages und den Pfad zum Quellcode als Parameter übergeben.

Der JSliceParser filtert daraufhin die Slices entsprechend der relevanten Packages, um die Operationen der JVM und der Java-API herausfiltern zu können. Weiterhin ist dieses Filtern für modulare Systeme interessant, bei denen bestimmte Frameworks und Bibliotheken nur übernommen, und nicht beeinflusst werden können, und somit für die Fehlersuche uninteressant sind. Daraufhin werden die Programmzeilen der Slices aus dem Quellcode gesammelt und im Slice gespeichert. Abschließend finden Mengenoperationen auf den Slices und die Berechnung der Verdachtswerte statt.

Für die Ausführung der Auswertung mit dem JSliceParser wurde ein kleines Skript verwendet, um das wiederholte Editieren sämtlicher Eingabeparameter zu vereinfachen. Der Inhalt des Skriptes ist in 4.3 gelistet.

Listing 4.3: Startskript startMerge.bat für die Analyse der Slices

```
1  @ECHO OFF
2  REM This program filters and merges given slices
3  REM %1 should be the test name
4  @ECHO off
5  mvn exec:java
6      -Dexec.mainClass="de.dedaj.faultlocation.slicing.Main"
7      -Dexec.args=
8         "reports/%1_fail.slice
9          reports/%1_success.slice
10         edu.ncsu.csc326
11         e:/dev/dev/workspace/CoffeeMaker_FIT/src/
12         e:/dev/dev/workspace/CoffeeMaker_FIT/tests/"
```

Die Zeilen 8-12 sind die Parameter, die der main-Methode des JSliceParsers übergeben werden. Zeile 8 entspricht dem zu analysierenden fehlgeschlagenen Slice. Zeile 9 ist in diesem konkreten Beispiel ein erfolgreicher Slice, hier kann ebenso eine Komma-separierte Liste von erfolgreichen Slices übergeben werden. Zeile 10 enthält den zu filternden Packagenamen, andere Packages erscheinen nicht in der Ausgabe, hier kann ebenfalls eine komma-

separierte Liste übergeben werden. In den letzten beiden Zeile wird der Pfad zu dem Quellcode des CoffeeMakers und zu den JUnit-Tests übergeben, damit die Slices mit konkreten Codezeilen angereichert werden können.

4.4.4 Mengenoperationen und Berechnung von Verdachtswerten

Im Kapitel 3 ist erläutert worden, wie bei den Verfahren Set Union und Set Intersect vorgegangen wird. Diese Arbeit versucht eine Weiterentwicklung der genannten Verfahren durch Kombination dieser zu leisten. Anstatt die verdächtigen Programmzeilen des erfolgreichen Tests von denen des fehlgeschlagenen Tests bzw. die verdächtigen Programmzeilen des fehlgeschlagenen Tests von denen des erfolgreichen Tests zu subtrahieren, soll mit dieser Arbeit ein anderer Ansatz untersucht werden.
Es werden die Untermengen $B_{\Delta f} = B_f - Be$ und $B_{\Delta e} = B_e - Bf$ erzeugt, unterschiedlich bewertet $\forall b \in B_{\Delta e}$ gilt $v(b_e)) = 0,5$ und $\forall b \in B_{\Delta f}$ gilt $v(b_f)) = 1,0$, wobei $v()$ für den Verdachtswert steht. Die Verdachtswerte von $0,5$ für Befehlszeilen, die ausschließlich von erfolgreichen Tests ausgeführt werden, und $1,0$ für Befehlszeilen, die ausschließlich von fehlgeschlagenen Tests ausgeführt werden, beruhen auf der Annahme, dass eine Befehlszeile einer fehlgeschlagenen Testfallausführung erst einmal verdächtiger ist, als andere. Es kann aber ebenso ein Fehler durch eine ausgelassene Anweisung entstanden sein. Aufgrund dessen wird $v(b_e) = 0.5$ gesetzt. Diese Werte sind eine initiale Idee und sollten in der Praxis erprobt werden. Die bewerteten Befehlszeilen werden anschließend vereinigt $B_{final} = B_{\Delta e} \cup B_{\Delta f}$ und dem Entwickler mit den entsprechenden Verdächtigkeitswerten präsentiert.

Kapitel 5

Demonstration

In diesem Kapitel soll das entwickelte Vorgehen in praktischem Umfeld untersucht und ausgewertet werden. Dazu werden vorerst die in Kapitel 4.1.3 identifizierten Failure Modes als Grundlage für die Erstellung von konkreten Faults verwendet. Zu jedem Softwarefehler werden zwei Testfälle erstellt. Ein erfolgreicher und ein fehlschlagender Testfall.

Zu jedem Fault folgt daraufhin ein eigenes Unterkapitel, um den Versuch der Fehlerlokalisierung bzw. -eingrenzung zu dokumentieren und zu analysieren.

Im Anschluss wird die gewählte Fault Localization Methode anhand ihrer Erfolgsquote in der praktischen Demonstration bewertet.

5.1 Erstellung konkreter Faulttypen

Zu jedem Failure Mode aus Kapitel 4.1.3 sollen hier relevante Faults ausgewählt werden. Aus diesen ergeben sich folgende drei konkrete Failure Modes:

1. Ein Fault in der Ausprägung eines Laufzeitfehlers. Diese würde in einer Serverumgebung zu einem HTTP Error *„500: Internal Server Error“* führen.
2. Ein Fault, der zu einem offensichtlich falschen Ergebnis führt.
3. Ein Fault, der augenscheinlich korrekt ist, das Ergebnis jedoch tatsächlich falsch ist.

5.2 Faulttyp 1: Laufzeitfehler

Das Auftreten eines Laufzeitfehlers wird durch alle Exception-Klassen, welche von der $RuntimeException$ erben, repräsentiert.

5.2.1 Beispiel-Fault 1: NullPointerException

Eine häufig auftretende Exception aus der in Abbildung 2.3 dargestellten Klassenhierarchie ist die $NullPointerException$. Diese Exception weist auf einen konkreten Programmierfehler des Entwicklers hin, da das Objekt, auf das ein Zugriff stattfinden soll, nicht korrekt initialisiert oder zumindest die Überprüfung auf einen potenziellen Nullwert vergessen wurde. Die $NullPointerException$ soll als Basis für den ersten Beispiel-Fault dienen.

5.2.1.1 Quellcode

Der Quellcode 5.1 zeigt einen üblichen Programmierfehler. Vor der Anweisung in Zeile 254 ist eine Überprüfung auf einen potenziellen Nullwert nötig (vgl. Zeile 257).

Listing 5.1: Beispiel-Fault 1: NullPointerException aus CoffeeMaker.java

```
249  private static void printRecipeNames(Recipe[] recipes) {
250    for (int i = 0; i < recipes.length; i++) {
251      //@formatter:off
252      //*/ XXX: NullPointerException because
253      // of iteration over complete recipes which may not exist
254      System.out.println((i + 1) + ". " + recipes[i].getName());
255      /*/
256      if(recipes[i] != null){ // necessary null-check
257        System.out.println((i + 1) + ". " + recipes[i].getName());
258      }
259      //*/
260      //@formatter:on
261    }
262  }
```

5.2.1.2 Erfolgreicher Testfall

Dieser Test wird manuell durchgeführt.
Testfallbeschreibung:
Dem CoffeeMaker werden drei Rezepte hinzugefügt, daraufhin wird eines der Rezepte editiert.
Ergebnis:
Die Durchführung des erfolgreichen Testfalls führt zu einem fehlerfreien Programmablauf.

5.2.1.3 Fehlschlagender Testfall

Dieser Test wird manuell durchgeführt.
Testfallbeschreibung:
Dem CoffeeMaker wird ein Rezept hinzugefügt, daraufhin wird versucht dieses Rezept zu

editieren.
Ergebnis:
Dieser Testfall führt wie erwartet zu einer NullPointerException, da bei dem Versuch den Namen auszugeben auf einen Nullwert zugegriffen wird.

5.2.1.4 Slicing Criterion

Das Slicing Criterion für einen Laufzeitfehler ist relativ leicht zu identifizieren. Die Zeile des Quellcodes, an der die Ausnahme bei der Ausführung des fehlgeschlagenen Tests aufgetreten ist, wird direkt im Stacktrace ausgegeben. Der Zugriff auf die entsprechende nicht korrekt bzw. nicht vollständig initialisierte Variable ist in diesem Fall der Ausgangspunkt für den zu erstellenden Trace. Daraus ergibt sich das Slicing Criterion: $edu.ncsu.csc326.coffeemaker.Main.printRecipeNames : 254$. Die Beobachtung der Variablen *recipes* kann leider nicht vollzogen werden, da der JavaSlicer das Erstellen von Slices für Parameter leider nicht unterstützt.

5.2.1.5 Durchführung und Analyse der Fehlerlokalisierung

Die beiden erstellten Slices ergeben nach dem Anwenden der in Kapitel 4.4.4 genannten Methoden die im Anhang A.1 befindliche Ausgabe. Die Mengenoperationen führen, wie in der Ausgabe ersichtlich, zu einer leeren Menge. Das bedeutet für diesen ersten Testfall ein negatives Ergebnis.
Das Ergebnis ist auf eine Unzulänglichkeit des JavaSlicers zurückzuführen. Dieser führt das mehrmalige Ausführen eines speziellen Befehls bei einer Kontrollflussanalyse nicht erneut auf. Diese Eigenschaft führt dazu, dass für die Fehlerlokalisierungsmethode beide Ausführungen identisch aussehen, obwohl bei der Ausführung des erfolgreichen Testfalls das mehrfache Hinzufügen von Rezepten auftauchen sollte.
Es wäre zu erwarten gewesen, dass der JavaSlicer das mehrmalige Ausführen von Anweisungen ebenfalls aufnimmt, da er laut Dokumentation dynamische Slices erstellt (vgl. Hammacher (2008)).
Zum Zwecke der Demonstration wird im folgenden Unterkapitel ein ähnlicher Trace erstellt. Hier soll der fehlschlagende Test gar kein Rezept hinzufügen. Es wird erwartet, dass das fehlende Hinzufügen eines Rezeptes zu besagtem Fehler führt.

5.2.2 Beispiel-Fault 1.1: NullPointerException

Wie vorangehend bemerkt, soll hier erneut das Auftreten einer $NullPointerException$ verursacht werden.

5.2.2.1 Quellcode

Der Quellcode des Faults ist identisch zu dem in 5.1.

5.2.2.2 Erfolgreicher Testfall

Der erfolgreiche Testfall ist identisch zu dem in 5.2.1.2.

5.2.2.3 Fehlschlagender Testfall

Der fehlschlagende Test wird bei diesem Testlauf kein Rezept hinzufügen, um den Fehler und einen anderen Trace zu provozieren. Der Test wird manuell durchgeführt:
Testfallbeschreibung:
Es wird direkt versucht, ein Rezept zu editieren.
Ergebnis:
Dieser Testfall führt wie erwartet zu einer NullPointerException, da bei dem Versuch den Namen auszugeben auf einen Nullwert zugegriffen wird.

5.2.2.4 Slicing Criterion

Das Slicing Criterion ist ebenfalls wie in Kapitel 5.2.1.4 definiert.

5.2.2.5 Durchführung und Analyse der Fehlerlokalisierung

Der vereinigte Slice 5.2 enthält die für den Fault verantwortliche Codezeile $addRecipe()$ (Zeile 4 im Report) mit dem Verdachtswert $v = 0,5$. Der Entwickler wäre indirekt auf die Fehlerursache hingewiesen worden. Die fehlende Ausführung der $addRecipe()$-Methode ist die Ursache für das Auftreten der $NullPointerException$.

Listing 5.2: Vereinigter Slice für Fault 1.1

```
1 class, method and linenumber                          |v()|code
2 |-----------------------------------------------------|---|---------------------------------------
3 edu.ncsu.csc326.coffeemaker.Main#mainMenu:37           |0.5|        if (userInput == 1)
4 edu.ncsu.csc326.coffeemaker.Main#mainMenu:38           |0.5|          addRecipe();
5 edu.ncsu.csc326.coffeemaker.Main#addRecipe:103         |0.5|      mainMenu();
6 edu.ncsu.csc326.coffeemaker.Main#editRecipe:137        |0.5|    int recipeToEdit = recipeListSelection("Please select the
      number of the recipe to edit.");
7 edu.ncsu.csc326.coffeemaker.Main#editRecipe:139        |0.5|    if (recipeToEdit <= 0) {
8 edu.ncsu.csc326.coffeemaker.Main#editRecipe:140        |0.5|      mainMenu();
```

5.2.3 Beispiel-Fault 2: ArrayIndexOutOfBoundsException

Ein ebenfalls häufig auftretender Fehler ist die Java-Ausnahme $ArrayIndexOutOfBoundsException$. Sie tritt auf, sobald versucht wird auf einen Bereich eines Arrays zuzugreifen, der außerhalb des initialisierten Bereiches liegt.

5.2.3.1 Quellcode

Der folgende Quellcode zeigt in Zeile 66 eine fehlerhafte Indizierung. Dieser Fehler könnte nach einem Refactoring entstanden sein, bei dem die Schnittstelle der Methode mit dem Beginnen des Indexes von 1 auf 0 geändert und dabei die Zeile 66 vergessen bzw. übersehen wurde.

Listing 5.3: Beispiel-Fault 1.2: ArrayIndexOutOfBoundsException aus RecipeBook.java

```
63  public synchronized String deleteRecipe(int recipeToDelete) {
64    // @formatter:off
65    //*/ XXX: Fault2 ArrayIndexOutOfBoundsException
66    if (recipeArray[recipeToDelete - 1] != null) {
67    /*/
68    if (recipeArray[recipeToDelete] != null) {
69    //*/
70    // @formatter:on
71      String recipeName = recipeArray[recipeToDelete].getName();
72      recipeArray[recipeToDelete] = null;
73      return recipeName;
74    } else {
75      return null;
76    }
77  }
```

5.2.3.2 Erfolgreicher Testfall

Der Test wird manuell durchgeführt:
Testfallbeschreibung:
Es werden drei Rezepte hinzugefügt und daraufhin das zweite gelöscht.
Ergebnis:
Dieser Testfall führt zwar zu einem technisch fehlerfreien Durchlauf des Programms, ist jedoch mit einem fachlichen Fehler behaftet. Durch den falschen Zugriff auf das Array wird das falsche Rezept gelöscht. Dies ist für die Fehlerlokalisierung hier nicht relevant, da die Ursache des Auftretens der $ArrayIndexOutOfBoundsException$ untersucht werden soll.

5.2.3.3 Fehlschlagender Testfall

Der Test wird manuell durchgeführt:
Testfallbeschreibung:
Es wird ein Rezept hinzugefügt und anschließend gelöscht.
Ergebnis:
Dieser Testfall führt zum Auftreten besagter $ArrayIndexOutOfBoundsException$ an Codezeile 66.

5.2.3.4 Slicing Criterion

Die Definition des Slicing Criterions ist mit Hilfe des Stacktraces einfach. Die Zeile, an der sich der Fehler bemerkbar macht, wird erneut im Stack Trace ausgegeben. Daraus ergibt sich das Slicing Criterion $edu.ncsu.csc326.coffeemaker.RecipeBook.deleteRecipe : 66$. In diesem Fall kann ebenfalls die Variable $recipeArray$ für die Erstellung des Slices miteinbezogen werden, da dies kein Parameter, sondern eine Instanzvariable ist und der JavaSlicer somit die beeinflussenden Anweisungen identifizieren kann. Das kombinierte Slicing Criterion ist $edu.ncsu.csc326.coffeemaker.RecipeBook.deleteRecipe : 66 : \{recipeArray\}$.

5.2.3.5 Durchführung und Analyse der Fehlerlokalisierung

Die vereinigte Befehlsmenge der Slices macht den Entwickler darauf aufmerksam, dass die Ausführung des Hinzufügens von Rezepten bei dem erfolgreichen Test stattgefunden hat. Bei diesem Fehler wird die Fehlerursache (der Zugriff mit einem falschen Index) nicht in den vereinigten Slice mit aufgenommen. Es wird jedoch ersichtlich, dass zusätzliche Zugriffe auf die Instanzvariable stattgefunden haben. Dennoch ist diese Fehlerlokalisierung als negativ zu bewerten, da der fehlerbehaftete Code nicht erkannt wird.

Listing 5.4: Vereinigter Slice für Fault 1.2

```
1  class, method and linenumber                          |v()|code
2  |-----------------------------------------------------|---|---------------------------------------
3  edu.ncsu.csc326.coffeemaker.CoffeeMaker#addRecipe:31  |0.5|    return recipeBook.addRecipe(r);
4  edu.ncsu.csc326.coffeemaker.Main#addRecipe:84         |0.5|    Recipe r = new Recipe();
5  edu.ncsu.csc326.coffeemaker.Main#addRecipe:93         |0.5|      boolean recipeAdded = coffeeMaker.addRecipe(r);
6  edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:34   |0.5|    boolean exists = false;
7  edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:36   |0.5|    for (int i = 0; i < recipeArray.length; i++) {
8  edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:43   |0.5|    boolean added = false;
9  edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:45   |0.5|    if (!exists) {
10 edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:46   |0.5|      for (int i = 0; i < recipeArray.length && !added; i++) {
11 edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:47   |0.5|        if (recipeArray[i] == null) {
12 edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:48   |0.5|          recipeArray[i] = r;
```

Die gewählte Methode zur Fehlerlokalisierung kann hier nicht erfolgreich sein, weil die Fehlerursache und die Fehlerwirkung an derselben Anweisung liegen. Ein Backward Slice nimmt diese Zeile nicht mit auf.

5.2.4 Beispiel-Fault 3: NullPointerException JUnit

Aufgrund des in Kapitel 5.3.1 erkannten fehlenden Features des JavaSlicers, Komponentenübergreifend Parameter verfolgen zu können, soll mit diesem Fault das Analysieren einer einzelnen Methode auf Basis von JUnit-Tests erprobt werden.

Dazu wurden zwei Tests erstellt, welche die $addRecipe(recipe)$ Methode der $RecipeBook$ Klasse testen.

5.2.4.1 Quellcode

Die Fehlerwirkung tritt in Zeile 38 des in Listing 5.5 ersichtlichen Codes auf, wenn der Übergabeparameter recipe dem Wert $null$ entspricht.

Listing 5.5: Beispiel-Fault 1.3: NullPointerException bei falschem Übergabeparameter aus RecipeBook.java

```
32  public synchronized boolean addRecipe(Recipe r) {
33    // Assume recipe doesn't exist in the array until find out otherwise
34    boolean exists = false;
35    // Check that recipe doesn't already exist in array
36    for (int i = 0; i < recipeArray.length; i++) {
37      // XXX: Fault1.3 Unchecked access to object. Possible NullPointerException
38      if (r.equals(recipeArray[i])) {
39        exists = true;
40      }
41    }
42    // Assume recipe cannot be added until find an empty spot
43    boolean added = false;
44    // Check for first empty spot in array
45    if (!exists) {
46      for (int i = 0; i < recipeArray.length && !added; i++) {
47        if (recipeArray[i] == null) {
48          recipeArray[i] = r;
49          added = true;
50        }
51      }
52    }
53    return added;
54  }
```

5.2.4.2 Erfolgreicher Testfall

Für den erfolgreichen Testfall wird ein Rezept instanziiert und dem RecipeBook mit Hilfe der $addRecipe(recipe)$ Methode hinzugefügt.

Listing 5.6: JUnit Testfall aus Fault13success.java

```
12  public void testAddRecipe_oneRecipe_onlyFirstRecipeSet() {
13    Recipe recipe = new Recipe();
14    assertEquals("Recipe book should have a maximum and inital capacity
          of three", 3, recipeBook.getRecipes().length);
15    assertTrue("Adding of a recipe should work", recipeBook.addRecipe(
          recipe));
16    assertEquals("Recipe book should have a maximum and inital capacity
          of three", 3, recipeBook.getRecipes().length);
```

```
17      assertNotNull("Added recipe should exist", recipeBook.getRecipes()
            [0]);
18      assertNull("No other recipe should be in the book", recipeBook.
            getRecipes()[1]);
19      assertNull("No other recipe should be in the book", recipeBook.
            getRecipes()[2]);
20    }
```

5.2.4.3 Fehlschlagender Testfall

Der fehlschlagende Test simuliert eine Fehlerursache, bei der die aufrufende Komponente einen Nullwert als Parameter übergibt.

Listing 5.7: JUnit Testfall aus Fault13fail.java

```
12    public void testAddRecipe_null_shouldThrowNPE() {
13      recipeBook.addRecipe(null);
14    }
```

5.2.4.4 Slicing Criterion

Das Slicing Criterion kann anhand des Stacktraces identifiziert werden. Der Stacktrace weist auf den Failure an Zeile 37 des RecipeBooks hin. Daraus ergibt sich das Slicing Criterion $edu.ncsu.csc326.coffeemaker.RecipeBook.addRecipe : 38$.

5.2.4.5 Durchführung und Analyse der Fehlerlokalisierung

Die vollständige Ausgabe des JSliceParsers ist in Anhang 5.2.4 ersichtlich. Der vereinigte Slice in Listing 5.8 zeigt den fehlerverursachenden Aufruf des Tests in Zeile 6.
Die Fehlerlokalisierung ist bei diesem Fault als erfolgreich zu bewerten.

Listing 5.8: Vereinigter Slice für Fault 1.3

```
1 class, method and linenumber                                |v()|code
2 |-----------------------------------------------------------|---|----------------------------------------
3 edu.ncsu.csc326.coffeemaker.Fault_1_3_success#setUp:9 |0.5|    recipeBook = new RecipeBook();
4 edu.ncsu.csc326.coffeemaker.Fault_1_3_success#testAddRecipe_oneRecipe_onlyFirstRecipeSet:15|0.5|    assertTrue("Adding of
      a recipe should work", recipeBook.addRecipe(recipe));
5 edu.ncsu.csc326.coffeemaker.Fault_1_3_fail#setUp:9    |1.0|    recipeBook = new RecipeBook();
6 edu.ncsu.csc326.coffeemaker.Fault_1_3_fail#testAddRecipe_null_shouldThrowNPE:13|1.0|    recipeBook.addRecipe(null);
```

5.3 Faulttyp 2: Offensichtlicher Fehler

Bei diesem Faulttyp sind offensichtliche Fehler des CoffeeMakers zu lokalisieren. Dazu gehören Failures, bei denen der Anwender unmittelbar ein Fehlverhalten identifizieren kann. Diese Fehler werden jedoch nicht wie die Faults des vorhergehenden Faulttyps durch eine explizite Ausnahme ersichtlich, sondern sind fachliche Fehler.

5.3.1 Beispiel-Fault 1: Doppeltes Rezept

Bei dem ersten Fault dieses Faulttyps wird angenommen, dass sich ein Logik-Fehler eingeschlichen hat. Dieser vermeidet die korrekte Ausführung des CoffeeMakers bezüglich der definierten Anforderung, kein Rezept mit einem identischen Namen hinzufügen zu können.

5.3.1.1 Quellcode

Der Fehler ist in Zeile 39 ersichtlich. Die Variable $exists$ wird hier auf einen inkorrekten Wert gesetzt. Das führt dazu, dass der Code in Zeile 45-51 ein Rezept mit gleichem Namen erneut hinzufügt.

Listing 5.9: Beispiel-Fault 2.1: Doppeltes Rezept aus RecipeBook.java

```
32  public synchronized boolean addRecipe(Recipe r) {
33    // Assume recipe doesn't exist in the array until find out otherwise
34    boolean exists = false;
35    // Check that recipe doesn't already exist in array
36    for (int i = 0; i < recipeArray.length; i++) {
37      if (r.equals(recipeArray[i])) {
38        // XXX: Fault4 should be exists = true
39        exists = false;
40      }
41    }
42    // Assume recipe cannot be added until find an empty spot
43    boolean added = false;
44    // Check for first empty spot in array
45    if (!exists) {
46      for (int i = 0; i < recipeArray.length && !added; i++) {
47        if (recipeArray[i] == null) {
48          recipeArray[i] = r;
49          added = true;
50        }
51      }
52    }
53    return added;
54  }
```

5.3.1.2 Erfolgreicher Testfall

Dieser Test wird manuell durchgeführt.
Testfallbeschreibung:
Es werden drei Rezepte mit unterschiedlichem Namen hinzugefügt.
Ergebnis:
In dem $makeCoffee$ Menü wird ersichtlich, dass drei verschiedene Rezepte vorhanden sind.

5.3.1.3 Fehlschlagender Testfall

Dieser Test wird manuell durchgeführt.
Testfallbeschreibung:
Es werden ebenfalls drei Rezepte hinzugefügt, wobei zwei den selben Namen haben.
Ergebnis:
Der Fehler wird in dem $makeCoffee$ Menü erkannt, da zwei Rezepte mit identischem Namen angezeigt werden.

5.3.1.4 Slicing Criterion

Die Definition eines sinnvollen Slicing Criterions ist in diesem Fall eine Herausforderung. Die Aufrufhierarchie ist in 5.1 dargestellt.

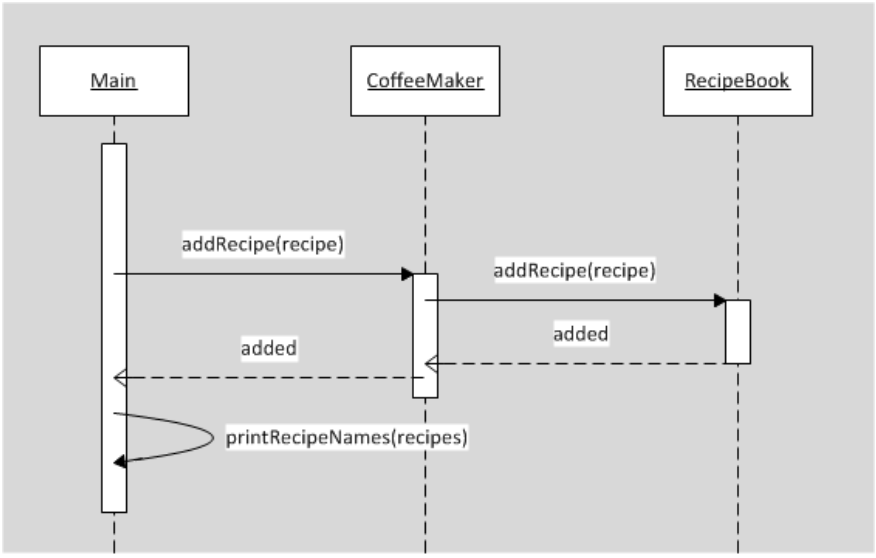

Abbildung 5.1: Ein Ausschnitt aus der Aufrufhierarchie für das Hinzufügen von Rezepten und die darauf folgende Ausgabe dieser.

Der erste denkbare Einstiegspunkt ist die Methode, in der die Rezepte dargestellt werden. Die $Main.printRecipes$ Methode eignet sich für das Detektieren dieses Fehlers jedoch nicht, da der Übergabeparameter $recipes$ nicht von dem Tracer aufgenommen wird und somit die beeinflussenden Anweisungen nicht in dem Trace auftauchen würden.

Die nächste Möglichkeit ist ein Slicing Criterion basierend auf dem Rückgabewert der $CoffeeMaker.addRecipe$, der leider ebenfalls nicht verwendet werden kann, weil die Übergabeparameter auch in diese Richtung nicht verfolgt werden. Den Aufruf der $RecipeBook.addRecipe$ Methode in der $CoffeeMaker.addRecipe$ Methode kann aus dem selben Grund ebenfalls nicht verwendet werden.

Damit bleibt nur die in Listing 5.9 dargestellte fehlerbeherbegende Methode als Definitionsbereich für das Slicing Criterion. Das erste und einfachste Slicing Criterion für das Finden des Fehlers in dieser Methode ist der Rückgabewert $return\ added$. Daraus resultiert das Slicing Criterion $edu.ncsu.csc326.coffeemaker.RecipeBook.addRecipe : 53 : \{added\}$.

5.3.1.5 Durchführung und Analyse der Fehlerlokalisierung

Durch die Wahl dieses Slicing Criterions ergibt sich folgender vereinigte Slice (sämtliche erstellten Mengen sind in Anhang A.5 ersichtlich):

Listing 5.10: Vereinigter Slice für Fault 2.1

```
1  class, method and linenumber                          |v()|code
2  |-----------------------------------------------------|---|----------------------------------------
3  edu.ncsu.csc326.coffeemaker.Main#addRecipe:67         |1.0|    String name = inputOutput("\nPlease enter the recipe name:
       ");
4  edu.ncsu.csc326.coffeemaker.Main#addRecipe:84         |1.0|    Recipe r = new Recipe();
5  edu.ncsu.csc326.coffeemaker.Main#addRecipe:86         |1.0|      r.setName(name);
6  edu.ncsu.csc326.coffeemaker.Recipe#setName:100        |1.0|    if (name != null) {
7  edu.ncsu.csc326.coffeemaker.Recipe#setName:101        |1.0|      this.name = name;
8  edu.ncsu.csc326.coffeemaker.Recipe#equals:164         |1.0|    if (this == obj)
9  edu.ncsu.csc326.coffeemaker.Recipe#equals:166         |1.0|    if (obj == null)
10 edu.ncsu.csc326.coffeemaker.Recipe#equals:168         |1.0|    if (getClass() != obj.getClass())
11 edu.ncsu.csc326.coffeemaker.Recipe#equals:170         |1.0|    final Recipe other = (Recipe) obj;
12 edu.ncsu.csc326.coffeemaker.Recipe#equals:171         |1.0|    if (name == null) {
13 edu.ncsu.csc326.coffeemaker.Recipe#equals:174         |1.0|    } else if (!name.equals(other.name))
14 edu.ncsu.csc326.coffeemaker.Recipe#equals:176         |1.0|    return true;
15 edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:37   |1.0|      if (r.equals(recipeArray[i])) {
16 edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:39   |1.0|        exists = false;
17 edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:48   |1.0|          recipeArray[i] = r;
```

Die fehlerhafte Anweisung aus Zeile 39 von Listing 5.9 ist in dem erstellten Slice vorhanden (siehe Ausgabe 5.10). Die Eingrenzung des Fehlers war hier teilweise erfolgreich. Der Fehler konnte zwar innerhalb der Methode mit dem entsprechenden Slicing Criterion gefunden werden, die Definition des Slicing Criterions konnte aber nur mit Wissen über den Quellcode und dem konkreten Fault stattfinden.
Aufgrund dieser Umstände ist die Fehlerlokalisierung für diesen Fault als nicht erfolgreich zu bewerten, da diese nicht der Motivation der Arbeit gerecht wird.

5.3.2 Beispiel-Fault 2: Inventarbestand

Hier wird ein Fault eingeführt, der zu einem inkorrekten Rückgabewert der $enoughIngredients$ Methode führt. Diese Methode dient der Überprüfung des Inventars, ob für einen zu erstellenden Kaffee ausreichende Bestände vorhanden sind. Sie wird in der $useIngredients$ Methode aufgerufen, welche wiederum vor dem Erstellen eines Kaffees von der $CoffeeMaker$ Klasse aufgerufen wird (vgl. Abbildung 5.2).

5.3.2.1 Quellcode

Die Methode $enoughIngredients$ ist mit einem Logik-Fehler behaftet (vgl. Listing 5.11). Diese Methode wird immer $false$ zurückgeben und dadurch ist kein Erstellen von Kaffee mehr

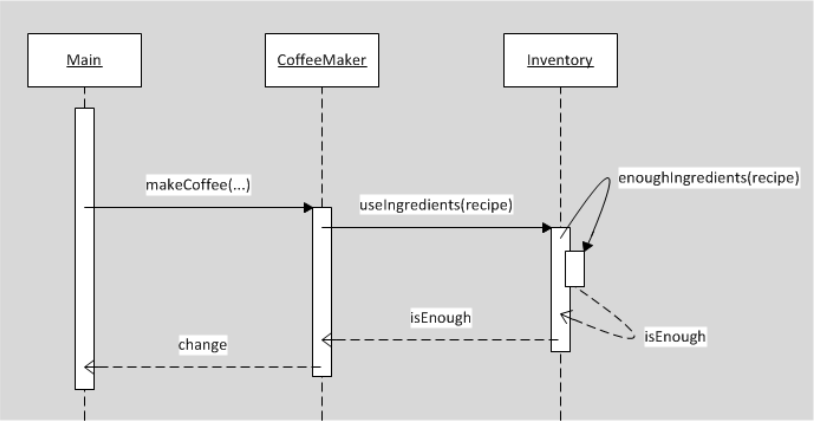

Abbildung 5.2: Auszug aus der Aufrufhierarchie für die Prüfung des Inventarbestands.

möglich. Nachdem alle Werte überprüft wurden, sollte in Zeile 194 der Wert *true* zurückgegeben werden.

Weiterhin ist zu beachten, dass diese Methode nicht zu der öffentlichen Schnittstelle des Inventars gehört, aufgrund dessen werden die Tests so definiert, dass sie die öffentliche Schnittstelle über die *useIngredients* Methode verwenden.

Listing 5.11: Beispiel-Fault 2.2: enoughIngredients aus Inventory.java

```
160  protected synchronized boolean enoughIngredients(Recipe r) {
161    boolean isEnough;
162    // @formatter:off
163    /*/
164    isEnough = true;
165    if (Inventory.coffee < r.getAmtCoffee()) {
166      isEnough = false;
167    }
168    if (Inventory.milk < r.getAmtMilk()) {
169      isEnough = false;
170    }
171    if (Inventory.sugar < r.getAmtSugar()) {
172      isEnough = false;
173    }
174    if (Inventory.chocolate < r.getAmtChocolate()) {
175      isEnough = false;
176    }
177    /*/
178    // XXX: Bug 6 Logic fault: should not return false immediately.
179    isEnough = false;
180    if (Inventory.coffee < r.getAmtCoffee()) {
181      return isEnough;
182    }
```

```
183     if (Inventory.milk < r.getAmtMilk()) {
184       return isEnough;
185     }
186     if (Inventory.sugar < r.getAmtSugar()) {
187       return isEnough;
188     }
189     if (Inventory.chocolate < r.getAmtChocolate()) {
190       return isEnough;
191     }
192     //*/
193     //@formatter:on
194     return isEnough;
195   }
```

5.3.2.2 Erfolgreicher Testfall

Dieser Test ist durch einen JUnit-Testfall definiert.
Testfallbeschreibung:
Mit diesem Testfall wird der Inventarbestand für ein Rezept, das zu viel Kaffee für den derzeitigen initialen Inventarbestand von 15 benötigt, überprüft.
Ergebnis:
Die Methode gibt wie erwartet $false$ zurück, da es nicht genug Kaffee im Automaten gibt.

Listing 5.12: JUnit Testfall aus Fault22success.java

```
17   public void testUseIngredients_moreCoffeeNeeded_shouldReturnFalse()
         throws RecipeException {
18     Recipe recipe = new Recipe();
19     recipe.setAmtChocolate("10");
20     recipe.setAmtMilk("10");
21     recipe.setAmtSugar("10");
22     recipe.setName("Standard Recipe");
23
24     recipe.setAmtChocolate("20");
25
26     assertFalse("Ingredients should be too less", inventory.
           useIngredients(recipe));
27   }
```

5.3.2.3 Fehlschlagender Testfall

Dieser Test ist durch einen JUnit-Testfall definiert.
Testfallbeschreibung:
Ein Rezept mit jeweils nur einer einzigen benötigten Einheit wird erstellt. Daraufhin wird versucht den Inventarbestand zu überprüfen.

Ergebnis:
Obwohl der initiale Inventarbestand 15 Einheiten je Zutat enthält, schlägt dieser Test, wie erwartet, fehl.

Listing 5.13: JUnit Testfall aus Fault22fail.java

```
17    public void
          testUseIngredients_lesserIngredientsNeeded_shouldReturnTrue()
          throws RecipeException {
18    Recipe recipe = new Recipe();
19    recipe.setName("Standard Recipe");
20
21    recipe.setAmtChocolate("1");
22    recipe.setAmtSugar("1");
23    recipe.setAmtCoffee("1");
24    recipe.setAmtMilk("1");
25    // Initial ingredients is 15 each, this should be enough
26    assertTrue("Ingredients should be enough", inventory.useIngredients(
          recipe));
27  }
```

5.3.2.4 Slicing Criterion

Das Slicing Criterion wird anhand des Aufrufes der $useIngredients$ Methode ausgewählt. Dabei wird die einfachste Auswahl eines Slicing Criterions vollzogen, indem der letzte Wert der Schnittstelle als Slicing Criterion definiert wird. Wie in Listing 5.16 in Zeile 220 ersichtlich, ist der Rückgabewert jedoch keine Variable, sondern $false$ wird direkt zurückgegeben. Das führt dazu, dass die Definition eines Slicing Criterions auf Basis dieses Codes nicht möglich ist, da die Ausführung besagter Zeile unter Umständen nicht stattfindet.

Aufgrund dieser Einschränkung wird eine geringfügige Modifikation der $useIngredients$ Methode vollzogen. Die Einführung einer lokalen Variablen für den Rückgabewert führt zu der in Listing 5.14 dargestellten Methode.

Listing 5.14: Die useIngredients Methode aus Inventory.java

```
203  public synchronized boolean useIngredients(Recipe r) {
204    boolean isEnough;
205    if (enoughIngredients(r)) {
206      Inventory.coffee -= r.getAmtCoffee();
207      Inventory.milk -= r.getAmtMilk();
208      Inventory.sugar -= r.getAmtSugar();
209      Inventory.chocolate -= r.getAmtChocolate();
210      isEnough = true;
211    } else {
212      isEnough = false;
213    }
```

```
214     return isEnough;
215   }
```

Diese Veränderung ist minimal invasiv, entspricht jedoch üblichen Programmierregeln und führt zu einer einfachen Definition des Slicing Criterions $edu.ncsu.csc326.coffeemaker.Inventory:214:\{isEnough\}$.

5.3.2.5 Durchführung und Analyse der Fehlerlokalisierung

Der vereinigte Slice enthält auffälligerweise zwei identische return-Anweisungen in der selben Methode. In Zeile 190 und 194 der *Inventory* Klasse wird jeweils *isEnough* zurückgegeben. Damit konnte die Fehlerursache in Zeile 194 identifiziert werden.

Listing 5.15: Vereinigter Slice für Fault 2.2

```
1 class, method and linenumber                          |v()|code
2 |-----------------------------------------------------|---|----------------------------------------
3 edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:190|0.5|      return isEnough;
4 edu.ncsu.csc326.coffeemaker.Recipe#<init>:22          |0.5|    this.amtCoffee = 0;
5 edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:194|1.0|    return isEnough;
6 edu.ncsu.csc326.coffeemaker.Recipe#setAmtCoffee:55    |1.0|    this.amtCoffee = getAmountFromString(coffee);
```

5.4 Faulttyp 3: Scheinbar richtiges Verhalten

Im Folgenden wird ein Fehler eingeführt, der unter bestimmten Eingaben verborgen bleibt.

5.4.1 Beispiel-Fault 1: Inventarberechnung

Der Fehler „Inventarberechnung“ überschreibt das Inventar mit den Werten der zu verbrauchenden Kaffeezutaten (Kaffee, Milch, Zucker und Schokolade) aus dem Rezept. Das Ausführen eines einzelnen Testfalles, bei dem das Rezept die Hälfte der im Inventar vorhandenen Zutaten verwendet, kann den Fehler nicht offenbaren. Das Verhalten wäre scheinbar richtig. Für die nähere Erläuterung wird ein Beispiel angeführt: Angenommen im Inventar sind 20 Einheiten Kaffee enthalten, und ein Rezept benötigt 10 Einheiten. Der korrekte Inhalt des Inventars wäre nach dem Brühen des Kaffees 10 Einheiten Kaffee. Ein identischer Testfall mit besagtem Fehler würde ebenfalls zu 10 verbleibenden Einheiten führen. Dieser Fehler würde unerkannt bleiben und mit der Ausführung anderer Werte auffallen, indem eine Differenz zwischen tatsächlichem und berechnetem Bestand erkannt wird.

5.4.1.1 Quellcode

Bei dem Vergleich von Zeilen 207-210 mit 212-215 fällt auf, dass ein vergessenes Minuszeichen dazu führt, dass der ursprüngliche Wert des Inventars mit den Werten des Rezeptes überschrieben wird.

Listing 5.16: Beispiel-Fault 3.1: Inventarberechnung aus CoffeeMaker.java

```
203  public synchronized boolean useIngredients(Recipe r) {
204    if (enoughIngredients(r)) {
205      // @formatter:off
206      //*/ XXX: Fault 3.1: Inventory  is not calculated correctly.
207      Inventory.coffee = r.getAmtCoffee();
208      Inventory.milk = r.getAmtMilk();
209      Inventory.sugar = r.getAmtSugar();
210      Inventory.chocolate = r.getAmtChocolate();
211      /*/
212      Inventory.coffee -= r.getAmtCoffee();
213      Inventory.milk -= r.getAmtMilk();
214      Inventory.sugar -= r.getAmtSugar();
215      Inventory.chocolate -= r.getAmtChocolate();
216      //*/
217      // @fromatter:on
218      return true;
219    } else {
220      return false;
221    }
222  }
```

5.4.1.2 Erfolgreicher Testfall

Der erfolgreiche Testfall ist für diesen Fault nicht wirklich erfolgreich. Es scheint nur so, als wäre er erfolgreich.
Dieser Test wird manuell durchgeführt.
Testfallbeschreibung:
Der CoffeeMaker hat einen initialen Inventarbestand von 15 Einheiten für alle Zutaten. Daraufhin werden 5 Einheiten für alle Zutaten hinzugefügt, ein Rezept mit jeweils 10 benötigten Einheiten hinzugefügt und anschließend dieses Rezept gekauft.
Ergebnis:
Der CoffeeMaker erstellt den dem Rezept entsprechenden Kaffee, und die folgende Überprüfung des Inventars offenbart einen korrekten Inventarbestand.

5.4.1.3 Fehlschlagender Testfall

Dieser Test wird manuell durchgeführt.
Testfallbeschreibung:
Bei dem fehlschlagenden Testfall wird dem CoffeeMaker ein Rezept mit jeweils 5 benötigten Zutaten hinzugefügt. Der initiale Inventarbestand ist hier ebenfalls 15 pro Zutat, es wird kein zusätzliches Inventar hinzugefügt.
Ergebnis:

Der Kauf des besagten Rezeptes führt durch die direkte Zuweisung der Rezeptangaben zu dem fehlerhaften Inventarbestand von 5 je Zutat.

5.4.1.4 Slicing Criterion

Die Wahl des Slicing Criterions ähnelt der in Kapitel 5.3. Begonnen mit der höchsten Aufruf-Hierarchie führt eine schrittweise Näherung zu der fehlerbehafteten Methode. Weiterhin kann hier ebenfalls in dieser Methode kein Criterion gefunden werden. Das hängt mit der Struktur der Methode und der Unzulänglichkeit des JavaSlicers zusammen. Die Methode selbst enthält keinen Wert, der am Ende der Methode so beobachtet werden könnte, dass die eingebrachten Fehler einen Einfluss auf diesen hätten. Weiterhin sind keine lokalen Variablen enthalten. Das ist auch der Grund, warum der JavaSlicer hier keinen Slice erstellen könnte, da Parameter nicht von dem Tracer verfolgt werden und Instanzvariablen ebenso nicht. Es werden ausschließlich lokale Variablen und Kontrollflüsse verfolgt.

5.4.1.5 Durchführung und Analyse der Fehlerlokalisierung

Die Durchführung mit diversen Slicing Criterions führte immer zu einem leeren Ergebnis. Eine Analyse dieser kann hier nicht stattfinden und das Ergebnis wird als negativ bewertet. Auch mit Hilfe von JUnit-basierten Tests kann hier keine Fehlerlokalisierung stattfinden.

5.5 Auswertung der Demonstration von Fehlerlokalisierung und Fehlereingrenzung am CoffeeMaker-Beispiel

Die Reduktion der Komplexität ist zwar in kleinem Maße (innerhalb einer fehlerbehafteten Methode) erfolgreich gewesen, die Komplexität größerer Software-Architekturen wird sich in der Regel aber nicht über komplexe Methoden, sondern über komplexe Module bzw. Komponenten definieren. Aus diesem Grund ist mit Hilfe der Eingrenzung des Fehlers in einer Methode nur eine beschränkte Reduktion der Komplexität gewonnen.
Das, die Fehlerlokalisierung nur erfolgreich innerhalb einer Methode durchgeführt werden kann, liegt an dem fehlenden Feature des JavaSlicers auch Parameter tracen zu können.

Die Erstellung der Testfälle ist, wie die Demonstration mit den manuellen Testfällen zeigen konnte, nicht nach der in Kapitel 4.2 beschriebenen Herangehensweise durchführbar. Die Fehlerlokalisierung kann bei Fehlern, die ausschließlich auf einer fehlerhaften Berechnung stattfinden, nicht erfolgreich sein, wenn sich die Tests sehr ähnlich sind. Aufgrund dessen sollten ebenfalls Tests verwendet werden, die einen anderen Kontrollfluss als der Fehlgeschlagene haben. Dadurch können die fehlerhaften Anweisungen dann wiederum erkannt werden.

Aufgrund der erkannten Einschränkungen des JavaSlicers wurden ebenfalls Testfälle auf JUnit-Basis ausgeführt. Diese Komponententests konnten in beiden Fällen auf die Fehlerursache aufmerksam machen. Die Instrumentierung von JUnit-Tests ist somit als Erfolg zu bewerten.

Kapitel 6

Schluss

Dieses Kapitel gibt einen Überblick über die vorliegende Arbeit. Zuerst werden in der Zusammenfassung die Kapitel und deren Erkenntnisse noch einmal komprimiert dargestellt. Daraufhin findet eine Bewertung der erarbeiteten Ansätze und Lösungskonzepte in dem Fazit statt. Abschließend wird ein Ausblick auf mögliche Weiterentwicklungen von Fehlerlokalisierung und Fehlereingrenzung mit Bezug auf diese Arbeit gegeben.

6.1 Zusammenfassung

Das Ziel der Arbeit war das Eingrenzen bzw. Lokalisieren von Fehlern in Javaprogrammen mit Client-Server-Architekturen. Dazu wurden in Kapitel 2 die Grundlagen für das Anwendungsfeld und die Fault Localization Methode erarbeitet. Insbesondere hat dort eine Abgrenzung des Anwendungsfeldes stattgefunden. Weiterhin ist die wichtige Identifikation der für diese Arbeit relevanten Failure Modes geleistet worden. Zudem wurden in diesem Kapitel die verwendeten Tools erläutert.

Nachdem in Kapitel 3 auf andere Arbeiten geschaut wurde, ist auf Basis dieser, aus der Kombination von Herangehensweisen, ein Konzept für die in den Grundlagen erarbeiteten Failure Modes erstellt worden. Die Kombination beruht im Kern auf dem Bewerten von verdächtigen Codezeilen (vgl. Jones und Harrold (2005)) und einer Kombination der Methoden *Set Union* und *Set Intersection* (vgl. Agrawal u. a. (1995)). Das daraus entstandene Konzept sieht die Ausführung von mindestens einem fehlschlagenden und einem erfolgreichen Test vor. Daraufhin wird mittels Mengenoperationen auf den entstandenen Slices jede Codezeile mit einem Verdachtswert belegt und dem Entwickler präsentiert.

Das Kapitel 5 stellt das erarbeitete Konzept unter Verwendung eines praktischen Beispiels auf die Probe. Dort wurden diverse, den Failure Modes entsprechende, Faults und dazugehörige Testfälle ermittelt, um die Möglichkeiten des Konzeptes mit Blick auf das Ziel der Fehlerlokalisierung bzw. Fehlereingrenzung auszuwerten.

6.2 Gewonnene Erkenntnisse

Während der Durchführung der Beispiele in Kapitel 5 hat sich herausgestellt, dass eine Verfolgung des Kontrollflusses ohne die Betrachtung der Werte von Variablen nicht ausreicht, um alle identifizierten Faults erkennen zu können. Programmabläufe, an denen der Kontrollfluss durch Variablen beeinflusst wird, werden korrekt verfolgt. Sobald jedoch bspw. eine Berechnung inkorrekte Werte erzeugt, kann der Ursprung nicht mehr eindeutig identifiziert werden, weil unter Umständen, die selben Berechnungen in einem identischen Programmablauf ausgeführt werden. Das führt wiederum dazu, dass sich die erstellten Slices vollständig überlagern und die Mengenoperationen zu leeren Mengen führen.

Weitere gewonnene Erkenntisse sind die Einschränkungen des Javaslicers, die dazu führen, Variablen über Methodengrenzen hinweg nicht verfolgen zu können, und das Nichterkennen von mehrfach ausgeführten Anweisungen innerhalb eines Programmablaufes. Aufgrund letzterem kann ein Fehler, der wegen von zu häufiger oder zu seltener Ausführung bestimmter Anweisungen beruht, nicht erkannt werden.

Die Annahme, die Anwendung könne über das Bereitstellen einer Schnittstelle wie eine Webapplikation getestet werden, führt zu der Nichtbetrachtung der für JEE-Applikationen üblichen Verwendung von Übergabewerten[23]. Dieses einheitliche Verhalten kann zu einer Vereinfachung der Erstellung von Testfällen bzw. Slicing Criterions führen.

Das Vereinfachen bzw. der Ausschluss des Browsers für die Fehleranalyse hat sich bewährt, da Frontendtechnologien (wie Javascript, ActiveX etc.) durch eine fehlerfreie Übertragung der Netzwerkschicht und der Zusicherung von korrekten Werten an der Serverschnittstelle gekapselt betrachtet werden können.

6.3 Fazit

Dem in der Einführung beschriebenen Ziel der Eingrenzung von Fehlern in Java-Programmen konnte die Demonstration nicht vollständig gerecht werden.
Das hängt primär mit der Verwendung des Javaslicers zusammen. Dieser konnte für den der Arbeit zu Grunde liegenden Anwendungsfall keine zufriedenstellenden Traces und somit auch keine aussagekräftigen Slices erstellen, da Übergabeparameter nicht verfolgt werden. Eine übergreifende Erweiterung der Analyse auf Komponenten- und Modulebene war somit nicht möglich.

[23] Hier werden i.d.R. Response- und Request-Objekte während eines Request-Lebenszyklusses verwendet und mit Daten befüllt.

Die Auswahl der Slicing Criterions hat sich aufgrund der Beschränkungen des Javaslicers als schwierig gezeigt. Das Ziel, auf hoher Abstraktionsebene Einstiegspunkte zu finden, konnte nicht erreicht werden. Die Ursache hierfür liegt ebenfalls an dem fehlenden Feature des Javaslicers, Parameter verfolgen zu können.

Eine weitere Einschränkung des Slicers lag darin, mehrfache Ausführungen nicht zusätzlich in den Slice mitaufzunehmen. Dieses Verhalten führte bei Fault 1.1 dazu, dass die Fehlerursache nicht in dem Slice aufgetreten ist.

Das Konzept kann muss der Demonstration ebenfalls kritisch betrachtet werden.
Innerhalb einer Methode konnte jedoch gezeigt werden, dass das Eingrenzen von verdächtigen Anweisungen lokal erfolgreich sein kann.

Die Auswahl der Testfälle, erläutert in Kapitel 4.2, hat nicht das erwartete Erfolgsverhalten zeigen können. Das liegt primär daran, dass sich die Testfälle oft zu ähnlich waren, bzw. die geringe Komplexität des CoffeeMakers in Kombination mit den sich ähnelnden Testfällen identische Ausführungspfade zeigte. Identische Ausführungspfade können ohne Betrachtung der Variablenwerte nicht zu hilfreichen Ergebnismengen führen, da diese leer sind. Die zusätzliche Betrachtung von Variablenwerten hätte in Betracht gezogen werden können, hätte jedoch auch zu deutlich erhöhter Komplexität geführt.

Das Konzept ist somit zumindest teilweise als zielführend zu bezeichnen.

Die Fehlerlokalisierung für Laufzeitfehler, bei denen Java-Exceptions geworfen werden, kann unter Beachtung von Designentscheidungen bzw. Veränderungen des Quellcodes auch mit dem verwendeten Javaslicer erfolgreich sein.

Sämtliche Methoden sollten eine lokale Variable als Rückgabewert haben, um dem Tracer zu ermöglichen die Werte innerhalb von Methoden verfolgen zu können.

Es sollten keine direkten Zugriffe auf Instanzvariablen stattfinden, da diese nicht von dem Tracer verfolgt werden.

Weiterhin sind Methoden mit mehrfachen return-Anweisungen zu vermeiden, da ansonsten kein Slicing Criterion definiert werden kann, welches auch bei diversen Programmläufen erreicht wird.

6.4 Weiterführende Arbeiten

Die Demonstration hat gezeigt, dass ein Slicer, der Übergabeparameter nicht verfolgt, für die hier angestrebte erfolgreiche Fehlerlokalisierung nicht ausreicht. Weiterhin hätte, wie bei Fault 5.2.1 gezeigt, eine Analyse auf Basis der Häufigkeit der Ausführung bestimmter Anweisungen hilfreich sein können, um Fehler zu identifizieren, die durch zu häufige bzw. zu seltene Ausführung einzelner Befehle entstehen. Eine mögliche weiterführende Erforschung

dieses Themengebietes könnte somit die Entwicklung eines Slicers sein, der diese Kriterien erfüllt. Die Weiterentwicklung des Javaslicers sollte dabei unbedingt in Betracht gezogen werden, da dieser gut bedienbar ist und schnelle Ausführungszeiten erreicht.

Die Kombination des Slicers mit Daten aus Code Coverage Werkzeugen ist zumindest für das Erkennen von mehrfach ausgeführten Anweisungen ebenfalls denkbar. In der Praxis verwendete Code Coverage Werkzeuge[24] annotieren die jeweilige Zeile bereits mit der Anzahl der Ausführung. Eine Nutzung solcher Werkzeuge und das Anreichern der Traces bzw. der Slices steht nach Schätzung des Autors in vertretbarem Rahmen und kann erheblichen Mehrwert bringen.

In Kapitel 5.5 wurde eine Problematik erkannt, die durch zu ähnliche Testfälle entstanden ist. Es kam in der Demonstration vor, dass die gewählten Mengenoperationen auf den Slices bei sehr ähnlichen Ausführungen des Programmes zu leeren Ergebnismengen führten. Eine Distanzierung der Testfälle kann wiederum dazu führen, dass die Ergebnismengen zu groß werden und kein Vorteil für den Entwickler entsteht. Es kann aufgrund dessen in Betracht gezogen werden, eine Distanzfunktion für die Unterstützung der Auswahl geeigneter Testfälle heranzuziehen. Die Einflussnahme auf eine solche Distanzfunktion durch eine skalierende Variable sollte dabei unbedingt miteinbezogen werden, um dynamisch auf zu große bzw. zu kleine Ergebnismengen zu reagieren. Methoden für die Berechnung von Distanzen zwischen Testfällen werden in Renieris und Reiss (2003) und Cleve und Zeller (2005) vorgestellt.
Weiterhin kann die Evaluierung des gewählten Verfahrens auf deutlich komplexere Systeme in Betracht gezogen werden. Es ist möglich, dass sich ähnelnde Testfälle in komplexeren Systemen zu Ausführungspfaden mit ausreichend Distanz führen.

In Verbindung mit der zuvor genannten Einführung einer Distanzfunktion sollte als weitere Möglichkeit für eine bessere Skalierung die Aufnahme mehrerer erfolgreicher Tests für die Erstellung der Ergebnismenge diskutiert werden (vgl. Jones und Harrold (2005)).
Eine zusätzliche Vertiefung kann durch eine weiterführende Analyse der Failure Modes für Laufzeitfehler mit Java Exceptions ergeben.
Zur realistischen Bewertung dieses Ansatzes, sollte dieser durch zuvor genannte Erweiterungen verfeinert werden und daraufhin mit anderen Fehlerlokalisierungsmethoden verglichen werden.

Diese Arbeit hat keine Fehler auf Seiten des Clients betrachtet. Fehler, die in dem Browser, in Browser-Addons, in Browser-Frontendtechnologien oder in Anwendungen, die den Browser zum Download von Code und Daten verwenden, wurden nicht betrachtet. Diese

[24] vgl. http://emma.sourceforge.net/ und http://cobertura.sourceforge.net/

können nach dem Teile und Herrsche[25] Lösungsansatz durch Invertierung der Annahme, der Browser sei fehlerfrei und der Server evtl. fehlerbehaftet, isoliert betrachtet werden.

[25] engl.: divide and conquer

Anhang A

Anhang

A.1 HTTP/1.1-Statuscodes

```
1xx: Informational
  100: Continue
  101: Switching Protocols

2xx: Successful
  200: OK
  201: Created
  202: Accepted
  203: Non-Authoritative Information
  204: No Content
  205: Reset Content
  206: Partial Content

3xx: Redirection
  300: Multiple Choices
  301: Moved Permanently
  302: Found
  303: See Other
  304: Not Modified
  305: Use Proxy
  307: Temporary Redirect

4xx: Client Error
  400: Bad Request
  401: Unauthorized\
  402: Payment Required
  403: Forbidden
  404: Not Found
  405: Method Not Allowed
  406: Not Acceptable
  407: Proxy Authentication Required
  408: Request Time-out
  409: Conflict
  410: Gone
  411: Length Required
  412: Precondition Failed
  413: Request Entity Too Large
  414: Request-URI Too Large
  415: Unsupported Media Type
  416: Requested range not satisfiable
  417: Expectation Failed

5xx: Server Error
  500: Internal Server Error
  501: Not Implemented
  502: Bad Gateway
  503: Service Unavailable
  504: Gateway Time-out
  505: HTTP Version not supported
```

A.2 Diagramme

A.2.1 Flussdiagramm des JSliceParsers

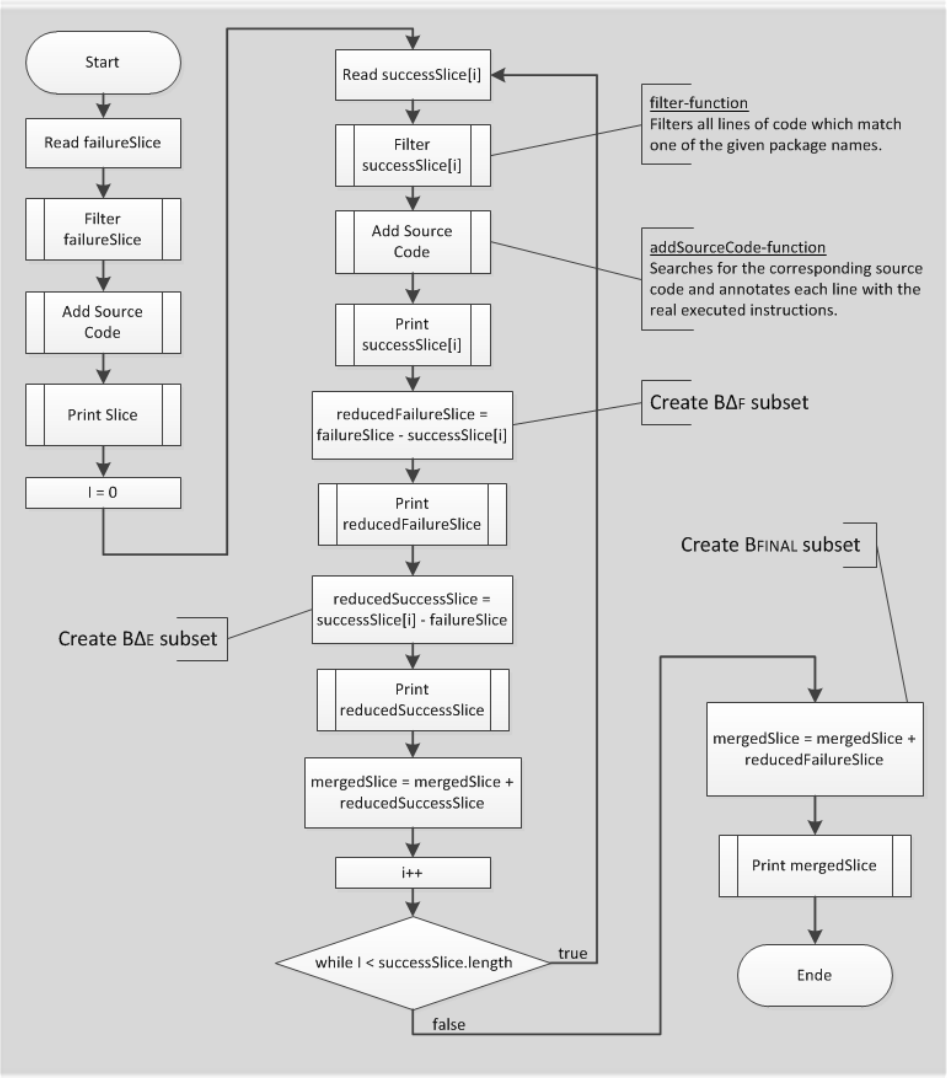

Abbildung A.1: Der Programmfluss des JSliceParsers für das Filtern, Hinzufügen von Source Code und die Mengenoperationen.

A.3 Slices

A.3.1 Fault 1.1

Listing A.1: Slicing Criterion: edu.ncsu.csc326.coffeemaker.Main.printRecipeNames:254

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building JSliceParser 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser ---
**************************Failure Slice**************************
################################################################################################################
Generating slice representation for slice: e:/dev/dev/workspace/CoffeeMaker_FIT/reports/Fault1_fail.slice with package filters: edu.ncsu
################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.CoffeeMaker#<init>:19     |0.0|    recipeBook = new RecipeBook();
edu.ncsu.csc326.coffeemaker.CoffeeMaker#getRecipes:114|0.0|    return recipeBook.getRecipes();
edu.ncsu.csc326.coffeemaker.Main#mainMenu:34          |0.0|      int userInput = Integer.parseInt(inputOutput("Please press the number that corresponds to what you would like the coffee maker to do."));
edu.ncsu.csc326.coffeemaker.Main#mainMenu:36          |0.0|      if (userInput >= 0 && userInput <= 6) {
edu.ncsu.csc326.coffeemaker.Main#mainMenu:37          |0.0|        if (userInput == 1)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:38          |0.0|          addRecipe();
edu.ncsu.csc326.coffeemaker.Main#mainMenu:41          |0.0|        if (userInput == 3)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:42          |0.0|          editRecipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:103        |0.0|      mainMenu();
edu.ncsu.csc326.coffeemaker.Main#editRecipe:134       |0.0|    Recipe[] recipes = coffeeMaker.getRecipes();
edu.ncsu.csc326.coffeemaker.Main#editRecipe:135       |0.0|    printRecipeNames(recipes);
edu.ncsu.csc326.coffeemaker.Main#printRecipeNames:250 |0.0|    for (int i = 0; i < recipes.length; i++) {
edu.ncsu.csc326.coffeemaker.Main#printRecipeNames:253 |0.0|        System.out.println((i + 1) + ". " + recipes[i].getName());
edu.ncsu.csc326.coffeemaker.Main#inputOutput:272      |0.0|    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
edu.ncsu.csc326.coffeemaker.Main#inputOutput:275      |0.0|      returnString = br.readLine();
edu.ncsu.csc326.coffeemaker.Main#inputOutput:280      |0.0|    return returnString;
edu.ncsu.csc326.coffeemaker.Main#main:313             |0.0|    coffeeMaker = new CoffeeMaker();
edu.ncsu.csc326.coffeemaker.Main#main:315             |0.0|    mainMenu();
**************************Success Slice**************************
################################################################################################################
Generating slice representation for slice: e:/dev/dev/workspace/CoffeeMaker_FIT/reports/Fault1_success.slice with package filters: edu.ncsu
################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.CoffeeMaker#<init>:19     |0.0|    recipeBook = new RecipeBook();
edu.ncsu.csc326.coffeemaker.CoffeeMaker#getRecipes:114|0.0|    return recipeBook.getRecipes();
edu.ncsu.csc326.coffeemaker.Main#mainMenu:34          |0.0|      int userInput = Integer.parseInt(inputOutput("Please press the number that corresponds to what you would like the coffee maker to do."));
edu.ncsu.csc326.coffeemaker.Main#mainMenu:36          |0.0|      if (userInput >= 0 && userInput <= 6) {
edu.ncsu.csc326.coffeemaker.Main#mainMenu:37          |0.0|        if (userInput == 1)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:38          |0.0|          addRecipe();
edu.ncsu.csc326.coffeemaker.Main#mainMenu:41          |0.0|        if (userInput == 3)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:42          |0.0|          editRecipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:103        |0.0|      mainMenu();
edu.ncsu.csc326.coffeemaker.Main#editRecipe:134       |0.0|    Recipe[] recipes = coffeeMaker.getRecipes();
edu.ncsu.csc326.coffeemaker.Main#editRecipe:135       |0.0|    printRecipeNames(recipes);
edu.ncsu.csc326.coffeemaker.Main#printRecipeNames:250 |0.0|    for (int i = 0; i < recipes.length; i++) {
edu.ncsu.csc326.coffeemaker.Main#printRecipeNames:253 |0.0|        System.out.println((i + 1) + ". " + recipes[i].getName());
edu.ncsu.csc326.coffeemaker.Main#inputOutput:272      |0.0|    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
edu.ncsu.csc326.coffeemaker.Main#inputOutput:275      |0.0|      returnString = br.readLine();
edu.ncsu.csc326.coffeemaker.Main#inputOutput:280      |0.0|    return returnString;
edu.ncsu.csc326.coffeemaker.Main#main:313             |0.0|    coffeeMaker = new CoffeeMaker();
edu.ncsu.csc326.coffeemaker.Main#main:315             |0.0|    mainMenu();
**************************sum(failureSlice) - sum(successSlice) ->>****************************
################################################################################################################
Generating slice representation for slice: Reduced Failure Slice with package filters: edu.ncsu
################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
**************************sum(successSlice) - sum(failureSlice) ->>****************************
```

```
################################################################################################################
Generating slice representation for slice: Reduced Success Slice with package filters: edu.ncsu
################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
**************************Merged and Weighted Slice**************************
################################################################################################################
Generating slice representation for slice: mergedSlice with package filters: edu.ncsu
################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
Execution took: 54.0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.700s
[INFO] Finished at: Wed May 23 21:27:01 CEST 2012
[INFO] Final Memory: 5M/79M
[INFO] ------------------------------------------------------------------------
```

A.3.2 Fault 1.1.1

Listing A.2: Slicing Criterion: edu.ncsu.csc326.coffeemaker.Main.printRecipeNames:254

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building JSliceParser 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser ---
****************************Failure Slice*****************************
################################################################################################################
Generating slice representation for slice: e:/dev/dev/workspace/CoffeeMaker_FIT/reports/Fault1_1_fail.slice with package
      filters: edu.ncsu
################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.CoffeeMaker#<init>:19     |0.0|    recipeBook = new RecipeBook();
edu.ncsu.csc326.coffeemaker.CoffeeMaker#getRecipes:114|0.0|    return recipeBook.getRecipes();
edu.ncsu.csc326.coffeemaker.Main#mainMenu:34          |0.0|      int userInput = Integer.parseInt(inputOutput("Please
      press the number that corresponds to what you would like the coffee maker to do."));
edu.ncsu.csc326.coffeemaker.Main#mainMenu:36          |0.0|      if (userInput >= 0 && userInput <= 6) {
edu.ncsu.csc326.coffeemaker.Main#mainMenu:41          |0.0|        if (userInput == 3)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:42          |0.0|          editRecipe();
edu.ncsu.csc326.coffeemaker.Main#editRecipe:134       |0.0|    Recipe[] recipes = coffeeMaker.getRecipes();
edu.ncsu.csc326.coffeemaker.Main#editRecipe:135       |0.0|    printRecipeNames(recipes);
edu.ncsu.csc326.coffeemaker.Main#printRecipeNames:250 |0.0|    for (int i = 0; i < recipes.length; i++) {
edu.ncsu.csc326.coffeemaker.Main#printRecipeNames:253 |0.0|        System.out.println((i + 1) + ". " + recipes[i].getName
      ());
edu.ncsu.csc326.coffeemaker.Main#inputOutput:272      |0.0|    BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in));
edu.ncsu.csc326.coffeemaker.Main#inputOutput:275      |0.0|      returnString = br.readLine();
edu.ncsu.csc326.coffeemaker.Main#inputOutput:280      |0.0|    return returnString;
edu.ncsu.csc326.coffeemaker.Main#main:313             |0.0|    coffeeMaker = new CoffeeMaker();
edu.ncsu.csc326.coffeemaker.Main#main:315             |0.0|    mainMenu();
****************************Success Slice*****************************
################################################################################################################
Generating slice representation for slice: e:/dev/dev/workspace/CoffeeMaker_FIT/reports/Fault1_1_success.slice with
      package filters: edu.ncsu
################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.CoffeeMaker#<init>:19     |0.0|    recipeBook = new RecipeBook();
edu.ncsu.csc326.coffeemaker.CoffeeMaker#getRecipes:114|0.0|    return recipeBook.getRecipes();
edu.ncsu.csc326.coffeemaker.Main#mainMenu:34          |0.0|      int userInput = Integer.parseInt(inputOutput("Please
      press the number that corresponds to what you would like the coffee maker to do."));
edu.ncsu.csc326.coffeemaker.Main#mainMenu:36          |0.0|      if (userInput >= 0 && userInput <= 6) {
edu.ncsu.csc326.coffeemaker.Main#mainMenu:37          |0.0|        if (userInput == 1)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:38          |0.0|          addRecipe();
edu.ncsu.csc326.coffeemaker.Main#mainMenu:41          |0.0|        if (userInput == 3)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:42          |0.0|          editRecipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:103        |0.0|      mainMenu();
edu.ncsu.csc326.coffeemaker.Main#editRecipe:134       |0.0|    Recipe[] recipes = coffeeMaker.getRecipes();
edu.ncsu.csc326.coffeemaker.Main#editRecipe:135       |0.0|    printRecipeNames(recipes);
edu.ncsu.csc326.coffeemaker.Main#editRecipe:137       |0.0|    int recipeToEdit = recipeListSelection("Please select the
      number of the recipe to edit.");
```

```
edu.ncsu.csc326.coffeemaker.Main#editRecipe:139       |0.0|    if (recipeToEdit <= 0) {
edu.ncsu.csc326.coffeemaker.Main#editRecipe:140       |0.0|      mainMenu();
edu.ncsu.csc326.coffeemaker.Main#printRecipeNames:250 |0.0|    for (int i = 0; i < recipes.length; i++) {
edu.ncsu.csc326.coffeemaker.Main#printRecipeNames:253 |0.0|        System.out.println((i + 1) + ". " + recipes[i].getName
      ());
edu.ncsu.csc326.coffeemaker.Main#inputOutput:272      |0.0|    BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in));
edu.ncsu.csc326.coffeemaker.Main#inputOutput:275      |0.0|      returnString = br.readLine();
edu.ncsu.csc326.coffeemaker.Main#inputOutput:280      |0.0|    return returnString;
edu.ncsu.csc326.coffeemaker.Main#main:313             |0.0|    coffeeMaker = new CoffeeMaker();
edu.ncsu.csc326.coffeemaker.Main#main:315             |0.0|    mainMenu();
**************************sum(failureSlice) - sum(successSlice) ->>**************************
########################################################################################################################
Generating slice representation for slice: Reduced Failure Slice with package filters: edu.ncsu
########################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
**************************sum(successSlice) - sum(failureSlice) ->>**************************
########################################################################################################################
Generating slice representation for slice: Reduced Success Slice with package filters: edu.ncsu
########################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.Main#mainMenu:37           |0.5|        if (userInput == 1)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:38           |0.5|          addRecipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:103         |0.5|      mainMenu();
edu.ncsu.csc326.coffeemaker.Main#editRecipe:137        |0.5|    int recipeToEdit = recipeListSelection("Please select the
      number of the recipe to edit.");
edu.ncsu.csc326.coffeemaker.Main#editRecipe:139        |0.5|    if (recipeToEdit <= 0) {
edu.ncsu.csc326.coffeemaker.Main#editRecipe:140        |0.5|      mainMenu();
**************************Merged and Weighted Slice**************************
########################################################################################################################
Generating slice representation for slice: mergedSlice with package filters: edu.ncsu
########################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.Main#mainMenu:37           |0.5|        if (userInput == 1)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:38           |0.5|          addRecipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:103         |0.5|      mainMenu();
edu.ncsu.csc326.coffeemaker.Main#editRecipe:137        |0.5|    int recipeToEdit = recipeListSelection("Please select the
      number of the recipe to edit.");
edu.ncsu.csc326.coffeemaker.Main#editRecipe:139        |0.5|    if (recipeToEdit <= 0) {
edu.ncsu.csc326.coffeemaker.Main#editRecipe:140        |0.5|      mainMenu();
Execution took: 60.0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.715s
[INFO] Finished at: Wed May 23 21:30:55 CEST 2012
[INFO] Final Memory: 5M/80M
[INFO] ------------------------------------------------------------------------
```

A.3.3 Fault 1.2

Listing A.3: Slicing Criterion: edu.ncsu.csc326.coffeemaker.RecipeBook.deleteRecipe:66

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building JSliceParser 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser ---
**************************Failure Slice**************************
########################################################################################################################
Generating slice representation for slice: e:/dev/dev/workspace/CoffeeMaker_FIT/reports/Fault2_a_fail.slice with package
      filters: edu.ncsu
########################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.CoffeeMaker#<init>:19      |0.0|    recipeBook = new RecipeBook();
edu.ncsu.csc326.coffeemaker.CoffeeMaker#deleteRecipe:42|0.0|    return recipeBook.deleteRecipe(recipeToDelete - 1);
edu.ncsu.csc326.coffeemaker.Main#mainMenu:34           |0.0|      int userInput = Integer.parseInt(inputOutput("Please
      press the number that corresponds to what you would like the coffee maker to do."));
edu.ncsu.csc326.coffeemaker.Main#mainMenu:36           |0.0|      if (userInput >= 0 && userInput <= 6) {
edu.ncsu.csc326.coffeemaker.Main#mainMenu:37           |0.0|        if (userInput == 1)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:38           |0.0|          addRecipe();
```

```
edu.ncsu.csc326.coffeemaker.Main#mainMenu:39          |0.0|        if (userInput == 2)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:40          |0.0|          deleteRecipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:103        |0.0|      mainMenu();
edu.ncsu.csc326.coffeemaker.Main#deleteRecipe:114     |0.0|    int recipeToDelete = recipeListSelection("Please select
     the number of the recipe to delete.");
edu.ncsu.csc326.coffeemaker.Main#deleteRecipe:120     |0.0|    String recipeDeleted = coffeeMaker.deleteRecipe(
     recipeToDelete);
edu.ncsu.csc326.coffeemaker.Main#inputOutput:272      |0.0|    BufferedReader br = new BufferedReader(new
     InputStreamReader(System.in));
edu.ncsu.csc326.coffeemaker.Main#inputOutput:275      |0.0|      returnString = br.readLine();
edu.ncsu.csc326.coffeemaker.Main#inputOutput:280      |0.0|    return returnString;
edu.ncsu.csc326.coffeemaker.Main#recipeListSelection:291|0.0|    String userSelection = inputOutput(message);
edu.ncsu.csc326.coffeemaker.Main#recipeListSelection:294|0.0|      recipe = Integer.parseInt(userSelection);
edu.ncsu.csc326.coffeemaker.Main#recipeListSelection:304|0.0|    return recipe;
edu.ncsu.csc326.coffeemaker.Main#main:313             |0.0|    coffeeMaker = new CoffeeMaker();
edu.ncsu.csc326.coffeemaker.Main#main:315             |0.0|    mainMenu();
edu.ncsu.csc326.coffeemaker.RecipeBook#<init>:19      |0.0|    recipeArray = new Recipe[NUM_RECIPES];
**************************Success Slice**************************
####################################################################################################################
Generating slice representation for slice: e:/dev/dev/workspace/CoffeeMaker_FIT/reports/Fault2_a_success.slice with
     package filters: edu.ncsu
####################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.CoffeeMaker#<init>:19     |0.0|    recipeBook = new RecipeBook();
edu.ncsu.csc326.coffeemaker.CoffeeMaker#addRecipe:31  |0.0|    return recipeBook.addRecipe(r);
edu.ncsu.csc326.coffeemaker.CoffeeMaker#deleteRecipe:42|0.0|    return recipeBook.deleteRecipe(recipeToDelete - 1);
edu.ncsu.csc326.coffeemaker.Main#mainMenu:34          |0.0|      int userInput = Integer.parseInt(inputOutput("Please
     press the number that corresponds to what you would like the coffee maker to do."));
edu.ncsu.csc326.coffeemaker.Main#mainMenu:36          |0.0|      if (userInput >= 0 && userInput <= 6) {
edu.ncsu.csc326.coffeemaker.Main#mainMenu:37          |0.0|        if (userInput == 1)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:38          |0.0|          addRecipe();
edu.ncsu.csc326.coffeemaker.Main#mainMenu:39          |0.0|        if (userInput == 2)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:40          |0.0|          deleteRecipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:84         |0.0|    Recipe r = new Recipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:93         |0.0|      boolean recipeAdded = coffeeMaker.addRecipe(r);
edu.ncsu.csc326.coffeemaker.Main#addRecipe:103        |0.0|      mainMenu();
edu.ncsu.csc326.coffeemaker.Main#deleteRecipe:114     |0.0|    int recipeToDelete = recipeListSelection("Please select
     the number of the recipe to delete.");
edu.ncsu.csc326.coffeemaker.Main#deleteRecipe:120     |0.0|    String recipeDeleted = coffeeMaker.deleteRecipe(
     recipeToDelete);
edu.ncsu.csc326.coffeemaker.Main#inputOutput:272      |0.0|    BufferedReader br = new BufferedReader(new
     InputStreamReader(System.in));
edu.ncsu.csc326.coffeemaker.Main#inputOutput:275      |0.0|      returnString = br.readLine();
edu.ncsu.csc326.coffeemaker.Main#inputOutput:280      |0.0|    return returnString;
edu.ncsu.csc326.coffeemaker.Main#recipeListSelection:291|0.0|    String userSelection = inputOutput(message);
edu.ncsu.csc326.coffeemaker.Main#recipeListSelection:294|0.0|      recipe = Integer.parseInt(userSelection);
edu.ncsu.csc326.coffeemaker.Main#recipeListSelection:304|0.0|    return recipe;
edu.ncsu.csc326.coffeemaker.Main#main:313             |0.0|    coffeeMaker = new CoffeeMaker();
edu.ncsu.csc326.coffeemaker.Main#main:315             |0.0|    mainMenu();
edu.ncsu.csc326.coffeemaker.RecipeBook#<init>:19      |0.0|    recipeArray = new Recipe[NUM_RECIPES];
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:34   |0.0|    boolean exists = false;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:36   |0.0|    for (int i = 0; i < recipeArray.length; i++) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:43   |0.0|    boolean added = false;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:45   |0.0|    if (!exists) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:46   |0.0|      for (int i = 0; i < recipeArray.length && !added; i++) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:47   |0.0|        if (recipeArray[i] == null) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:48   |0.0|          recipeArray[i] = r;
**************************sum(failureSlice) - sum(successSlice) ->>**************************
####################################################################################################################
Generating slice representation for slice: Reduced Failure Slice with package filters: edu.ncsu
####################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
**************************sum(successSlice) - sum(failureSlice) ->>**************************
####################################################################################################################
Generating slice representation for slice: Reduced Success Slice with package filters: edu.ncsu
####################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.CoffeeMaker#addRecipe:31  |0.5|    return recipeBook.addRecipe(r);
edu.ncsu.csc326.coffeemaker.Main#addRecipe:84         |0.5|    Recipe r = new Recipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:93         |0.5|      boolean recipeAdded = coffeeMaker.addRecipe(r);
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:34   |0.5|    boolean exists = false;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:36   |0.5|    for (int i = 0; i < recipeArray.length; i++) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:43   |0.5|    boolean added = false;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:45   |0.5|    if (!exists) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:46   |0.5|      for (int i = 0; i < recipeArray.length && !added; i++) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:47   |0.5|        if (recipeArray[i] == null) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:48   |0.5|          recipeArray[i] = r;
**************************Merged and Weighted Slice**************************
####################################################################################################################
Generating slice representation for slice: mergedSlice with package filters: edu.ncsu
####################################################################################################################
```

```
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.CoffeeMaker#addRecipe:31  |0.5|    return recipeBook.addRecipe(r);
edu.ncsu.csc326.coffeemaker.Main#addRecipe:84         |0.5|    Recipe r = new Recipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:93         |0.5|      boolean recipeAdded = coffeeMaker.addRecipe(r);
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:34   |0.5|    boolean exists = false;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:36   |0.5|    for (int i = 0; i < recipeArray.length; i++) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:43   |0.5|    boolean added = false;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:45   |0.5|    if (!exists) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:46   |0.5|      for (int i = 0; i < recipeArray.length && !added; i++) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:47   |0.5|        if (recipeArray[i] == null) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:48   |0.5|          recipeArray[i] = r;
Execution took: 60.0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.701s
[INFO] Finished at: Fri May 25 09:32:38 CEST 2012
[INFO] Final Memory: 6M/79M
[INFO] ------------------------------------------------------------------------
```

A.3.4 Fault 1.3

Listing A.4: Slicing Criterion: edu.ncsu.csc326.coffeemaker.RecipeBook.addRecipe:38

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building JSliceParser 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser ---
**************************Failure Slice**************************
################################################################################################################
Generating slice representation for slice: e:/dev/dev/workspace/CoffeeMaker_FIT/reports/Fault_1_3_fail.slice with package filters: edu.ncsu
################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.Fault_1_3_fail#setUp:9    |0.0|    recipeBook = new RecipeBook();
edu.ncsu.csc326.coffeemaker.Fault_1_3_fail#testAddRecipe_null_shouldThrowNPE:13|0.0|    recipeBook.addRecipe(null);
edu.ncsu.csc326.coffeemaker.RecipeBook#<init>:19      |0.0|    recipeArray = new Recipe[NUM_RECIPES];
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:36   |0.0|    for (int i = 0; i < recipeArray.length; i++) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:38   |0.0|      if (r.equals(recipeArray[i])) {
**************************Success Slice**************************
################################################################################################################
Generating slice representation for slice: e:/dev/dev/workspace/CoffeeMaker_FIT/reports/Fault_1_3_success.slice with package filters: edu.ncsu
################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.Fault_1_3_success#setUp:9 |0.0|    recipeBook = new RecipeBook();
edu.ncsu.csc326.coffeemaker.Fault_1_3_success#testAddRecipe_oneRecipe_onlyFirstRecipeSet:15|0.0|    assertTrue("Adding of a recipe should work", recipeBook.addRecipe(recipe));
edu.ncsu.csc326.coffeemaker.RecipeBook#<init>:19      |0.0|    recipeArray = new Recipe[NUM_RECIPES];
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:36   |0.0|    for (int i = 0; i < recipeArray.length; i++) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:38   |0.0|      if (r.equals(recipeArray[i])) {
**************************sum(failureSlice) - sum(successSlice) ->>**************************
################################################################################################################
Generating slice representation for slice: Reduced Failure Slice with package filters: edu.ncsu
################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.Fault_1_3_fail#setUp:9    |1.0|    recipeBook = new RecipeBook();
edu.ncsu.csc326.coffeemaker.Fault_1_3_fail#testAddRecipe_null_shouldThrowNPE:13|1.0|    recipeBook.addRecipe(null);
**************************sum(successSlice) - sum(failureSlice) ->>**************************
################################################################################################################
Generating slice representation for slice: Reduced Success Slice with package filters: edu.ncsu
################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.Fault_1_3_success#setUp:9 |0.5|    recipeBook = new RecipeBook();
edu.ncsu.csc326.coffeemaker.Fault_1_3_success#testAddRecipe_oneRecipe_onlyFirstRecipeSet:15|0.5|    assertTrue("Adding of a recipe should work", recipeBook.addRecipe(recipe));
**************************Merged and Weighted Slice**************************
################################################################################################################
Generating slice representation for slice: mergedSlice with package filters: edu.ncsu
```

```
################################################################################################################################
class, method and linenumber                                |v()|code
|-------------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.Fault_1_3_success#setUp:9 |0.5|    recipeBook = new RecipeBook();
edu.ncsu.csc326.coffeemaker.Fault_1_3_success#testAddRecipe_oneRecipe_onlyFirstRecipeSet:15|0.5|    assertTrue("Adding of a recipe should work", recipeBook.addRecipe(recipe));
edu.ncsu.csc326.coffeemaker.Fault_1_3_fail#setUp:9    |1.0|    recipeBook = new RecipeBook();
edu.ncsu.csc326.coffeemaker.Fault_1_3_fail#testAddRecipe_null_shouldThrowNPE:13|1.0|    recipeBook.addRecipe(null);
Execution took: 16.0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.656s
[INFO] Finished at: Sun Jun 03 14:14:25 CEST 2012
[INFO] Final Memory: 6M/109M
[INFO] ------------------------------------------------------------------------
```

A.3.5 Fault 2.1

Listing A.5: Slicing Criterion: edu.ncsu.csc326.coffeemaker.RecipeBook.addRecipe:53:added

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building JSliceParser 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser ---
****************************Failure Slice*****************************
################################################################################################################################
Generating slice representation for slice: e:/dev/dev/workspace/CoffeeMaker_FIT/reports/Fault3_b_fail.slice with package filters: edu.ncsu
################################################################################################################################
class, method and linenumber                                |v()|code
|-------------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.CoffeeMaker#<init>:19     |0.0|    recipeBook = new RecipeBook();
edu.ncsu.csc326.coffeemaker.CoffeeMaker#addRecipe:31  |0.0|    return recipeBook.addRecipe(r);
edu.ncsu.csc326.coffeemaker.Main#mainMenu:34          |0.0|      int userInput = Integer.parseInt(inputOutput("Please press the number that corresponds to what you would like the coffee maker to do."));
edu.ncsu.csc326.coffeemaker.Main#mainMenu:36          |0.0|      if (userInput >= 0 && userInput <= 6) {
edu.ncsu.csc326.coffeemaker.Main#mainMenu:37          |0.0|        if (userInput == 1)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:38          |0.0|          addRecipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:67         |0.0|    String name = inputOutput("\nPlease enter the recipe name: ");
edu.ncsu.csc326.coffeemaker.Main#addRecipe:84         |0.0|    Recipe r = new Recipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:86         |0.0|      r.setName(name);
edu.ncsu.csc326.coffeemaker.Main#addRecipe:93         |0.0|      boolean recipeAdded = coffeeMaker.addRecipe(r);
edu.ncsu.csc326.coffeemaker.Main#addRecipe:103        |0.0|      mainMenu();
edu.ncsu.csc326.coffeemaker.Main#inputOutput:272      |0.0|    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
edu.ncsu.csc326.coffeemaker.Main#inputOutput:275      |0.0|      returnString = br.readLine();
edu.ncsu.csc326.coffeemaker.Main#inputOutput:280      |0.0|    return returnString;
edu.ncsu.csc326.coffeemaker.Main#main:313             |0.0|    coffeeMaker = new CoffeeMaker();
edu.ncsu.csc326.coffeemaker.Main#main:315             |0.0|    mainMenu();
edu.ncsu.csc326.coffeemaker.Recipe#setName:100        |0.0|    if (name != null) {
edu.ncsu.csc326.coffeemaker.Recipe#setName:101        |0.0|      this.name = name;
edu.ncsu.csc326.coffeemaker.Recipe#equals:164         |0.0|    if (this == obj)
edu.ncsu.csc326.coffeemaker.Recipe#equals:166         |0.0|    if (obj == null)
edu.ncsu.csc326.coffeemaker.Recipe#equals:168         |0.0|    if (getClass() != obj.getClass())
edu.ncsu.csc326.coffeemaker.Recipe#equals:170         |0.0|    final Recipe other = (Recipe) obj;
edu.ncsu.csc326.coffeemaker.Recipe#equals:171         |0.0|    if (name == null) {
edu.ncsu.csc326.coffeemaker.Recipe#equals:174         |0.0|    } else if (!name.equals(other.name))
edu.ncsu.csc326.coffeemaker.Recipe#equals:176         |0.0|    return true;
edu.ncsu.csc326.coffeemaker.RecipeBook#<init>:19      |0.0|    recipeArray = new Recipe[NUM_RECIPES];
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:34   |0.0|    boolean exists = false;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:36   |0.0|    for (int i = 0; i < recipeArray.length; i++) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:37   |0.0|      if (r.equals(recipeArray[i])) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:39   |0.0|        exists = false;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:43   |0.0|    boolean added = false;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:45   |0.0|    if (!exists) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:46   |0.0|      for (int i = 0; i < recipeArray.length && !added; i++) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:47   |0.0|        if (recipeArray[i] == null) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:48   |0.0|          recipeArray[i] = r;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:49   |0.0|          added = true;
****************************Success Slice*****************************
################################################################################################################################
```

```
Generating slice representation for slice: e:/dev/dev/workspace/CoffeeMaker_FIT/reports/Fault3_b_success.slice with package filters: edu.ncsu
#################################################################################################################
class, method and linenumber                                |v()|code
|-----------------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.CoffeeMaker#<init>:19            |0.0|    recipeBook = new RecipeBook();
edu.ncsu.csc326.coffeemaker.CoffeeMaker#addRecipe:31         |0.0|    return recipeBook.addRecipe(r);
edu.ncsu.csc326.coffeemaker.Main#mainMenu:34                 |0.0|      int userInput = Integer.parseInt(inputOutput("Please press the number that corresponds to what you would like the coffee maker to do."));
edu.ncsu.csc326.coffeemaker.Main#mainMenu:36                 |0.0|      if (userInput >= 0 && userInput <= 6) {
edu.ncsu.csc326.coffeemaker.Main#mainMenu:37                 |0.0|        if (userInput == 1)
edu.ncsu.csc326.coffeemaker.Main#mainMenu:38                 |0.0|          addRecipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:93                |0.0|      boolean recipeAdded = coffeeMaker.addRecipe(r);
edu.ncsu.csc326.coffeemaker.Main#addRecipe:103               |0.0|      mainMenu();
edu.ncsu.csc326.coffeemaker.Main#inputOutput:272             |0.0|    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
edu.ncsu.csc326.coffeemaker.Main#inputOutput:275             |0.0|      returnString = br.readLine();
edu.ncsu.csc326.coffeemaker.Main#inputOutput:280             |0.0|    return returnString;
edu.ncsu.csc326.coffeemaker.Main#main:313                    |0.0|    coffeeMaker = new CoffeeMaker();
edu.ncsu.csc326.coffeemaker.Main#main:315                    |0.0|    mainMenu();
edu.ncsu.csc326.coffeemaker.RecipeBook#<init>:19             |0.0|    recipeArray = new Recipe[NUM_RECIPES];
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:34          |0.0|    boolean exists = false;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:36          |0.0|    for (int i = 0; i < recipeArray.length; i++) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:43          |0.0|    boolean added = false;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:45          |0.0|    if (!exists) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:46          |0.0|      for (int i = 0; i < recipeArray.length && !added; i++) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:47          |0.0|        if (recipeArray[i] == null) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:49          |0.0|          added = true;
*****************************sum(failureSlice) - sum(successSlice) ->>*****************************
#################################################################################################################
Generating slice representation for slice: Reduced Failure Slice with package filters: edu.ncsu
#################################################################################################################
class, method and linenumber                                |v()|code
|-----------------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.Main#addRecipe:67                |1.0|    String name = inputOutput("\nPlease enter the recipe name: ");
edu.ncsu.csc326.coffeemaker.Main#addRecipe:84                |1.0|    Recipe r = new Recipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:86                |1.0|      r.setName(name);
edu.ncsu.csc326.coffeemaker.Recipe#setName:100               |1.0|    if (name != null) {
edu.ncsu.csc326.coffeemaker.Recipe#setName:101               |1.0|      this.name = name;
edu.ncsu.csc326.coffeemaker.Recipe#equals:164                |1.0|    if (this == obj)
edu.ncsu.csc326.coffeemaker.Recipe#equals:166                |1.0|    if (obj == null)
edu.ncsu.csc326.coffeemaker.Recipe#equals:168                |1.0|    if (getClass() != obj.getClass())
edu.ncsu.csc326.coffeemaker.Recipe#equals:170                |1.0|    final Recipe other = (Recipe) obj;
edu.ncsu.csc326.coffeemaker.Recipe#equals:171                |1.0|    if (name == null) {
edu.ncsu.csc326.coffeemaker.Recipe#equals:174                |1.0|    } else if (!name.equals(other.name))
edu.ncsu.csc326.coffeemaker.Recipe#equals:176                |1.0|    return true;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:37          |1.0|      if (r.equals(recipeArray[i])) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:39          |1.0|        exists = false;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:48          |1.0|          recipeArray[i] = r;
*****************************sum(successSlice) - sum(failureSlice) ->>*****************************
#################################################################################################################
Generating slice representation for slice: Reduced Success Slice with package filters: edu.ncsu
#################################################################################################################
class, method and linenumber                                |v()|code
|-----------------------------------------------------------|---|---------------------------------------
*****************************Merged and Weighted Slice*****************************
#################################################################################################################
Generating slice representation for slice: mergedSlice with package filters: edu.ncsu
#################################################################################################################
class, method and linenumber                                |v()|code
|-----------------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.Main#addRecipe:67                |1.0|    String name = inputOutput("\nPlease enter the recipe name: ");
edu.ncsu.csc326.coffeemaker.Main#addRecipe:84                |1.0|    Recipe r = new Recipe();
edu.ncsu.csc326.coffeemaker.Main#addRecipe:86                |1.0|      r.setName(name);
edu.ncsu.csc326.coffeemaker.Recipe#setName:100               |1.0|    if (name != null) {
edu.ncsu.csc326.coffeemaker.Recipe#setName:101               |1.0|      this.name = name;
edu.ncsu.csc326.coffeemaker.Recipe#equals:164                |1.0|    if (this == obj)
edu.ncsu.csc326.coffeemaker.Recipe#equals:166                |1.0|    if (obj == null)
edu.ncsu.csc326.coffeemaker.Recipe#equals:168                |1.0|    if (getClass() != obj.getClass())
edu.ncsu.csc326.coffeemaker.Recipe#equals:170                |1.0|    final Recipe other = (Recipe) obj;
edu.ncsu.csc326.coffeemaker.Recipe#equals:171                |1.0|    if (name == null) {
edu.ncsu.csc326.coffeemaker.Recipe#equals:174                |1.0|    } else if (!name.equals(other.name))
edu.ncsu.csc326.coffeemaker.Recipe#equals:176                |1.0|    return true;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:37          |1.0|      if (r.equals(recipeArray[i])) {
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:39          |1.0|        exists = false;
edu.ncsu.csc326.coffeemaker.RecipeBook#addRecipe:48          |1.0|          recipeArray[i] = r;
Execution took: 69.0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.711s
[INFO] Finished at: Fri May 25 13:58:58 CEST 2012
```

```
[INFO] Final Memory: 7M/109M
[INFO] ------------------------------------------------------------------------
```

A.3.6 Fault 2.2

Listing A.6: Slicing Criterion: edu.ncsu.csc326.coffeemaker.Inventory.useIngredients:214:isEnough

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building JSliceParser 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ jslice-parser ---
**************************Failure Slice**************************
################################################################################################################
Generating slice representation for slice: e:/dev/dev/workspace/CoffeeMaker_FIT/reports/Fault_2_2_fail.slice with package
      filters: edu.ncsu.csc326.coffeemaker.Recipe,edu.ncsu.csc326.coffeemaker.Inventory
################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.Inventory#<init>:22       |0.0|    setCoffee(15);
edu.ncsu.csc326.coffeemaker.Inventory#<init>:23       |0.0|    setMilk(15);
edu.ncsu.csc326.coffeemaker.Inventory#<init>:24       |0.0|    setSugar(15);
edu.ncsu.csc326.coffeemaker.Inventory#<init>:25       |0.0|    setChocolate(15);
edu.ncsu.csc326.coffeemaker.Inventory#setChocolate:44 |0.0|    if (chocolate >= 0) {
edu.ncsu.csc326.coffeemaker.Inventory#setChocolate:45 |0.0|      Inventory.chocolate = chocolate;
edu.ncsu.csc326.coffeemaker.Inventory#setCoffee:76    |0.0|    if (coffee >= 0) {
edu.ncsu.csc326.coffeemaker.Inventory#setCoffee:77    |0.0|      Inventory.coffee = coffee;
edu.ncsu.csc326.coffeemaker.Inventory#setMilk:107     |0.0|    if (milk >= 0) {
edu.ncsu.csc326.coffeemaker.Inventory#setMilk:108     |0.0|      Inventory.milk = milk;
edu.ncsu.csc326.coffeemaker.Inventory#setSugar:138    |0.0|    if (sugar >= 0) {
edu.ncsu.csc326.coffeemaker.Inventory#setSugar:139    |0.0|      Inventory.sugar = sugar;
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:179|0.0|    isEnough = false;
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:180|0.0|    if (Inventory.coffee < r.getAmtCoffee()) {
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:183|0.0|    if (Inventory.milk < r.getAmtMilk()) {
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:186|0.0|    if (Inventory.sugar < r.getAmtSugar()) {
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:189|0.0|    if (Inventory.chocolate < r.getAmtChocolate()) {
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:194|0.0|    return isEnough;
edu.ncsu.csc326.coffeemaker.Inventory#useIngredients:205|0.0|    if (enoughIngredients(r)) {
edu.ncsu.csc326.coffeemaker.Inventory#useIngredients:212|0.0|      isEnough = false;
edu.ncsu.csc326.coffeemaker.Recipe#getAmtChocolate:32 |0.0|    return amtChocolate;
edu.ncsu.csc326.coffeemaker.Recipe#setAmtChocolate:40 |0.0|    this.amtChocolate = getAmountFromString(chocolate);
edu.ncsu.csc326.coffeemaker.Recipe#getAmtCoffee:47    |0.0|    return amtCoffee;
edu.ncsu.csc326.coffeemaker.Recipe#setAmtCoffee:55    |0.0|    this.amtCoffee = getAmountFromString(coffee);
edu.ncsu.csc326.coffeemaker.Recipe#getAmtMilk:62      |0.0|    return amtMilk;
edu.ncsu.csc326.coffeemaker.Recipe#setAmtMilk:70      |0.0|    this.amtMilk = getAmountFromString(milk);
edu.ncsu.csc326.coffeemaker.Recipe#getAmtSugar:77     |0.0|    return amtSugar;
edu.ncsu.csc326.coffeemaker.Recipe#setAmtSugar:85     |0.0|    this.amtSugar = getAmountFromString(sugar);
edu.ncsu.csc326.coffeemaker.Recipe#getAmountFromString:135|0.0|      amount = Integer.parseInt(amountStr);
edu.ncsu.csc326.coffeemaker.Recipe#getAmountFromString:142|0.0|    return amount;
**************************Success Slice**************************
################################################################################################################
Generating slice representation for slice: e:/dev/dev/workspace/CoffeeMaker_FIT/reports/Fault_2_2_success.slice with
      package filters: edu.ncsu.csc326.coffeemaker.Recipe,edu.ncsu.csc326.coffeemaker.Inventory
################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.Inventory#<init>:22       |0.0|    setCoffee(15);
edu.ncsu.csc326.coffeemaker.Inventory#<init>:23       |0.0|    setMilk(15);
edu.ncsu.csc326.coffeemaker.Inventory#<init>:24       |0.0|    setSugar(15);
edu.ncsu.csc326.coffeemaker.Inventory#<init>:25       |0.0|    setChocolate(15);
edu.ncsu.csc326.coffeemaker.Inventory#setChocolate:44 |0.0|    if (chocolate >= 0) {
edu.ncsu.csc326.coffeemaker.Inventory#setChocolate:45 |0.0|      Inventory.chocolate = chocolate;
edu.ncsu.csc326.coffeemaker.Inventory#setCoffee:76    |0.0|    if (coffee >= 0) {
edu.ncsu.csc326.coffeemaker.Inventory#setCoffee:77    |0.0|      Inventory.coffee = coffee;
edu.ncsu.csc326.coffeemaker.Inventory#setMilk:107     |0.0|    if (milk >= 0) {
edu.ncsu.csc326.coffeemaker.Inventory#setMilk:108     |0.0|      Inventory.milk = milk;
edu.ncsu.csc326.coffeemaker.Inventory#setSugar:138    |0.0|    if (sugar >= 0) {
edu.ncsu.csc326.coffeemaker.Inventory#setSugar:139    |0.0|      Inventory.sugar = sugar;
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:179|0.0|    isEnough = false;
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:180|0.0|    if (Inventory.coffee < r.getAmtCoffee()) {
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:183|0.0|    if (Inventory.milk < r.getAmtMilk()) {
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:186|0.0|    if (Inventory.sugar < r.getAmtSugar()) {
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:189|0.0|    if (Inventory.chocolate < r.getAmtChocolate()) {
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:190|0.0|      return isEnough;
edu.ncsu.csc326.coffeemaker.Inventory#useIngredients:205|0.0|    if (enoughIngredients(r)) {
```

```
edu.ncsu.csc326.coffeemaker.Inventory#useIngredients:212|0.0|      isEnough = false;
edu.ncsu.csc326.coffeemaker.Recipe#<init>:22          |0.0|    this.amtCoffee = 0;
edu.ncsu.csc326.coffeemaker.Recipe#getAmtChocolate:32 |0.0|    return amtChocolate;
edu.ncsu.csc326.coffeemaker.Recipe#setAmtChocolate:40 |0.0|    this.amtChocolate = getAmountFromString(chocolate);
edu.ncsu.csc326.coffeemaker.Recipe#getAmtCoffee:47    |0.0|    return amtCoffee;
edu.ncsu.csc326.coffeemaker.Recipe#getAmtMilk:62      |0.0|    return amtMilk;
edu.ncsu.csc326.coffeemaker.Recipe#setAmtMilk:70      |0.0|    this.amtMilk = getAmountFromString(milk);
edu.ncsu.csc326.coffeemaker.Recipe#getAmtSugar:77     |0.0|    return amtSugar;
edu.ncsu.csc326.coffeemaker.Recipe#setAmtSugar:85     |0.0|    this.amtSugar = getAmountFromString(sugar);
edu.ncsu.csc326.coffeemaker.Recipe#getAmountFromString:135|0.0|      amount = Integer.parseInt(amountStr);
edu.ncsu.csc326.coffeemaker.Recipe#getAmountFromString:142|0.0|    return amount;
**************************sum(failureSlice) - sum(successSlice) ->>****************************
#########################################################################################################################
Generating slice representation for slice: Reduced Failure Slice with package filters: edu.ncsu.csc326.coffeemaker.Recipe
     ,edu.ncsu.csc326.coffeemaker.Inventory
#########################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:194|1.0|    return isEnough;
edu.ncsu.csc326.coffeemaker.Recipe#setAmtCoffee:55    |1.0|    this.amtCoffee = getAmountFromString(coffee);
**************************sum(successSlice) - sum(failureSlice) ->>****************************
#########################################################################################################################
Generating slice representation for slice: Reduced Success Slice with package filters: edu.ncsu.csc326.coffeemaker.Recipe
     ,edu.ncsu.csc326.coffeemaker.Inventory
#########################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:190|0.5|      return isEnough;
edu.ncsu.csc326.coffeemaker.Recipe#<init>:22          |0.5|    this.amtCoffee = 0;
**************************Merged and Weighted Slice**************************
#########################################################################################################################
Generating slice representation for slice: mergedSlice with package filters: edu.ncsu.csc326.coffeemaker.Recipe,edu.ncsu.
     csc326.coffeemaker.Inventory
#########################################################################################################################
class, method and linenumber                          |v()|code
|-----------------------------------------------------|---|---------------------------------------
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:190|0.5|      return isEnough;
edu.ncsu.csc326.coffeemaker.Recipe#<init>:22          |0.5|    this.amtCoffee = 0;
edu.ncsu.csc326.coffeemaker.Inventory#enoughIngredients:194|1.0|    return isEnough;
edu.ncsu.csc326.coffeemaker.Recipe#setAmtCoffee:55    |1.0|    this.amtCoffee = getAmountFromString(coffee);
Execution took: 63.0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.702s
[INFO] Finished at: Mon Jun 04 13:30:08 CEST 2012
[INFO] Final Memory: 7M/109M
[INFO] ------------------------------------------------------------------------
```

Literaturverzeichnis

[dom] *Document Object Model*. – URL http://www.w3.org/DOM/. – Zugriff am: 09.05.2012

[kaffe] *Kaffe*. – URL https://github.com/kaffe/kaffe. – Zugriff am: 07.05.2012

[Agrawal und Horgan 1990] AGRAWAL, Hiralal ; HORGAN, Joseph R.: *Dynamic Program Slicing*. 1990

[Agrawal u. a. 1995] AGRAWAL, Hiralal ; HORGAN, Joseph R. ; LONDON, Saul ; WONG, W. E.: *Fault Localization using Execution Slices and Dataflow Tests*. 1995

[Artzi u. a. 2010] ARTZI, Shay ; DOLBY, Julian ; TIP, Frank ; PISTOIA, Marco: Practical fault localization for dynamic web applications. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA : ACM, 2010 (ICSE '10), S. 265–274. – URL http://doi.acm.org/10.1145/1806799.1806840. – ISBN 978-1-60558-719-6

[Cleve und Zeller 2005] CLEVE, Holger ; ZELLER, Andreas: Locating causes of program failures. In: *Proceedings of the 27th international conference on Software engineering*. New York, NY, USA : ACM, 2005 (ICSE '05), S. 342–351. – URL http://doi.acm.org/10.1145/1062455.1062522. – ISBN 1-58113-963-2

[DeMillo u. a. 1997] DEMILLO, R.A. ; PAN, H. ; SPAFFORD, E.H.: Failure and fault analysis for software debugging. In: *Computer Software and Applications Conference, 1997. COMPSAC '97. Proceedings., The Twenty-First Annual International*, aug 1997, S. 515 –521

[ECMA 2011] ECMA: *Standard ECMA-262*. ECMA International, 2011. – URL http://www.ecma-international.org/publications/standards/Ecma-262.htm

[Fielding u. a. 1999] FIELDING, R. ; GETTYS, J. ; MOGUL, J. ; FRYSTYK, H. ; MASINTER, L. ; LEACH, P. ; BERNERS-LEE, T.: *Hypertext Transfer Protocol – HTTP/1.1 (RFC2616)*. 1999

[Garrett 2005] GARRETT, Jesse J.: Ajax: A New Approach to Web Applications. In: *adaptivpath.com* (2005)

[Gold und Harman 2007] GOLD, Nicolas ; HARMAN, Mark: An empirical study of static program slice size. In: *ACM Transactions on Software Engineering and Methodology (TOSEM* 16 (2007), S. 2007

[Hammacher 2008] HAMMACHER, Clemens: *Design and Implementation of an Efficient Dynamic Slicer for Java*. Bachelor's Thesis. November 2008

[IEEE-24765 2010] IEEE-24765: Systems and software engineering – Vocabulary. In: *ISO/IEC/IEEE 24765:2010(E)* (2010), 15, S. 1 –418

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

[Jones und Harrold 2005] JONES, James A. ; HARROLD, Mary J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA : ACM, 2005 (ASE '05), S. 273–282. – URL http://doi.acm.org/10.1145/1101908.1101949. – ISBN 1-58113-993-4

[Jones u. a. 2002] JONES, James A. ; HARROLD, Mary J. ; STASKO, John: Visualization of test information to assist fault localization. In: *Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA : ACM, 2002 (ICSE '02), S. 467–477. – URL http://doi.acm.org/10.1145/581339.581397. – ISBN 1-58113-472-X

[Jones u. a. 2001] JONES, James A. ; HARROLD, Mary J. ; STASKO, John T.: Visualization for Fault Localization. In: *in Proceedings of ICSE 2001 Workshop on Software Visualization*, 2001, S. 71–75

[Korel und Laski 1988] KOREL, B. ; LASKI, J.: Dynamic program slicing. In: *Inf. Process. Lett.* 29 (1988), Oktober, Nr. 3, S. 155–163. – URL http://dx.doi.org/10.1016/0020-0190(88)90054-3. – ISSN 0020-0190

[Leiner u. a. 1985] LEINER, B. ; COLE, R. ; POSTEL, J. ; MILLS, D.: The DARPA internet protocol suite. In: *Communications Magazine, IEEE* 23 (1985), march, Nr. 3, S. 29 –34. – ISSN 0163-6804

[Liang und Xu 2005] LIANG, Donglin ; XU, Kai: Debugging object-oriented programs with behavior views. In: *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. New York, NY, USA : ACM, 2005 (AADEBUG'05), S. 133–142.

– URL http://doi.acm.org/10.1145/1085130.1085148. – ISBN 1-59593-050-7

[Lloyd und Lipow 1977] LLOYD, David.K ; LIPOW, Myron: *Reliability: Management, Methods and Mathematics*. Second Edition. Redondo Beach, California : Published by the Authors, 1977

[Lutz und Woodhouse 1999] LUTZ, R.R. ; WOODHOUSE, R.M.: Bi-directional Analysis for Certification of Safety-Critical Software. In: *Proceedings, ISACC'99 International Software Assurance Ceritification Conference*. Chantilly, VA : Proceedings, ISACC'99 International Software Assurance Ceritification Conference, February 1999, S. unknown

[Reifer 1979] REIFER, Donald J.: Software Failure Modes and Effects Analysis. In: *Reliability, IEEE Transactions on* R-28 (1979), aug., Nr. 3, S. 247 –249. – ISSN 0018-9529

[Renieris und Reiss 2003] RENIERIS, M. ; REISS, S.P.: Fault localization with nearest neighbour queries. In: *ASE*, 2003, S. 30–39

[Ristord und Esmenjaud 2001] RISTORD, L. ; ESMENJAUD, C.: FMEA Per-oredon the SPINLINE3 Operational System Softwareas part of the TIHANGE 1 NIS Refurbishment Safety Case. In: *Licensing and Operating Experience of Computer Based I & C Systems.* Ceske Budejovice : CNRAICNSI Workshop, September 2001, S. unknown

[Spillner u. a. 2006] SPILLNER, Andreas ; LINZ, Tilo ; SCHAEFER, Hans: *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. 1. O'Reilly Media, Mai 2006. – URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/3898643638. – ISBN 3898643638

[Svobodova 1985] SVOBODOVA, Liba: Client/Server Model of Distributed Processing. In: *Kommunikation in Verteilten Systemen (1)'85*, 1985, S. 485–498

[Tanenbaum 2011] TANENBAUM, A.S.: *Computernetzwerke*. 5. Pearson, 2011. – ISBN 978-3-8273-7046-4

[Tanenbaum und Steen 2008] TANENBAUM, A.S. ; STEEN, M.: *Verteilte Systeme: Prinzipien und Paradigmen*. Pearson Studium, 2008 (Pearson Studium). – ISBN 9783827372932

[Tassey 2002] TASSEY, G.: The economic impacts of inadequate infrastructure for software testing / National Institute of Standards and Technology. 2002. – Forschungsbericht

[TIOBE 2012] TIOBE: *TIOBE Programming Community Index for May 2012*. 2012. – URL http://www.tiobe.com/content/paperinfo/tpci/index.html

[Tip 1995] TIP, F.: A Survey of Program Slicing Techniques. In: *JOURNAL OF PROGRAMMING LANGUAGES* 3 (1995), S. 121–189

[Wang und Roychoudhury 2004] WANG, Tao ; ROYCHOUDHURY, A.: Using compressed bytecode traces for slicing Java programs. In: *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1317473, 2004, S. 512–521

[Weiser 1979] WEISER, Mark: *Program Slicing: Formal, Psychological, and Practical Investigation of an Automatic Program Abstraction Method*. Ann Arbor, Michigan, The University of Michigan, Dissertation, 1979

[Weiser 1981] WEISER, Mark: Program slicing. In: *Proceedings of the 5th international conference on Software engineering*. Piscataway, NJ, USA : IEEE Press, 1981 (ICSE '81), S. 439–449. – URL http://dl.acm.org/citation.cfm?id=800078.802557. – ISBN 0-89791-146-6

[Wong und Debroy 2009] WONG, W. E. ; DEBROY, Vidroha: *Software Fault Localization*. 2009

[Zhang u. a. 2006] ZHANG, Xiangyu ; GUPTA, Neelam ; GUPTA, Rajiv: Locating faults through automated predicate switching. In: *Proceedings of the 28th international conference on Software engineering*. New York, NY, USA : ACM, 2006 (ICSE '06), S. 272–281. – URL http://doi.acm.org/10.1145/1134285.1134324. – ISBN 1-59593-375-1

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, 11. Juni 2012	
Ort, Datum	Unterschrift