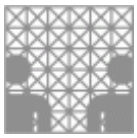
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

MIN-Fakultät
Fachbereich Informatik
Arbeitsbereich Angewandte und
Sozialorientierte Informatik



Bachelorarbeit

Regressionstest mit Ant und Eclipse für das Simulationsrahmenwerk DESMO-J

Bearbeiter:	René Wecker Vörstekoppel 22 22399 Hamburg
Matrikelnummer:	5954421
E-Mail:	wecker.rene@googlemail.com
Erstkorrektor:	Prof. Dr. Bernd Page
Zweitkorrektor:	Dr. Axel Schmolitzky
Betreuer:	Philip Joschko
Abgabedatum:	13. Januar 2012

Zusammenfassung

In dieser Bachelorarbeit geht es darum, für das Simulationsrahmenwerk DESMO-J einen Regressionstest zu entwickeln. Dazu werden diverse Modelle in zwei unterschiedliche DESMO-J Versionen eingegeben. Dessen Ausgaben gilt es zu vergleichen. Beim Auftreten von Unterschieden, soll dessen Ursache möglichst genau eingegrenzt werden. Der Regressionstest wird mit Ant und Eclipse gesteuert. Die Eingrenzung der Fehlerquellen geschieht mit Hilfe von Analysen der Quelltextabdeckung.

Danksagungen

An dieser Stelle möchte ich mich zuerst bei Professor Page für das entgegengebrachte Vertrauen bedanken, welches es mir ermöglichte kurzfristig diese Arbeit anzumelden. Weiterhin danke ich Dr. Schmolitzky dafür, dass er sich die Zeit als Zweitkorrektor genommen hat. Ein besonderer Dank geht auch an Philip Joschko, da er mir als Betreuer stets für Fragen zur Verfügung stand. Abschließend möchte ich auch noch Johannes Göbel dafür danken, dass er mir für diese Arbeit viele Simulationsmodelle zur Verfügung gestellt hat.

Inhaltsverzeichnis

1. Einleitung

In dieser Bachelorarbeit geht es darum, einen Regressionstest für das Simulationsrahmenwerk DESMO-J[1] zu entwickeln. Der folgende Abschnitt wird die Motivation zu diesem Vorhaben erläutern. Anschließend wird die Zielsetzung genauer präzisiert und eine Übersicht der einzelnen Kapitel gegeben.

1.1. Motivation

Software ist ein Produkt, welches aus einem Entwicklungsprozess heraus entsteht. In diesem Prozess der Entwicklung können sich Fehler oder Mängel einschleichen, die u.U. erst spät oder überhaupt nicht entdeckt werden. Es ist daher unverzichtbar, eine Software möglichst schon während des Entwicklungsprozesses, in geeigneter Weise zu testen.

An der Universität Hamburg wurde das Simulationsrahmenwerk DESMO-J entwickelt. Es bietet alle elementaren Elemente, welche für eine Simulation benötigt werden. Außerdem ermöglicht es dem Benutzer eigene Modelle zu modellieren[2]. Dieses Rahmenwerk ist im laufe der Jahre gewachsen und wird ständig weiterentwickelt. Was bei der Entwicklung allerdings ausblieb, waren Methoden, um die Qualität der Software sicherzustellen.

Aus diesem Grund ist es nötig sich mit der Frage auseinanderzusetzen, ob es möglich ist für DESMO-J Konzepte zu entwickeln, welche die Qualität der Software prüfen und gewährleisten können.

[1] **D**iscrete-**E**vent **S**imulation and **MO**delling in **J**ava
[2] Ausführliche Informationen zu Simulation und DESMO-J sind in Kapitel 3 zu finden.

1.2. Zielsetzung

Ziel dieser Arbeit ist es, den Entwicklungsprozess zu unterstützen, indem eine geeignete Testmethode zur Verfügung gestellt wird. Diese Testmethode soll ein Regressionstest sein, welcher mit Hilfe von Ant und Eclipse[3] gesteuert wird. Dem Entwickler soll es somit möglich sein nach oder bereits während der Entwicklung Fehlerwirkungen festzustellen. Wieso für dieses Vorhaben ein Regressionstest eingesetzt wird, wird in Kapitel 3.3 diskutiert.

Der Entwickler soll allerdings nicht nur auf das mögliche Vorhandensein von Fehlern hingewiesen werden. Durch eine Analyse der Testergebnisse, soll der Bereich um einen Fehler so gut wie möglich eingegrenzt werden.

Der Test soll mit verschiedenen Modellen durchgeführt werden, sodass deren Reportausgaben auf Richtigkeit geprüft werden. Als richtig wird hierbei die Ausgabe einer früheren Version angenommen. Falls dabei Unterschiede festgestellt werden, soll zu den entsprechenden Modellen eine Analyse der Quelltextabdeckung angefertigt werden. Dies bedeutet, dass aufgezeichnet wird, welche Quelltextbereiche von DESMO-J das Modell aufgerufen hat. Der Bereich für einen möglichen Fehler, lässt sich durch diese Analyse eingrenzen.

1.3. Übersicht

Dieser Abschnitt stellt die folgenden Kapitel dieser Arbeit kurz vor und erläutert, wieso sie für das Erreichen der Zielsetzung wichtig sind.

Im zweiten Kapitel werden die Grundlagen von Softwaretests vorgestellt. Hierbei geht es um allgemeingültige Grundsätze beim Testen von Software. Außerdem werden die typischen Arten von Softwaretests und deren Herangehensweise vorgestellt.

Das dritte Kapitel behandelt dann die Grundlagen von Simulation und DESMO-J. Die Grundlagen der Simulation sind wichtig um DESMO-J zu verstehen. Da DESMO-J ein zentraler Bestandteil dieser Arbeit ist, ist es besonders wichtig dessen Funktionsweise

[3]Eclipse IDE: http://www.eclipse.org/

zu verstehen. Da an dieser Stelle sowohl die Grundlagen zu DESMO-J als auch zu Softwaretests bekannt sind, wird abschließend der Frage nachgegangen, wie DESMO-J in geeigneter Weise zu testen ist.

Bevor der soeben bestimmte Test erläutert wird, werden im vierten Kapitel zunächst die dafür benötigten Werkzeuge vorgestellt. Zu diesen Werkzeugen gehört *Apache Ant* für die Ablaufsteuerung, *XMLUnit* zum Vergleich der Reports und *JaCoCo* für die Analyse der Quelltextabdeckung.

Im fünften Kapitel werden die letzten Überlegungen des dritten Kapitels aufgegriffen und ein softwaretechnisches Konzept zu dem gefundenen Test vorgestellt. In diesem Konzept werden zunächst Anforderungen an den Test definiert. Anschließend wird vorgestellt, wie diese Anforderungen mit Hilfe der Werkzeuge aus dem vierten Kapitel, gelöst werden können.

Das sechste Kapitel beschäftigt sich dann mit der Implementation des Konzepts. Hier wird darauf eingegangen mit welchen softwaretechnischen Mitteln das Konzept umgesetzt wurde oder aus welchen Gründen vom Konzept abgewichen werden musste.

In Kapitel sieben wird diskutiert, ob mit der Implementation das definierte Ziel erreicht werden konnte. Hier werden die Vor- und Nachteile der entwickelten Lösung vorgestellt.

Abschließend wird in Kapitel acht ein Ausblick gegeben. Hier wird erklärt, welche Punkte noch verbessert oder ausgebaut werden können.

2. Grundlagen von Softwaretests

Mit dieser Arbeit soll eine geeignete Testmethode geschaffen werden, welche den Entwicklungsprozess unterstützt. Dazu ist es erst einmal notwendig die Möglichkeiten zu kennen, mit denen Software geprüft und getestet werden kann.

Generell geht es bei Softwaretests genau so darum, die festgelegten Anforderungen zu prüfen, wie bei materiell gefertigten Produkten.

> „Die Prüfung der Teilprodukte bzw. des Endprodukts gestaltet sich allerdings schwieriger, da das erstellte Produkt nicht ≫greifbar≪, also immateriell ist und eine Prüfung nicht ≫handfest≪ durchgeführt werden kann.“ [SL10, S.6].

Aus diesem Grund wird weniger geprüft, als systematisch getestet. Hierzu werden stichprobenartig die Soll- und Istverhalten miteinander verglichen, um so die Fehlerwirkungen nachzuweisen, die Qualität zu bestimmen sowie das Vertrauen in das Programm zu erhöhen.[1]

2.1. Sieben Grundsätze des Testens

Bevor die unterschiedlichen Arten von Softwaretests erläutert werden, sollen die sieben Grundsätze des Testens vorgestellt werden, die sich laut [SL10, Kapitel 2.4] in den letzten 40 Jahren herauskristallisiert haben.

1. **Testen zeigt die Anwesenheit von Fehlern**
   Selbst wenn alle Tests keine Fehler aufzeigen, bedeutet dies nicht zwangsweise, dass das System auch fehlerfrei ist. Tests können lediglich die Anwesenheit von Fehlern

[1] vgl. [SL10, S.9])

nachweisen. Durch gründliches Testen kann so das Vertrauen in die Software erhöht werden, da die Wahrscheinlichkeit für vorhandene Fehler sinkt.

2. **Vollständiges Testen ist nicht möglich**
   Aufgrund der unzähligen Möglichkeiten an Eingaben und Verzweigungen in einem Programm, ist ein vollständiger Test nicht durchführbar, es sei denn das Testobjekt ist sehr klein und trivial. Daher muss man beim Testen Prioritäten setzen und Stichproben verwenden.

3. **Mit dem Testen frühzeitig beginnen**
   Nur wenn man rechtzeitig mit dem Testen beginnt, kann man auch frühzeitig Fehler erkennen und somit viel Zeit und Geld sparen.

4. **Häufung von Fehlern**
   Die in einem System vorhandenen Fehler sind nicht gleichverteilt, sondern häufen sich meist an bestimmten Stellen. Es muss also bedacht werden, dass bei einem gefundenen Fehler oft weitere zu finden sind.

5. **Zunehmende Testresistenz (pesticide paradox)**
   So wie im laufe der Entwicklung die Software wächst, müssen entsprechend auch die Tests angepasst und erweitert werden. Wenn vorhandene Tests nur wiederholt werden, können neu entstandene Fehler nicht aufgedeckt werden. Weiterhin könnten Fehler nicht aufgedeckt werden, die bereits vorhanden waren, sich aber erst durch die neue Konstellation bemerkbar machen.

6. **Testen ist abhängig vom Umfeld**
   Systeme haben immer unterschiedliche Ziele und Anforderungen und werden auch in unterschiedlichen Umgebungen eingesetzt. Daher ist es wichtig, beim Testen individuell auf diese Unterschiede einzugehen.

7. **Trugschluss: Keine Fehler bedeutet ein brauchbares System**
   Wenn ein System durch Testen keine Fehler aufweist und das Vertrauen in die Robustheit durch gute Testabdeckung hoch ist, bedeutet es nicht gleichzeitig, dass die Software auch problemlos vom Nutzer eingesetzt werden kann. Es ist daher wichtig den Nutzer frühzeitig in den Entwicklungszyklus einzubinden und entsprechende Prototypen bereitzustellen.

2.2. Arten von Softwaretests

Aus dem vorherigem Abschnitt wurde klar, dass es den einen Test nicht geben kann und dass es immer auf viele verschiedene Faktoren ankommt.

Je nach Entwicklungsmodell und Programmierparadigma, gestalten sich die Phasen und Herangehensweisen der Programmierung unterschiedlich. Zu den unterschiedlichen Phasen gehören meist auch unterschiedliche Teststufen. Dieser Abschnitt erklärt die einzelnen Stufen anhand des *V-Modells* nach [Boe79], welches in Abbildung 2.1 gezeigt wird.

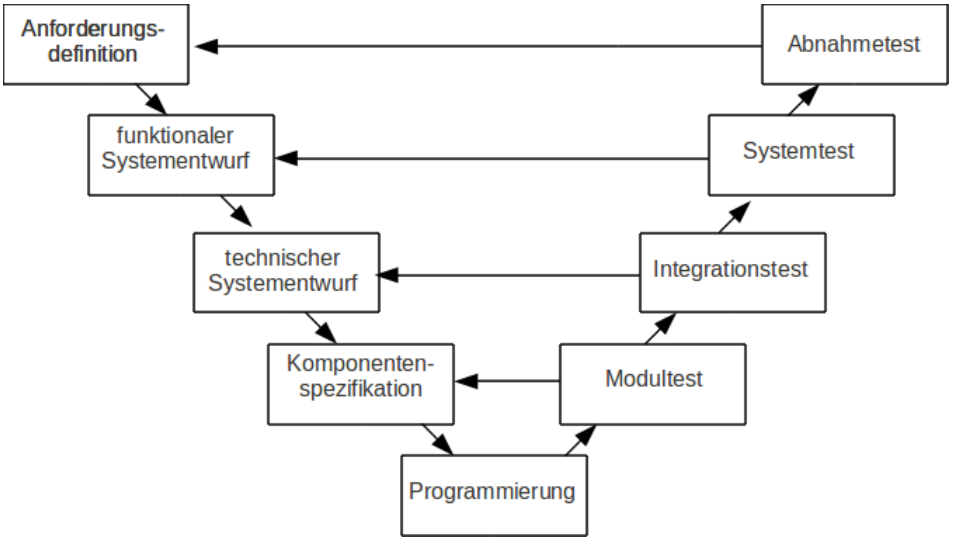


Abbildung 2.1.: Das V-Modell ([SL10, Abb.3-1] nachempfunden.)

> „Auch wenn im Projekt nach einem anderen Vorgehensmodell gearbeitet wird, lassen sich die im Folgenden dargestellten Prinzipien übertragen“.
> [SL10, S.39]

Das V-Modell beginnt damit, dass ein Kunde seine Anforderungen an das zu programmierende System definiert. Aus diesen Anforderungen werden dann Funktionen und Dialogabläufe abgeleitet. Im technischen Systementwurf wird die Architektur der Software festgelegt. Dazu werden Teilsysteme mit entsprechenden Schnittstellen definiert. Anschließend werden bei der Modulspezifikation die Aufgaben und der innere Aufbau der Teilsysteme beschrieben. Auf der linken Seite wird somit die Software immer detail-

reicher geplant, bis mit der Programmierung angefangen wird. Anschließend geht es auf der rechten Seite mit den entsprechenden Teststufen weiter. Zu jeder Entwicklungsstufe gibt es eine entsprechende Teststufe, welche horizontal auf der selben Stufe steht. Die Stufen *Modultest* und *Integrationstest* werden in den folgenden Abschnitten ausführlich behandelt. Der *System-* sowie *Abnahmetest* bekommen keinen eigenen Abschnitt, da sie programmiertechnisch von nicht so großer Bedeutung sind. Beim Systemtest werden alle Anforderungen mit Testdaten getestet. Im Abnahmetest geht es darum, dass der Kunde bzw. Anwender die Software auf seine Anforderungen hin prüft. Das V-Modell beschreibt das Vorgehen bei neu zu entwickelnder Software. Doch auch bestehende Software muss gewartet werden oder wird weiterentwickelt. In diesem Fall muss über eine neue Teststrategie nachgedacht werden. Dazu wird es einige Informationen im Abschnitt über den *Regressionstest* geben.

2.2.1. Modultest

Der Modultest ist die erste Stufe und erfolgt unmittelbar nach der Programmierung der Teilsysteme aus der Modulspezifikation. Die zu testenden Teilsysteme bilden die kleinstmögliche Einheit, wie etwa eine Klasse in der objektorientierten Programmierung. Wichtig ist, dass ein Modultest isoliert abläuft, damit Seiteneffekte von anderen Modulen das Testergebnis nicht verfälschen können. Getestet werden die Anforderungen und Schnittstellen, die in der Modulspezifikation festgelegt wurden. Konkret bedeutet dies, dass die Ausgaben auf bestimmte Eingaben kontrolliert werden. Wie im zweiten Grundsatz des Testens bereits beschrieben, ist vollständiges Testen nicht möglich. Bei der Wahl der Eingaben werden daher Äquivalenzklassen gebildet, um so ein möglichst breites Spektrum abzudecken[2]. Weiterhin soll ein Modultest nicht bloß die Richtigkeit berechneter Daten sicherstellen, sondern auch die Robustheit dieser Einheit prüfen. Dazu werden beim so genannten *Negativtesten* bewusst unzulässige Eingaben gemacht. Erwartet wird dann, dass mit einer entsprechenden Fehlerbehandlung reagiert wird und nicht etwa das Programm abstürzt. Manche Einheiten haben neben den üblichen Anforderungen auf Korrektheit und Robustheit noch weitere, wie etwa Effizienz. Solche Anforderungen sind ebenfalls bereits im Modultest zu prüfen, da Fehler dieser Art im fortgeschrittenem Stadium (Module sind bereits mit anderen integriert) schwerer zu lokalisieren sind (bei einer Zeitvorgabe bspw. muss klar sein welche Einheit zu langsam reagiert, auch hier dürfen keine Seiteneffekte den Test beeinflussen).

[2] Weitere Informationen zur Äquivalenzklassenbildung in [Lig09, Kapitel 5.1].

Beim Modultest ist der Quelltext des Moduls bekannt, daher spricht man auch vom so genannten *Whitebox-Test*. Oft werden die entsprechenden Tests von den Entwicklern geschrieben, welche auch das Modul entwickelt haben. Dies kann zu psychologischen Problemen führen, da viele Entwickler ihrer Arbeit zu optimistisch gegenüberstehen oder wichtige Testfälle vergessen bzw. nur oberflächlich behandeln.[3] Aus diesem Grund z.B. wird manchmal auch paarweise programmiert, sodass der Kollege die Einheit testen kann. Eine weitere Lösung ist der modernere *Test-first*-Ansatz. Bei dieser Variante wird zuerst der Test geschrieben. Anschließend wird das Modul solange ausprogrammiert, bis der Test fehlerfrei ist.[4] In diesem Falle wäre der Modultest ein *Blackbox-Test*, da der Quelltext zum Test nicht bekannt, bzw. noch nicht geschrieben ist.

2.2.2. Integrationstest

Nachdem die einzelnen Module entworfen worden sind und deren funktionale Korrektheit mit Modultests geprüft wurde, geht es zur nächsten Stufe, dem Integrationstest. Hierbei geht es darum das Zusammenspiel der einzelnen Komponenten zu testen. Es ist ein Trugschluss zu glauben, dass dies nicht nötig sei, da bereits alle verwendeten Teilsysteme für sich fehlerfrei funktionieren. Fehler können z.B. immer noch durch eine falsche Datenübergabe an den Schnittstellen vorhanden sein. Ein Modultest kann solche Art von Fehlern nicht aufdecken, da hier lediglich beispielhaft Daten eingegeben werden. Im späteren Einsatz werden die Teilsysteme jedoch miteinander über definierte Schnittstellen kommunizieren, sodass auch diese getestet werden müssen.

In der Praxis werden oft mehr als zwei Teilsysteme zu einem größeren System integriert. Es ist daher sinnvoll die Integrationstest ebenfalls so inkrementell zu gestalten, dass nach und nach die Integrationsstufen getestet werden, indem immer mehr Teilsysteme hinzugefügt werden. Zur inkrementellen Integration stehen drei wesentliche Strategien zur Verfügung, die da wären: *Bottom-up-Integration*, *Top-down-Integration* sowie *Ad-hoc-Integration*.

Beim Bottom-up-Verfahren wird mit den Teilsystemen auf unterster Ebene begonnen. Sie werden von höheren Teilsystemen benutzt, um Aufgaben zu lösen, benötigen aber selbst keine anderen Teilsysteme, um ihre Anforderungen zu erfüllen. Somit ergibt sich

[3] vgl. [SL10, Kapitel 2.3]
[4] Für mehr Informationen zur testgetriebenen Entwicklung siehe [Bec03].

der Name des Verfahrens daher, dass das System von elementaren (unten) bis hin zum komplexen (oben) System wächst.

Genau entgegen gesetzt verhält es sich beim Top-down-Verfahren. Hier wird mit den Systemen begonnen, welche selbst nicht von anderen benötigt werden und in der Hierarchie ganz oben stehen. Diese Systeme greifen auf die Funktionalität der untergeordneten Systeme zu. Da diese während des Integrationstests noch nicht zur Verfügung stehen, müssen sie durch so genannte *Stubs* (oder auch *Mocks*) repräsentiert werden. Sie dienen dabei als Platzhalter und simulieren das gewünschte Verhalten.

Die Ad-hoc-Integration schreibt keine Reihenfolge vor. Die Integration ergibt sich hierbei aus den Zeitpunkten der Fertigstellung einzelner Teilsysteme. Wann immer ein Modultest abgeschlossen ist, wird geprüft, ob dieses Modul mit anderen Modulen integriert werden kann. Gelegentlich müssen hierbei natürlich auch Stubs verwendet werden.

Jedes Verfahren hat gewisse Vor- und Nachteile. Es sollte aber stets auf eine frühzeitige Integration geachtet werden. Wird zu lange gewartet, sodass am ende alle Teilsysteme integriert werden, wird es mit hoher Wahrscheinlichkeit zu Fehlern kommen, welche nur schwer zu lokalisieren sind.

2.2.3. Regressionstest

Bis hierhin ging es immer um Software, welche gerade neu entwickelt wird. Vorhandene Software, welche bereits eingesetzt wird, wird jedoch auch gelegentlich verbessert, weiterentwickelt oder angepasst. Diese Änderungen an einer Software müssen natürlich auch entsprechend getestet werden. Für dieses Szenario eignet sich bspw. ein Regressionstest.

> „Das Ziel des Regressionstests ist nachzuweisen, dass Modifikationen von Software keine unerwünschten Auswirkungen auf die Funktionalität besitzen.“ [Lig09, S.192]

Gemeint sind hierbei vor allem Seiteneffekte auf bereits vorhandenen, unveränderten Code. Dadurch, dass Teile der Software verändert wurden oder neue Komponenten geschrieben wurden, kann es nämlich sein, dass Fehler, die bis jetzt maskiert waren, nun zum tragen kommen. Ein Regressionstest verwendet die Ergebnisse von Tests, die bereits

erfolgreich durchlaufen worden sind. Bei Modifikation oder Erweiterungen in der Software, werden diese Tests mit den selben Eingaben ein zweites Mal durchlaufen. Dieses Mal werden die Ist-Zustände allerdings nicht mit den Soll-Zuständen der Spezifikation verglichen. Stattdessen werden die neuen Testergebnisse mit den bereits vorhandenen verglichen. Wenn dabei keine Unterschiede auftauchen, war der Test fehlerfrei. Ein Regressionstest über das komplette System kann sehr aufwändig werden. Es ist daher wichtig, dass der Test automatisiert abläuft.

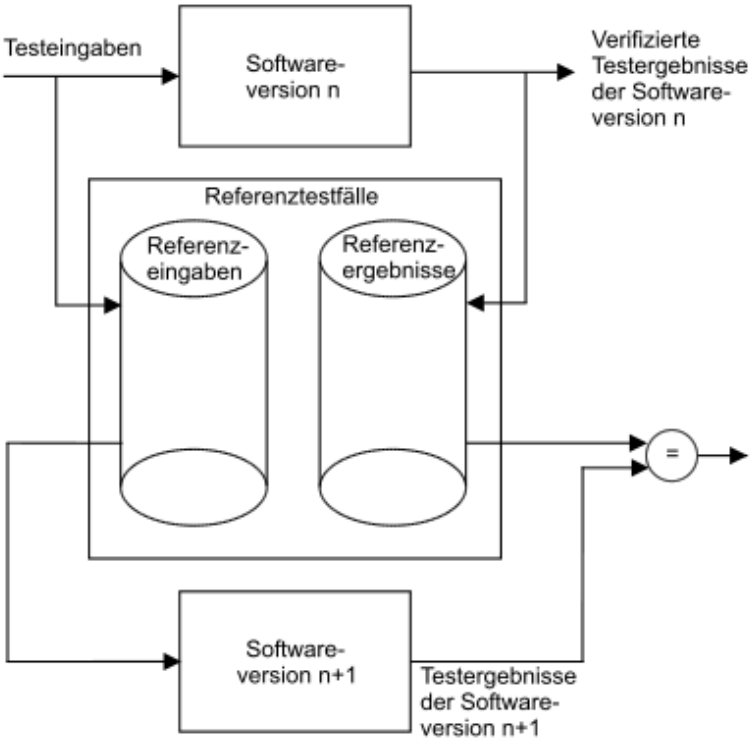


Abbildung 2.2.: Schema eines Regressionstests [Lig09, S.194]

3. Grundlagen zu DESMO-J und Simulation

Ziel dieser Arbeit soll sein, eine geeignete Testmethode für das Rahmenwerk DESMO-J zu finden. Daher wurden im letzten Kapitel bereits die Grundlagen von Softwaretests im allgemeinen vorgestellt. In diesem Kapitel geht es nun um den Grundaufbau einer Simulation und die Funktionsweise von DESMO-J. Erst wenn auch diese Grundlagen geklärt sind, lässt sich die Frage, wie DESMO-J am besten zu testen ist, geeignet diskutieren.

3.1. Simulation

Simulation mit Hilfe von Computern ist ein wichtiges Mittel geworden, um komplexe Systeme zu modellieren und analysieren. Mit Systemen ist hier ein Ausschnitt der realen Welt gemeint, welchen es zu untersuchen gilt. Die Simulation findet allerdings nicht direkt am zu untersuchenden System statt, sondern es wird zu diesem System ein Modell gebildet. Dies ist nötig, da eine Simulation am realen System oftmals zu teuer oder aufwändig ist. Außerdem könnte sie auch viel zu gefährlich sein (bspw. bei Simulation eines Reaktorunfalls) oder zu lange dauern. Um geeignete Modelle zu modellieren reduziert man hierbei die Komplexität. Dies geschieht einerseits durch Abstraktion (Verallgemeinerung) und andererseits durch Idealisierung (Außerachtlassen von Unerwünschtem und Irrationalem).[1]

Im Bereich der Simulation unterscheidet man zwischen kontinuierlicher und diskreter Simulation. Daneben gibt es noch statische Simulation, ohne Zeitbezug. Kontinuierliche Simulation, bei der meist Differenzialgleichungen kontinuierlich über die Zeit Zustandsänderungen bewirken, soll hier nicht weiter erläutert werden, da DESMO-J ein Rahmenwerk für diskrete Simulation ist. Ein wichtiger Begriff in der diskreten Simulation ist die *Entität*.

[1] nach [PK05, S. 3ff.])

> „Entities model a real system's components and may interact with each other. The terms *entity* and *object*, as used in object-oriented programming, are closely related.“ ([PK05, S.24])

Eine Entität zeichnet sich durch ihre Attribute aus und enthält einen Satz an Transformationsregeln. Ihr Verhalten ist somit über die Simulationszeit definiert.

Simulationszeit vergeht u.U. schneller oder langsamer als die Realzeit und ist von dieser isoliert zu betrachten. In der diskreten Ereignis-Simulation springt die Zeit von Ereignis zu Ereignis. Bis zum Eintreten des nächsten Ereignisses passiert im Modell nichts. Ein Ereignis wird abgearbeitet, indem die Simulationsuhr auf den Eintrittszeitpunkt gestellt wird und die Transformationsregeln der betroffenen Entitäten ausgeführt werden. Als Folge der Abarbeitung sind evtl. neue Ereignisse entstanden. Noch auszuführende Ereignisse werden, sortiert nach Eintrittszeitpunkt, auf der *Ereignisliste* vermerkt und können dort auch wieder gelöscht oder verschoben werden. Einige Änderungen erstrecken sich über mehrere Ereignisse und werden *Aktivität* genannt. Den kompletten Lebenszyklus einer Entität nennt man *Prozess*. Solange kein definiertes Abbruchkriterium erreicht wurde, läuft die Simulation i.d.R. bis die Ereignisliste keine Ereignisse mehr enthält.

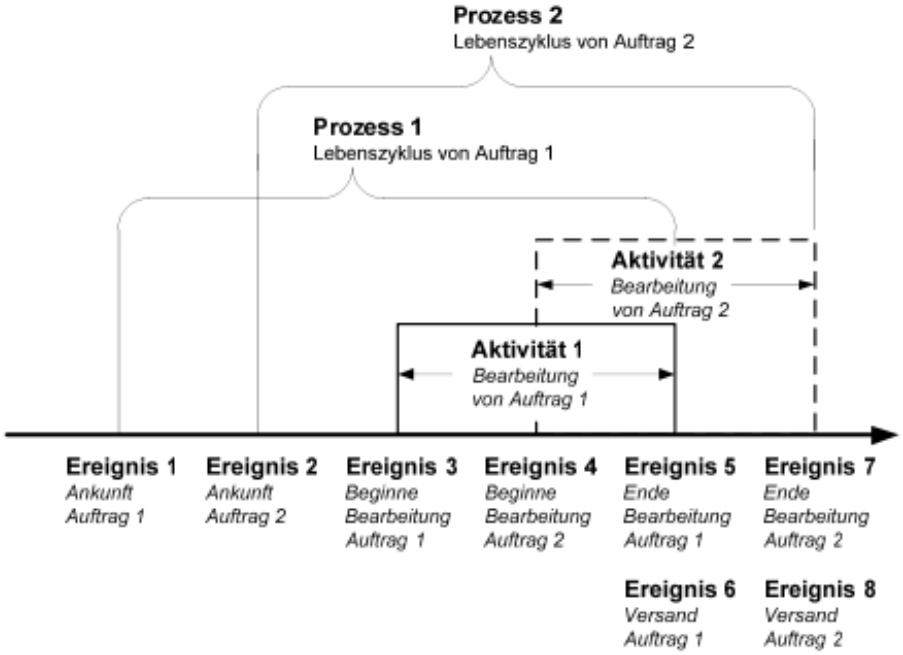


Abbildung 3.1.: Ereignisse, Aktivitäten und Prozesse [PK05, S.28]

Durch die unterschiedlichen Arten Zeitkonsumption zu beschreiben, haben sich zwei wesentliche Modellierungsstile entwickelt, nämlich *prozess-* und *ereignisorientiert*.

Bei der prozessorientierten Modellierung nimmt man die Sicht der Entitäten aus der Froschperspektive ein. Eine Entität hält als Prozess die Programmkontrolle und führt ihren Lebenszyklus aus. Dadurch, dass die Entitäten und somit auch die Prozesse miteinander interagieren, ergibt sich die Modelldynamik. Es gibt in den Prozessen aktive und passive Phasen. Ein Prozess kann die Programmkontrolle an einen anderen Prozess abgeben und passiviert sich dadurch. Bis der Prozess wieder reaktiviert wird und die Programmkontrolle erhält, vergeht Simulationszeit. Auf diese Art wird bei der prozessorientierten Simulation Zeitkonsumption realisiert. Das Abgeben der Programmkontrolle für eine bekannte Zeit wird hierbei als *hold* bezeichnet. Manchmal ist die Wartezeit allerdings nicht bekannt. In diesem Fall spricht man von einem *wait*. Der Begriff der Passivität muss hierbei noch genauer eingegrenzt werden. Bei einem hold wird impliziert, dass der Prozess für eine bekannte Zeit eine Tätigkeit ausführt. Aus Sicht des Prozesses ist er also aktiv, während er aus Sicht des Programms passiv ist, da er seine Kontrolle für diese Zeit an einen anderen Prozess abgibt. Dies ist beim wait nicht der Fall. Auch hier gibt der Prozess zwar die Kontrolle ab und ist daher aus programmtechnischer Sicht passiv, allerdings ist er dies auch aus inhaltlicher Sicht des Prozesses, da er für unbestimmte Zeit auf eine Reaktivierung wartet. Die Ablaufsteuerung bei der prozessorientierten Simulation wird also durch die aktiven und passiven Phasen der Entitäten bestimmt. Die Koordination der beteiligten Prozesse wird dabei vom *Scheduler* übernommen.

Bei der ereignisorientierten Simulation hingegen unterteilt man den Simulationsablauf in diskrete Zeitpunkte und nennt diese *Ereignis*. In einem Ereignis können bei mehreren Entitäten Zustandsänderungen durchgeführt werden. Während dieser Änderungen vergeht keine Simulationszeit. Zeitkonsumption wird durch den „Sprung“ zum nächsten Ereignis realisiert. Dadurch, dass man hier nicht die Sichtweise der Entitäten einnimmt, sondern den Simulationsablauf distanziert betrachtet, spricht man beim ereignisorientierten Modellierungsstil auch von der Vogelperspektive. Zentrales Steuerungselement ist hierbei die *Ereignisliste*. Hier können Ereignisse vorgemerkt, verschoben oder entfernt werden.

Wenn ein Simulationsrahmenwerk also sowohl die prozess-, als auch die ereignisorientierte Simulation unterstützen will, muss es zusammengefasst mindestens folgende Dinge bereitstellen:

- Eine **Simulationsuhr**
- Die **Prozess-/Ereignisliste**
- **Statistische Zähler**
- **Scheduler** zur Steuerung
- **Auswertungs-/ Reportmethode**

Darüber hinaus müssen natürlich die *Prozess-* bzw. *Ereignismethoden* definiert werden können, der *Systemzustand* muss ersichtlich sein und es muss eine *Initialisierungsmethode* sowie einen *Hauptprozess* geben. [2]

3.2. DESMO-J

Der letzte Abschnitt endete mit einer Zusammenfassung dessen, was ein Simulationsrahmenwerk mitbringen sollte. DESMO-J erfüllt diese Anforderungen, unterstützt jedoch nicht den kompletten Modellbildungszyklus, sondern legt den Schwerpunkt auf die Implementierung und Ausführung des Computermodells.[3]

DESMO-J ist ein Rahmenwerk, das aus Black- und Whitebox Anteilen besteht. Blackbox Anteile sind bereits fertige Komponenten, welche für die Simulationsinfrastruktur zu verwenden sind. Dazu gehören bspw. die Simulationsuhr oder die Ereignisliste. Whitebox Komponenten sind hingegen nicht direkt nutzbar. Sie bestehen aus abstrakten Klassen (Hot Spots) und Interfaces, welche vom Anwender implementiert und erweitert werden müssen. Sie bilden die modellspezifischen Komponenten. Abbildung 3.2 zeigt einen Überblick der Black- und Whitebox-Klassen. Die Trennung von Modell und Experiment spielt in DESMO-J eine zentrale Rolle. Durch diese Trennung ist es möglich, Modelle für andere Experimente wiederzuverwenden. Die Experimentklasse kapselt außerdem die Funktionalitäten vom Scheduler, der Ereignisliste, der Simulationsuhr sowie der Report-Erzeugung, sodass Anwender keinen direkten Zugriff auf diese haben. Bei der Modellbildung gibt es zwei grundlegende Arten von Klassen. Einmal die vormerkbaren sowie die reportfähigen. Entitäten und Ereignise sind vormerkbar und können

[2] vgl. [Pag10, Kapitel 2, Folien 19-21]
[3] siehe [Pag10, Kapitel 4, Folie 8]

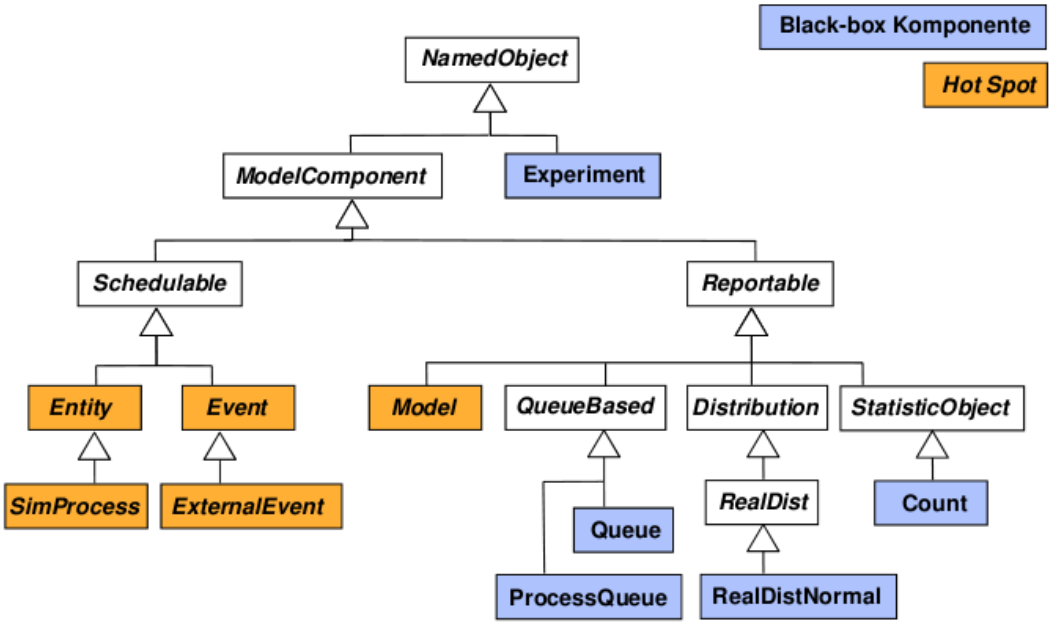


Abbildung 3.2.: Vererbungshierarchie von DESMO-J [Pag10, Kapitel 4, Folie 13]

deshalb vom Scheduler verwaltet werden. Reportfähig sind alle statischen Objekte, wie z.B. Warteschlangen oder Statistikzähler.

Das Rahmenwerk ist in verschiedene Packages unterteilt. Das wichtigste ist hierbei das Package *core*. Es bildet den Kern und ist essentiell für DESMO-J. Weitere Packages sind Erweiterungen und als optional anzusehen.

3.3. Wie ist DESMO-J zu testen?

In Kapitel 2 wurden die Grundlagen zu Softwaretests gegeben. Dort wurde vor allem klar, dass ein geeigneter Softwaretest von vielen Faktoren abhängig ist. Nachdem dann im letzten Abschnitt auch die Grundlagen von DESMO-J erläutert wurden, kann nun diskutiert werden, wie DESMO-J am besten zu testen ist.

Zuerst muss festgestellt werden, dass DESMO-J keine neu zu entwickelnde Software ist. Sie wurde in den 1990er Jahren von DESMO (Modula-2) portiert und seitdem stetig weiterentwickelt. Leider wurden in der Entwicklung keine automatisierten Tests mit

einbezogen, was nun zu erschwerten Bedingungen führt. Eine klassische Herangehensweise, wie sie bspw. das V-Modell(siehe Kapitel 2.2) vorsieht, ist daher nicht direkt anwendbar. Nachträglich sämtliche Modul- und Integrationstests nachzuholen wäre ein enormer Aufwand und führt zu gewissen Problemen, wie der Projektbericht von [BW11] bereits gezeigt hat. Im Rahmen einer Projektarbeit wurde hier eine Testumgebung für DESMO-J entwickelt. Es gab vor allem Probleme abstrakte Klassen zu testen, da diese nicht instanziiert werden konnten. Viele Methoden sind zudem als *private* oder *protected* deklariert, sodass diese auch nicht getestet werden konnten. Weiterhin sind die Module einfach schon zu sehr miteinander integriert, sodass man Klassen z.B. häufig ein Experiment übergeben muss, obwohl dies für den Test nicht relevant wäre. Ein isoliertes testen ohne Seiteneffekte ist deswegen nicht möglich.[4]

Es muss daher ein neuer Ansatz gefunden werden, um die Software in Zukunft testen zu können. Da die Software bereits Jahre lang benutzt wurde, wurde sie im Prinzip auch von Anwendern getestet. Etliche Modelle wurden bereits zufriedenstellend simuliert. Dadurch erhöht sich das Vertrauen in die Software. Nicht auszuschließen sind aber vor allem auch maskierte Fehler. Dies sind Fehler die in der jetzigen Konstellation nicht auftreten, wohl aber durch Erweiterungen(Seiteneffekte) oder Beseitigung anderer Fehler ihr Dasein finden.

Somit ist es für die Zukunft wichtig eine Teststrategie zu finden, mit der die Konsistenz der Software überprüft werden kann. Eine geeignete Methode für solche Anforderungen ist der Regressionstest (siehe Kapitel 2.2.3). Beim Regressionstest werden sämtliche automatisierten Tests mit definierten Referenzeingaben noch einmal durchlaufen und mit den Referenzergebnissen des letzten Tests verglichen. Dies führt wieder zum ursprünglichen Problem. Nämlich, dass es gar keine automatisierten Tests gibt.

Wenn man sich die Funktionsweise von DESMO-J allerdings noch einmal genau betrachtet, erkennt man, dass ein Regressionstest auch mit einer anderen Herangehensweise durchführbar ist. Bei der Simulation eines Modells mit bestimmten Eingabeparametern, erhält man am ende des Simulationslaufs einen Report. Wiederholt man diesen Lauf mit den selben Parametern, so erhält man auch den gleichen Report. Hier ist also ein klares Ein-/Ausgabe Verhalten definiert, welches man nutzen kann. Unterschiedliche Modelle nutzen üblicher Weise auch unterschiedliche Teile des Programmcodes. Aus diesem Grund kann man ein Modell auch als einen Test ansehen.

[4] siehe [BW11, Kapitel 8].

Was nun noch fehlt ist eine Möglichkeit diese Tests in Form von Modellen automatisiert ablaufen zu lassen. Für einen richtigen Regressionstest müssen die Modelle zuerst die Reports mit der stabilen Version n erzeugen. Anschließend müssen die gleichen Reports mit der zu testenden Version $n+1$ erzeugt werden, damit ein Vergleich stattfinden kann. Die dazu benötigten Werkzeuge werden im folgenden Kapitel vorgestellt.

4. Benötigte Werkzeuge

Im letzten Kapitel wurde der Frage nachgegangen wie DESMO-J zu testen ist und dabei wurde festgestellt, dass sich ein Regressionstest eignen würde. Bevor nun das Softwaretechnische Konzept für diesen Test erläutert wird, sollen zunächst ein paar Werkzeuge vorgestellt werden, welche bei dem Konzept zum Einsatz kommen werden.

4.1. Apache Ant

Software-Projekte können schnell wachsen und komplex werden. Nicht selten bestehen Projekte aus hunderten von Quelltextdateien sowie zusätzlich eingebundenen Bibliotheken. Entsprechend komplex fällt dann auch die Vorbereitung sowie die Nacharbeit aus, die getan werden muss, um die Software auszuliefern. Ant kann einem Entwickler diese mühselige Arbeit abnehmen und automatisiert durchführen. Dieser Abschnitt wird einen kurzen Überblick über das Werkzeug geben und außerdem die wichtigsten Aufgaben, welche in dieser Arbeit zum tragen kommen, vorstellen.

Ant ist ein Projekt der *Apache Software Foundation*[1] und steht für „**A**nother **N**eat **T**ool“. Es ist ein in Java geschriebenes Werkzeug zur automatisierten Ausführung bestimmter Aufgaben. Zu diesen Aufgaben zählen bspw. das Löschen, Verschieben oder Kopieren von Dateien oder Verzeichnissen sowie das Kompilieren von Quelltexten.

Die Ablaufsteuerung für ein Projekt muss in eine XML-Datei geschrieben werden. Sie wird üblicherweise *build.xml* genannt und kann von Ant ausgeführt werden. Um in dieser Build-Datei Aufgaben zu definieren, gibt es eine feste Syntax und spezifische Begriffe. Um den Aufbau eines solchen Skriptes besser zu verstehen, werden zunächst die wichtigsten Begriffe erklärt.

[1] http://www.apache.org/

Target: Ein Target wird durch einen Namen identifiziert. Dieser muss eindeutig sein und darf im gesamten Skript nur einmal vorkommen. Es dient als Container oder Hülle und kann verschiedene Aufgaben und Kommandos enthalten.

Task: Als einen Task bezeichnet man den eigentlichen Befehl der ausgeführt wird. Für die meisten Aufgaben gibt es bereits von Ant zur Verfügung gestellte Tasks, man kann aber auch eigene schreiben. Tasks befinden sich üblicher Weise in einem Target und werden durch verschiedene Attribute genau spezifiziert.

Property: In einer Property lässt sich ein Wert oder Name speichern, welcher im Skript über ${*name*} abgerufen werden kann. Dies wird z.B. bei häufig auftretenden Pfadangaben genutzt. Eine Property ist konstant und lässt sich nicht wie eine Variable überschreiben.

Datatype: Beim Datatype geht es darum nach einem bestimmten Muster Dateien zu suchen und diese als entsprechenden Datentyp bereitzustellen. Wenn z.B. alle Quelltextdateien mit der Dateiendung *.java verschoben werden sollen, dann müssen diese Dateien entsprechend gesucht und bereitgestellt werden. Dies geschieht in einem Datentyp.

Project: Als Project bezeichnet man die gesamte Ablaufsteuerung, es bildet daher auch das Wurzelelement der XML-Datei. Wichtig sind hierbei die Eigenschaften *name*, *basedir* und *default*. Mit ihnen kann man einen Namen für das Projekt, dessen Standardverzeichnis sowie den Einstiegspunkt für das Skript festlegen.

Die eigentlichen Aufgaben des Skripts werden also von den Tasks abgearbeitet. Ein Target dient folglich nur dem Zweck die Tasks zu organisieren und deren Ablaufsteuerung zu regeln. Es wurde bereits gesagt, dass ein Einstiegspunkt festgelegt werden kann. Dieser Einstiegspunkt wird in form von einem Target angegeben. Dieses Target wird zuerst betrachtet und durch dessen Eigenschaften wird der Ablauf gesteuert. Die Eigenschaften zur Ablaufsteuerung wären:

depends: Manchmal ist es notwendig, dass für die Ausführung einer Aufgabe erst etwas vorbereitet werden muss. In diesem Fall kann man mit *depends* angeben, dass ein Target von einem anderen Target abhängt. Es wird dann erst das mittels depends angegebene Target ausgeführt.

if: Für einige Aufgaben macht es nur Sinn sie auszuführen, wenn dies auch erforderlich ist. Mit der Eigenschaft *if* lässt sich eine Property angeben, sodass das Target nur ausgeführt wird, wenn diese auch existiert.

unless: Diese Eigenschaft bildet das Gegenstück zum *if*. Das Target wird nur ausgeführt, wenn die entsprechende Property nicht existiert.

```
<target name="compile" depends="clean">
    <mkdir dir="${classes.dir}"/>
    <javac srcdir="${src.dir}" destdir="${classes.dir}"
</target>
```

Listing 4.1: Beispiel für ein Target

Das Listing 4.1 zeigt ein simples Beispiel für ein Target zum kompilieren. Es beinhaltet zwei Tasks und hängt vom Target *clean* ab. Im ersten Task wird mit der Anweisung *mkdir* ein Verzeichnis angelegt, damit im zweiten Task durch *javac* die kompilierten Klassen in dieses eingefügt werden können. In beiden Tasks werden für die Pfadangaben vordefinierte Eigenschaften verwendet.

Neben den eben vorgestellten Tasks *mkdir* und *javac* gibt es noch weitere häufig verwendete, die wichtigsten sind: *copy* um Dateien zu Kopieren, *delete* um Dateien oder Verzeichnisse zu löschen, *move* um Dateien oder Verzeichnisse zu verschieben bzw. umzubenennen sowie *jar* um Dateien zu einem Jar-Archiv zu packen.

In einigen Tasks, wie zB. dem Kompilieren mittels *javac* ist es wichtig den zu verwendenden Classpath zu bestimmen. Hierzu hat man mehrere Möglichkeiten. Die direkteste Möglichkeit wäre den Classpath mit dem Attribut *classpath* im Task anzugeben. Die Selektion der im Classpath enthaltenen Elemente würde mit einem Datatype realisiert werden. Eine weitere Möglichkeit wäre über *classpathref* auf einen zuvor definierten Classpath zu referenzieren. Die Definition geschieht über das *path* Element. Hier ein kleines Beispiel dazu:

```
<path id="classpath">
    <fileset dir="libs" includes="**/*.jar"/>
</path>
```

Listing 4.2: Definition eines Classpaths

Dieser Classpath kann nun im *javac* Task mittels *classpathref="classpath"* genutzt werden. Beim kompilieren stehen dann alle Jar-Bibliotheken aus dem Verzeichnis *libs* zur Verfügung. Der Ausdruck ***/*.jar* gibt an, dass alle Dateien mit der Endung .jar in sämtlichen Unterverzeichnissen einbezogen werden sollen.

4.2. JaCoCo

JaCoCo[2] (Abkürzung für ***Ja**va **Co**de **Co**verage*) ist ein Werkzeug um die Test- bzw. Codeabdeckung in einem Software-Projekt festzustellen. Es hat den Anspruch die Standardtechnologie für Codeabdeckung zu werden und verspricht eine gute Performanz, geringe Abhängigkeit zu externen Bibliotheken sowie eine einfache Nutzbarkeit[3].

Um die Abdeckungsanalyse durchzuführen stehen Ant Tasks oder ein Maven Plug-in zur Verfügung. Auf das Maven Plug-in wird hier nicht näher eingegangen, da Maven in dieser Arbeit nicht verwendet wird. Die Ergebnisse der Analyse können automatisch im XML, HTML oder CSV Format erstellt werden. Für individuelle Reports oder anderweitige Nutzung der Ausgaben steht eine dokumentierte API[4] zur Verfügung. Für eine optimale Nutzung sollten die zu untersuchenden Klassen mit der Option *debug="true"* kompiliert worden sein. Ist dies der Fall, kann bei vorliegendem Quelltext die Abdeckung bis auf Zeilenebene aufgezeigt werden. Fehlt die Option oder der Quelltext, kann die Abdeckung lediglich auf Methodenebene festgestellt werden. In Abbildung 4.1 ist zu sehen wie detailliert die Codeabdeckung aufgezeigt werden kann. Grün markierte Zeilen bedeuten, dass sie zur Laufzeit ausgeführt wurden, rote Zeilen hingegen nicht.

Da im letzten Abschnitt des Kapitels bereits Ant als Werkzeug zur automatisierten Ablaufsteuerung vorgestellt wurde, soll nun darauf eingegangen werden, wie JaCoCo in diese Automatisierung eingebunden werden kann. Zuerst muss dazu im Wurzelelement des Build-Skripts der Namensraum für JaCoCo definiert werden. Außerdem wird mit dem *Taskdef-Tag* angegeben, wo sich die Bibliothek für die entsprechenden Ant Befehle befindet. Erst mit dieser Definition lassen sich die JaCoCo-Befehle in Ant nutzen. Möchte man jetzt bspw. eine Anwendung starten und dabei die Codeabdeckung prüfen, nutzt man das *java* Task und umgibt dieses mit den entsprechenden JaCoCo Contai-

[2] http://www.eclemma.org/jacoco/
[3] vgl. [MGCK09, Kapitel Mission].
[4] http://www.eclemma.org/jacoco/trunk/doc/api/index.html

```
537.        // set class variables for basic messagetypes
538.        try
539.        {
540.            tracenote = (Class<TraceNote>) Class
541.                    .forName("desmoj.core.report.TraceNote");
542.            debugnote = (Class<DebugNote>) Class
543.                    .forName("desmoj.core.report.DebugNote");
544.            errormessage = (Class<ErrorMessage>) Class
545.                    .forName("desmoj.core.report.ErrorMessage");
546.            reporter = (Class<Reporter>) Class
547.                    .forName("desmoj.core.report.Reporter");
548.        } catch (ClassNotFoundException cnfEx)
549.        {
550.            System.err.println("Can not create Experiment!");
551.            System.err.println("Constructor of desmoj.core.Experiment.");
552.            System.err.println("Classes are probably not installed correctly.");
553.            System.err.println("Check your CLASSPATH setting.");
554.            System.err.println("Exception caught : " + cnfEx);
555.        }
```

Abbildung 4.1.: Auszug eines Reports

```
<project name="Example" xmlns:jacoco="antlib:org.jacoco.ant">

<taskdef uri="antlib:org.jacoco.ant" resource="org/jacoco/ant/antlib.xml">
   <classpath path="path_to_jacoco/lib/jacocoant.jar"/>
</taskdef>
```

Listing 4.3: Vorbereitungen, um JaCoCo zu nutzen

nern *<jacoco:coverage>* und *</jacoco:coverage>*. Diese Anweisung sorgt dafür, dass der eingeschlossene Bereich automatisch auf Codeabdeckung überprüft wird. Ohne weitere Angaben wird das Ergebnis dieser Abdeckungsanalyse in eine Datei namens *jacoco.exec* geschrieben. Diese Datei ist noch nicht lesbar, sondern hält lediglich die Informationen in einem Zwischenformat. Wie diese Informationen nun aufbereitet werden, hängt vom Interesse und Einsatzgebiet des Benutzers ab. Wie schon beim Prüfen der Codeabdeckung, stehen auch für die Erzeugung der Reports Ant-Befehle bereit. Listing 4.4 zeigt wie diese zu nutzen sind. Über das *classfiles*-Tag lässt sich bestimmen, welche Klassen in der Abdeckungsanalyse mit einbezogen werden sollen. Dies ist nützlich, falls das gesamte Projekt unüberschaubar groß ist und man sich nur für bestimmte Klassen interessiert. Mit dem *sourcefiles*-Tag kann man die entsprechenden Quelltexte zu den Klassen angeben, damit in den Reports auch entsprechend in die Tiefe gegangen werden kann. Die Tasks *html* und *xml* erzeugen dann den Report. Wie so ein HTML Report aussehen kann zeigt Abbildung 4.2. Aufgelistet werden sämtliche Packages, die in den angegebenen Classfiles zu finden waren. Klickt man auf eines dieser Packages bekommt man eine

```
<jacoco:report>
        <executiondata>
                <file file="jacoco.exec"/>
        </executiondata>
        <structure name="Regressionstest-Abdeckung">
                <classfiles>
                <fileset dir="${classes.dir}"/>
                </classfiles>
                <sourcefiles>
                <fileset dir="${src.dir}"/>
                </sourcefiles>
        </structure>
        <html destdir="report"/>
        <xml destfile="report.xml"/>
</jacoco:report>
```

Listing 4.4: Ant-Tasks um JaCoCo Reports zu erzeugen

Regressionstest

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
desmoj.core.simulator		23%		24%	1.715	2.185	3.434	4.983	454	781	30	70
desmoj.core.report		38%		25%	461	610	1.171	1.871	170	279	34	67
desmoj.core.advancedModellingFeatures		33%		36%	358	526	1.025	1.615	87	195	1	8
org.apache.commons.math.analysis		0%		0%	309	309	801	801	125	125	20	20
desmoj.core.dist		14%		9%	368	429	753	922	172	227	40	58
desmoj.core.statistic		29%		28%	261	354	581	945	74	146	4	13
org.apache.commons.collections.map		19%		14%	296	335	559	688	147	174	17	23
org.apache.commons.math.util		0%		0%	234	234	428	428	88	88	5	5
org.apache.commons.math.distribution		0%		0%	256	256	475	475	154	154	18	18
org.apache.commons.math		0%		0%	54	54	133	133	47	47	9	9
org.apache.commons.math.special		0%		0%	56	56	75	75	22	22	5	5
org.apache.commons.collections.list		67%		64%	54	134	84	279	16	56	0	3
org.apache.commons.collections.keyvalue		0%		0%	21	21	25	25	11	11	3	3
desmoj.core.advancedModellingFeatures.report		93%		38%	26	40	19	274	1	15	0	5
desmoj.core.util		0%		0%	13	13	23	23	9	9	2	2
org.apache.commons.collections.iterators		0%		n/a	18	18	22	22	18	18	3	3
desmoj.core.exception		89%		n/a	2	4	3	8	2	4	1	3
Total	52.447 of 69.545	25%	5.044 of 6.419	21%	4.502	5.578	9.611	13.567	1.597	2.351	192	315

Abbildung 4.2.: HTML Report der Abdeckungsanalyse

Klassenübersicht, danach die Methodenübersicht und schließlich den Quelltext, falls dieser vorhanden ist. Rechts daneben ist stets die Statistik zur Abdeckung angegeben. Sie enthält verschiedene Werte und Metriken, welche im Folgenden aufgezählt und erklärt werden.

Instructions: Zuerst wird angegeben wie viele programmiertechnische Anweisungen vorhanden sind und wie viele davon ausgeführt wurden. Dieser Wert wird prozentual angegeben und durch einen Balken visualisiert. Der grüne Anteil steht dabei für die ausgeführten Anweisungen. Rot beinhaltet die nicht ausgeführten. Das Package (bzw. Klasse oder Methode) mit den meisten Anweisungen steht dabei an erster Stelle und gibt die Balkenbreite vor. Die Balken der folgenden Einträge fallen etwas schmaler aus, da sie weniger Anweisungen enthalten und im Verhältnis angepasst werden.

Branches: Gemeint sind hiermit die Verzweigungen im Programmfluss. Dies können z.B. if-Anweisungen sein. Angegeben wird dieser Wert wieder in Prozenten, unterstützt durch einen Balken. Dieser Wert sagt aus, wie viele Verzweigungen im Programmflus möglich wären und wie viele davon tatsächlich durchlaufen wurden.

Cxty: Dieser Wert ist die Abkürzung für *complexity*, genauer gesagt *cyclomatic complexity* und beschreibt die so genannte *zyklomatische Komplexität* oder auch *McCabe-Metrik*. Diese Metrik basiert auf dem Kontrollflussgraphen[5] und gibt die Komplexität des Kontrollflusses in einem bestimmten Abschnitt an (zur genauen Berechnung und Bedeutung dieser Metrik siehe [McC76]).

Lines: Hierbei geht es schlicht und ergreifend um die Anzahl der Zeilen im Programmcode und wie viele davon aufgerufen wurden. Wie schon bei der *complexity* werden die Werte nun in absoluten Zahlen dargestellt.

Methods: Gibt an wie viele Methoden in dieser Klasse (oder in diesem Package) zur Verfügung stehen und wie viele davon ausgelassen wurden.

Classes: Äquivalent zu *methods* wird hier angegeben wie viele Klassen in einem Package vorhanden sind und wie viele davon genutzt, bzw. nicht genutzt wurden.

In der abschließenden Zeile werden alle Werte summiert und zusammengefasst.

[5] http://de.wikipedia.org/wiki/Kontrollflussgraph

Ein automatisch generierter Report enthält jede Menge an Informationen. Nicht immer sind alle Informationen nötig und manchmal möchte man sie weiter verarbeiten. Zu diesem Anlass gibt es eine API, welche einem die Möglichkeit gibt die Statistiken in seinen eigenen Java Programmen zu verwenden. Wie aus der *Exec-Datei*, welche die Informationen hält (*coverage.exec* o.Ä.), die Informationen extrahiert werden können, soll im Folgenden kurz erläutert werden. Zuerst muss die Datei mit der Methode *loadExecutionData* eingelesen werden (siehe Listing 4.5). Die dabei verwendeten Klassen *ExecutionDataReader*, *ExecutionDataStore* sowie *SessionInfoStore*, werden von JaCoCo mitgeliefert und sollen an dieser Stelle nicht weiter erklärt werden.

```
private void loadExecutionData() throws IOException {
        //executionDataFile = coverage.exec
        final FileInputStream fis = new FileInputStream(executionDataFile);
        final ExecutionDataReader executionDataReader = new
                ExecutionDataReader(fis);
        executionDataStore = new ExecutionDataStore();
        sessionInfoStore = new SessionInfoStore();

        executionDataReader.setExecutionDataVisitor(executionDataStore);
        executionDataReader.setSessionInfoVisitor(sessionInfoStore);

        while (executionDataReader.read()) {
        }
        fis.close();
}
```

Listing 4.5: Quelltext um coverage.exec einzulesen

Nachdem die Datei eingelesen wurde, kann damit begonnen werden, dessen Struktur zu analysieren, dies geschieht in der Methode *analyzeStructure* (Listing 4.6). Die Klassen *Analyzer* und *CoverageBuilder* sind wieder Teil von JaCoCo und werden nicht näher erläutert. Was man von der Methode erhält ist ein Objekt des Typs *IBundleCoverage*. Aus diesem Objekt lassen sich nun sämtliche Informationen der Abdeckungsanalyse ableiten. Um aus diesem *bundle* z.B. die Packages auszulesen genügt folgende Zeile:

```
ArrayList<IPackageCoverage> packages =
        (ArrayList<IPackageCoverage>) bundleCoverage.getPackages();
```

Um aus den Packages wiederum die Klassen zu bekommen macht man weiter mit:

```
for(IPackageCoverage p : packages) {
        ArrayList<IClassCoverage> classes =
                (ArrayList<IClassCoverage>) p.getClasses();
        ...
}
```

```
private IBundleCoverage analyzeStructure() throws IOException {
        final CoverageBuilder coverageBuilder = new CoverageBuilder();
        final Analyzer analyzer = new Analyzer(executionDataStore,
                        coverageBuilder);

        final FileInputStream fis = new FileInputStream(desmoLibFile);
        analyzer.analyzeAll(fis);

        return coverageBuilder.getBundle(title);
}
```

Listing 4.6: Quelltext um Informationen zu extrahieren

Um auf der jeweiligen Ebene schließlich an die Statistiken zu gelangen, stehen dann diverse getter-Methoden zur Verfügung.

4.3. XMLUnit

XMLUnit[6] ist eine Bibliothek, welche das Testen von Software unterstützt und vereinfachen kann. Wie wichtig das Testen von Software ist, wurde bereits in Kapitel 2 herausgestellt. Um in Java Tests zu schreiben, wird als Testrahmenwerk in erster Linie JUnit[7] verwendet. Es ist ein sehr mächtiges Rahmenwerk und bietet viele Methoden, um Tests zu schreiben und Datentypen zu vergleichen. An einem gewissen Punkt stößt allerdings auch JUnit an seine Grenzen. Dies ist z.B. bei externen Ressourcen, wie etwa XML-Dokumenten der Fall. Denn ein XML-Dokument ist nicht bloß eine herkömmliche Textdatei, in der Zeichenketten hintereinander weggeschrieben werden. Bei XML gibt es eine definierte Syntax und eine gewisse Semantik, die sich entsprechend aus dem Kontext ergibt. Es kann also durchaus vorkommen, dass zwei Dokumente auf den ersten Blick vollkommen verschieden aussehen, bei genauerer Betrachtung jedoch die gleiche Bedeutung haben und den selben Inhalt transportieren. Tabelle 4.1 wird dieses Phänomen verdeutlichen.

Zu sehen sind zwei Ausschnitte aus XML-Dokumenten, welche zunächst verschieden aussehen, da sie ein unterschiedliches Erscheinungsbild aufweisen. Die Syntax weist in keinem Dokument einen Fehler auf. Beide Dokumente beschreiben ein Buch mit zwei Kapiteln. In den Kapiteln steht der gleiche Inhalt, lediglich die Reihenfolge der Kapitel

[6] http://xmlunit.sourceforge.net/
[7] http://www.junit.org/

<Buch>
  <Kapitel Eins>
  Inhalt von Kapitel Eins
  </Kapitel Eins>
  <Kapitel Zwei>
  Inhalt von Kapitel Zwei
  </Kapitel Zwei>
</Buch>
	<Buch>
	  <Kapitel Zwei>
	  Inhalt von Kapitel Zwei</Kapitel Zwei>
	  <Kapitel Eins>Inhalt von Kapitel Eins</Kapitel Eins>
	</Buch>

Tabelle 4.1.: Beispiel für zwei XML-Dokumente mit korrekter Syntax und gleicher Semantik

ist vertauscht. Doch eine unterschiedliche Reihenfolge muss bei XML nicht grundsätzlich falsch sein. Dieses kleine Beispiel macht bereits deutlich, dass XML-Dokumente nicht so einfach zu testen sind. Ein einfacher Zeichenvergleich reicht hierbei nicht aus. Aus diesem Grund wurde mit XMLUnit ein Werkzeug entwickelt, welches für diesen Einsatz gedacht ist. Wie der Name bereits vermuten lässt bietet XMLUnit eine Erweiterung für JUnit an. Mit den Klassen *XMLTestCase* und *XMLAssert* lassen sich Testfälle im gewohnten JUnit-Stil generieren. XMLUnit bietet noch viele weitere Klassen, dessen Funktionalitäten auch ohne JUnit genutzt werden können. In dieser Arbeit wird nicht viel Funktionalität von XMLUnit gebraucht werden. Daher soll im Weiteren auch nur auf einen kleinen Teil eingegangen werden[8].

Benötigt wird für diese Arbeit eine Möglichkeit zwei XML-Dokumente einzulesen und dessen Inhalt auf Unterschiede zu prüfen. Diese Aufgabe kann bereits von folgenden Zeilen gelöst werden:

```
try {
        FileReader fr1 = new FileReader(pfadRef);
        FileReader fr2 = new FileReader(pfadNew);

        Diff diff = new Diff(fr1, fr2);
        boolean similar = diff.similar();
        boolean identical = diff.identical();
} catch (FileNotFoundException e) {
        e.printStackTrace();
} catch...
```

Listing 4.7: Quelltext um XML-Dokumente zu vergleichen

[8]Für den kompletten Funktionsumfang von XMLUnit siehe [BB09]

Wie zu erkennen ist, wird bei einem Vergleich in XMLUnit zwischen *simililar* und *identical* unterschieden. Similar bedeutet hierbei, dass beide Dokumente semantisch gleich sind und den selben Inhalt transportieren. Identical ist ein schärferes Kriterium und beachtet auch die Reihenfolge der Einträge. Bei der Klasse *Diff* wurde Wert auf Performanz gelegt. Der Vergleich bricht daher bereits ab, sobald der erste Unterschied festgestellt wurde. Wenn gewünscht wird, dass die Dokumente komplett verglichen werden, muss mit der Klasse *DetailedDiff* weitergemacht werden. [9]. Sie kann auch gleich die gefundenen Fehler benennen, wie folgendes Beispiel zeigt.

```
DetailedDiff detDiff = new DetailedDiff(diff);
List<?> differences = detDiff.getAllDifferences();
for (Object object : differences) {
        Difference difference = (Difference)object;
        System.out.println(difference);
}
```

Listing 4.8: Quelltext für Unterschiede in XML-Dokumenten

[9] vgl. [BB09, Example 4 und Example 5]

5. Softwaretechnisches Konzept

An dieser Stelle sind nun die Grundlagen geklärt. Auch die Frage nach einer geeigneten Teststrategie für DESMO-J wurde beantwortet. Es soll ein Regressionstest werden, für welchen verschiedene Modelle einzusetzen sind. Wie der Test im einzelnen aufgebaut ist und mit welchen softwaretechnischen Mitteln er realisiert werden soll, wird Thema dieses Kapitels sein. Dazu werden zunächst die Anforderungen an den Test sowie das Testwerkzeug formuliert und anschließend darauf eingegangen, wie diese Anforderungen zu erfüllen sind.

5.1. Anforderungen an die Software und den Test

Die folgenden Abschnitte behandeln die Anforderungen an die Software und erklären, wieso sie für den Test wichtig sind.

5.1.1. Automatisierung

Bei einem Regressionstest werden meist sehr viele Tests durchlaufen, deren Ergebnisse es gilt auf Gleichheit zu prüfen. Die Ausgaben der Tests sind die Reports der Simulationsabläufe. Konkret geht es bei jedem Modell um die Ausgaben *Trace* und *Report*. Besonders im Trace, aber auch schon im Report, stecken sehr viele Daten und Informationen, die es alle zu prüfen gilt. Würde diese Prüfung manuell durchgeführt werden, kann man sehr leicht durcheinander kommen. Es wäre zudem eine sehr ermüdende Tätigkeit.

> Ermüdungserscheinungen „führen dazu, dass aufgetretene Diskrepanzen häufig nicht bemerkt werden. Dies ist besonders kritisch, weil Fehler in Regressionstests selten sind.“ [Lig09, S.193]

Doch nicht bloß der Vergleich von Reports wäre händisch ein großer Aufwand. Auch das Starten der Simulationsläufe mit anschließendem Sortieren der Ergebnisse, würde sich als umständlich erweisen. Jedes Modell müsste zweimal mit verschiedenen Versionen gestartet werden. Außerdem müssten zwischen den Simulationsläufen die Reports verschoben oder umbenannt werden, damit die Referenzreports nicht durch die neuen Überschrieben werden.

Es muss also vom Starten der Simulationsläufe, bis hin zum Vergleich der Reports ein möglichst hoher Automatisierungsgrad gegeben sein. Nur so können die Tests wirtschaftlich und zuverlässig ablaufen.

5.1.2. Erweiterbarkeit

DESMO-J ist eine Software, die regelmäßig durch neue Funktionalitäten erweitert wird. Daher ist es notwendig, dass auch der Test in regelmäßigen Zeitabständen erweitert wird. Wie gut der Test das System abdeckt, hängt letztendlich von den Modellen ab. Wenn ein Modell nur einen Teil an Funktionalität von DESMO-J benutzt, kann man logischer Weise über die restlichen Funktionalitäten keine Aussage treffen. Wenn neue Funktionalität eingebaut wird, muss also entweder auch ein neues Modell in den Test eingefügt werden, welches diese Funktionalität benutzt oder ein bestehendes Modell muss dahingehend erweitert werden, dass es die neuen Funktionen nutzt.

Eine wichtige Anforderung ist daher, dass der Test erweiterbar ist. Hierfür muss es möglich sein ohne großen Aufwand neue Modelle in den Test zu integrieren, damit diese die Abdeckung erweitern.

5.1.3. Vollständigkeit

Im zweiten Grundsatz zum Testen von Software (siehe Kapitel 2.1) wurde bereits beschrieben, dass vollständiges Testen nicht möglich ist. Ziel eines jeden Tests ist aber auch eine möglichst hohe Testabdeckung. Damit der Test also überhaupt eine gewisse Aussagekraft hat, müssen alle Programmteile wenigstens einmal durchlaufen worden sein. In der Anforderung zur Erweiterbarkeit wurde schon angedeutet, dass die Abdeckung des Regressionstest von den Modellen abhängt. Sie bestimmen, welche Teile des Frameworks

genutzt werden. Um eine hohe Abdeckung zu erreichen, muss es folglich mindestens ein Modell geben, welches das Framework so stark wie möglich ausreizt. Da DESMO-J für viele Bereiche der Simulation Funktionalität anbietet, wird es schwer sein ein solches Modell zu finden. Möglich ist aber mehrere Modelle zu benutzen, welche für sich einen gewissen Teil nutzen und in ihrer Summe die Gesamtheit des Frameworks ausnutzen. Abschnitt 5.3 *Fehlerlokalisation* wird zeigen, dass der Ansatz viele kleinere Modelle zu verwenden, sogar deutlich besser geeignet ist.

5.2. Konzept zur Realisierung

Es soll nun das Konzept vorgestellt werden, welches die Implementation des Regressionstest ermöglicht und dabei den Anforderungen gerecht wird.

5.2.1. Automatisierung

Als erstes geht es darum, einen möglichst hohen Automatisierungsgrad zu erreichen. Eine Möglichkeit, um Vorgänge in Java automatisiert ablaufen zu lassen, bietet das Werkzeug Ant, welches bereits in Kapitel 4.1 vorgestellt wurde. Es eignet sich daher ideal, um die Steuerung des Tests zu übernehmen und soll hierfür eingesetzt werden. Benötigt werden für einen Testlauf die Referenzversion von DESMO-J sowie die zu untersuchende Version. Darüber hinaus natürlich sämtliche Modelle, welche in dem Test mit einbezogen werden sollen. Der Ablauf, den das entsprechende Ant-Skript nun steuern muss, gliedert sich in folgende Punkte:

1. Kompilieren der Modelle mit der Referenzversion von DESMO-J
2. Modelle starten
3. Referenzreports verschieben
4. Kompilieren der Modelle mit der neuen Version von DESMO-J
5. Modelle starten

6. Reports verschieben

Für das Kompilieren gibt es in Ant einen vorgefertigten Task. Man kann dort über die Eigenschaft *classpathref* angeben welcher Classpath beim Kompiliervorgang mit einbezogen werden soll. Für die beiden DESMO-J Versionen müssen folglich zwei verschiedene *path*-Elemente angelegt werden, welche sich durch die *id* unterscheiden. Zum starten der Modelle benutzt man das *java*- und zum verschieben der Reports das *move*-Task. An dieser Stelle ist der Test jedoch noch nicht fertig, denn es fehlt ein entscheidender Schritt.

7. Reports vergleichen

Hierzu muss von Ant, nach dem letzten Verschieben der Reports, eine Java Klasse aufgerufen werden, welche den Vergleich durchführt. Folgende Aufgaben muss diese Klasse lösen können:

- Einlesen von Reports aus einem Verzeichnis
- Vergleich von zwei XML-Dokumenten auf Gleichheit
- Feststellen, ob die Reportverzeichnisse gleich sind
- Bei Ungleichheit die entsprechenden Modelle benennen

Die Klasse ließe sich theoretisch schon mit dem Java Development Kit alleine realisieren. Für den Vergleich der XML-Dokumente wird allerdings die Bibliothek *XML-Unit* zum Einsatz kommen, da es speziell für diesen Fall geschaffen wurde, effizient arbeitet und somit Zeit bei der Entwicklung spart (siehe Kapitel 4.3).

Die Ablaufsteuerung des Tests lässt sich also mit Hilfe von Ant automatisiert gestalten. Abbildung 5.1 fasst die bisherigen Überlegungen in einem Schema zusammen.

5.2.2. Erweiterbarkeit

Die folgenden Ausführungen behandeln die nächste Anforderung, nämlich *Erweiterbarkeit*. Es soll ohne großen Aufwand möglich sein neue Modelle in den Test zu integrieren.

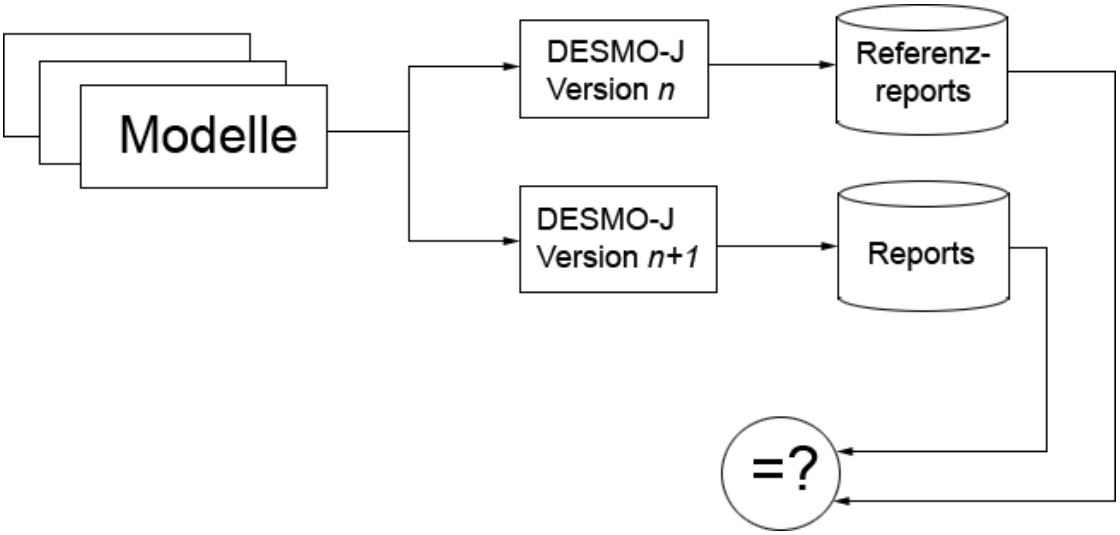


Abbildung 5.1.: Ablaufschema des Regressionstests

Daher wird der zweite Punkt *Modelle starten* weiter angepasst. Es wurde nämlich lediglich gesagt, dass die einzelnen Modelle von Ant, jeweils mit dem *java*-Task, gestartet werden sollen. Hierzu müsste für jedes Modell ein eigener Task definiert werden. Die *build.xml* würde dadurch unnötig aufgebläht werden. Das Starten der Modelle soll daher in einer Java Klasse verwaltet werden und das Ant-Skript ruft dann statt der Modelle die Verwaltungsklasse auf.

Erst einmal reicht es aus, wenn die Klasse alle Modelle startet. Dies sind entweder die von *Modell* abgeleiteten Klassen mit einer Main-Methode oder spezielle Runner Klassen, welche für dieses Modell geschrieben wurden.

Für die spätere Fehlerlokalisation, soll die Klasse noch mehr Funktionalität bekommen. Es sollte möglich sein, nur eine Teilmenge der zur Verfügung stehenden Modelle zu starten. Daher muss eine weitere Methode geschaffen werden, welche eine Liste mit Modellen entgegen nimmt und diese dann mit Hilfe von Reflection ausführt. Eine Java Klasse, namens *ModelList*, hält die nötigen Informationen zum Starten der Modelle. Sie soll als eine Art Verzeichnis dienen und enthält zu jedem Modell- bzw. Experimentnamen die entsprechende Startklasse. Denn dadurch, dass es diese beiden Eigenschaften gibt, entsteht ein Problem. Die Report-Dateien, welche am ende eines jeden Simulationslaufs generiert werden, tragen zur eindeutigen Identifikation u.A. den Experimentnamen. Im siebten Punkt des Testablaufs, *Reports vergleichen*, wurde die Aufgabe gefordert bei

Ungleichheit die entsprechenden Modelle zu benennen. Dies hat den Hintergrund, dass die entsprechenden Modelle zur Fehlerfindung nochmals isoliert ausgeführt werden sollen. Die Klasse, welche den Vergleich der Reports vornimmt, hat entsprechend auch nur die Report-Dateien zur Verfügung. Wenn also ein Unterschied festgestellt wird, lässt sich über den Dateinamen des Reports auch der Experimentname des Modells ableiten. Unklar ist jedoch mit welcher Klasse dieses Experiment gestartet wurde. Wenn nach dem Vergleich eine Liste mit Namen der fehlerhaften Experimente erstellt wird, könnten diese also nicht gestartet werden. Durch das Modellverzeichnis kann die Verwaltungsklasse allerdings auch eine Liste von Modellen anhand des Experimentnamens starten.

Zusammengefasst sieht das Konzept zum Starten der Modelle also folgendermaßen aus: Es gibt eine Liste von Modellen, welche alle nötigen Informationen zu den vorhandenen Modellen enthält. Dazu gehört der Experimentname sowie die Startklasse des Modells. Zum starten der Modelle wird eine Starterklasse verwendet, welche entweder alle Modelle aus der Liste startet oder nur eine ausgewählte Teilmenge. Möchte man den Test also in Zukunft um Modelle erweitern, so muss man lediglich den Experimentnamen sowie die Startklasse in die Liste der Modelle eintragen.

Die Anforderung auf Vollständigkeit ist pauschal nicht zu beantworten und wird zu keinem Zeitpunkt hundert Prozentig zufriedenstellend sein. Wie vollständig der Test ist, entscheidet sich letztendlich an den Anwendungsgebieten der Modelle. Die Wahl des richtigen Modells, bzw. mehrerer Modelle wird auch im nächsten Abschnitt eine wichtige Rolle spielen. Aus diesem Grund werden die Überlegungen zur Erfüllung der Vollständigkeit dort behandelt.

5.3. Fehlerlokalisation

Bis jetzt wurde das Konzept zur Testdurchführung vorgestellt. Dabei wurde auf die verlangten Anforderungen eingegangen, sodass diese bis auf Vollständigkeit erfüllt wurden. Nachdem ein Testlauf beendet wurde, liefert er ein Ergebnis. Das Resultat hängt von den verglichenen Reports ab. Wenn keine Unterschiede festgestellt wurden, war der Test positiv und Fehler konnten nicht nachgewiesen werden. Wenn aber in bestimmten Reports Unterschiede festgestellt wurden, fällt der Test negativ aus. Dies bedeutet, dass in einer Version von DESMO-J bestimmte Quelltextabschnitte nun zu einem anderen Experimentablauf führen. Dieses Verhalten lässt sich auf zwei Arten interpretieren. Entweder

es wurde gezielt ein Fehler ausgebessert, somit ist das Verhalten natürlich gewünscht oder aber es tritt unerwartet auf und ist als fehlerhaft einzustufen. Welcher Fall vorliegt, kann von dem Test nicht beantwortet werden. Daher wird der Test in diesem Fall immer als negativ eingestuft. Da der Test vor allem als Unterstützung für die Entwickler gedacht ist, werden diese auch am besten wissen, ob sie gerade Fehler ausgebessert haben oder nicht. Falls allerdings unerwartet Unterschiede festgestellt wurden, stellt sich die entscheidende Frage, wo im Quelltext diese Fehler gemacht wurden.

Wenn ein Fehler festgestellt wurde, kann er sich logischer Weise nur in einem Quelltextabschnitt befinden, der auch zur Laufzeit des Programms zur Ausführung kam. Es ist daher unbedingt nötig nach dem Testlauf eine Übersicht über die Quelltextabdeckung zu bekommen. Nur so ist es möglich, Rückschlüsse über die Fehlerquelle zu ziehen. Ein Werkzeug für solche Abdeckungsanalysen des Quelltexts wurde in Kapitel 4.2 bereits vorgestellt und soll nun auch eingesetzt werden. Da dieser Schritt in erster Linie zur Fehlerfindung dient, liegt es nahe diesen auch nur zu gehen, falls der Test negativ ausfällt. Da die Abdeckungsanalyse aber auch ein Indikator dafür ist, wie gut die Modelle das Framework ausnutzen, soll diese Analyse schon beim ersten Durchlauf aller Modelle gemacht werden. So lässt sich bei einem positiven Testergebnis besser bewerten, wie aussagekräftig dieser Test überhaupt war.

Bei einem negativen Testausgang hingegen wird es nicht ausreichen die Quelltextabdeckung des gesamten Testlaufs zu betrachten. Im Idealfall wurden die Modelle so gewählt, dass sie in ihrer Summe das gesamte Framework ausreizen. Wenn dann eine Fehlerquelle zu suchen ist, kommt man mit der Abdeckungsanalyse nicht weit. Bei der Abdeckungsanalyse wurde die Abdeckung für den gesamten Test aufgezeichnet. Daher sind auch sämtliche Modelle enthalten, welche in ihrer Gesamtheit das komplette Framework benutzen. Die Abdeckungsanalyse wird also anzeigen, dass sehr viele Bereiche durchlaufen wurden. Die meisten allerdings von Modellen, bei denen keine Unterschiede festgestellt wurden. Folglich wäre es sinnvoller die Analyse ein zweites mal zu starten und zwar nur mit den Modellen, die unterschiedliche Reports verursachen. Diese Schlussfolgerung verdeutlicht auch die Notwendigkeit, dass im siebten Schritt (Reports vergleichen) gefordert wurde, dass bei ungleichen Reports die entsprechenden Modelle benannt werden müssen. An dieser Stelle kann nun auch die Funktionalität der Starterklasse genutzt werden, nur eine bestimmte Teilmenge an Modellen zu starten.

Gemäß dem Fall, dass nun einige Reports Ungleichheiten aufweisen, kann aufgrund der Dateinamen auf den Experimentnamen zurückgeschlossen werden. Die Namen der betroffenen Experimente werden nun an die Starterklasse übergeben. Diese führt den Regressionstest erneut durch. Der Unterschied zum ersten Durchlauf ist, dass nun nur die Modelle gestartet werden, welche unterschiedliche Reports erzeugen und deswegen vermutlich auch fehlerhafte Quelltextabschnitte aufrufen. Dies ist der erste Schritt zur Lokalisation der fehlerhaften Abschnitte. Alle Quelltextbereiche, die bei diesem zweiten Durchgang nicht in der Abdeckungsanalyse auftauchen, wurden folglich von Modellen aufgerufen, bei denen keine Fehler nachgewiesen werden konnten. Im Idealfall wurden die Quellen für potentielle Fehler nun erheblich reduziert. Es wird sich mit diesen Informationen höchstwahrscheinlich noch keine genaue Aussage darüber treffen lassen, wo genau der Fehler zu vermuten ist. Die Technik hinter dem Konzept bietet jetzt allerdings alle nötigen Grundlagen um weitere Untersuchungen zu starten. Da es nun möglich ist beliebige Modelle zu starten und deren Quelltextabdeckung zu analysieren, lassen sich weitere Strategien zur Fehlerfindung praktizieren.

Fehler, die durch einen Regressionstest aufgedeckt werden, kommen meistens durch Seiteneffekte zustande.[1] Dies kann passieren, wenn durch Änderungen in der Software nun andere Programmteile durchlaufen werden oder eine bedingte Anweisung dadurch eine andere Abzweigung nimmt. Derartige Fehler könnten wie folgt aufgedeckt werden: Zuerst identifiziert man die Modelle, in welchen die Fehler aufgetreten sind. Man lässt sie dann mit beiden Versionen von DESMO-J durchlaufen, sodass man zu jedem Lauf eine Abdeckungsanalyse bekommt. Die Analyseberichte werden zu einem großen Teil identisch sein. Da aber unterschiedliche Reports von DESMO-J erzeugt wurden, muss es auch in gewissen Bereichen des Quelltextes Unterschiede geben. Die so gefundenen Unterschiede wären ein potentieller Kandidat für eine Fehlerursache.

Eine weitere Strategie wäre, sich die unterschiedlichen Modellierungsstile zu Nutze zu machen. Wie in Kapitel 3 bereits vorgestellt, gibt es in der Simulation vor allem zwei Modellierungsstile die sich durchgesetzt haben. Dies sind der prozess- und der ereignisorientierte Ansatz, welche beide von DESMO-J unterstützt werden. Einige Modelle werden sowohl im ereignis- als auch im prozessorientierten Stil programmiert. Die Ausgaben der Reports müssen hierbei natürlich gleich sein, da lediglich die Sichtweise eine andere ist. Inhaltlich sagen die Modelle aber das gleiche aus und haben folglich auch das gleiche Ein-/Ausgabeverhalten. Wenn in einem Modell also unterschiedliche Reports festgestellt

[1] vgl. Abschnitt 2.2.3

wurden, kann man u.U. folgendes Anwenden: Man stellt fest mit welchem Modellierungsstil das Modell entwickelt wurde und prüft, ob für dieses Modell auch eine Entwicklung nach dem gegenteiligen Stil vorliegt. Falls dem so ist, hat man nun vier Konstellationen für einen Vergleich. Nämlich prozess- und ereignisorientiert mit der Referenzversion von DESMO-J sowie prozess- und ereignisorientiert mit der neuen Version von DESMO-J. Alle müssen theoretisch die gleiche Ausgabe liefern. In der praxis liegen jedoch vermutlich drei identische und ein unterschiedlicher Report vor. Es sollten nun nochmals zwei Abdeckungsanalysen durchgeführt werden. Nämlich eine von dem Programmdurchlauf mit dem falschen Report sowie von dem Modell mit dem anderen Modellierungsstil. Die Stellen, die im ersten Bericht gegenüber dem zweiten verschieden sind, sind wieder als potentielle Fehlerquellen anzusehen.

Keine der genannten Strategien wird den Fehler exakt finden können und letztgenannte kann nicht immer angewendet werden. Jede von ihnen hilft aber das Gebiet um den Fehler einzugrenzen. Durch das Anwenden mehrerer Strategien ließe sich z.B. auch eine Schnittmenge der potentiellen Fehler bilden, sodass man priorisieren kann, in welchem Bereich ein Fehler besonders wahrscheinlich erscheint. Der Erfolg der Strategien hängt stark von der Wahl der Modelle ab. Wenn es nur wenige Modelle gibt und diese aber fast das gesamte Framework abdecken, ist eine Fehlerfindung schwierig, da nicht viel eingegrenzt werden kann. Empfehlenswert ist daher die Variante, möglichst viele und kleine Modelle zu wählen, welche in ihrer Summe das gesamte Framework abdecken.

6. Implementation

Im letzten Kapitel wurden die Anforderungen an den zu entwerfenden Regressionstest definiert. Es wurde außerdem ein Konzept erarbeitet, welches diese Anforderungen erfüllen kann. Ziel dieses Kapitels wird es nun sein dieses Konzept softwaretechnisch zu implementieren. Es wird dazu Schritt für Schritt erklärt, mit welchen Maßnahmen das Konzept umgesetzt wurde und an welchen Stellen es Schwierigkeiten gab, sodass vom Konzept abgewichen werden musste. Nachdem der Test dann fertiggestellt wurde, soll noch kurz zusammengefasst werden, wie der Test genau zu benutzen ist.

6.1. Entwicklungsverlauf

Dieser Abschnitt widmet sich der Entwicklung des Regressionstests. Hierzu wird nach und nach das Konzept betrachtet und entsprechend erklärt, wie der jeweilige Punkt umgesetzt wurde oder welche Probleme dabei auftraten.

6.1.1. Grundlegende Überlegungen

Als erstes wird festgelegt, dass für die entstehenden Java-Klassen, zur besseren Übersicht, das Package *test.regression* angelegt wird. Für sämtliche Bibliotheken, wozu DESMO-J und externe Werkzeuge gehören, wird das Verzeichnis *lib* angelegt. Das Build-Skript, zur Ablaufsteuerung mit Ant, wird im Hauptverzeichnis des Projekts angelegt. DESMO-J gibt es, wie in Kapitel 3.2 bereits beschrieben, in verschiedenen Ausführungen. Da man nur den Simulationskern oder das komplette Paket mit sämtlichen Erweiterungen nutzen kann, muss sich auch beim Regressionstest entschieden werden, welche Bibliothek zum Einsatz kommen soll. Generell spricht nichts dagegen, das gesamte Paket zu verwenden. Aus Sicht des Tests wäre dies sogar wünschenswert, um die komplette Software testen zu

können. Da dies allerdings einen unschönen Nebeneffekt mit sich bringt, wird hier erstmal nur der Kern verwendet. Die hier verwendeten Modelle nutzen ohnehin nur diesen. Der erwähnte Nebeneffekt ergibt sich aus dem Werkzeug JaCoCo für die Abdeckungsanalyse. In Kapitel 4.2 wurde bereits erklärt wie die Statistiken aus dem Analysereport zu interpretieren sind. Einige werden durch einen Balken visualisiert, dessen Größe relativ zu dem größten Package steht. Beim kompletten Paket von DESMO-J wird u.A. auch *Java 3D* eingesetzt. Diese Bibliothek hat so viele Klassen, dass die eigentlich relevanten Packages, nämlich die von DESMO-J, dagegen sehr klein aussehen. Daher sind die Statistiken nur schwer zu erkennen. Da DESMO-J für den Regressionstest als Bibliothek eingebunden wird, lassen sich einzelne Klassen nicht so einfach aus dem Report ausschließen. Daher soll zur besseren Übersicht vorerst nur der Kern benutzt werden. Diese Entscheidung lässt sich allerdings auch jederzeit ändern. Da in dieser Arbeit nun nur der Kern verwendet wird, muss noch eine kleine Anpassung gemacht werden. Denn dadurch, dass die Reports der Modelle später mit XMLUnit verglichen werden sollen, müssen diese auch als XML exportiert werden. Der Export von Reports in XML ist aber Bestandteil einer Erweiterung von DESMO-J. Um diesen Missstand zu beheben, wurde die benötigte Funktionalität aus der entsprechenden Erweiterung extrahiert und in eine Jar-Datei verpackt. Diese kleine Bibliothek muss also auch im Classpath mit aufgenommen werden.

6.1.2. Ablaufsteuerung

Da alle nötigen Vorüberlegungen getroffen wurden, kann mit der Implementierung der automatisierten Ablaufsteuerung begonnen werden. Diese wird von Ant übernommen. Um diesen Abschnitt besser zu verstehen, wurde die Build-Datei *build.xml* im Anhang schrittweise eingefügt. Als erstes werden dort ein paar Propertys festgelegt. Sämtliche Verzeichnisse sowie die Versionen von DESMO-J werden in Propertys gespeichert. Auf diese Weise ist der Test auch für die Zukunft leichter anzupassen. Weiterhin wurde auch die JaCoCo Bibliothek bekannt gemacht. Für die Punkte 1 und 4 aus dem Konzept (Kompilieren mit der Referenzversion/neuen Version von DESMO-J) werden außerdem verschiedene Classpaths definiert. Im Folgenden müssen viele Aufgaben zweimal ausgeführt werden, da die verschiedenen Versionen verglichen werden sollen. Aus diesem Grund wird den Targets, zur besseren Lesbarkeit, das Suffix *-ref* bzw. *-new* angehängt. Die ersten Targets, welche im Anhang unter Listing A.2 zu sehen sind, zeigen keine Besonderheiten auf. Die Software, welche es noch zu schreiben gilt, wird kompiliert,

verpackt und gestartet. Die dabei entstehenden Reports der Modelle werden anschließend in ein spezielles Verzeichnis verschoben. Die richtige Reihenfolge der Schritte ergibt sich aus den Abhängigkeiten, die mittels *depends* ausgedrückt werden. Durch diese vier Targets wird die Ablaufsteuerung der ersten drei Punkte des Konzepts realisiert. Die weiteren drei sehen fast identisch aus und sind daher nicht im Anhang zu finden. Bei der Bezeichnung der Targets wurde *-ref* durch *-new* ersetzt und der zweite Classpath verwendet. Außerdem wurden die Quellen mit der Option *debug=„true“* kompiliert sowie die Codeabdeckungsanalyse eingeschaltet. Dazu wurde im Target *run-new* das *java*-Task mit dem Container ⟨*jacoco:coverage*⟩ . . . ⟨*/jacoco:coverage*⟩ umgeben. Ergebnis dieser acht Tasks ist nun ein Ordner namens *desmo-reports* mit den Unterordnern *ref* und *new*. Inhalt dieser Ordner sind die jeweiligen Reports der Modelle. Durch den Einsatz von JaCoCo im zweiten Durchlauf existiert nun im Hauptverzeichnis auch die Datei *jacoco.exec*. Aus dieser Datei lässt sich ein Report zur Abdeckungsanalyse erzeugen.

Wie die Erzeugung des Analysereports aussieht wird in Listing A.3 gezeigt. Anschließend wird noch die Klasse *ReportComparison* gestartet. Sie soll die entstandenen Reports der Modelle vergleichen. Damit wäre die Ablaufsteuerung für den Test soweit vorbereitet. Ergebnis soll nun eine Textdatei im Hauptverzeichnis sein, welche ein Protokoll über den Test enthält. Enthalten soll diese Datei eine Auskunft darüber, welche Modelle getestet wurden und welches Ergebnis dieser Test geliefert hat. Der Report zur Abdeckungsanalyse, welche die Aussagekraft des Tests angibt, ist im Ordner *report* zu finden.

6.1.3. Implementation der Funktionalität

Da nun das Gerüst für die Ablaufsteuerung vorhanden ist, müssen die entsprechenden Java-Klassen programmiert werden. Da gewisse Bezeichnungen, wie z.B. Ordner und Dateinamen häufiger verlangt werden, wird eine Klasse namens *TestConfig* angelegt. Hier sollen alle relevanten Konfigurationen gespeichert werden, damit der Test leicht anzupassen ist. Die erste Klasse, die beim Testablauf aufgerufen wird ist *ModelStarter*. Sie soll alle Modelle oder nur eine Teilmenge starten können und diese aus einer Liste entnehmen. Deshalb wird zuerst die Klasse *ModelList* erklärt. Sie besitzt die innere Klasse *Model*. Die Objekte von Model sollen Elemente der Liste sein und bringen zwei Eigenschaften mit sich. Die erste ist *starter*, sie hält den Namen der Startklasse des Modells. Die andere *expName* und gibt den Namen dessen Experiments an. Die Liste selbst ist eine schlichte ArrayList. Im Konstruktor werden alle Modelle mit entsprechenden

Angaben eingetragen und im Array eingefügt. Die Klasse besitzt weiterhin zwei Methoden, nämlich *getList*, um an die gesamte Liste zu gelangen sowie *getModelByName*, um ein Modell über dessen Experimentnamen zu finden. Die Klasse *ModelStarter* nutzt nun *getList*, um alle Modelle zu starten. In der Methode *runAll* wird die Liste der Modelle iteriert und der jeweilige Name der Startklasse an die Methode *runClass* übergeben. Listing 6.1 zeigt wie dort die Main-Methode der Klasse ausgeführt wird. Die Methode *runSubset* funktioniert genau so, nur dass dort eine Liste mit Startklassen bestimmter Modelle übergeben wird. Diese Liste von Modellen wird dann ebenfalls mit *runClass* ausgeführt.

```
private void runClass(String className, String[] args) {
    try {
            Class<?> c = Class.forName(className);
            Class<?>[] argTypes = { args.getClass(), };
            Object passedArgv[] = { args };

            Method m = c.getMethod("main", argTypes);
            m.invoke(null, passedArgv);

    } catch (Exception e) {
            e.printStackTrace();
    }
}
```

Listing 6.1: Mit Reflection wird eine Klasse ausgeführt

Da Ant die Klasse und somit dessen Main-Methode startet, kann aus der Build-Datei nicht direkt eine Methode aufgerufen werden. In der Main-Methode der Klasse ModelStarter wird daher eine Unterscheidung eingebaut. Werden der Klasse keine Argumente übergeben, werden alle Modelle gestartet. Wenn ein Argument übergeben wird, kommt runSubset zum Einsatz. Das Argument soll der Pfad zu einer Textdatei sein. Diese Textdatei enthält dann eine Liste mit den zu startenden Modellen.

Mit den bisherigen Klassen und Methoden können, die Modelle bereits gestartet werden. Als nächstes müssen also die Reports von DESMO-J miteinander verglichen werden. Für den Vergleich ist die Klasse *ReportComparison* verantwortlich. Genau genommen ist die Klasse aber nur eine Art Wrapper, um den Vergleich besser zu koordinieren. Der eigentliche Vergleich der XML-Dokumente findet in der Klasse *XMLDiff* statt. ReportComparison erzeugt eine Instanz von XMLDiff und übergibt ihr dabei den Verzeichnisnamen der Reports die es zu vergleichen gilt. Über die Methode *compareAll* bekommt die Klasse ReportComparison dann die Information von XMLDiff, ob alle Reports identisch sind

oder ob irgendwo Unterschiede festgestellt wurden. Falls Unterschiede festgestellt wurden, wird die Methode *getDifModelList* aufgerufen. Sie liefert eine ArrayList mit den Namen der Startklassen betroffenener Modelle. Dies ist wichtig für die spätere Fehlerlokalisation und den Abschlussreport. In Listing 6.2 ist die genaue Funktionsweise der Methoden compareAll sowie addDifModel zu sehen. Die Methode addDifModel fügt das Modell bei gefundenen Unterschieden in die Liste der fehlerhaften Modelle ein, welche über getDifModelList zu bekommen ist. Die Methode compareAll speichert die Report-

```
public boolean compareAll() {
        boolean retVal = true;

        File[] refFiles = _refDir.listFiles();
        File[] newFiles = _newDir.listFiles();

        for(int i=0; i< refFiles.length;i++) {
                if(!filesAreEqual(
                                refFiles[i].getPath(),
                                newFiles[i].getPath())
                ) {
                        retVal = false;
                        addDifModel(refFiles[i]);
                }
        }
        return retVal;
}

private void addDifModel(File report) {
        String fileName = report.getName();
        String modelName = "";
        if(fileName.contains("_report.xml")) {
                modelName = fileName.replace("_report.xml", "");
        }
        else {
                modelName = fileName.replace("_trace.xml", "");
        }
        ModelList.Model m = _modelList.getModelByName(modelName);
        String modelStarter = m.get_starter();
        _failedModelList.add(modelStarter);
}
```

Listing 6.2: Die Methoden compareAll und addDifModel der Klasse XMLDiff

Dateien in Arrays, um sie anschließend paarweise mit der Methode *filesAreEqual* zu vergleichen. Der Vergleich der einzelnen Dateien geschieht so wie er bereits in Kapitel 4.3 mit den Listings 4.7 und 4.8 erklärt wurde. Wenn bei diesem Vergleich ein Unterschied festgestellt wird, so wird die Startklasse des Modells direkt über die Methode *addDifModel* in die Liste der Modelle mit unterschiedlichen Reports aufgenommen.

6.1.4. Fehlerlokalisation

Da der Test nicht immer positiv ausfallen wird, muss nach einem negativen Ergebnis weiterhin eine Fehlerlokalisation stattfinden. In Kapitel 5.3 wurden im softwaretechnischen Konzept mehrere Möglichkeiten aufgezeigt, um eine Eingrenzung der Fehlerquelle zu ermöglichen. Wünschenswert wäre daher eine Klasse, welche alle dort vorgestellten Strategien anwenden kann. Ein zentraler Punkt in jeder Strategie ist die Abdeckungsanalyse. Eine solche Analyse wurde, mit Hilfe von Ant, bereits für den Testablauf mit allen Modellen erstellt. Jetzt geht es allerdings darum Analysen zu erstellen, welche nur die Modelle mit Unterschieden beinhalten. Diese Modelle wurden in der Klasse ReportComparison bereits benannt. Ein weiterer Unterschied ist, dass nun nicht einfach nur ein Bericht zur Abdeckung erzeugt werden soll. Im Schritt der Fehlerlokalisation soll die Abdeckungsanalyse, mit Hilfe der API von JaCoCo, weiter untersucht werden, um den Bereich der Fehlerquelle einzugrenzen. Aus diesem Grund ist es problematisch eine Klasse zu entwickeln, welche alle Strategien ausführt. Dies ergibt sich daraus, dass die Abdeckungsanalyse in der Build-Datei durch eine spezielle Umgebung eingeleitet wird. Das *JaCoCo execution data file* (*.exec Datei), welches es zu untersuchen gilt, wird erst erzeugt, nachdem die Umgebung ausgeführt wurde. Die Interpretation der Daten kann somit erst im darauf folgenden Task stattfinden. Das Problem ist also, dass es in den Build-Tasks immer einen Wechsel zwischen Ablauf der Modelle und Interpretation der Abdeckungsanalyse geben muss. Da zwischen diesen beiden Schritten keine Kommunikation stattfinden kann, ist ein koordinierter Ablauf der Fehlerlokalisation schwierig. Hierzu müsste es möglich sein zur Laufzeit einer Java-Klasse selbst zu bestimmen, welche Aufrufe auf Quelltextabdeckung zu untersuchen sind. Weiterhin müsste es möglich sein zu diesen Untersuchungen die Exec-Datei zu erzeugen, damit sie anschließend direkt analysiert werden kann. JaCoCo hält für diesen Zweck auch Lösungen bereit. Diese sind allerdings nicht so trivial anzuwenden, wie eine Analyse mit Hilfe von Ant, da Klassen für diesen Zweck mit speziellen Classloadern instrumentalisiert werden müssen[1]. In der Bearbeitungszeit dieser Arbeit, wurde es leider nicht geschafft, diese Variante zufriedenstellend zu entwickeln. Daher wird im Folgenden eine Alternative vorgestellt, sodass zumindest die erste Strategie zur Fehlerlokalisation angewendet werden kann.

In der ersten Strategie ging es darum, dass die Modelle, in denen unterschiedliche Reports festgestellt wurden, nun ein zweites mal durchlaufen werden sollen. Dadurch lässt

[1] Für ein Beispiel dieser Instrumentalisierung siehe:
http://www.eclemma.org/jacoco/trunk/doc/examples/java/CoreTutorial.java

sich dann feststellen, welche Bereiche im Quelltext genutzt wurden und die Fehlerquelle darstellen könnten. Die Modelle mit Unterschieden wurden bereits von ReportComparison benannt und sollen von ModelStarter nochmals gestartet werden. Aus diesem Grund werden die betroffenen Modelle von der Klasse ReportComparison nun in eine Textdatei gespeichert. Wie in Listing A.3 zu sehen ist, wird nach dem Aufruf von ReportComparison, mit dem *available*-Task, eine Property gesetzt. Diese Property gibt an, ob die Textdatei vorhanden ist. Gleichzeitig lässt sich somit in der Build-Datei feststellen, ob Unterschiede in Modellen festgestellt wurden. Das Target *localize-failure* (Listing A.4) bezieht sich auf diese Property durch die Option *if="list.present"* und wird nur bei negativem Testausgang ausgeführt. In dem Target wird wieder die Klasse ModelStarter mit JaCoCo ausgeführt. Als Argument wird diesmal die Textdatei mit den betroffenen Modellen übergeben. So steht nach Beendigung des Targets die benötigte Exec-Datei zur Verfügung. Ausgewertet wird diese Datei dann bei der Erstellung des Abschlussreports, sodass dort die nötigen Hinweise auf Fehlerquellen gegeben werden können.

6.1.5. Abschlussreport

Der Test ist an dieser Stelle abgeschlossen. Die Reports der Modelle wurden verglichen und für eventuelle Fehler wurde eine Lokalisation durchgeführt. Die Ergebnisse des Tests sollen nun in einem abschließenden Report präsentiert werden. Da der Test sowohl positiv als auch negativ ausfallen kann muss der Report auch entsprechend gestaltet werden. Ob der Test positiv oder negativ ausgefallen ist, lässt sich inzwischen anhand einer Textdatei prüfen. Wenn die Datei mit den fehlerhaften Modellen vorhanden ist, war der Test negativ, ansonsten positiv.

Bei positivem Testausgang soll dem Benutzer aufgelistet werden, welche Modelle für den Test verwendet wurden und wo der Report zur Quelltextabdeckung zu finden ist. Der Benutzer kann dadurch die Aussagekraft des Tests besser beurteilen. Falls der Test negativ ausgegangen ist, soll dem Benutzer weiterhin aufgezeigt werden, wo die Fehlerquelle zu vermuten ist. Wie die Reports bspw. aussehen könnten ist in den Abbildungen B.1-3 zu sehen. Verantwortlich für den Abschlussreport ist die Klasse *ReportGenerator*, sie wird im letzten Target *generate-report* aufgerufen. Es wird zuerst überprüft, ob die entsprechende Textdatei vorhanden ist. Je nach Ausgang der Prüfung wird ein entsprechender Konstruktor aufgerufen, welcher die nötigen Parameter für den Abschlussreport setzt. Mit der Methode *startReport* wird dann der Report generiert.

Wenn die Textdatei vorhanden ist bedeutet dies, dass der Test negativ war. Im Konstruktor wird daher für das Testergebnis *false* festgelegt und die Textdatei wird eingelesen. In der Methode startReport wird zunächst ein entsprechender Standardtext in den Report geschrieben. Da die verwendeten Modelle alle von der Klasse ModelList bereitgestellt werden, kann die Auflistung auch hier von dieser bezogen werden. Die Aufzählung der fehlerhaften Modelle stammt von der eingelesenen Textdatei und zur Auflistung der möglichen Fehlerquellen wird, zur besseren Übersicht, eine zweite Datei erstellt und auf diese verwiesen. In der Methode *analyzeFailedModels* wird dann die Datei mit möglichen Fehlerquellen gefüllt. Mit der Methode *loadExecutionData* wird zunächst die Exec-Datei eingelesen. Durch die Methode *analyzeStructure* wird sich anschließend ein Objekt vom Typ *IBundleCoverage* erzeugt[2]. Mit diesem Objekt gelangt man an sämtliche Statistiken zur Quelltextabdeckung. Mit der erweiterten For-Schleife wird so über alle Packages, Klassen, Methoden und Anweisungen iteriert. Die Anweisungen werden schließlich mit der Methode *getCoveredRatio* geprüft. Wenn der Wert größer ist als Null, bedeutet dies, dass die Anweisung mindestens einmal ausgeführt wurde. Auf diese Weise lässt sich feststellen, welche Klassen und welche Methoden überhaupt von den betroffenen Modellen benutzt wurden.

Bei einem positiven Testergebnis wird lediglich ein Standardtext ausgegeben und anschließend werden die verwendeten Modelle von ModelList aufgezählt. Abschließend wird auf den Ordner *report* hingewiesen. Hier lässt sich die Abdeckungsanalyse einsehen.

6.2. Wie der Test zu benutzen ist

Der Test und die anschließende Fehlerlokalisation wurde nun implementiert. Dieser Abschnitt soll daher noch einmal kurz zusammenfassen, wie der Test zu benutzen ist und worauf geachtet werden muss.

Zuerst sollten die zu untersuchenden Versionen von DESMO-J in das Verzeichnis *lib* kopiert werden. Die genauen Dateinamen der Versionen sind in der Build-Datei unter den Propertys *desmo.ref* und *desmo.new* anzugeben. Für eine anschließende Abdeckungsanalyse sollte die Version von desmo.new mit der Option *debug="true"* kompiliert werden. Die Property für das Verzeichnis von *desmo.source* muss im Weiteren auf den aktuellen

[2] siehe auch Listing 4.5 und 4.6

Quelltext von DESMO-J (also desmo.new) gesetzt werden. Für eine korrekte Abdeckungsanalyse in der Fehlerlokalisation muss außerdem der String *DESMOLIBFILE* in der Klasse *TestConfig* auf den gleichen Wert wie desmo.new gesetzt werden.

Falls für den Test weitere Modelle eingesetzt werden sollen, sind folgende Schritte zu tun. Das Modell muss in das Package *test.regression.models* kopiert werden. Die Erzeugung des Experiments in dem Modell muss ähnlich wie in Listing 6.3 erfolgen.

```
Experiment exp = new Experiment("ArztpraxisExperimentEreig", ".",
                 TimeUnit.MILLISECONDS, TimeUnit.MINUTES, null,
                 "desmoj.extensions.xml.report.XMLReportOutput",
                 "desmoj.extensions.xml.report.XMLTraceOutput",
                 "desmoj.core.report.HTMLErrorOutput",
                 "desmoj.core.report.HTMLDebugOutput");
```

Listing 6.3: Beispiel für die Erzeugung eines Experiments

Wichtig ist hierbei vor allem der erste Parameter für den Experimentnamen. Weiterhin wichtig ist, dass die Ausgaben für Report und Trace auf XML gesetzt werden. Übrige Parameter können dem Modell entsprechend belegt werden. Empfehlenswert ist außerdem mit dem Aufruf *exp.setShowProgressBar(false);* das Einblenden eines Fortschrittsbalkens zu unterbinden. Damit das Modell schließlich im Test benutzt wird, muss es in der Klasse *ModelList* bekannt gemacht werden. Hierzu wird ein Objekt der inneren Klasse *ModelList.Model* mit folgendem Aufruf erzeugt:
Model modelName = new Model(starterClass, expName);
Verlangt werden hierbei zwei Strings. Bei *starterClass* handelt es sich um die Startklasse des Modells und *expName* steht für den Namen, der bei der Erzeugung des Experiments gewählt wurde. Dieses Objekt wird nun in die Arrayliste von ModelList aufgenommen und dadurch beim nächsten Testdurchlauf ebenfalls überprüft.

Um den Test zu starten, muss die Build-Datei *build.xml* ausgeführt werden. Alle implementierten Schritte des Tests laufen nun automatisch ab. Anschließend lässt sich im Hauptordner das Ergebnis aus der Datei *AbschlussReport.txt* ablesen.

7. Fazit/Diskussion

Da die Implementation des Regressionstests nun abgeschlossen ist, soll diskutiert werden, ob sie den Anforderungen gerecht geworden ist. Dazu werden zuerst die positiven Aspekte, dann die negativen herausgestellt. Anschließend wird es ein Fazit darüber geben, wie das Ergebnis dieser Arbeit zu bewerten ist.

Ziel der Arbeit war es, den Entwicklungsprozess durch einen Regressionstest zu unterstützen. Der Regressionstest sollte mit Ant und Eclipse gesteuert werden. Weiterhin sollten durch den Test gefundene Fehler, so gut wie möglich eingegrenzt werden. Dies sollte mit Hilfe von Analysen der Quelltextabdeckung geschehen.

Dadurch, dass die Grundlagen von Softwaretests, Simulation und DESMO-J vorgestellt wurden, konnte bestätigt werden, dass ein Regressionstest die richtige Wahl für DESMO-J ist. Die im vierten Kapitel ausgesuchten Werkzeuge halfen bei der Erstellung eines softwaretechnischen Konzepts und konnten auch in der Implementierung erfolgreich eingesetzt werden. Dadurch war es möglich, den Regressionstest ohne Probleme umzusetzen. Positiv anzumerken ist, dass weitere Modelle ohne großen Aufwand eingefügt werden können und dass der Vergleich aller Modellreports komplett automatisch erfolgt. Der Test arbeitet sehr schnell und findet zuverlässig Unterschiede in Reports. Zu Testzwecken wurde absichtlich ein Fehler in DESMO-J eingebaut. Dieser wurde vom Test im richtigen Modell festgestellt.[1]. Darüber hinaus sind während der Entwicklung auch echte Unterschiede festgestellt worden. Ein solcher Unterschied ist in Abbildung 7.1 zu sehen.

Die Eingrenzung der Fehlerquellen konnte nicht so zufriedenstellend implementiert werden, wie eingangs erhofft. Dies lag vor allem daran, dass die dafür benötigten Mittel falsch eingeschätzt wurden. Als das Problem erkannt wurde, war die Zeit schon zu weit fortgeschritten, sodass eine minimalistische Lösung gefunden werden musste. Außerdem

[1] Siehe hierzu auch Abbildung B.2 und B.3

wurde sich zu sehr darauf konzentriert, die Eingrenzung der Fehlerquellen im Anschluss an den Regressionstest ebenfalls komplett automatisiert ablaufen zu lassen. Es wäre wahrscheinlich funktionell mehr möglich gewesen, wenn auf etwas Automatisierung verzichtet worden wäre. Der Entwickler hätte dann im Fehlerfall Werkzeuge zur Verfügung, um die Fehlerfindung zwar manuell, dafür aber präziser durchführen zu können.

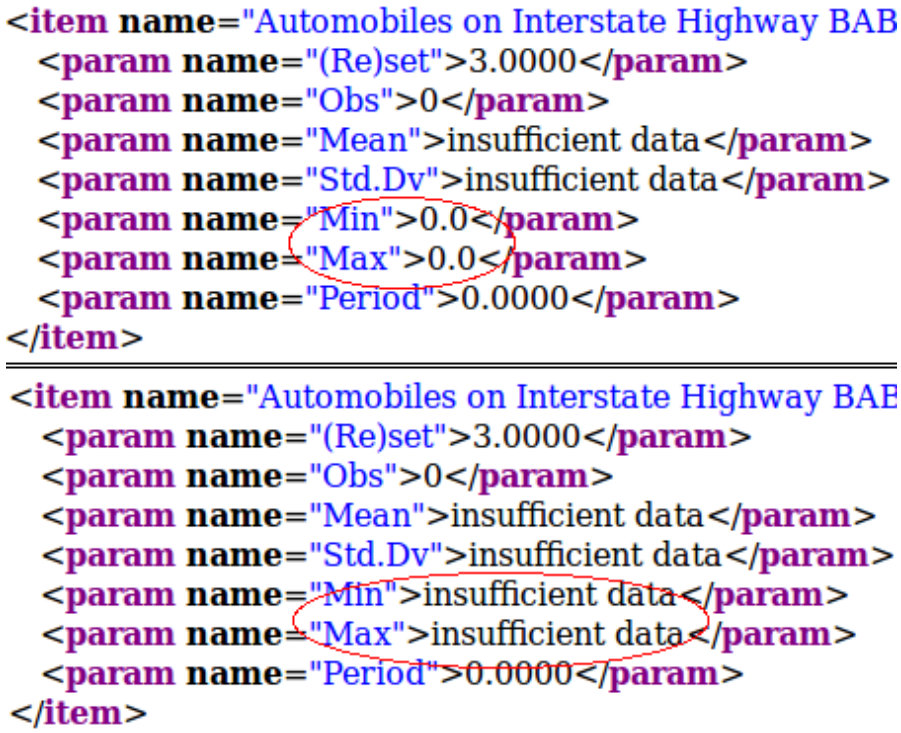


Abbildung 7.1.: Oben Version 2.3.2, darunter 2.3.4beta

Weiterhin nicht ganz zufriedenstellend ist die Abdeckung, die der Test mit sich bringt. Aktuell umfasst der Test 16 Modelle, von denen aber viele die gleiche Funktionalität nutzen. Um die Abdeckung dennoch ein wenig zu erhöhen, wurden einige Modelle mehrfach eingesetzt und dabei verschiedene Statistiken und Verteilungen verwendet.

Als Fazit lässt sich sagen, dass das Ziel, wenn auch mit kleinen Abstrichen, erreicht wurde. Mit dem Regressionstest können Entwickler zukünftig ihre Arbeit besser kontrollieren und werden sofort auf ein unterschiedliches Verhalten aufmerksam gemacht. Die Fehlerfindung bietet zwar noch Verbesserungspotential, schränkt die möglichen Quellen aber schon ein. Die noch niedrige Testabdeckung ist kein Hindernis den Test zu benutzen. Sie kann durch neue Modelle erhöht werden. Dem Entwickler sollte allerdings immer bewusst sein, welche Aussagekraft hinter dem Test steckt.

8. Ausblick

Abschließend soll nun ein Ausblick gegeben werden. Dieser besteht aus Verbesserungsvorschlägen zur Implementation sowie aus Vorschlägen zu weiteren Schritten.

Da die Testabdeckung noch nicht sehr hoch ist, wäre es wünschenswert, weitere Modelle in den Test einzufügen oder spezielle Modelle für den Test zu entwickeln. Dies ist sehr wichtig, um dem Test die nötige Aussagekraft zu verleihen.

In dieser Arbeit konnte im Bereich Fehlerlokalisation nur eine von drei vorgestellten Strategien umgesetzt werden. Zur weiteren Eingrenzung der Fehlerquellen, müssten die zwei weiteren Strategien implementiert werden. Denkbar wäre auch für Entwickler ein Werkzeug bereitzustellen, mit dem die Eingrenzung manuell vorgenommen wird.

Um den Release-Zyklus von DESMO-J zu verbessern, wäre es denkbar die Build-Datei von DESMO-J mit der Build-Datei des Regressionstests zu kombinieren. So wird bei Herausgabe einer neuen Version der Regressionstest gestartet, um mögliche Fehler zu finden.

Bei der Entwicklung des Regressionstests ist aufgefallen, dass die Klassen zur XML-Erzeugung in einer Erweiterung von DESMO-J zu finden sind. Diese Klassen sind sehr klein und werden wahrscheinlich häufiger benötigt. Es sollte daher überlegt werden, sie in den Kern aufzunehmen.

A. build.xml

```
<?xml version="1.0" ?>

<project name="Regressionstest" basedir="." default="main" xmlns:jacoco="antlib:org.jacoco.ant">

    <property name="src.dir"      value="src"/>
    <property name="lib.dir"      value="lib"/>
    <property name="build.dir"    value="build"/>
    <property name="classes.dir" value="${build.dir}/classes"/>
    <property name="jar.dir"      value="${build.dir}/jar"/>
    <property name="main-class"   value="test.regression.ModelStarter"/>

    <property name="desmo.ref"    value="desmoj-2.3.2-core-bin.jar"/>
    <property name="desmo.new"    value="desmoj-2.3.3beta-core-bin.jar"/>

    <property name="executiondata.first"     value="jacoco.exec"/>
    <property name="executiondata.second"    value="test.exec"/>
    <property name="report.dir"              value="report"/>
    <property name="models.tostart"          value="failedModels.txt"/>

    <taskdef uri="antlib:org.jacoco.ant" resource="org/jacoco/ant/antlib.xml">
        <classpath path="${lib.dir}/jacocoant.jar"/>
    </taskdef>


    <path id="classpath-ref">
        <fileset dir="${lib.dir}" includes="${desmo.ref},
␣␣␣␣␣␣␣␣␣␣␣␣desmoj_xml_extension.jar,␣xmlunit-1.3.jar,␣jacocoant.jar" />
    </path>

    <path id="classpath-new">
        <fileset dir="${lib.dir}" includes="${desmo.new},
␣␣␣␣␣␣␣␣␣␣␣␣desmoj_xml_extension.jar,␣xmlunit-1.3.jar,␣jacocoant.jar" />
    </path>
```

Listing A.1: Teil 1, Definition von Propertys und Classpath

```
<target name="compile-ref">
    <mkdir dir="${classes.dir}"/>
    <javac srcdir="${src.dir}" destdir="${classes.dir}" classpathref="classpath-ref"/>
</target>

<target name="jar-ref" depends="compile-ref">
    <mkdir dir="${jar.dir}"/>
    <jar destfile="${jar.dir}/${ant.project.name}.jar" basedir="${classes.dir}">
        <manifest>
            <attribute name="Main-Class" value="${main-class}"/>
        </manifest>
    </jar>
</target>

<target name="run-ref" depends="jar-ref">
    <java classname="${main-class}" fork="true">
        <classpath>
            <pathelement location="${classes.dir}"/>
            <path refid="classpath-ref"/>
            <path location="${jar.dir}/${ant.project.name}.jar"/>
        </classpath>
    </java>
</target>

<target name="move-reports-ref" depends="run-ref">
    <mkdir dir="desmo-reports"/>
    <mkdir dir="desmo-reports/ref"/>
    <mkdir dir="desmo-reports/new"/>
    <move todir="desmo-reports/ref">
        <fileset dir=".">
            <exclude name="build.xml"/>
            <include name="*.xml"/>
        </fileset>
    </move>
</target>
```

Listing A.2: Teil 2, Erster Lauf der Modelle

```
<target name="jacoco-report" depends="move-reports-new">
    <jacoco:report>

        <executiondata>
            <file file="${executiondata.first}"/>
        </executiondata>

        <structure name="Regressionstest">
            <classfiles>
                <fileset dir="${lib.dir}"        includes="${desmo.new}"/>
            </classfiles>
            <sourcefiles>
                <fileset dir="desmo-source"      includes="**/*.java" />
            </sourcefiles>
        </structure>

        <html     destdir="${report.dir}"/>

    </jacoco:report>
</target>

<target name="run-repdif" depends="jacoco-report">
    <java classname="test.regression.ReportVergleich" fork="true">
        <classpath>
            <pathelement location="${classes.dir}"/>
                <path refid="classpath-ref"/>
            <path location="${jar.dir}/${ant.project.name}.jar"/>
        </classpath>
    </java>

    <available file="${models.tostart}" property="list.present" />

</target>
```

Listing A.3: Teil 3, Report der Abdeckungsanalyse und Vergleich der DESMO-J Reports

```
<target name="localize-failure" depends="run-repdif" if="list.present">
        <jacoco:coverage destfile="${executiondata.second}">
                <java classname="${main-class}" fork="true">
                        <arg value="${models.tostart}"/>
                        <classpath>
                                <pathelement location="${classes.dir}" />
                                <path refid="classpath-new" />
                                <path location="${jar.dir}/${ant.project.name}.jar" />
                        </classpath>
                </java>
        </jacoco:coverage>
</target>
```

Listing A.4: Teil 4, Fehlerlokalisation

```
<target name="generate-report" depends="localize-failure">

        <java classname="test.regression.ReportGenerator" fork="true">
                <classpath>
                        <pathelement location="${classes.dir}" />
                        <path refid="classpath-new" />
                        <path location="${jar.dir}/${ant.project.name}.jar" />
                </classpath>
        </java>
        <delete>
                <fileset dir=".">
                        <exclude name="build.xml" />
                        <include name="*.xml" />
                        <include name="*.html" />
                        <include name="${executiondata.first}" />
                        <include name="${executiondata.second}" />
                        <include name="${models.tostart}" />
                </fileset>
        </delete>
</target>
```

Listing A.5: Teil 5, Abschlussreport

B. Abschlussreports

```
1 Der Regressionstest wurde mit folgenden Modellen abgeschlossen:
2
3 ArztpraxisExperimentEreig
4 ArztpraxisExperimentPro
5 mining
6 EventExampleExperiment
7 ProcessExampleExperiment
8 BalticSeaEvent
9 BalticSeaProcess
10 BinExampleExperiment
11 CondQueueExampleExperiment
12 CountExampleExperiment
13 InterruptsExampleExperiment
14 ResExampleExperiment
15 StatisticsExampleExperiment
16 StockExampleExperiment
17 WaitQueueExampleExperiment
18 mits
19
20 Das Testergebnis ist positiv. Fehler konnten somit nicht nachgewiesen werden. Die
   Abdeckung des Tests kann im Ordner 'report' nachgesehen werden.
```

Abbildung B.1.: Abschlussreport eines positiven Tests

```
Der Regressionstest wurde mit folgenden Modellen abgeschlossen:

ArztpraxisExperimentEreig
ArztpraxisExperimentPro
mining
EventExampleExperiment
ProcessExampleExperiment
BalticSeaEvent
BalticSeaProcess
BinExampleExperiment
CondQueueExampleExperiment
CountExampleExperiment
InterruptsExampleExperiment
ResExampleExperiment
StatisticsExampleExperiment
StockExampleExperiment
WaitQueueExampleExperiment
mits

Das Testergebnis ist negativ, da in folgenden Modellen Unterschiede festgestellt
wurden:

test.regression.models.desmoj_tutorial2_ResExample.ResExample


In welchen Klassen/Methoden der Fehler zu vermuten ist, kann in der Datei
'potentielleFehler.txt' eingesehen werden.
```

Abbildung B.2.: Abschlussreport eines negativen Tests

```
    updateTakenBackRes
    <clinit>
desmoj/core/advancedModellingFeatures/Res$UsedResources
    <init>
    getProcess
    getOccupiedResources
desmoj/core/exception/DESMOJException
    <init>
desmoj/core/exception/SimFinishedException
    <init>
desmoj/core/simulator/EventNote
    <init>
    equals
    compareTo
    getEntity1
    getEntity2
    getEntity3
    getEvent
    getNumberOfEntities
    getTime
    setTime
    toString
desmoj/core/simulator/ExternalEventStop
    <init>
    eventRoutine
    schedule
desmoj/core/simulator/TimeInstant
```

Abbildung B.3.: Auflistung der potentiellen Fehlerquellen (Klassen mit eingerückten Methoden).

Literaturverzeichnis

[BB09] Bodewig, Stefan und Tim Bacon: *XMLUnit - Documentation*. `http://xmlunit.sourceforge.net/userguide/html/index.html`, 2009. [Online; abgerufen am 27.12.2011].

[Bec03] Beck, Kent: *Test-Driven Development: By Example*. The Addison-Wesley Signature Series. Addison-Wesley, 2003.

[Boe79] Boehm, B. W.: *Guidelines for Verifying and Validating Software Requirements and Design Specifications*. In: Samet, P. A. (Herausgeber): *Euro IFIP 79*, Seiten 711–719. North Holland, 1979.

[BW11] Blümm, Clara und Sascha Winde: *Eine Testumgebung für DESMO–J*. Projekt: Simulationsprogrammierung. Universität Hamburg, Fachbereich Informatik, WS 2010/2011.

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

[McC76] McCabe, T. J.: *A Complexity Measure*. IEEE Trans. Softw. Eng., 2:308–320, July 1976.

[MGCK09] Mountainminds GmbH & Co. KG, Munich: *JaCoCo - Documentation*. `http://www.eclemma.org/jacoco/trunk/doc/`, 2009. [Online; abgerufen am 08.11.2011].

[Pag10] Page, Prof. Dr.-Ing. Bernd: *Folien zur Vorlesung Informatikgestützte Gestaltung und Modellierung von Organisationen (IGMO)-Modellierung und Simulation (MUS), Kapitel 2 und 4*. Universität Hamburg, Fachbereich Informatik, SS 2010.

[PK05] Page, B. und W. Kreutzer: *The Java Simulation Handbook: Simulating Discrete Event Systems with UML and Java*. Berichte aus der Informatik. Shaker Verlag, 2005.

[SL10] Spillner, A. und T. Linz: *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester- Foundation Level nach ISTQB-Standard*. Dpunkt.Verlag GmbH, 2010.

Erklärung

Ich versichere hiermit, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen - benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

_____

René Wecker
Hamburg, 13. Januar 2012