Fehlersuche im Modell - Modellbasiertes Testen und Debuggen

Dissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
(Dr. rer. nat.)

im Fachgebiet Software Engineering
Prof. Dr. Albert Zündorf
Fachbereich Elektrotechnik / Informatik
der Universität Kassel

vorgelegt von
Dipl. Inform. Leif Geiger
Disputation am 11. April 2011

Zusammenfassung

Kern der vorliegenden Arbeit ist die Erforschung von Methoden, Techniken und Werkzeugen zur Fehlersuche in modellbasierten Softwareentwicklungsprozessen.

Hierzu wird zuerst ein von mir mitentwickelter, neuartiger und modellbasierter Softwareentwicklungsprozess, der sogenannte Fujaba Process, vorgestellt. Dieser Prozess wird von Usecase Szenarien getrieben, die durch spezielle Kollaborationsdiagramme formalisiert werden. Auch die weiteren Artefakte des Prozess bishin zur fertigen Applikation werden durch UML Diagrammarten modelliert. Es ist keine Programmierung im Quelltext nötig. Werkzeugunterstützung für den vorgestellte Prozess wird von dem Fujaba CASE Tool bereitgestellt. Große Teile der Werkzeugunterstützung für den Fujaba Process, darunter die Toolunterstützung für das Testen und Debuggen, wurden im Rahmen dieser Arbeit entwickelt.

Im ersten Teil der Arbeit wird der Fujaba Process im Detail erklärt und unsere Erfahrungen mit dem Einsatz des Prozesses in Industrieprojekten sowie in der Lehre dargestellt.

Der zweite Teil beschreibt die im Rahmen dieser Arbeit entwickelte Testgenerierung, die zu einem wichtigen Teil des Fujaba Process geworden ist. Hierbei werden aus den formalisierten Usecase Szenarien ausführbare Testfälle generiert. Es wird das zugrunde liegende Konzept, die konkrete technische Umsetzung und die Erfahrungen aus der Praxis mit der entwickelten Testgenerierung dargestellt.

Der letzte Teil beschäftigt sich mit dem Debuggen im Fujaba Process. Es werden verschiedene im Rahmen dieser Arbeit entwickelte Konzepte und Techniken vorgestellt, die die Fehlersuche während der Applikationsentwicklung vereinfachen. Hierbei wurde darauf geachtet, dass das Debuggen, wie alle anderen Schritte im Fujaba Process, ausschließlich auf Modellebene passiert. Unter anderem werden Techniken zur schrittweisen Ausführung von Modellen, ein Objekt Browser und ein Debugger, der die rückwärtige Ausführung von Programmen erlaubt (back-in-time debugging), vorgestellt. Alle beschriebenen Konzepte wurden in dieser Arbeit als Plugins für die Eclipse Version von Fujaba, Fujaba4Eclipse, implementiert und erprobt. Bei der Implementierung der Plugins wurde auf eine enge Integration mit Fujaba zum einen und mit Eclipse auf der anderen Seite geachtet.

Zusammenfassend wird also ein Entwicklungsprozess vorgestellt, die Möglichkeit in diesem mit automatischen Tests Fehler zu identifizieren und diese Fehler dann mittels spezieller Debuggingtechniken im Programm zu lokalisieren und schließlich zu beheben. Dabei läuft der komplette Prozess auf Modellebene ab. Für die Test- und Debuggingtechniken wurden in dieser Arbeit Plugins für Fujaba4Eclipse entwickelt, die den Entwickler bestmöglich bei der zugehörigen Tätigkeit unterstützen.

Abstract

This thesis presents methods, techniques and tools for finding bugs in model-based software development processes.

Therefor this thesis first introduces a new model-based software development process, the so-called Fujaba Process, that was co-developed by me. This process is driven by use cases, which are formalized by customized collaboration diagrams. All other artifacts of the process including the application itself are also modeled using UML diagrams. The Fujaba CASE Tool offers tool support for every phase of this process. Main parts of the tool support for the Fujaba Process, including support for testing and debugging, were developed as part of this thesis.

The first part of this thesis describes the Fujaba Process in detail and shows the experiences we made using this process in industrial projects and for teaching.

The second part describes the test generation, that was developed as part of this thesis and became an important component of the Fujaba Process. During the test generation the formalized use case scenarios are automatically transformed into executable test cases. The second part of the thesis describes the underlying concepts and the technical realization of the test generation. Our experiences from case studies using the test generation are also presented.

The last part of this thesis is about debugging in the Fujaba Process. It presents several concepts and techniques, that were implemented for this thesis and that simplify the fault diagnostics during application development. As all other steps of the Fujaba Process are performed on model level, the presented debugging should be model-based, too. My work will for example present techniques for stepwise execution of models, it will present an object browser for runtime model representation and a back-in-time debugger on model level. All presented concepts are realized and evaluated as plugins for the Eclipse version of Fujaba, Fujaba4Eclipse. The plugin implementations offer a tight integration with eclipse on the one hand and with Fujaba on the other hand.

To conclude, I will present a development process and the possibility to identify faults using tests. Furthermore I will show an approach how to locate these faults using debugging techniques to finally correct these faults. In my approach the complete process is done on model level. I developed plugins for Fujaba4Eclipse for the test generation and the debugging techniques, that support the developer at the best during this process.

Inhaltsverzeichnis

Zusammenfassung i

Abstract ii

Inhaltsverzeichnis v

1. Einleitung 1
- 1.1. Forschungsgegenstand . . . 3
- 1.2. Aufbau des Dokuments . . . 4

2. Anforderungen 5
- 2.1. Kernanforderungen . . . 5
- 2.2. Ergänzende funktionale Anforderungen . . . 5
- 2.3. Nichtfunktionale Anforderungen . . . 6
- 2.4. Zielplattform . . . 6

I. Der Fujaba Process 7

3. Motivation 9

4. Beispiel 13

5. Aufbau 15
- 5.1. Sammeln der Anforderungen mit Usecases . . . 15
- 5.2. Anforderungsanalyse mit Objektspiel . . . 17
- 5.3. Storyboarding - Verfeinern der Usecase Szenarien . . . 20
- 5.4. Ableiten des Klassendiagramms . . . 25
- 5.5. Ableiten von automatischen JUnit Tests . . . 27
- 5.6. Ableiten des Verhaltens der Methoden . . . 27
- 5.7. Codegenerierung mit Fujaba . . . 31
- 5.8. Automatische Validierung der Szenarien . . . 32

6. Werkzeugunterstützung 35

7. Verwandte Arbeiten 41

8. Einsatz 45

9. Fazit 47

II. Testen auf Modellebene 49

10. Motivation 51

11. Konzept 53

12. Umsetzung 55
12.1. Beispiel . . . 55
12.2. Testgenerierung . . . 58
12.2.1. Alternativszenarien . . . 62
12.3. Anpassen des Codegenerators . . . 66
12.4. Eclipse Integration . . . 72
12.5. Debugging Hilfen . . . 72
12.6. Coverage . . . 73
12.6.1. Technische Umsetzung der Modellüberdeckung . . . 75

13. Einsatz 81

14. Verwandte Arbeiten 85

15. Fazit 89

III. Debuggen auf Modellebene 91

16. Motivation 93

17. Konzept 97

18. Umsetzung 101
18.1. Design Level Debugging . . . 101
18.1.1. Quelltext-Modell Abbildung in CodeGen2 . . . 101
18.1.2. JSR-045 und SMAP . . . 106
18.1.3. Eclipse Integration . . . 107
18.2. eDOBS . . . 111
18.2.1. eDOBS - Der Objektstruktureditor . . . 113
18.2.2. Modell-Abstraktion . . . 114
18.2.3. Debugger Integration . . . 117
18.2.4. Abstraktionen . . . 117
18.2.5. eDOBS Integrationen . . . 120
18.2.6. Der Zetteltest und die Fujaba Maschine . . . 120
18.2.7. Testintegration . . . 125
18.2.8. Back-in-time Debugger . . . 128

19. Einsatz 143
19.1. Fujaba . . . 143
19.2. OBA . . . 144

19.3. Lehre 145

20. Verwandte Arbeiten **147**

21. Fazit **153**

IV. Zusammenfassung **155**

22. Zusammenfassung **157**

23. Ausblick **159**

A. Anhang **163**
A.1. Veranstaltungsverzeichnis 163
A.2. Template für die zu-n Suche 166
Literaturverzeichnis 179
Abbildungsverzeichnis 181
Tabellenverzeichnis 182
Listingsverzeichnis 182

1. Einleitung

Modellbasierte Softwareentwicklung ist einer der Hauptforschungsschwerpunkte der Softwaretechnik der letzten Jahre. Auch in der Praxis werden heute in vielen Projekten Modelle und Modellierung eingesetzt. Die Unified Modeling Language (UML) [BRJ98] der Object Management Group (OMG) [Obj10] hat sich hierbei zur meist verbreiteten und meist genutzten Modellierungssprache entwickelt. Weiterhin wird auch das Konzept der Domain Specific Language (DSL), also kleine, speziell auf bestimmte Probleme zugeschnittene Sprachen, vielfach angewendet. Für einen erfolgreichen Einsatz von Modellierung in Softwareentwicklungsprojekten, ist eine gute Werkzeugunterstützung der Modellierungssprache und ein geeigneter Prozess, in den die Modellierungstätigkeiten eingearbeitet sind, zwingend notwendig.

Eine geeignete Werkzeugunterstützung sollte natürlich das Erstellen, Bearbeiten und Austauschen von Modellen ermöglichen. Zusätzlich sollte ein Modellierungswerkzeug aber auch Modelltransformationen und Codegenerierung unterstützen.

Modelltransformationen dienen meist dazu verschiedene Modelle ineinander zu transformieren, wie beispielsweise ein plattformunabhängiges Modell in ein plattformspezifisches Modell [OMG03]. Die OMG hat mit der Query View Transformation Sprache (QVT) [OMG08] eine solche Modelltransformationssprache vorgeschlagen, die jedoch noch wenig Verbreitung gefunden hat.

Um Modelle ausführbar zu machen und die mit ihnen spezifizierte Struktur im weiteren Entwicklungsverlauf nutzbar zu machen, werden meist Codegeneratoren eingesetzt. Das heißt, die Modelle werden in Quelltextfragmente übersetzt, die der weiteren Entwicklung als Grundlage dienen. In den meisten Werkzeugen, wird Codegenerierung jedoch nur für den statischen Anteil des Modells (also in einem UML Modell für die Klassendiagramme) angeboten. Das Verhalten wird dann später im Quelltext von Hand implementiert.

Das Fujaba CASE Tool hingegen ermöglicht es dem Entwickler seine komplette Applikation, also statische Datenstruktur und Verhalten, komplett modellbasiert zu entwickeln. Hierzu werden für den statischen Teil, wie bei vielen anderen Werkzeugen auch, UML Klassendiagramme verwendet. Zur Verhaltensspezifikation verwendet Fujaba sogenannte Storydiagramme [FNTZ98]. Hierbei handelt es sich um eine Kombination aus UML Aktivitätsdiagrammen und Kollaborationsdiagrammen. Die Aktivitätsdiagramme werden verwendet um den Kontrollfluss innerhalb einer Methode zu modellieren. In jede Aktivität eines solchen Aktivitätsdiagramms kann ein spezielles Kollaborationsdiagramm eingebettet werden. Diese Kollaborationsdiagramme beschreiben Änderungen an der Laufzeitdatenstruktur. Semantisch handelt es sich bei den Kollaborationsdiagrammen um Graphersetzungsregeln, die Graphensuche und Graphtransformationen auf dem Laufzeitobjektgraph durchführen (siehe [GGZ+05]). Zur Überführung des Modells zur fertigen Applikation wird in Fujaba

ein Codegenerator verwendet. Dieser erzeugt aus den Klassendiagrammen und den Storydiagrammen lauffähigen Java Quelltext. Das Fujaba CASE Tool wird in dieser Arbeit als Modellierungsplattform verwendet.

Um nun modellbasiert Software entwickeln zu können, bedarf es neben einer Modellierungssprache und dem passenden Werkzeug auch einem geeigneten und speziell auf die modellbasierte Arbeitsweise angepassten Softwareentwicklungsprozess. Klassische modellbasierte Entwicklungsprozesse, wie der Unified Process (siehe [JBR99]), schlagen eine klare Trennung zwischen modellbasiertem Design und quelltextbezogener Implementierung vor. Doch da zunehmend Quelltext aus Modellartefakten generiert wird, ist eine solch strikte Trennung vielfach nicht möglich.

Die OMG stellt mit der Model Driven Architecture (MDA) zwar keinen kompletten Prozess jedoch immerhin einen Softwareentwicklungsansatz vor, der Modelltransformationen und Generatoren an vielen Stellen im Prozess erlaubt und vorsieht. MDA ist explizit darauf ausgelegt, dass die meisten Schritte mithilfe von Modellen und Modelltransformationen gelöst werden. Es ist aber weiterhin möglich Teile der Implementierung im Quelltext durchzuführen.

Ein Problem in vielen modellbasierten Entwicklungsansätzen ist der Umgang mit Fehlern. Es ist oft ungeklärt, wie Fehler im Modell identifiziert werden können und falls ein Fehler entdeckt wird, wie die Fehlerursache eingegrenzt werden kann. Jedoch ist ein Vermeiden und ein frühzeitiges Entdecken und Beheben von Fehlern enorm wichtig. Klassische Schätzungen gehen davon aus, dass ca. 50% der Kosten eines Softwareprojekts durch das Auffinden und Beseitigen von Fehlern entstehen, [Fag02]. Durch den Einsatz modellbasierter Techniken lässt sich die Fehleranzahl in den meisten Fällen reduzieren, ein frühzeitiges Entdecken von Fehlern im Modell kann aber die Kosten eines Projekts noch weiter reduzieren.

Die klassischen Verfahren zur Fehlersuche sind Testen, Reviews und Debugging. Diese lassen sich auch bei modellbasierter Vorgehensweise anwenden, verlangen aber teilweise gute Werkzeugunterstützung um auch produktiv einsetzbar zu sein. Reviewverfahren lassen sich in modellbasierten Ansätzen ähnlich einsetzen wie in klassischen quelltextbezogenen Prozessen. Die extremste Form des Reviews, das Pair Programming des Extreme Programming (XP), [BA04], in der jeweils 2 Entwickler vor einem Rechner sitzen, lässt sich auch im modellbasierten Umfeld durchführen. Auch das klassische Review, in dem ein Entwickler im Nachhinein die Änderungen eines anderen am Quelltext überprüft, lässt sich auf Modelle übertragen. Hier ist allerdings Werkzeugunterstützung in Form eines Diff-Algorithmus, der Modelländerungen visualisieren kann, hilfreich (siehe [TBWK07]).

Auch das Testen sollte modellbasiert erfolgen. Dementsprechend bietet zum Beispiel die UML ein eigenes Profil zum Modellieren von Tests an, [BDG+04]. Dies dient vor allem dazu, die statische Teststruktur zu spezifizieren. So können beispielsweise Klassen durch bestimmte Stereotypen als Testklassen markiert werden. Einfache Tests lassen sich als Sequenzdiagramme modellieren. Zusätzlich existiert ein Mapping mit dem diese Testklassen auf das Java-basierte Testframework JUnit [Men10] abgebildet werden können. Komplexe Testfälle sind mit diesem Vorgehen allerdings nur schwer zu erstellen. Hierzu müsste das Testverhalten meist in Java Quelltext

implementieren werden. Zusätzlich existiert kein Vorgehensmodell, das beschreibt, wie solche Testmodelle zu entwickeln sind. Für Tests, die beispielsweise überprüfen, ob eine bestimmte Anforderung erfüllt ist, wäre ein Prozess wünschenswert, der definiert, wie solche Tests aus dem Anforderungsmodell abzuleiten sind, oder sogar ein Werkzeug, das in der Lage ist, solche Tests automatisch zu generieren.

Das Debuggen von Verhaltensmodellen aus denen ausführbarer Quelltext generiert wird, stellt bei vielen Ansätzen ein Problem dar. Es gibt dort entweder die Möglichkeit mit einem herkömmlichen Debugger den generierten Quelltext zu debuggen, oder aber das Modell unabhängig vom generierten Code in einem Interpreter schrittweise auszuführen. Beide Verfahren haben erhebliche Nachteile. Beim quelltextbasierten Debuggen muss der Entwickler auf die Quelltextebene wechseln, was beim modellbasierten Ansatz ja eigentlich verhindert werden sollte. Um die Codestellen, die er im Debugger analysiert auch dem Modell zuordnen zu können, muss er im Kopf die Codegenerierung rückgängig machen. Die Tatsache, dass generierter Code meist auch noch recht unleserlich ist, erschwert das Problem zusätzlich. Der interpretierte Ansatz birgt Probleme sobald sich Interpreter und generierter Code unterschiedlich verhalten. Das kann beispielsweise dann der Fall sein, wenn externe Bibliotheken eingebunden werden, die nur im Quelltext vorliegen. So kann es beispielsweise zu Fehlern kommen, die im fertigen Code auftauchen, im Interpreter aber nicht passieren und somit auch nicht untersucht werden können.

1.1. Forschungsgegenstand

Gegenstand dieser Arbeit ist die Erforschung von Methoden, Techniken und Werkzeugen zur Fehlersuche in modellbasierten Softwareentwicklungsprozessen. Dabei liegt der Schwerpunkt auf Fehlersuche durch Testen und Debuggen. Es wird insbesondere erörtert, inwieweit Werkzeugunterstützung und Generatoren bei der Fehlersuche eingesetzt werden können.

Hierzu wurde zuerst am Lehrstuhl für Software Engineering der Universität Kassel unter meiner Mitwirkung ein komplett modellbasierter Softwareentwicklungsprozess entwickelt. Dieser Prozess ist durch Anwendungsszenarien getrieben und macht Objektstrukturen und deren Änderungen während der Programmausführung zum Modellierungsschwerpunkt. Der Prozess erhält durch das am Lehrstuhl mitentwickelte CASE Tool Fujaba Werkzeugunterstützung.

Dieser Prozess wird durch diese Arbeit um einen Testgenerator ergänzt, der in der Lage ist, aus den Anwendungsszenarien ausführbare Tests zu generieren, die testen, ob die beschriebene Anforderung von der entwickelten Software abgedeckt wird. Die prinzipielle Machbarkeit eines solchen Testgenerators wurde bereits in meiner Diplomarbeit [Gei04] gezeigt. Der Ansatz wurde hier weiter verfeinert und durch zusätzliche Funktionalität, wie Alternativszenarien oder Coverageanalyse, ergänzt, deren Notwendigkeit sich aus der praktischen Anwendung des Ansatzes ergab.

Des Weiteren wird in dieser Arbeit erforscht, wie Debugging auf Modellebene funktionieren kann. Die eingesetzten Debuggingtechniken sollen den auf Quelltextebene

üblichen Techniken in Komfort und Einsetzbarkeit entsprechen, trotzdem jedoch die durch das modellbasierte Vorgehen erreichte höhere Abstraktionsebene erhalten. Im Zuge dieser Forschung wurde in dieser Arbeit und in von mir betreuten Diplomarbeiten ein modellbasierter Debugger, ein Objektbrowser zur Analyse von komplexen Objektstrukturen und ein Verfahren zum „Rückwärtsdebuggen“ auf Modellen entwickelt.

Die entwickelten Ansätze und Werkzeuge wurden durch den praktischen Einsatz in der Lehre aber auch in Entwicklungsprojekten evaluiert und verbessert. Alle entwickelten Werkzeuge wurden in das CASE Tool Fujaba und die Entwicklungsumgebung Eclipse integriert.

1.2. Aufbau des Dokuments

Die Arbeit unterteilt sich in drei Hauptteile. Teil I beschreibt den von Albert Zündorf, Ira Diethelm und mir entwickelten und dieser Arbeit zugrunde liegenden modellbasierten Entwicklungsprozess. In Teil II wird die Aktivität des Testens in diesem Softwareentwicklungsprozess näher beleuchtet. Hierbei wird im Detail auf die in dieser Arbeit entwickelte Testgenerierung eingegangen. Teil III beschäftigt sich dann mit der Fehlersuche bei modellbasierten Vorgehensweisen. Hier werden insbesondere die in dieser Arbeit entwickelten Debuggingverfahren und -hilfen vorgestellt.

Diese drei zentralen Teile der Arbeit besitzen einen ähnlichen Aufbau. Im ersten Kapitel wird jeweils das dem zugehörigen Teil zugrunde liegende Problem beschrieben. Im nächsten Kapitel werden mögliche Lösungsansätze skizziert und danach die konkrete Umsetzung, wie sie in dieser Arbeit erfolgt ist, dargestellt. Es folgt eine Beschreibung der Erfahrungen, die beim praktischen Einsatz der entwickelten Werkzeuge und Konzepte gemacht werden konnten. Im darauf folgenden Kapitel wird auf verwandte Arbeiten eingegangen und der gewählte Ansatz mit anderen verglichen. Im letzten Kapitel jedes Teils wird schließlich ein Fazit gezogen.

Der letzte Teil der Arbeit (Teil IV) fasst nochmal die Ergebnisse der Arbeit zusammen, zieht ein Fazit und skizziert sich an die Arbeit anschließende, weiterführende Forschungsfragen.

2. Anforderungen

In dieser Arbeit sind mehrere Softwarebausteine, die den Fujaba Process unterstützen, wie z.B. Testgeneratoren oder Debuggingwerkzeuge, entwickelt worden. Dieses Kapitel beschreibt die generellen Anforderungen, die an diese Softwarekomponenten gestellt werden.

2.1. Kernanforderungen

Es soll ein Testgenerator entwickelt werden, der Anwendungsszenarien in ausführbare Testfälle umsetzt. Zusätzlich soll ein Debugger für Modelle erstellt werden. Generell sollen die entwickelten Komponenten komplett auf Modellebene arbeiten. Es soll also an keiner Stelle notwendig sein, in den Quelltext zu wechseln. Trotzdem soll dies allerdings überall möglich sein. Es soll also an jeder Stelle die Navigierbarkeit sowohl zwischen zusammengehörigen Modellelementen als auch zwischen Modellelementen und dem daraus generierten Code gewährleistet werden. So soll es beispielsweise möglich sein, aus dem Quelltext eines generierten Tests zurück in das Szenario zu wechseln, aus dem dieser Test generiert wurde. Dies ermöglicht den nahtlosen Einsatz von Techniken und Werkzeugen, die bislang nur auf Quelltextebene arbeiten.

2.2. Ergänzende funktionale Anforderungen

Der Testgenerator soll vollautomatisch aus Szenarien Testfälle generieren können. Zusätzlich soll aber auch ein manuelles Modellieren von Testfällen ermöglicht werden. Solche Tests sind neben den Szenarientests, die die Anforderungen abtesten, auch meistens erforderlich, um beispielsweise interne Funktionalität zu testen.

Der modellbasierte Debugger soll die wichtigsten Funktionen von üblichen textuellen Debuggern bereitstellen. Er soll beispielsweise das schrittweise Ausführen von Modellen beherrschen, in der Lage sein, Variablenbelegung und die momentan im Speicher bestehende Datenstruktur darstellen zu können und es erlauben Haltepunkte an markanten Stellen zu setzen.

Weitere detaillierte Anforderungen werden in den Teilen II und III beschrieben, in denen die Testgenerierung und der modellbasierte Debugger behandelt werden.

2.3. Nichtfunktionale Anforderungen

Die entwickelten Verfahren und Werkzeuge sollen auch für große Modelle skalieren. Es soll also möglich sein, auch für komplexe Objektstrukturen Tests generieren und diese dann debuggen zu können.

Die eingesetzten Verfahren sollen gut verständlich sein. Es soll also beispielsweise möglich sein, ein Szenario, aus dem ein Testfall generiert wird, mit einem fachfremden Domänenexperten zu diskutieren. Außerdem sollen es die Debuggingtechniken Anfängern ermöglichen, Programmabläufe besser zu verstehen. Die erstellten Werkzeuge sollen also nach kurzer Lernphase auch für Anfänger und Fachfremde nutzbringend einsetzbar sein. Trotzdem sollen die Werkzeuge natürlich auch dem erfahrenen Nutzer einen Mehrwert bringen. Das heißt, an Stellen, die zusätzliche Schritte einführen, um die Komplexität für Anfänger möglichst gering zu halten, muss es Abkürzungen geben. Diese Abkürzungen können dann von erfahrenen Anwendern genutzt werden, die sonst durch die zusätzlichen Schritte aufgehalten würden.

Zusätzlich sollen alle Softwarekomponenten in den in Teil I vorgestellten modellbasierten Softwareentwicklungsprozess nahtlos integriert werden und diesen, wenn möglich, leiten.

2.4. Zielplattform

Die Konzepte, die den zu entwickelnden Softwarekomponenten zugrunde liegen, sollen so allgemein wie möglich gehalten werden. Es soll also einfach möglich sein, diese Konzepte auf andere Werkzeuge, Programmiersprachen und Entwicklungsumgebungen zu übertragen. Trotzdem muss eine konkrete Implementierung, wie sie im Rahmen dieser Arbeit entstand, natürlich auf einer bestimmten Platform aufbauen. Die Software wurde komplett in Java umgesetzt und auch als Zielsprache des Codegenerators wurde Java verwendet. Die Komponenten sind in das CASE Tool Fujaba integriert. Genauer setzen sie auf der Eclipse Version von Fujaba, Fujaba4Eclipse, auf. Die Entwicklungsumgebung Eclipse stellt selbst schon Komponenten zur Umsetzung von Werkzeugen, wie etwa Debuggern, bereit. Hier sollen sich die zu entwickelnden Komponenten natürlich ebenso integrieren. Als Testumgebung für die generierten Testfälle wird das in der Java-Entwicklung weit verbreitete JUnit Framework verwendet.

Teil I.

Der Fujaba Process

3. Motivation

Im Jahre 2001 wurde an der Technischen Universität Braunschweig das XProM (eXtreme Projekt Management) Projekt von Professor Zündorf und mir gestartet [GSZ03]. In diesem Projekt wurde der FUjaba Process (FUP) [GZ05] basierend auf den Ideen von [KNNZ00] entwickelt. In den folgenden Jahren wurde dieser Prozess von Professor Zündorf, Ira Diethelm und mir weiter ausgearbeitet und verfeinert [GZ06a].

Der FUP ist ähnlich dem Rational Unified Process ([JBR99]) ein Usecase-basierter, iterativer Softwareentwicklungsprozess. Der Fujaba Process stellt ein klar definiertes, technisch, inhaltlich orientiertes Vorgehensmodell bereit, das durch Fujaba komplett werkzeugunterstützt ist. Es sind nicht nur die einzelnen Phasen mit den dazugehörigen Abschlussdokumenten fest vorgegeben, sondern der FUP definiert zusätzlich, welche UML Diagramme in welcher Phase zu welchem Zweck benutzt werden können und wie sich die einzelnen Diagramme aus einander ableiten lassen.

Der Fujaba Process ist ein agiler Softwareentwicklungsprozess bei dem sich jede Iteration in die folgenden Schritte unterteilt:

- Zu Beginn jeder Iteration werden (neue) Funktionalitäten mit Hilfe von Usecase Diagrammen gesammelt.
- Für jeden Usecase werden textuelle Szenario-Beschreibungen angelegt.
- Jedes Szenario wird im Team diskutiert und mit Hilfe des *Objektspiels* (siehe Abschnitt 5.2) analysiert.
- In der Analysephase werden die einzelnen Schritte der Szenarien als UML Objekt- oder Kollaborationsdiagramme modelliert. Das Ergebnis sind sogenannte Storyboards, die den Ablauf der Beispielszenarien zeigen.
- Aus den Storyboards lassen sich Informationen über Klassen, Attribute, Methoden und Assoziationen gewinnen. Diese werden in einem initialen Klassendiagramm gesammelt. Dieses Klassendiagramm muss üblicherweise noch verfeinert werden. Hier werden jetzt z.B. Kardinalitäten angepasst.
- Aus diesem initialen Klassendiagramm werden nun vollautomatisch Klassenrümpfe für die Implementierung generiert.
- Im nächsten Schritt werden vollautomatisch JUnit Test aus den Storyboards generiert.
- Als nächstes muss das Verhalten der Applikation implementiert werden. Der Entwickler kann direkt Java-Code in die generierten Klassen schreiben, oder das Verhalten mit Hilfe von sogenannten Storydiagrammen modellieren. In

[DGZ04] zeigen wir, wie sich solche Diagramme systematisch aus den Storyboards herleiten lassen. Zusätzlich beschreibt unser Dreisprung-Ansatz, siehe [DGZ08], wie sich Kontrollstrukturen und komplexe Algorithmik systematisch analysieren und in Storydiagrammen umsetzen lassen.

- Wenn die Implementierung abgeschlossen ist, sollte die Ausführung aller generierten Tests erfolgreich sein. Fehlschlagende Tests können auf Szenarien hinweisen, die noch nicht von der Implementierung abgedeckt sind. Andererseits kann man mit Überdeckungsverfahren Fälle finden, für die noch kein Szenario existiert, vgl. [GZ05].

Abbildung 3.1 zeigt eine schematische Darstellung des Prozesses. In den einzelnen Abschnitten von Kapitel 5 werden die einzelnen Phasen des FUPs aus Abbildung 3.1 im Detail erläutert. In Kapitel 4 wird das Beispiel, das in dieser Arbeit verwendet wird, vorgestellt. In Kapitel 6 wird der für den FUP neu entwickelte Codegenerator erklärt. Kapitel 7 diskutiert verwandte Arbeiten, Kapitel 8 zeigt unsere Praxiserfahrungen mit dem FUP und Kapitel 9 zieht schließlich ein Fazit.

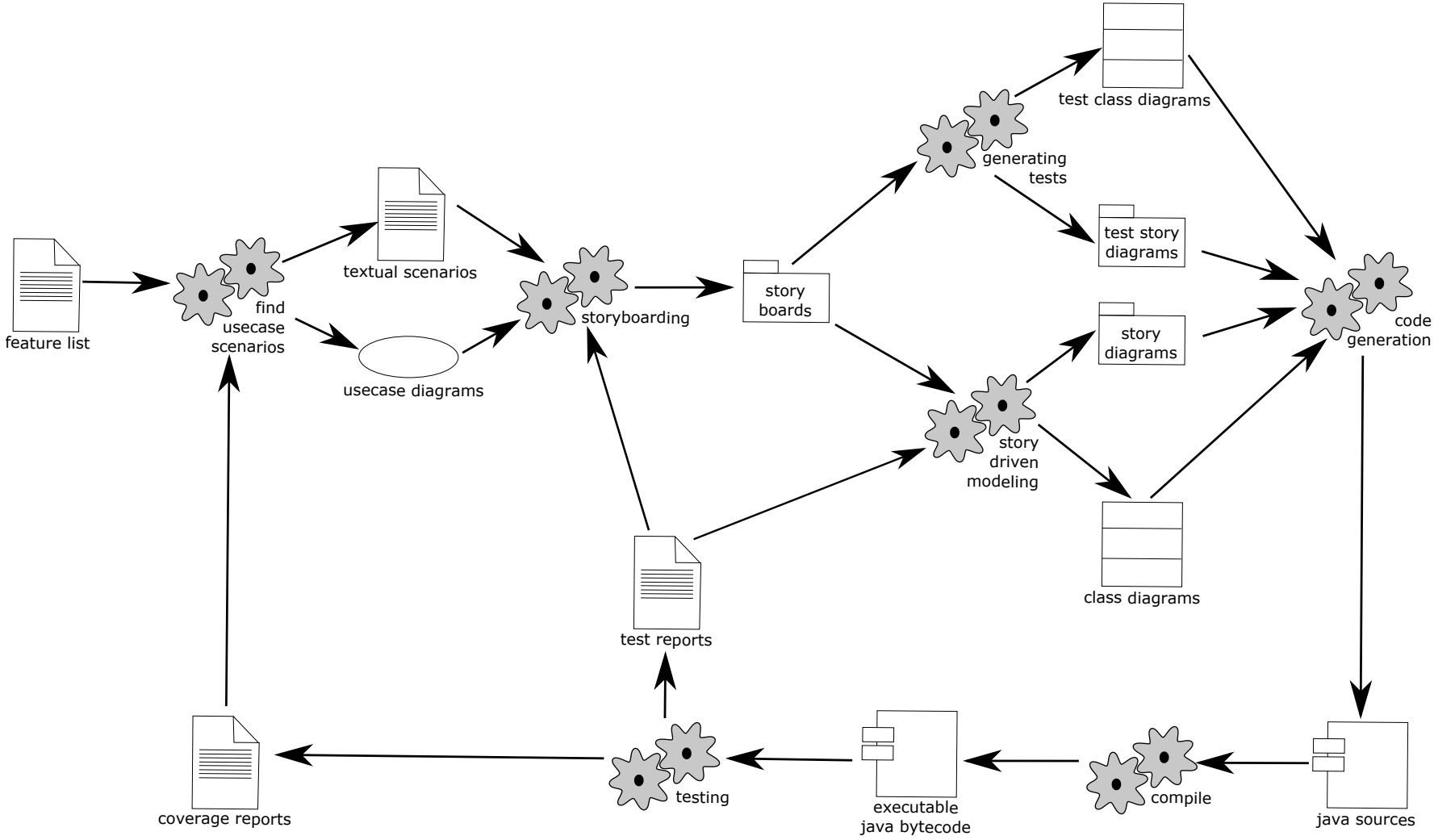


Abbildung 3.1.: Schematische Darstellung des Fujaba Process

4. Beispiel

Das in dieser Arbeit verwendete Beispiel war Fallstudie auf dem zweiten "Workshop on Graph-Based Tools (GraBaTs)" 2004 in Rom, vgl. [MST04]. Einreichungen sollten mit den Werkzeugen ihrer Wahl einen Statecharteditor bauen, in dem sich Statecharts, wie sie in der UML 1.5 Spezifikation beschrieben sind, modellieren lassen. Als Mindestvoraussetzung sollte es möglich sein das Statechart aus Abbildung 4.1 im Editor zu erstellen. Zusätzlich sollten mögliche Transformationen auf den Statechartmodellen, wie etwa die Konvertierung in einen endlichen Automaten, beschrieben werden. Unsere Einreichung benutzt die Fujaba Tool Suite um einen Statecharteditor zu erstellen, siehe [GZ04, GGZ+05, GZ06c]. Die darin beschriebene Lösung dient im folgenden als Beispiel, um den Fujaba Process zu erläutern.

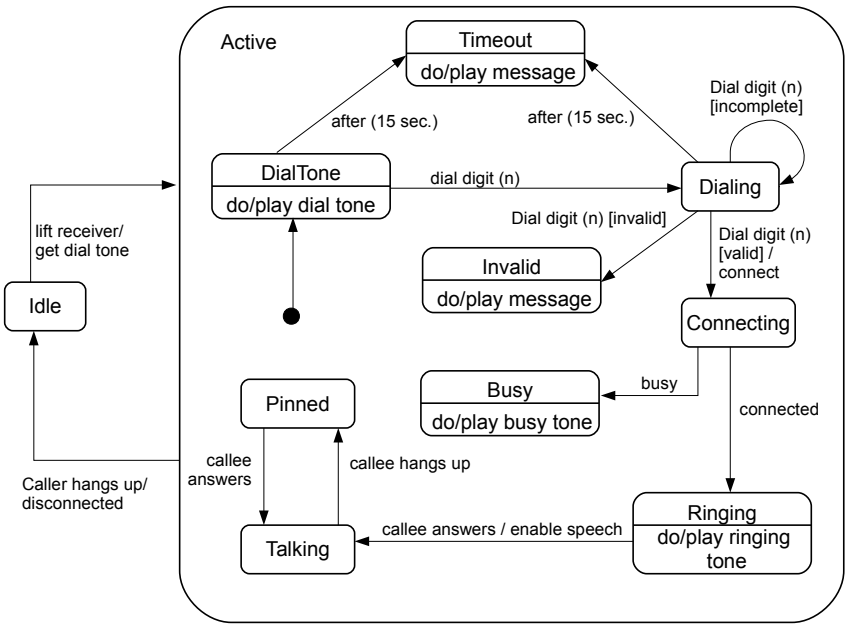


Abbildung 4.1.: Beispielstatechart für den Statecharteditor

5. Aufbau

5.1. Sammeln der Anforderungen mit Usecases

Der erste Schritt im FUP ist das Sammeln von Anforderungen in Usecasediagrammen. Abbildung 5.1 zeigt ein solches Usecasediagramm für das Statechart Beispiel in Fujaba. Das Diagramm enthält zwei Anforderungen: Zum einen soll eine Transformation geschrieben werden, die ein bestehendes Statechart in einen endlichen Automaten überführt (Usecase Flattening) zum zweiten soll ein Interpreter für Statecharts implementiert werden (Usecase Execution).

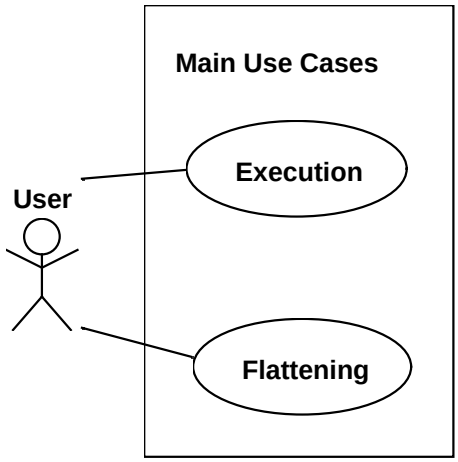


Abbildung 5.1.: Usecasediagramm für den Statecharteditor

Nachdem die zu entwickelnde Anwendung jetzt durch die Usecases in Teilprobleme unterteilt wurde, gilt es die Detailanforderungen zu jedem Usecase zu sammeln. Diese werden üblicherweise zusammen mit dem Kunden erarbeitet und in textueller Form festgehalten. Häufig werden Anforderungen auch in einem Datenbanksystem wie z.B. DOORS [IBM10b] abgelegt. In vielen Fällen führt dies zu einer Ansammlung von schwach zusammenhängenden „Anforderungsschnipseln“. Solche Anforderungen behandeln einen kleinen Teil der Systemfunktionalität in einem stark fokussierten Kontext. Eine solche Anforderung für den Usecase Flattening könnte z.B. folgendermaßen aussehen:

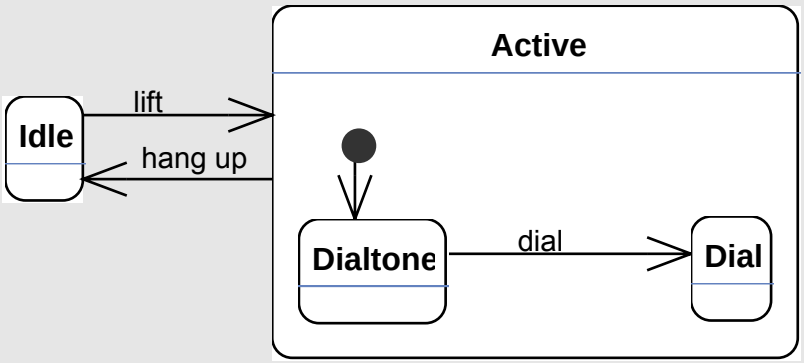
2.2. Flattening
Der Editor soll eine Operation anbieten, um ein Statechart in einen endlichen Automaten zu konvertieren. Hierbei soll die Ausführungssemantik des Statecharts, wie sie im UML2 Standard beschrieben ist, erhalten bleiben. Es müssen also geschachtelte

Or-States durch geeignetes Umrouten der Transitionen entschachtelt werden. Hierbei ist unter anderem die Priorisierung von inneren Kanten bei Namensgleichheit zu beachten. ...

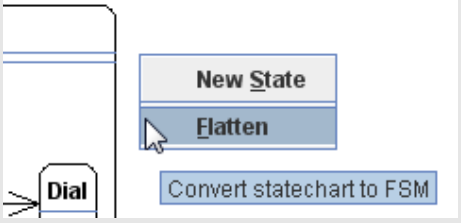
Diese Art der Anforderungsbeschreibung ist weit verbreitet. Wir haben allerdings die Erfahrung gemacht, dass diese Beschreibungen zu allgemein und meist auch zu abstrakt sind. Dies erschwert beispielsweise eine Diskussion mit dem Kunden. In solchen Diskussionen konnten wir beobachten, dass zur Erklärung komplexer Sachverhalte meist sehr konkrete Beispiele herangezogen wurden. Daher baut der FUP auf eben solchen konkreten Beispielen auf.

Der FUP ist also (wie viele andere) ein Szenario-getriebener Prozess. Daher werden im nächsten Schritt zu jedem Usecase textuelle Beispielabläufe geschrieben. Diese Beispielabläufe beschreiben Aktorinteraktion und die zugehörige Reaktion des Systems. Ein Szenario beginnt mit der Startsituation, gefolgt von mehreren Ausführungsschritten und endet mit der Endsituation. Die Ausführungsschritte können entweder Aktionen des Aktors sein (sogenannte *actor steps*) oder die Reaktion des Systems beschreiben (*system steps*). Der folgende Block zeigt ein Szenario für den Usecase Flattening:

start situation Der Editor zeigt das unten dargestellte Statechart mit einem Or-State `Active`, der eine eingehende Transition `lift` und eine ausgehende Transition `hang up` besitzt.



1. step Der Benutzer `Karl` ruft über das Kontextmenü des Editors das Kommando zum Konvertieren des Statecharts in einen endlichen Automaten auf, wie im nachstehenden Screenshot gezeigt.

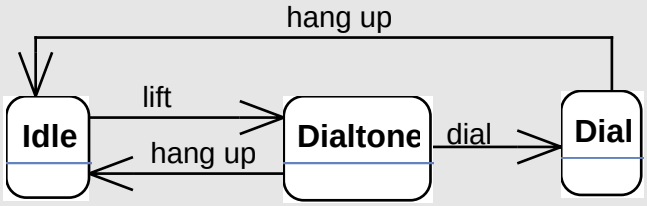


2. step Die eingehende Transition `lift` wird auf den inneren Startstate `Dialtone` umgeleitet.

3. step Die ausgehende Transition `hang up` wird durch ausgehende Transitionen von allen inneren States ersetzt. Es entsteht also eine Transition vom State `Dialtone` zum State `Idle` sowie eine Transition von `Dial` nach `Idle`.

4. step Der jetzt isolierte Or-State wird gelöscht. Seine inneren Zustände sowie die zugehörigen Transitionen bleiben hierbei erhalten.

result situation Das Statechart ist in einen endlichen Automaten konvertiert.



Dieses Szenario beschreibt den Standardfall der Konvertierung eines einfachen Or-States. Szenarien für Fehler- oder Sonderfälle (wie z.B. innere und ausgehende Transitionen mit gleichem Namen) sollten selbstverständlich durch zusätzliche Szenarien behandelt werden.

Nach unseren Erfahrungen mit Studentenarbeiten, Forschungsarbeiten und einigen Projekten mit Partnern aus der Industrie, bilden solche Szenario-basierten Anforderungen eine deutlich bessere Grundlage für die weiteren Entwicklungsschritte als die traditionellen „Anforderungsschnipsel“, siehe auch Kapitel 8. Falls die Anforderung allerdings bereits aus solchen Schnipseln bestehen, schlagen wir vor, diese in Usecase Szenarien zu übersetzen, bevor mit der Analysephase begonnen wird.

In unseren Projekten konnten sich Entwickler sehr leicht auf textuelle Szenariobeschreibungen einigen. Allerdings sind diese Szenarien meist noch ziemlich vage und teilweise wenig konkret. So kann es passieren, dass selbst wenn alle Beteiligten mit den textuellen Szenarien einverstanden sind, später beim Formalisieren der Szenarien tiefgreifende Missverständnisse und Meinungsverschiedenheiten aufgedeckt werden können. Daher halten wir textuelle Anforderungen allein für nicht ausreichend für eine gute Anforderungsanalyse.

5.2. Anforderungsanalyse mit Objektspiel

Als nächstes muss ein geeignetes Datenmodell gefunden werden, mit dem sich die Szenarien formalisieren und später dann implementieren lassen. Dementsprechend wird nach einer geeigneten Form gesucht, die Elemente der Szenarien im Rechner darstellen zu können. Es findet also eine Übersetzung von Domänenebene auf Modellebene statt. Viele Ansätze schlagen hier das Entwickeln eines Klassendiagramms vor, siehe z.B. [Bal96]. Wir haben jedoch beobachten können, dass das direkte Ableiten

des Klassendiagramms aus den textuellen Szenarien eine große Abstraktionshürde darstellt. Durch diese Hürde führt dieser Schritt bei Anfängern zu großen Schwierigkeiten (siehe [Die07]). Auch erschwert diese Hürde schon in dieser frühen Phase die Kommunikation mit Nicht-Informatikern, wie Kunden und Domänenexperten. Zusätzlich ist eine große Hürde auch immer ein Punkt, an dem leicht Fehler auftreten können und Fehler in einer so frühen Phase können oft gravierende Auswirkungen haben, gerade, wenn sie erst spät entdeckt werden. Daher zieht der FUP auf dem Weg zum Klassendiagramm zwei Zwischenschritte ein, um die Komplexitätssprünge zu verkleinern. Diese Schritte werden in den folgenden zwei Abschnitten beschrieben.

Um die Usecase Szenarien zu verfeinern und später zu formalisieren, schlägt der FUP das sogenannte Objektspiel als Zwischenschritt vor. Im Objektspiel wird jeder Schritt des zu untersuchenden Usecase Szenarios einzeln betrachtet und mit einem Objekt- oder Kollaborationsdiagramm modelliert. Dies funktioniert auch gut in Gruppenarbeit. Die Entwickler entwerfen ihre Objektdiagramme am Whiteboard und können so darüber diskutieren, Elemente ändern, hinzufügen oder auswischen. Die Schritte des Objektspiels werden als Sequenz von Objektdiagrammen protokolliert, z.B. durch Abfotografieren oder mithilfe eines digitalen Smartboards. Das Objektspiel lässt sich in bestimmten Fällen auch im Objektbrowser eDOBS rechnergestützt durchführen, siehe Abschnitt 18.2.4.

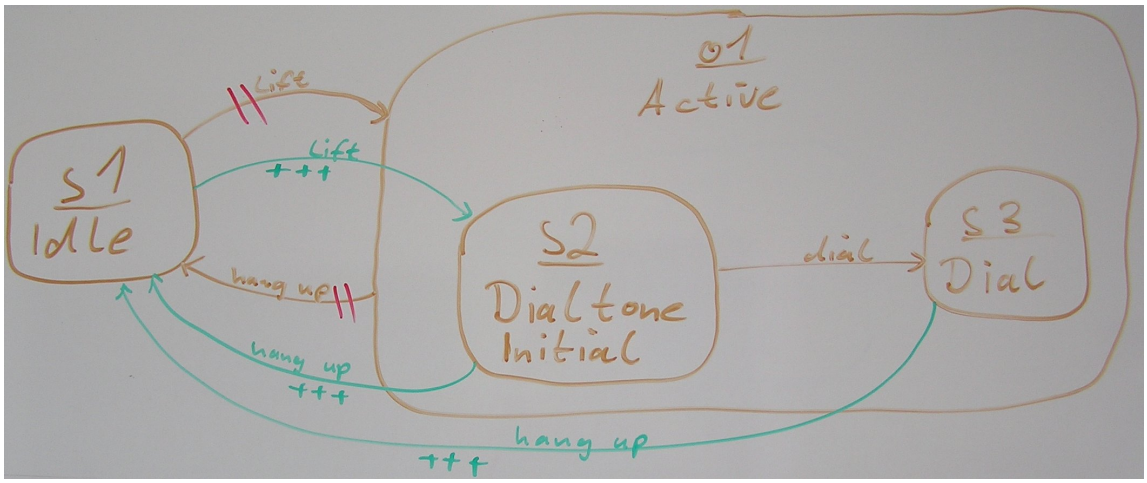


Abbildung 5.2.: Objektspiel

Abbildung 5.2 zeigt ein Objektdiagramm des Objektspiels für das Statechart-Beispiel. Das Objektspiel begann mit einem Statechart bestehend aus dem State `s1` und dem Or-State `o1`. `o1` enthält zwei weitere States `s2` und `s3`. Der Or-State besitzt eine eingehende Transition `lift` sowie eine ausgehende Transition `hang up`. Um nun das Statechart in einen endlichen Automaten zu überführen, muss der Or-State aufgelöst werden. Dies geschieht dadurch, dass die `lift` Transition gelöscht wird und eine neue zum Startzustand des inneren Statecharts `o2` gezogen wird. Das Löschen wird durch das zweimalige durchstreichen mit rotem Stift visualisiert, neu anzulegende Links und Objekte werden in grün gezeichnet und mit "+++" markiert. Im nächsten Schritt wird nun die ausgehende Transition gelöscht und durch

zwei neue, von den States `s2` und `s3` ausgehende Transitionen, ersetzt. Der Flattening Vorgang ist damit abgeschlossen. Zur besseren Übersichtlichkeit können einmal gelöschte Objekte am Whiteboard nach dem Fotografieren des zugehörigen Schrittes auch ausgewischt werden. So könnte man nun in Abbildung 5.2 die Links sowie den State `o1` wegwischen und würde nur noch den endlichen Automaten sehen. Das Objektspiel vermittelt also einen ersten Überblick über die zu verwendende Daten- bzw. Objektstruktur. Zusätzlich können erste Details des zu entwickelnden Algorithmus diskutiert werden.

Ziel des Objektspiels ist der Wechsel der Perspektive von "Ich führe diesen Szenarioschritt aus" hin zu "Der Computer führt diesen Szenarioschritt aus". Durch diesen Perspektivwechsel muss der Entwickler spezifizieren, wie die Datenstruktur für einen bestimmten Szenarioschritt im Programm aussieht und wie das Programm auf diesen Daten arbeiten soll. Zusätzlich erfordert dieser Perspektivwechsel und die Benutzung semi-formaler Objektdiagramme eine präzisere Formulierung der Datenstruktur und des Ablaufs als es die textuellen Beschreibungen tun. Nach unseren Erfahrungen wird in dieser Phase deutlich mehr und intensiver diskutiert, da hier oft Missverständnisse und Meinungsverschiedenheiten zwischen Gruppenmitgliedern zu Tage treten. Es hat sich in unseren Projekten als sehr wertvoll herausgestellt solche Missverständnisse in einer so frühen Phase zu entdecken.

Zusätzlich zum Entwicklerteam können auch Domänenexperten und Kunden am Objektspiel teilnehmen. Solche haben üblicherweise wenig Erfahrung in objektorientierter Modellierung, UML oder Programmierung. Um solchen Gruppenmitgliedern die Teilnahme an der Diskussion zu erleichtern, sollte man die Objektdiagramme der Anwendungsdomäne anpassen. Dies kann durch zusätzliche Icons an den Objekten geschehen oder durch geeignete Abstraktion. In Abbildung 5.2 wurde beispielsweise die `contains` Beziehung, die das Enthaltensein der States `s2` und `s3` in dem Or-State `o1` beschreibt, durch Schachtelung abstrahiert. Des Weiteren wurden die Transitionen als Links dargestellt. Allerdings enthalten die Transitionen zusätzliche Informationen (den Labelnamen) und Fujaba bzw. Java unterstützt keine attributierten Kanten. Hier sind also Kantenobjekte nötig von denen im Objektspiel abstrahiert wurde. Dadurch sieht Abbildung 5.2 einem echten Statechart schon sehr ähnlich. Nach unseren Erfahrungen helfen solche Vereinfachungen den Kunden enorm, ihr Domänenwissen im Objektspiel an die Entwickler weiterzugeben.

Um Missverständnisse unter den Entwicklern zu vermeiden, kann es sinnvoll sein, das Objektspiel, oder zumindest Teile des Objektspiels, nochmal mit aufgelösten Abstraktionen durchzugehen. Für das Beispiel aus Abbildung 5.2 müssten dann für alle Transitionen Objekte erstellt und die Schachtelung durch Links modelliert werden.

Im FUP werden typischerweise Objekt- und Kollaborationsdiagramme während des Objektspiels eingesetzt. Dies ist sinnvoll, wenn der Programmablauf überwiegend von der Objektstruktur und den Änderungen hierin abhängt. Bei eher statischer Objektstruktur und nachrichtenbasiertem Verhalten, wie z.B. bei Protokollen, können an dieser Stelle auch Sequenzdiagramme genutzt werden.

Das vorgeschlagene Vorgehen reduziert die vorhandene Komplexität auf drei Arten:

- Durch die Verwendung von Szenarien kann sich der Entwickler auf eine konkrete Situation zur Zeit beschränken. Der Entwickler behandelt also immer einen bestimmten Durchlauf und muss sich daher hier beispielsweise noch nicht mit komplexen Fallunterscheidungen beschäftigen.
- Innerhalb eines Szenarios kann der Entwickler jeden Szenarioschritt separat bearbeiten. Er kann sich also darauf konzentrieren, wie sich das System in dieser bestimmten Situation zu diesem bestimmten Zeitpunkt verhält.
- Die Szenarien sind auf Instanzebene beschrieben. Es werden also Objekte statt Klassen und Links statt Assoziationen verwendet. Der Entwickler muss also nur die aktuelle Datenstruktur beschreiben und nicht etwa schon das abstraktere Datenschema in dieser frühen Phase spezifizieren.

Wir haben das Objektspiel sehr erfolgreich in Kursen zur Einführung in die objektorientierte Modellierung an der Universität aber auch in der Oberstufe an Gymnasien eingesetzt. Die Diskussion auf Objektebene erleichterte unseren Studenten / Schülern das Verständnis der Funktionsweise objektorientierter Programme enorm. Weiterhin setzen wir das Objektspiel in Forschungs- und industriellen Projekten ein. Ein Industrieprojekt beschäftigt sich beispielsweise mit der Konfiguration elektrischer Elemente in einem Auto, vgl. [GSZ+05c]. Unsere Ansprechpartner in diesem Projekt sind zwei Elektrotechniker mit sehr gutem Domänenwissen, aber wenig Programmier- und keinerlei UML-Erfahrung. Nach einer kurzen Einführung, waren die Elektrotechniker in der Lage beim Objektspiel mitzudiskutieren. Nach einigen Wochen zeichneten sie eigene Objektdiagramme, um bestimmte Aspekte des Designs mit uns zu diskutieren und um uns auf noch nicht berücksichtigte Sonderfälle hinzuweisen. In diesem Projekt war das Objektspiel eine große Hilfe für den Transfer des Domänenwissens zwischen Elektrotechnikern und Softwaretechnikern.

5.3. Storyboarding - Verfeinern der Usecase Szenarien

Als Resultat der Objektspielphase erhält man für jedes Szenario eine Sequenz von Fotos, die während der Diskussion gemacht wurden. Im nächsten Schritt werden diese Ergebnisse als sogenannte Storyboards im Fujaba CASE Tool festgehalten. Ein Storyboard ist eine Sequenz von Kollaborationsdiagrammen, welche die Änderungen in der Objektstruktur ähnlich wie in einem Comic visualisiert. Dieses Übersetzen des Objektspiels resultiert in einer formalen, druckbaren und maschinenlesbaren Form, die nun z.B. Konsistenzchecks oder Toolunterstützung für nachfolgende Schritte ermöglicht. Normalerweise werden beim Storyboarding weitere Details hinzugefügt, die noch nicht im Objektspiel enthalten waren, wie z.B. Typinformationen, Linknamen, Attributnamen usw. Zusätzlich werden in diesem Schritt oft noch fehlende (Hilfs-)Objekte oder Funktionalität hinzugefügt, die im Objektspiel beispielsweise aus Gründen der Übersichtlichkeit ausgelassen wurden. Die Abbildungen 5.3 und 5.4 zeigen ein solches Storyboard für das Beispielszenario.

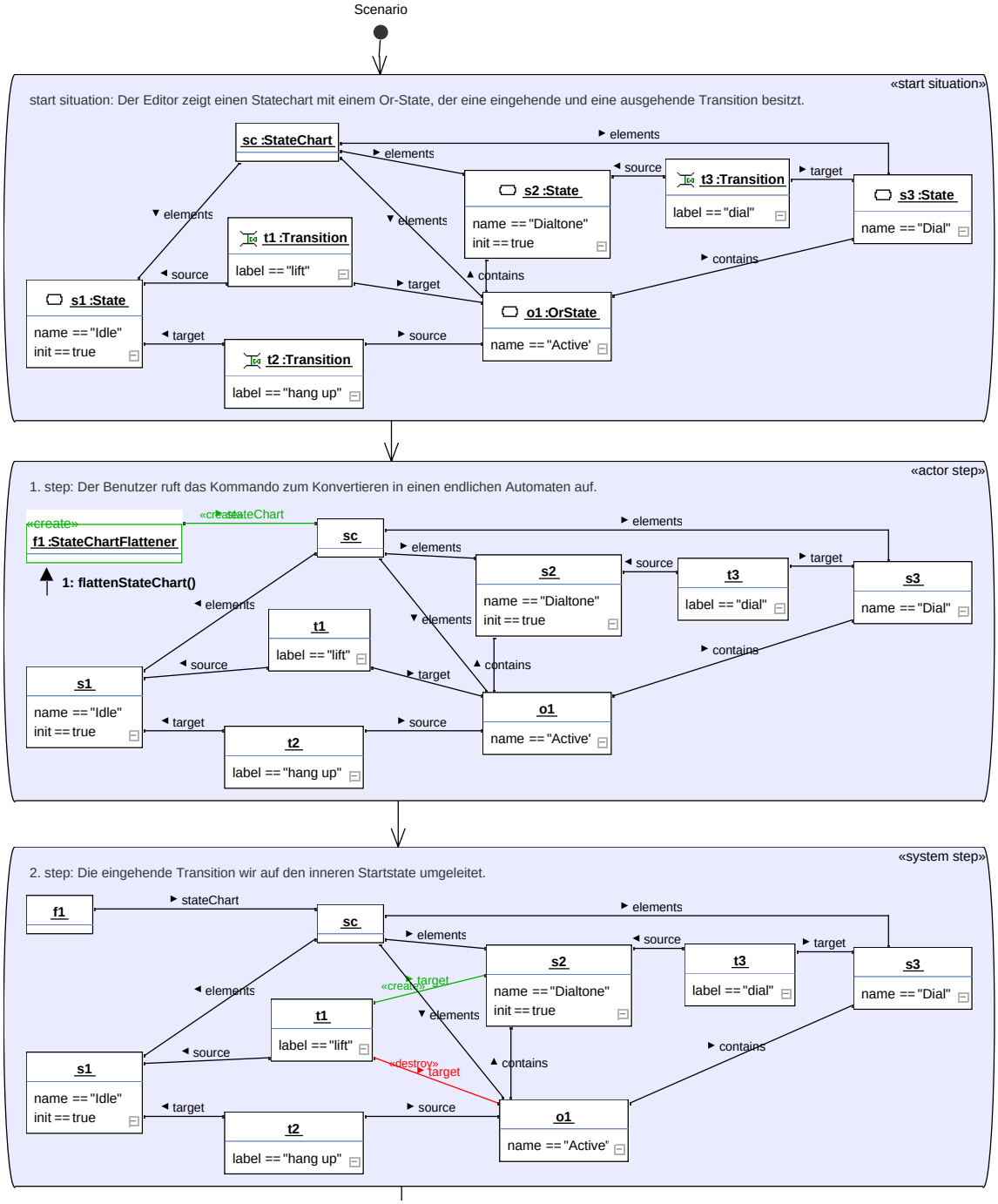


Abbildung 5.3.: Storyboard Teil 1

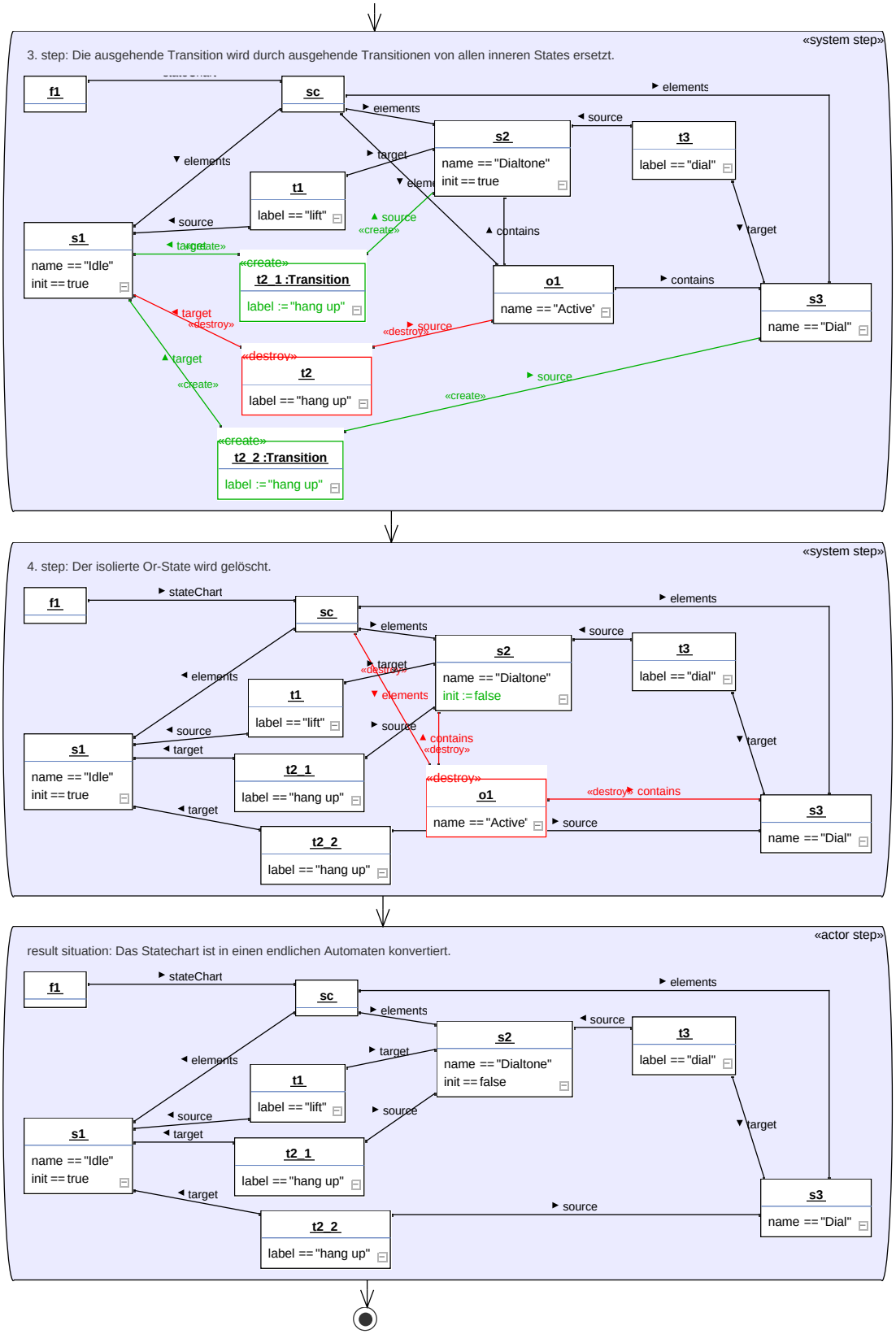


Abbildung 5.4.: Storyboard Teil 2

Im ersten Schritt des Storyboardings bietet das Fujaba Tool ein Kommando, um für ein textuelles Szenario ein initiales Storyboard anzulegen. Dieses Storyboard besteht aus einer Aktivität pro Schritt im textuellen Szenario. Der Text des Schrittes wird als Kommentar der Aktivität angezeigt. Aufgabe des Entwicklers ist es nun, aufbauend auf den Ergebnissen des Objektspiels, jede Aktivität mit einem Objektdiagramm zu füllen, das den entsprechenden Schritt modelliert. Um Dynamik abbilden zu können, dürfen diese Objektdiagramme auch Nachrichten, Attributänderung sowie das Erzeugen und Löschen von Objekten und Links enthalten.

Die erste Aktivität in Abbildung 5.3 beschreibt die Ausgangssituation des Beispielszenarios für den Usecase Flattening. Die Objekte `s1`, `s2`, `s3` und `o1` aus dem Objektspiel, die die einzelnen States repräsentieren, tauchen auch hier auf. Die Schachtelung der States `s2` und `s3` im Or-State `o1` wird nun durch `contains` Links modelliert, da Objektdiagramme keine Hierarchie erlauben. Weiterhin hat sich der Entwickler entschieden den Namen eines States in einem String-Attribut mit dem Bezeichner `name` zu speichern. Die Tatsache, das der State `s2` der Startzustand des inneren Statecharts ist, wurde durch das boolsche Flag `init` modelliert. Gegenüber dem Objektspiel wurde eine weitere Abstraktion aufgelöst: Für die Transitionen wurden eigene Objekte angelegt. Dies ist notwendig um die `label` Information zu speichern, da Fujaba keine attributierten Kanten unterstützt. Zusätzlich wurde das Statechart-Objekt `sc` hinzugefügt um zu modellieren, dass alle States zu einem gemeinsamen Statechart gehören.

Fujaba ermöglicht es graphische Stereotypen in Objektdiagrammen zu benutzen. Ähnlich wie im Objektspiel sollen diese Annotationen Domänenexperten und Kunden ein Mitarbeiten auch in dieser Phase erleichtern. Graphische Stereotypen wurden exemplarisch in der ersten Aktivität in Abbildung 5.3 verwendet.

Die zweite Aktivität zeigt üblicherweise die Aktion die die Szenarioausführung anstößt. Dies ist im FUP zumeist eine Nachricht / ein Methodenaufruf. Im Beispiel aus Abbildung 5.3 ist das die Methode `flattenStateChart`. Diese wird auf einem Objekt `f1` vom Type `StateChartFlattener` aufgerufen, welches zuvor angelegt wurde und durch einen `stateChart` Link mit dem zu konvertierenden Statechart verbunden wurde. Das Anlegen dieses Objekts und des Links wird in Fujaba durch die Farbe grün und den Stereotyp ≪`create`≫ dargestellt. Analog werden zu löschende Objekte und Links rot mit dem Stereotyp ≪`destroy`≫ hervorgehoben.

Falls eine große Objektstruktur benötigt wird, um ein Szenario zu beschreiben, ist es möglich, in den einzelnen Schritten lediglich einen Ausschnitt aus dieser Objektstruktur zu zeigen, der nur die für diesen Schritt relevanten Objekte enthält. Für den Szenarioschritt uninteressante Objekte können einfach weggelassen werden, bisher noch nicht betrachtete können hinzugenommen werden. Man beachte, dass das Weggelassenen von Objekte keineswegs deren Löschung bedeutet, Löschen wird immer explizit durch ≪`destroy`≫ modelliert. Bei Objekten die in einem Schritt neu hinzugenommen werden, wird hinter dem Objektnamen der Typ angegeben. Falls ein Objekt schon aus einem vorigen Schritt bekannt ist, kann dieser weggelassen werden. Daher stehen hinter den Objekten im zweiten Schritt aus Abbildung 5.3 keine Typbezeichner mehr.

In dem ersten Ausführungsschritt, der dritten Aktivität in Abbildung 5.3, wird nun die eigentliche Konvertierung zum endlichen Automaten durchgeführt. Hierzu wird zuerst die eingehende Transition des Or-States auf den Startzustand des eingeschachtelten States umgelenkt. Im Objektdiagram wird also die `target` Kante zwischen dem Transitionsobjekt `t1` und dem Or-State `o1` gelöscht und ein neuer `target` Link von `t1` zum inneren State `s2` erzeugt.

Im nächsten Ausführungsschritt (der ersten Aktivität in Abbildung 5.4) werden nun die ausgehenden Transitionen behandelt. Dazu wird das Transitionsobjekt `t2` gelöscht und zwei neue `t2_1` und `t2_2` werden angelegt. Die neuen Transitionen erhalten das gleiche Label wie `t2` und das selbe `target s1`. Die neuen Transitionen starten bei jeweils einem der inneren States.

Im letzten Ausführungsschritt kann der nun isolierte Or-State gelöscht werden. Die letzte Aktivität in Abbildung 5.4 beschreibt nun die Endsituation. Das Statechart ist erfolgreich in einen endlichen Automaten konvertiert worden.

In diesem Kapitel wurde nur ein Storyboard für den Usecase Flattening gezeigt. Normalerweise existieren aber viele Szenarien zu einem Usecase, die jeweils unterschiedliche Abläufe, Sonder- und Fehlerfälle beschreiben. Da Storyboards auf Aktivitätsdiagrammen aufbauen, könnte man hier auch Fallunterscheidungen und Schleifen modellieren um mehrere Fälle in einem Storyboard unterzubringen. In anderen Ansätzen wird oft der Einsatz von Aktivitätsdiagrammen zur Modellierung von Szenarien vorgeschlagen, siehe beispielsweise [Bal05]. Wir haben jedoch die Erfahrung gemacht, dass es sehr viel einfacher ist, sich auf einen sequentiellen Beispieldurchlauf zu konzentrieren, da der Entwickler so nur einen konkreten Fall zur Zeit und nicht alle möglichen Fälle auf einmal berücksichtigen muss. Dies stellt eine enorme Komplexitätsreduktion dar. Insbesondere Anfängern wird durch ein solches Vorgehen die Modellierung und das Verständnis von formalen Anforderungen erleichtert. Kunden und Neueinsteiger können sich also im Projekt sehr viel leichter mit den Szenarien auseinandersetzen. Der FUP schlägt somit die Verwendung von mehreren einfachen Alternativszenarien anstelle von wenigen komplexen Aktivitätsdiagrammen vor.

Das Objektspiel und die Storyboardingphase sind inhaltlich recht ähnlich. Das Objektspiel ist aber freier im Umgang mit den Objektdiagrammen und eignet sich besser für Teamdiskussionen. Da die Storyboards mit dem Fujaba Werkzeug erstellt werden ist hier ein formaler Umgang mit Objektdiagrammen zwingend erforderlich. Insbesondere erfordert Fujaba für jedes Element die Angabe eines zugehörigen Typs. Hier können neue Typen angelegt oder bereits bestehende wiederverwendet werden. Auf diese Weise entsteht bereits ein initiales Klassendiagramm des zu entwickelnden Systems. Wie dieses Klassendiagramm verfeinert und komplettiert wird, ist in Abschnitt 5.4 zu lesen.

Die Qualität eines Storyboards ist, da es normalerweise nicht ausführbar ist, erstmal schlecht messbar. Die Fragen „Wann ist mein Storyboard vollständig?“, „Wann ist es gut?“, „Hat es den richtigen Detailgrad?“ sind also nicht so leicht zu beantworten. Hauptziel eines Storydiagramms ist es ja, ein Szenario möglichst gut auf Modellebene zu beschreiben. Um also trotzdem ein Gefühl für die Güte eines Storyboards zu bekommen, empfehlen wir dieses anderen Teammitgliedern oder Domänenexperten

vorzulegen. Das kann durch Präsentationen oder Reviewverfahren passieren. Stellen, an denen Fragen oder Verständnisschwierigkeiten auftauchen sind meist auch die Stellen, an denen noch Probleme im Modell existieren oder an denen der Detailgrad des Storyboards noch nicht ausreichend ist. Durch einen solchen Austausch der Storyboards lassen sich also gut Schwächen hierin herausarbeiten.

In unseren Projekten entwickeln wir die ersten Objektspiele und Storyboards im Team. Danach hat jedes Teammitglied eine detaillierte Vorstellung der Datenstruktur und der Ablauflogik des zu entwickelnden Systems. Weiterhin einigt man sich hiermit schon auf ein erstes grobes Klassendiagramm, das als Grundlage für das weitere Vorgehen dient. Jetzt kann jedes Teammitglied unabhängig von den anderen eine Teilmenge der Usecases bearbeiten. Hierbei wird natürlich das iterative, agile Vorgehen des FUP verwendet. Allerdings kann hierbei, da ja keine Gruppendiskussion geführt werden muss, das Objektspiel von erfahrenen Entwicklern ausgelassen werden. Das Fujaba Werkzeug erleichtert dieses verteilte Arbeiten durch einen ausgereiften Persistenzmechanismus für den Mehrbenutzerbetrieb, vgl. [SZN04]. Dieser Persitenzmechanismus erlaubt insbesondere auch das Mischen von Änderungen beispielsweise am Klassendiagramm. Auf diese Weise wird das Klassendiagramm zu einem teamübergreifenden Verzeichnis der in den Szenarien verwendeten Typen.

Nach unseren Erfahrungen hat sich das Objektspiel und die Storyboardingphase als exzellentes Werkzeug zur Verfeinerung textueller Anforderung und zur Diskussion von Designfragen herausgestellt. Durch die Verwendung von Icons und geeigneten Abstraktionen werden die Objektdiagramme auch für Domänenexperten und Kunden leicht verständlich. In unseren Projekten funktionierte das sehr gut, sogar mit Kunden, die wenig bis keine Erfahrung im Programmieren und Modellieren hatten. Für Kunden und Domänenexperten sind solche Szenarien unseren Erfahrungen nach wesentlich einfacher zu verstehen als Ablaufbeschreibungen durch beispielsweise Statecharts oder komplexe Aktivitätsdiagramme. Ein weiterer Gewinn durch die Storyboards entsteht durch die Möglichkeit aus ihnen automatische Tests generieren zu lassen. Diese Testgenerierung wird in Abschnitt 5.5 genauer beschrieben. Zu Beginn eines Projekts, oder wenn wichtige Fragen mit den Kunden diskutiert werden müssen, werden meist sehr detaillierte Storyboards entwickelt. Erfahrenere Entwickler neigen dazu eher grobgranularere Storyboards zu erstellen und sich auf die komplexen Szenarien zu beschränken. Solche Entwickler lassen oft einfache Szenarien aus oder fassen mehrere Schritte eines Szenarios zu einem zusammen, da die Zwischenschritte für sie klar sind. Nichts desto trotz stellen Objektspiel und Storyboard auch für den erfahrenen Entwickler einen Gewinn dar. Wie bereits erwähnt, sind sie ein gutes Mittel, um Designdiskussionen zu führen, und zusätzlich dienen sie zur Testgenerierung. Wir benutzen Objektspiel und Storyboards jeden Tag in unserer Arbeitsgruppe.

5.4. Ableiten des Klassendiagramms

Nachdem ein Problem durch das Objektspiel analysiert und als Storyboard protokolliert und verfeinert wurde, ist der nächste Schritt nun das systematische Ableiten des

Klassendiagramms. Nach unseren Erfahrungen werden viele wichtige Designentscheidungen meist schon während der Storyboarding Phase getroffen. Zusätzlich wird der Entwickler während des Storyboardings dazu aufgefordert Typen für Objekte und Links anzugeben. Hierbei kann der Entwickler entweder einen existierenden Typ auswählen oder einen neuen anlegen. Diese neuen Klassen und Assoziationen werden in einem ersten konzeptionellen Klassendiagramm gesammelt, siehe Abbildung 5.5.

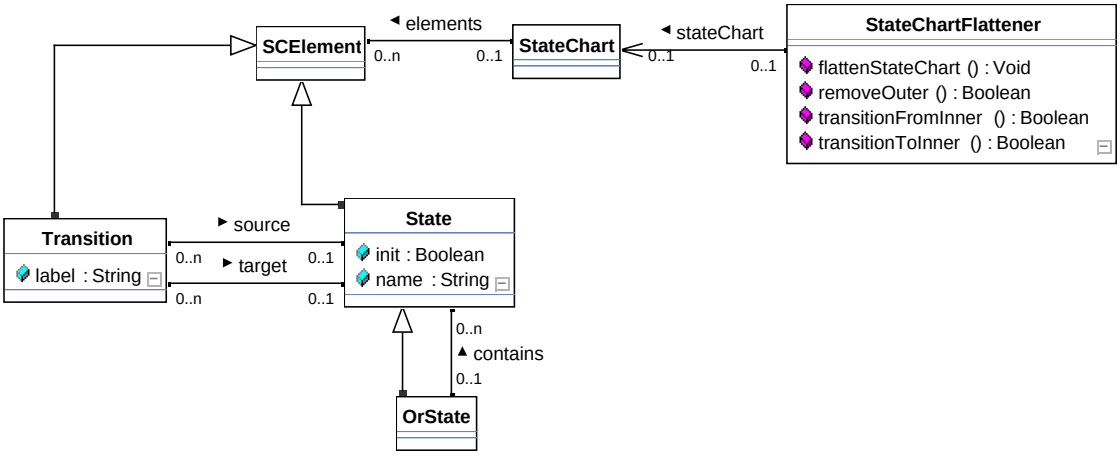


Abbildung 5.5.: Klassendiagramm in Fujaba

Natürlich muss ein während der Storyboarding Phase implizit erstelltes Klassendiagramm nachträglich noch überarbeitet werden. Beispielsweise enthält ein solches Klassendiagramm noch keine Vererbungshierarchie. Der Entwickler sollte gegebenenfalls also gemeinsame Attribute und Methoden verschiedener Klassen durch Refactorings in Oberklassen zusammenfassen. Weiterhin müssen meist Kardinalitäten, Attributtypen und Pakete angepasst werden. Design Pattern, wie das Composite Pattern oder das Observer Pattern können in dieser Phase auch eingebaut werden. Allerdings werden viele dieser Pattern schon auf Objektdiagrammebene sichtbar. Wenn man beispielsweise ein Delegation Pattern nutzt oder ein Proxyobjekt verwendet, wird dieses meist schon im Storyboard auftauchen und das Klassendiagramm muss anschließend nur noch entsprechend refaktoriert werden.

Zusammenfassend kann man also sagen, dass die meisten Entscheidungen, die die Datenstruktur betreffen, schon in den Objektdiagrammen getroffen werden. Das implizit während der Storyboardingphase erstellte Klassendiagramm enthält also schon die meisten für die Datenstruktur wichtigen Informationen. Eine Ausnahme bilden allerdings die Kardinalitäten. Aber auch hier kann man argumentieren, dass aus den Objektdiagrammen meist zumindest hervorgeht, ob ein Objekt eines Typs immer nur ein Nachbarobjekt mit einem bestimmten Kantentyp erreicht (zu-1) oder ob auch mehrere solche Nachbarn in den Szenarien vorkommen (zu-n). Die Hauptaufgabe dieser Phase ist also das Verbessern der Programmstruktur, wie es beispielsweise durch Strukturierung in Pakete oder durch Vermeidung von dupliziertem Code mittels Vererbung geschieht.

Die Storyboards sowie das zugehörige Klassendiagramm können in frühen Analysephasen auf einem eher abstrakten, plattformunabhängigen Niveau modelliert werden.

Später können sowohl Storyboard als auch Klassendiagramm durch plattformspezifische Details, wie Proxies oder Fassaden, verfeinert werden.

Wie steht es nun aber mit der Qualität dieser implizit generierten Klassendiagramme? Viele Probleme und die damit verbundenen Designentscheidungen werden bereits auf Objektdiagrammebene diskutiert. Sobald eine Designentscheidung auf Objektdiagrammebene getroffen ist, ergibt sich die zugehörige Struktur des Klassendiagramms meist automatisch. Daher hängt es also vom Problem ab, ob eine Diskussion auf Objektdiagramm- oder auf Klassendiagrammebene sinnvoller ist. Für Szenariobasierte Fragen eignen sich die Objektdiagramme sicher gut, für strukturelle Fragen, wie die Einführung von Vererbungsbeziehungen, bieten sich eher Klassendiagramme an. Es werden also tatsächlich beide Diagrammarten benötigt. Wir konnten jedoch eine eindeutige Qualitätsverbesserung der Klassendiagramme unserer Studenten feststellen seit wir Storydiagramme in den Entwicklungsprozess, den die Studenten in unseren Vorlesungen praktizieren, eingebaut haben, siehe [DGZ03a].

5.5. Ableiten von automatischen JUnit Tests

Im nächsten Schritt werden JUnit Tests generiert. Fujaba bietet hierzu ein Kommando an, welches ein Storyboard automatisch in einen einfachen JUnit Test übersetzt, vgl. [GZ03, GZ05]. Im wesentlichen besteht ein so generierter Test aus drei Teilen. Der erste Teil wird aus der Startsituation des Szenarios abgeleitet. Dieser Teil legt zur Ausführungszeit eine Objektstruktur an, wie sie in der Startsituation modelliert ist. Danach folgt üblicherweise ein Schritt, der vom Aktor des zugehörigen Usecase initiiert wird und die Ausführung des Szenarios anstößt. Die vom Aktor ausgeführten Schritte werden in die Tests übernommen. Ein solcher Test führt also nach dem Anlegen der Startsituation alle Aktoraktionen aus. Der letzte Teil des Tests vergleicht die Laufzeitobjektstruktur am Ende des Tests mit der Objektstruktur, die in der Endsituation des Szenarios spezifiziert wurde. Stimmen diese nicht überein, wird ein Fehler an das JUnit Framework gemeldet. Abbildung 5.7 zeigt einen erfolgreichen Durchlauf des Tests, der für das beschriebene Szenario des Usecase Flattening generiert wurde. Die Testgenerierung wird ausführlich in Teil II beschrieben.

5.6. Ableiten des Verhaltens der Methoden

Im nächsten Schritt muss das Verhalten implementiert werden. Während in den vorhergehenden Phasen vorwiegend die Datenstruktur betrachtet wurde, rücken nun Details des Algorithmus in den Vordergrund.

In diesem Schritt muss vom Verhalten in ausgewählten Beispielen (den Szenarien) auf das allgemeine Verhalten geschlossen werden. Es muss also ein Algorithmus entwickelt werden, der in allen möglichen Fällen, das gewünschte Ergebnis liefert. Dieses stellt erneut eine Komplexitätshürde dar und ist insbesondere für Anfänger nicht einfach. Ein Mischen der Szenarien wie in [DGMZ02, DGZ02, DGZ05c] beschrieben,

bietet eine zusätzliche Hilfestellung, konnte die anfänglichen Probleme unserer Studenten bei der Verhaltensmodellierung jedoch nicht zufriedenstellend lösen. Unsere Beobachtungen zeigen, dass die Ursache hierfür die Tatsache ist, dass die Sicht der Szenarien auf das vorliegende Problem eine sehr datenorientierte ist. Außerdem behandelt ein Szenario immer nur einen konkreten Beispielablauf. Einen geeigneten Algorithmus findet man also nur durch Vergleichen mehrerer Szenarien und dem Ableiten von Kontrollstrukturen wie Fallunterscheidungen und Schleifen hieraus. Es wäre also wünschenswert, für das Entwerfen des Verhaltens ähnliche komplexitätsreduzierende Hilfestellungen anzubieten, wie das Objektspiel und die Szenarien für die Datenmodellierung darstellen. Der FUP schlägt hier den sogenannten Dreisprung-Ansatz vor (siehe [Die07] und [DGZ08]).

Der Dreisprung-Ansatz zur Entwicklung des Verhaltensmodells besteht aus folgenden drei „Sprüngen“:

Sprung 1: Textuelle Verhaltensmodellierung auf Applikationsebene
Im ersten Schritt wird das generelle Verhalten für den betrachteten Usecase auf Applikationsebene beschrieben. Da im FUP meist Methoden für das Anstoßen eines Szenarios und somit auch eines bestimmten Verhaltens verantwortlich sind, werden hier also in der Regel die Effekte einer oder mehrerer Methoden beschrieben. Der folgende Auszug beschreibt den ersten Schritt beim Konvertieren eines Statecharts in einen endlichen Automaten:

> Für jeden Or-State des Statecharts werden alle eingehenden Transitionen auf den inneren Startzustand umgeleitet. ...

Diese Beschreibung ist, wie schon erwähnt, auf Applikationsebene formuliert, also in der Sprache der Kunden und Domänenexperten. In dieser ersten konzeptionellen Phase beschäftigt man sich also noch nicht mit dem konkreten Datenmodell sondern konzentriert sich auf den Algorithmus. Das reduziert die Komplexität, da man sich noch keine Gedanken machen muss, wie der Algorithmus auf das konkrete Datenmodell anzuwenden ist. Zusätzlich bietet es eine gute Grundlage um Details des Algorithmus mit dem Team, mit Kunden oder mit Domänenexperten zu diskutieren. Die Beschreibungen enthalten, falls vorhanden, Vorbedingungen der Methode sowie Schleifen, Fallunterscheidungen und Aufrufe von Verhalten, das an anderer Stelle beschrieben ist. Ein solcher Aufruf wird später meist als Methodenaufruf interpretiert.

Sprung 2: Textuelle Verhaltensmodellierung auf Modellebene
Im zweiten Schritt muss jetzt der entwickelte Algorithmus so angepasst werden, dass er auf dem bestehenden Datenmodell arbeitet. Dafür wird die Verhaltensbeschreibung des ersten Schrittes so umformuliert, dass die Begriffe des Modells benutzt werden. Für das Statechart Beispiel sähe das so aus:

> Über den `elements` Link des Statecharts wird ein Objekt `or` vom Typ `OrState` gesucht. Wird eine Transition gefunden, die einen `target` Link zu `or` hat, wird

dieser `target` Link gelöscht und ein neuer zum inneren Startzustand gezogen. Der innere Startzustand ist das Objekt, das von `or` über den `contains` Link zu erreichen ist und dessen `initial` Attribut den Wert `true` hat. ...

Hier wird also darauf eingegangen, in Objekten welcher Klasse Informationen gespeichert und gesucht werden, welche Kanten zum Suchen benutzt werden und in welchen Attributen der Zustand des Systems wie abgelegt wird. Hier hilft es einen Blick in die Szenarien zu werfen, da diese Fragestellungen aus den Storyboards meist gut abzuleiten sind. Zusätzlich zu den in den zugehörigen Storyboards modellierten Objekten können hier aber, abhängig vom gewählten Algorithmus, noch weitere lokale Variablen (z.B. Zählvariablen bei Schleifen) oder sogar Hilfsobjekte nötig sein. Diese sind dann natürlich auch im Klassendiagramm nachzutragen.

Das obige Beispiel beschreibt die Effekte der Methode `flattenStateChart()` der Klasse `StateChartFlattener`, wie sie in dem Beispielszenario dieses Kapitels in der letzten Aktivität in Abbildung 5.3 sowie in den ersten beiden Aktivitäten in Abbildung 5.4 zu sehen sind. In diesem Fall wurde entschieden jeweils separate Methoden für das Umsetzen der eingehenden Transitionen, das Erzeugen der ausgehenden Transitionen und das Löschen des Or-States zu erstellen. Zur Behandlung der eingehenden Transitionen ist im Storyboard die letzte Aktivität aus Abbildung 5.3 zuständig.

Sprung 3: Graphische Verhaltensmodellierung mit Storydiagrammen
Im letzten Schritt muss jetzt die textuelle Beschreibung auf Modellebene in eine fertige Implementierung umgesetzt werden. Aufgrund der Zwischenschritte, ist die Komplexität bei diesem Schritt nach unseren Erfahrungen eher gering. Um die Komplexität noch weiter zu minimieren, benutzen wir eine graphische Programmiersprache, in der sich Suchen und Änderungsoperationen ebenfalls auf Modellebene beschreiben lassen. Das Ergebnis des Dreisprungs ist also üblicherweise eine graphische Verhaltensspezifikation. Wir verwenden spezielle UML Interaktionsdiagramme, sogenannte Storydiagramme, siehe Abbildung 5.6. Fujaba kann nun aus solchen Diagrammen ausführbaren Java Quelltext generieren. Alternativ ist es allerdings auch möglich das Verhalten der Methode von Hand in Java zu implementieren. Der FUP ist somit auch einsetzbar, wenn auf den Einsatz von Storydiagrammen verzichtet wird, weil z.B. keine Codegenerierung für die gewünschte Zielsprache vorliegt.

Formal spezifiziert Abbildung 5.6 eine Graphersetzungsregel. Diese Graphersetzungsregel sucht ausgehend vom aktuellen `StateChartFlattener` Objekt `this` nach einem Statechart `sc`, das einen Or-State `or` enthält, welcher eine eingehende Transition `aToOr` hat. Zusätzlich wird nach einem Substate `inner` des Or-States gesucht, bei welchem das `init` Attribut mit dem Wert `true` belegt ist. Dieser State ist der Startzustand des inneren Statecharts. Wird eine solche Belegung gefunden, wird der `target` Link zwischen dem Objekt `aToOr` und `or` gelöscht (erkennbar an dem ≪`destroy`≫ Stereotyp) und ein neuer Link zwischen den Objekten `aToOr` und `inner` angelegt (≪`create`≫ Stereotyp). Wenn diese Transformation erfolgreich durchgeführt wurde, wird die Aktivität mit der `success` Kante verlassen, das heißt, es wird `true` an die aufrufende Methode zurückgeliefert. Konnte das spezifizierte Graphpat-

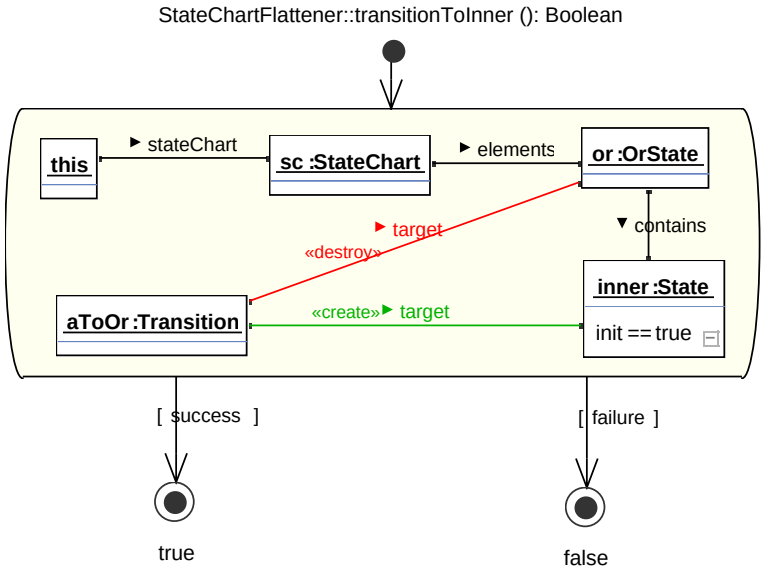


Abbildung 5.6.: Storydiagramm für das Umleiten eingehender Transitionen

tern nicht gefunden werden, wird die `failure` Kante benutzt und `false` zurückgeliefert. Die aufrufende Methode ist so implementiert, dass sie `transitionToInner` solange ausführt, wie `true` zurückgeliefert wird. So können alle zu Or-States führenden Transitionen zu deren inneren Startzustand umgeleitet werden. Man beachte, dass in Abbildung 5.5 die Klasse `OrState` von der Klasse `State` erbt. Das bedeutet, dass immer wenn nach einem Objekt der Klasse `State` gesucht wird, auch ein Objekt der Klasse `OrState` ersatzweise verwendet werden kann. Für die Graphersetzungsregel bedeutet das, dass das Objekt `inner` entweder ein normaler State oder ein Or-State sein kann. Daher funktioniert die Regel auch für geschachtelte Or-States.

Unseren ersten Erfahrungen nach hilft der Dreisprung Ansatz unseren Studenten ungemein bei der Entwicklung ihrer ersten Methoden. Mit diesem Ansatz können sie jede Hürde einzeln überwinden: Erst entwickeln sie einen geeigneten Kontrollfluss, dann passen sie die Such- und Änderungsoperationen an das Datenmodell an und zum Schluss formalisieren sie ihren Algorithmus in einer (graphischen) Programmiersprache. Nach jedem Schritt können die Ergebnisse gut in der Gruppe diskutiert werden. Das hilft beim Lernen, bietet aber auch erfahrenen Entwicklern die Möglichkeit Details bei der Implementierung mit dem Kunden oder mit Domänenexperten zu diskutieren. Ansonsten können einzelne Schritte des Dreisprungs von erfahreneren Entwicklern natürlich auch zusammengefasst oder übersprungen werden.

Die vorangegangenen Schritte dienen vor allem dazu ein geeignetes Datenmodell zu erstellen. Dieses Datenmodell stellt eine Abbildung der Applikationsdomäne in die objektorientierte Datenwelt dar. Das Modell soll dabei auch für Domänenexperten noch leicht verständlich sein. Das Datenmodell soll aber noch eine zweite Voraussetzung erfüllen: Die Modellierung des Verhaltens sollte durch ein geeignetes Datenmodell möglichst gut umsetzbar sein, das heißt dem Entwickler sollten bei der eigentlichen Implementierung keine unnötigen Hürden entstehen. Es kann also durchaus passieren, dass während dieser Phase noch Änderungen an der Datenstruktur

nötig werden. Dann empfiehlt es sich allerdings auch die Storyboards entsprechend anzupassen.

In der Implementierung des Verhaltens berücksichtigen die Entwickler oft Fälle, die bisher in keinem existierenden Szenario enthalten sind. Meist ist es dennoch eine gute Idee, solche zusätzlichen Fälle auch mit Domänenexperten und Kunden durchzusprechen. Also sollten für solche Fälle zusätzliche Szenarien und Storyboards angelegt werden. Um Funktionalität zu finden, die noch nicht von einem Szenario abgedeckt wird, bietet es sich an, Coverageanalysen während der Ausführung der generierten JUnit Tests zu erstellen. Eine Methode zur modellbasierten Coverageanalyse wird in Abschnitt 12.6 vorgestellt.

Das Erstellen von Storydiagrammen ähnelt in vielen Punkten dem traditionellen Programmieren in einer textuellen Programmiersprache. Trotzdem liegen Storydiagramme, unserer Meinung nach, auf einer höheren Abstraktionsebene, da Such- und Änderungsoperationen auf Modellebene notiert sind. Zusätzlich erleichtert die notationelle Ähnlichkeit der Storydiagramme zu den Storyboards der Szenarien die Erstellung sowie das Verständnis der Methodenspezifikationen. In den Projekten, die am Lehrstuhl durchgeführt wurden, war es für die beteiligten Domänenexperten immer möglich nach einer kurzen Einarbeitungszeit die von uns erstellten Storydiagramme zu verstehen und mit uns darüber zu diskutieren. Einige brachten sogar Verbesserungsvorschläge an den Diagrammen ein oder modellierten selbst kleinere Methoden (siehe [ZLM+06]).

5.7. Codegenerierung mit Fujaba

Der nächste Schritt ist das Generieren des Quellcodes. Fujaba ist in der Lage aus den Klassendiagrammen sowie aus den Storydiagrammen ausführbaren Java Code zu generieren, vgl. [FNTZ98,KNNZ00]. Zusätzlich wird zu den aus den Storyboards generierten Tests JUnit kompatibler Code generiert. Wenn alle Methoden mit Storydiagrammen modelliert wurden, kann also die komplette Applikationslogik inklusive Tests generiert werden. Manuelles Programmieren ist höchstens noch zur Erstellung der graphischen Benutzerschnittstelle nötig. In Kapitel 6 wird die Funktionsweise der in dieser Arbeit verwendeten Codegenerierung genauer erläutert.

Fujaba generiert für eine Seite Storydiagramm ungefähr 5 bis 10 Seiten Java Code. Handgeschriebener Code wäre vermutlich etwas kürzer als der generierte, da im generierten Code zahlreiche Sicherheitsabfragen stecken, die Entwickler im eigenen Code häufig weglassen. Dennoch sehen wir die Tatsache, das Storydiagramme „kürzer“ sind als der äquivalente Code, als Hinweis darauf an, dass Storydiagramme auf einer höheren Abstraktionsebene liegen. Zusätzlich haben wir die Beobachtung gemacht, dass eine Seite Storydiagramm einfacher zu lesen und zu verstehen ist als mehrere Seiten Java Quelltext.

5.8. Automatische Validierung der Szenarien

Direkt nach der Storyboardingphase können JUnit Tests generiert und ausgeführt werden. In dieser frühen Phase *sollte* der Test normalerweise noch fehlschlagen, da die zugehörige Funktionalität ja noch nicht implementiert wurde. Diese so generierten Tests stellen somit einen Gradmesser da, wie weit die Funktionalität schon implementiert wurde. Ein erfolgreich durchlaufender Test bedeutet hier also, dass das zugehörige Szenario von der Implementierung bereits abgedeckt ist. Abbildung 5.7 zeigt den erfolgreichen Test für den Usecase Flattening.

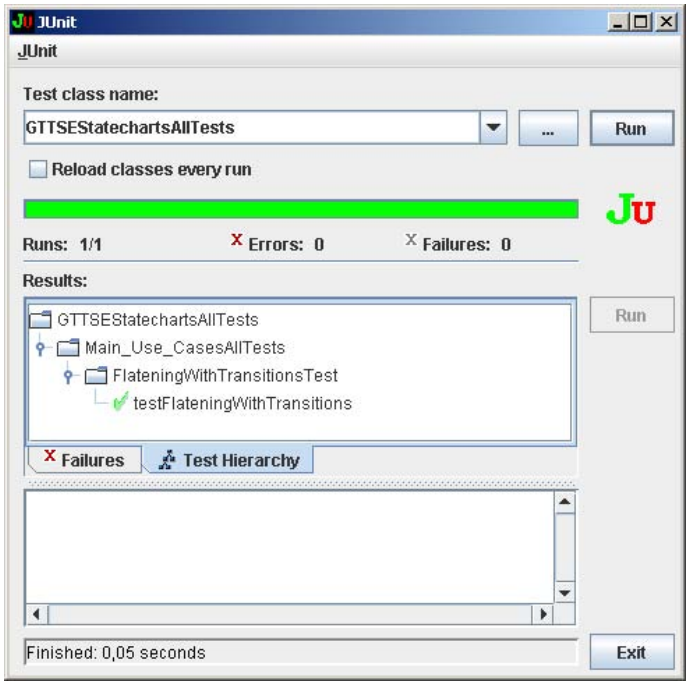


Abbildung 5.7.: Ausführung des JUnit Test für die Statechartkonvertierung

Da die generierten Tests Auskunft über bereits implementierte Funktionalität geben, lassen sie sich auch zu Projektmanagementzwecken gebrauchen. Beispielsweise lässt sich durch die Anzahl der bereits durchlaufenden und der noch fehlschlagenden Tests eines Usecases auf den Fortschritt bei der Implementierung dieses Usecases schließen. Da die JUnit Tests automatisiert durchführbar sind, das heißt keine weitere Benutzerinteraktion benötigen, ließe sich sogar eine automatische Fortschrittsanzeige realisieren. Diese ließe sich auch zur automatischen Zeiterfassung benutzen um beispielsweise Statistiken anzulegen, wie lange die Implementierung einzelner Usecases jeweils gedauert hat.

Zu beachten ist, dass die generierten Tests nur überprüfen, ob die modellierten Szenarien von der Implementierung abgedeckt sind. Sie stellen also keine generelle, alles umfassende Testmethode dar. Der Entwickler sollte noch zusätzliche Tests bereitstellen, die beispielsweise kritische Methoden durch systematische Variation der Parameter ausgiebig testen. Nur so kann eine komplette Testüberdeckung gewährleistet werden.

Da die generierten Test aber die in den Szenarien modellierten Funktionalitäten überprüfen, eignen sich die Tests, wie im XP vorgeschlagen [BA04], gut, um bei Umbauten, Wartungsarbeiten oder Refactorings den Erhalt der bereits implementierten Funktionalität sicherzustellen. Durch die Tests erhält der Entwickler sofort Feedback, wenn er bei Umbaumaßnahmen Fehler in anderen Teilen der Applikation eingebaut hat. Dieser Mehrwert der Szenarien ist unseren Erfahrungen nach eine hohe Motivation für die Entwickler, viele, qualitativ hochwertige Szenarien zu erstellen und diese auch über die gesamte Projektlaufzeit aktuell zu halten.

Auch garantieren die Test eine gewisse Konsistenz zwischen Anforderungsdefinition und Implementierung. Sobald der Programmierer größere Umbauten am Modell oder am Verhalten gemacht hat, muss er gegebenenfalls auch die Szenarien anpassen, da sonst die Tests fehlschlagen oder sogar nicht mehr gegen das angepasste Modell kompilierbar sind. Wird dies konsequent durchgehalten, dann passen die Szenarien immer zum aktuellen Stand der Implementierung, das heißt, das Problem des Veralten der Dokumentation gegenüber der Implementierung (siehe Teil II) ist im FUP deutlich kleiner als bei anderen Prozessen. Der Mehraufwand durch die zusätzliche Pflege der Szenarien wird unseren Erfahrungen nach von den Entwicklern akzeptiert, da die Szenarien durch die generierten Tests, wie oben beschrieben, auch während der Implementierung noch einen Mehrwert bieten.

Während der Durchführung der JUnit Test bietet es sich an, die Quellcodeüberdeckung der Tests zu protokollieren. Eine solche Analyse kann Fälle aufdecken, die bei der Implementierung schon berücksichtigt wurden, für die aber noch kein Szenario und somit kein Test vorliegt. Ein solches Vorgehen liefert einen Anhaltspunkt in wie weit das entwickelte System durch die Szenarien abgedeckt ist.

Für das Fehlschlagen eines Tests kann es mehrere Gründe geben: Die Implementierung deckt das getestete Szenario noch nicht vollständig ab, das Szenario ist fehlerhaft oder es liegt ein Fehler in der Implementierung vor. Im letzten Fall muss der Fehler in der Implementierung gefunden und beseitigt werden. Das kann jedoch ein komplexer und zeitaufwendiger Prozess sein. Der FUP bietet daher Werkzeugunterstützung zum Debuggen an. Im Falle eines fehlschlagenden Tests wird die Fehlerursache ausgegeben und automatisch die Objektstruktur nach der Ausführung des Tests visualisiert. Diese kann dann leicht mit der erwarteten Objektstruktur verglichen und so die Ursache des Fehlschlagens ermittelt werden. Zur Darstellung der Objektstruktur wurde im Rahmen dieser Arbeit der Objektbrowser eDOBS [GZ06b] entwickelt. Abbildung 5.8 zeigt den eDOBS bei der Visualisierung einer Objektstruktur des Statechartbeispiels. In Teil III werden der eDOBS sowie weitere bereitgestellte Debugginghilfen näher beleuchtet.

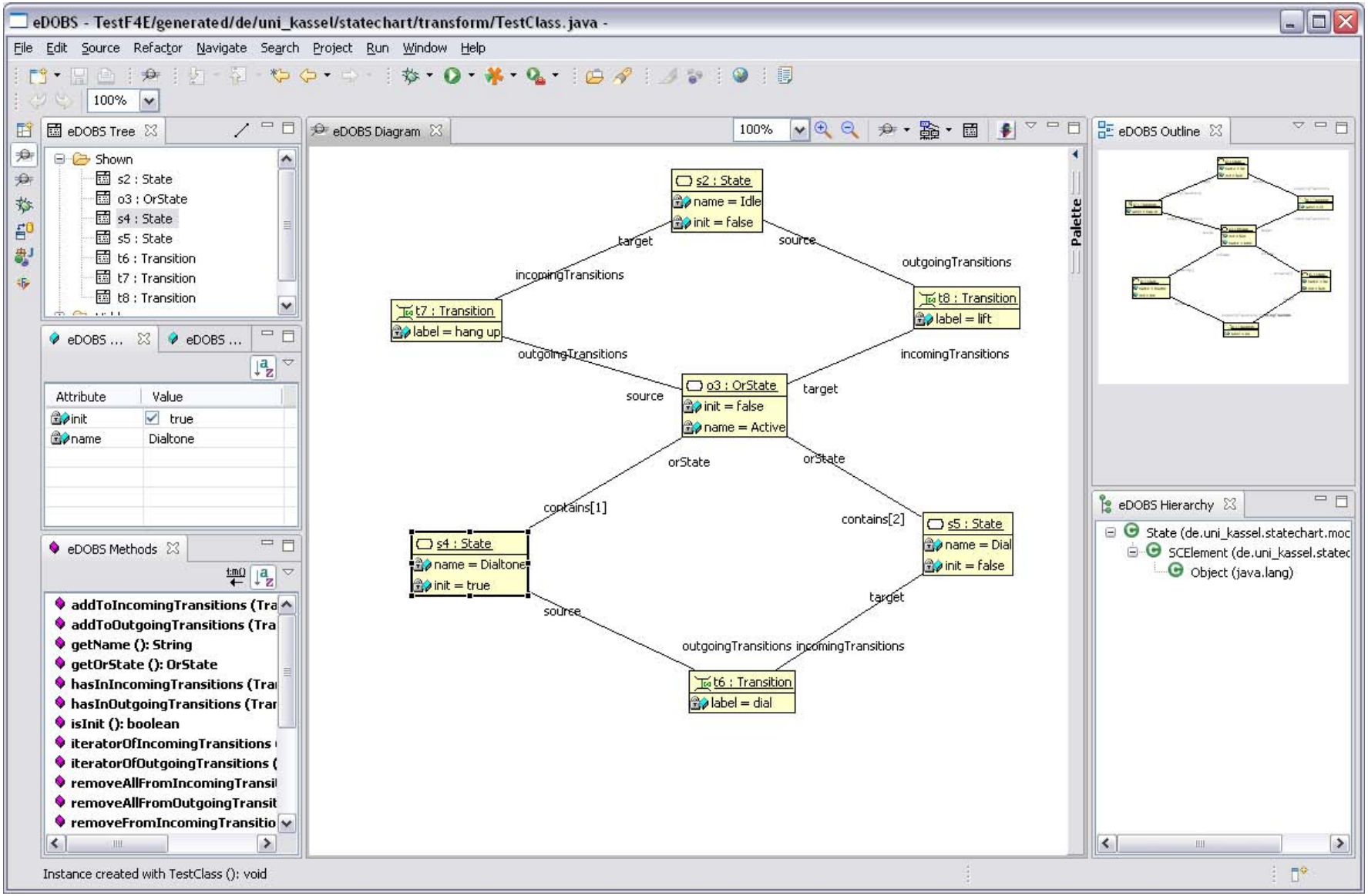


Abbildung 5.8.: Objektstruktur im eDOBS

6. Werkzeugunterstützung

Der beschriebene Fujaba Process erhält durch das Fujaba CASE Tool komplette Werkzeugunterstützung. In diesem Abschnitt wird ein wichtiger Teil dieser Unterstützung, nämlich die Codegenerierung, genauer erläutert.

Im vorgestellten Fujaba Process ermöglicht die Codegenerierung aus den unterschiedlichen Diagrammen, die während des FUPs entstehen, ausführbaren Code zu generieren. Hierbei sorgt eine flexible Codegenerierung dafür, dass der FUP in möglichst vielen Anwendungsfällen inklusive Codegenerierung anwendbar ist. Durch diese Flexibilität ist es möglich, zum Beispiel unterschiedliche Quelltextkonventionen, unterschiedliche Frameworks, wie EMF, oder gar unterschiedliche Sprachen zu unterstützen.

Die ursprüngliche Codegenerierung von Fujaba war komplett in Java Quelltext realisiert. Dadurch war eine Anpassen der Codegenerierung relativ schwierig. Man musste den Quelltext des Generators entsprechend anpassen, Fujaba neu kompilieren und neu ausliefern. Daher wurde in 2005 von Christian Schneider, Carsten Reckord und mir die Codegenerierung von Fujaba einem kompletten Redesign unterzogen. Das entstandene CodeGen2 (vgl. [GSR05, BGSZ08]) ist nun komplett in Fujaba spezifiziert und benutzt Templates zur Codegenerierung. Diese Templates lassen sich zur Laufzeit ändern und diese Änderungen wirken sich sofort auf die Codegenerierung aus. Weiterhin stellt CodeGen2 Mechanismen bereit, die ein einfaches Erweitern der Codegenerierung erlauben. So ist es beispielsweise problemlos möglich die Codegenerierung für neue Modellelemente zu erweitern. Von diesen Erweiterungsmöglichkeiten wird in den nächsten beiden Teilen dieser Arbeit für die Testgenerierung und für das Debugging intensiv Gebrauch gemacht.

In diesem Abschnitt wird die Funktionsweise von CodeGen2 kurz erläutert. Der Codegenerierungsprozess ist bei CodeGen2 in drei Teilabschnitte unterteilt, vgl. Abbildung 6.1.

CodeGen2 übersetzt das Originalmodell zuerst in eine Zwischenschicht aus sogenannten Token. Diese Zwischenschicht definiert die Reihenfolge, in der die Codegenerierung das Modell durchlaufen soll. Allerdings können für ein Modellelement auch mehrere Token angelegt werden. So kann beispielsweise für eine Klasse im Originalmodell ein Token für die Generierung eines Java Interfaces und eins für die Generierung der zugehörigen Implementierungsklasse angelegt werden. Das Ergebnis diesen Schritts ist ein Tokenbaum[1]. Der Tokenbaum wird durch Kinder der Klasse `Token` aufgespannt. Von jedem `Token` lassen sich die Kinder über Links der Assoziation `children` erreichen.

[1] Genauer gesagt handelt es sich um einen Tokengraph, da die Token noch zusätzliche Referenzen auf andere Token enthalten können.

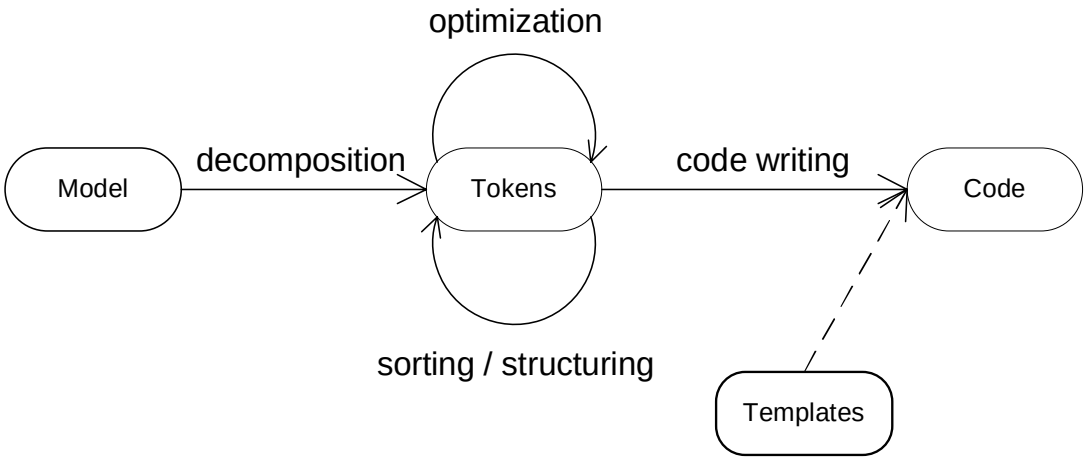


Abbildung 6.1.: Teilabschnitte der Codegenerierung mit eingehenden und resultierenden Daten

Abbildung 6.2 zeigt auf der linken Seite die Objektstruktur eines einfachen Fujaba-Klassendiagramms. Das Klassendiagramm besteht aus zwei Klassen `Teacher` und `Course` vom Typ `Class`. Diese Klassen sind über die Assoziation `gives` verknüpft. Eine Assoziation besteht in Fujaba immer aus einem Objekt für die rechte Rolle, einem Objekt für die Assoziation selbst und einem Objekt für die linke Rolle. Auf der rechten Seite von Abbildung 6.2 ist der zugehörige Tokenbaum abgebildet. Dieser enthält `Token` für das Paket, die Klassen und die Rollen.

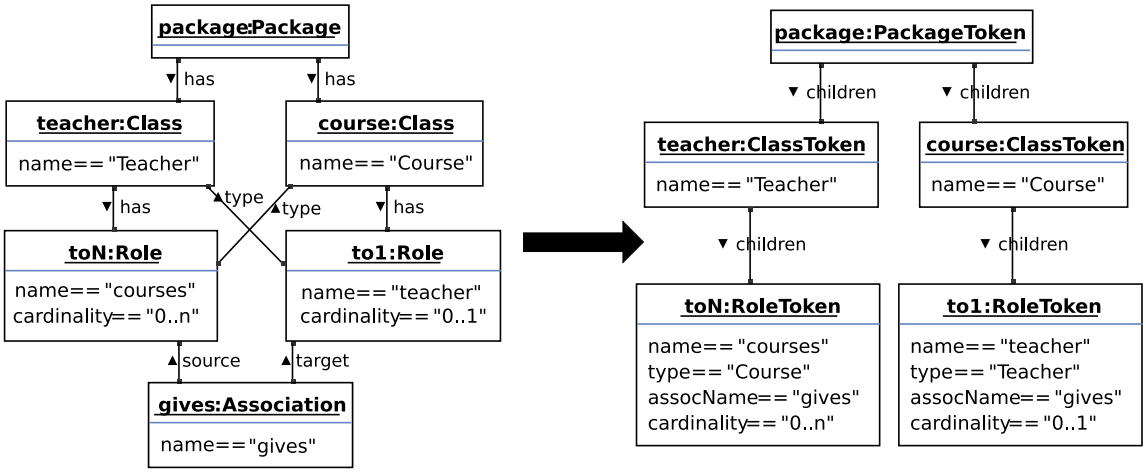


Abbildung 6.2.: Modell mit zugehörigem Tokenbaum

Der generierte Tokenbaum stellt die Zwischensprache der Codegenerierung dar. Auf diesem Baum können jetzt Änderungen erfolgen, wie zusätzliche Strukturierung, Optimierung oder Sortierung der Token. Dies ist insbesondere für die Codegenerierung für Storydiagramme (vgl. Abschnitt 5.6) nötig. Dort müssen beispielsweise Suchpläne für die Suche von Objekten erstellt werden. Token, die Klassendiagrammelemente repräsentieren, erfahren normalerweise wenig bis keine Änderungen in diesem Schritt.

Der letzte Schritt der Codegenerierung erzeugt nun den eigentlichen Code für den umgebauten Tokenbaum. Hierzu wird der Tokenbaum in Postorder besucht. Jedes besuchte Token wird an eine Menge von sogenannten `CodeWritern` gegeben, die in einer Chain of Responsibility (vgl. [GHJV05]) organisiert sind. Der `CodeWriter`, der für dieses Token zuständig ist, öffnet normalerweise eine Templatedatei und übergibt dieses Template, das Token und zusätzliche Informationen an die `TemplateEngine`. Zu den zusätzlichen Informationen gehört unter anderem der Code der für die Kinder des Tokens in der Tokenhierarchie generiert wurde. Die Template Engine erzeugt aus diesen Informationen dann den eigentlichen Quelltext.

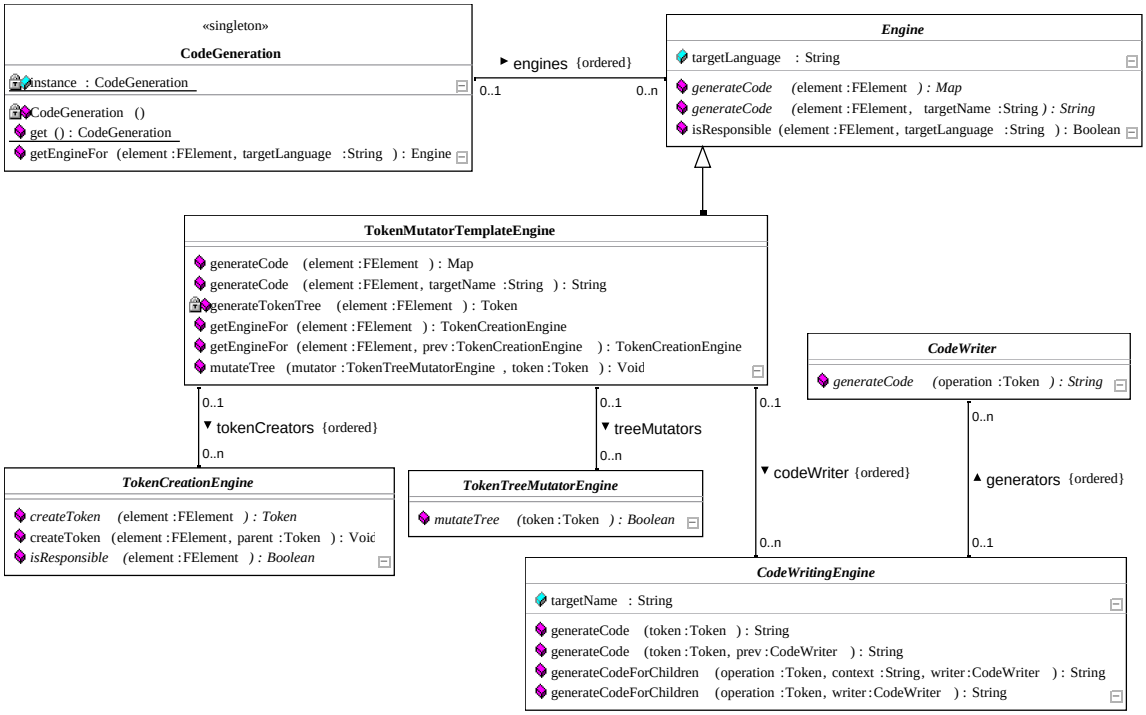


Abbildung 6.3.: Übersichtsklassendiagramm der Codegenerierung

In Abbildung 6.3 ist ein Übersichtsklassendiagramm der Codegenerierung gezeigt. Die Codegenerierung ist über den Singleton `CodeGeneration` erreichbar. Die `CodeGeneration` enthält in der `engines` Assoziation je eine `Engine` für jede unterstützte Sprache (Attribut `targetLanguage`). Bei den hier besprochenen `Engines` handelt es sich um `TokenMutatorTemplateEngines`. Diese Klasse implementiert genau die drei besprochenen Teilschritte: Token erzeugen, Tokenbaum verändern, Templates auf Tokenbaum anwenden. Für jeden dieser Schritte sind unterschiedliche Engines zuständig. Die Tokenerzeugung übernehmen die Objekte der Klasse `TokenCreationEngine`, für Änderungen am Tokenbaum sind die `TokenTreeMutatorEngines` zuständig und das Anwenden der Templates übernehmen die `CodeWritingEngines`. Für jede Dateiart existiert eine `CodeWritingEngine`. So gibt es für C++ Generierung eine `CodeWritingEngine` für C++ Headerdateien (`.h`) und eine `CodeWritingEngine` für C++ Programmdateien (`.cpp`). Jede dieser `CodeWritingEngines` enthält eine der bereits erwähnten Chain of Responsibility von `CodeWritern`.

Man beachte, dass die Codegenerierung viele Punkte bietet, um Erweiterungen hinzuzufügen. Für neue Sprachen kann eine neue `Engine` beim `CodeGeneration` Singleton angemeldet werden. Für neue Modellelemente können Token durch eine neue `TokenCreationEngine`, die an der gewünschten `Engine` angemeldet wird, erzeugt werden. Neue Optimierungsverfahren können durch zusätzliche `TokenTreeMutatorEngines` ergänzt werden. Der eigentliche Quelltext kann außerdem durch zusätzliche `CodeWritingEngines` oder durch Verändern der Templates angepasst werden. Die Templates in CodeGen2 werden im folgenden genauer erklärt, da angepasste Templates unter anderen bei der Testgenerierung in Teil II ein Rolle spielen.

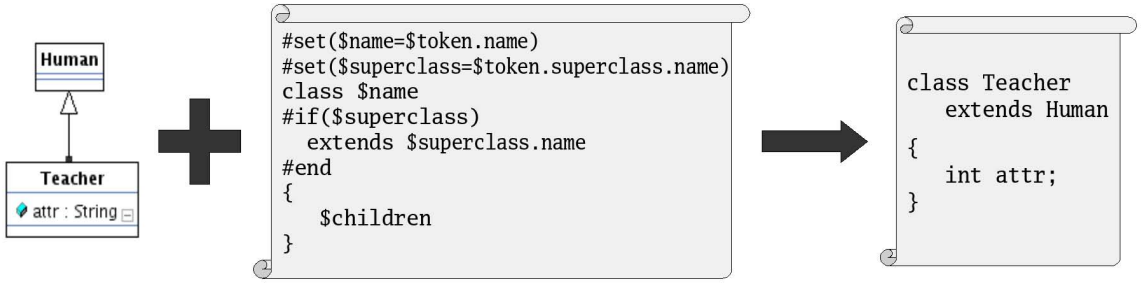


Abbildung 6.4.: Beispiel für die template-basierte Codegenerierung

Unsere Implementierung verwendet die Velocity Template Engine [The09]. Abbildung 6.4 zeigt ein Beispielmodell links, eine vereinfachte Version des Velocity Templates für Klassentoken in der Mitte und rechts den generierten Quelltext. In Velocity beginnen Anweisungen mit dem Doppelkreuz (#) und Variablen mit dem Dollarzeichen ($). Beim Anwenden der Templates wird von CodeGen2 die Variable `$token` mit dem Modellelement belegt, das zum Token gehört, auf welches das Template angewendet wird. In der ersten Zeile des Velocity Templates aus Abbildung 6.4 wird also die Variable `$name` auf das Namensattribut des zugehörigen Klassenobjekts gesetzt. Velocity erlaubt das Auslesen von Attributwerten von Objekten, wie hier für das Objekt in `$token` mit dem Attribut `name` geschehen. Zusätzlich sind auch Methodenaufrufe erlaubt. Diese Möglichkeiten machen die Codegenerierung einfach erweiterbar, da das komplette Modell in den Templates erreichbar ist und ausgelesen werden kann. In der nächsten Zeile wird die Variable `$superclass` auf den Namen der Superklasse gesetzt. Hierzu wird vom Objekt `$token` zuerst das `superclass` Attribut und auf dem Ergebnis dann das `name` Attribut ausgelesen. In Zeile 3 wird schließlich Quelltext erzeugt: Die Zeichenkette "`class `" gefolgt von dem Inhalt der Variable `$name` wird der Ausgabe hinzugefügt. In Zeile 4 wird nun überprüft, ob die Variable `$superclass` belegt ist. Ist dies der Fall, wird die Ausgabe um das Schlüsselwort "`extends `" gefolgt von dem Inhalt von `$superclass` erweitert. Velocity unterstützt also einfache Kontrollstrukturen wie bedingte Anweisungen, aber auch simple Formen von Schleifen. Nach einer öffnenden Klammer in Zeile 7 wird der Inhalt der Variable `$children` eingefügt. CodeGen2 übergibt in dieser Variablen den generierten Quelltext für alle Token, die im Tokenbaum als Kinder des aktuellen Tokens auftauchen. Für ein Klassentoken wäre das also der Quelltext für alle Methoden und Attribute der zugehörigen Klasse. Im Beispiel in Abbildung 6.4

wurde hier der Quelltext für das `attr` Attribut eingefügt. Am Schluss steht noch die schließende Klammer. Der so generierte Quelltext (ganz rechts in der Abbildung) wird schließlich von einem `CodeToFileWriter` in eine Datei geschrieben. Damit ist die Codegenerierung für diese Klasse abgeschlossen.

In diesem Kapitel wurde die neue Codegenerierung von Fujaba CodeGen2 vorgestellt. Diese zeichnet sich insbesondere durch ihre einfache Erweiterbarkeit aus. Durch neue Templatesätze kann CodeGen2 um neue Stile oder Sprachen erweitert werden. Änderungen, die durch Templates nicht möglich sind, können durch den modularen Aufbau von CodeGen2 und die Tatsache, dass CodeGen2 komplett modellbasiert entwickelt ist, einfach und schnell umgesetzt werden. Dies hat zur Folge, dass bereits zahlreiche neue Sprachen für CodeGen2 entstanden sind: So existieren eine Unterstützung für das Eclipse Modellierungsframework EMF, rudimentäre Templates für C++, Codegenerierung für das Webframework GWT und eine Erweiterung für die Java Metadata Interface JMI.

7. Verwandte Arbeiten

Ein weit verbreiteter Softwareentwicklungsprozess ist der Rational Unified Process RUP [JBR99]. Er ist ähnlich dem FUP Usecase-getrieben, iterativ und inkrementell. Die einzelnen Arbeitsschritte[1] sind mit denen des FUP vergleichbar. Im FUP werden allerdings große Teile der Implementierung durch die Storydiagramme eher in der Sprache des Designs formuliert. Weiterhin ist die Phase, in der die Tests modelliert werden, durch die Testgenerierung aus Storyboards im FUP wesentlich früher angesiedelt und fällt zum Teil mit der Anforderungsanalyse zusammen. Der RUP stellt die Architektur in den Mittelpunkt aller Arbeitschritte. Damit unterscheidet er sich vom FUP, bei dem das Klassendiagramm eher als gemeinsames Glossar gesehen wird und somit eine nicht ganz so zentrale Rolle spielt. Der Schwerpunkt im FUP liegt eher auf den Objektdiagrammen, die in den unterschiedlichen Arbeitschritten eingesetzt werden. Zusätzlich konzentriert sich der RUP eher auf Aspekte der Projektorganisation. Er gibt zwar die UML als Modellierungssprache vor und spezifiziert grob Abhängigkeiten der unterschiedlichen Diagramme in den unterschiedlichen Phasen, gibt aber wenig Hilfestellung, wie sich die einzelnen Diagramme ineinander überführen lassen. So sagt der RUP beispielsweise wenig darüber, wie sich aus den Usecase-Szenario-Beschreibungen Interaktionsdiagramme ableiten lassen. Weiterhin ist unklar, wie der Entwickler dann Klassendiagramm und Verhaltensmodell konsistent zu den Szenarien entwickeln kann. Der RUP gibt keine Auskunft, wie sich Tests ableiten lassen. Da der FUP also zu jeder Phase genau angibt, welche Diagramme zu verwenden sind, wie diese mit den Diagrammen der vorherigen und nachfolgenden Phasen in Relation stehen und sogar Hinweise gibt sowie Methodiken vorschlägt, wie sich diese Diagramme aus deren Vorgängern herleiten lassen, halten wir den FUP für einfacher umsetzbar, vor allen für Anfänger und neu zusammengesetzte Teams.

Ein weiterer beliebter Softwareentwicklungsprozess ist das eXtreme Programming (XP) von Kent Beck, vgl. [BA04]. XP ist weniger starr als der RUP und stellt die Team-interne Kommunikation sowie die eigentliche Software in den Mittelpunkt. Auch dieser Entwicklungsprozess ist iterativ, inkrementell und Szenario-basiert. Szenarien werden hier auf sogenannten Storycards festgehalten. Diese Storycards sind vergleichbar mit den textuellen Szenarien im FUP. Allerdings wird im XP nach dieser Phase (dem sogenannten Planning-Game) sofort mit der Implementierung begonnen. Hier werden als erstes, gemäß dem Test-first Prinzip, Tests geschrieben, die die zu entwickelnde Funktionalität überprüfen. Anschließend wird die geforderte Funktionalität programmiert. Der FUP beruht auch auf der *Best Practice* des Test-first, da die Tests ja direkt aus den Szenarien generiert werden. Allerdings ist hier noch kein Programmieren nötig. Bis zur eigentlichen Implementierungsphase gibt es im

[1] Die Arbeitsschritte des RUP sind Geschäftsprozessmodellierung, Anforderungsanalyse, Analyse & Design, Implementierung, Test und Auslieferung.

FUP noch mehrere Zwischenschritte, die die Komplexität der Implementierungsphase reduzieren sollen. XP baut auf dem sogenannten *on-site customer* auf, das heißt ein Kundenvertreter ist immer vor Ort, nimmt an den Diskussionen teil und hilft sogar beim Entwickeln der Tests. Eine häufige Kritik am XP ist die Tatsache, dass es in der Praxis oft schwer ist, einen Kundenvertreter zu bekommen, der die Domäne und die Anforderungen der Software gut kennt und zusätzlich aber auch in der Lage ist, Tests zu programmieren und an Entwicklerdiskussionen teilzunehmen. Der FUP setzt, ähnlich wie XP, auf die Kommunikation im Team sowie auf den Austausch mit Kundenvertretern und Domänenexperten. Diese Diskussionen finden aber im FUP, wie in den vorangegangenen Kapiteln gezeigt, meist auf einer Ebene statt, die auch für einen Kundenvertreter ohne Programmiererfahrung noch verständlich ist. Somit sollte es im FUP wesentlich einfacher sein einen geeigneten *on-site customer* zu bekommen.

Der FUP definiert feingranulare Abhängigkeiten zwischen den verwendeten UML Diagrammen und Diagrammelementen. So ist beispielsweise jedes Storyboard eindeutig einem Usecase zugeordnet. Für diese Abhängigkeiten wird durch die Fujaba Tool Suite Werkzeugunterstützung (z.B. zur Navigation) zur Verfügung gestellt. Diese Abhängigkeiten sind im FUP sehr strikt und gehen somit weit über die UML Definition [BRJ98] und über die Abhängigkeiten, wie sie der RUP vorgibt, hinaus. Da der UML Standard solche feingranularen Relationen zwischen unterschiedlichen Diagrammen nicht vorsieht, werden vergleichbare Abhängigkeiten auch von den meisten UML Werkzeugen, wie z.B. Poseidon, Rational, Rhapsody oder MagicDraw, nicht unterstützt [Gen10,IBM10a,No 10]. Eine Ausnahme bildet lediglich das Gebiet der Anforderungsanalyse. Ein hier viel eingesetztes Werkzeug ist das DOORS Tool, siehe [IBM10b]. Prozesse, die DOORS zum Erfassen der Anforderungen verwenden, benutzen häufig die IDs, die DOORS vergibt, um in späteren Dokumenten auf die zugehörigen Anforderungen zu verweisen, vergleiche [KS06]. Dies ermöglicht es, ähnlich wie im FUP, feingranulare Referenzen zwischen dem Anforderungdokument und nachfolgenden Projektartefakten zu erstellen. Zusätzlich erlauben manche UML Werkzeuge auch Links zwischen Klassen und den vorhandenen UML Interaktionsdiagrammen. Diese Links können dann meist zur Konsistenzprüfung benutzt werden. Eine weitere zum Teil angebotene Interdiagrammrelation ist die zwischen Statecharts und deren Verwendungsstelle. So können beispielsweise Statecharts mit aktiven Klassen verbunden werden, wobei das Statechart dann das nebenläufige Verhalten der Klasse modelliert. Auch können in Diagrammen des sogenannten „Real-Time Object-Oriented Modeling“ von ObjecTime Statecharts mit Komponentenports verbunden werden. Die Statecharts spezifizieren hier das Protokoll zur Benutzung des entsprechenden Ports, siehe [SGW94].

Das Objektspiel ist vergleichbar mit der CRC-Karten Methode von Booch, siehe [Boo90]. Allerdings liefert die CRC Methode lediglich eine Art Klassendiagramm als Resultat. Mit dem Objektspiel entstehen hingegen Sequenzen von Objektdiagrammen die Momentaufnahmen der Objektstruktur während eines Szenarios darstellen. Während einer CRC Session wird genau wie im Objektspiel ein bestimmtes Szenario durchgespielt. Allerdings werden die einzelnen Schritte der Ausführung sowie die Veränderung der Objektstruktur nicht mitprotokoliert. Es werden lediglich

die beteiligten Klassen, Relationen und Methoden festgehalten. Daher geht im CRC Ansatz wertvolle Information verloren, die in nachfolgenden Phasen nicht mehr verwendet werden kann.

Es existieren einige Ansätze, die es ermöglichen (halb-)automatisch aus Sequenzdiagrammen Message Sequence Charts oder Statecharts abzuleiten, siehe zum Beispiel [WS00, JW03, JK01, KGSB99]. In dem von mir mitverfassten Beitrag [DGMZ02] wird ein solcher Ansatz für das Fujaba Werkzeug beschrieben. Unser Ansatz sowie die meisten anderen benutzen Sequenzdiagramme zur Beschreibung von Usecase Szenarien. Hieraus wird dann eine zustandsbasierte Verhaltensbeschreibung in Form eines Statecharts synthetisiert. Diese Ansätze bieten eine weit bessere Automatisierung für das Ableiten von Verhalten aus Szenarien als der im vorigen Kapitel vorgestellte Dreisprung-Ansatz. Jedoch können diese Ansätze nicht mit komplexen, sich über die Zeit verändernden Objektstrukturen umgehen, da sie lediglich Zustandsübergänge berücksichtigen. In vielen Fällen sind die Szenarien aber nicht rein zustandsbasiert sondern beinhalten eben gerade diese dynamischen Objektstrukturen. In solchen Fällen sind Storyboards und Storydiagramme geeignetere Darstellungsformen als Sequenzdiagramme und Statecharts. In größeren Projekten können beide Ansätze benutzt werden, Storyboarding für datenlastige Szenarien und Sequenzdiagramme für zustands- und protokollbasierte Problemstellungen.

Die Catalysis Methode benutzt auch Objektdiagramme als Vor- und Nachbedingung für Usecase Beschreibungen und für Methoden, siehe [DW98]. Einige Ideen und sogar einige Notationen sind ähnlich den hier beschriebenen. Allerdings verwendet Catalysis Vor- und Nachbedingung eher im traditionellen Sinne als algebraische Spezifikationen. Die zugrunde liegende Idee ist die des *desing-by-contract* wie sie unter anderem von Eiffel [ECM06] vorgeschlagen wird. Der Vertrag bildet den Rahmen für die Methodenimplementierung. Es wird jedoch keine Hilfestellung angeboten, wie die Methode nun innerhalb dieses Rahmens zu implementieren ist. Zwischenschritte und algorithmische Aspekte werden per Definition nicht berücksichtigt. Unserer Meinung nach, ist das Modellieren von Vor- und Nachbedingung aber in vielen Fällen nicht ausreichend und auch Zwischenschritte sind durchaus diskussionswürdig. Daher behandelt die Storyboardingphase Zwischenschritte und algorithmische Details explizit. Diese zusätzlichen Details liefern Anhaltspunkte für das spätere Implementieren des Verhaltens. Für das in diesem Kapitel betrachtete Szenario, das die Konvertierung eines Statecharts in einen endlichen Automaten beschreibt, würde die Catalysis Methode lediglich ein Objektdiagramm als Vorbedingung und eins als Nachbedingung vorgeben. Die Vorbedingung wäre ein Objektdiagramm, das ein Statechart darstellt und die Nachbedingung würde den zugehörigen Automaten modellieren. Damit ist also völlig offen, wie die eigentliche Konvertierung durchgeführt werden soll. Der Entwickler hat aber meist auch in frühen Phasen schon eine Idee, wie diese Konvertierung implementiert werden soll. Dieses wichtige Detail würde mit der Catalysis Methode verloren gehen. Für große Systeme wäre allerdings auch eine Kombination aus beiden Ansätzen denkbar: Als erstes werden generellere Usecases betrachtet und für diese Vor- und Nachbedingungen mit Objektdiagrammen formuliert. Danach können algorithmische Aspekte für bestimmte Szenarion durch Überführen der graphischen Verträge in Storyboards behandelt werden.

Das Graphersetzungssystem PROGRES [Pro04] kann als Vorgänger des Fujaba CASE Tools bezeichnet werden. Daher sind viele Konzepte in den Graphtransformationsregeln der Storydiagramme von Fujaba aus PROGRES übernommen. [SWZ95] beschreibt einen Softwareentwicklungsprozess mit PROGRES, den sogenannten PROGRES process. Im Gegensatz zu dem hier vorgestellten FUP hat der PROGRES process allerdings keine explizite Analysephase wie das Objektspiel oder das Storyboarding. Man kann also sagen, das der FUP eine Weiterentwicklung des PROGRES process ist.

8. Einsatz

Der hier beschriebene systematische Software-Entwicklungsprozess FUP wurde in mehreren Lehrveranstaltungen, in zwei Industrieprojekten und in der Entwicklung von Fujaba eingesetzt und erprobt. Da die hierbei gewonnenen Erkenntnisse direkt zur Verbesserung des Prozesses herangezogen wurden, sind viele dieser Projekte schon in den vorangegangenen Abschnitten erwähnt worden. In diesem Kapitel wird der Einsatz des Fujaba Process in diesen und weiteren Projekten noch einmal zusammengefasst.

Wir setzen den Fujaba Process in der Programmiermethodik Vorlesung an der Universität Kassel seit nunmehr 7 Jahren ein. An diesen Kursen nehmen jeweils ca. 60 Studenten des zweiten oder dritten Semesters teil. In diesem Kursen wird zunächst eine Iteration des Prozesses anhand eines einfachen Szenarios von uns durchgeführt und erklärt. Danach entwerfen die Studenten eigene Szenarien und wenden den FUP schließlich an einem kleinen Projekt an. Unseren Erfahrungen nach finden sich die Studenten sehr schnell in diesen neuen Prozess ein. Es ist in der Regel an jeder Stelle klar, was es als nächstes zu tun gibt. Der FUP leitet die Studenten also klar und einfach zur modellbasierten Softwareentwicklung an. Die Studenten investieren unseren Beobachtungen nach einige Zeit in gute Szenarien, da diese zur Diskussion im Team, zur Anforderungsdefinition und zur Testspezifikation dienen. Durch den frühen Einsatz von Objekten im Objektspiel und in der Storyboardingphase wurde, unserer Meinung nach, die Qualität der von unseren Studenten abgegebenen Modelle signifikant verbessert. Eine Auflistung aller Vorlesungen und der darin behandelten Inhalte findet sich in Anhang A.1.

Der Fujaba Process wurde ausgiebig im Forschungsprojekt Optimierung von Fahrzeug-Bordnetz-Architekturen (OBA) [GSZ+05c, ZLM+06, Gem08] eingesetzt. OBA ist eine Kooperation der Fachgebiete „Fahrzeugsysteme und Grundlagen der Elektrotechnik“ und „Software Engineering“ der Universität Kassel zusammen mit der Volkswagen AG, der Continental AG (ehemals Siemens VDO) und Sumitomo Electric Bordnetze (ehemals VW Bordnetze). Im OBA Projekt entstand eine Software, die unter anderem die Kabelbäume in Fahrzeugen optimieren kann. Zur Entwicklung dieser Software war eine enge Zusammenarbeit der Domänenexperten des Fachgebiets „Fahrzeugsysteme“ mit den Entwicklern des Fachgebiets „Software Engineering“ unerlässlich. Um diese Zusammenarbeit zu erleichtern, wurde nach dem Fujaba Process vorgegangen. Hier stellten sich die Objektspiele und die Storyboards als hervorragend geeignet zur Anforderungsanalyse und zu frühen Designbetrachtungen heraus. Nach einer kurzen Eingewöhnungsphase waren die Domänenexperten in der Lage, die Storyboards zu verstehen und von den Softwaretechnikern entwickelte Szenarien zu überprüfen und zu kommentieren. Innerhalb kurzer Zeit entwickelten die Elektrotechniker dann sogar eigene Szenarien, die sie den Entwicklern vorlegten,

um diese auf noch nicht betrachtete Sonderfälle oder Probleme hinzuweisen. Insgesamt wird der Einsatz von Objektdiagrammen und Storyboards in diesem Projekt als Schlüssel zum Erfolg betrachtet.

In einem anderen Industrieprojekt entwickelte ein Student für die Firma Krauss-Maffei Wegmann Grundlagen eines Kommunikationsprotokolls [Czo04]. In dieser Firma wurden zuvor CRC Karten zur objektorientierten Analyse eingesetzt. In diesem Projekt wurden nun das Objektspiel und Storyboards erprobt. Das Projekt wurde komplett in C# entwickelt. Da Fujaba bislang keine C# Codegenerierung unterstützt, wurde hier auf den Einsatz von Fujaba verzichtet. Stattdessen wurde Visio zum Erstellen der Storyboards eingesetzt und die Methodenimplementierungen direkt in C# verfasst. Trotzdem empfanden die Mitarbeiter von KMW die Storyboards als sehr hilfreich. Dies lag zum einen daran, dass Designaspekte gut mit den Storyboards diskutiert werden konnten. Zusätzlich erwiesen sich die Storyboards auch als gute Grundlage zur Implementierung von Methoden, auch wenn diese nicht mit Storydiagrammen sondern direkt in einer objektorientierten Programmiersprache erstellt wurden. Der hier vorgestellte Ansatz kann also auch unabhängig vom Fujaba CASE Tool eingesetzt werden.

An der Universität Kassel wird der FUP auch zum Entwickeln von Fujaba selbst eingesetzt. In unserer Gruppe arbeiten wir an ca. einer Millionen Zeilen Fujaba Quelltext. Zusätzlich kommen noch etliche Werkzeuge, Bibliotheken und Quellcode von Drittherstellern hinzu. Insbesondere das Eclipse Framework mit seinen unzählige Plugins erhöht hier nochmals die Komplexität. Aber sogar in solchen eher umfangreichen Systemen, lassen sich Objektspiel und Storyboards ideal einsetzen, um die Analyse von neuer oder geänderter Funktionalität durchzuführen. Auch um die Integration der neuen Funktionalität in das bestehende System zu untersuchen und zu diskutieren, setzen wir oft Storyboards ein. Methodenimplementierungen mit Storydiagrammen sind allerdings nicht immer möglich, da die Reverse-Engineering-Funktionalität von Fujaba noch nicht besonders gut mit Methodenrümpfen umgehen kann. Daher werden diese oft von Hand in Java implementiert. Teilprojekte, die komplett neu entwickelt werden, wie etwa CodeGen2, werden allerdings meist komplett modellbasiert entwickelt.

In [GSZ05a] haben wir den Fujaba Process auf die Entwicklung von eingebetteten Systemen übertragen. Im so entstandenen μFUP wird zuerst mit dem Vorgehen, das der FUP beschreibt, eine generelle Simulationsumgebung entwickelt. Dann werden einzelne Komponenten verfeinert, verteilt, auf das eingebettete System migriert und schließlich mit der physikalischen Umgebung, also den echten Sensoren anstelle der Simulation, verbunden. Die Simulationskomponente kann dann jedoch weiterhin als Monitoring-Komponente verwendet werden, um die korrekte Funktionalität der Software sicherzustellen. Dieser μFUP wurde in einer von mir betreuten Diplomarbeit entwickelt und zur Entwicklung eines LEGO-Modells eines Hochregallagers eingesetzt.

9. Fazit

In Teil I wurde der von mir zu großen Teilen mitentwickelte Fujaba Process vorgestellt. Der FUP ist ein systematischer Softwareentwicklungsansatz, bei dem zuerst die Anforderungen mit Hilfe von Usecases gesammelt und strukturiert werden. Für jeden Usecase werden dann Szenarien in Form von Beispielabläufen entwickelt. Bei der Erstellung der Szenarien wird nach dem Objects-first Ansatz vorgegangen. Zuerst wird hierbei ein Szenario im Objektspiel durchgespielt und analysiert. Die Ergebnisse des Objektspiels werden dann als Storyboards formalisiert. Aus diesen Storyboards lässt sich zum einen leicht ein initiales Klassendiagramm ableiten, weiterhin ist es möglich hieraus automatische Tests zu generieren. Zusätzlich bieten die Storyboards einen ersten Anhaltspunkt, wie das Verhalten zu implementieren ist. Das Verhalten wird im Fujaba Process durch Storydiagramme üblicherweise ebenfalls modellbasiert entwickelt.

Für alle Schritte des Fujaba Process wird durch diese Arbeit Werkzeugunterstützung im Rahmen des Fujaba CASE Tools bereitgestellt. Einzelne Komponenten von Fujaba wurden hierzu angepasst, eine Reihe von Komponenten entwickelt und alle Komponenten zu einer integrierten Unterstützung des FUPs zusammengeschlossen. Dies bedeutet zum einen, dass alle im Fujaba Process entstehenden Diagrammarten durch Fujaba unterstützt werden, zum anderen bietet Fujaba durch den CodeGen2 Codegenerator die Möglichkeit aus den entstandenen Modellelementen ausführbaren Java Quelltext zu generieren. Hierbei wird aus den Klassendiagrammen und den Storydiagrammen der Applikationscode generiert sowie aus den Storyboards der Testcode. Die generierte Applikation und die Tests lassen sich mit einer Standard Java Umgebung kompilieren und ausführen.

Wir haben die einzelnen Schritte des Fujaba Process immer wieder in studentischen Projekten erprobt und bei auftretenden Schwierigkeiten verbessert. Hierbei wurde insbesondere darauf geachtet, dass durch Reduktion von Komplexität eventuelle Lernhürden und Verständnisschwierigkeiten ausgeräumt werden konnten. Zusätzlich wurde der Fujaba Process in einigen Industrieprojekten eingesetzt. Die Ergebnisse dieser Fallstudien zeigen insbesondere, dass durch den Einsatz des Fujaba Process in den studentischen Projekten die Qualität der entwickelten Klassendiagramme merklich gestiegen ist und das in den industriellen Projekten die Kommunikation zwischen Entwicklern und Domänenexperten deutlich verbessert werden konnte.

Um die Codegenerierung im Fujaba Process flexibler zu gestalten, wurde von mir, Christian Schneider und Carsten Record die in Abschnitt 5.7 beschriebene templatebasierte Codegenerierung CodeGen2 neu entwickelt. Diese wurde zu großen Teilen in Fujaba selbst entwickelt und umfasst ca. 260 Klassen. Dies entspricht ungefähr 60.000 Zeilen generiertem Java Quelltext. Zusätzlich wurden Templates für Java, für das Modellierungsframework von Eclipse EMF sowie erste Templates für die Sprache

C++ geschrieben. In einer anderen Arbeit entstanden mittlerweile auch Templates für das Google Web Toolkit (GWT), [ADH+09].

Zusammenfassend ist also ein komplett modellbasierter Softwareentwicklungsprozess entstanden, für den für jeden Schritt Werkzeugunterstützung bereitsteht. Für viele Schritte wurden Generatoren entwickelt, die dem Benutzer fehleranfällige Aufgaben abnehmen. Komplexitätshürden im Prozess wurden erkannt und, zum Beispiel durch zusätzliche Zwischenschritte, beseitigt. Die Praxistauglichkeit des Prozesses wurde anhand von Fallstudien gezeigt. Der Fujaba Prozess eignet sich also ideal zum modellbasierte Entwickeln objektorientierter Software.

Teil II.

Testen auf Modellebene

10. Motivation

In Teil I wurden Storyboards als geeignetes Mittel zur Formalisierung von Usecase Szenarien vorgestellt. Diese Storyboards dienen als formalisierte Anforderungsdefinition, partitionieren das Projekt in Teilaspekte, bilden die Grundlage für erste Diskussionen über Designfragen und geben erste Ansatzpunkte, wie die Implementierung zu realisieren ist. Aufgrund der ähnlichen Syntax von Storyboards und den im Fujaba Process im Implementierungsmodell verwendeten Storydiagrammen, können sogar Teile aus Storyboards in der Implementierung wiederverwendet werden. Zusammenfassend lässt sich sagen, dass die Storyboards den weiteren Prozess leiten. Während aus Storydiagrammen direkt ausführbarer Java Quelltext generiert wird, tragen die Storyboards jedoch bislang nicht direkt zum eigentlichen Programm bei.

In längeren Projekten passiert es häufig, dass die Implementierung überarbeitet, das zugrunde liegende Modell angepasst oder Anforderungen vom Kunden geändert werden. Da die zugrunde liegenden Storyboards nur eine informelle Beziehung zur Implementierung besitzen, werden diese aus Zeitmangel häufig nicht mit aktualisiert. Andersrum ist auch nicht sichergestellt, dass, wird ein Storyboard angepasst, die bisherige Implementierung noch zu diesem Storyboard passt, also das modellierte Szenario enthält. Somit laufen Anforderungsmodell und Design- / Implementierungsmodell auseinander. Die Dokumentation passt also nicht mehr zum Produkt. Eine nachträgliche Synchronisierung der Dokumente ist in der Regel sehr zeitaufwendig. Das Problem der asynchronen Anforderungsdokumente und der Implementierung ist kein für den FUP spezifisches sondern ist auch in anderen Prozessen zu beobachten [Sih05].

Um eine engere Bindung zwischen Implementierung und Szenarien herzustellen existieren mehrere Möglichkeiten:

1. Es könnte versucht werden die Implementierung aus den Szenarien zu gewinnen. Diese Idee liegt beispielsweise den Ansätzen im „Scenario to state machines“ Kontext zugrunde. Dieses Vorgehen wird in Abschnitt 14 näher beleuchtet.
2. Außerdem könnte versucht werden aus einer fertigen Implementierung Szenarien zurückzugewinnen. Dies könnte beispielsweise durch Analyse des Implementierungsmodells geschehen. Liegt zum Beispiel ein Statechart als Implementierungsmodell vor, sind Kandidaten für mögliche Szenarien unterschiedliche Pfade durch das Statechart. Alternativ könnte man solche Szenarien auch durch Protokolle der Ausführung der Applikation erstellen. So aus der Implementierung zurückgewonnenen Szenarien müssten dann wieder mit den bestehenden Szenarien verglichen werden. Sollten bei diesem Vergleich zwischen zwei Szenarien aber Unterschiede auftreten, ist ohne Benutzerinteraktion nicht erkennbar, ob es sich bei dem aus der Implementierung gewonnenen Szenario

um ein neues handelt oder ob tatsächlich ein Konflikt, zum Beispiel durch eine fehlerhafte Implementierung, vorliegt. Ein weiteres Problem stellt hier der Detailgrad dar: Protokolle, die aus der Implementierung gewonnen werden, haben meist einen sehr viel höheren Detailgrad als Analyseszenarien. Um solche Implementierungsprotokolle wieder in lesbare Szenarien zu verwandeln, muss also eine Abstraktion erfolgen und bei automatischen Abstraktionen besteht immer die Gefahr interessante Details „wegzuabstrahieren“ oder den Detailgrad nicht genug zu reduzieren.

3. In dieser Arbeit werden aus den Szenarien Tests generiert. Die Tests prüfen, ob die aktuelle Implementierung das zugehörige Szenario enthält. Somit resultiert eine einseitige Änderung des Szenarios oder der Implementierung in einem fehlschlagenden Test. Der Entwickler muss dann entscheiden ob entweder die Implementierung fehlerhaft ist, oder das Szenario angepasst werden muss. Auf diese Art ist die Synchronität von Szenarien und Implementierung gewährleistet.

Der Testgenerierung aus Storyboards liegt folgende Idee zugrunde: Ein Storyboard beschreibt in der ersten Aktivität die Vorbedingung in Form einer Objektstruktur. Diese wird durch den Test hergestellt. Dann folgen eine Reihe von Aktionen, die vom Test durchgeführt werden. Am Schluss jeder Aktion wird eine bestimmte Objektstruktur erreicht, die ebenfalls durch das Storyboard beschrieben ist. Das erreichen dieser Nachbedingungen wird vom Test sichergestellt. Wird eine solche Objektstruktur nach Ausführung einer Aktion nicht gefunden, wird ein Testfehler gemeldet. Die Storyboards müssen hierfür eine geeignete Struktur aufweisen. Diese Struktur wird im nächsten Abschnitt diskutiert.

Das automatische Generieren von Tests unterstreicht die Tatsache, dass der Prozess von den Szenarien getrieben wird noch zusätzlich. Nach Erstellung eines Storyboards kann also sofort automatisch ein Test generiert werden. Solange dieser Test fehlschlägt, kann davon ausgegangen werden, dass die Implementierung dieses Szenario noch nicht abdeckt. Dies entspricht dem „test first principle"´ des eXtreme Programming, vergleiche [BA04].

Durch die generierten Tests bekommen die Szenarien also eine eigene Ausführungssemantik. Dadurch besteht das Problem des „Veraltens“ von Szenarien nicht mehr, da die Szenarien durch die Tests zum Produktivcode beitragen und somit mitgepflegt werden. Die Tests überprüfen, ob die Implementierung das entsprechende Szenario enthält. Somit überprüfen die Tests, ob die zuvor spezifizierten Anforderungen eingehalten werden. Aus den Usecases im FUP werden validierte Usecases.

Zusätzlich wird in diesem Teil auf die Möglichkeit eingegangen, alternative Szenarien zu spezifizieren. Weiterhin werden Ansätze, um vernünftige Fehlermeldungen zu generieren, vorgestellt. Es wird die Möglichkeit eigene Tests zu modellieren diskutiert und der Aspekt der Testüberdeckung erläutert.

11. Konzept

Um die im vorangegangenen Kapitel angesprochenen validierten Usecases zu erhalten, ist es Ziel aus Usecase Szenarien, die durch Storyboards formalisiert wurden, Testfälle zu generieren. Dieses Kapitel beschreibt meine Lösungsidee, die dieser Arbeit zugrunde liegt.

In meinem Ansatz müssen die Storyboards eine bestimmte Struktur aufweisen. Die erste Aktivität beschreibt die *start situation*, enthält also die Objektstruktur, die den Zustand des Systems zu Beginn des Szenarios beschreibt. Hierbei kann sich der Entwickler auf die für dieses spezielle Szenario wichtigen Objekte beschränken. Diese erste Aktivität beschreibt sozusagen die Vorbedingung des Szenarios. Gefolgt wird dieser Schritt üblicherweise von einem *actor step*. Ein *actor step* beschreibt eine Einwirkung des Aktors auf das System. Aktor bezieht sich hier auf den Aktor im zugehörigen Usecase. Dies kann zum Beispiel eine Interaktion mit dem Benutzer der Applikation sein. Die Aktoraktion wird meist durch einen Methodenaufruf modelliert. Dieser Methodenaufruf stößt das Szenario an, es folgt eine Reaktion des Systems. Statt eines Methodenaufrufs wäre genauso eine Änderung in der Objektstruktur, wie z.B. eine Attributänderung, denkbar, die dann beispielsweise über ein Listenerkonzept die Systemreaktion auslöst. Ein *actor step* wird üblicherweise von einer Menge an *system steps* gefolgt. In diesen wird die Systemreaktion, also das Verhalten, dass durch die Aktorinteraktion ausgelöst wurde, modelliert. Ist die Systemreaktion abgeschlossen, kann wieder ein *actor step* folgen, an den sich wiederum *system steps* anschließen usw. Die letzte Aktivität ist üblicherweise wieder ein *actor step*, der aber keinen Methodenaufruf enthält. In dieser sogenannten *result situation* wird die Objektstruktur nach Durchlauf des Szenarios beschrieben, es wird also die Nachbedingung des Szenarios modelliert.

Diese spezielle Struktur der Storyboards wird durch das Fujaba Tool unterstützt, da die Textvorlagen, mit denen man die Usecase Szenarien textuell spezifiziert, bereits die entsprechende Form haben. Diese Struktur wird beim Erzeugen eines leeren Storyboards aus einer textuellen Beschreibung übernommen.

Ein Test, der prüft, ob das Szenario von der aktuellen Implementierung abgedeckt wird, sollte folgendermaßen ablaufen:

1. Die Objektstruktur, wie sie in der *start situation* beschrieben ist, wird angelegt.
2. Für jeden folgenden *actor step* wird
   a) getestet, ob die beschriebene Objektstruktur auch tatsächlich erreicht wurde.
   b) der beschriebene Methodenaufruf oder die beschriebene Änderung durchgeführt.

3. *system steps* werden vom Test ignoriert. Auf die Behandlung von *system steps* wird in Abschnitt 12.2 genauer eingegangen.

Ein solcher Test wird in Form eines JUnit Testfalls generiert. Wir verwenden JUnit, da Fujaba standardmäßig Java generiert, und da JUnit in der Java Welt weit verbreitet und einfach einsetzbar ist. Allerdings ist der hier beschriebene Ansatz nicht von der JUnit Technologie abhängig. Er lässt sich leicht auch für andere Testframeworks implementieren.

Die Grundlagen zur hier vorgestellten Testgenerierung sind bereits in meiner Diplomarbeit [Gei04] beschrieben. Jedoch fehlen dort noch die Behandlung von mehreren Aktorinteraktionen, die Behandlung von Alternativszenarien, ein modellbasierter Testabdeckungsansatz und die Integration in Eclipse. Das Konzept aus [Gei04] wird auch in [GZ03] erläutert. [GZ05] stellt die Behandlung von mehreren Aktorinteraktionen vor.

In Kapitel 12 wird die Umsetzung der Testgenerierung beschrieben. Hierzu wird in Abschnitt 12.1 zunächst das in diesem Kapitel verwendete Beispiel erklärt. In Abschnitt 12.2 wird dann die Funktionsweise der Testgenerierung dargestellt, in Abschnitt 12.3 werden die hierfür benötigten Änderungen am Fujaba Codegenerator beschrieben und in Abschnitt 12.4 wird die Integration in Eclipse gezeigt. Im Falle eines fehlschlagenden Tests bieten die in Abschnitt 12.5 beschriebenen Techniken erste Anhaltspunkte auf der Suche nach der Fehlerursache. Der letzte Abschnitt von Kapitel 12 beschäftigt sich schließlich mit der Erstellung von modellbasierten Abdeckungsberichten. In Kapitel 13 wird der Einsatz der zuvor beschriebenen Testgenerierung diskutiert und Erfahrungen aus diesem Einsatz dargelegt. Kapitel 14 vergleicht den Testgenerierungsansatz mit bestehenden Arbeiten und in Kapitel 15 wird schließlich ein Fazit gezogen.

12. Umsetzung

12.1. Beispiel

Um die genaue Funktion der Testgenerierung zeigen zu können, wird ein Szenario mit mehreren *actor steps* benötigt. Es wird daher ein Szenario zum Usecase Execution aus dem Usecasediagramm in Abbildung 5.1 betrachtet. Dieser Usecase behandelt die Ausführungsengine des Statecharteditors. Der Einfachheit halber werden hier nur solche Statecharts betrachtet, die schon mit dem im letzten Teil vorgestellten `StateChartFlattener` in einen endlichen Automaten konvertiert wurden. Die textuelle Beschreibung des Szenarios beschreibt die linke Spalte von Tabelle 12.1. Die Abbildungen 12.1 und 12.2 zeigen das zugehörige Storyboard für das Execution Szenario.

In der ersten Aktivität in Abbildung 12.1 wird, ähnlich wie im Storyboard in Abbildung 5.3, die Objektstruktur gezeigt, die das zu testende Statechart beschreibt.

Im nächsten Schritt wird ein Objekt vom Typ `FSMSimulator` angelegt und der aktuelle Zustand des Simulators durch die `current` Kante auf den Startzustand des Statecharts gesetzt. Dann wird das Event `lift` an den Simulator geschickt. Dies kann zum Beispiel durch eine Benutzereingabe geschehen und wird durch den Methodenaufruf `handleEvent(“lift“)` modelliert.

Im nächsten Schritt wird das Event vom System verarbeitet. Da vom aktuellen Zustand `s1` eine Transition mit dem Label `“lift“` (`t1`) existiert, wird das Event verarbeitet und der Zustand der Simulators wechselt von `s1` entlang `t1` in den Zustand `s2`.

In der nächsten Aktivität (die erste in Abbildung 12.2) ist nun also `s2` der aktuelle Zustand des Simulators. An dieser Stelle könnte das Szenario schon beendet sein, der Entwickler hat sich aber entschlossen, eine Sequenz von zwei Events zu modellieren. Daher wird im diesem Schritt das Event `dial` an den Simulator geschickt. Dieses Event wird im nächsten Schritt verarbeitet, so dass in der letzten Aktivität von Abbildung 12.2 der Zustand `s3` den aktuellen Zustand des Simulators bildet.

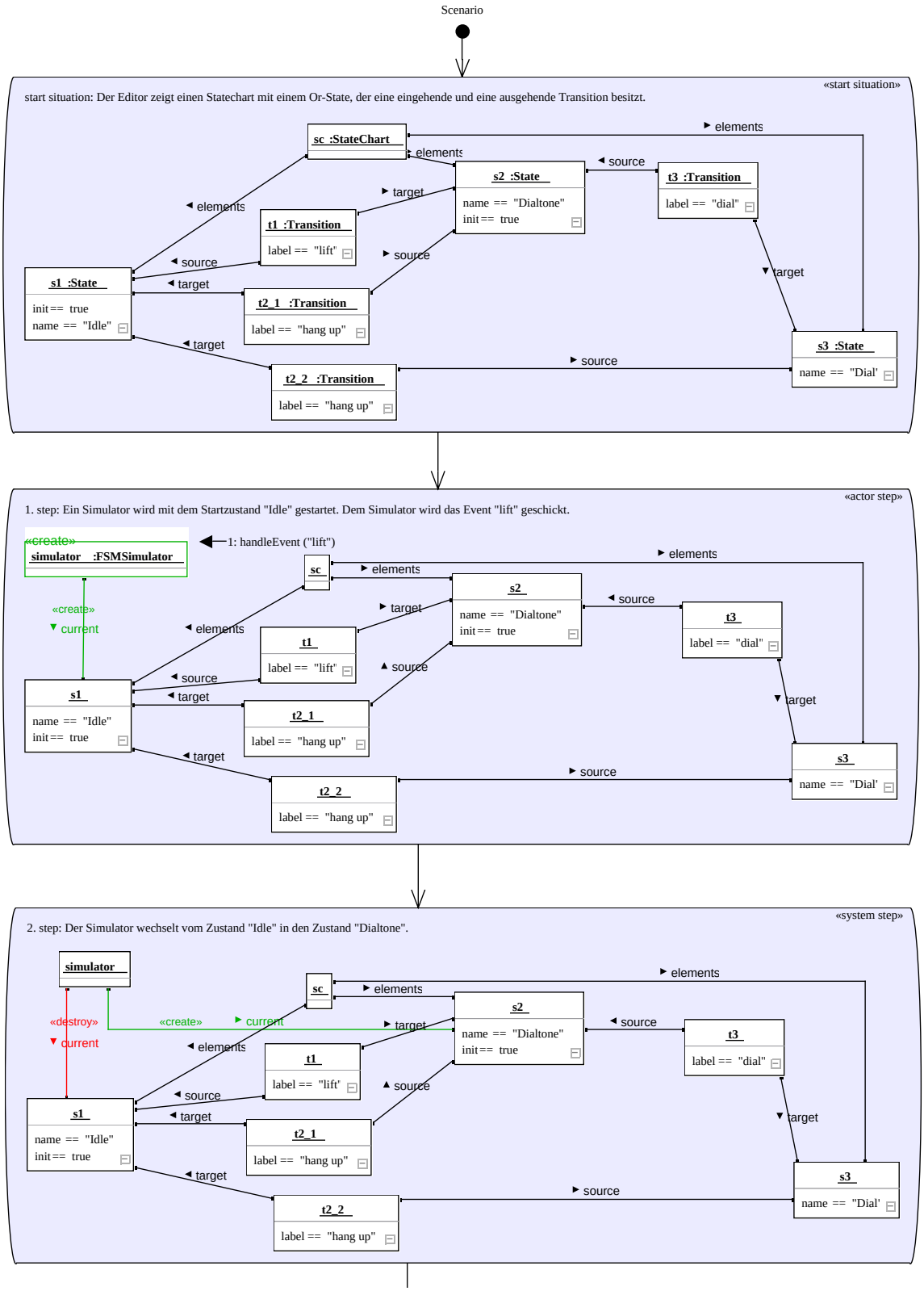


Abbildung 12.1.: Storyboard Execution Teil 1

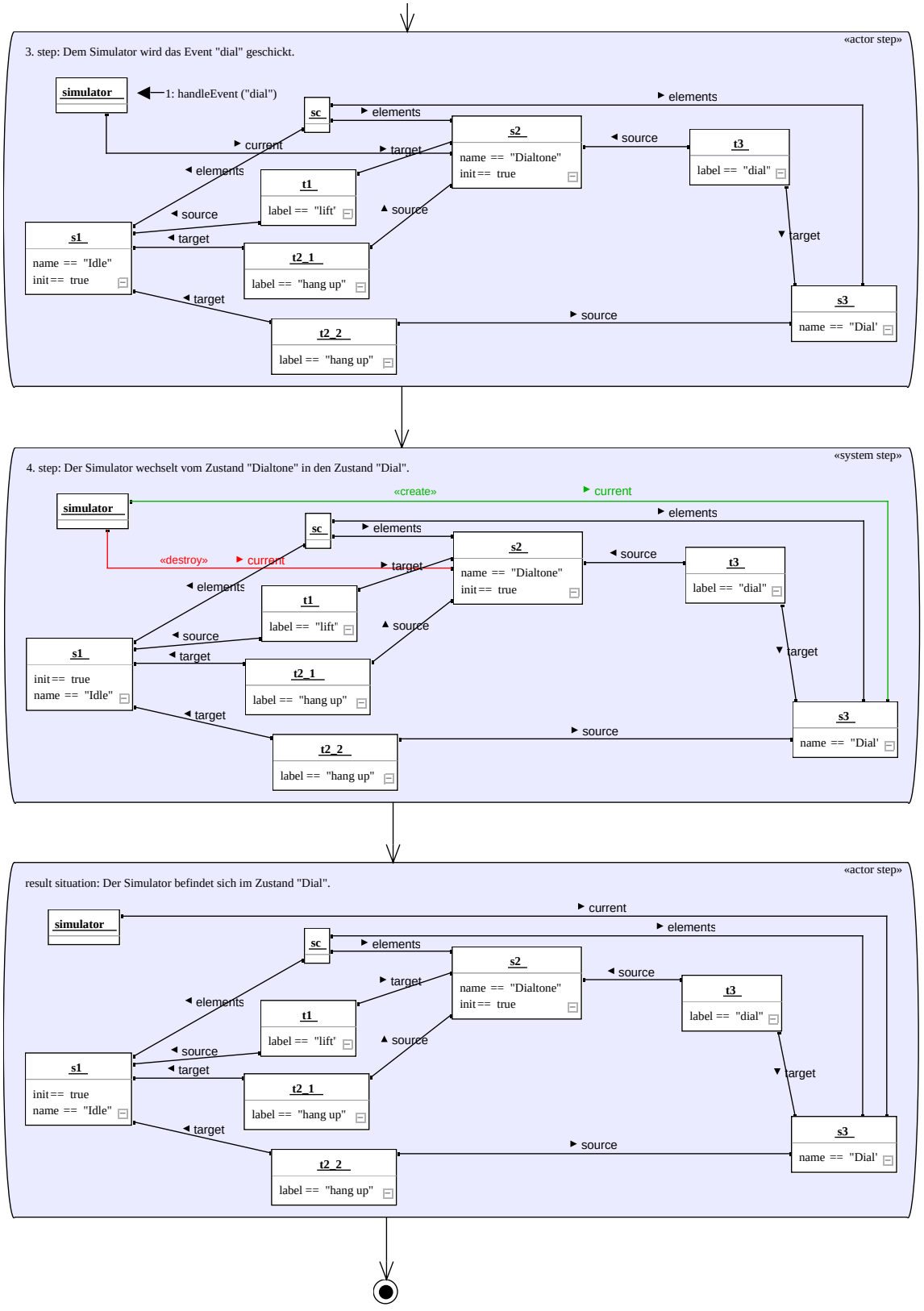


Abbildung 12.2.: Storyboard Execution Teil 2

12.2. Testgenerierung

Um aus Storyboards Tests zu generieren, muss der Entwickler zuerst spezifizieren, welche Aktivitäten vom Aktor und welche vom System ausgeführt werden. Dafür werden die Stereotypen ≪`actor step`≫ und ≪`system step`≫ verwendet. Der Stereotyp ≪`start situation`≫ kennzeichnet zusätzlich die Startsituation des Szenarios. In Abbildung 12.1 wurde daher die erste Aktivität als ≪`start situation`≫ markiert, die folgende als ≪`actor step`≫, die nächste als ≪`system step`≫ usw. Zusätzlich muss das Storyboard selbst noch mit dem Stereotyp ≪`test scenario`≫ versehen werden, damit es bei der Codegenerierung berücksichtigt wird. Auf diese Weise lassen sich auch „informelle“ Storyboards erstellen, die nur zu Dokumentationszwecken dienen und aus denen keine Tests generiert werden.

Auf dem Weg vom Storyboard zum JUnit Test wird in dieser Arbeit noch ein Zwischenschritt eingelegt. Es wird also nicht direkt aus dem Storyboard JUnit kompatibler Quellcode generiert, sondern erst Fujaba-Klassen mit Storydiagrammen aus denen dann der Quelltext erzeugt wird. Das Generieren der entsprechenden Storydiagramme aus Storyboards ist aufgrund der syntaktischen Ähnlichkeit der beiden Diagrammtypen sehr einfach. Vielfach muss lediglich die Objektstruktur aus dem Storyboard ins Storydiagramm kopiert und anschließend nur noch leicht modifiziert werden. Für dieses Kopieren wird der von mir mitentwickelte Kopiermechanismus der Fujaba Tool Suite verwendet, siehe [GS07]. Dieser Mechanismus setzt auf dem in [Sch07] beschriebenen CoObRA Persistenz-Framework auf. Das CoObRA Framework wird genauer in Abschnitt 18.2.8 beschrieben.

Um aus einem Storyboard einen Testfall zu generieren, müssen also die Objektstrukturen des Storyboards in eine Menge von Storydiagrammen kopiert werden. Um einen Test zu erzeugen werden folgende Schritte automatisch vom Fujaba CASE Tool während der Codegenerierung ausgeführt:

1. Es wird eine JUnit Testklasse für jedes Storyboard angelegt. Für das Szenario aus Abbildung 12.1 wird also eine Klasse `Scenario_ExecutionTest` erzeugt, die von der JUnit Frameworkklasse `junit.framework.TestCase` erbt.

2. Fujaba legt eine `setUp()` Methode in der Testklasse an. Diese Methode wird vom JUnit Framework zu Beginn jedes Tests gerufen und dient dazu die für den Test benötigte Datenstruktur aufzubauen. In das zur `setUp()` Methode gehörende Storydiagramm wird die Startsituation des Storyboards mithilfe des oben erwähnten CoObRA Kopiermechanismus kopiert. Sämtliche Objekte und Links werden mit dem ≪`create`≫ Stereotyp markiert. Zusätzlich werden alle Attributüberprüfungen zu Attributzuweisungen. Um in der Testmethode und später beim Debuggen einfach auf die angelegte Objektstruktur zugreifen zu können, werden alle erzeugten Objekte über Links mit dem Testobjekt verlinkt. Hierzu werden entsprechende Assoziationen im Klassendiagramm von der Testklasse zu den Klassen der entsprechenden Objekten erstellt.

   Abbildung 12.3 zeigt die generierte `setUp()` Methode für das Storyboard aus Abbildung 12.1. In der ersten Aktivität wird ein Listener, der sogenannte

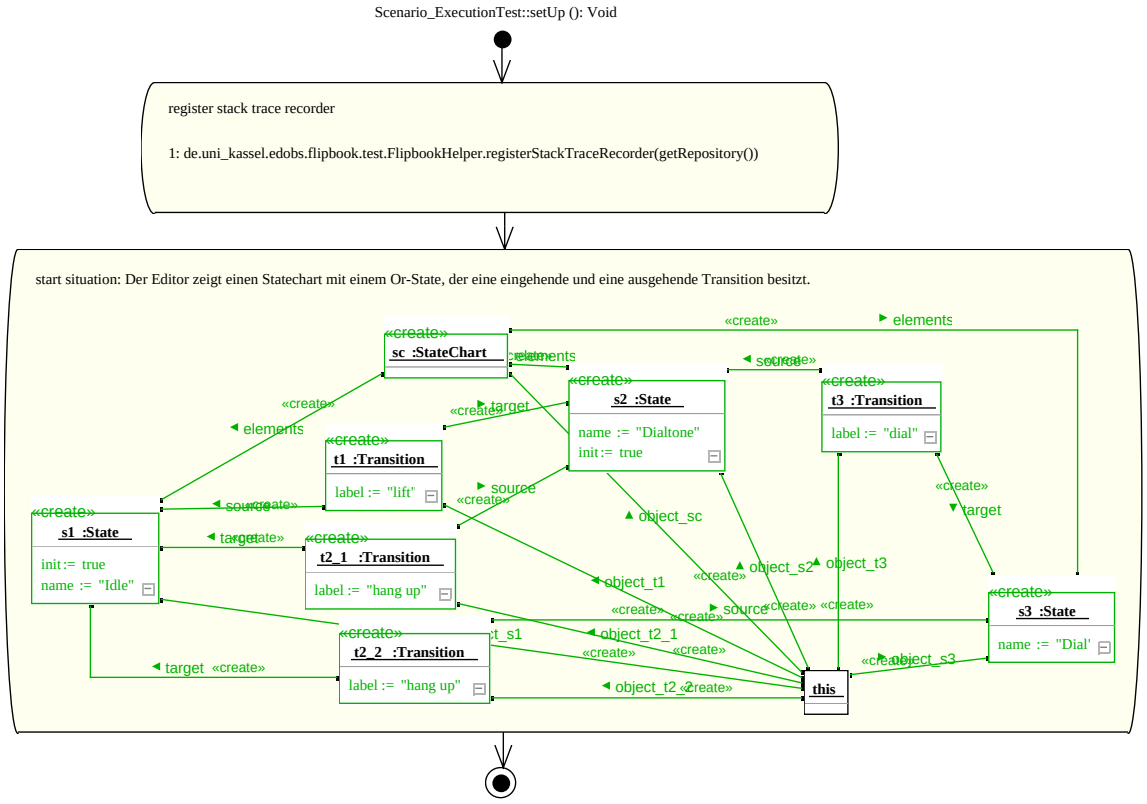


Abbildung 12.3.: Storydiagramm der JUnit `setUp()` Methode

`StackTraceRecorder` angemeldet, der die Debuggingfähigkeiten erweitert. Näheres wird in Abschnitt 18.2.8 erläutert. In der zweiten Aktivität wird die Objektstruktur erzeugt und mit dem Testobjekt (`this`) verlinkt. Die verwendeten Assoziationen heißen hierbei `object_` gefolgt von dem zugehörigen Objektnamen.

Das Ergebnis der `setUp()` Methode ist also, das zu Beginn jeder Testausführung genau die Objektstruktur angelegt wird, die im zugehörigen Usecase Szenario als Startpunkt modelliert wurde.

3. Der Testgenerator legt anschließend eine Testmethode in der Testklasse an. In JUnit Version 3 müssen solche Methoden immer mit dem Prefix `test` beginnen, daher wird eine Methode mit diesem Prefix gefolgt vom Szenarionamen angelegt. Der Testgenerator iteriert dann über die folgenden Aktivitäten des Storyboards. Für jeden *actor step* wird eine `assertStepX` Methode generiert, wobei `X` eine laufende Nummer ist. Diese Methode wird dann in der Testmethode aufgerufen.

   Abbildung 12.4 zeigt die Testmethode für das Beispielszenario. Die Testmethoden haben immer den selben einfachen Aufbau. Es werden lediglich nacheinander die `assertStepX` Methoden der einzelnen *actor steps* gerufen.

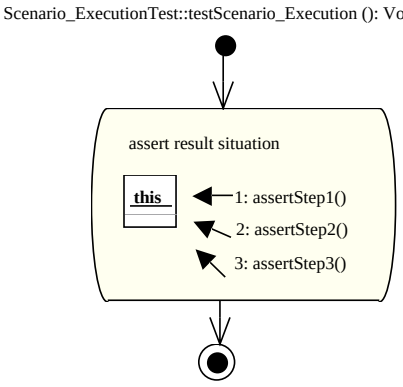


Abbildung 12.4.: Storydiagramm der Testmethode

4. In das Storydiagramm jeder `assertStepX` Methode wird die zugehörige Aktivität aus dem Storyboard kopiert. Der Aktivität wird der ≪`assert`≫ Stereotyp hinzugefügt, der der Codegenerierung signalisiert für diese Aktivität Testcode zu generieren. Dieser Testcode überprüft, ob die modellierte Objektsituation gefunden werden konnte und erzeugt, falls dies nicht der Fall ist, eine JUnit Fehlermeldung. Konnte die Objektstruktur erkannt werden, werden, falls vorhanden, die modellierten Änderungsoperationen ausgeführt. Details zur Testcodegenerierung werden in Abschnitt 12.3 erläutert. Objekte, die in dieser Methode neu erzeugt oder gesucht werden, werden wie in der `setUp()` Methode über Links beim Test angemeldet.

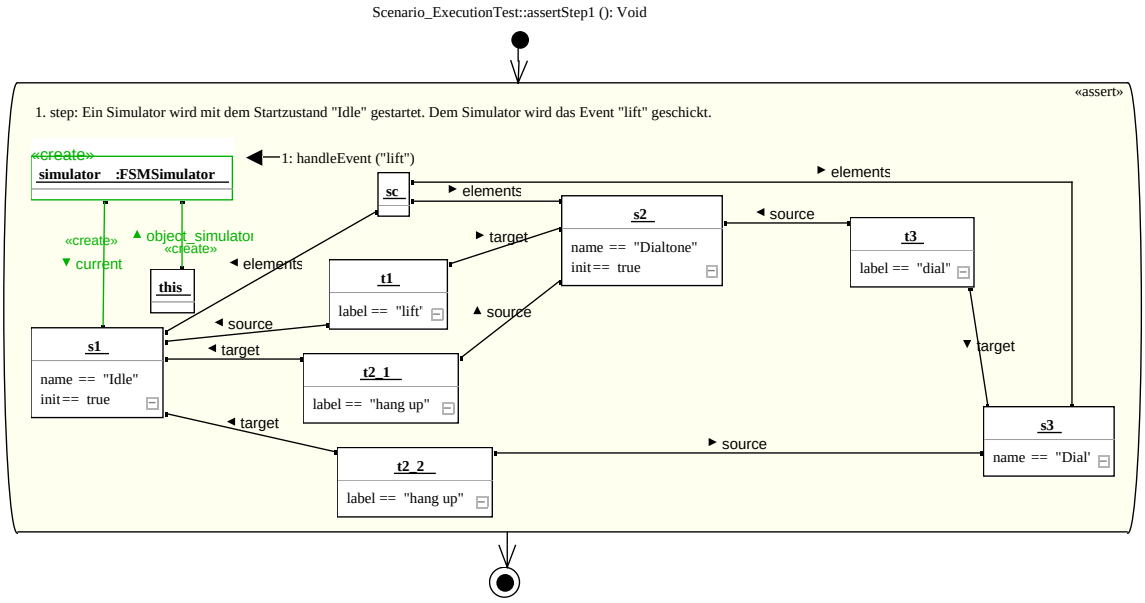


Abbildung 12.5.: Storydiagramm der Methode `assertStep1()`

Abbildung 12.5 zeigt die `assertStep1()` Methode, die für den ersten *actor step* des Storyboards aus Abbildung 12.1 generiert wurde. Diese Aktivität überprüft, ob die modellierte Objektstruktur vorhanden ist, erzeugt dann

das `simulator` Objekt und setzt auf diesem schließlich den Methodenaufruf `handleEvent("lift")` ab. Das neu erzeugt `simulator` Objekt wird über einen `object_simulator` Link beim Testobjekt registriert.

Man beachte, dass die restlichen Objekte in Abbildung 12.5 alle als gebunden markiert sind, also ohne Klassennamen dargestellt werden. Das bedeutet, dass sie zur Laufzeit bereits bekannt sind. Hier ist dies durch eine spezielle Codegenerierung der `object_` Links des Testobjekts der Fall.

Durch die `assertStepX` Methoden wird also sowohl die aktuelle Situation während des Testlaufs mit der Soll-Situation verglichen, als auch die Aktorinteraktion ausgeführt. So lassen sich sowohl *actor steps* formulieren, die nur Aktorinteraktion durchführen als auch solche, die nur eine Überprüfung vornehmen. Ein typisches Beispiel für den ersten Fall ist oft der erste *actor step* (auch *invocation* genannt), ein Beispiel für den letzten Fall ist oft der letzte *actor step* (auch *result situation*), siehe Abbildung 12.1.

Die Testmethode in Abbildung 12.4 erzeugt also zuerst in der Methode `assertStep1()` einen `FSMSimulator` und sendet ihm das Event `lift`. In der Methode `assertStep2()` wird dann überprüft, ob der Simulator in den Zustand `Dialtone` gewechselt ist. Ist dies nicht der Fall, wird ein JUnit Fehler an das Framework gemeldet. War die Überprüfung erfolgreich, wird das Event `dial` gesendet. In der Methode `assertStep3()` wir nun wiederum überprüft, ob sich der Simulator im Zustand `Dial` befindet. Ist dies der Fall, ist der generierte Test erfolgreich absolviert.

5. Aktivitäten, die mit dem ≪`system step`≫ Stereotyp markiert sind, werden von der Testgenerierung übersprungen. Sie dienen lediglich als Kommentare, welche Aktionen vom System ausgeführt werden und als Protokoll des Objektspiels. Meist bieten diese Schritte dem Entwickler dennoch wichtige Anhaltspunkte, wie die Implementierung der beschriebenen Funktionalität realisiert werden kann.

   Im Rahmen dieser Arbeit wurden verschiedene Ansätze erprobt, die *system steps* in die Testgenerierung miteinzubeziehen und diese so zum Beispiel für die Fehlersuche im Falle eines fehlschlagenden Tests zu verwenden. Diese Ansätze brachten jedoch in unseren Versuchen in der Regel wenig Mehrwert. Das liegt zum einen daran, dass die Zustände, die in den *system steps* beschrieben werden, natürlich eine Vereinfachung dessen sind, was tatsächlich passiert. Es kommt also oft vor, dass die beschriebenen Zustände so in der tatsächlichen Ausführung gar nicht vorkommen. Zum anderen wurden die *system steps* von unseren Studenten oft sehr knapp oder sogar informell gehalten. Oft enthielten diese nicht mehr Information als: „Hier passiert jetzt eine magische Reaktion des Systems.“. Solche *system steps* eignen sich natürlich auch nicht zur weiteren Fehlersuche.

6. Im nächsten Schritt generiert Fujaba Quelltext für die erzeugten Testklassen und -methoden. Wie in Fujaba üblich wird auch hier normaler Java Quelltext generiert. Daher lassen sich die Tests einfach mit manuell programmierten

Programmteilen oder externe Bibliotheken benutzen. Die nötigen Anpassungen am Fujaba Codegenerator werden in Abschnitt 12.3 beschrieben.

7. Der generierte Quelltext wird durch die Eclipse IDE kompiliert. Hierbei wird das JUnit Framework als externe Bibliothek verwendet.
8. Die generierten Tests können jetzt mithilfe des JUnit Frameworks ausgeführt werden. Dieses geschieht üblicherweise durch die JUnit-Integration von Eclipse.

   Abbildung 12.10 in Abschnitt 12.4 zeigt die Ausführung einer generierten Testsuite für das Statechart Beispiel. Hier schlägt der Test für den Usecase Flattening fehl.
9. Wenn ein Test fehlschlägt wird eine aussagekräftige Fehlermeldung, die es dem Entwickler ermöglichen soll, schnell die Fehlerursache zu erfassen, an das JUnit Framework geliefert. Zusätzlich werden mit dem eDOBS, Design Level Debugging und dem Flipbook weitere Debugginghilfen zur Suche der Fehlerursache angeboten. Diese werden ausführlich in Teil III diskutiert.

Zusammenfassend wird aus einem Storyboard also eine JUnit Testmethode generiert, die drei Hauptschritte ausführt. Zuerst wird die Objektstruktur angelegt, die die Startsituation des betrachteten Szenarios modelliert. Dann wird für jeden als *actor step* markierten Schritt des Storyboards zuerst die aktuelle Objektstruktur zur Testlaufzeit mit der in diesem Schritt modellierten verglichen. Dann werden die modellierten Änderungen, wie zum Beispiel Methodenaufrufe, des *actor steps* ausgeführt.

Solche Tests können natürlich nur dazu dienen sicherzustellen, dass die Implementierung das modellierte Szenario vollständig und fehlerfrei abdeckt. Sie bilden also keinen vollständigen Systemtest. Szenario-basierte Tests sollten immer durch andere Testverfahren, wie beispielsweise systematische Tests, die systematisch Mengen von möglichen Eingaben durchprobieren und die Ausgaben überprüfen, ergänzt werden.

12.2.1. Alternativszenarien

Bei Szenariobeschreibungen ist es oft üblich zu einem Standardszenario zusätzlich mehrere Alternativszenarien zu modellieren, siehe zum Beispiel [BD04]. Solche Szenarien ähneln bis zu einem bestimmten Punkt dem Standardszenario, dann tritt eine abweichende Aktoraktion auf und das System reagiert anders. Unter Umständen ist das System nach einiger Zeit wieder „normal“, verhält sich also wie im Standardszenario ab einem bestimmten späteren Zeitpunkt beschrieben. Typische Alternativszenarien sind zum Beispiel Fehlerfälle, bei denen beispielsweise der Aktor an einer Stelle im Ablauf eine fehlerhafte Eingabe macht und das System diese abfangen muss.

Ein Alternativszenario für das Execution Beispiel ist in Tabelle 12.1 aufgeführt. Die linke Spalte beschreibt das eigentliche Szenario. In der Spalte daneben ist das Alternativszenario aufgeführt. Beide Szenarien starten mit der gleichen Startsituation und es folgt die selbe Aktoraktion. In Schritt 3 allerdings wird im Standardfall das Event `dial` geschickt, im Alternativfall jedoch `hang up`. Das System reagiert

dementsprechend unterschiedlich und landet zum Schluss in zwei unterschiedlichen Endsituationen.

Standardfall	Alternative 1
start situation: Der Editor zeigt einen Statechart mit einem Or-State, der eine eingehende und eine ausgehende Transition besitzt.	
1. step: Ein Simulator wird mit dem Startzustand `Idle` gestartet. Dem Simulator wird das Event `lift` geschickt.	
2. step: Der Simulator wechselt vom Zustand `Idle` in den Zustand `Dialtone`.	
3. step: Dem Simulator wird das Event `dial` geschickt.	3.1. step: Dem Simulator wird das Event `hang up` geschickt.
4. step: Der Simulator wechselt vom Zustand `Dialtone` in den Zustand `Dial`.	4.1. step: Der Simulator wechselt vom Zustand `Dialtone` in den Zustand `Idle`.
result situation: Der Simulator befindet sich im Zustand `Dial`.	result situation: Der Simulator befindet sich im Zustand `Idle`.

Tabelle 12.1.: Alternativszenario für den Usecase Execution

Abbildung 12.7 zeigt einen Ausschnitt aus dem Storyboard, dass das Standardsszenario und das Alternativszenario enthält. Gezeigt ist der Ausschnitt in dem die Szenarien auseinanderlaufen. Hier tritt im Storyboard eine Verzweigung auf. Sollte das Alternativszenario wieder einen Zustand erreichen, der einem im Standardszenario entspricht, ist es hier auch möglich, dass diese Strängen wieder zusammenlaufen.

Die Testgenerierung unterstützt solche Alternativszenarien. Es werden also Storyboards unterstützt, die an beliebigen Stellen verzweigen und, falls gewollt, später wieder zusammenlaufen. Einzige Voraussetzung ist, dass alle Pfade durch ein solches verzweigendes Storyboard bei der *start situation* beginnen und ohne Schleifen bei einer Stop-Aktivität enden. Die Testgenerierung erzeugt dann für jeden möglichen Pfad eine eigene Testmethode, die die `assertStepX` Methoden der *actor steps* ruft, die auf diesem Pfad liegen.

Für das betrachtete Alternativszenario stellt Abbildung 12.6 die generierte Testmethode dar. Hier wird zuerst dieselbe *actor step* Methode gerufen wie im Standardfall, vergleiche Abbildung 12.4. Dann allerdings werden neue *actor step* Methoden aufgerufen, die die *actor steps* 3.1 und die neue *result situation* aus Tabelle 12.1 behandeln.

Die hier vorgestellten Alternativszenarien werden in Usecase-Diagrammen oft durch ≪`extends`≫ Beziehungen zwischen Standard- und Alternativszenario dargestellt. Eine solche Modellierung ist in Fujaba auch möglich, muss aber manuell erfolgen

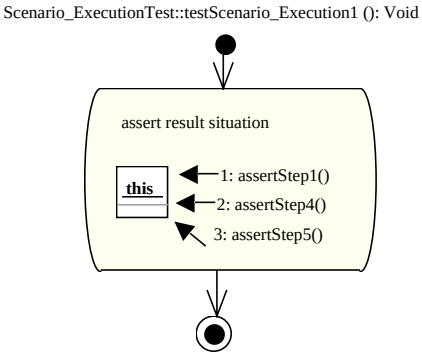


Abbildung 12.6.: Testmethode des Alternativszenarios

und wird nicht automatisch erzeugt, da die Bedeutung der ≪`extends`≫ oft unterschiedlich interpretiert wird.

Eine weitere mögliche Beziehung zwischen Usecases ist die ≪`include`≫ Beziehung. Diese besagt, dass ein bestimmter Teil eines Usecases durch einen anderen Usecase verfeinert wird. Auch diese Beziehung kann implizit in Storyboards modelliert werden. In diesem Fall enthält das Basisszenario einen Methodenaufruf und beschreibt anschließend nur das Resultat. Die Funktionalität dieser Methode kann dann in einem weiteren Usecaseszenario modelliert werden. Hierzu wird im zugehörigen Storyboard im Aktorschritt diese Methode gerufen und dann im Detail die Reaktion des Systems beschrieben. Ein Beispiel für diesen Fall enthält [DGZ03a].

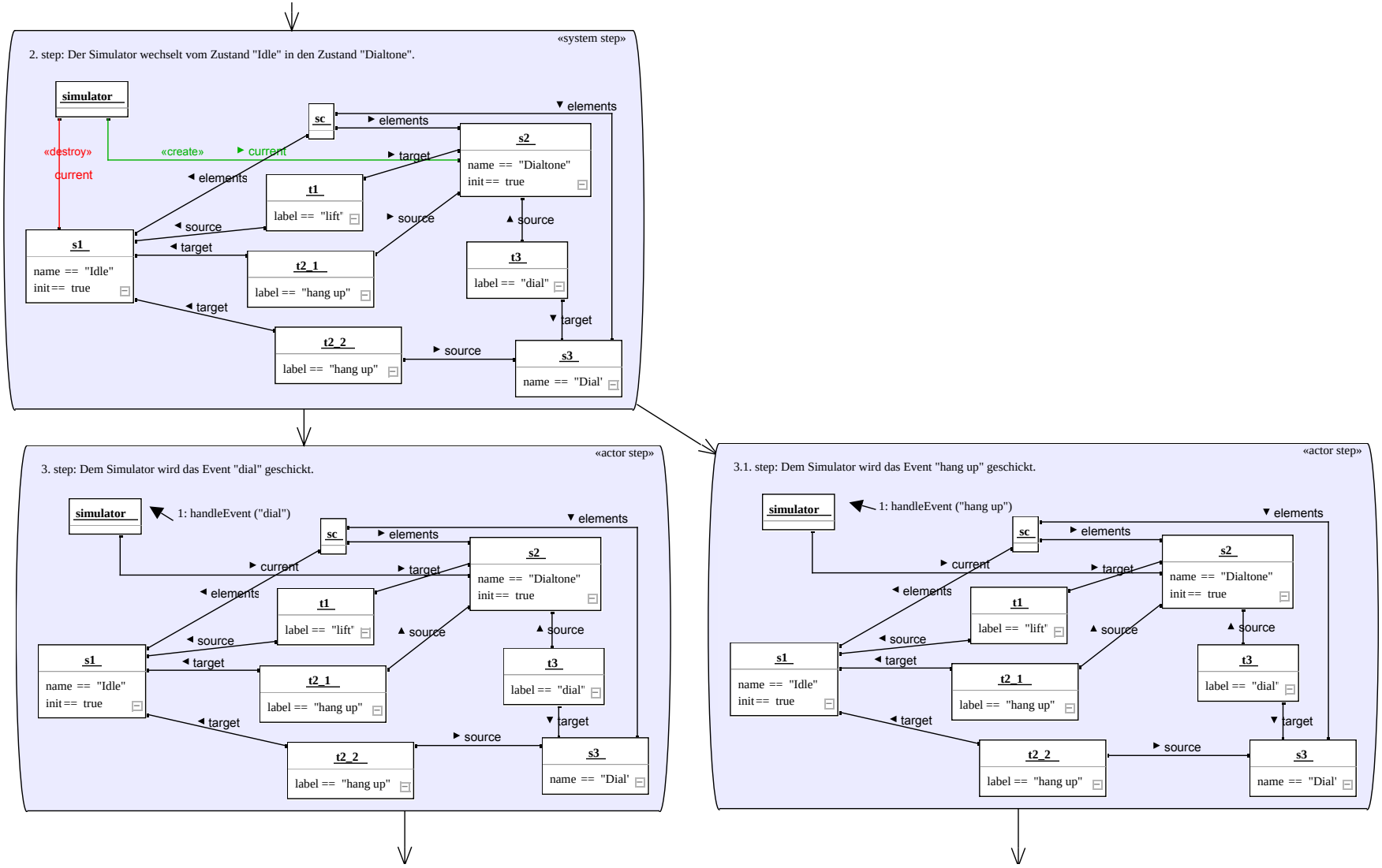


Abbildung 12.7.: Storyboard des Alternativszenarios

12.3. Anpassen des Codegenerators

Um aus Storyboards JUnit Tests zu generieren, muss zuerst die Erzeugung der Testklassen und Testmethoden in den Codegenerierungsprozess, wie er in Abschnitt 5.7 beschrieben ist, integriert werden. Bisher wurden Storyboards bei der Codegenerierung ignoriert, das heißt es wurden keine Token in der Zwischensprache der Codegenerierung hierfür angelegt. Um nun die Tests zu generieren wird eine neue `TokenCreationEngine` bei der `Engine` angemeldet, die für die Java Codegenerierung zuständig ist (siehe Abbildung 6.3). Diese neue `TokenCreationEngine` ist für Storyboards zuständig und stößt die im vorigen Kapitel beschriebene Generierung der Testklassen und -methoden an. Anschließend werden für diese neu erzeugten Elemente die normalen Token erzeugt und in den Tokenbaum gehängt. Für die Testklassen und -methoden wird also ab hier die normale Codegenerierung angewandt. Dies führt auch zum gewünschten Resultat, lediglich die `assertStepX` Methoden müssen speziellen Code generieren. Dies passiert über die Anpassung der zugehörigen Templates.

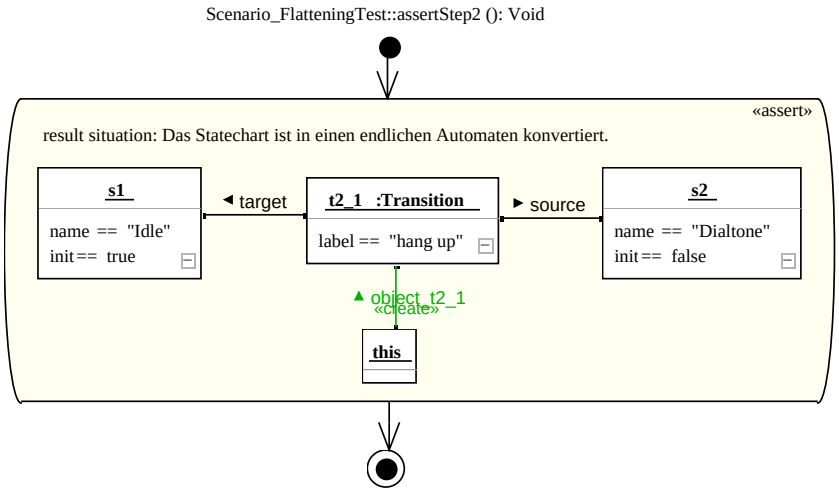


Abbildung 12.8.: Beispielmethode für die Testcodegenerierung

Um die vorgenommenen Anpassungen an den Templates zu verstehen, ist erstmal ein generelles Verständnis der Fujaba Codegenerierung für Storydiagramme notwendig. Daher wird diese kurz an einem kleinen Beispiel erklärt. Dieses Beispiel stellt eine vereinfachte Version der `assertStep2` Methode dar, die aus dem Storyboard für den Usecase Flattening (Abbildung 5.4) generiert wurde. Abbildung 12.8 zeigt das Storydiagramm dieser Methode. Hier werden die Attributwerte der Objekte `s1` und `s2` überprüft und die neu hinzugekommene Transition `t2_1` gesucht und anschließend beim Test angemeldet.

Listing 12.1 zeigt den Quelltext, der aus diesem Storydiagramm mit der normalen Codegenerierung generiert wird. Listing 12.1 beginnt mit der Deklaration der Methode gefolgt von Deklarationen der lokalen Variablen, die hier der Übersichtlichkeit halber ausgelassen wurden. Der komplette Quelltext des Storypatterns ist durch

einen `try-catch` Block gekapselt (Zeile 3-32), um das Pattern bei fehlschlagender Teilgraphsuche durch Auslösen einer `JavaSDMException` verlassen zu können. In Zeile 6 wird dann überprüft, ob das Objekt `s1` auch wirklich existiert. Hierzu wird die `JavaSDM.ensure()` Methode gerufen. Diese Methode erwartet einen booleschen Parameter und kehrt sofort zurück, wenn dieser den Wert `true` hat. Wird aber `false` übergeben, löst die Methode eine `JavaSDMException` aus. Diese Exception führt zum sofortigen Verlassen des Patternblocks durch das `catch` in Zeile 30. Anschließend wird noch die lokale Variable `fujaba__Success` auf `false` gesetzt. Dadurch wird festgehalten, dass die Aktivität fehlgeschlagen ist. In Zeile 8 wird dann ebenfalls mit der `JavaSDM.ensure()` Methode eine Attributbedingung für das Objekt `s1` überprüft. Es folgen weitere Überprüfungen, die hier ausgelassen sind.

In den Zeilen 10 bis 25 wird dann nach einem geeigneten Objekt für `t2_1` gesucht. Hierzu wird solange über alle Nachbarn von `s1`, die über die `incomingTransition` Kante erreichbar sind, iteriert, bis ein passendes Objekt gefunden wurde. Für jeden möglichen Kandidaten, der in Zeile 13 gefunden wird, werden in den nächsten Zeilen zusätzliche Bedingungen überprüft. Die gefundene Transition `t2_1` muss eine `source` Kante zum Objekt `s2` (Zeile 15) und das passende Label besitzen (Zeile 17). Schlägt einer dieser Checks fehlt, wird hier nicht das komplette Pattern verlassen, sondern nur der aktuelle Kandidat verworfen (Zeile 21). Wurde kein passendes Objekt gefunden, wird in Zeile 25 wiederum die Aktivität verlassen. Existiert aber ein geeignetes Objekt, wird es in Zeile 27 am Testobjekt angemeldet.

```
1  public void assertStep2 () {
2     ...
3     try  {
4        fujaba__Success = false;
5        // check object s1 is really bound
6        JavaSDM.ensure ( s1 != null );
7        // attribute condition init == true
8        JavaSDM.ensure ( s1.isInit () == true );
9        ...
10       fujaba__IterS1ToT2_1 = s1.iteratorOfIncomingTransitions ();
11       while ( !(fujaba__Success) && fujaba__IterS1ToT2_1.hasNext () ) {
12          try {
13             t2_1 = (Transition) fujaba__IterS1ToT2_1.next ();
14             // check link source from t2_1 to s2
15             JavaSDM.ensure (s2.equals (t2_1.getSource ()));
16             // attribute condition label == "hang up"
17             JavaSDM.ensure ( JavaSDM.stringCompare (t2_1.getLabel (),
18                "hang up") == 0 );
19             fujaba__Success = true;
20          }
21          catch ( JavaSDMException fujaba__InternalException ) {
22             fujaba__Success = false;
23          }
24       }
25       JavaSDM.ensure (fujaba__Success);
26       // create link object_t2_1 from this to t2_1
27       this.setT2_1 (t2_1);
28       fujaba__Success = true;
29    }
```

```
30      catch ( JavaSDMException fujaba__InternalException ) {
31          fujaba__Success = false;
32      }
33  }
```

Listing 12.1: Quelltext für die Beispieltestmethode

Ziel ist es nun, die soeben beschriebene Codegenerierung so anzupassen, dass ein JUnit konformer Fehler mit aussagekräftigen Fehlermeldung generiert wird, wenn eine mit einem ≪assert≫ Stereotyp versehene Aktivität fehlschlägt. In [Gei02a] habe ich einfach die Aufrufe der `JavaSDM.ensure()` Methode durch Aufrufe der `Assert.assertTrue()` Methode des JUnit Frameworks getauscht. Mit diesem Ansatz konnten aber für geschachtelte Suchen keine aussagekräftigen Fehlermeldungen generiert werden, da nur die Information zur Verfügung stand, dass die Suche fehlgeschlagen ist. In solchen Fällen ist es aber auch interessant aus welchen Gründen die überprüften Kandidaten nicht geeignet waren. Daher habe ich in dieser Arbeit den Ansatz verwendet erst alle geworfenen `JavaSDMExceptions` einzusammeln und am Schluss daraus einen JUnit Fehler zu erzeugen (siehe auch Listing 12.2).

Listing 12.2 zeigt den mit diesem Ansatz aus dem Storydiagramm in Abbildung 12.8 generierten Quelltext. In den Zeilen 6 und 8 kann man erkennen, dass dem `JavaSDM.ensure()` Aufruf nun zusätzlich ein Kommentar als zweiter Parameter mitgegeben wurde. Dieser Kommentar wird als Fehlertext der erzeugten `JavaSDMException` benutzt, wenn eine solche ausgelöst wird.

In Zeile 12 wird jetzt eine Liste angelegt, die die Fehler für alle fehlgeschlagenen Kandidaten sammelt. Wird ein Kandidat verworfen, wird die zugehörige Exception in Zeile 25 dieser Liste hinzugefügt. Sollte kein passender Kandidat für das Objekt `t2_1` gefunden werden, wird in Zeile 30 zusätzlich zu der Fehlermeldung auch noch diese Liste an die `JavaSDM.ensure()` Methode übergeben.

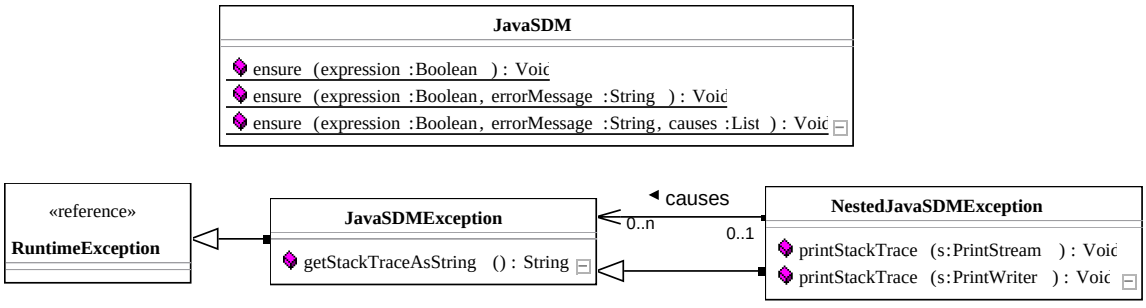


Abbildung 12.9.: Klassendiagramm der Klassen `JavaSDM` und `JavaSDMException`

Diese neue `JavaSDM.ensure()` Methode legt, wenn als erster Parameter `false` übergeben wird, eine neue Exception vom Typ `NestedJavaSDMException`, siehe Abbildung 12.9, an. Wie aus Abbildung 12.9 ersichtlich, kann eine solche Exception mehrere `JavaSDMExceptions` in der `causes` Kante speichern. Zusätzlich sind die `printStackTrace()` Methoden so überschrieben, dass sie beim Ausgeben des Stacktraces die Stacktraces der Exceptions, die in der `causes` Menge abgelegt sind, als eingeschachtelte Exceptions mit ausgeben. Durch den Aufruf aus Zeile 30 werden

also gegebenenfalls die Objekte in der Menge `causesS1ToT2_1` einer neu erzeugten `NestedJavaSDMException` als `causes` eingetragen.

Schlägt das Storypattern fehl, wird die entsprechende `JavaSDMException` in den Zeilen 37-40 schließlich in einen `AssertionFailedError` des JUnit Frameworks konvertiert und anschließend ausgelöst. Das führt dazu, dass das JUnit Framework diesen Test als fehlgeschlagen registriert.

```
1  public void assertStep2 () {
2     ...
3     try {
4        fujaba__Success = false;
5        // check object s1 is really bound
6        JavaSDM.ensure ( s1 != null, "check object s1 is really bound" );
7        // attribute condition init == true
8        JavaSDM.ensure ( s1.isInit () == true,
9           "attribute condition init == true" );
10       ...
11       fujaba__IterS1ToT2_1 = s1.iteratorOfIncomingTransitions ();
12       List causesS1ToT2_1 = new ArrayList<JavaSDMException>();
13       while ( !(fujaba__Success) && fujaba__IterS1ToT2_1.hasNext () ) {
14          try {
15             t2_1 = (Transition) fujaba__IterS1ToT2_1.next ();
16             // check link source from t2_1 to s2
17             JavaSDM.ensure (s2.equals (t2_1.getSource ()),
18                "check link source from t2_1 to s2");
19             // attribute condition label == "hang up"
20             JavaSDM.ensure ( JavaSDM.stringEquals (t2_1.getLabel (),
21                "hang up"), "attribute condition label == \"hang up\"" );
22             fujaba__Success = true;
23          }
24          catch ( JavaSDMException fujaba__InternalException ) {
25             causesS1ToT2_1.add (fujaba__InternalException);
26             fujaba__Success = false;
27          }
28       }
29       JavaSDM.ensure (fujaba__Success,
30          "iterate to-many link target from s1 to t2_1", causesS1ToT2_1);
31       // create link object_t2_1 from this to t2_1
32       this.setT2_1 (t2_1);
33       fujaba__Success = true;
34    }
35    catch ( JavaSDMException fujaba__InternalException ) {
36       fujaba__Success = false;
37       AssertionFailedError assertionFailedError = new AssertionFailedError
38          (fujaba__InternalException.getStackTraceAsString());
39       assertionFailedError.setStackTrace(new StackTraceElement[] {});
40       throw assertionFailedError;
41    }
42 }
```

Listing 12.2: Quelltext für die Beispieltestmethode unter Anwendung der Testtemplates

Abbildung 12.10 zeigt die Ausgabe des JUnit Eclipse Plugins, nachdem der Test für das Flattening Szenario in genau der hier als Beispiel gewählten `assertStep2()` Methode fehlgeschlagen ist. Im rechten unteren Abschnitt des dargestellten Views wird der Stacktrace des Fehlers angezeigt. Hier kann der Entwickler Details erfahren, wo und warum der Test fehlgeschlagen ist. Die erste Zeile des Stacktraces verrät, dass der Test mit der Fehlermeldung „iterate to-many link target from s1 to t2_1“ fehlgeschlagen ist. Es konnte also kein geeigneter Kandidat für `t2_1` gefunden werden. Die Zeilen im Stacktrace darunter geben an, wo es zu diesem Fehler kam. Ein Klick auf einen Eintrag springt zur entsprechenden Stelle im Quelltext. Weiter unten werden schließlich die eingeschachtelten Exceptions ausgegeben. Hier ist in Abbildung 12.10 zu erkennen, dass es nur einen Kandidaten gab und für diesen Kandidaten die Suche mit der Fehlermeldung „check link source from t2_1 to s2“ gescheitert ist. Das entsprechende Objekt hatte also keinen `source` Link zum Objekt `s2`. Im unteren Teil von Abbildung 18.17 aus Abschnitt 18.2.7 ist die Objektstruktur, die zum Fehlschlagen des Tests führte, als Objektdiagramm dargestellt.

Beim normalen Debuggen von Fujaba generiertem Quelltext trat des öfteren das Problem auf, dass man eine Suche schrittweise ausgeführt hat und die Suche an irgendeiner Stelle kein passendes Objekt finden konnte. Im Nachhinein herauszufinden, warum die Suche gescheitert ist, war dann nur sehr schwer möglich. Man musste meist die Debugsession wiederholen. Die hier vorgestellte Codegenerierung speichert aber genau diese Information, so dass sie auch später noch einfach analysiert werden kann. Daher ist es jetzt möglich die Codegenerierung in einen Debugmodus zu schalten, indem für jede Aktivität der Code nach dem oben vorgestellten Konzept generiert wird. Lediglich die Erzeugung der JUnit Exception (in Listing 12.2 die Zeilen 37-40) wird hier weggelassen. Auf diese Art generierter Code bietet nach unseren Beobachtungen eine deutliche Erleichterung beim Debuggen. Allerdings ist der so generierte Code auch deutlich langsamer, da normalerweise in der `JavaSDM.ensure()` Methode immer die selbe Exception ausgelöst wird. Das Anlegen von Exceptions, wie es hier der Fall ist, braucht in Java sehr viel Zeit. Daher sollte man für Produktivcode das Debugflag der Codegenerierung wieder abschalten.

Wie bereits erwähnt, ist die hier beschriebene Testcodegenerierung allein durch Anpassen der CodeGen2-Templates realisiert. Listing 12.3 zeigt exemplarisch das Template für den Check, ob ein gebundenes Objekt auch wirklich existiert (vgl. Zeile 6 in Listing 12.2). In Zeile 1 wird ein anderes Template importiert, das Variablen, wie zum Beispiel `$name` belegt. Die nächste Zeile ist für die Testgenerierung verändert. Hier wird nicht mehr direkt der Kommentar ausgegeben, sondern dieser erst der Variable `$opComment` zugewiesen und erst in der nächsten Zeile geschrieben. Die Variable `$opComment` wird sich auch noch für das Erstellen der Debuginformationen in Abschnitt 18.1.1 als hilfreich erweisen. In Zeile 5 werden dann, falls nötig, die zusätzlichen Parameter der `JavaSDM.ensure()` Methode hinzugefügt. Dies ist in ein weiteres Template ausgelagert, um es auch in anderen Templates wiederbenutzen zu können.

```
1 #parse("$lang/default:storyPattern/object/import.vm" )
2 #set( $opComment = "check object $name is really bound" )
3 // $opComment
```

```
4 JavaSDM.ensure ( $name != null##
5 #parse("$lang/default:storyPattern/sdmComment.vm") );
```

Listing 12.3: Template bound.vm

Listing 12.4 zeigt das Template, welches die zusätzlichen Parameter generiert. Hier wird zuerst überprüft, ob das Debugflag aktiviert oder der ≪`assert`≫ Stereotyp auf der zugehörigen Aktivität gesetzt ist. Ist dies der Fall, wird der in der Variable `$opComment` gespeicherte Kommentar als zusätzlicher Parameter angehängt. Hierbei wird er zuerst durch den Aufruf der Methode `java()` des in der `$esc` Variable gespeicherten Hilfsobjekt in einen Java kompatiblen String konvertiert. Hierbei werden zum Beispiel Anführungsstriche durch einen vorangehenden Backslash geschützt.

Ähnlich wie beim hier vorgestellten Check für gebundene Objekte wurden auch die Templates der anderen Überprüfungen, wie zum Beispiel die Überprüfung der Attributwerte, angepasst. Das Template zur Objektsuche für zu-1 Assoziationen konnte hierbei analog zu den beschriebenen Änderungen angepasst werden.

```
1 #if( $debug ||
2 $elem.firstFromDiagrams.parentElement.hasKeyInStereotypes("assert") )
3 , "$esc.java($opComment)"
4 #end
```

Listing 12.4: Template sdmComment.vm

Weitere Anpassungen mussten noch am Template zur Objektsuche von zu-n Assoziationen und dem Template für Storypattern vorgenommen werden. Listing 12.5 zeigt das Template für die Storypattern. Hier ist lediglich die Erzeugung des JUnit kompatiblen `AssertionFailedError` hinzugefügt. Die Erzeugung erfolgt in den Zeilen 11-17. Das Template für die Suche entlang zu-n Assoziation ist in Anhang A.2 abgedruckt. Geänderte Zeilen sind dort hervorgehoben.

```
1  // story pattern $!name
2  try
3  {
4     fujaba__Success = false;
5  $!children##
6     fujaba__Success = true;
7  }
8  catch ( JavaSDMException fujaba__InternalException )
9  {
10    fujaba__Success = false;
11 #if( $elem.revStoryPattern.hasKeyInStereotypes("assert") )
12 #set($r = $imports.addToImports("junit.framework.AssertionFailedError"))
13    AssertionFailedError assertionFailedError = new AssertionFailedError
14       (fujaba__InternalException.getStackTraceAsString());
15    assertionFailedError.setStackTrace(new StackTraceElement[] {});
16    throw assertionFailedError;
17 #end
18 }
```

Listing 12.5: Template storyPattern.vm

Die in diesem Abschnitt vorgestellte JUnit konforme Codegenerierung für Aktivitäten, die mit dem ≪`assert`≫ Stereotyp markiert sind, kann auch für Tests benutzt werden, die nicht aus Storyboards generiert sind. Der Entwickler kann also manuell eigene Testmethoden in Fujaba modellieren. Dazu muss er seine Testklasse lediglich von `junit.framework.TestCase` erben lassen, den Namen der Testmethode mit `test` beginnen lassen[1] und in dieser Methode an die Aktivitäten, die JUnit Überprüfungen modellieren den ≪`assert`≫ Stereotyp vergeben. Der daraus generierte Quelltext lässt sich dann mit dem JUnit Framework ausführen. Auf diese Weise ist das Spezifizieren von Tests auf Modellebene, also mit Fujaba Diagrammen, möglich. Dies ist insbesondere deshalb sinnvoll, da ja, wie bereits beschrieben, die Testfälle, die aus den Storyboards generiert werden, meist keinen kompletten Systemtest darstellen. Man sollte daher die Testsuite um manuell geschriebene Tests erweitern.

12.4. Eclipse Integration

Da der hier vorgestellte Mechanismus normale JUnit Testklassen generiert, gestaltet sich die Integration in Eclipse sehr einfach. Die JUnit Klassen werden mit dem Eclipse Compiler übersetzt und mit dem JUnit Plugin ausgeführt. Es wurden lediglich Kommandos hinzugefügt, die das Starten des generierten Tests auf dem Storyboard und das Starten der kompletten Testsuite auf dem Projekt erlauben. Abbildung 12.10 zeigt die Ausgabe des JUnit Plugins nach Ausführung des in diesem Kapitel besprochenen Testfalls.

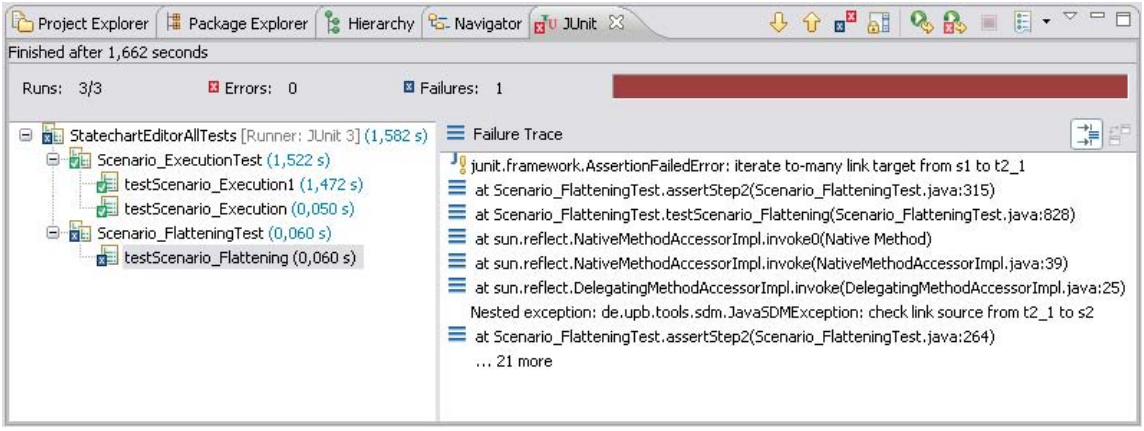


Abbildung 12.10.: Testausführung in Eclipse

12.5. Debugging Hilfen

Im Falle eines fehlgeschlagenen Tests wird mit der passenden Fehlermeldung und dem Exception-Stacktrace, wie in Abschnitt 12.3 beschrieben, schon eine sinnvolle

[1] Dies ist in der hier verwendeten Version 3 des JUnit Frameworks nötigt. In der aktuellen JUnit Version 4 ist es auch möglich Testklassen und -methoden durch Java Annotationen zu markieren.

Hilfe geliefert, den Grund des Fehlschlagens herauszufinden. Bei komplexen Objektstrukturen reicht jedoch manchmal eine rein textuelle Fehlermeldung nicht. Vielmehr wäre es hier hilfreich, die tatsächliche Objektstruktur mit der erwarteten vergleichen zu können. Hierzu wird der in Abschnitt 18.2 vorgestellte Objektbrowser eDOBS bereitgestellt.

Falls dies noch nicht reicht, um den Fehler zu identifizieren, kann der Entwickler den Test im Debugmodus neu starten, an interessanten Stellen Haltepunkte setzen und den Test ab dort schrittweise ausführen. Nach jedem Schritt werden Änderungen im eDOBS visualisiert. Das hierzu erforderliche Debuggen auf Modellebene wird im Abschnitt 18.1 vorgestellt.

Zusätzlich wird in Abschnitt 18.2.8 auch noch die Möglichkeit vorgestellt, die Änderungen an der Objektstruktur während eines Tests protokollieren zu lassen, um dann beliebig darin navigieren zu können. Wie die Testintegration dieser Verfahren geschieht, wird in den Abschnitten 18.2.7 und 18.2.8 näher beleuchtet. Insgesamt werden Werkzeuge bereitgestellt, die es ermöglichen im Fehlerfall die Fehlerursache schnell und einfach herauszufinden.

12.6. Coverage

In unseren Projekten konnten wir häufiger beobachten, dass zuerst Szenarien für den „normalen“ Programmdurchlauf erstellt, bei der Umsetzung in Storydiagramme dann aber häufig weitere Sonderfälle implementiert werden. In solchen Fällen existieren also Implementierungsteile, zu denen es kein Szenario, also auch keinen Test, gibt. In vielen dieser Sonderfälle wäre aber eine Rücksprache mit dem Kunden wünschenswert, um herauszufinden, ob sich die Applikation hier richtig verhält oder ob es sich gar um überflüssige Funktionalität handelt. Solche Implementierungsteile ohne Szenarien im Nachhinein manuell zu finden ist jedoch keine einfache Aufgabe. Hier wäre Toolunterstützung sinnvoll. Im Rahmen dieser Arbeit werden Coverage-Analysen herangezogen, um ungetestete Implementierungsteile und somit Teile, die in keinem Szenario vorkommen, zu finden. Es wird also die Abdeckung der aus Szenarien generierten Tests ermittelt. Die Abdeckung sollte hierbei natürlich auf Modellebene erstellt werden.

Um die Quelltextabdeckung der eigenen Testsuite zu analysieren, werden oft sogenannte Coverage-Tools verwendet. Diese instrumentieren den Quelltext so, dass sobald eine Quelltextzeile ausgeführt wird, diese Zeile in einem Protokoll als abgedeckt eingetragen wird. Um die Testabdeckung zu messen, wird dann die entsprechende Testsuite ausgeführt und anschließend das dabei erstellte Protokoll analysiert und übersichtlich (meist als Baumstruktur) dargestellt. Ziel dieses Kapitels ist es, einen solchen Coverage Mechanismus auch für modellbasierte Tests anzubieten.

Zur Bewertung von quelltextbasierter Abdeckung existieren im wesentlichen vier Kriterien (zur Berechnung der Abdeckung des Kontrollflusses) [Bal08]:

C0. Statement Coverage Die C0 Überdeckung fordert, dass jede Anweisung mindestens einmal ausgeführt wird. Damit ist sichergestellt, dass kein Code exi-

stiert, der nicht von einem Test erreicht wird.

C1. Branch Coverage In C1 Überdeckungstests müssen zusätzlich zur C0 Überdeckung alle Zweige durchlaufen werden. Das bedeutet, dass auch leere Blöcke von Fallunterscheidungen ausgeführt werden müssen. Schleifen müssen sowohl durchlaufen als auch übersprungen werden.

C2. Path Coverage C2 Überdeckungstests müssen alle möglichen Pfade durch eine Methode testen. C2 Überdeckung erfordert exponentiellen Aufwand und ist daher in der Praxis schwer zu erreichen.

C3. Condition Coverage Bei zusammengesetzten Bedingungen werden alle Möglichkeiten getestet (Beispiel: Für `if(a || b)` wird der Test mit `a == b == false`, `a == b == true`, `a == true; b == false` und `a == false; b == true` ausgeführt.). Der Aufwand um C3 Überdeckung zu erreichen ist ebenfalls sehr hoch.

Für modellbasierte Szenarien existieren noch keine solchen Überdeckungskriterien. Daher liegt es nahe, diese Überdeckungskriterien auf den modellbasierten Ansatz zu übertragen. Hierbei stellt sich zuerst die Frage, was dort als atomare Anweisung im Sinne der Statement Coverage angesehen werden kann. Betrachtet man eine komplette Aktivität im Storydiagramm als atomare Einheit, lassen sich die Kriterien eins zu eins übernehmen. C0 bedeutet so beispielsweise, dass jede Aktivität einmal ausgeführt wird, C1, dass jeder Zweig und C2, dass jeder Pfad im Storydiagramm durchlaufen worden ist. Diese Kriterien nenne ich im Folgenden Activity Coverage (AC0 - AC3).

Da Aktivitäten allerdings im Gegensatz zu normalen Quellcodeanweisungen neben der erfolgreichen Ausführung auch immer fehlschlagen können, lassen sich diese Kriterien noch erweitern. Statement Coverage bedeutet so erweitert, dass jede Aktivität einmal erfolgreich und einmal mit einem Fehlschlag durchlaufen werden muss. Die weiteren Kriterien lassen sich analog anpassen (extended Activity Coverage eAC0 - eAC3).

Eine solche Betrachtung der Aktivitäten als atomare Einheit berücksichtigt allerdings die einzelnen Operationen zur Graphsuche nicht. Existieren in einer Aktivität beispielsweise optionale Knoten, stellt selbst die Condition Coverage mit dem obigen Ansatz nicht sicher, dass ein optionaler Knoten einmal gefunden und einmal nicht gefunden wird. Hierfür müssen die einzelnen Operationen der Graphsuche als atomare Einheit verwendet werden. Die so entstehenden Coverage Kriterien bezeichne ich als Search Coverage (SC0 - SC3). SC0 Coverage bedeutet dann, dass jede Operation zur Graphsuche mindestens einmal ausgeführt wird. Das bedeutet, dass die Graphsuche zumindest einmal inklusive aller optionalen und Mengenknoten erfolgreich sein muss. SC1 bedeutet, dass jede Operation einmal erfolgreich und einmal nicht erfolgreich durchgeführt wird. SC2 heißt in diesem Ansatz, dass beispielsweise für optionale Objekte jede Kombination von „wird gefunden“ und „wird nicht gefunden“ existieren muss. SC3 muss dann zusätzlich auch für Constraints sämtliche Evaluierungsmöglichkeiten berücksichtigen.

Da ein Storydiagramm immer aus Kontrollfluss durch Aktivitäten und Graphensuche in den Aktivitäten besteht, lässt sich eine gute Abdeckung nur durch Kombination

der beiden Kriterien Activity Coverage und Search Coverage erreichen. Die ideale Abdeckung ist dann also eAC3 + SC3. Diese ist jedoch sehr aufwendig zu erreichen und dürfte daher nur in Einzelfällen zum Einsatz kommen.

Zusätzlich besteht ein Modell in Fujaba neben Aktivitätsdiagrammen auch noch aus Klassendiagrammen. Es liegt also nahe, auch für Klassendiagramme Abdeckung zu fordern. Ein Klassendiagramm erachte ich als komplett abgedeckt, wenn

- jede Blattklasse instanziiert und ein Objekt der Klasse wieder gelöscht wird.
- jede Assoziation abgedeckt ist (also angelegt, traversiert und gelöscht wird).
- jedes Attribut abgedeckt ist (lesender und schreibender Zugriff).
- jede Methode (inklusive der überschriebenen) aufgerufen wird.

Diese Kriterien zur Abdeckung der Klassendiagramme lassen sich auch wieder mit den vorherigen Kriterien kombinieren um eine möglichst gute Modellüberdeckung zu erreichen.

12.6.1. Technische Umsetzung der Modellüberdeckung

Da in dem in diesem Teil beschriebenen Ansatz zur Testgenerierung normale JUnit Tests generiert werden, können quelltextbasierte Coverage-Tools auch mit den generierten Tests eingesetzt werden. Allerdings wird bei diesen Tools die Abdeckungsstatistik auf Quelltextzeilen bezogen. Dieses würde einen Medienbruch darstellen, da sich der bisherige Prozess vollständig auf Modellebene abspielt. In dieser Arbeit wird daher der Ansatz verfolgt, bestehende Coverage-Tools zu verwenden, um einen Abdeckungsreport zu erzeugen, diesen dann jedoch wieder auf Modellebene abzubilden. Eine solche Abbildung der Quelltextüberdeckung auf Modellüberdeckung ist sinnvoll, da diese Überdeckungen korrelieren, wie in [BCSW03] gezeigt ist. Wie später gezeigt wird, kann durch dieses Vorgehen auch eine gute Abdeckung im Sinne der oben definierten Kriterien gewährleistet werden.

In dieser Arbeit wird das Opensource Coverage-Tool eclEmma [Hof09] eingesetzt. Bei eclEmma handelt es sich um die Eclipse Integration des weit verbreiteten Coverage-Tools EMMA [Rou05]. Um mit eclEmma einen Abdeckungsreport zu erstellen, muss lediglich die Testsuite über ein bestimmtes Kommando gestartet werden, und es wird automatisch die Suite ausgeführt und der erzeugte Abdeckungsreport in einer Baumansicht dargestellt. Zusätzlich lassen sich die erzeugten Abdeckungsinformationen leicht über eine API abfragen. Das hier beschriebene Verfahren zur Modellüberdeckung lässt sich einfach auf andere quelltextbasierte Coverage-Tools übertragen. eclEmma wurde hier wegen der Eclipse Integration und der einfach anzusprechenden API gewählt.

Um die von eclEmma erzeugten Informationen auf Modellebene abbilden zu können, ist es zuerst notwendig, herauszufinden, welche Modellelemente Quelltext generiert haben und welche Zeilen hierbei generiert wurden. Diese Information liefert der in Abschnitt 18.1.1 beschriebene Design Level Debugging Mechanismus. Hierbei wird ein Baum von `DLRToken` aufgebaut, wobei jedes `DLRToken` speichert, welche Zeilen

in welcher Datei von welchem Modellelement erzeugt worden sind (vergleiche Abbildung 18.3).

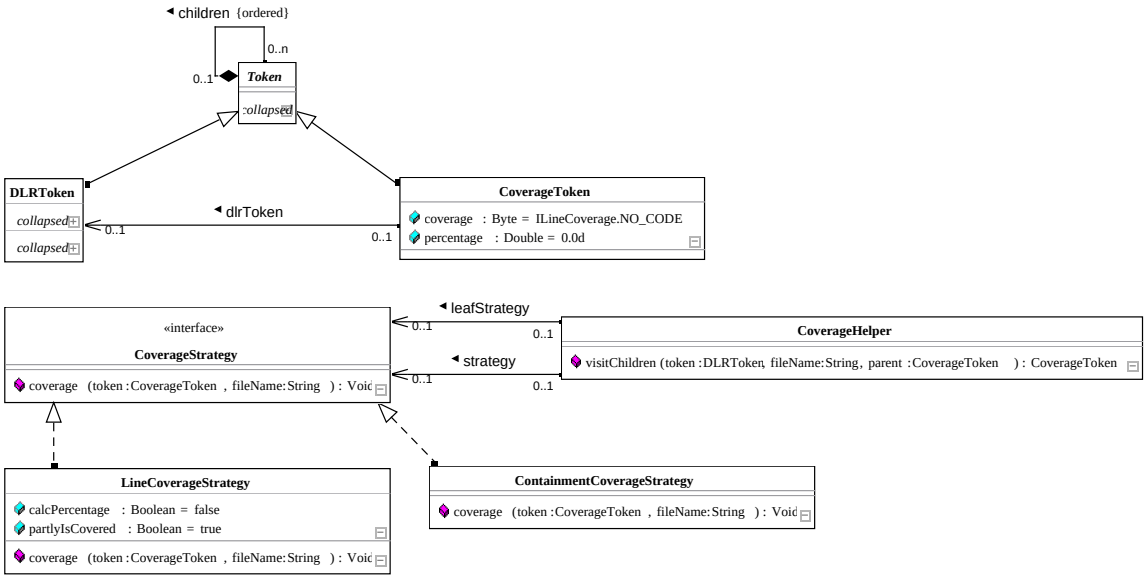


Abbildung 12.11.: Klassendiagramm des Coverage Mechanismus

Um einen Abdeckungsreport auf Modellebene zu erstellen, wird zu jedem `DLRToken` ein `CoverageToken` (siehe Abbildung 12.11) erstellt. Diese `CoverageToken` können im Attribut `coverage` ablegen, ob das zugehörige Element komplett, teilweise oder nicht abgedeckt wurde. Im Attribut `percentage` kann der prozentuale Anteil der Abdeckung abgelegt werden. Es existieren dann unterschiedliche Strategien, die entweder das `coverage` Attribut, das `percentage` Attribut oder auch Kombinationen zur Berechnung der Gesamtabdeckung heranziehen.

Die Erstellung des `CoverageToken`-Baums übernimmt ein Objekt der Klasse `CoverageHelper`, siehe Abbildung 12.11. Diese Helper-Klasse erzeugt zuerst den Baum und berechnet dann zu jedem `CoverageToken` die Abdeckung. Um unterschiedliche Verfahren zur Berechnung der Abdeckung verwenden und erproben zu können, ist diese Berechnung über das Strategy-Pattern in die Klasse `CoverageStrategy` ausgelagert. Ein `CoverageHelper` besitzt zwei solcher Strategien. Die erste ist über die Assoziation `leafStrategy` zu erreichen und ist ausschließlich für Blätter des Token-Baums zuständig. Die zweite Strategie wird über die `strategy` Assoziation erreicht und wird für alle anderen `CoverageToken` verwendet. Diese Strategie wird immer verwendet, falls keine `leafStrategy` gesetzt ist.

Abbildung 12.11 zeigt zwei Implementierungen dieser Strategien. Die `LineCoverageStrategy` schlägt die Abdeckung über die eclEmma API nach. Hierbei werden alle Quelltextzeilen, die dem aktuellen `CoverageToken` über das `DLRToken` zugeordnet sind, an das eclEmma Framework weitergereicht. Liefert eclEmma für mindestens eine Zeile, dass diese abgedeckt ist, wird das `coverage` Attribut des bearbeiteten `CoverageToken` auf „abgedeckt“ gesetzt. Wird nur eine

teilweise abgedeckte Zeile gefunden, wird das Attribut auf „teilweise abgedeckt“ gesetzt und sonst auf „nicht abgedeckt“. Ist das `calcPercentage` Attribut gesetzt, wird in `percentage` die Prozentzahl abgedeckter Zeilen in Bezug auf die Menge aller Zeilen, die das entsprechende Token generiert hat, abgelegt. Das Attribut `partlyIsCovered` gibt dabei an, ob teilweise abgedeckte Zeilen hierbei als abgedeckt mitgezählt werden sollen oder nicht.

Bildet man quelltextbasierte Coverage auf Modellebene ab, muss man sich entscheiden, was mit Modellelementen passiert, die mehrere Quelltextzeilen generiert haben, von denen jedoch nur einige abgedeckt sind. Die `LineCoverageStrategy` lässt sich, wie oben beschrieben, in diesem Punkt unterschiedlich konfigurieren. Als geeignet erwies sich der Ansatz, dass ein Modellelement immer dann abgedeckt ist, sobald eine aus diesem Element generierte Quelltextzeile abgedeckt ist. Dieser Ansatz wurde gewählt, da generierter Quelltext, insbesondere aus Fujaba generierter Code, viele zusätzliche Quelltextfragmente, wie beispielsweise zusätzliche Zugriffsmethoden oder zusätzliche Sicherheitsabfragen enthält, die man in selbstgeschriebenen Quelltext weglassen würde. Eine Überdeckung für jedes dieser Fragmente herzustellen, kann sehr aufwendig sein. Eine einfache zu-n Assoziation in Fujaba generiert beispielsweise schon sechs Zugriffsmethoden, die wiederum unterschiedliche Fälle (Parameter ist `null`, Gegenseite existiert nicht, etc.) enthalten. Den kompletten Quelltext einer solchen Assoziation abzudecken ist also sehr aufwendig und unter der Annahme, dass der Codegenerator hier fehlerfrei funktioniert, auch meist nicht besonders sinnvoll. Wird in solchen Fällen die prozentualen Abdeckung wie oben beschrieben berechnet, ist zusätzlich ohne Blick in den Quelltext nicht ersichtlich, warum ein bestimmtes Modellelement beispielsweise nur zu 50% abgedeckt ist. Diese Abdeckungsberechnung ist nach meinen Erfahrungen nur zum Erproben und Verbessern des Coverage Ansatzes sinnvoll.

Die zweite Implementierung der Klasse `CoverageStrategy` stellt die Klasse `ContainmentCoverageStrategy` dar. Diese Strategie errechnet die prozentuale Abdeckung des aktuellen `CoverageToken` aus den Abdeckungen aller Kinder. Die Abdeckung einer Klasse würde sich so beispielsweise aus den Abdeckungen aller in ihr enthaltenen Kinder, also Methoden und Attributen, berechnen.

Für Aktivitäten ist es sinnvoll für die komplette Abdeckung sowohl mindestens einen erfolgreichen Durchlauf, als auch mindestens einen Fehlschlag zu fordern. Zu diesem Zweck wurde die Klasse `ContainmentCoverageStrategy` um eine Sonderbehandlung für Aktivitäten erweitert. Zur Verdeutlichung betrachte ich die Abdeckung des Quelltext aus Listing 12.1 für die Aktivität, die den Code von Zeile 3 bis 32 generiert: Zur Berechnung der Überdeckung wird sichergestellt, das der Code, der `fujaba__Success` auf `true` setzt (Zeile 28) sowie der Code, der die `JavaSDMException` fängt (Zeile 30 bis 32), abgedeckt ist. Sollte eines der beiden Codefragmente nicht abgedeckt sein, ist auch die Aktivität nicht abgedeckt.

Abbildung 12.12 zeigt den Abdeckungsreport für die in dieser Arbeit behandelte Testsuite. Hierbei wurde für die Blatt-Strategy ein Objekt der Klasse `LineCoverageStrategy` verwendet, wobei das `calcPercentage` Attribut nicht gesetzt wurde. Die Standard-Strategy wurde durch ein Objekt der

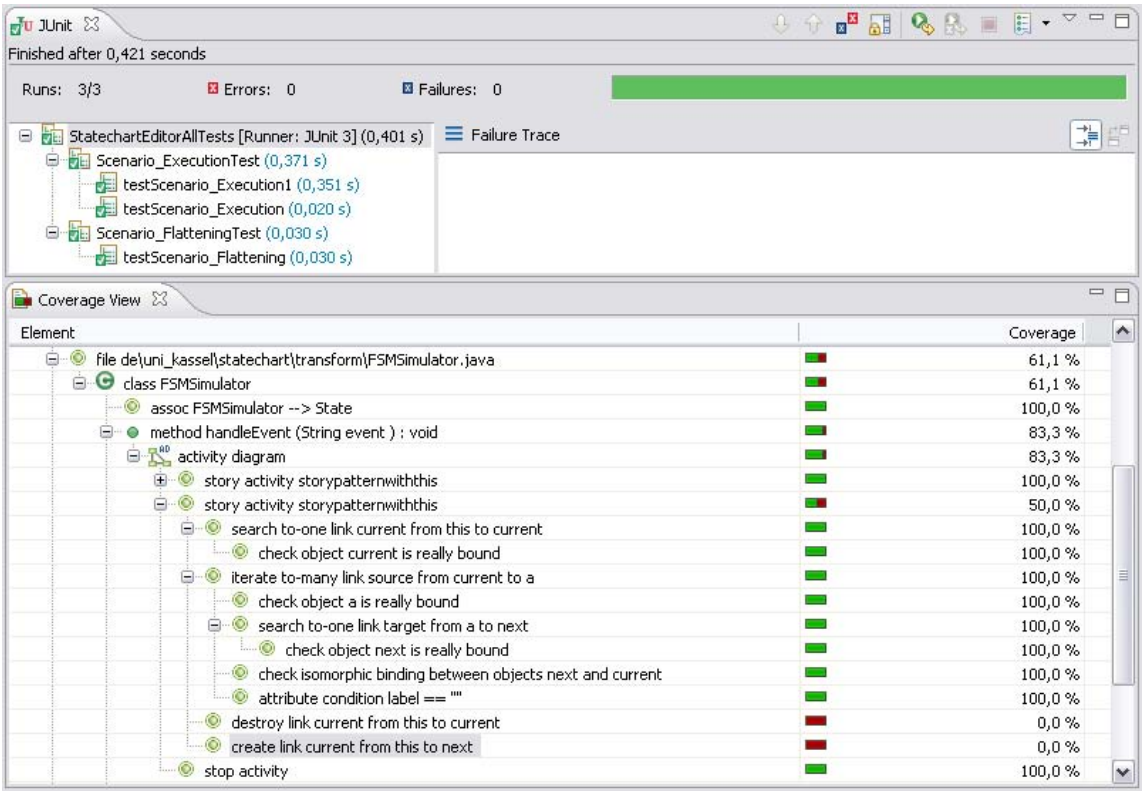


Abbildung 12.12.: Abdeckungsreport für die Beispiel-Tests

Klasse `ContainmentCoverageStrategy` gebildet. Zur Darstellung der einzelnen `CoverageToken` wird in dieser Baumansicht das `comment` Attribut des zugehörigen `DLRTokens` verwendet. Dieses Attribut liefert eine genau Beschreibung des generierten Codeblocks und wird während der Erzeugung der `DLRTokens` gesetzt. Dies ist genauer in Abschnitt 18.1.1 erläutert.

Im Abdeckungsreport in Abbildung 12.12 ist nun zu erkennen, dass das Implementierungsmodell der Klasse `FSMSimulator` nur zu 61,1% abgedeckt ist. Das bedeutet, dass das Szenario aus Abbildung 12.1, welches ja den Simulator und damit diese Klasse testet, nicht alle möglichen Fälle abdeckt. Insbesondere lässt sich aus Abbildung 12.12 erkennen, dass in der zweiten Aktivität der Methode `handleEvent()` der Klasse `FSMSimulator` eine Linkerzeugung und eine Linklöschung nicht abgedeckt sind. Führt der Benutzer nun zum Beispiel einen Doppelklick auf der nicht abgedeckten Linkerzeugung aus, wird in Fujaba4Eclipse das zugehörige Modellelement dargestellt. Dies ist der in Abbildung 12.13 blau unterlegte `current` Link in der zweiten Aktivität zwischen den Objekten `this` und `next`.

Die zweite Aktivität in Abbildung 12.13 dient dazu, nach Verarbeitung eines Events, eine vom aktuellen Zustand abgehende Transition mit leerem Label ebenfalls abzuarbeiten. Dies wird solange wiederholt, bis keine solche Transition mehr gefunden werden kann. Diese Aktivität wird laut Abdeckungsreport auch erreicht, einige darin enthaltene Elemente, wie beispielsweise die Suche nach dem `next` Objekt auch ausgeführt, aber die Linkerzeugung wird nie erreicht. Das lässt darauf schließen, dass

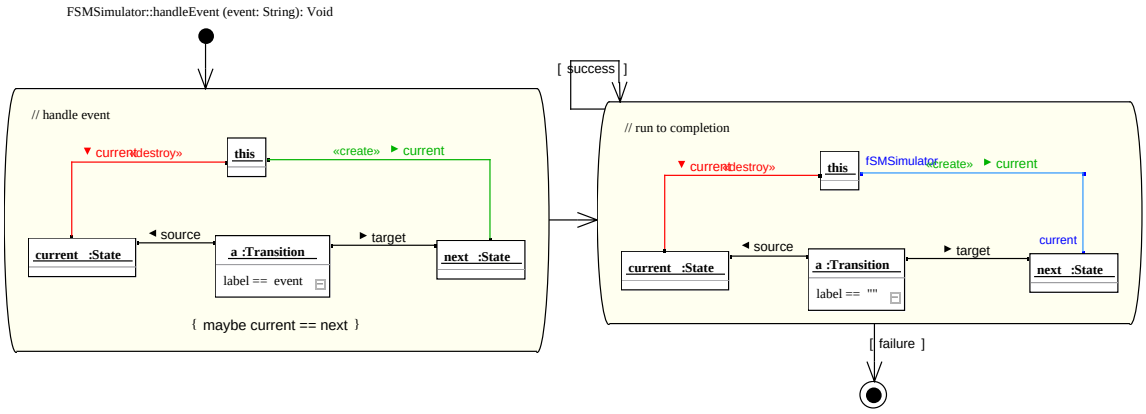


Abbildung 12.13.: Storydiagramm der Methode `handleEvent()`

kein Testfall existiert, in dem diese Aktivität erfolgreich ausgeführt werden kann. Es gibt also keinen Test, in dem leere Transitionen abgearbeitet werden. Betrachtet man den einzigen Testfall der Testsuite, der die Simulation behandelt (Abbildung 12.1) stellt man fest, dass solche Transitionen dort tatsächlich nicht auftauchen. Aufgabe des Entwicklers ist es also nun, ein Szenario zu spezifizieren, in dem leere Transitionen durch den Simulator verarbeitet werden müssen. Durch den modellbasierten Coverage Mechanismus ist es also möglich, Fälle in der Implementierung zu finden, die noch durch kein Szenario abgedeckt sind.

Der hier vorgestellte, auf quelltextbasierten Abdeckungsanalysen, basierende Ansatz zur Modellüberdeckung, stellt sicher, dass alle Modellelemente der Graphensuche, also alle Operationen, mindestens einmal überdeckt sind. Es handelt sich also um SC0. Auf Aktivitätsebene wird ebenfalls sichergestellt, dass jede Aktivität mindestens einmal durchlaufen wird. Hierbei wird sowohl eine Überdeckung des Quelltextes gefordert, der für das erfolgreiche Durchlaufen der Aktivität verantwortlich ist, als auch des Codes, der das Fehlschlagen verursacht. Es handelt sich also um eAC0. Zusätzlich wird sichergestellt, dass sämtlicher aus dem Klassendiagramm generierter Quelltext, wie Zugriffsmethoden für Attribute und Assoziationen abgedeckt wird. Es handelt sich also um eine Abdeckung von eAC0 + SC0 + Klassendiagrammüberdeckung. Diese Überdeckung lässt sich, wie in diesem Kapitel gezeigt, technisch einfach umsetzen, ist effizient einsetzbar und reicht dennoch um Szenarien, die in der Implementierung enthalten sind, für die aber noch kein Storyboard existiert, zu ermitteln. Das Ziel einen Mechanismus zur Identifizierung von Implementierungsteilen ohne Szenario zu entwickeln, wurde also erreicht. Weiterführende Arbeiten können sich dann mit der Frage beschäftigen, in wie weit es möglich ist, für solche Szenarien automatisch Storyboards zu generieren, die das bisher nicht getestete Verhalten abdecken. Diese Storyboards können dann dem Kunden vorgelegt werden, damit dieser entscheiden kann, ob dieses Szenario ein gewünschtes ist, oder ob ein Fehler in der Implementierung vorliegt. Weiterhin kann genauer untersucht werden, in wie weit sich die hier vorgestellten Überdeckungskriterien als allgemeine Modellüberdeckungskriterien eignen.

13. Einsatz

Der hier beschriebene Ansatz zur Generierung von Testfällen aus Szenarien wurde in mehreren Studentenprojekten mit insgesamt ca. 350 Studenten erfolgreich eingesetzt. In Anhang A.1 ist eine Liste mit den Kursen aufgeführt, in denen unter anderem die Testgenerierung zum Einsatz kam. Während dieser Kurse konnten wir die folgenden Beobachtungen machen.

Da im hier beschriebenen Ansatz Szenarien in Form von Storyboards zur Testgenerierung verwendet werden, verlangt der Prozess sehr detaillierte Szenarien von den Studenten. Wir betrachten das aber eher als Vorteil denn als Nachteil. Ohne die nachfolgende Testgenerierung konnten wir in früheren Studentenprojekten beobachten, dass die Analyseszenarien unserer Studenten oft wichtige Details vermissen ließen. In diesen früheren Projekten wurde oft nur ein sehr kleiner Ausschnitt der wichtigen Entwurfsaspekte der zu entwickelnden Applikation in der Analysephase bedacht. Wichtige Entscheidungen bezüglich Struktur und Verhalten der Applikation wurden in der Regel erst spät in der Design- oder in der Implementierungsphase getroffen. Mit dem neuen Testgenerierungsmechnismus wurden die Studenten in den neueren Projekten gezwungen, Objektstrukturen zu entwerfen, die die wichtigsten Aspekte der Beispielapplikation bereits abdeckten. Hierdurch kamen viel früher Diskussionen über wichtige Design-Entscheidungen zustande und somit wurden mögliche Probleme frühzeitig identifiziert. Zu beobachten war außerdem eine Qualitätsverbesserung der Szenarien und der Analysedokumente.

Zusätzlich konnten wir beobachten, dass unsere Studenten motivierter waren, Arbeit in den Entwurf von Szenarien zu stecken, da sie wussten, dass sich dies später durch automatisch generierte Tests bezahlt machen würde.

Man beachte, dass es, obwohl in den Szenarien ein Detailgrad verlangt wird, der das Generieren von Quelltext ermöglicht, dennoch möglich ist, die Modellierung auf einem sehr hohen Abstraktionslevel vorzunehmen. Die Szenarien müssen lediglich die relevanten Teile der Applikationsdomäne modellieren. Implementierungsdetails sind hier nicht erforderlich. Insbesondere ist die Realisierung der im Szenario beschriebenen (und damit getesteten) Funktionalität oft nur angedeutet. Es werden also normalerweise nur Tipps zur Umsetzung dieser Funktionalität gegeben. In unseren Studentenprojekten wurden die zusätzlichen Implementierungsdetails daher auch erst in der Design- oder Implementierungsphase hinzugefügt. In diesen Phasen wurden also zum Beispiel komplexe Algorithmen oder Probleme mit der graphischen Benutzerschnittstelle angegangen.

All unsere Studentenprojekte verfolgten den in Teil I vorgestellten Fujaba Process. Dieser wird wesentlich durch die aus den Szenarien generierten Tests getrieben. Das heißt, unsere Studenten generierten erst JUnit Tests und begannen dann die in den Szenarien beschriebene Funktionalität Schritt für Schritt umzusetzen. Hierbei

dienten die Tests ihnen als ausführbare Überprüfungen der Anforderungen. Unserer Ansicht nach hat auch dieses Vorgehen sehr gut funktioniert. Unsere Studenten arbeiteten sehr fokussiert an ihrem Design und ihrer Implementierung, wobei der Versuch Testfälle zur erfolgreichen Ausführung zu bringen einen äußerst motivierenden Faktor darstellte. Wir hatten zunächst Bedenken, dass die Studenten sehr spezielle Implementierungen erstellen könnten, die lediglich die modellierten Szenarien abdecken und den allgemeinen Fall unberücksichtigt lassen. Dies kam jedoch nur sehr selten und wenn, dann nur gegen Ende der Projekte vor, wenn die entsprechende Gruppe unter Zeitnot geriet. Meist implementierten die Studenten jedoch Funktionalität, die auch den generellen Fall abdeckte (oder es zumindest versuchte), also alle möglichen Szenarien und nicht nur die spezifizierten.

Die erhoffte Konsistenz zwischen Analyseszenarien und Design- und Implementierungsartefakten wurde in den Studentenprojekten erreicht. Nach unseren Erfahrungen stellen die Szenarien mit Testgenerierung eine weitaus bessere Dokumentation des implementierten Systems dar als die Szenarien in früheren Projekten ohne Testgenerierung. Das liegt zum einen daran, dass aufgrund des Mehrwerts der Szenarien mehr Arbeit in diese investiert wird und zum anderen daran, dass bei Änderungen der Implementierung, die ein Szenario betreffen, auch das Szenario angepasst werden muss, da sonst der Test nicht mehr kompiliert oder durchläuft.

Neben den angesprochenen positiven Beobachtungen konnten wir aber durchaus auch noch Verbesserungsmöglichkeiten in den studentischen Projekten entdecken. Zuerst mussten wir feststellen, dass das Erstellen von geeigneten Storyboards einen nicht trivialen Lernprozess darstellt. Unsere Studenten brauchten einige Zeit und einige Versuche um herauszufinden, wie die Startsituation und die nachfolgenden ≪`actor steps`≫ zu modellieren sind, um den Anforderungen der Testgenerierung zu genügen. Ein häufiger Fehler war beispielsweise, dass die Startsituation noch nicht alle nötigen Objekte, Links und Attributzuweisungen enthielt, die für die problemlose Ausführung der im Szenario beschriebenen Methode nötig waren. Die Studenten neigten dazu, diese Objekte erst in den ≪`system steps`≫ einzuführen, wo sie das erste Mal benutzt wurden. Dieses Problem wurde von uns auf zwei Arten angegangen. Zum einen werden diese Aspekte jetzt intensiver in der begleitenden Vorlesung behandelt, zusätzlich wurde Fujaba auch dahingehend erweitert, dass es per einfachem Kommando ein später eingeführtes Objekt der Startsituation hinzufügen kann.

Zusätzlich konnten wir bei unseren Studenten Unsicherheiten im Umgang mit der zugrunde liegenden graphischen Modellierungssprache feststellen. So wurden Konstrukte wie ≪`create`≫ und ≪`destroy`≫ Stereotypen, gebundene Objekte usw. bei einigen Studenten manchmal falsch in den Storyboards eingesetzt. Diese Fehler wurden dann erst bei der Testgenerierung, beim Kompilieren des generierten Codes, oder in einigen Fällen sogar erst bei der Testausführung bemerkt. In diesen Fällen konnten sich die Studenten nicht sicher sein, ob der Test wegen einer fehlerhaften Implementierung oder eines solchen Fehlers im Szenario fehlgeschlagen ist. Um dieses Problem abzumindern, verfolgen wir den Ansatz Fehler im Szenario möglichst schnell dem Entwickler zu melden. So generiert das *Inspections-Plugin* bei jeder Modelländerung für die betroffene Komponente im Hintergrund Quelltext und zeigt dabei auftretende Warnungen und Fehler dem Benutzer im *Problems View* an.

Fujaba besitzt keinen Parser für Java-Ausdrücke, wie beispielsweise Methodenaufrufe oder Constraints. Daher werden Fehler wie falsch geschriebene Methodennamen erst vom Compiler erkannt. Um solche Fehler, die der Codegenerator nicht bemerkt, auch frühzeitig melden zu können, wäre ein Kompilieren der generierten Sourcen im Hintergrund nötig. Dies ist allerdings nicht realisiert worden, da bereits damit begonnen wurde Fujaba um einen Parser für Methodenaufrufe etc. zu erweitern und dieser dann die Fehlermeldungen produzieren kann.

14. Verwandte Arbeiten

Die Idee aus Storyboards JUnit Testfälle abzuleiten wurde von unterschiedlichen Ideen im Bereich der Statechartsynthese aus Szenarien inspiriert. In diesem Kontext existieren eine Menge Arbeiten, die versuchen durch die Analyse von Szenarien, die zum Beispiel als Sequenzdiagramme vorliegen, Statecharts für die beteiligten Objekte zu synthetisieren, die mindestens das in den Szenarien beschriebene Verhalten abbilden, siehe [USZ02, EGK+03, GK04, BE05, WGM06]. Diese Ansätze benutzen auch fein granulare Beziehungen zwischen den Diagrammelementen der Szenarien und denen der generierten Statecharts. Ein ähnliches Vorgehen haben wir in [DGMZ02] ebenfalls im Fujaba Kontext erprobt. Nach unseren Erfahrungen benötigt man allerdings sehr gut ausgearbeitete und detaillierte Szenarien um eine sinnvolle Verhaltensspezifikation automatisch ableiten zu können. Zusätzlich müssen die Szenarien oft noch mit Annotationen angereichert werden, die beispielsweise beschreiben in welchem Zustand sich das System gerade befindet. Solche Szenarien wirken nicht selten künstlich und extra für den speziellen Algorithmus der Verhaltenssynthese konstruiert. Daher haben wir uns entschlossen nicht das komplette Verhalten aus den Szenarien zu generieren, sondern lediglich Tests, die überprüfen, ob der Teil des Verhaltens, den das Szenario beschreibt, auch von der Implementierung abgedeckt wird.

Es gibt mehrere andere Ansätze, die sich mit der Generierung von Tests aus Szenarien beschäftigen. Der Rational Quality Architect [RQA02] zum Beispiel benutzt Sequenzdiagramme zur Modellierung von Szenarien. Aus diesen Sequenzdiagrammen werden Testfälle und Testtreiber generiert. Während eines Testlaufs wird der Signalfluss protokolliert. Dieses Protokoll wird in ein Sequenzdiagramm konvertiert, was wiederum mit dem Originalsequenzdiagramm verglichen wird. Sequenzdiagramme eignen sich vor allem zur Analyse von Applikationen mit komplexem Signalfluss, wie zum Beispiel für Kommunikationsprotokolle. Für die Modellierung von Applikationen mit komplexen Datenstrukturen eignen sich unserer Meinung nach Storydiagramme eindeutig besser. So lassen sich beispielsweise komplexe Nachbedingungen, wie in der *result situation* in Abbildung 5.4, nur sehr schwer und unübersichtlich mit Sequenzdiagrammen darstellen. Eher würde man OCL Ausdrücken verwenden, um komplexe Nachbedingungen zu spezifizieren. [LLQC07] beschreibt einen Ansatz in dem aus Sequenzdiagrammen und OCL Ausdrücken, mit denen Vor- und Nachbedingung beschrieben werden, automatisch Tests generiert werden. Die Endsituation des Beispielszenarios, die durch die letzte Aktivität in Abbildung 5.4 modelliert wird, wird in Listing 14.1 als OCL Ausdruck beschrieben. Wie die konkrete Objektsituation am Ende des Szenarios aussieht, lässt sich hieraus nur sehr schwer erkennen. Unseren Erfahrungen nach sind Storydiagramme oft sehr viel einfacher zu lesen als längere OCL Ausdrücke.

```
1  context TestCase::testStep()
2  post: f1.stateChart=sc and sc.elements->includes(s1) and
3  sc.elements->includes(s2) and sc.elements->includes(s3) and
4  s1.name="Idle" and s1.init=true and s2.name="Dialtone" and
5  s2.init=false and s3.name="Dial" and t1.source=s1 and
6  t1.target=s2 and t1.label="lift" and t3.source=s2 and
7  t3.target=s3 and t3.label="dial" and
8  s2.outgoingTransitions->exists(t2_1_i|t2_1_i.target=s1 and
9  t2_1_i.label="hang up") and
10 s3.outgoingTransitions->exists(t2_2_i|t2_2_i.target=s1 and
11 t2_2_i.label="hang up")
```

Listing 14.1: OCL Ausdruck zum Prüfen der Endsituation

Allerdings ist es auch möglich, OCL Ausdrücke in Storydiagramme einzubetten. In einer von mir mitbetreuten Studienarbeit an der TU Dresden wurde die OCL Engine Dresden OCL Toolkit in Fujaba integriert [SZG07].

Die SCENT Methode [RG00] erzeugt Statecharts aus textuellen Szenarien. Dann werden Pfade durch das erzeugte Statechart gesucht, die verschiedenen Pfadabdeckungskriterien gehorchen. Diese Pfade werden als Testfälle verwendet. Dieser Ansatz bietet jedoch auch keine Unterstützung für komplexe Objektstrukturen. Allerdings werden im FUP bisher nur sequentielle Abläufe als Szenarien betrachtet. Die Verschmelzung vieler Szenarien zu einem „Sytemstatechart“ wird bislang nicht betrachtet. Es könnte also durchaus Sinn machen, die im SCENT Ansatz verwendeten Techniken zur Pfadanalyse auch auf Storyboards anzuwenden. Auf diese Art könnten Pfade durch die Implementierung gefunden werden, für die bislang noch keine Storyboards existieren. Dieses Vorgehen wurde in dieser Arbeit nicht mehr behandelt.

Der in [BL01] vorgestellte TOTEM Ansatz benutzt auch Sequenzdiagramme zur Modellierung von Szenarien. In diesem Ansatz werden auch Beziehungen zwischen Szenarien berücksichtigt, wie sie in Usecase Diagrammen modelliert werden können. Wie in Abschnitt 12.2.1 diskutiert, werden im FUP solche Inter-Szenario Beziehungen entweder implizit durch Methodenaufrufe im Storyboard oder durch verzweigende Storyboards modelliert. Zur Testgenerierung werden hier nur die ≪`extends`≫ Beziehungen, also die verzweigenden Szenarien, verwendet.

In [OA00] werden ähnlich zum hier beschriebenen Vorgehen Kollaborationsdiagramme zur automatischen Testgenerierung verwendet. In diesem Ansatz werden die Kollaborationsdiagramme dazu verwendet, die Ausführung einer Operation zu beschreiben. Aus dieser Beschreibung wird ein sogenannter *Message Sequence Path*, also ein Protokoll des Nachrichtenflusses zwischen den Objekten, generiert. Dieses Protokoll wird dann mit echten Traceinformationen verglichen, die während der Ausführung des zugehörigen Tests erstellt wurden.

In [Böc06] wird ein Ansatz vorgestellt, wie für eingebettete Systeme Tests entwickelt werden können. Es wird ebenfalls Fujaba zum Modellieren von Testszenarien verwendet. In [Böc06] werden allerdings die Testmethoden manuell als Storydiagramme spezifiziert. Es wird also keine Testgenerierung verwendet. Es wird eine geeignete

Schnittstelle bereitgestellt, die die Funktionalität des JSystem Testframeworks in Fujaba anbietet. Durch das Aufrufen von Methoden dieser Schnittstelle aus den Storydiagrammen (beispielsweise der `fail()` Methode, die ein Fehlschlagen des aktuellen Tests auslöst) werden die Tests implementiert. Für diesen Ansatz ist also keine angepasste Codegenerierung nötig. Allerdings ist das Ausgeben von aussagekräftigen Fehlermeldungen in diesem Ansatz auch sehr aufwändig. Eine komplexe Überprüfung müsste hierbei in viele einzelne Überprüfungen zerlegt und für jeden Fehlschlag ein Aufruf der `fail()` Method mit passender Fehlermeldung implementiert werden.

15. Fazit

In diesem Teil wurde ein Konzept vorgestellt, das es erlaubt aus Analyseszenarien Tests zu generieren. Diese Tests prüfen, ob die Implementierung das modellierte Szenario enthält. Zusätzlich wurde auf Hilfestellungen zum Finden der Fehlerursache eingegangen und Techniken zur modellbasierten Abdeckungsanalyse entwickelt.

Nochmals sei erwähnt, dass Storyboards keine generelle Testmethode darstellen. Sie treiben die Spezifikation des Verhaltens gemäß des Test-first Prinzips der agilen Prozesse voran, können aber keine komplette Testabdeckung garantieren. Hierzu müssen meist noch zusätzliche Tests spezifiziert werden, was mit der hier vorgestellten Codegenerierung jedoch auch komplett modellbasiert erfolgen kann. Wir haben die Beobachtung gemacht, dass erfahrene Entwickler weniger Storyboards mit weniger Schritten produzieren als Anfänger. Diese Schritte decken allerdings meist die spannenden Fälle ab. In diesen Fällen werden Storyboards also gezielt zum Testen von interessanten Fällen eingesetzt.

Die Testgenerierung wurde für das Fujaba CASE Tool als Eclipse Plugin implementiert. Die Java Codegenerierung von Fujaba wurde so angepasst, dass ausführbare JUnit Testklassen aus den Szenarien generiert werden. Das vorgestellte Konzept ist jedoch nicht auf Java und JUnit beschränkt. Die hier vorgestellten Ansätze eignen sich ebenso für andere Zielsprachen und Testplattformen. Soll der vorgestellte Ansatz auf eine andere Zielsprache übertragen werden, müssen in der Regel nur die Templates angepasst werden. Da die eigentliche Testgenerierung lediglich Klassen und Storydiagramme aus Storyboards aber keinen Quelltext generiert, muss hier nichts oder nur wenig geändert werden. Die Codegenerierung aus den generierten Storydiagrammen erfolgt durch neue beziehungsweise angepasste Templates für die Zielsprache. Sollen anderen Testplattformen unterstützt werden, müssen zusätzlich zu Änderungen in den Templates gegebenenfalls andere Klassen und Methoden während der Testgenerierung erzeugt werden, um zu der gegebenen Test-API kompatibel zu sein. Aber auch solche Änderungen sind in der Regel ohne viel Aufwand möglich.

Der vorgestellte Ansatz ist auch nicht auf das Fujaba CASE Tool oder den Fujaba Process eingeschränkt. Man könnte beispielsweise die Storyboards auf Papier oder Whiteboard entwerfen und die Tests manuell daraus ableiten, also die Tests von Hand implementieren. Hier verliert man natürlich die automatische Synchronisation. Diese müsste dann ebenfalls manuell erfolgen. Ein solches Vorgehen wurde in dem bereits erwähnten Industrieprojekt mit der Firma Krauss-Maffei Wegmann durchgeführt [Czo04]. In diesem Projekt wurden Storyboards zur Analyse eingesetzt. Da Fujaba aber keine C# Codegenerierung bietet, konnte auch die automatische Testgenerierung nicht eingesetzt werden. Die Tests mussten also manuell aus den Szenarien abgeleitet werden. Trotz dieses Mehraufwandes wurde das Vorgehen von den KMW Mitarbeitern als sehr hilfreich eingestuft.

Um die automatische Testgenerierung für andere UML Werkzeuge zu implementieren, braucht das entsprechende Werkzeug gute Unterstützung für Objektdiagramme. Da dies in den meisten Werkzeugen nicht der Fall ist, würde eine Integration hier einen größeren Aufwand bedeuten, um die für Storyboards nötige Mächtigkeit der Objektdiagramme zu erreichen.

Die hier vorgestellte Testgenerierung konnte mit lediglich 2.000 Quellcodezeilen realisiert werden. Zusätzlich wurden die in diesem Teil und in Anhang A.2 beschriebenen Änderungen an den Codegenerierungstemplates vorgenommen. Alle Templates wurden so angepasste, dass die `$opComment` Variable mit einem, das zugehörige Template beschreibenden, geeigneten Wert belegt wird (siehe auch Abschnitt 18.1.1). Weiterhin wurden dem Fujaba CASE Tool sechs neue Stereotypen hinzugefügt.

Es wurde also ein Verfahren entwickelt, das durch Testgenerierung die Fehlerhäufigkeit bei der Softwareentwicklung verringert und somit die Qualität der fertigen Applikation erhöht. Da die Anforderungsszenarien in diesem Ansatz zur Testgenerierung heran gezogen werden, erlangen sie auch mehr Aufmerksamkeit im Prozess. Hierdurch steigt zusätzlich die Qualität der Anforderungsdokumentation. Die hier vorgestellte Testgenerierung trägt also einen wichtigen Teil zur Qualitätsverbesserung in Softwareprozessen bei.

Teil III.

Debuggen auf Modellebene

16. Motivation

Teil I dieser Arbeit beschreibt einen komplett modellbasierten Softwareentwicklungsprozess. Dieser beinhaltet die in Teil II im Detail beschriebene Methodik zur Testgenerierung aus Szenariomodellen. Der Quellcode für Struktur, Verhalten und Tests wird im vorgestellten Prozess komplett aus dem Modell generiert. Für den Entwickler ist es also zu keiner Zeit nötig die Modellebene zu verlassen und direkt Quellcode zu schreiben oder zu verstehen. Dieses Vorgehen erhöht das Abstraktionslevel bei der Anwendungsentwicklung und erhöht somit unseren Beobachtungen nach die Produktivität und verringert die Fehlerwahrscheinlichkeit.

Allerdings können bei der modellbasierten Entwicklung natürlich trotzdem Fehler auftreten. Solche Fehler können auf unterschiedliche Weise sichtbar werden:

Modellinkonsistenzen Die Konsistenzanalyse des Modellierungstools meldet einen Fehler. Beispielsweise könnte eine Klasse nicht eindeutig benannt sein. Enthält das Modellierungstool eine entsprechende Überprüfung kann es den Entwickler schon zur Modellierzeit auf solche Fehler aufmerksam machen. Fehlerauftreten, -meldung und -behebung finden also allesamt im Modellierungswerkzeug statt noch bevor Quelltext generiert wird.

Compilerfehler Beim Kompilieren der generierten Sourcen treten Fehler auf. Dies kann zum Beispiel passieren, wenn externe Bibliotheken falsch benutzt werden, etwa wenn die Parametertypen beim Aufruf einer externen Methode nicht stimmen. Solche Fehler werden vom Fujaba CASE Tool durch Konsistenzanalyse zur Zeit noch nicht erkannt. In die selbe Kategorie fallen Fehler oder Warnungen die von externen Tools generiert wurden, die direkt auf dem Quelltext arbeiten, wie zum Beispiel FindBugs [HP04].

Fehlschlagende Tests Ein aus einem Szenario generierter Test schlägt fehl. Dies wird üblicherweise durch das JUnit Framework angezeigt und bedeutet, dass das zugehörige Szenario nicht von der Verhaltensimplementierung abgedeckt wird. Die Ursachen hierfür können in einem fehlerhaft modellierten Szenario oder in einer fehlerhaften oder unvollständigen Implementierung liegen.

Fehler im Verhalten Die Applikation zeigt nicht das gewünschte Verhalten. Es kommt zum Beispiel zu Programmabstürzen durch nicht abgefangene Exceptions. Nach der Lehre der Softwaretechnik sollte in einem solchen Fall ein Test geschrieben werden, der diesen Fehlerfall reproduziert. Somit würde ein solcher Fehler auch wieder durch einen fehlschlagenden Test angezeigt werden.

Fehler im Timing Die Applikation zeigt nicht das gewünschte Zeitverhalten. So könnten Antworten beispielsweise zu langsam oder nicht richtig synchronisiert gegeben werden oder Deadlocks auftreten. Diese Kategorie von Fehlern wird in dieser Arbeit nicht betrachtet.

Tritt beim Kompilieren ein Fehler auf, gibt der Compiler eine Fehlermeldung, die Klasse, in der der Fehler aufgetreten ist, sowie die zugehörige Zeilennummer zurück. In integrierten Entwicklungsumgebungen, wie Eclipse, ist es möglich durch Selektion der Fehlermeldung direkt zur Fehlerursache in der entsprechenden Datei zu springen. Im hier vorgestellten modellbasierten Ansatz landet der Entwickler so aber im generierten Quelltext, den er im schlimmsten Fall noch nicht einmal versteht. Er muss nun versuchen, die passende Stelle im Modell zu finden, die diese Quelltextzeile generiert hat. Wünschenswert ist es aber, dass automatisch zur Fehlerursache im Modell gesprungen wird. Hierzu muss eine Abbildung zwischen Modell und Quelltext existieren, die die Navigation zum Modell erlaubt.

Im Falle eines fehlgeschlagenen Tests liefert das JUnit Framework eine Fehlermeldung und einen Stacktrace der fehlgeschlagenen Überprüfung zurück. Die Verständlichkeit und der Modellbezug der Fehlermeldung wird im Falle des aus einem Szenario generierten Test, wie in Kapitel 12.3 beschrieben, sichergestellt. Allerdings enthält auch der Stacktrace quelltextbezogene Zeilennummern. Diese ließen sich mit dem im vorigen Absatz erwähnten Quelltext-Modell Mapping zur Navigation ins Modell benutzen. Allerdings ist die Information welche Überprüfung nicht erfolgreich war, zur genauen Analyse des Fehlers normalerweise nicht ausreichend. Der Entwickler weiß dann zwar, dass die gewünschte Endsituation nicht erreicht wurde und auch, dass zum Beispiel ein bestimmtes Objekt in der Objektstruktur nicht gefunden wurde, aber es stellt sich oft die Frage: Wie sah die Objektstruktur nach Durchlauf des Tests aus? In solchen Fällen wäre ein Werkzeug hilfreich, das in der Lage ist, die momentane Objektstruktur zu einem bestimmten Zeitpunkt der Ausführung auf Modellebene zu visualisieren.

Durch einen solchen Objektbrowser, ließe sich die Wirkung eines Fehlers bereits sehr gut auf Modellebene analysieren. Dies ist für den Entwickler eine gute Hilfe, um die Ursache des Fehlers einzugrenzen. Allerdings sind zum genauen Eingrenzen der Fehlerursache in den meisten Fällen noch zusätzliche Schritte nötig. Ein häufig eingesetztes Verfahren ist hier das sogenannte Debuggen. Hier wird die Ausführung eines Programms an einer „verdächtigen“ Stelle angehalten und von da an schrittweise ausgeführt um den Fehler weiter einzugrenzen. Existierende IDEs bieten häufig Unterstützung für textuelles Debuggen. Auch hier wäre allerdings ein Debuggen auf Modellebene wünschenswert, also ein schrittweises Ausführen der Storydiagramme.

Allerdings benötigt man zum erfolgreichen Debuggen immer einen geeigneten Startpunkt, an dem man die Ausführung das erste Mal anhalten kann. Die Suche nach einem solchen Startpunkt lässt sich am ehesten mit „systematischem Raten“ beschreiben. Oft liegt man mit dem ersten Rateversuch daneben, siehe [KMCA06, KDV07]. Manchmal überspringt man auch wichtige Teile und wünscht sich hinterher wieder zurück gehen zu können. Bei Methoden, die oft aufgerufen werden, ist es auch nicht leicht eine passende Bedingung zu finden, wann an dieser Stelle angehalten werden soll und wann nicht. Bedingungen wie „Halte an, wenn dieser Link zwischen Objekt A und Objekt B erzeugt wird.“ lassen sich mit den vorhandenen Debuggern nur schwer stellen. Um diese Probleme zu lösen, wäre es hilfreich auf der Objektstruktur vorwärts und rückwärts in der Zeit navigieren zu können, die Änderungen über die Zeit beobachten zu können und Anfragen stellen zu können wie „Wann ist dieses Ob-

jekt erzeugt worden?“, „Wer hat diesen Link gelöscht?“ oder auch „Warum ist dieses Attribut gerade nicht gesetzt worden?“.

In vielen modellbasierten Ansätzen geschieht das Debuggen in einem Interpreter. Es wird also zusätzlich zum Codegenerator noch ein Interpreter geschrieben, der in der Lage ist die einzelnen Verhaltensmodelle auszuführen. Dieser Interpreter kann dann auch leicht schrittweise ausgeführt werden. Wenn die einzelnen Interpreterzustände gespeichert werden, erlaubt ein solcher Ansatz sogar das Rückwärtsdebuggen. Allerdings existieren bei einem solchen Ansatz mehrere Probleme.

1. Es ist nicht sichergestellt, dass sich Interpreter und generierter Code exakt gleich verhalten. Fehler die beispielsweise im Interpreter auftauchen, müssen nicht zwangsläufig auch bei der Ausführung des generierten Codes auftauchen und umgekehrt.
2. Interpreter sind meist sehr viel langsamer als kompilierter Code. Das kann dazu führen, dass manche Probleme im Interpreter nicht analysierbar sind, weil die Ausführung zu lange dauern würde, oder dass im Interpreter bestimmte Timingrobleme nicht auftreten oder zusätzlich hinzukommen.
3. Das Einbinden von externen Komponenten ist in interpretierten Ansätzen oft schwierig. So ist es meist schwer externe Bibliotheken aus dem Interpreter heraus anzusprechen. Auch ist es in manchen Fällen nicht möglich Interpreter auf der Zielplattform auszuführen, beispielsweise wenn es sich um ein eingebettetes System handelt. Dann ist es nicht möglich während des Debuggens im Interpreter externe Hardware, wie Sensoren, anzusprechen.

Aus diesen Gründen wurde in dieser Arbeit zum Debuggen kein Interpreter herangezogen. Vielmehr wird versucht, bestehende Debuggingtechniken für das Debuggen von Quellcode weiterhin einzusetzen und die Resultate zurück auf die Modellebene abzubilden.

17. Konzept

Nach den Betrachtungen aus dem vorangegangenen Kapitel soll also ein Konzept entwickelt werden, dass die komplette Fehlersuche auf Modellebene ermöglicht. Die Idee, die dem hier vorgestellten Ansatz zugrunde liegt, ist möglichst viele der existierenden quelltextbasierten Debuggingtechniken und Tools wiederzuverwenden. Durch Wiederverwendung der quelltextbasierten Tools ist sichergestellt, dass sich der modellbasierte Debugger genauso verhält, wie der Debugger auf Quelltextebene. Zusätzlich vereinfacht die Wiederverwendung von Eclipse-eigenen Werkzeugen die enge Integration in die Eclipse Entwicklungsumgebung. Zusätzlich ist der Einsatz von Werkzeugen, die auf Quelltextebene arbeiten und in dieser Arbeit nicht berücksichtigt wurden, weiterhin möglich. Dies wäre beim Einsatz eines Interpreters auf Modellebene zum Debuggen nicht mehr möglich. Das Debuggen auf Modellebene, also auf den Elementen, die üblicherweise in der Analyse- und Designphase verwendet werden, nenne ich im Folgenden *Design Level Debugging*.

Um nun quelltextbezogene Informationen, wie zum Beispiel Fehlermeldungen auf Modellebene, abbilden zu können, ist es nötig zu wissen, aus welchem Modellelement die zugehörigen Quelltextzeilen generiert wurden. Um diese Zuordnung zu erhalten, wird in diesem Ansatz während der Codegenerierung mitprotokolliert, welches Modellelement welche Zeilen erzeugt hat. Mit diesen Informationen ist es dann möglich, Fehlermeldungen, Warnungen und Stacktraces so anzupassen, dass sich diese auf Modellelemente beziehen. Zusätzlich kann man mit dieser Zuordnung einen Debugger auf Modellebene implementieren. Hierzu werden Haltepunkte im Modell einfach auf Haltepunkte im Quelltext abgebildet. Ein Schritt bei der schrittweisen Ausführung im Modell führt so viele Schritte im Quelltext aus, bis eine Quelltextzeile erreicht wird, die zu einem neuen Modellelement gehört. Für Java existiert mit dem JSR-045 [Sun09] eine Implementierung zum Debuggen von anderen Sprachen, die viele dieser Mechanismen bereits unterstützt. Diese Implementierung wird in dieser Arbeit verwendet um modellbasiertes Debuggen mit generiertem Java Quelltext zu unterstützen.

In [Gei02a, GZ02a, Gei02b] wurden von uns bereits erste Ansätze zum modellbasierten Debuggen beschrieben. Diese boten aber eher eine lose Kopplung zwischen externem Debugger und Fujaba. Als Debugger wurde noch nicht der Eclipse Debugger eingesetzt, weshalb auch keine Integration mit Eclipse und Fujaba4Eclipse erfolgte. Zur Zeit dieser alten Arbeiten setzte Fujaba noch nicht die Template-basierte Codegenerierung CodeGen2 ein. Daher musste die Abbildung zwischen Modell und Code fest in die Codegenerierung implementiert werden. Dies konnte in dieser neuen Arbeit sehr viel flexibler gegenüber Änderungen an der Codegenerierung gestaltet werden. Zusätzlich setzen die vorherigen Arbeiten noch nicht auf dem JSR-045 auf. Dadurch legt sich der alte Ansatz auf den damals verwendeten externen Debugger

fest. In dieser neuen Arbeit kann jeder JSR-045 kompatible Debugger verwendet werden. [Gei08] beschreibt den in dieser neuen Arbeit vorgestellten Stand des modellbasierten Debuggens.

Eine weitere Kernidee dieser Arbeit ist die des *objects first* Ansatzes [DGZ05c], die Objekte zu den wichtigsten Artefakten der Softwareentwicklung macht. Da im hier beschriebenen Fujaba Process in Szenarien (Storyboards) und im Verhaltensmodelll (Storydiagramme) jeweils Objektdiagramme verwendet werden, ist eine Fokussierung auf Objekte und Änderungen an der Objektstruktur während des Debuggings sinnvoll. Da in den meisten objektorientierten Applikationen der Systemzustand durch die Objektstruktur im Speicher beschrieben wird, ist also für einen modellbasierten Debugger die Möglichkeit, Objektstrukturen lesbar anzeigen zu können, unabdingbar.

Um Datenstrukturen im Speicher anzeigen zu können, wurde von mir der eDOBS entwickelt. eDOBS visualisiert Objektstrukturen als Objektdiagramme. Änderungen an der visualisierten Objektstruktur werden sofort im eDOBS sichtbar. Somit lässt sich die Entwicklung von Objektstrukturen über die Zeit verfolgen. eDOBS ist in der Lage, Strukturen beliebiger Modelle (z.B. von EMF Modellen) anzuzeigen. Hierzu wird eine spezielle Modellabstraktionsschicht eingesetzt.

eDOBS ist eine Weiterentwicklung des in Fujaba 4 enthaltenen Dynamic Object Browsing Systems DOBS [Hag99, NZ00]. DOBS wurde im Rahmen einer Projektgruppe 1998 an der Universität Paderborn gestartet. DOBS war in den Anfängen fester Bestandteil des Fujaba Kerns und diente dazu, Objektstrukturen der aktuellen Java VM als Objektdiagramm darzustellen. Im Rahmen meiner Studienarbeit [Gei02a] wurde DOBS bereits 2002 eine einfache Debuggerintegration hinzugefügt. In 2003 wurde DOBS von mir in ein eigenes Plugin ausgelagert, um seine Funktionalität auch in anderen Applikationen als Fujaba nutzbar zu machen. 2004 schließlich erhielt der Lehrstuhl Software Engineering der Universität Kassel den von IBM ausgezeichneten „eclipse Innovation Award 2004“ für die Idee, eine Portierung des DOBS als Eclipse Plugin zu starten. Dieses neue Plugin eDOBS wurde in den folgenden Jahren von mir basierend auf den Ideen von DOBS neuentwickelt und durch neue, über die Möglichkeiten von DOBS hinausgehende, Funktionalität ergänzt. Der eDOBS wird auch in [GZ06b] beschrieben.

Da, wie bereits erwähnt, bei den meisten mit Fujaba entwickelten Applikationen die Objektstruktur den Zustand des Systems beschreibt, werden Zustandsübergänge durch Änderungen an der Objektstruktur beschrieben. Ein Großteil der Funktionalität führt also Änderungen an der Objektstruktur aus. Daher stellen sich während des Debuggings oft Fragen, wann und wo bestimmte Änderungen durchgeführt wurden, beispielsweise „Wann ist dieses Objekt erzeugt worden?“. Solche Fragen kann die Flipbook Erweiterung des eDOBS beantworten. Das Flipbook Plugin wurde in der von mir betreuten Diplomarbeit von Jörn Dreyer implementiert [Dre08]. Das Flipbook Plugin protokolliert alle Änderungen an der Objektstruktur und ist in der Lage beliebig in diesen Änderungen vor und zurück zu navigieren. Die aktuelle Objektstruktur wird dabei jeweils im eDOBS angezeigt.

Da der Ausgangspunkt für eine Debuggingsession oft ein fehlgeschlagener Test ist,

wird das in Teil II vorgestellte Testkonzept eng in die Debuggingfunktionalität, die in diesem Teil vorgestellt werden, integriert. So wird beispielsweise automatisch der eDOBS geöffnet und die aktuelle Objektstruktur angezeigt, wenn ein Test fehlgeschlagen ist. Der eDOBS kann dann zur Analyse des Fehlers benutzt werden. Zusätzlich kann mithilfe des Flipbook Plugins zu der Stelle navigiert werden an der der Fehler entstanden sein könnte. Weiterhin wird auch das schrittweise Ausführen der Storyboards während der Testabarbeitung unterstützt.

In Kapitel 18 wird zunächst die Umsetzung der hier vorgestellten Ideen beschrieben. Dabei wird in Abschnitt 18.1 zunächst auf das Debuggen auf Modellebene eingegangen. Abschnitt 18.2 beschreibt den Objektbrowser eDOBS. Insbesondere wird hierbei im Abschnitt 18.2.7 auf die Integration mit dem Testkonzept dieser Arbeit eingegangen und in Abschnitt 18.2.8 die Flipbook Erweiterung des eDOBS diskutiert. In Kapitel 19 werden schließlich unsere Erfahrungen mit dem Einsatz der hier beschriebenen Debugginghilfen beschrieben, Kapitel 20 vergleicht unsere Konzepte mit verwandten Arbeiten und in Kapitel 21 wird schließlich ein Fazit gezogen.

18. Umsetzung

Für die Fehlersuche auf Quellcodeebene bieten moderne Entwicklungsumgebungen wie Eclipse eine Vielzahl an hilfreichen Werkzeugen. In diesem Kapitel wird beschrieben, wie solche Hilfen zur Fehlersuche für das Fujaba4Eclipse CASE Tool auf Modellebene realisiert wurden.

18.1. Design Level Debugging

Wie im letzten Kapitel beschrieben, basiert ein wesentlicher Teil des modellbasierten Debuggens auf einer Abbildung zwischen Modell und Quelltext. An diese Abbildung stellen sich folgende Anforderungen:

Flexibel Der gewählte Ansatz sollte möglichst einfach in den Codegenerierungsprozess integrierbar sein. So sollte es ohne viel Aufwand möglich sein, ein solches Mapping auch für neue Sprachelemente, Templates und sogar komplett neue Sprachen anzubieten.

Anpassbar Das Mapping sollte auch aus Templates heraus anpassbar sein. Falls der Benutzer ein Template so abändert, dass es zum Beispiel noch Code für ein weiteres Modellelement generiert, sollte er auch in der Lage sein, dass Mapping entsprechend anpassen zu können.

Stabil Das Mapping sollte möglichst stabil gegenüber Änderungen sein, die dem eigentlichen Codegenerierungsprozess nachgelagert sind, wie zum Beispiel ein nachträgliches Pretty-Printing.

Bidirektional Das Mapping sollte in beide Richtungen navigierbar sein: Vom Code zum Modell und vom Modell zum generierten Code.

Integriert Das Mapping sollte bestmöglich in Eclipse integriert werden um beispielsweise Compilerfehlermeldungen ins Modell zurückverfolgen zu können.

Wie diese Anforderungen erfüllt werden können, wird in den folgenden Abschnitten vorgestellt.

18.1.1. Quelltext-Modell Abbildung in CodeGen2

Das in dieser Arbeit vorgestellte Konzept hängt sich in den bestehenden Codegenerierungsprozess ein. Wird für ein Modellelement Code generiert, wird vor und hinter den generierten Codeblock ein Kommentar geschrieben, der einen Identifier enthält, der dem gekapselten Modellelement zugeordnet werden kann. Der so erzeugte Quellcode wird ausgegeben und kann nun zum Beispiel durch einen Pretty-Printer

neu eingerückt oder durch einen externen Codegenerator mit zusätzlichem Code angereichert werden[1]. Nachdem diese Änderungen erfolgt sind, werden die generierten Dateien erneut eingelesen, die Kommentare entfernt und die Quelltext-Modell Abbildung angelegt. Durch die Entscheidung, Kommentare in den Quelltext zu generieren, wurde also die Anforderung der Stabilität aus dem vorangegangenen Kapitel erfüllt.

In dieser Arbeit wird für die Quelltext-Modell Abbildung ein Baum verwendet, der die Position jedes durch einen Kommentar gekapselten Codeblocks in einer Datei mit dem zugehörigen Modellelement verknüpft. Die Modell-Quelltext Abbildung kann beispielsweise beschreiben, dass ein bestimmter Codeblock aus einer bestimmten Klasse im Modell generiert wurde. Durch die Verwendung einer Baumstruktur kann dies allerdings noch weiter verfeinert werden. So kann die Abbildung besagen, dass ein Teil des Codeblocks von einer Methode der Klasse generiert wurde, wovon wiederum ein Teil von einer bestimmten Aktivität und ein Teil hiervon von einer Linksuche generiert wurde. Die Modell-Quelltext Abbildung erhält somit eine Hierarchie.

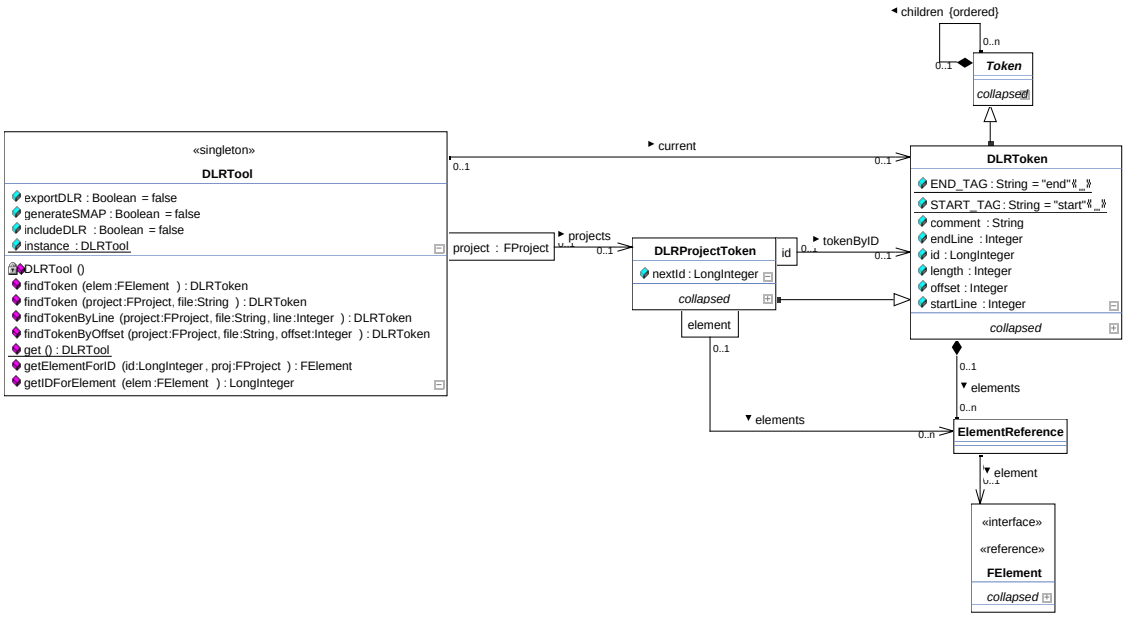


Abbildung 18.1.: Klassendiagramm des Code-Modell Mapping Baums

Die Klassen, die im Klassendiagramm in Abbildung 18.1 dargestellt sind, dienen zum Aufbau des Baums der das Code-Modell Mapping speichert. Die Elemente des Baums werden durch Objekte der Klasse `DLRToken` repräsentiert. Die Baumstruktur wird durch die `children` Assoziation der Oberklasse `Token` erzeugt. Zeilennummern und Zeichenpositionen werden in den Attributen `startLine`, `endLine`, `offset` und `length` gespeichert. Ein Kommentar, der das Mapping genauer spezifiziert wird im Attribut `comment` abgelegt. Zusätzlich verfügt jedes `DLRToken` über eine eindeutige ID. Auf die Modellelemente, die den aktuellen Codeblock generiert haben, wird

[1] Im Fujaba Tool wird beispielsweise zur EMF Codegenerierung nach der Fujaba Codegenerierung der generierte Code durch den Eclipse EMF Codegenerator um EMF kompatible Zugriffsmethoden etc. erweitert.

über Objekte der Klasse `ElementReference` verwiesen. Für jedes Projekt, für das Code generiert wird, wird ein Objekt der Klasse `DLRProjectToken` erzeugt. Dieses Objekt enthält eine Map, die die `DLRToken` über ihre ID zugreifbar macht und die `ElementReferences` über das entsprechende Modellelement. Schließlich hält der Singleton `DLRTool` eine Map aller `DLRProjectToken`, die über das Projekt zugreifbar sind. Zusätzlich enthält die `DLRTool` Klasse, eine Menge von Methoden, die zum Auflösen des Modell-Code Mappings dienen. So lässt sich beispielsweise mit der `findTokenByLine()` Methode bei Übergabe eines Projekts, eines Dateinamens und einer Zeilennummer, das `DLRToken` herausfinden, das diesen Codeblock repräsentiert.

Um die Anforderung der Flexibilität, die im vorigen Kapitel gefordert wird, zu erfüllen, wähle ich in dieser Arbeit den Ansatz um jedes `Token`, und somit für jedes atomare Element der Codegenerierung, für das Quelltext generiert wird, automatisch entsprechende Kommentare herum zu generieren. Somit muss für neue Sprachelemente lediglich angegeben werden, welche Modellelemente zum zugehörigen Token gehören. Dann werden diese neuen Sprachelemente automatisch in das Mapping einbezogen. Generiert ein Template aber mehrere Quelltextblöcke für unterschiedliche Modellelemente, kann dieses durch einen Methodenaufruf im Template signalisiert werden. Es kann also im Template auf einfache Weise ein neuer Knoten im Mappingbaum angelegt werden. Dadurch ist die Anforderung der Anpassbarkeit erfüllt.

Abbildung 18.2 zeigt die `CodeWriter` (vergleiche Abbildung 6.3), die zum Erstellen des Code-Modell Mappings in die Codegenerierung eingehängt werden. Der `ProjectDLRCodeWriter` legt ein `DLRProjectToken` an, sobald für ein Projekt Code generiert wird und meldet dieses beim `DLRTool` an. Der `DLRCodeWriter` legt für jedes `Token` ein Mappingobjekt der Klasse `DLRToken` mit einer eindeutigen ID an und generiert zu dem für das Token generierten Quelltextblock einen Startkommentar und einen Endkommentar, die auf diese eindeutige ID verweisen. Dieses `DLRToken` wird beim zugehörigen `DLRProjectToken` in der entsprechenden Map hinterlegt. Der `FileDLRCodeWriter` schließlich baut den Baum der Mappingobjekte auf. Sobald Quelltext für ein `UMLFile` Objekt generiert wurde, der dann in eine Datei geschrieben werden kann, wird dieser Quelltext nach den vorher generierten Kommentaren durchsucht. Wird ein Startkommentar gefunden, wird das zugehörige `DLRToken` in den aktuellen Baum eingehängt und die aktuelle Zeile und die aktuelle Zeichenposition im `DLRToken` gesetzt. Alle Kommentare, die vor dem zugehörigen Endkommentar stehen, werden als Kinder in den Tokenbaum eingehängt. Sobald der zugehörige Endkommentar gefunden wird, wird die Endzeile und die Zeichenlänge im `DLRToken` gespeichert. Aus der Velocityumgebung wird die Variable `$opComment` ausgelesen, die in den Codegenerierungs-Templates mit einer Beschreibung der aktuellen Aktion gefüllt werden kann (siehe Abschnitt 12.3). Ist die Variable `$opComment` gesetzt wird die Beschreibung in das `comment` Attribut des zugehörigen `DLRToken` geschrieben. Dieses Attribut wird später als Beschreibung der entsprechenden Aktion während des Debuggens verwendet. So ist es in den Templates möglich, individuelle Kommentare für das Debuggen zu hinterlegen. Weiterhin wird der Quelltext von den Kommentaren bereinigt in die Quelltextdatei geschrieben. Der `FileDLRCodeWriter` erzeugt auch die im nächsten Kapitel beschriebenen SMAP Dateien, weshalb die Klasse von `CodeToFileFromContextWriter` erbt.

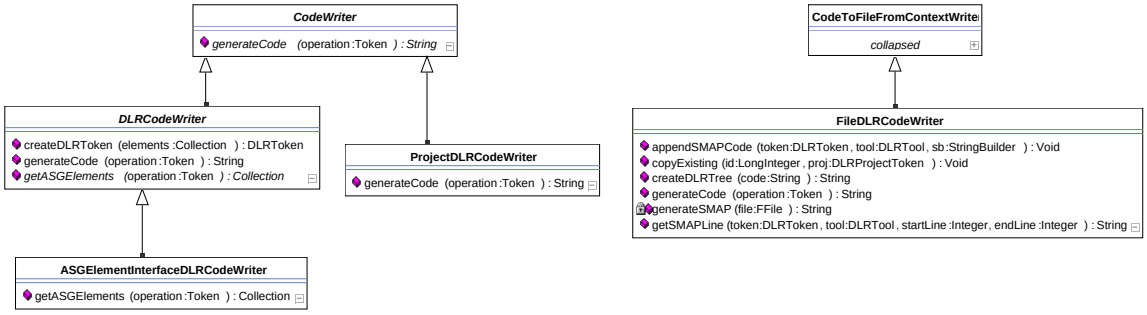


Abbildung 18.2.: Klassendiagramm der zusätzlichen `CodeWriter`

Listing 18.1 zeigt den Quelltext einer einfachen `State` Klasse. In diesem Code befinden sich noch die vom `FileDLRCodeWriter` generierten Kommentare. Man sieht beispielsweise, dass dem kompletten Quelltext ein `DLRToken` mit der ID 6 zugeordnet ist. Dieses Token repräsentiert den Quelltext, der für eine Datei, sprich ein Objekt der Klasse `UMLFile` im Fujaba Modell, generiert wurde. Der Code der Klasse selbst wird durch ein Token mit der ID 7 dargestellt. Die Methode `visit()` wurde von Token 9 generiert, usw.

```
1  /*start id=6*//*
2   * generated by Fujaba - CodeGen2
3   */
4  package de.uni_kassel.statechart.model;
5
6
7  /*start id=7*/public class State
8  {
9  /*start id=8*/
10     private String name;
11
12     public void setName (String value)
13     {
14        this.name = value;
15     }
16
17     public String getName ()
18     {
19        return this.name;
20     }
21 /*end id=8*//*start id=9*/
22     public boolean visit ()
23     {
24 /*start id=10*/
25        /*start id=12*/return true;
26     /*end id=12*//*end id=10*/}
27 /*end id=9*/
28     public void removeYou()
29     {
30     }
```

```
31 }
32 /*end id=7*/
33 /*end id=6*/
```

Listing 18.1: Beispielquelltext mit Kommentaren

Aus dem Quelltext mit den Kommentaren in Listing 18.1 erstellt der `FileDLRCodeWriter` schließlich einen Baum, wie er durch die Objektstruktur in Abbildung 18.3 beschrieben wird.

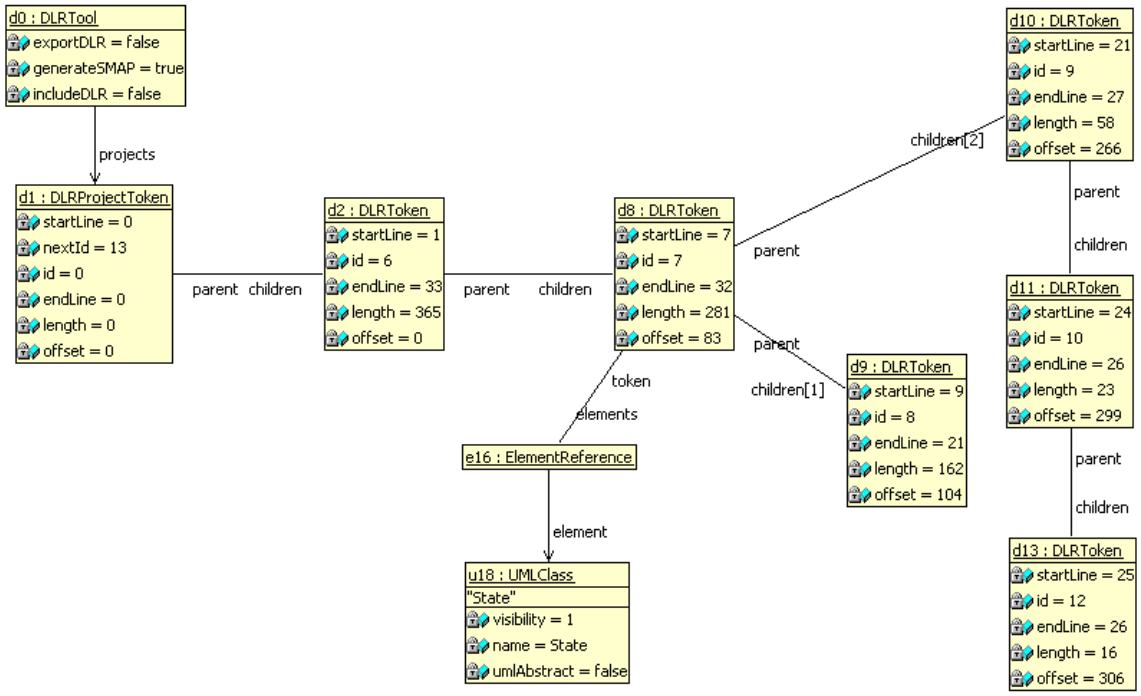


Abbildung 18.3.: Mappingbaum

Das `DLRTool` Objekt `d0` hält eine Liste mit allen Projekten, hier dargestellt durch den `projects` Link zum `DLRProjectToken` Objekt `d1`. Der Baum wird durch die `children` Links aufgespannt. Beim `DLRToken` mit der ID 7 wurde exemplarisch das `ElementReference` Objekt mit dargestellt, welches über den `elements` Link erreichbar ist. Dieses Objekt verweist auf das `UMLClass` Objekt `u18`. Durch die `startLine` und `endLine` Attribute des zugehörigen `DLRTokens`, lässt sich der Quelltext von Zeile 7 bis Zeile 32 in Listing 18.1 also eindeutig der Klasse zuordnen, die durch das Fujaba Modellelement `u18` repräsentiert wird. Die Maps des Objekts `d1`, sowie die `comments` Attribute wurden aus Gründen der Übersichtlichkeit in diesem Diagramm versteckt.

Der Baum, der die Modell-Code-Abbildung für ein Projekt speichert, wird zusätzlich in eine XML-Datei serialisiert. Hierdurch kann die Abbildung auch nach dem Schließen und erneuten Öffnen eines Projekts wieder hergestellt werden, ohne dass erneut Quelltext generiert werden muss.

18.1.2. JSR-045 und SMAP

Im vorangegangenen Abschnitt wurde ein Mapping vom Code zum Modell vorgestellt. Um nun auch auf Modellebene debuggen zu können, muss dieses Mapping in den Debugprozess integriert werden. Das heißt, es muss beispielsweise möglich sein, Haltepunkte auf Modellelemente zu setzen, und bei der schrittweisen Ausführung eines generierten Programms nicht zur nächsten Zeile sondern zum nächsten Modellelement zu springen. Für solche Zwecke bietet Sun für die Java VM eine Schnittstelle an. Diese ist im JSR-045 „Debugging Support for Other Languages“ definiert [Sun09].

Der JSR-045 definiert folgenden Prozess: Ein Entwickler spezifiziert seine Applikation in einer (üblicherweise textuellen) Quellsprache. Beispielsweise könnte das eine als Java Server Page (JSP) [Sun10] beschriebene dynamische Webseite sein. Aus dieser Quellsprache wird dann eine Zielsprache (meist Java) generiert. Zusätzlich wird noch eine sogenannte Source Map (SMAP) Datei generiert. Diese Datei beschreibt welcher Quelltextblock in der Quellsprache welchem Quelltextblock in der Zielsprache zugeordnet werden kann. Diese Zuordnung erfolgt auf Datei- und Zeilenebene.

Die Dateien der Zielsprache werden jetzt kompiliert. Die so entstandenen class Dateien werden eingelesen und durch ein sogenanntes SourceDebugExtension Attribut erweitert. Dieses Attribut enthält den Inhalt der SMAP Dateien. Der Nachbearbeitungsschritt des Anreicherns von class Dateien mit SMAP Informationen wird bislang leider weder vom Sun Java Compiler unterstützt noch bietet Sun eine Referenzimplementierung an, weshalb in dieser Arbeit der SMAP Postcompiler des Eclipse ANTLR Plugins [Jue06] verwendet wurde.

Wenn nun die so entstandenen class Dateien in einem Debugger ausgeführt werden, der das Java Debug Interface (JDI) der JPDA Spezifikation von Sun (Java Platform Debugger Architecture) implementiert, kann dieser neben den echten Zeilennummer auch die Zeilennummern und Dateinamen der Quellsprache zurückliefern sowie die Quellsprache schrittweise ausführen. Für die korrekte Darstellung des Quelldokuments muss hierbei die Entwicklungsumgebung sorgen, die den Debugger benutzt (hier Eclipse).

Um nun Fujaba Diagramme mit dem JSR-045 debuggen zu können, ist es nötig aus dem im letzten Abschnitt beschriebenen Mapping Baum eine SMAP Datei zu generieren. Hierbei ist zuerst zu beachten, dass in diesem Fall eine graphische auf eine textuelle Sprache abgebildet wird. In einer graphischen Sprache gibt es natürlich keine Zeilennummern, die das SMAP Format allerdings voraussetzt. Das stellt aber kein Problem dar, da als Zeilennummer im Modell einfach die ID des zugehörigen `DLRToken` verwendet werden kann. Liefert der SMAP Mechanismus nun während des Debuggens eine solche ID-Zeilennummer, wird über die `tokenByID` Map des `DLRProjectToken` (siehe Abbildung 18.1) das `DLRToken` mit der entsprechenden ID gesucht und dessen Modellelement zurückgeliefert. Dieses Modellelement ist dann die aktuelle Position des Debuggers. Zur Erzeugung einer SMAP Datei muss zusätzlich auch noch der Baum in eine Liste übersetzt werden. Hierzu traversiert man den Baum depth-first und ordnet jeder Zeile der Zielsprache dem Knoten zu, der diese Zeile generiert hat und der am tiefsten im Baum zu erreichen ist.

Der Baum in Abbildung 18.3 würde eine SMAP Datei generieren wie sie Listing 18.2 zeigt. Die SMAP Datei beginnt mit einem Header in dem die Zieldatei (State.java, Zeile 2) und der Name der Quellsprache (Fujaba, Zeile 3) aufgeführt sind. In Zeile 6 steht der Name der Quelldatei und ab Zeile 8 folgende die Zeilenzuordnungen. Eine Zeilenzuordnung hat folgenden Aufbau: Es beginnt mit der Startzeile in der Quelldatei, in diesem Fall handelt es sich also um die ID eines `DLRTokens`. Dann steht ein „#“ Zeichen gefolgt von der Nummer der Quelldatei. Diese ist hier immer eins, da es nur eine Quelldatei, nämlich die Fujaba Modelldatei gibt (siehe Zeile 6). Dann kann, durch ein Komma getrennt, die Anzahl an Zeilen angegeben werden, die diese Zuordnung in der Quelldatei abdeckt. Da die Quellzeilennummern hier aber IDs von Modellelementen repräsentieren, ist es nicht sinnvoll anzugeben, das eine Zuordnung beispielsweise für die Modellelemente mit den IDs 104-107 gilt. Diese Angabe macht also nur für den Fall einer Text-Text Abbildung Sinn, ist hier immer eins und kann daher weggelassen werden. Nach einem Doppelpunkt folgt dann die Zeilenangabe in der Zieldatei und durch ein Komma getrennt die Anzahl der Zeilen über die sich der generierte Block erstreckt.

```
1  SMAP
2  State.java
3  Fujaba
4  *S Fujaba
5  *F
6  1 ..\..\..\..\StatechartDLR.ctr
7  *L
8  6#1:1,6
9  7#1:7,2
10 8#1:9,12
11 9#1:21,3
12 10#1:24
13 12#1:25
14 9#1:26
15 7#1:27,5
16 6#1:32
17 *E
```

Listing 18.2: SMAP Datei des Quelltexts aus Listing 18.1

Die Zeilen 9 und 15 in Listing 18.2 geben beispielsweise an, dass die Zeilen 7-8 sowie 27-32 in der Java Datei aus Listing 18.1 vom `DLRToken` mit der ID 7 also vom Objekt, dass in Fujaba die Klasse repräsentiert, generiert worden sind.

18.1.3. Eclipse Integration

Um nun einen praktischen Nutzen aus dem in den letzten zwei Abschnitten beschriebenen Modell-Code Mapping zu ziehen, muss dieses Mapping in die eingesetzten Werkzeuge und Prozesse integriert werden. Um modellbasiertes Debuggen zu ermöglichen, müssen zunächst die SMAP Dateien nach dem Kompiliervorgang in die class Dateien integriert werden. Dafür definiert das CodeGen2 Eclipse Plugin eine sogenannte Fujaba Nature. Eine Nature enthält sogenannte Builder durch

welche sich der Eclipse Buildprozess erweitern lässt. Die Fujaba Nature enthält den `SMAPInstallerBuilder` des ANTLR Plugins. Dieser Builder schreibt die SMAP Informationen nach jedem Kompiliervorgang in die entsprechenden class Dateien. Zusätzlich enthält die Fujaba Nature noch einen Builder, der die Fehlermarkierungen, die beim Kompiliervorgang entstanden sind analysiert und mit Hilfe der Methoden der Klasse `DLRTool` auf Modellebene übersetzen.

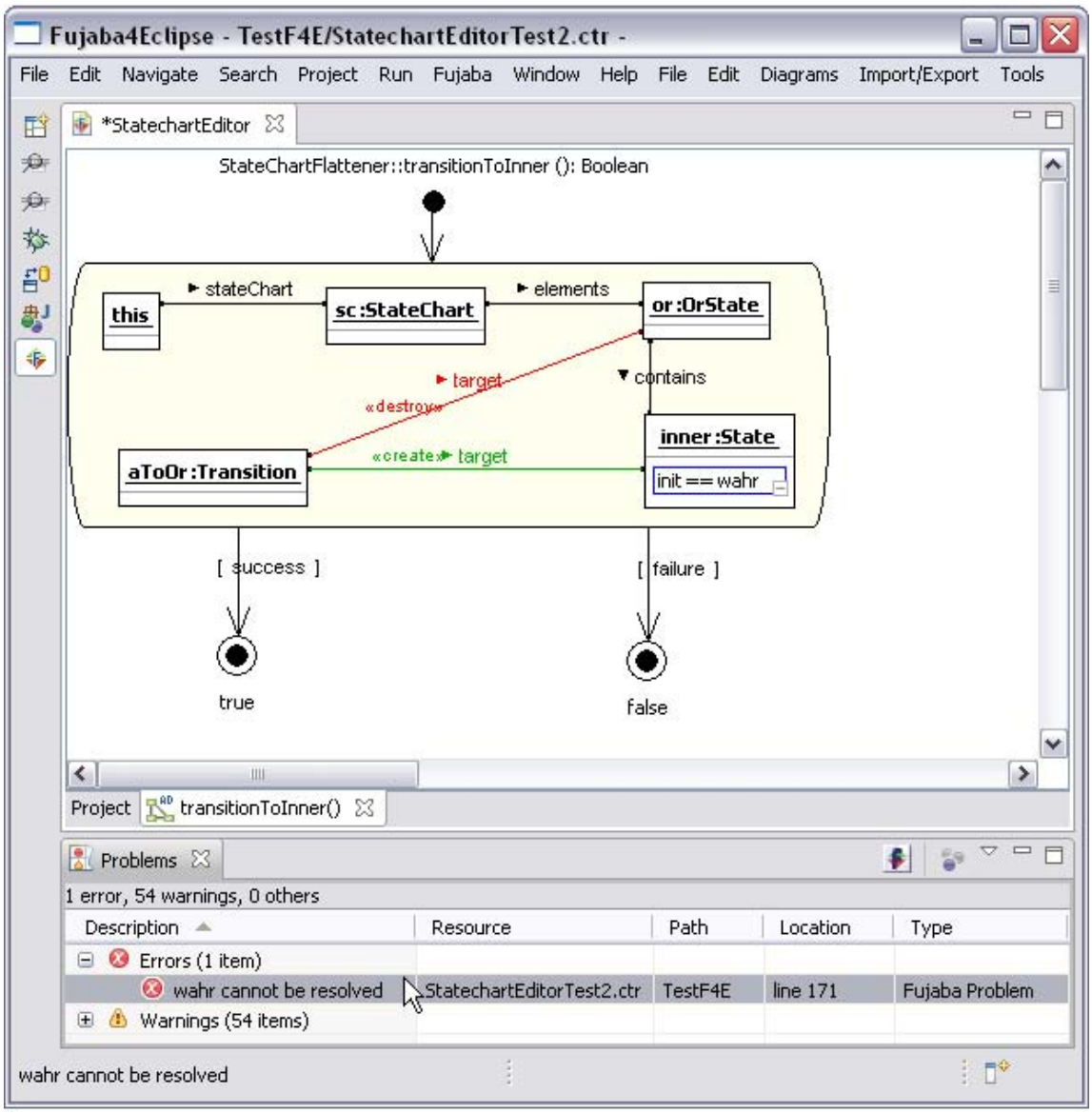


Abbildung 18.4.: Fehlermeldung in Fujaba4Eclipse

Die Modell-Code Navigation erfolgt in den nachfolgenden Beispielen immer über die `DLRToken`. Zuerst wird das passende `DLRToken` über seine ID nachgeschlagen. Dann wird das `DLRToken` nach den zugehörigen Modellelementen gefragt. Um nun ein Modellelement in Fujaba4Eclipse darzustellen, wird ein Fujaba4Eclipse API-Aufruf angestoßen, der das Diagramm öffnet, in dem das Modellelement zu sehen ist. Durch einen Eclipse API-Aufruf wird an die entsprechende Stelle gescrollt, an der sich das

Modellelement im Diagramm befinden und dieses selektiert.

Abbildung 18.4 zeigt übersetzte Fehlermeldungen in Fujaba4Eclipse. In der Ansicht *Problems* wird der Compilerfehler „wahr cannot be resolved“ angezeigt. Führt man auf dieser Markierung einen Doppelklick aus, wird das im oberen View sichtbare Diagramm angezeigt und das Element, das den fehlerhaften Code erzeugt hat, hervorgehoben. In Abbildung 18.4 ist beispielsweise leicht zu sehen, dass bei der Attributüberprüfung des Objekts `inner` ein Fehler vorliegt. Hier muss als Vergleichswert natürlich `true` anstelle von `wahr` stehen, damit der Java-Compiler den entstehenden Code übersetzen kann.

Abbildung 18.5 zeigt nun den Debugger auf Modellebene im Einsatz. Durch die SMAP Integration ist es jetzt möglich, Breakpoints auf Modellelemente zu setzen, wie im View in Abbildung 18.5 oben rechts (2) zu sehen ist. Einen Breakpoint im Modell setzt man, indem man das zugehörige Modellelement selektiert und in der Toolbar oder im Kontextmenü das *Add breakpoint* Kommando ausführt.

In Abbildung 18.5 ist der Debugger gerade auf einen solchen Breakpoint gelaufen, weshalb das zugehörige Modellelement im *Fujaba View* (3) hervorgehoben wird. In der Statusleiste wird außerdem ein Kommentar angezeigt, der angibt, was als nächstes passieren soll. In diesem Fall soll im nächsten Schritt der `target` Link erzeugt werden. Im *Debug View* (1) oben links wird der aktuelle Stack angezeigt. Selektiert man hier einen Stackframe, wird die entsprechende Stelle im Modell angezeigt, an dem sich der Programmzähler dieses Frames gerade befindet. Möchte man an dieser Stelle doch den zugehörigen Java Quelltext betrachten, um beispielsweise die Korrektheit des Codegenerators zu überprüfen, lässt sich über ein Kontextmenu auf den Stackframes auch der Debugger zwischen Debuggen auf Modellebene und Debuggen auf Quellcodeebene umschalten. Nach einem Umschalten auf Quellcodeebene wird im *Debug View* (1) der aktuelle Stack mit Quelltextpositionen angezeigt.

In der Ansicht unten rechts (4) schließlich lässt sich die aktuelle Variablenbelegung analysieren. Dies ist in der von Eclipse angebotenen Baumansicht allerdings noch ziemlich mühsam und Objektstrukturen in dieser Ansicht lassen sich nur schwer mit den zugehörigen Pattern in Fujaba vergleichen. Diese Lücke schließt der im nächsten Kapitel vorgestellte eDOBS.

Weiterhin wurde das CodeGen2 Eclipse Plugin so erweitert, dass Modell-Code Navigation an allen Stellen angeboten wird, wo es sinnvoll erscheint. So kann man natürlich von einem Breakpoint direkt zum Modellelement springen, dem dieser zugeordnet ist. Zusätzlich ist es möglich aus einer Zeile im Quelltext direkt zu dem Modellelement springen, das diese Zeile generiert hat und umgekehrt von einem beliebigen Modellelement zu den daraus generierten Zeilen springen.

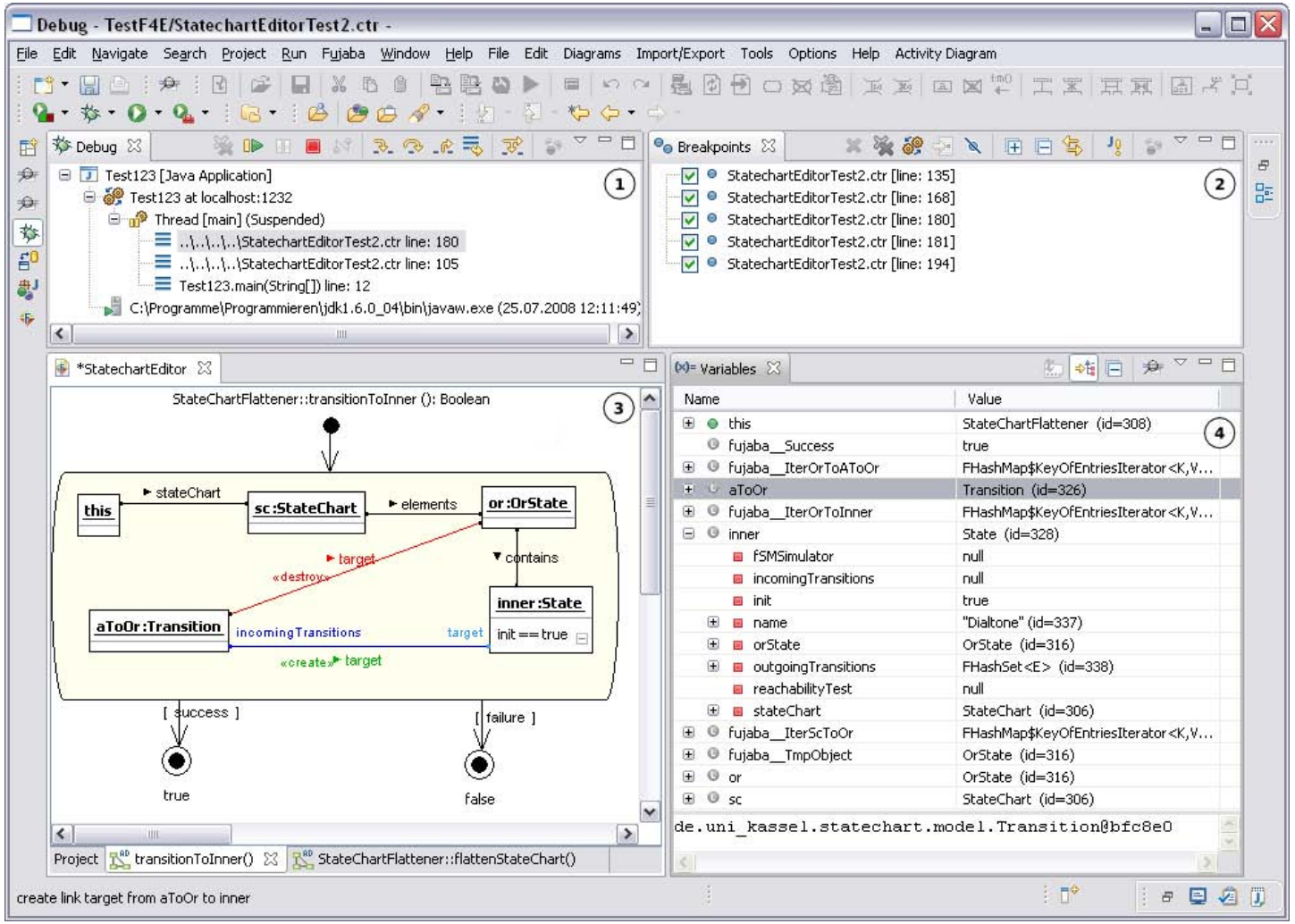


Abbildung 18.5.: Modellbasiertes Debuggen

18.2. eDOBS

Im vorangegangenen Kapitel wurde ein Vorgehen vorgestellt, das Debuggen auf Modellebene in Fujaba4Eclipse erlaubt. Es wurde beschrieben, wie schrittweises Ausführen eines Modells ermöglicht werden kann. Die Auswertung der aktuellen Variablenbelegung und der aktuellen Objektstruktur stellte sich dabei aber als mühsam heraus. Abbildung 18.5 zeigt den Debugger im Einsatz. Im View rechts unten (4) lässt sich die aktuelle Variablenbelegung ablesen und, durch Ausklappen des Baums, durch die Objektstruktur navigieren. In Abbildung 18.5 ist beispielsweise die lokale Variable `inner` ausgeklappt, so dass unter anderem ersichtlich wird, dass es sich hier um ein Objekt vom Typ `State` handelt, dessen `name` Attribut den Wert `"Dialtone"` hat und die über einen `orState` Link auf ein Objekt der Klasse `OrState` verweist. Möchte man nun herausfinden, um welches `OrState` Objekt es sich handelt, muss man dieses wiederum in der Baumansicht ausklappen. Um nun beispielsweise herauszufinden, warum ein bestimmtes Pattern fehlgeschlagen ist, muss man das Pattern mit eben dieser Baumansicht vergleichen. Im Beispiel in Abbildung 18.5 muss man also das Pattern im *Fujaba View* unten links (3) mit der Variablenbelegung unten rechts (4) vergleichen. Dies kann sich gerade bei größeren Objektstrukturen als sehr schwierig und zeitaufwendig erweisen. Besser wäre es, wenn die Objektstruktur in einer dem Pattern ähnlicheren Visualisierung vorliegen würde.

Zur Lösung dieses Problems wurde in dieser Arbeit eDOBS entwickelt. eDOBS ist eine Weiterentwicklung des Dynamic Object Browsing Systems (Dobs) [Hag99,NZ00]. eDOBS ist ein Eclipse Plugin, das den aktuellen Java Heap als UML Objektdiagramm visualisiert. Damit wird zur Visualisierung der aktuell im Speicher gehaltenen Objektstruktur die selbe Diagrammart verwendet wie zum Spezifizieren der Storypattern. So lassen sich gerade größere Objektgraphen besser analysieren und mit einzelnen Pattern vergleichen. Während einer Debuggingsession, wenn der Debugger auf einen Haltepunkt gelaufen ist, ist es nun möglich den Inhalt ausgewählter Variablen im eDOBS als Objekte anzeigen zu lassen. eDOBS zeigt dann die Attributbelegung der angezeigten Objekte und deren Links untereinander.

Abbildung 18.6 zeigt den eDOBS beim Debuggen des Beispiels aus Abbildung 18.5. Der Debugger ist vor der Erzeugung des `target` Links auf einen Breakpoint gelaufen. Jetzt kann der Entwickler im Fujaba Diagram (2), im Quelltext, im *Variables View* (3) oder in einem der anderen von Eclipse angebotenen *Debug Views* eine lokale Variable auswählen und diese mit dem *Browse in eDOBS* Kommando im eDOBS anzeigen lassen. Im Beispiel hier wurde die lokale Variable `inner` ausgewählt, welche als Objekt `s0` im eDOBS Diagramm (6) zu sehen ist. Sobald ein Objekt selektiert ist, werden alle seine Attribute im *eDOBS Attribute View* (4) und alle Methoden im *eDOBS Method View* (5) angezeigt. In Abbildung 18.6 wurde das Objekt `s0` angewählt und seine Attribute und Methoden dargestellt.

Zu jeder Zeit kann das eDOBS Diagramm um Nachbarobjekte der bereits angezeigten Objekte erweitert werden. Dies erlaubt es komplexe Objektstrukturen schrittweise zu analysieren. In Abbildung 18.6 wurde das *Expand Object* Kommando auf dem Objekt `s0` ausgeführt. Dadurch wurden die Objekte `t2`, `t3` und `o4` dem Diagramm hinzugefügt. Die ebenfalls inkludierten Objekte `s1` der Klasse `StateChart` und `s7` der Klasse

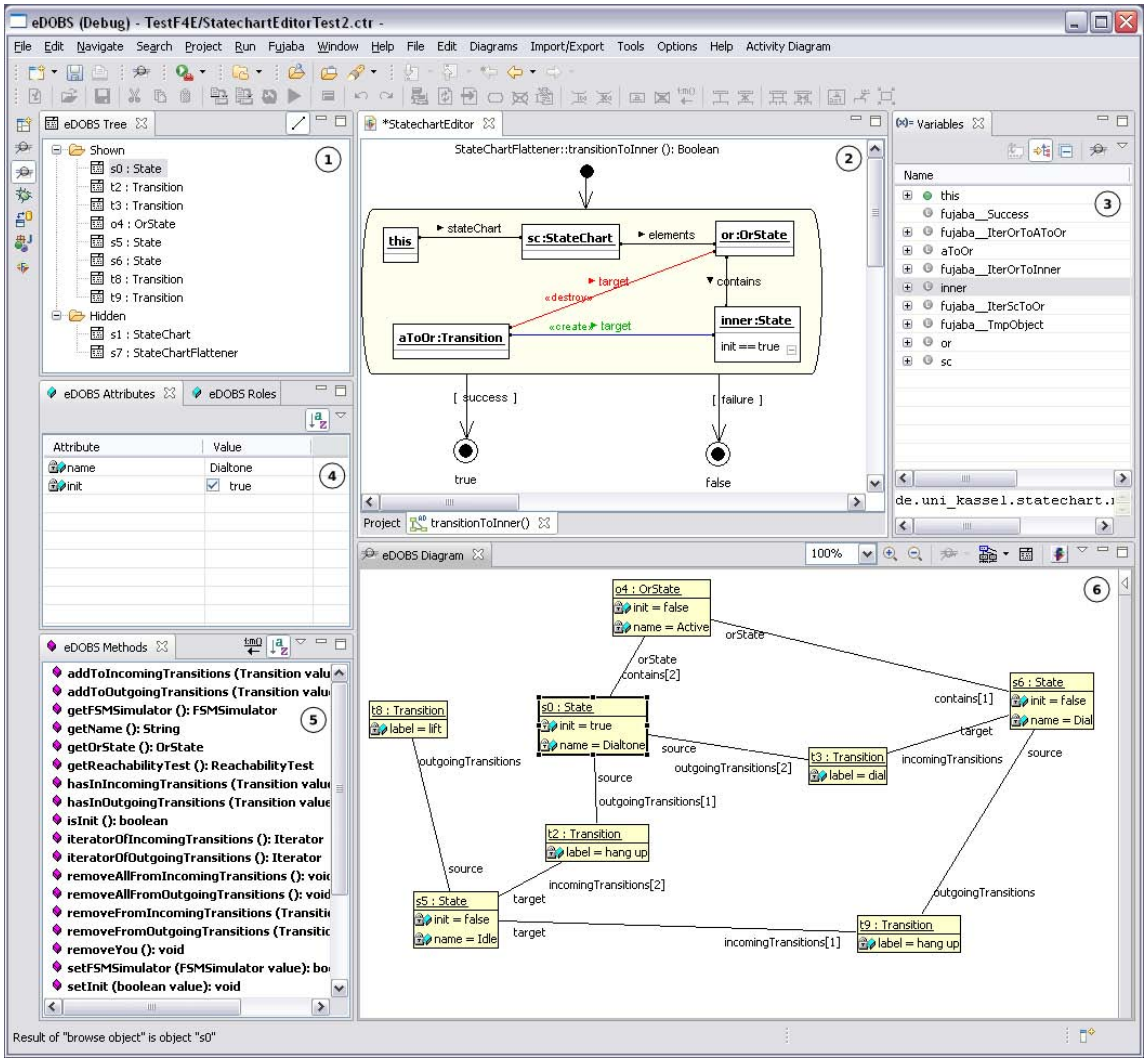


Abbildung 18.6.: eDOBS

`StateChartFlatterner` wurden der Übersichtlichkeit halber versteckt, wie im *Tree View* (1) zu erkennen ist. Weiteres Expandieren der neuen Objekte zeigt schließlich die Objektstruktur des kompletten Statecharts im eDOBS Diagramm. Der eDOBS visualisiert Referenzen, die einzelne Objekte auf andere haben, als Links. Hierbei werden Links von bidirektionalen Assoziationen nur als ein Link dargestellt. Bei zu-n Assoziationen, welche in Java meist durch Collection Objekte oder Arrays implementiert werden, versteckt eDOBS die interne Containerstruktur. Eine zu-n Relation wird also einfach als viele Links angezeigt und nicht etwa als komplexe Binärbaumstruktur. Das hebt das Abstraktionslevel von einfachen Programmierstrukturen auf die Ebene von UML Objektdiagrammen. Dieses höhere Abstraktionslevel vereinfacht zusätzlich die Analyse von komplexen Objektstrukturen.

Nachdem der Entwickler die Objektstruktur im eDOBS analysiert hat, kann er beispielsweise entscheiden, den Debugger weiterlaufen zu lassen und am nächsten Haltepunkt zu stoppen oder den Debugger schrittweise ausführen. Während einer solchen schrittweisen Ausführung oder beim Stoppen am Breakpoint zeigt der eDOBS sofort

die Änderungen an der momentan visualisierten Objektstruktur an.

Während der Analyse einer Objektstruktur in eDOBS wird für alle Elemente die Navigation zum zugehörigen Java beziehungsweise Fujaba Artefakt angeboten. Das bedeutet, dass man von einem Objekt, einem Attribut, einer Methode direkt in das zugehörige Fujaba Diagramm oder auch in den Quelltext springen kann.

18.2.1. eDOBS - Der Objektstruktureditor

Zusätzlich ermöglicht eDOBS dem Entwickler Änderungen an der Objektstruktur vorzunehmen. Wenn der Entwickler während einer Debuggingsession eine Fehlkonfiguration der Objektstruktur feststellt, kann er diese sofort korrigieren und dann gegebenenfalls weiter debuggen. eDOBS ermöglicht es Objekte und Links zu erzeugen oder zu entfernen sowie Attributwerte zu ändern. Außerdem können Methoden auf den angezeigten Objekten aufgerufen werden. Änderungen, die während eines solchen Methodenaufrufs an der Objektstruktur erfolgen, werden hier natürlich wieder sofort im eDOBS visualisiert.

Abbildung 18.7 zeigt eine Zusammenfassung der Editieroperationen im eDOBS. Im *Attribute View* (1) lassen sich primitive Attributwerte direkt ändern. Über den *New Object* Dialog (2) lassen sich neue Objekte anlegen. Sollten häufig Objekte einer Klasse angelegt werden, lässt sich diese Klasse in die Palette (3) aufnehmen. Ist dort eine Klasse selektiert, kann man durch einfachen Klick in das eDOBS Diagramm Objekte dieser Klasse anlegen. Zusätzlich bietet die Palette ein Werkzeug zum Erzeugen von Links. Im *Method View* (4) schließlich lassen sich Methoden auf dem selektierten Objekt aufrufen.

Man muss die Arbeit mit eDOBS nicht zwangsläufig während einer Debuggingsession beginnen. Der Entwickler kann ebenso ein leeres eDOBS Diagramm anlegen und dann durch die Editierkommandos des eDOBS eine eigene Objektstruktur erzeugen. Auf dieser Objektstruktur kann er nun beispielsweise wieder Methoden aufrufen und deren Änderungen im eDOBS verfolgen. Diese Funktionalität eignet sich besonders gut für Java Anfängerkurse, da ein so kompliziertes Konstrukt wie `public static void main (String[] args)` zum Erstellen einer initialen Objektstruktur nicht notwendig ist. Man kann sofort damit beginnen in einem eDOBS Diagramm Objekte anzulegen. Wir nutzen diese Möglichkeit mit Erfolg in unseren Anfängervorlesungen sowie in Kursen zur Einführung in die objektorientierte Programmierung an der Schule.

Ist die im eDOBS erzeugte oder dargestellte Objektstruktur mit Hilfe des CoObRA Frameworks (siehe Abschnitt 18.2.8) persistierbar, kann man diese Objektstruktur speichern und zu einem späterem Zeitpunkt im eDOBS wieder laden. Die zentrale Klasse des CoObRA Framework ist das `Repository`, siehe Abbildung 18.18. Bei diesem werden alle Änderungen der Objektstruktur protokolliert. Zusätzlich muss das `Repository` in der Regel konfiguriert werden. Beispielsweise muss angegeben werden, wie auf das Modell zugegriffen werden kann, da dieses sich beispielsweise für Fujaba-Modelle und EMF-Modelle erheblich unterscheidet. Ist ein so konfiguriertes `Repository` im eDOBS erreichbar, werden Kommandos zum Laden und Speichern

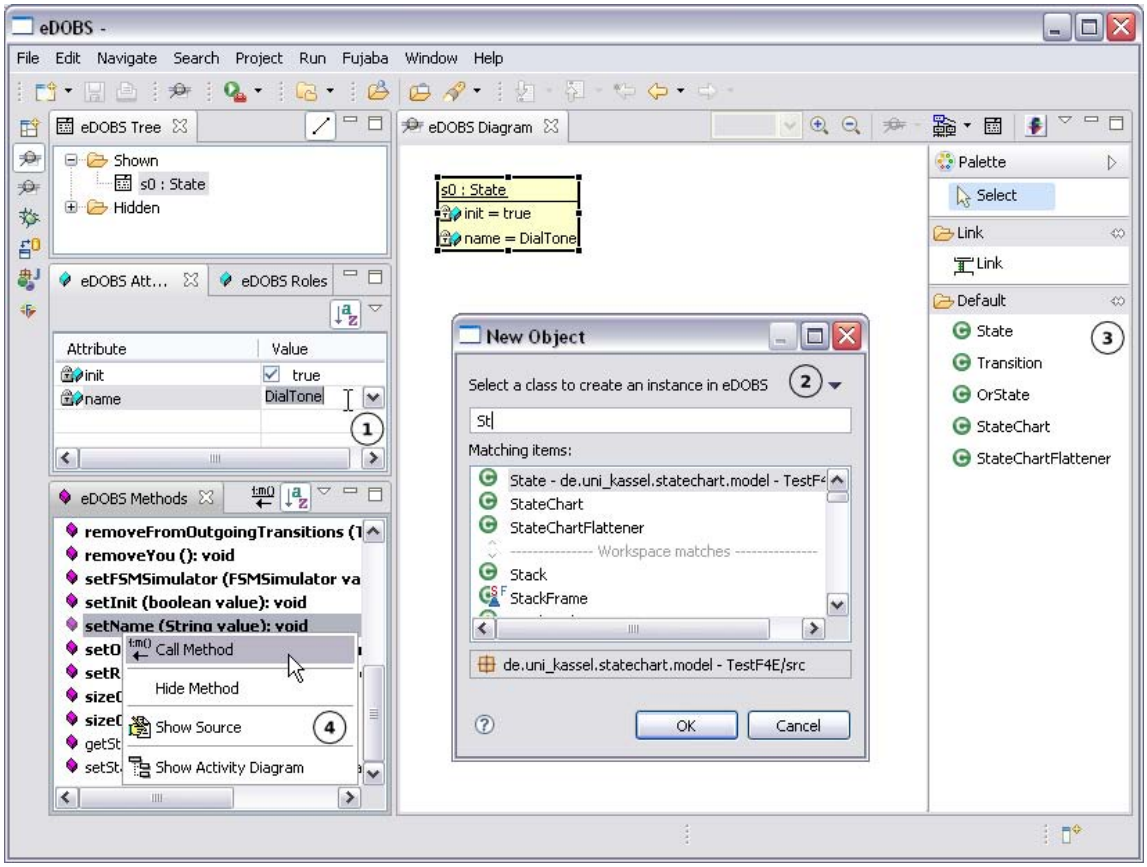


Abbildung 18.7.: Editieroperationen in eDOBS

angeboten, die dieses `Repository` benutzen. Durch diesen einfachen Lade/Speicher-Mechanismus ist der eDOBS ein vollwertiger Objektstruktureditor. Der verwendete Persistenzmechanismus wird in Abschnitt 18.2.8 genauer erläutert.

18.2.2. Modell-Abstraktion

Die Klassen der Objektstrukturen, die der eDOBS visualisieren soll, können auf unterschiedlichste Art und Weise implementiert sein. Es könnte sich beispielsweise um eine EMF konforme Implementierung handeln, um Klassen, die den JMI Standard der MOF Spezifikation umsetzen oder um normale Java Beans. Zugriffe auf diese unterschiedlichen Implementierungen, zum Beispiel das Auslesen von Attributwerten oder aber auch das Abfragen von Superklassen und -interfaces, können sich je nach Implementierung stark unterscheiden. Um trotzdem mit unterschiedlich Modellimplementierungen umgehen zu können, verwendet eDOBS eine Modellabstraktionsschicht, die in [Sch07] beschriebene *Java Feature Abstraction* Schicht. Diese bietet eine Schnittstelle zum reflektiven Zugriff auf ein beliebiges Modell. Die verwendete Abstraktionsschicht ermöglicht folgende Zugriffe auf ein Modell:

- Die Abstraktionsschicht bietet eine Schnittstelle zur **Analyse der Metamodell-Informationen** für ein gegebenes Objekt:

  - Es kann die Klasse eines Objekts zurückgegeben werden.
  - Zu einer Klasse können Oberklassen und Interfaces herausgefunden werden.
  - Zu jeder Klasse kann die Liste aller Methoden, Attribute und Rollen[2] abgefragt werden. Rollenobjekte stellen hierbei zusätzlich ihre Kardinalität zur Verfügung
  - Zu Methoden kann die Liste der Parameter sowie der Rückgabetyp, und für Attribute der Typ erfragt werden.
  - Für alle Elemente können Sichtbarkeiten und Eigenschaften (zum Beispiel `static` oder `abstract`) analysiert werden.
- Die Abstraktionsschicht erlaubt den **lesenden Zugriff** auf Modellelemente:
  - Attributwerte können ausgelesen werden.
  - Referenzen, die durch die Rollenobjekte dargestellte werden, können aufgelöst werden. Hierbei werden auch zu-n Rollen unterstützt.
- Die Abstraktionsschicht erlaubt den **schreibenden Zugriff** auf Modellelemente:
  - Objekte einer bestimmten Klasse können angelegt oder gelöscht werden.
  - Attributwerte können geändert werden.
  - Referenzen zu anderen Objekten können aufgebaut oder gelöscht werden.
  - Methoden können aufgerufen und der Rückgabewert zurückgeliefert werden.

Die in [Sch07] beschriebene Schnittstelle enthält im Wesentlichen die Funktionalität, die zur Modellserialisierung notwendig ist. Sie wurde im Rahmen dieser Arbeit komplettiert um den Einsatz mit eDOBS zu ermöglichen. So kam beispielsweise die Möglichkeit der Abfrage von Sichtbarkeiten hinzu.

Im Moment existieren vier Implementierungen der Abstraktionsschicht:

- [Sch07] liefert eine **reflektive Zugriffsschicht**, die zum Zugriff auf das Metamodell die Funktionalität des `java.lang.reflect` Paket des JDKs nutzt. Hier wird unter anderem die Assoziationserkennung (Zugriffsmethoden, Rückrichtung) von Fujaba generierten Klassen sowie von Swing UI Klassen unterstützt. Diese Zugriffsschicht ermöglicht dem eDOBS auch interne Klassen von Eclipse zu analysieren. Dies ist besonders hilfreich, wenn man selbst Eclipse Plugins entwickelt und hierzu Informationen über die interne Struktur mancher Eclipse Komponenten braucht.
- Diese Arbeit fügt eine **Debugger Zugriffsschicht** hinzu, die es ermöglicht, Objektstrukturen im Debugger zu erkunden. Diese kam in den bisherigen eDOBS Beispielen zum Einsatz und wird in Abschnitt 18.2.3 genauer diskutiert.

[2] Assoziationen werden in der Feature Abstraction durch ihre Rollen abgebildet.

- In Zusammenarbeit mit der Universität Bayreuth entstand eine **EMF Zugriffsschicht**, die es erlaubt, Modelle, die mit dem Eclipse Modeling Framework generiert wurden, zu analysieren.
- In Zusammenarbeit mit der TU Darmstadt schließlich entstand die **JMI Zugriffsschicht**, die den Zugriff auf MOF Repositories, wie etwa NetBeans MDR [Net10] erlaubt. Hiermit lassen sich auch die Klassen untersuchen, die mit dem Fujaba MOF Plugin MOFLON [AKRS06] generiert wurden.

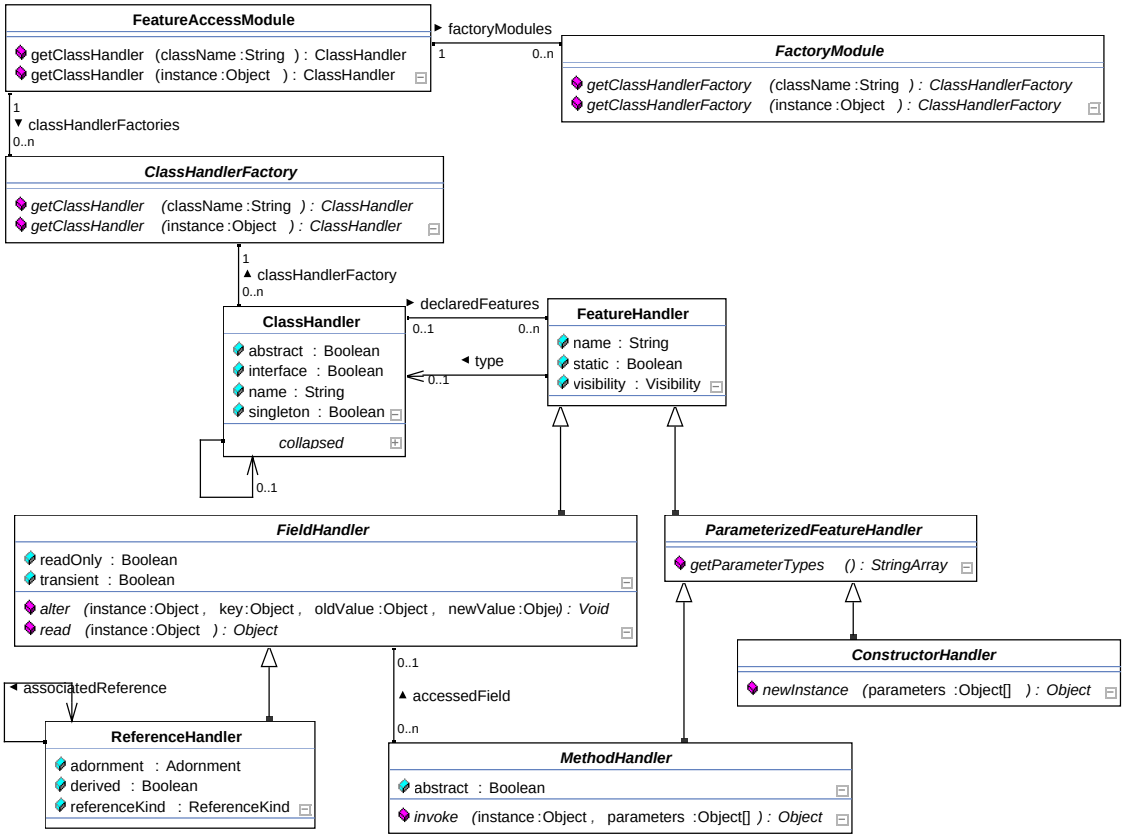


Abbildung 18.8.: Schnittstelle der Java Feature Abstraction (aus [Sch07])

Abbildung 18.8 zeigt das Klassendiagramm, das die Schnittstelle der Abstraktionsschicht definiert. Die Klasse `ClassHandler` bildet hier das Konzept der Klasse ab. Man kann beispielsweise den Namen oder die `abstract` Eigenschaft abfragen. Ein `ClassHandler` hat eine Menge an `FeatureHandlern`. Ein `FeatureHandler` ist dabei entweder ein Konstruktor (`ConstructorHandler`), eine Methode (`MethodHandler`), oder ein Feld (`FieldHandler`). Attribute und Rollen werden in der Feature Abstraction ähnlich behandelt und sind jeweils Subklassen von `FieldHandler`. Neben der Möglichkeit, Informationen über das Metamodell (wie Klassennamen, Attributnamen etc.) abzufragen, bietet die Schnittstelle auch die Möglichkeit, reflektiv auf das Modell zuzugreifen. Die Methode `newInstance(...)` der Klasse `ConstructorHandler` legt beispielsweise ein neues Objekt an, die Methode `invoke(...)` der Klasse `MethodHandler` ruft eine Methode auf dem übergebenen Objekt auf, usw. Zum einfachen Zugriff bietet die Feature Abstraction die Klasse

`FeatureAccessModule` an. Bei ihr kann man zu einem beliebigen Objekt bzw. einem beliebigen Klassennamen den zugehörigen `ClassHandler` erfragen. Intern wird diese Anfrage von einer Chain of Responsibility von `FactoryModules` beantwortet, die ihrerseits wieder eine Liste von `ClassHandlerFactories` verwenden.

18.2.3. Debugger Integration

In Java läuft das zu debuggende Programm immer in einer separaten Virtual Maschine (VM), dem sogenannten Debuggee. Dies ist notwendig, damit der Debugger, hält er die VM des Debuggee an, sich nicht selbst anhält. Um auf Objekte in dieser VM aus der Debugger VM zugreifen zu können, bietet Sun seit der Version 1.3 des JDKs das sogenannte Java Debug Interface (JDI) [Sun04] an. In Eclipse wird diese Schnittstelle durch die Java Development Tools (JDT) erneut gekapselt. eDOBS verwendet die Schnittstelle, die das Paket `org.eclipse.jdt.debug.core` des JDTs definiert um auf Objekte im Debuggee zugreifen zu können. Es wird also eine Implementierung der im letzten Kapitel beschriebenen Feature Abstraction mithilfe des JDTs verwendet.

18.2.4. Abstraktionen

eDOBS bietet mehrere Abstraktionsmechanismen an. Diese dienen in erster Linie dazu, interne Objektstrukturen, also die abstrakte Syntax, der eigentlichen Darstellung, der konkreten Syntax, anzunähern. Ziel ist es, die eDOBS Diagramme somit auch für Domänenexperten oder Kunden zugänglich zu machen. eDOBS wird dadurch zu einer Prototyping-Umgebung.

Die einfachste Abstraktion ist das Zuweisen von Icons. Mit einem Konfigurationsdialog kann man jeder Klasse ein Icon zuordnen. Wird dann ein Objekt dieser Klasse angezeigt, wird je nach Konfiguration nur das Icon, oder das Objekt mit dem Icon links neben dem Objektnamen angezeigt. Abbildung 18.9 zeigt das eDOBS Diagramm für das Statechart-Beispiel mit Icons.

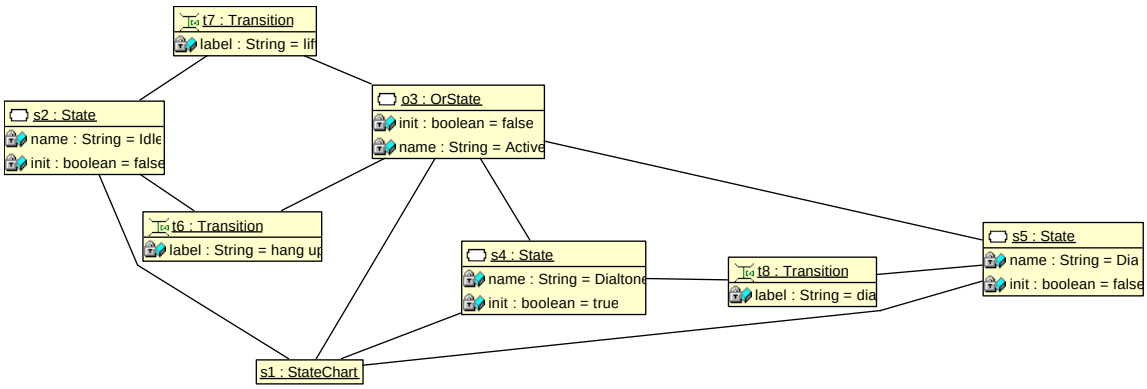


Abbildung 18.9.: eDOBS mit Icons

eDOBS bietet auch rudimentären Support für dynamische Icons. Besitzt ein Objekt eine `getIcon()` Methode, die eine Zeichenkette zurückliefert, wird diese Methode bei jeder Änderung im eDOBS aufgerufen und der Rückgabewert als Dateiname für ein Icon interpretiert. Kann dieses Icon geladen werden, wird es für das entsprechende Objekt übernommen. Dieses Vorgehen widerspricht zwar der in der Softwaretechnik durch das Model-View-Controller Pattern vorgeschlagenen Trennung von Modell und Darstellung, da hier Quelltext zur Darstellung direkt in den Modellklassen implementiert ist. Es ermöglicht es aber, schnell dem Modell zustandsbasierte Icons hinzuzufügen.

Auf ähnliche Weise lässt sich auch ein automatischer Layoutmachanismus realisieren. eDOBS sucht zusätzlich zur `getIcon()` Methode noch nach einer `getPos()` Methode, die ein Objekt vom Type `java.awt.Point` zurückliefert. Wird eine solche Methode gefunden, wird der Rückgabewert als aktuelle Position des zugehörigen Objekts verwendet. Durch diese beiden Mechanismen lassen sich simple Visualisierungen auf einfachste Art realisieren. Wir haben hiermit insbesondere in Anfängerkursen gute Erfahrungen gemacht, wo auf diese Art beispielsweise eine einfache Mensch-ärgere-dich-nicht Darstellung erstellt wurde.

Abbildung 18.10 zeigt den eDOBS bei der Darstellung der Objektstruktur einer einfachen Fabriksimulation bestehend aus Robotern, Transportshuttles, Weichen und einfachen Wegabschnitten[3]. Das Layout in Abbildung 18.10 entsteht durch geschickte Implementierung der `getPos()` Methode. Zusätzlich ist es hier möglich, durch eine zustandsabhängige Implementierung der `getIcon()` und der `getPos()` Methode eine Animation mit sich bewegenden Shuttles und arbeitenden Robotern zu erstellen. eDOBS kann also auch als Umgebung zur Erstellung von einfachen Prototypen benutzt werden.

Zusätzlich unterstützt eDOBS die in Abschnitt 5.2 beschriebene Kante-Knoten-Kante Abstraktion sowie die Schachtelung von Objekten. Es ist also möglich mit einem Konfigurationsdialog zu definieren, dass Objekte einer bestimmten Klasse als Kante darzustellen sind, wobei ein bestimmtes Attribut den Kantenbezeichner enthält, eine Rolle das Startobjekt der Kante und eine weitere Rolle das Zielobjekt. Ähnlich lässt sich definieren, dass eine Rolle einer bestimmten Klasse nicht als Link dargestellt wird sondern zur Einschachtelung aller dadurch erreichbaren Nachbarn führt. Abbildung 18.11 zeigt diese Abstraktionen für das Statechart-Beispiel. Vergleicht man Abbildung 18.11 mit Abbildung 18.9 ist ersichtlich, dass Objekte der Klasse `Transition` als Kante dargestellt werden und der `contains` Link der Klasse `OrState` zur Schachtelung verwendet wird. Damit ähnelt Abbildung 18.11 sehr dem Ergebnis des Objektspiels aus Abbildung 5.2.

Liegt das Klassendiagramm eines Modells schon vor, eignet sich der eDOBS also auch dazu ein Objektspiel am Rechner durchzuführen. Hierbei können die Editieroperationen des eDOBS genutzt werden, um eine Objektstruktur zu erstellen und diese zu verändern. Methodenaufrufe können hierbei als „Abkürzungen“ benutzt werden. Wir haben mit diesem Vorgehen insbesondere in der Lehre gute Erfahrungen gemacht. Zum Kennenlernen eines unbekannten Modells hat sich hier das Objektspiel

[3]Das Beispiel entstammt dem ISILEIT Project der Universität Paderborn [NSZ03].

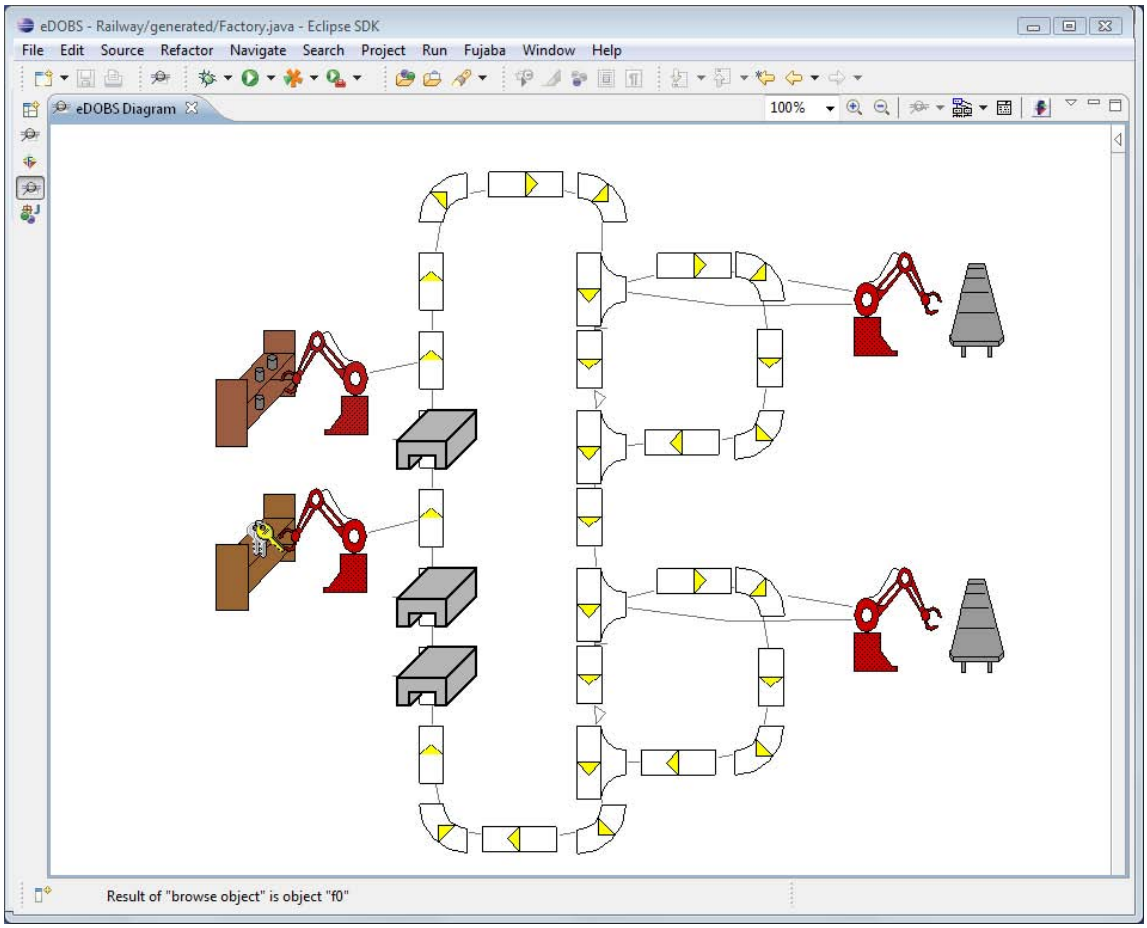


Abbildung 18.10.: Fabrikbeispiel im eDOBS

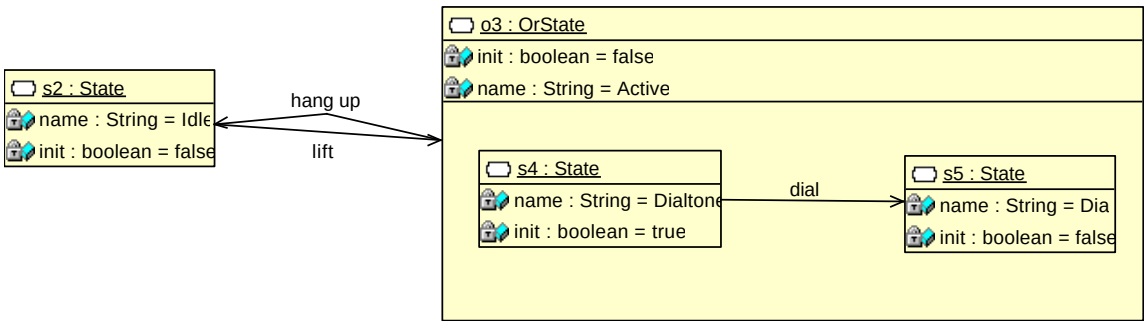


Abbildung 18.11.: eDOBS mit Abstraktionen

im eDOBS als sehr hilfreich erwiesen, da die Studenten probieren konnten, welche Funktionalität ein Modell bereitstellt und wie diese zu benutzen ist.

18.2.5. eDOBS Integrationen

Die Basisversion von eDOBS kommt als normales Eclipse Plugin, das eng in das JDT integriert ist. Das ermöglicht es, den Heap von normalen Java Programmen während einer Debuggingsession als UML Objektdiagramm anzuzeigen. Zusätzlich ist es möglich, im eDOBS Eclipse-interne Strukturen (also den Heap der Eclipse VM) zu untersuchen. Außerdem liegt eDOBS als Standalone-Version (einer sogenannten Rich Client Applikation) vor. Diese RCP Version kann sehr einfach in jede Java Applikation eingebunden werden, um schnell Zugriff auf die interne Datenstruktur zu ermöglichen entweder zu Darstellungs- oder zu Debuggingzwecken. Auf diese Art ist eDOBS beispielsweise in die letzte Standalone Version von Fujaba, Fujaba 5, integriert. Zusätzlich ist der eDOBS in den eHomeConfigurator [NM06] der Universität Aachen integriert. Der eHomeConfigurator dient der Konfiguration von Home-Automation-Anlagen und der eDOBS wird hier als Debugginghilfe verwendet.

Für den eDOBS existiert weiterhin ein Plugin, dass ihn in Fujaba4Eclipse integriert. Dieses Plugin bietet die geforderte Navigierbarkeit von eDOBS-Elementen (Objekten, Attributen, Methoden) zu den zugehörigen Fujaba Diagrammelementen und zurück. Weiterhin wird eine Funktion angeboten, die ein eDOBS Diagramm in Storypattern übersetzen kann. So kann man zu einer ausgewählten Objektstruktur im eDOBS eine Methode erzeugen lassen, die durch ein entsprechendes Storypattern diese Objektstruktur erzeugt.

Abbildung 18.12 zeigt eine so generierte `setUp()` Methode für das Statechart-Beispiel. Hier wurden die selektierten Objekte und Links aus dem eDOBS Diagramm in ein neu erzeugtes Storypattern übernommen. Zusätzlich wurden alle Objekte und Links im Storypattern mit dem ≪`create`≫ Stereotyp versehen. Attributzuweisungen müssen nach Erzeugung des Patterns separat über ein weiteres Kommando kopiert werden. Dies verhindert ein unnötiges Überfrachten des Patterns mit Attributzuweisungen.

Natürlich kann der beschriebene Mechanismus nicht nur zum Erstellen von Methoden zur Objektstrukturerzeugung benutzt werden. Er eignet sich auch hervorragend zum Erstellen von Storyboards. Der Entwickler kann ein Szenario im eDOBS durch Anlegen und Editieren einer Objektstruktur durchspielen und an geeigneten Punkten die aktuelle Struktur als Pattern dem zugehörigen Storyboard hinzufügen.

18.2.6. Der Zetteltest und die Fujaba Maschine

Der Entwickler hat mit den vorgestellten Ansatz Werkzeuge zur Hand, um Storydiagramme schrittweise ausführen und während dessen die Objektstruktur und deren Änderungen im eDOBS verfolgen zu können. Diese Werkzeuge eignen sich auch hervorragend für Einsteiger in die Objektorientierung mit Fujaba. Sie erleichtern es,

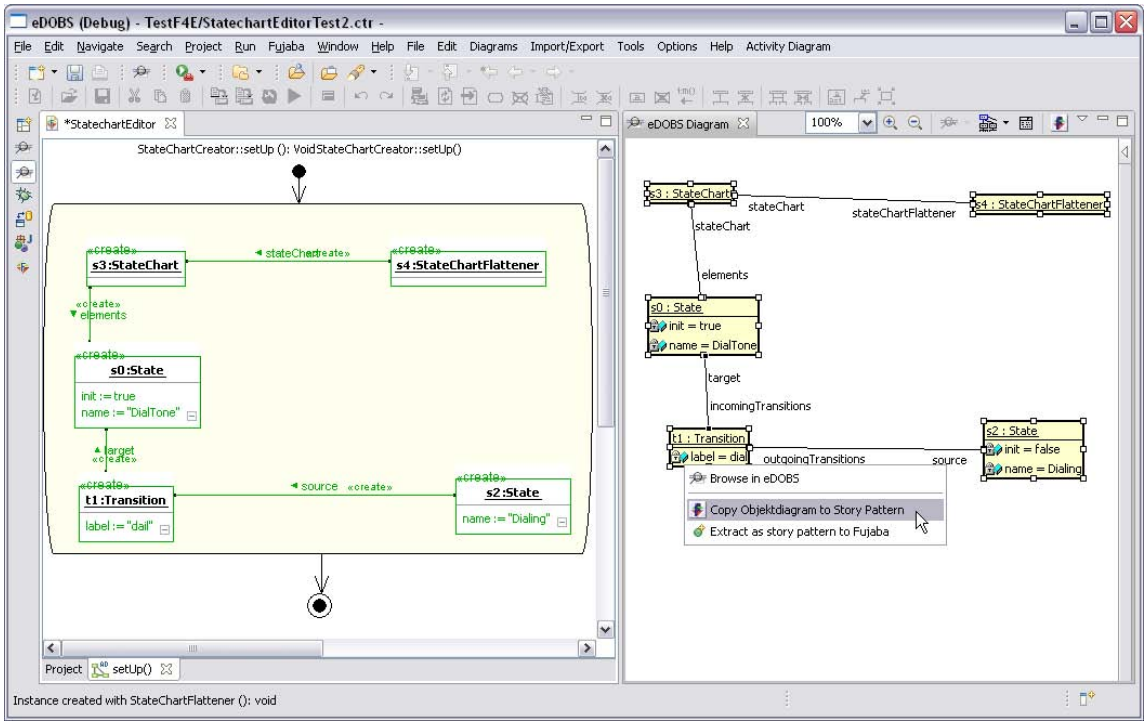


Abbildung 18.12.: Aus eDOBS Diagramm generierte Methode

fremde Storydiagramme zu verstehen und Fehler in eigenen Diagrammen zu finden. Trotzdem konnten wir bei unseren Studenten und Schülern oft noch Schwierigkeiten beim Verständnis bestimmter Storypattern ausmachen. Diese Lernhürden stellten wir insbesondere bei Storypattern mit komplizierten Konstrukten, wie beispielsweise Rekursion, fest.

Es stellte sich heraus, dass unter anderem das Konzept der lokalen Variable, das in Storydiagrammen eher implizit als Objektname auftaucht, Verständnisschwierigkeiten hervorruft. Um dieses Konzept in imperativen Programmierkursen zu vermitteln, werden meist Wertetabellen benutzt. Diese eignen sich jedoch für objektorientierte Datenstrukturen nicht, da sich mit Wertetabellen die Beziehungen der Objekte untereinander nicht gut darstellen lassen. Wir haben zur Lösung dieses Problems in [DGZ05a] den sogenannten Zetteltest entwickelt.

Hierbei werden der Methodenrumpf (also das Storydiagramm) und die aktuelle Objektstruktur nebeneinander an der Tafel oder am Smartboard dargestellt. Dann wird die Methode schrittweise ausgeführt. Der *Program Counter* wird dabei durch einen Klebezettel mit beispielsweise einer zeigenden Hand dargestellt. Der PC zeigt dabei immer auf das Objekt / den Link, der als nächstes gesucht, geprüft, erzeugt oder gelöscht wird. Ebenfalls kann der PC auf Attribute zeigen, die gelesen oder gesetzt, auf Constraints, die überprüft, oder auf Methodenaufrufe, die getätigt werden. Beim Aufruf einer Methode wird auf das Objekt in der Objektstruktur, auf dem die Methode aufgerufen werden soll, ein Klebezettel mit der Aufschrift `this` geklebt. Werden Objekte als Parameter übergeben, werden diese mit einem Klebezettel mit dem zugehörigen Parameternamen versehen. Wird nun während der Methodenausführung

ein Objekt beispielsweise durch eine Suche gebunden, wird es mit einem Klebezettel mit dem zugehörigen Variablennamen versehen. Änderungen an der Objektstruktur, die während der Methodenausführung passieren, wie zum Beispiel eine Objekterzeugung, werden sofort in der Objektstruktur an der Tafel nachvollzogen. So zeigt das Tafelbild immer die aktuelle Objektstruktur und durch die Klebezettel auch die aktuelle Variablenbelegung.

Abbildung 18.13 zeigt ein Beispiel dieses Vorgehens aus einer Anfängervorlesung. Das Beispiel zeigt zwei Methoden aus einer Implementierung des Malefiz Brettspiels. In diesem Beispiel werden geschachtelte Methodenaufrufe erklärt. Die Methode oben rechts ruft gerade die Methode unten rechts auf (zu erkennen an dem gelben PC Klebezettel). Um die unterschiedlichen Variablenbelegungen pro Methode (also die unterschiedlichen Stackframes) darzustellen, wurden für die lokalen Variablen der zweiten Methode Klebezettel einer anderen Farbe gewählt. Das Übereinanderkleben von Klebezetteln stellt hierbei eine Metapher des Prozedurenstapels dar.

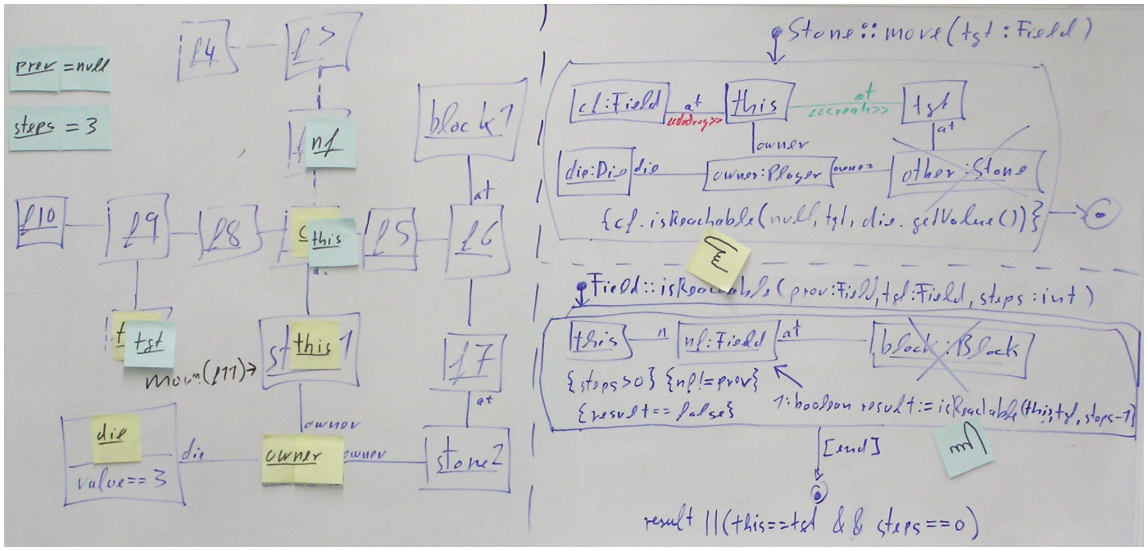


Abbildung 18.13.: Klebezettelmethode beim schrittweisen Durchspielen einer Methode

Die Klebezettelmethode lässt sich nicht nur gut an der Tafel durchführen, sondern eignet sich auch gut zur Veranschaulichung mittels Animationen. So existiert beispielsweise ein Satz Powerpoint-Folien[4], der zum Einstieg in Fujaba's Storydiagramme der Fujaba-Dokumentation beiliegt.

Im Rahmen dieser Arbeit wurde nun die Klebezetteldarstellung auch in den eDOBS integriert. Abbildung 18.14 zeigt den eDOBS mit dieser Funktionalität. Sobald im *Debug View* ein Stackframe selektiert ist, werden alle Objekte, die im eDOBS angezeigt werden und die einer lokalen Variable zugeordnet werden können, durch einen Zettel hinter dem Objektnamen erweitert, der den Namen der Variable anzeigt (in

[4] http://www.se.eecs.uni-kassel.de/se/fileadmin/se/courses/MSE/download/fujaba/FujabaTutorialStoryDrivenModeling.ppt

Abbildung 18.14 durch Pfeile markiert). Sollte ein Objekt in mehreren lokalen Variablen gespeichert sein, werden natürlich auch mehrere Zettel angezeigt. So ist beispielsweise das Objekt `s0` das aktuelle `this` Objekt und das Objekt `o3` ist in der lokalen Variable `or` abgelegt. Diese Visualisierung erleichtert auch die Zuordnung der Objekte im Fujaba Storydiagramm und der Objekte im eDOBS, wie in Abbildung 18.14 zu erkennen ist.

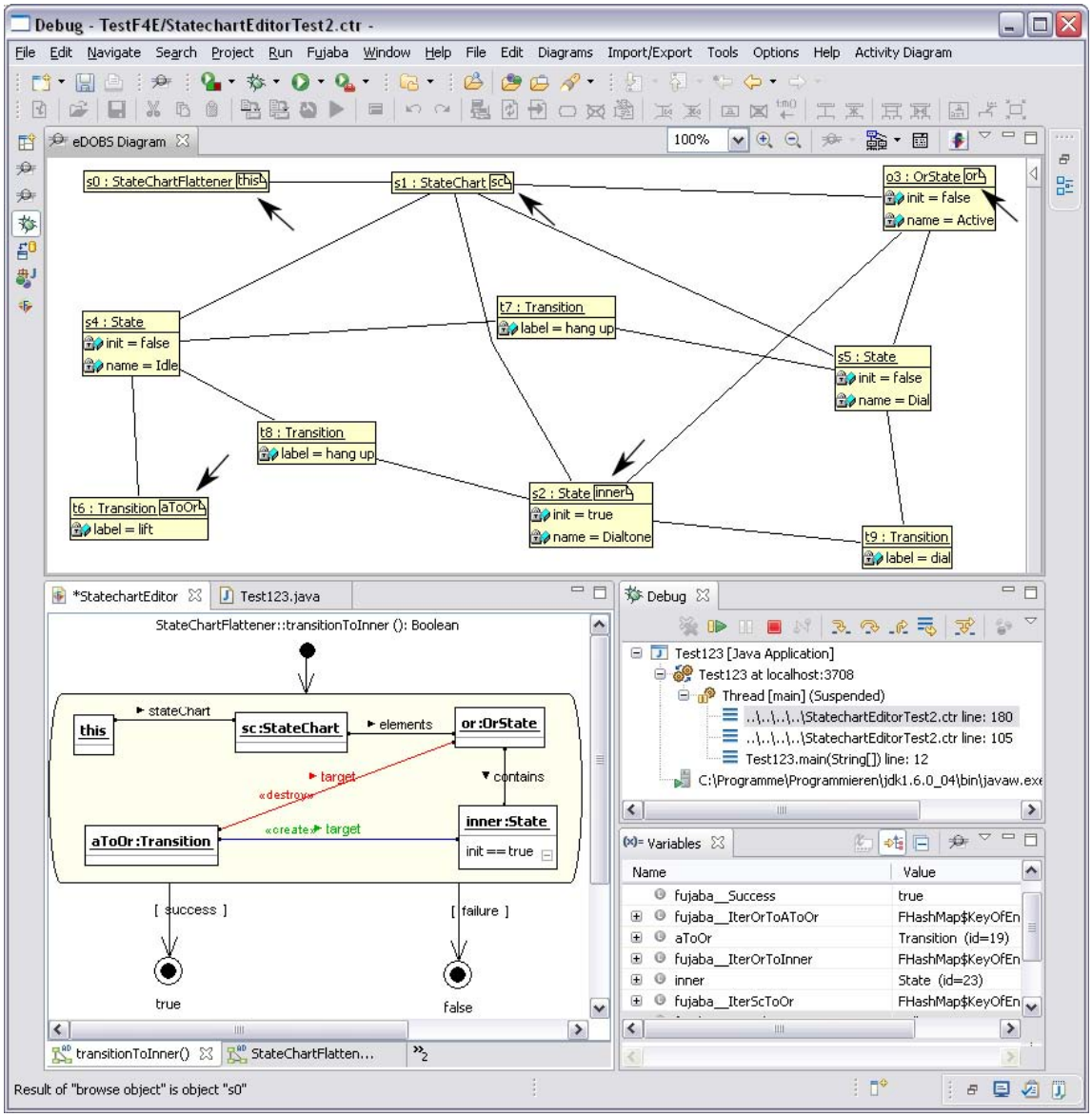


Abbildung 18.14.: Klebezettel in eDOBS

Durch die Zuordnung von eDOBS Objekten zu lokalen Variablen konnte auch die Integration von Fujaba4Eclipse und eDOBS weiter verbessert werden. So ist es jetzt beispielsweise möglich auf einem Objekt in einem Storypattern in Fujaba ein Kommando aufzurufen, dass in den lokalen Variablen nach dem zugehörigen Objektnamen sucht. Wird eine solche lokale Variable gefunden, wird der Wert der Variable im eDOBS angezeigt. Alternativ kann dieses Kommando auch auf einem Storypattern

ausgeführt werden, wobei alle enthaltenen Objekte im eDOBS angezeigt werden. Zusätzlich gibt es ein Layout Kommando, welches die Objekte im eDOBS genauso anordnet, wie in einem bestimmten Storypattern. Dadurch wird die Zuordnung zwischen eDOBS und Storypattern noch weiter vereinfacht.

Eine weitere Lernhürde stellt für unsere Schüler / Studenten die eingeschränkte Sicht eines Objektes auf den gesamten Objektgraphen dar. Ein Objekt kann in einer Methode natürlich nur auf Nachbarn zugreifen, zu denen es Links besitzt. Andere Objekte müssen mittels Durchsuchen der Objektstruktur erst gefunden werden. Diese Tatsache sorgt oft für Vorstellungsprobleme, da unsere Studenten ja immer die allwissende Draufsicht auf die Objektstruktur an der Tafel haben (vgl. Objektdiagramm in Abbildung 18.13). Um die eingeschränkte Sicht eines Objekts erlebbar zu machen, führen wir in [DGZ05a] das Objektspiel mit Augenbinde ein. Die Schüler spielen einen Programmteil durch, wobei jeder Schüler die Rolle eines Objekts wahrnimmt. Die Beschränkung der Sicht wird hierbei durch Verbinden der Augen der Schüler realisiert. Muss ein Schüler jetzt eine Aktion auf einem Objekt durchführen, mit dem er nicht direkt verbunden ist, muss er sich also „durchfragen“. Um diese Sichteinschränkung auch auf Folien oder an der Tafel verdeutlichen zu können, führen wir in [DGSZ06] den sogenannten *Fog Of Oblivion*, den Nebel des Vergessens, ein. Hierbei verschwinden alle Objekte, die in der aktuellen Ausführungssituation nicht bekannt sind und die noch nicht in lokalen Variablen gespeichert wurden, im Nebel des Vergessens, werden also halbtransparent dargestellt. Abbildung 18.15 zeigt das Malefiz Beispiel mit Nebel und Klebezetteln.

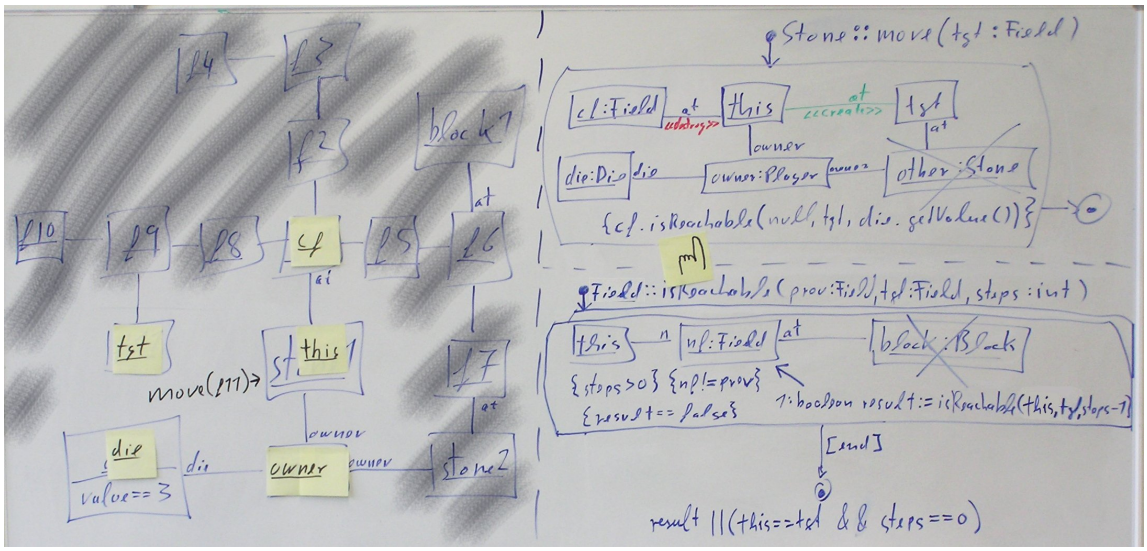


Abbildung 18.15.: Nebel des Vergessens

Auch diese Verständnishilfe wurde in den eDOBS integriert. Ist der Nebel des Vergessens aktiviert und ein Stackframe selektiert, werden alle Objekte, die keiner lokalen Variable zugeordnet werden können, halbtransparent dargestellt. Abbildung 18.16 zeigt dieses Verhalten für das Beispiel aus Abbildung 18.14.

In [DGSZ06] beschreiben wir ein komplettes Erklärungsmodell einer Virtuellen Maschine zur Ausführung von Storydiagrammen, die sogenannte Fujaba Maschine. Die-

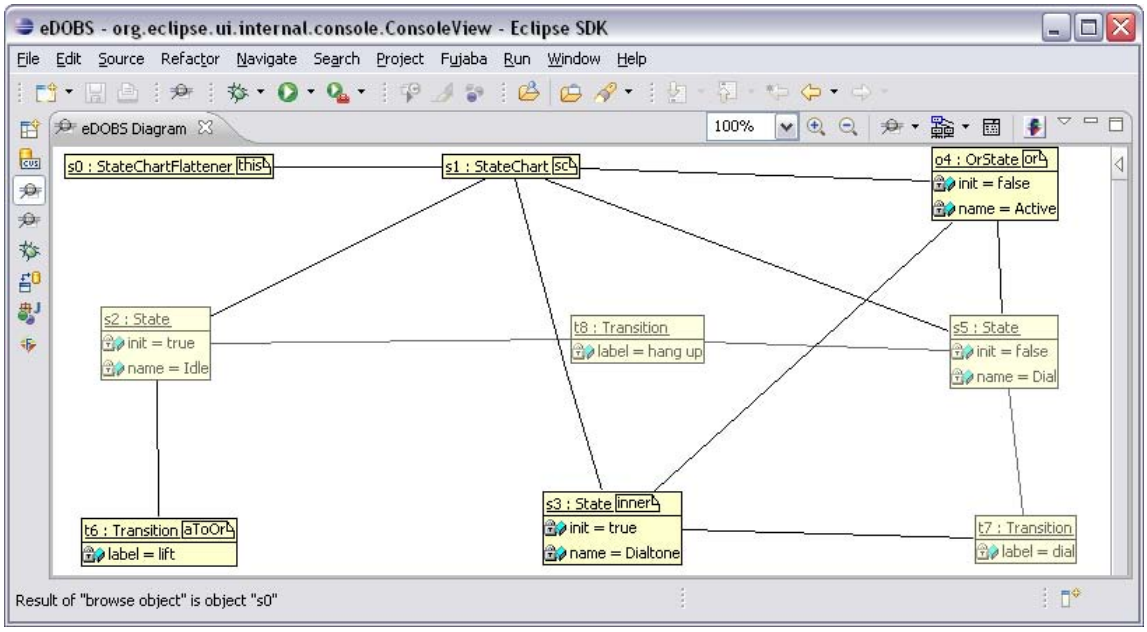


Abbildung 18.16.: Nebel des Vergessens in eDOBS

ses verwendet unter anderem die hier beschriebenen Metaphern der Klebezettel und des Nebels des Vergessens. Dieses Erklärungsmodell hilft nach unserem Beobachten den Schülern / Studenten Storydiagramme besser und schneller zu verstehen und räumt vorhandene Verständnishürden aus.

Durch die gerade dargestellten Erweiterungen des eDOBS um Visualisierungsmöglichkeiten für wesentliche Elemente der verwendeten virtuellen Maschine, wie Klebezettel, Program Counter und den Nebel des Vergessens, verschwinden für die Studierenden zahlreiche Lernhürden, die früher die Übertragung der mentalen Lernkonstrukte in die Werkzeugumgebung sehr erschwert haben.

18.2.7. Testintegration

Wie schon in Abschnitt 16 ausgeführt, stellen fehlschlagende Tests oft einen Startpunkt für Debuggingaktivitäten dar. Hier gilt es die Ursache für das Fehlschlagen des Testfalls zu finden. Es ist daher sinnvoll, Toolunterstützung für das Debuggen nach einem fehlgeschlagenen Test anzubieten.

Zu diesem Zwecke habe ich das eDOBS Plugin um eine Funktion erweitert, die nach einem fehlgeschlagenem Test, die aktuelle Objektstruktur im eDOBS darstellt. Hierzu wird ein Breakpoint im Konstruktor der Klasse `junit.framework.AssertionFailedError` hinzugefügt. Dieser `Error` wird immer dann ausgelöst, wenn eine Überprüfung des JUnit Frameworks einen Fehler ergab, also immer dann, wenn der Test selbst fehlschlägt. Hält die Ausführung des Tests bei diesem Breakpoint an, wird zuerst der Konstruktor durch eine *Step return* Anweisung des Debuggers verlassen, also zur eigentlichen Fehlerstelle gesprungen. Dann wird das aktuelle Testobjekt im eDOBS angezeigt und expandiert. Weiterhin wird

die Fehlermeldung des `Errors` in der eDOBS Statusleiste angezeigt. Anschließend wird das Testobjekt selbst wieder versteckt, da in der Regel nur die Nachbarn des Tests, also die getesteten Objekte, für die Fehleranalyse von Interesse sind. Zusätzlich wird der Stacktrace des `Errors` auf der Konsole ausgegeben. Mit Hilfe dieser Informationen ist eine genaue Analyse der Fehlersituation möglich.

Im Falle eines mit Fujaba modellierten oder aus einem Storyboard generierten JUnit Tests, ist natürlich auch die Navigation zurück ins Fujaba Modell wünschenswert. Dies erfordert insbesondere für die aus Storyboards generierten Tests zusätzlichen Aufwand. Da, wie in Teil II beschrieben, aus den Storyboards nicht sofort Java Code generiert wird, sondern erst Storydiagramme aus denen dann Code generiert wird, verweist das in diesem Kapitel vorgestellte Modell-Code Mapping auf die Elemente der generierten Storydiagramme. Dieses Mapping muss also so verändert werden, dass die Modellelementverweise nicht mehr auf die generierten Storydiagrammelemente verweisen, sondern auf die Storyboardelemente aus denen sie generiert worden sind. Hierzu wird während des Testgenerierungsprozess eine Map erstellt, die als Schlüssel die generierten Elemente und als Werte die zugehörigen generierenden Ausgangselemente enthält. In Abschnitt 12.2 wurde erläutert, dass zum Kopieren von Pattern aus dem Storyboard in die Storydiagramme der Testmethoden das CoObRA Framework verwendet wird. CoObRA erlaubt es nach der Kopie zu jedem kopierten Element das Original zu erfragen. Dieser Mechanismus wird hier verwendet um die Zuordnungsmap zu füllen. Die Codegenerierung wird nun um einen `CodeWriter` erweitert, der vor der Weiterverarbeitung des `DLRToken`-Baums, den Baum traversiert und alle Elemente, die in der Map als generiert verzeichnet sind, durch die zugehörigen Originale ersetzt. Der so veränderte Token-Baum wird dann zum generieren der SMAP-Dateien, zum Export und zur Navigation benutzt. Jetzt sind alle JUnit-Codefragmente direkt dem zugehörigen Storyboardelement zugeordnet. Somit ist es beispielsweise auch möglich, Breakpoints direkt in den Storyboards zu platzieren und dann schrittweise durch die Storyboards zu debuggen. Hält beispielsweise der Debugger auf dem im vorigen Absatz erwähnten Breakpoint im Konstruktor von `AssertionFailedError` an, wird automatisch die entsprechende Stelle im Storyboard, an der dieser `Error` erzeugt wurde, als aktuelle Programmposition angezeigt.

Abbildung 18.17 zeigt die beschriebene Testintegration nach einem fehlgeschlagenen Test. Hierbei handelt es sich um den Test der aus dem Storyboard zum Usecase Flattening aus Abbildung 5.3 generiert wurde. Im oberen View, dem Fujaba4Eclipse Editor, wird automatisch das Storypattern markiert und angezeigt, dass zum Fehlschlagen des Tests geführt hat. Im unteren View, dem *eDOBS View*, wird die aktuelle Objektstruktur visualisiert. Diese ist, um die Objekte besser zuordnen zu können, analog zum zugehörigen Storypattern gelayoutet. Zusätzlich wird die Fehlermeldung „iterate to-many link target from s1 to t2_1“ in der eDOBS Statuszeile angezeigt. Es ist nun sehr einfach zu erkennen, dass kein passender Kandidat für das Objekt `t2_1` gefunden werden konnte. Das zuletzt als `t2_1` belegte Objekt verweist über den `source` Link nicht auf das Objekt `s2` sondern auf `s3`. Es ist also ein Kandidat für das Objekt `t2_2`. Nun gilt es für den Entwickler herauszufinden, warum zwar das Objekt `t2_2` angelegt wurde, nicht aber das Objekt `t2_1`. Eine Hilfestellung für diese Suche wird im nächsten Kapitel vorgestellt.

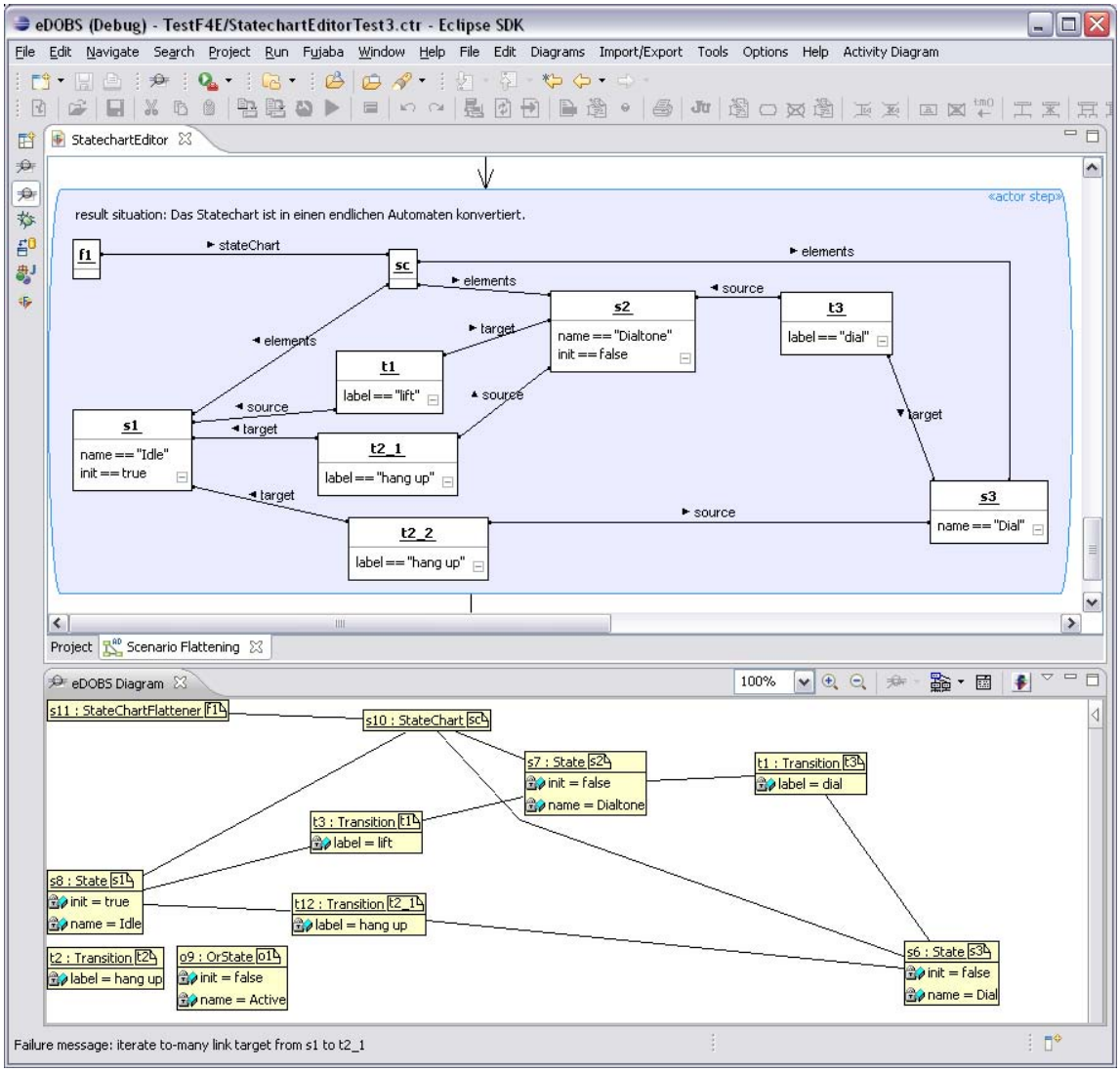


Abbildung 18.17.: eDOBS nach fehlgeschlagenem Test

Zusätzlich wird zur weiteren Fehleranalyse auch noch der Stacktrace des `AssertionFailedErrors` ausgegeben. Dieser enthält unter Umständen weitere Informationen warum beispielsweise ein bestimmtes Objekt nicht gefunden werden konnte (siehe Abschnitt 12.3). Leider ist die Java Virtual Maschine (JVM) nicht in der Lage, Stacktraces aufgrund der SMAP Informationen zur Laufzeit umzuwandeln. Dieser Fehler in der JVM ist im Bug Report Nummer 4972961[5] verzeichnet und wird hoffentlich in kommenden JVM Versionen berichtigt. Bis dahin verweisen die ausgegebenen Stacktraces von Exceptions also auf Quelltextzeilen. Von diesen kann man allerdings durch die in diesem Kapitel vorgestellten Navigationsmöglichkeiten mit einem Klick zum zugehörigen Modellelement springen.

[5] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4972961

18.2.8. Back-in-time Debugger

In den vorangegangenen Abschnitten wurden Methoden präsentiert, die es ermöglichen, eine Applikation auf Modellebene zu debuggen. Es wurde der eDOBS vorgestellt, der die aktuelle Objektstruktur und deren Änderungen während einer Debuggingsession als Objektdiagramm visualisieren kann. Liegt nun ein Fehler in der zu entwickelnden Applikation vor, kann der Entwickler das Programm an einer verdächtigen Stelle anhalten, von dort schrittweise ausführen und die aktuelle Objektstruktur und die Variablenbelegung im eDOBS analysieren, bis er den Fehler im Programm entdeckt hat. Eine Schwierigkeit liegt hier aber darin, diese „verdächtige“ Stelle zu finden, an der es sinnvoll ist, den initialen Haltepunkt zu setzen. Meistens kennt der Entwickler ja nur die Wirkung des Fehlers und hat wenig Information darüber, wo dieser Fehler passiert sein könnte. Fehlt in der Endsituation eines Tests beispielsweise ein bestimmter Link, ist es schwer herauszufinden, wann und wo dieser Link gelöscht wurde. Es kann in dem Programm viele Stellen geben, an denen ein solcher Link gelöscht wird.

Betrachtet man beispielsweise den Testfall aus dem letzten Abschnitt in Abbildung 18.17, ist eine Frage, die sich bei der Suche nach der Fehlerursache stellen könnte: „Warum wurde Objekt `t2_2` angelegt, Object `t2_1` aber nicht?“. Um diese Frage beantworten zu können ist sicherlich erstmal spannend, wann und wo das Objekt `t2_2` überhaupt angelegt worden ist. Es wird also eine Möglichkeit gesucht, beim Debuggen (rückwärts) durch die Zeit zu bestimmten Ereignissen navigieren zu können. Diese Möglichkeit zum Navigieren vorwärts und rückwärts durch die Zeit bietet das Flipbook Plugin. Das Flipbook Plugin wurde im Rahmen der von mir betreuten Diplomarbeit von Jörn Dreyer implementiert [Dre08].

Logging

Um ein Navigieren zurück und vor in der Zeit zu erlauben, müssen alle relevanten Informationen während eines Debuglaufes mitprotokolliert werden, damit diese später rückgängig gemacht, beziehungsweise erneut ausgeführt werden können. Die Informationen, die für das in dieser Arbeit vorgestellte Vorgehen insbesondere interessant sind, sind die Änderungen an der Objektstruktur. Das in [Sch07, SZN04] vorgestellte Persistenzframework für Objektstrukturen CoObRA protokolliert genau diese Informationen. Zum Speichern von Objektstrukturen werden sämtliche „atomaren“ Operationen, also Objekt- bzw. Linkerzeugungen und -löschungen sowie alle Attributzugriffe, als sogenannte Changes über die Zeit protokolliert. Dieses Vorgehen erlaubt ein späteres Wiederherstellen der Objektstruktur durch erneutes Ausführen der Changes sowie ein generisches Undo-Redo-Konzept durch Ausführen bzw. Rückgängigmachen von Changes. Weiterhin stellt CoObRA Mechanismen zur Mehrbenutzernutzung von Objektstrukturen mit optimistischem Locking, sowie ein generisches Konzept zum Kopieren von Teilstrukturen (siehe Abschnitt 12.3) bereit.

Abbildung 18.18 zeigt die wichtigen Klassen und Interfaces des CoObRA Frameworks. Wie bereits erwähnt, stellt die zentrale Klasse `Change` eine atomare Operation dar. Sie verfügt über ein `kind` Attribut, das die Art der Änderung beschreibt, also

Erzeugung, Löschung oder Änderung. Außerdem kann das `kind` Attribut noch den Wert `MANAGE` annehmen. Dann handelt es sich um einen sogenannten *Management Change*, der keine atomare Operation beschreibt, sondern Zusatzinformationen über das Modell speichert, wie z.B. Modellversionsnummern. Die `affectedObject` Assoziation der Klasse `Change` enthält die Information, welches Objekt geändert wurde, die Assoziationen `newValue` und `oldValue` enthalten den alten bzw. den neuen Wert, falls vorhanden, und die `key` Kante speichert den Schlüssel bei Operationen an qualifizierten Assoziationen. Weiterhin ist über die `field` Kante das Attribut bzw. die Rolle erreichbar, auf die die Operation sich bezieht. Hier wird ebenfalls die im eDOBS verwendete und in Abschnitt 18.2.2 beschriebene Modellabstraktionsschicht *Java Feature Abstraction* verwendet, wodurch CoObRA auf einer Vielzahl von Modellen arbeiten kann wie z.B. auf EMF oder MOF Modellen. Changes können in Transaktionen zusammengefasst werden, die durch Instanzen der Klasse `Transaction` repräsentiert werden. Sämtliche Changes werden in einem `Repository` abgelegt. Diese Klasse bietet unter anderem Methoden zur Navigation in den Changes, wie etwa die Methoden `undo()` und `redo()` zum Rückgängigmachen bzw. Wiederherstellen von Changes.

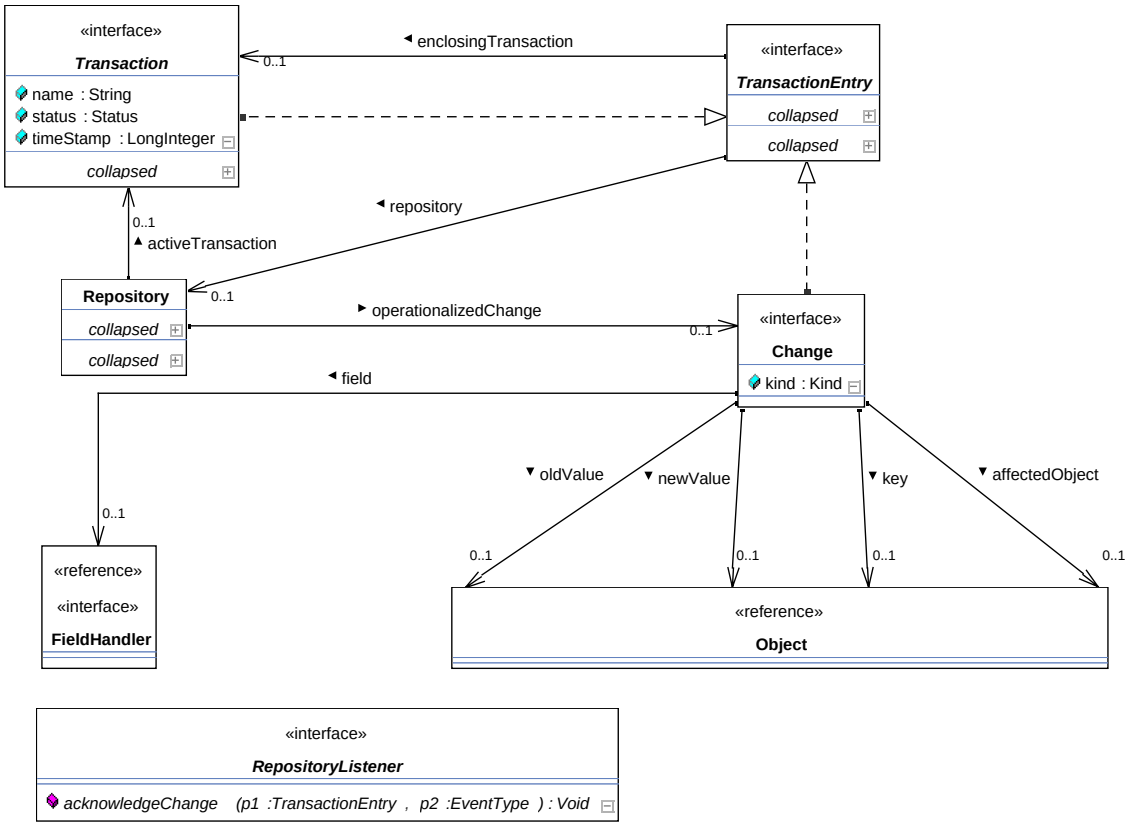


Abbildung 18.18.: Das CoObRA Framework

Um ein Modell mit CoObRA persistent zu machen, müssen neue Objekte am Repository angemeldet werden und das Repository über alle Änderungen an diesen Objekten informiert werden. Verfügt das Modell über einen Mechanismus zur Änderungsnotifikation, der von CoObRA unterstützt wird, wie z.B. der durch den Java

Beans Standard beschriebene `PropertyChangeSupport`, erfolgt diese Änderungsnotifikation automatisch, sobald ein Objekt beim Repository angemeldet ist. Da die Fujaba Codegenerierung das Generieren von Java Beans konformem Code unterstützt, ist es sehr einfach ein Fujaba Modell CoObRA-persistent zu machen: Alle persistenten Klassen werden mit dem ≪`Java Beans`≫ Stereotyp versehen, der die Java Beans konforme Codegenerierung aktiviert. Zusätzlich müssen diese Klassen durch einen Aufruf im Konstruktor oder im Konstruktor einer Oberklasse ergänzt werden, der das Objekt beim Repository anmeldet. Da sich in Fujaba Modelle so einfach mit CoObRA persistent machen und dadurch mit Undo-Redo-Funktionalität und Mehrbenutzerfähigkeit versehen lassen, wird CoObRA bereits in vielen Fujaba Projekten eingesetzt. Es liegt daher nahe eben diesen Mechanismus auch zum benötigten Protokollieren von Änderungen an der Objektstruktur zu verwenden.

CoObRA speichert in seinen Change-Protokollen was an der Objektstruktur geändert wurde und wann (im Vergleich zu anderen Änderungen) dies geschah. Eine Information, die nicht mitprotokolliert wird, die aber für das Rückwärts-Debuggen durchaus von Interesse ist, ist wo im Quellcode diese Änderung geschah. Ohne diese Information lässt sich zwar nachvollziehen, was geschehen ist, sollte dabei aber beispielsweise eine fehlerhafte Änderung entdeckt werden, ist es nicht möglich herauszufinden, in welcher Methode diese fehlerhafte Operation auf der Objektstruktur durchgeführt wurde. Es wird also die Information benötigt, in welcher Methode die aktuelle Änderung vorgenommen wurde.

Zu diesem Zweck enthält das Flipbook-Plugin die Klasse `StackTraceRecorder`. Das CoObRA Framework bietet die Möglichkeit beim Repository Objekte, die das `RepositoryListener` Interface implementieren, anzumelden. Diese werden informiert, sobald eine neue Änderung hinzugefügt wird. Die Klasse `StackTraceRecorder` implementiert das `RepositoryListener` Interface. Wird der `StackTraceRecorder` über einen neuen Change informiert, wird über die Methode `Exception.printStackTrace()` der Stacktrace des aktuellen Threads erfragt. Da die `RepositoryListener` synchron informiert werden und auch die `PropertyChangeSupport` Implementierung des Java Beans Standard synchron ist, enthält der Stacktrace auch die Methodenaufrufe, die die aktuelle Änderung bewirkt haben.

Listing 18.3 zeigt einen (vereinfachten) Beispiel-Stacktrace aus dem `StackTraceRecorder`. In Zeile 1 erkennt man die aktuelle Position des Program Counters in der Klasse `StackTraceRecorder`, die nächsten Zeilen zeigen Aufrufe des CoObRA Frameworks und der `PropertyChangeSupport` Implementierung. In Zeile 10 ist die eigentliche Änderungsstelle dargestellt. Hier wurde das `init` Attribut durch Aufruf der zugehörigen Zugriffsmethode gesetzt. Die nachfolgenden Zeilen beschreiben, dass diese Änderung in der Methode `removeOuter` (Zeile 11) stattfand, die wiederum von der Methode `flattenStateChart` (Zeile 12) gerufen wurde. Jeder Stacktraceeintrag enthält neben Klassen- und Methodennamen auch noch die genaue Zeilennummer, an der der Program Counter zur Zeit der Erstellung des Stacktraces in der entsprechenden Methode stand.

```
1  StackTraceRecorder.acknowledgeChange(StackTraceRecorder.java:42)
2  Repository.notifyListeners(Repository.java:403)
3  Repository.acknowledgeChange(Repository.java:320)
4  AbstractChangeRecorder.processChange(AbstractChangeRecorder.java:152)
5  PropertyChangeRecorder.propertyChange(PropertyChangeRecorder.java:102)
6  Listener.propertyChange(PropertyChangeRecorder.java:118)
7  PropertyChangeSupport.firePropertyChange(PropertyChangeSupport.java:339)
8  PropertyChangeSupport.firePropertyChange(PropertyChangeSupport.java:276)
9  PropertyChangeSupport.firePropertyChange(PropertyChangeSupport.java:318)
10 State.setInit(State.java:169)
11 StateChartFlattener.removeOuter(StateChartFlattener.java:149)
12 StateChartFlattener.flattenStateChart(StateChartFlattener.java:30)
```

Listing 18.3: Stacktrace einer CoObRA Änderungsoperation

Der so gewonnene Stacktrace wird durch Objekte der Klasse `StackTraceFilterStrategy` von einigen Einträgen, die für den Benutzer uninteressant sind, gereinigt. Hier werden die Einträge des CoObRA Frameworks und der Java Beans Implementierung entfernt. Der verbleibende Stacktrace wird als Management Change direkt hinter dem eigentlichen Change zum Repository hinzugefügt. Auf diese Art werden Stacktraceinformationen zu jedem Change mitgespeichert. Durch diese zusätzlichen Stacktraceinformationen werden nun alle benötigten Informationen im Repository abgelegt. Die Changes eines Repositories lassen sich nun beispielsweise in eine Datei speichern, die das Flipbook-Plugin später als Änderungsprotokoll laden kann.

eDOBS Cache Layer

Wie im vorigen Abschnitt beschrieben, stellt CoObRA die nötigen Änderungsprotokolle zur Navigation in der Objektstruktur über der Zeit bereit. Nun muss der eDOBS insofern erweitert werden, als dass er diese Protokolle lesen und verarbeiten kann, um die Änderungen auch zu visualisieren. Zwar wäre es auch möglich direkt die Undo-Redo-Funktionalität des Repositories zu verwenden, die Changes also zur Rückwärtsnavigation *tatsächlich* im Modell rückgängig zu machen und zur Vorwärtsnavigation die Changes *tatsächlich* wieder herzustellen, aus folgenden Gründen wurde allerdings auf dieses Vorgehen verzichtet:

- Bei diesem Vorgehen wird tatsächlich am Modell geändert. Je nach Modell können solche Änderungen unter Umständen Nebeneffekte haben, wie beispielsweise Datei- oder Datenbankzugriffe. Solche Nebeneffekte sind in der Regel nicht erwünscht und können zusätzlich die Navigation in den Changes enorm verlangsamen.
- Da die zu untersuchende Applikation im Debugger läuft, erfolgen alle Anwendungen von Änderungen, sowie die Analyse der Objektstruktur über das Debuginterface des JDKs. Solche Zugriffe sind langsam und erschweren somit eine performante Navigation in den Changes.
- Trennt man das Erstellen der Logdateien und die Analyse im eDOBS, ist es beispielsweise möglich, die zu untersuchende Applikation auf einem Rechner

laufen zu lassen und die Analyse der dort erstellten Protokolle auf einem anderen Rechner vorzunehmen. Dies ist insbesondere dann hilfreich, wenn Ziel- und Entwicklungsplattform unterschiedlich sind. Es kann sogar der Fall auftreten, dass der eDOBS nur die Protokolldatei kennt und der Quellcode bzw. das Meta-Modell der Applikation nicht vorliegt. Im Falle der Interpretation der CoObRA Protokolle im eDOBS stellt aber sogar das kein Problem dar.

Die aufgezählten Gründe zeigen, dass es wünschenswert ist, nicht den CoObRA internen Undo-Redo-Mechanismus zur Change-Navigation zu verwenden, sondern diese Funktionalität gesondert im eDOBS anzubieten. Es muss also möglich sein, im eDOBS Objektstrukturen darzustellen, die so nicht im Speicher existieren und zu denen unter Umständen noch nicht einmal das Meta-Modell bekannt ist. Da der eDOBS als reiner Viewer für Speicherstrukturen konzipiert ist, gab es diese Möglichkeit bisher noch nicht. Diese Arbeit löst dieses Problem, indem sie eine Caching-Schicht einführt, auf die auch schreibend zugegriffen werden kann. Der Cache, der die Objekte und deren Attributwerte enthält, kann also entweder aus bestehenden Objektstrukturen im Speicher, oder aber auch aus CoObRA Protokollen gefüllt werden.

Ein weiterer Vorteil dieses Ansatzes ist, dass Fujaba intern zur Persistenz auch das CoObRA Framework verwendet. Existiert also eine Möglichkeit beliebige CoObRA Protokolle im eDOBS analysieren zu können, ist dieses auch mit Fujaba Dateien möglich. Da beispielsweise durch Änderungen am internen Fujaba-Meta-Modell von Zeit zu Zeit defekte Fujaba Dateien entstehen, ist es für die Fujaba Entwickler sehr hilfreich ein Werkzeug zur Analyse dieser Dateien zur Hand zu haben. Mit diesem Ansatz ist es also möglich, defekte Fujaba Modelldateien zu öffnen und den Fehler darin mithilfe des eDOBS zu finden.

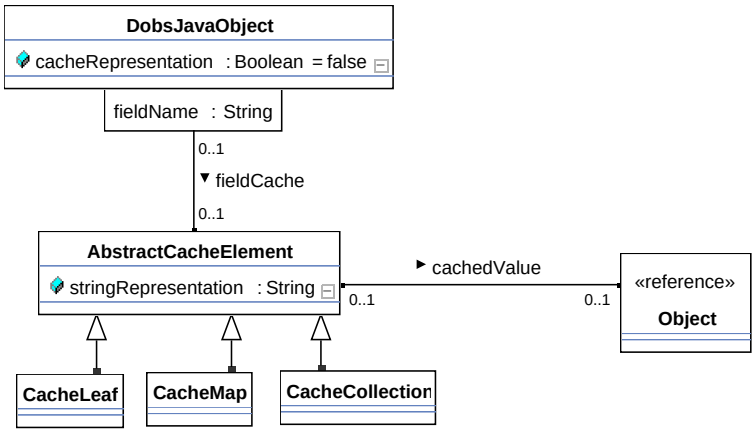


Abbildung 18.19.: Klassendiagramm der Caching-Schicht

Abbildung 18.19 zeigt ein Übersichtsklassendiagramm der Caching-Schicht. Ein Java-Objekt im eDOBS wird durch die Klasse `DobsJavaObject` repräsentiert. Diese wurde durch das boolesche Attribut `cacheRepresentation` erweitert. Dieses Attribut gibt an, ob es sich bei dem `DobsJavaObject` um ein reines Cache-Objekt handelt, wie es bei der Visualisierung von CoObRA Protokollen der Fall ist, oder ob es sich um

eine Abbildung einer existierenden Objektstruktur handelt, bei der die Cacheschicht durch die reale Struktur aktualisiert wird. Über die Assoziation `fieldCache` ist von der Klasse `DobsJavaObject` die Caching-Schicht erreichbar, die durch Unterklassen der Klasse `AbstractCacheElement` gebildet wird. Diese Assoziation ist über den Namen des im Cache hinterlegten Feldes qualifiziert. Ein `AbstractCacheElement` speichert eine String-Repräsentation des gecachten Wertes zur Anzeige im User Interface wie zum Beispiel im *Attribute View*. Zusätzlich ist der eigentliche Wert über die `cachedValue` Assoziation erreichbar. Die Caching-Schicht entpackt bereits mengenwertige Felder. Eine einfache Menge an Werten wird durch eine `CacheCollection` dargestellt, eine Tabelle durch eine `CacheMap`. Objekte dieser Klassen enthalten wiederum `AbstractCacheElements` als Kinder. Ein Blatt in einem solchen Cache-Baum wird durch die Klasse `CacheLeaf` implementiert.

Durch die beschriebene Caching-Schicht lassen sich im eDOBS Objektstrukturen darstellen, die so nicht im Speicher existieren. Zusätzlich beschleunigt sie die Darstellung von Objektstrukturen im eDOBS, da das Auslesen von Attributwerten mitunter langsam ist, zum Beispiel wenn dies über das Debuginterface geschieht. Durch die Caching-Schicht muss das Auslesen der Attributwerte nach einer Änderung aber nur einmal zentral passieren und nicht für jeden Zugriff des User Interface auf einen Attributwert erneut erfolgen.

Die eingeführte Caching-Schicht eignet sich also zum Darstellen von CoObRA Protokollen. Es wird lediglich zusätzlich ein Mechanismus benötigt, der in der Lage ist, CoObRA Protokolle in die Caching-Schicht des eDOBS zu laden.

eDOBS Cache Feature Abstraction Module

Wie bereits in Abschnitt 18.2.8 erwähnt, benutzt CoObRA die Modellabstraktionsschicht *Java Feature Abstraction* um Änderungsoperationen auszuführen. Tauscht man die Modellabstraktionsschicht von CoObRA durch eine aus, die direkt auf der eDOBS Caching-Schicht arbeitet, kann man den normalen CoObRA Lademechanismus zum Import eines Protokolls in die Caching-Schicht benutzen und die CoObRA Undo-Redo-Funktionalität zum Navigieren über die Änderungen verwenden.

Ein solches *Cache Feature Abstraction Module* wurde im Rahmen dieser Arbeit implementiert. Die wichtigsten Klassen dieser Abstraktionsschicht sind im Klassendiagramm in Abbildung 18.20 dargestellt. Es wurde ein `FactoryModule` (vgl. Abbildung 18.8) geschrieben, das spezielle `ClassHandler` erzeugt. Diese `CacheClassHandler` legen beim Aufruf der `newInstance()` Methode *kein* Objekt der entsprechenden Klasse an, sondern erzeugen ein `DobsJavaObject`. Die `FieldHandler`, die ein `CacheClassHandler` zurückliefert, schreiben dementsprechend in die Caching-Schicht eines `DobsJavaObjects`. Lesende Zugriffe erfolgen ebenfalls über die Caching-Schicht.

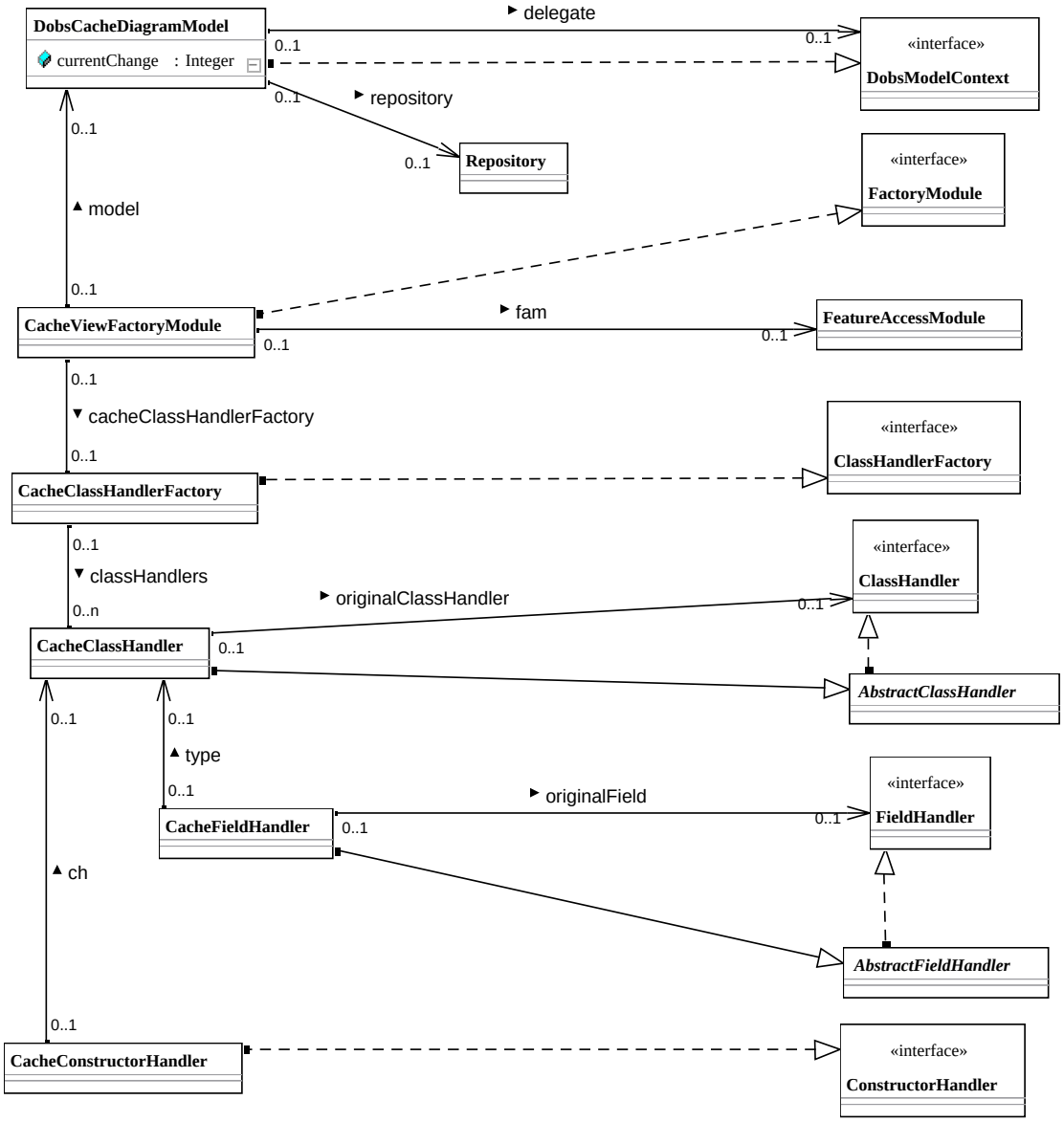


Abbildung 18.20.: Klassendiagramm des *Cache Feature Abstraction Module*

Jeder `CacheClassHandler` kennt den `ClassHandler`, den er ersetzt, also den `ClassHandler` der eigentlichen Klasse, sofern dieser existiert. Liegen beispielsweise die class-Dateien der entsprechenden Klasse nicht vor, kann ein solcher `ClassHandler` nicht erzeugt werden. In diesem Fall hat der `CacheClassHandler` erstmal keine Informationen außer dem zugehörigen Klassennamen, da CoObRA keine Strukturinformationen speichert. Bei jeder Attributänderung, die diese Klasse betrifft und aus einem CoObRA Protokoll gelesen wird, wird aber zumindest das entsprechende Attribut (also der entsprechende `CacheFieldHandler`) der Klasse hinzugefügt.

Existiert allerdings der `ClassHandler` der eigentlichen Klasse, werden alle Anfragen, die der `CacheClassHandler` selbst nicht beantworten kann, wie z.B. Sichtbarkeit, Superklassen usw., an den originalen `ClassHandler` weitergeleitet. So können z.B. auch `MethodHandler` erzeugt werden. Diese können allerdings nur dargestellt, nicht aber aufgerufen werden, da ja kein echtes Objekt der Klasse existiert.

Mit dem vorgestellten *Cache Feature Abstraction Module* können nun also CoObRA Protokolle zur Analyse im eDOBS in die Caching-Schicht geladen und dann im eDOBS angezeigt werden.

Eclipse Integration

In den vorangegangenen Abschnitten wurde die Möglichkeit vorgestellt im eDOBS CoObRA Protokolle anzeigen und durch diese navigieren zu können. Um diese Funktionalität auch nutzbar zu machen, sind zusätzliche UI Elemente nötig. So wurde im Rahmen dieser Arbeit ein zusätzlicher Eclipse View, der sogenannte *Flipbook View*, implementiert der die zusätzlichen UI Elemente vereint.

So enthält dieser View beispielsweise einen Button zum Laden von CoObRA Protokollen. Wird ein solches Protokoll geladen, wird im eDOBS ein neues Diagramm angelegt und alle Changes werden eingespielt. Die Cache-Schicht befindet sich also im Zustand am Ende der Protokollerstellung. Alle bis dahin erstellten Objekte werden dem eDOBS Diagramm als versteckte Elemente hinzugefügt um das Diagramm nicht zu überfrachten. Es können jetzt einzelne Objekte angezeigt und expandiert werden. Auf diese Weise kann der interessante Teil der Objektstruktur im Diagramm visualisiert werden.

Um nun die Änderungen einer ausgewählten Objektstruktur über die Zeit zu verfolgen, stellt der *Flipbook View* unter anderem einen Slider bereit. Durch ziehen dieses Sliders nach links bzw. rechts werden Changes rückgängig gemacht bzw. wiederhergestellt. Dadurch kann der Entwickler sich sehr einfach „durch die Zeit scrollen“. Zusätzlich bietet der View Kommandos an, um einzelne Changes oder ganze CoObRA Transaktionen rückgängig zu machen bzw. wiederherzustellen.

Zur Navigation in den Changes stehen aber noch weitere Kommandos zur Verfügung. Oft stellen sich dem Entwickler bei der Analyse der Objektstruktur Fragen über die Zeit wie: „Wann wurde dieses Objekt erstellt?“, „Wann wird dieser Link zerstört?“ oder „Wann wurde dieses Attribut auf den aktuellen Wert gesetzt?“. Für all diese

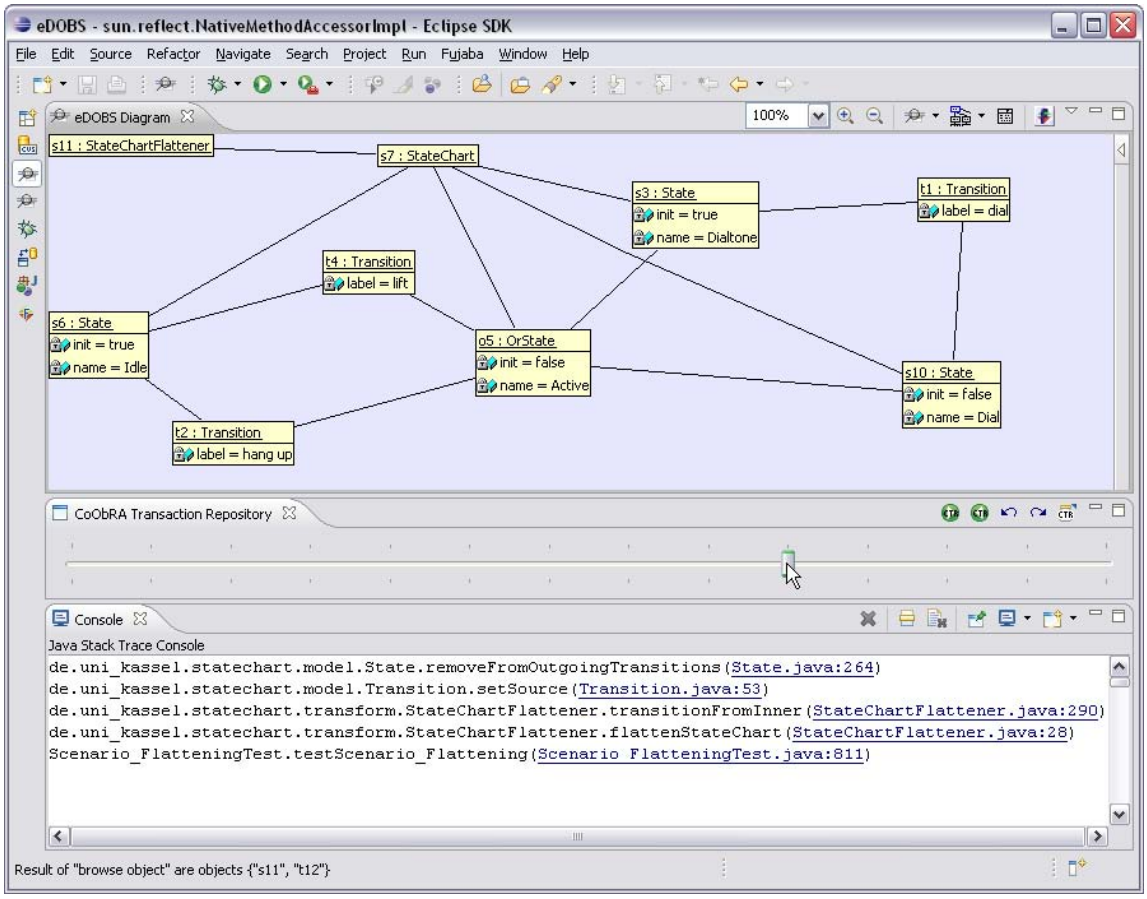


Abbildung 18.21.: Der *Flipbook View*

Fragen bietet das Flipbook Navigationskommandos an. So ist es beispielsweise möglich auf einem Objekt im eDOBS in einer Flipbook-Session zu fragen, wann dieses Objekt erzeugt wurde. Die Änderungen werden dann automatisch bis zum Zeitpunkt der Erzeugung des ausgewählten Objekts rückgängig gemacht.

Um jetzt auch noch die Frage beantworten zu können, an welcher Stelle in den Storydiagrammen die aktuelle Änderung vorgenommen wurde, ist es nötig, die protokollierten Stacktrace-Informationen (siehe Abschnitt 18.2.8) zu visualisieren. Zur Anzeige von Stacktrace-Informationen bietet Eclipse die *Java Stack Trace Console* im *Console View*. Diese benutzt auch das Flipbook Plugin. Wird ein Change im eDOBS rückgängig gemacht oder wiederhergestellt, wird der zugehörige Stacktrace in der *Java Stack Trace Console* angezeigt. Die Datei- und Zeileninformationen werden hier als Html-Links dargestellt. Klickt man auf einen solchen Link, öffnet Eclipse automatisch die entsprechende Stelle im Quellcode bzw. im Modell.

Abbildung 18.21 zeigt den *Flipbook View* sowie den eDOBS im Flipbook-Modus. Der oberste View zeigt die Objektstruktur an der aktuellen Position im CoObRA-Protokoll. Da ein eDOBS Diagramm im Flipbook-Modus kaum von einem „echten“ eDOBS Diagramm zu unterscheiden ist, kam es bei ersten Versuchen mit dem neuen Flipbook öfter zu Missverständnissen, was denn nun die reale Objektstruktur und was das zugehörige Protokoll sei. Daher haben wir uns entschlossen, den Hintergrund

im Flipbook-Modus blau einzufärben. Der View darunter ist der oben erwähnte *Flipbook View*. Der Benutzer ist hier gerade dabei durch Ziehen des Sliders „durch die Zeit zu reisen“. In diesem View sind auch die oben beschriebenen Kommandos in der Toolbar untergebracht. Darunter wiederum befindet sich der *Console View*, der den Stacktrace der aktuellen Änderung anzeigt. Durch Klicken auf die blau hinterlegten Links gelangt man zur zugehörigen Stelle im Modell.

Testintegration

Um eine Applikation im Flipbook analysieren zu können, sind, wie beschrieben, die folgenden Voraussetzungen nötig. Das Modell muss CoObRA-persistent sein, die Applikation muss einen `StackTraceRecorder` beim Repository anmelden und die Applikation muss ein CoObRA Protokoll in eine Datei schreiben. Diese Datei kann dann im eDOBS geöffnet und das enthaltene Protokoll analysiert werden.

Die erste Anforderung der CoObRA-Persistenz des Modells ist bei Fujaba-generierten Modellen sehr leicht zu realisieren. Die meisten unserer Anwendungen verwenden sowieso CoObRA als Persistenz-Schicht. Die Tatsache, dass man das Speichern des Protokolls in der Applikation explizit implementieren muss, kann aber mitunter etwas umständlich sein. Um mit CoObRA eine Datei zu speichern, muss an geeigneter Stelle ein entsprechender API Aufruf abgesetzt werden, dem unter anderem der Dateiname, unter dem gespeichert werden soll, mitgegeben wird. Allerdings weiß der Entwickler in der Regel zu Beginn einer Debugging-Session nicht, ob und an welcher Stelle er eine Back-In-Time Analyse vornehmen möchte. Alternativ kann der Benutzer auch das Kommando zum Speichern der aktuellen eDOBS Objektstruktur verwenden, das in Abschnitt 18.2.1 beschrieben wurde.

Zusätzlich wurde im Flipbook Plugin ein spezielles Kommando implementiert, was dem Entwickler das explizite Speichern des CoObRA Protokolls in vielen Fällen abnehmen kann. Analysiert der Benutzer gerade eine Objektstruktur im eDOBS und befindet sich ein Objekt vom Typ `Repository` in der Objektstruktur, kann der Benutzer dieses *Back in Time* Kommando ausführen. Dann wird automatisch auf dem Repository Objekt im eDOBS eine Methode aufgerufen, die das aktuelle Protokoll in eine Datei speichert und diese Datei mithilfe des Flipbook Plugins lädt. Die Objekte im Protokoll deren „Originale“ zuvor im eDOBS sichtbar waren, werden an selber Stelle und mit selben Namen im Flipbook Diagramm angezeigt. Damit unterscheidet sich das Flipbook Diagramm mit Ausnahme der Hintergrundfarbe und der fehlenden Marker für lokale Variablen nicht vom Originaldiagramm.

Ein häufiger Startpunkt für Debugging-Aktivitäten ist im Fujaba Process ein fehlschlagender Test. Daher wird im Rahmen dieser Arbeit eine Integration des Flipbook Plugins mit dem in Teil II vorgestellten Testkonzept geschaffen. Schlägt ein JUnit Test fehl, kann, wie in Abschnitt 18.2.7 beschrieben, die Ausführung angehalten und die aktuelle Objektstruktur im eDOBS visualisiert werden. Zusätzlich ist es durch die Flipbook Integration nun auch möglich, ausgehend von der Situation beim Fehlschlagen des Tests, rückwärts in der Zeit zu navigieren. Dazu muss der JUnit Test lediglich eine Methode `getRepository()` implementieren. Diese Methode soll-

te das Repository, das den Test protokolliert hat, zurückliefern. Dieses Repository wird, wenn die Back-in-time Funktionalität benötigt wird, automatisch vom Flipbook Plugin verwendet um das Protokoll in eine Datei zu schreiben und diese im eDOBS zu öffnen.

Im Falle, dass ein JUnit Test aus einem Storyboard generiert wurde, wäre es wünschenswert, auch die `getRepository()` Methode automatisch zu generieren. Da aber bei unterschiedlichen Projekten das Repository auf unterschiedlichste Weise erreicht werden kann, muss der Entwickler im Modell spezifizieren, wie dies für das aktuelle Projekt geschehen soll. In unserem Ansatz markiert der Entwickler eine statische Methode einer beliebigen Klasse, die das entsprechende Repository zurückliefert, mit dem `RepositoryAccessor` Stereotyp. Diese Methode wird dann in der `getRepository()` Methode gerufen.

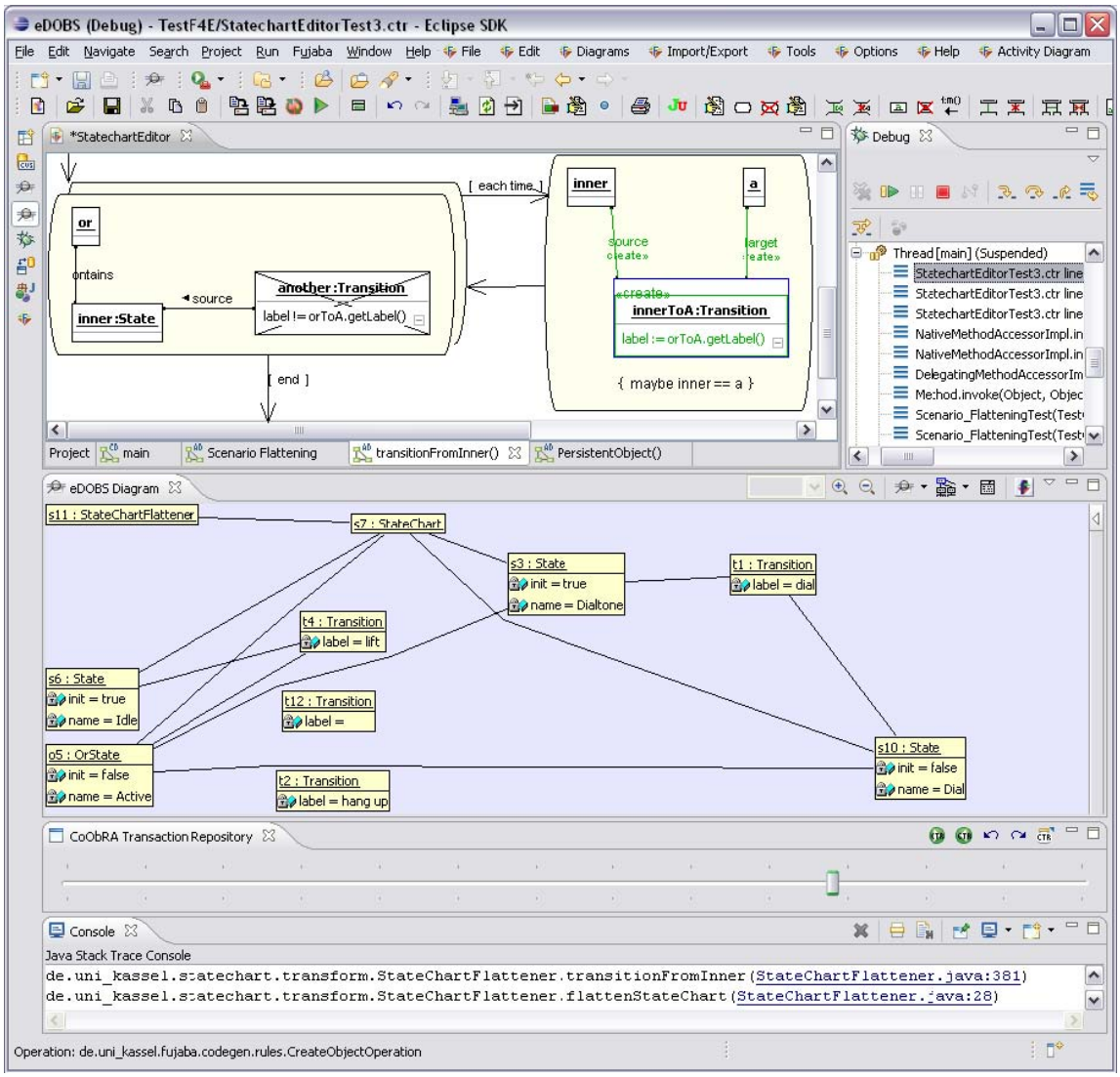


Abbildung 18.22.: Flipbook nach fehlschlagendem Test

Abbildung 18.22 zeigt das Flipbook bei der Fehleranalyse nach dem fehlgeschlagenen

Test aus Abbildung 18.17. Wie bereits erwähnt, gilt es für den Entwickler herauszufinden, warum zwar das Objekt `t2_2` angelegt wurde, nicht aber das Objekt `t2_1`. Eine spannende Frage ist also „Wann und an welcher Stelle wurde Objekt `t2_2` angelegt?“. In Abbildung 18.22 wurde um diese Frage zu beantworten das *Back in time* Kommando ausgeführt und mit Hilfe der Navigationskommandos des Flipbooks zur Objekterzeugung des Objekts `t2_2` (hier `t12`, vgl. Abbildung 18.17) gesprungen. Durch Folgen des Links des oberen Stacktrace-Eintrags im *Console View* gelangt man in die Methode `transitionFromInner()`, wie sie in Abbildung 18.22 im *Fujaba4Eclipse View* zu sehen ist. In dieser Methode wird für eine Transition, die vom Or-State ausgeht, für jeden inneren Zustand, der noch keine Transition mit gleichem Namen besitzt, eine Kopie dieser Transition angelegt, die vom betrachteten inneren Zustand ausgeht.

Navigiert man zurück in der Zeit, beobachtet man, dass vor der Objekterzeugung bereits die ursprüngliche Transition `t2` gelöscht wird. Nach der Objekterzeugung und dem Setzen der Kanten und Attribute von `t2_2` wird die Methode verlassen und der Or-State gelöscht. An dieser Stelle müsste aber auch das `t2_1` Objekt erzeugt werden. Durch das Flipbook Plugin konnte also genau die Stelle im Modell gefunden werden, wo der Fehler auftritt und auch die genau Objektstruktur, die zum Fehler führt. Nun muss der Entwickler nur herausfinden, wieso es an genau dieser Stelle mit genau dieser Objektsituation zum Fehler kommt.

In manchen Fällen ist aber selbst diese Überlegung noch schwierig. Das liegt daran, dass aus Skalierungsgründen im CoObRA Protokoll nur die Änderungen in der Objektstruktur gespeichert sind, nicht aber der komplette Verlauf. Es ist ohne weiteres also zum Beispiel nicht ersichtlich, welche Überprüfungen erfolgreich waren oder welche fehlschlugen und somit unter Umständen Änderungen an der Objektstruktur verhinderten. Es wäre also wünschenswert an der betrachteten Stelle zum betrachteten Zeitpunkt wieder mit dem Debugger aufzusetzen und die Ausführung schrittweise zu analysieren.

Die erste Idee hierzu könnte sein, einfach in der betrachteten Methode einen Breakpoint zu setzen und den Test erneut laufen zu lassen. Wird diese Methode aber, wie im betrachteten Test, mehrmals aufgerufen, kann es schwer und aufwendig sein den Zeitpunkt zu finden, an dem der Fehler auftritt. Des Weiteren könnte man diesem Breakpoint eine Einschränkung auf das aktuell geänderte Objekt hinzufügen. Auch das kann aber noch nicht ausreichend sein, um zum gewünschten Zeitpunkt anzuhalten. So könnte zum Beispiel ein Link zu einem Objekt 1000-mal gelöscht und wieder gezogen werden und beim 1001.ten Mal aufgrund eines geänderten Parameters ein Fehler auftreten. Daher wurde in dieser Arbeit ein anderer Ansatz gewählt: Das Flipbook speichert intern die wievielte Änderung vom Anfang der Datei gerade ausgeführt wurde. Es wird dann ein Breakpoint gesetzt, der die Einschränkung hat, erst dann anzuhalten, wenn die Änderungsanzahl zur Ausführungszeit der entspricht, die die gewählte Position im Flipbook repräsentiert. Auf diese Weise wird bei deterministisch ablaufenden Tests genau an der gewünschten Position zum gewünschten Zeitpunkt angehalten. Der Entwickler kann dann dort erneut seine Debugaktivität aufnehmen.

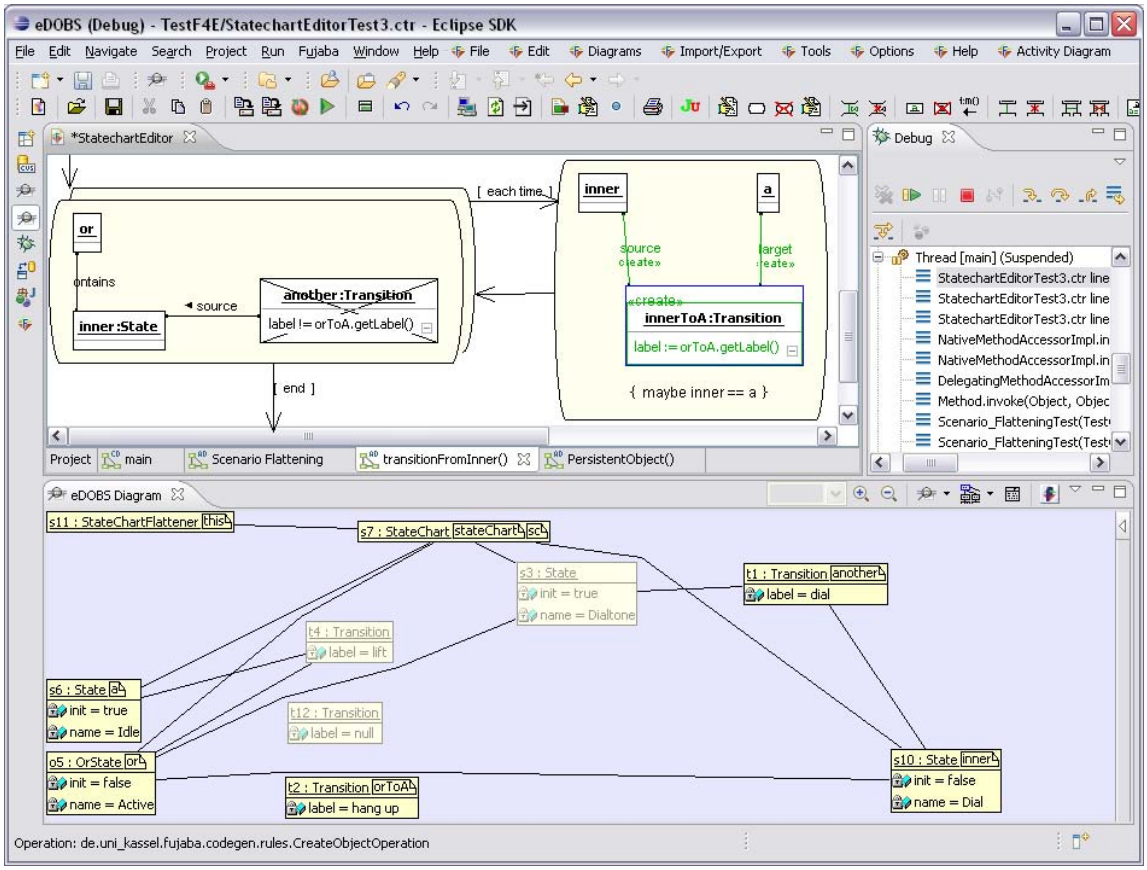


Abbildung 18.23.: Debuggingsession nach Wiederaufsetzen aus dem Flipbook

Bei Tests, die Nichtdeterminismus in der Form enthalten, dass sie Aktionen in wechselnder Reihenfolge abarbeiten, wie zum Beispiel Tests mit nebenläufigem Verhalten, trägt dieser Ansatz nicht. In diesen Fällen muss auf die oben angesprochenen aufwendigeren Debuggingtechniken zurückgegriffen werden. Allerdings sind solche Tests bei automatischen Unit Tests eher selten. Unseren Erfahrungen nach weisen die meisten Unit Tests, insbesondere solche, die aus Storyboards generiert wurden, den geforderten Determinismus auf.

Abbildung 18.23 zeigt die Situation des Flipbooks aus Abbildung 18.22 nach dem Wiederaufsetzen des Debuggers. Es wurde an genau der Stelle angehalten, an der das Objekt `t2_2` erzeugt wird. Durch schrittweises Ausführen an dieser Stelle kann schließlich leicht der eigentliche Fehler gefunden werden. Bei der Attributbedingung des negativen Objekts `another` liegt eine doppelte Negation vor. Hier darf keine Transition existieren, die ein Label besitzt, was sich vom Originallabel unterscheidet. Richtig wäre natürlich, dass keine Transition existieren darf, die ein identisches Label aufweist. Also müsste die Attributbedingung statt `label != orToA.getLabel()` richtig `label == orToA.getLabel()` lauten. An dieser Stelle wurden also innere Zustände ohne ausgehende Transitionen richtig und Zustände mit ausgehenden Transitionen falsch behandelt.

Durch das Zusammenspiel der Testgenerierung und der in diesem Teil vorgestellten

Debuggingfunktionen konnte im vorgestellten Beispiel ein Fehler im Implementierungsmodell auf einfache Weise gefunden werden. Der komplette Debuggingprozess erfolgte dabei auf Modellebene.

19. Einsatz

Im Folgenden wird auf den Einsatz der hier vorgestellten modellbasierten Debuggingtechniken eingegangen und über unsere Erfahrungen hierbei berichtet. Die Techniken wurden insbesondere bei der Entwicklung von Fujaba selbst, in einem Forschungsprojekt mit der Industrie und in der Lehre eingesetzt.

19.1. Fujaba

Bei der Entwicklung von Fujaba werden die hier vorgestellten Debugginghilfen bereits intensiv eingesetzt. Dies bietet sich an, da Teile von Fujaba, wie etwa die in Abschnitt 5.7 vorgestellte Codegenerierung CodeGen2, komplett in Fujaba modelliert sind. Das Debugging von CodeGen2 musste früher immer mühsam auf Quelltextebene erfolgen. Der hier vorgestellte Design Level Debugging Mechanismus ermöglicht es, endlich auch das Debugging auf Modellebene durchzuführen. Die Vorstellung dieser neuen Möglichkeit in [Gei08] auf den Fujaba Days 2008, der Konferenz der Fujaba Entwickler und Nutzer, führte zu enorm positiver Resonanz. Viele Fujaba Nutzer reagierten sinngemäß mit den Worten „Sowas wollte ich schon immer haben um meine Storydiagramme zu debuggen.“ Nach dieser Konferenz erhielt ich viele Anfragen zur konkreten Benutzung, Anmerkungen und Anregungen die auf einen verbreiteten Einsatz dieser Funktionalität schließen lassen.

Außerdem wird der eDOBS viel in der Fujaba Entwicklung eingesetzt. Im eDOBS werden beispielsweise Fujaba interne Objektstrukturen analysiert und anhand der eDOBS Diagramme neuen Entwicklern erklärt. Es kann auch vereinzelnd vorkommen, dass die Objektstruktur eines Fujaba Projekts durch einen Fehler in der Applikation so beschädigt wird, dass diese nicht mehr über die graphische Benutzeroberfläche repariert werden kann. Hier kann man entweder versuchen, den Fehler mit einem Texteditor in der Projektdatei zu finden, was sehr mühsam ist, oder den entsprechenden Modellausschnitt im internen eDOBS anzeigen lassen und dort reparieren. Zweiteres wird im Falle von defekten Projekten meist von erfahrenen Fujaba Entwicklern durchgeführt um diese Projektdateien zu retten. Auch zur Analyse von Eclipse Plugins, die nicht direkt zum Fujaba CASE Tool gehören wird der eDOBS gerne eingesetzt. Hier hat sich auch die Unterstützung von EMF Modellen als Vorteil herausgestellt, da viele Eclipse Plugins mittlerweile EMF zur Modellierung ihrer internen Datenstrukturen nutzen.

19.2. OBA

Im Forschungsprojekt Optimierung von Fahrzeug-Bordnetz-Architekturen (OBA), welches in Kapitel 8 bereits diskutiert wurde, erwies sich insbesondere der eDOBS als wertvolles Hilfsmittel. Er ermöglichte es, auch im frühen Stadium des Projekts, in dem noch keine graphische Benutzeroberfläche existierte, Kabelbäume übersichtlich darstellen zu können. Dadurch konnten die Domänenexperten des Elektrotechnikfachgebiets früh beim Testen von Funktionalität miteinbezogen werden. Die Domänenexperten benutzten den eDOBS also zur Visualisierung von Ergebniskabelbäumen um diese zu analysieren, mit dem Entwicklerteam zu diskutieren und gegebenenfalls auf fehlerhafte Resultate hinzuweisen.

Teilweise existierten in diesem Projekt Storyboards in Papierform oder als abfotografierte Tafel-/Whiteboardbilder. Die Domänenexperten nutzten dann den eDOBS zum manuellen Testen: Der im eDOBS visualisierte Ergebniskabelbaum wurde von ihnen mit dem Ergebnis des Storyboards verglichen.

Außerdem wurde der eDOBS in diesem Projekt zum manuellen Durchspielen von Szenarien genutzt. Eine Startsituation wurde erstellt und im eDOBS angezeigt. Dann wurden die Änderungen, die in diesem Szenario geschehen sollten diskutiert und im eDOBS nachvollzogen. Diese Diskussionen wurden üblicherweise im Team mit den Domänenexperten und den Entwicklern durchgeführt.

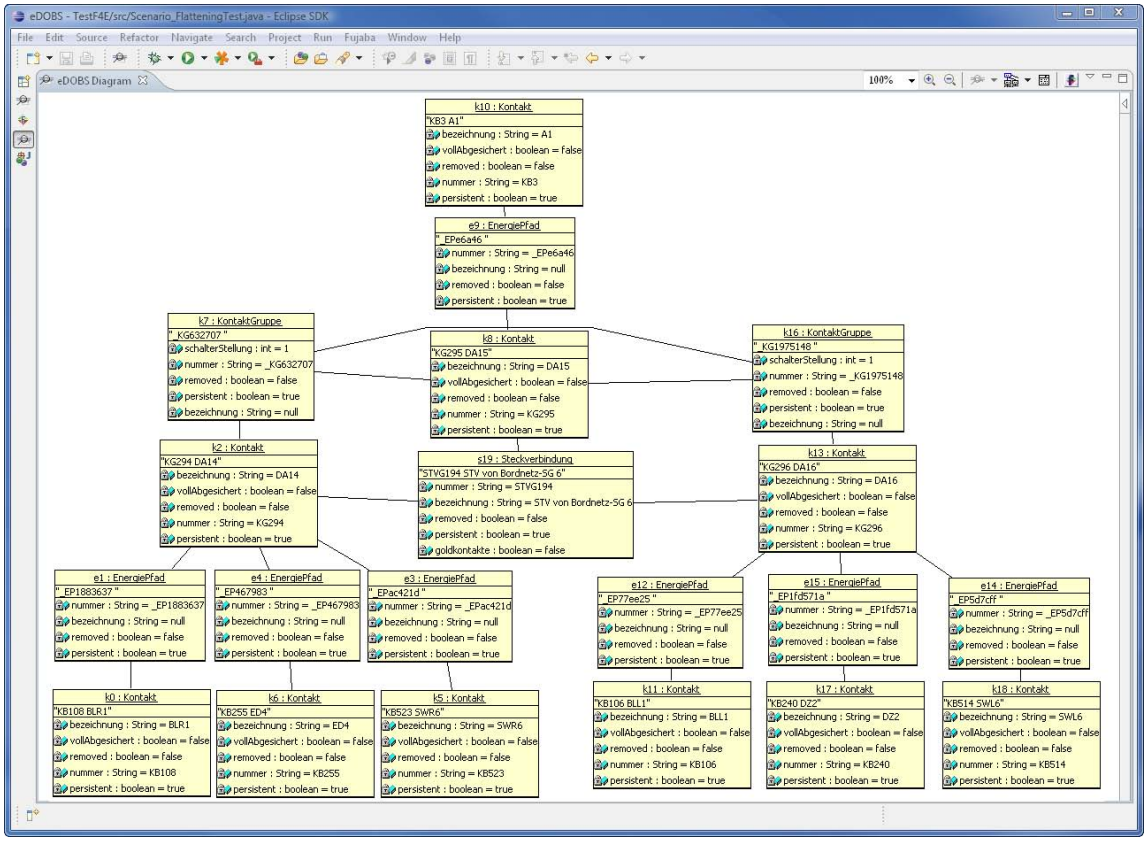


Abbildung 19.1.: Blinkerkabelbaum im eDOBS

Von den Entwicklern wurde der eDOBS während der Implementierung auch intensiv eingesetzt, um während des Debuggens den aktuellen Kabelbaum zu analysieren. Außerdem wurde der eDOBS hier viel benutzt, um Funktionalität, für die noch keine Kommandos in der graphischen Oberfläche existierten, auf bestehenden Kabelbäumen aufzurufen. Abbildung 19.1 zeigt den eDOBS bei der Darstellung der Objektstruktur eines Kabelbaums. Hierbei handelt es sich um einen Blinkerstrang, der im OBA Werkzeug optimiert wird.

Mit dem Ende des OBA Projekts wurde Anfang 2009 das entwickelte Werkzeug zur weiteren Pflege und Wartung an die aquintos GmbH übertragen. Während der Portierung des Werkzeugs auf die aquintos Plattform wurden die aquintos Mitarbeiter auch auf den eDOBS aufmerksam und planen, diesen in ihren weiteren Entwicklungen zu verwenden. Die aquintos Plattform unterstützt, ähnlich wie Fujaba, die Ausführung von Regeln auf den aktuellen Datenstrukturen. Diese Regelanwendungen lassen sich nicht direkt debuggen, jedoch existiert ein Post-Mortem Debugger. Dieser protokolliert die Regelanwendungen und kann dieses Protokoll später in einer baumbasierten Ansicht darstellen. Hier besteht auch das in Abschnitt 18.2 beschriebene Übersichtlichkeitsproblem. Daher ist geplant in Zusammenarbeit mit aquintos ein Projekt zu starten, was diese Protokolle im eDOBS visualisieren kann.

Zusammenfassend kann man sagen, dass der eDOBS im OBA Projekt dazu beitrug, dass die Zusammenarbeit von Domänenexperten und Softwareentwicklern sehr gut funktionierte, die Domänenexperten früh in den Testprozess integriert werden konnten und das mit der aquintos GmbH sogar ein beteiligtes Unternehmen großes Interesse am Einsatz und auch an der Anpassung des eDOBS für die eigenen Anwendungen gezeigt hat.

19.3. Lehre

In den in Kapitel 13 beschriebenen und in Anhang A.1 aufgeführten Studentenprojekten wurden auch die hier vorgestellten Debuggingtechniken eingesetzt. Diese kamen besonders durch die in Abschnitt 18.2.7 erläuterte Testintegration nach fehlgeschlagenen Tests, welche aus Storyboards generiert wurden, zum Einsatz. Hierbei wurde insbesondere der eDOBS zum Vergleich der erwarteten Objektstruktur, wie sie im Storyboard beschrieben ist, mit der tatsächlichen Objektstruktur, nach Abarbeitung des Tests, verwendet. Unsere Beobachtungen hierbei waren, dass es den Studenten durch das Hilfsmittel eDOBS sehr viel leichter fiel, Fehler in der Ergebnissituation zu identifizieren.

In den letzten Durchführungen der Studentenprojekte konnten auch schon erste Erfahrungen mit dem Design Level Debugging gemacht werden. Dieses stellte unserer Beobachtung nach eine enorme Erleichterung für die Studenten dar, da das Debugging in vorangegangenen Projekten immer auf Quelltextebene erfolgen musste, was die Studenten vor große Verständnishürden stellte. Das Flipbook Plugin konnte leider im Rahmen dieser Arbeit noch nicht in diesen Studentenprojekten eingesetzt werden. Dieses ist jedoch für das nächste Semester geplant.

Zusätzlich zu den Studentenprojekten wurden die beschriebenen Werkzeuge auch in Kursen zur Einführung in die objektorientierte Programmierung an der Universität Kassel und an der Gaussschule Braunschweig eingesetzt. Hierbei wurde der eDOBS als „Spielwiese“ für Objektstrukturen genutzt. So wurden beispielsweise die ersten Objektstrukturen im eDOBS angelegt, dort Szenarien durchgespielt und die Resultate von Methodenaufrufen erforscht. Mit einem solchen Vorgehen können die Studenten mit Objekten experimentieren ohne das Anlegen und Verändern der Objekte von Hand in einer Testapplikation implementieren zu müssen. Das erspart auch dem Lehrkörper das frühzeitige Erklären des in Java sehr sperrigen Konstrukts der main-Methode (`public static void main(Sring[] args)`).

Zum Erklären von Storydiagrammen hat sich das schrittweise Ausführen mit dem Design Level Debugging als sehr hilfreich erwiesen. Hiermit konnten Methoden einfach von den Schülern/Studenten durchgespielt und verstanden werden. Um die genaue Ausführung eines Storypatterns zu erklären, war vorher immer eine manuelle Dokumentation mittels Powerpoint durch den Dozenten nötig (siehe Abschnitt 18.2.6). Diese Arbeit kann nun meist entfallen.

Wurden typische Verständnisschwierigkeiten seitens der Schüler/Studenten von uns beobachtet, bemühten wir uns, die Vorlesung, aber auch die Werkzeuge, wenn möglich, so anzupassen, dass diese Schwierigkeiten reduziert werden konnten. So ist beispielsweise durch die von uns beobachtete Verständnisschwierigkeit des Perspektivwechsels von allwissender Draufsicht auf die Objektstruktur hin zur beschränkten Sicht eines einzelnen Objekts, von uns der Zetteltest entwickelt worden. Dieser wurde dann auch, wie in Abschnitt 18.2.6 beschrieben, im eDOBS umgesetzt. Wir sind der Meinung, dass diese Reduktion der Verständnishürden bei Teilnehmern unserer Anfängerveranstaltungen auch Nicht-Informatikern, wie beispielsweise Domänenexperten, das Verständnis, beispielsweise der eDOBS Diagramme, vereinfachen kann.

Nach unseren Erfahrungen sind solche Vereinfachungen aber auch für IT-Experten oft eine große Hilfe. Denn auch für IT-Experten stellen diese Vereinfachungen eine Reduktion der Komplexität ihrer Aufgaben dar, die es ihnen erlaubt komplexe Probleme schneller und sicherer zu bearbeiten.

20. Verwandte Arbeiten

Viele verwandte Arbeiten zum Design Level Debugging stammen aus dem Bereich der Software Visualisierung, siehe [Die02]. Diese Community beschäftigt sich mit der graphischen Darstellung von Programmabläufen.

Ein Werkzeug zur Software Visualisierung stellt beispielsweise das JACOT Tool [LRRM03] der Monash University dar. Hier werden UML Sequenzdiagramme benutzt um zur Debugzeit die Kommunikation zwischen den einzelnen Objekten und Threads zu visualisieren. Zusätzlich werden spezielle Statecharts, sogenannte Thread State Diagramme, zur Visualisierung der Zustände, in denen sich die einzelnen Threads befinden, verwendet. Dieses Werkzeug dient in erster Linie der Analyse und dem Verständnis von nebenläufiger Programmausführung. Daher werden auch spezielle Funktionen zur Analyse nebenläufiger Programme zur Verfügung gestellt, wie beispielsweise eine spezielle Visualisierung von Deadlocks.

Ähnliche Funktionalität wie JACOT bietet das JaVis Tool [Meh01] der Universität Paderborn. Hier werden ebenfalls über das Debuginterface Traceinformationen gesammelt und als Sequenzdiagramm dargestellt. Zur Darstellung wird hier auf das UML CASE Tool Together zurückgegriffen. JaVis unterstützt zusätzlich noch verschiedene Filter um nur die Objekte anzuzeigen, die für die momentane Deadlockanalyse von Interesse sind. Die beiden genannten Werkzeuge visualisieren die Ausführung von normalen Java Programmen. Im Gegensatz dazu zeigt der hier vorgestellte Design Level Debugging Ansatz für aus Modellen generierten Java Quelltext wieder die eigentlichen Modelle an. Mein Ansatz ist somit auch nicht auf bestimmte Probleme wie zum Beispiel Nebenläufigkeit oder bestimmte Diagrammarten eingeschränkt. Design Level Debugging lässt sich generell zur Analyse von aus Modellen generiertem Quelltext einsetzen.

Die beiden erwähnten Tools konzentrierten sich lediglich auf den Nachrichtenaustausch zur Programmlaufzeit. Es gibt allerdings auch Ansätze, die versuchen das komplette Verhalten während des Debuggens graphisch darzustellen. Ein solcher Ansatz liegt dem visuellen Debugger Jeliot 3 [BMM05] zugrunde. Hier wird durch unterschiedliche Diagramme und Animationen während einer Debuggingsession jedes einzelne Java Statement visualisiert. Für Methodenaufrufe gibt es eine *Method Area*, in der die Klassen mit ihren Methoden dargestellt werden, Konstanten werden in der *Constant Area* dargestellt, usw. Dieses Werkzeug ist vor allem auf Anfänger in der Java Programmierung ausgelegt. Diese sollen mit dem Debugger ein besseres Verständnis bekommen, wie Java Programme ausgeführt werden. Daher und zum Verständnis des Kontrollflusses, der in keiner graphischen Form vorliegt, wird in Jeliot auch immer der Java Quelltext während des Debuggens mit angezeigt. Wie schon bei JACOT und JaVis wird aber auch hier normaler Java Quelltext verarbeitet. Jeliot verfügt weiterhin noch über eine *Instance Area*. Hier werden ähnlich wie

im eDOBS Objekte dargestellt, die zur Laufzeit erzeugt wurden. Allerdings werden hier Referenzen zwischen Objekten nicht angezeigt und es ist auch nicht möglich, auf diesen Objekten zum Beispiel Methoden aufzurufen oder neue Objekte anzulegen. Die *Instance Area* dient lediglich zur Visualisierung.

Es gibt weitere Tools zur Programmablaufvisualisierung, die auch Objektdiagramme benutzen. JavaVis [OS02] beispielsweise setzt Sequenzdiagramme zur Visualisierung des Programmablaufs ein und Objektdiagramme zur Repräsentation der Datenstrukturen. JavaVis zeigt zu jedem Stackframe, also zu jedem aktiven Methodenaufruf, ein Objektdiagramm an, das die aktuelle Variablenbelegung wiedergibt. Es werden also lediglich die Daten auf dem Stack visualisiert und nicht wie im eDOBS auch die Objektstrukturen im Heap. Ähnlich wie JavaVis arbeitet Jive [CJ07]. Auch Jive benutzt Sequenzdiagramme für den Ablauf und Objektdiagramme zur Visualisierung des Stacks. Allerdings ist Jive genau wie eDOBS als Eclipse Plugin erhältlich und in den Eclipse Debugger integriert.

Die SMAP Technologie des JST-045, die in dieser Arbeit für das Design Level Debugging verwendet wird, ist seit Java 1.4 Teil der Java VM und wird durch eine Reihe von Debuggern, darunter auch dem Debugger des Eclipse JDTs, unterstützt. Diese Technologie wird bereits eingesetzt um textuelle Sprachen auf den daraus generierten Java Quelltext abzubilden. Oft geschieht das beispielsweise im Kontext der Websprachen, wie Java Server Pages (JSP). Um Modelle und Quellcode aufeinander abzubilden wurde die SMAP Technologie hingegen, meines Wissens nach, noch nicht eingesetzt.

Einige moderne Debugger, wie beispielsweise jGRASP [IHJB07], bieten die Möglichkeit den aktuellen Inhalt der lokalen Variablen während des Debuggens als Diagramm darzustellen. In jGRASP werden in diesen Diagrammen Objekte und deren Attributbelegungen angezeigt. Es können sogar zur Laufzeit Methoden aufgerufen und Attribute verändert werden. Allerdings zeigt jGRASP keine Links zwischen Objekten an.

Ähnliche Funktionalität bietet die in der Lehre weit verbreitete Entwicklungsumgebung BlueJ von Michael Kölling, [KQPR03]. BlueJ ist für Programmieranfänger gedacht und bietet daher eine eingeschränkte, auf die wichtigsten Funktionen reduzierte Benutzeroberfläche. BlueJ stellt dem Benutzer einen einfachen Klassendiagrammeditor, einen Editor für Java Quellcode und einen Objektbrowser zu Verfügung. Im Objektbrowser, in BlueJ *object bench* genannt, können interaktiv Objekte erzeugt und Methoden aufgerufen werden. BlueJ unterstützt aber ähnlich wie jGRASP keine Beziehungen, weder im Klassendiagramm noch im Objektbrowser. Eine Portierung des BlueJ Objektbrowsers nach Eclipse stellt e-BOB [Mor07] dar. Jedoch gelten für diese Portierung die selben Einschränkungen wie für den BlueJ Objektbrowser.

Unserer Meinung und unseren Erfahrungen nach ist die Unterstützung von Beziehungen zwischen Objekten aber sehr wichtig für das Verständnis von objektorientierten Programmen, da in vielen dieser Programme der Zustand des Programms zu einem großen Teil durch die aktuelle Objektstruktur im Speicher beschrieben ist. Zu dieser Objektstruktur gehören üblicherweise auch Links zwischen den einzelnen Objekten. Daher unterstützt das Fujaba CASE Tool Assoziationen im Klassendiagramm und

der eDOBS deren Instanzen auf Objektdiagrammebene. In mehreren Fallstudien an drei deutschen Gymnasien und den Universitäten Kassel und Braunschweig haben wir die Erfahrung gemacht, das der Einsatz von Fujaba/eDOBS das Verständnis von objektorientierten Prinzipien für unsere Schüler/Studenten vereinfacht hat, siehe [DGZ03b, DGZ02, DGZ05b, DGZ05c]. Ulrich Norbisrath hat ebenfalls in mehreren Vorlesungen über Software Engineering an der Universität Tartu in Estland das Objektspiel eingesetzt, daraus Objektdiagramme und Klassendiagramme in Fujaba abgeleitet und diese mit Hilfe des eDOBS analysiert [Nor10]. Er berichtet von beeindruckenden Resultaten und sehr positiven Rückmeldungen der Studenten.

Es gibt allerdings auch andere Debugger, die einen Objektbrowser bereitstellen, der Beziehungen zwischen Objekten darstellen kann, wie beispielsweise der Data Display Debugger (DDD) von Andreas Zellen, [ZL96]. DDD ist ein Frontend für verschiedene Debugger wie etwa den GDB. Er bietet die Möglichkeit verzeigerte Objektstrukturen als Graph in einer Objektdiagramm-ähnlichen Ansicht darzustellen. Da diese Diagramme aber Teil eines generellen Debugger-Frontends sind, werden interne Datenstrukturen wie zum Beispiel die Implementierungen von Listen oder Mengen auch dargestellt. In eDOBS werden solche Strukturen als mehrere Links zwischen den einzelnen Objekten visualisiert. In DDD würde aber beispielsweise die Darstellung eines durch eine Baumstruktur implementierten Sets eine Menge Objekte vom Typ `TreeNode` mit `leftChild` und `rightChild` Links anzeigen, die nicht Teil des Applikationsmodells sind. Die Darstellung eines durch ein `TreeSet` in Java implementierten Links im eDOBS würde ohne Abstraktion in mehr als doppelt so vielen Objekten und der vierfachen Menge an Links resultieren.

Mit dem SOL UML Debugger [Lab08] wird seit einiger Zeit ein kommerzieller Objektbrowser als Plugin für das CASE Tool Poseidon angeboten. Dieser Objektbrowser ist aber mehr ein Objektdiagrammeditor mit dem Objekte angelegt und Methoden aufgerufen werden, er bietet keine Möglichkeit die Objektstruktur während einer Debugsession zu visualisieren. Allerdings ist es im SOL UML Debugger zusätzlich möglich sogenannte Snapshots, also Momentaufnahmen der Objektstruktur, zu speichern und diese später, beispielsweise als Testfall, wiederherzustellen.

Einen Objektbrowser, mit dem es wie mit eDOBS möglich ist Eclipse interne Objektstrukturen darzustellen, stellt der von Erich Gamma und Kent Beck entwickelte JavaSpider [GB03] bereit. Im JavaSpider kann man ausgehend von einem Startobjekt die interne Objektstruktur von Eclipse durch das Aufrufen ausgewählter Methoden und das Auslesen gewählter Attribute erkunden. Mit dem JavaSpider ist es jedoch nicht möglich diese Objektstruktur zu verändern. Weiterhin lässt er sich nur auf internen Datenstrukturen, also auf Daten in der Eclipse VM einsetzen und nicht etwa zur Visualisierung von Objektstrukturen im Debugger. Zusätzlich zeigt der JavaSpider genau wie DDD interne Strukturen von Containerklassen mit an und bietet keine Möglichkeit hiervon zu abstrahieren.

Die Idee der Fujaba Maschine wurde von der Idee des animierten UMLs von Steimann et al [STSN02] inspiriert. Hier wird vorgeschlagen zum besseren Verständnis der objektorientierten Mechanismen animierte UML-Diagramme, zum Beispiel in Lehrfilmen zu verwenden. Als mögliche Animation wird in [STSN02] beispielsweise

das Stanzen von Objekten mit dem entsprechenden Klassenstempel vorgeschlagen. Im Gegensatz zu [STSN02] enthält unserer Ansatz aber keine aktiven Elemente des Klassendiagramms. Weiterhin wurde auf die vorgeschlagene dreidimensionale Darstellung verzichtet. Allerdings stellt der eDOBS mit den hier beschriebenen Erweiterungen, wie Klebezettel und Objektnebel, ein Tool dar, das die Animationen der Fujaba Maschine zumindest teilweise für beliebige Objektstrukturen erlebbar macht.

Einige der hier vorgestellten Debugger erstellen Protokolle der Programmausführung und visualisieren diese dann. Durch diese Protokolle ist es, ähnlich wie beim in dieser Arbeit vorgestellten Flipbook, möglich vorwärts und rückwärts durch diese Protokolle zu navigieren. So wird diese Funktionalität beispielsweise beim Jive Debugger angeboten. In Jive kann man vor und zurück durch die Sequenzdiagramme navigieren und dabei die Datenstruktur des Stacks im Objektdiagramm verfolgen. Zusätzlich ist es möglich, temporale Anfragen zu stellen, beispielsweise, wann ein bestimmtes Attribut geändert wurde. Diese Funktionalität ähnelt der des in dieser Arbeit vorgestellten Flipbooks. Es ist jedoch nicht möglich, bei der Navigation rückwärts in der Zeit an einem bestimmten Punkt erneut mit dem Debuggen aufzusetzen.

Es gibt auch konventionelle Java Debugger, die das debuggen rückwärts in der Zeit erlauben, sogenannte Omniscient Debugger, wie TOD [PTP07] oder ODB [Lew03]. Diese Debugger protokollieren alle Informationen, die während eines Debuggerlaufs anfallen. Hierzu gehören die Änderungen der Objektstrukturen im Heap aber auch die Belegung von lokalen Variablen sowie der aktuelle Programmzeiger. Omniscient Debugger bieten dann Kommandos an, um in diesen Protokollen zu navigieren, wie zum Beispiel einfaches rückwärts steppen oder auch komplizierte Anfragen, wie, wann hatte diese lokale Variable den Wert X, oder, wann wurde dieser Zweig einer Abfrage durchlaufen. Die hierbei verwendeten Protokolle werden üblicherweise sehr schnell sehr groß, weshalb sich diese Debugger nicht für große Applikationen und langlaufende Programmabläufe eignen. Beim Flipbook wurde daher die Entscheidung getroffen, nur die Änderungen der Objektstruktur zu protokollieren, da so die Protokolle auch bei langlaufenden Programmen handhabbar bleiben. Sollten die Traces im Flipbook trotzdem zu groß werden, gibt es in CoObRA die Möglichkeit durch Anmelden von Filtern, nur die Changes für relevante Objekte ins Protokoll aufzunehmen. Somit eignet sich der Flipbook Ansatz auch für große Applikationen. Die Nachteile, die der Flipbook Ansatz gegenüber den klassischen Omniscient Debugger hat, wie zum Beispiel, dass die Stackinformation nicht zur Verfügung steht und somit keine Aussagen über Variablenbelegung gemacht werden kann, werden zumindest teilweise durch die Möglichkeit an einer beliebigen Stelle im Trace wieder mit dem Debuggen aufsetzen zu können, vermindert.

Eine spezielle Form des Omnicient Debuggers stellt das Whyline Werkzeug [KM08] dar. In Whyline werden während des Debuggens auch Tracing Informationen mitgeschrieben. Whyline konzentriert sich nun aber auf die Möglichkeiten, in diesen Traces suchen zu können. In [KM08] stellen die Autoren fest, dass Entwickler während des Debuggingprozesses gerne Fragen wie „Warum passierte dies?“ oder „Warum passierte jenes nicht?“ stellen würden. Daher bietet Whyline sogenannte Why und Why-not Anfragen an. Es ist zum Beispiel möglich zu Fragen, warum ein Attributwert eines Objekts den Wert X angenommen hat, oder warum eine bestimmte Methode nicht

aufgerufen wurde. Solche Anfrage werden in Whyline direkt auf der graphischen Oberfläche oder der textuellen Ausgabe einer Applikation gestellt. Hierzu wird die Applikation entsprechend instrumentiert, so dass zu jedem Element der UI oder der Ausgabe ein Menu abrufbar ist, was entsprechende Fragen zu diesem Element anbietet. Durch Wählen einer Frage landet man dann an der entsprechenden Stelle im Quellcode (und in der Zeit), an der zum Beispiel ein bestimmtes Attribut gesetzt wurde. Von da aus kann man durch weiteres Fragen weiter durch die Zeit und durch den Quellcode navigieren. Alle Fragen, die man in Whyline bezogen auf die Objektstruktur stellen kann, sind so oder ähnlich auch im Flipbook möglich. Fragen über Methodenaufrufe oder lokale Variablen hingegen können im Flipbook nicht gestellt werden, da hierzu keine Protokollinformation erstellt wird. Eine „Fragerunde“ in Whyline muss aber zwingend auf den Ausgaben oder der UI eines Programms starten und auch nur auf dieser kann der aktuelle Zustand graphisch repräsentiert werden. Im Flipbook hingegen ist dies für beliebige Objektstrukturen möglich, Startpunkt muss hier also nicht zwingend ein UI Element sein und das Flipbook kann beispielsweise auch für Applikationen oder Tests benutzt werden, die keine Ausgaben erzeugen.

Alle hier vorgestellten Omnicient Debugger arbeiten auf Java Quelltext. Die Flipbook Komponente des eDOBS bietet aber ein Debuggen direkt auf Modellebene an. Der generierte Java Quelltext muss also bei dem hier vorgestellten Ansatz nicht mehr herangezogen werden.

21. Fazit

In diesem Teil wird ein Ansatz vorgestellt, der Debugging auf Modellebene erlaubt. Mit Hilfe des Design Level Debugging können Haltepunkte im Modell gesetzt werden und Modelltransformationen schrittweise ausgeführt werden. Der eDOBS erlaubt es dann, während des Debuggens, Ausschnitte der aktuellen Datenstruktur im Speicher als Objektdiagramme übersichtlich darzustellen. Das Flipbook Plugin ermöglicht das Navigieren in der Änderungshistorie einer Objektstruktur auch rückwärts in der Zeit. Alle Komponenten wurden eng in Eclipse, Fujaba4Eclipse und das in Teil II vorgestellte Testkonzept integriert. Somit lassen sich beispielsweise Fehler, die in fehlschlagenden, aus Storyboards generierten Tests resultieren, leicht finden und beheben. Der vorgestellte Ansatz wurde in einem großen Industrieprojekt, in der Entwicklung von Fujaba selbst und in der Lehre eingesetzt. Die Erfahrungen dieser Einsätze waren durchweg positiv. Die präsentierten Ansätze konnten die Fehlersuche in der Praxis tatsächlich erleichtern.

Der präsentierte Ansatz ist eng mit Eclipse und Fujaba4Eclipse verzahnt. Die präsentierten Ideen lassen sich aber auch auf andere Plattformen übertragen. Dennoch sind dort aufgrund unterschiedlicher technischer Voraussetzungen teilweise größere Hürden in der Umsetzung zu erwarten. Das Konzept zur Erzeugung einer Modell-Code Abbildung beispielsweise lässt sich bei jeder Template-basierten Codegenerierung anwenden. Ein Konzept zum Debuggen von anderen Sprachen, wie es der JSR-045 beschreibt, existiert meines Wissens allerdings nur für die Java VM. Möchte man also das Design Level Debugging auch für andere Sprachen, wie zum Beispiel C++ anwenden, muss diese Funktionalität selbst geschrieben werden. Der eDOBS hingegen ließe sich auch mit anderen Sprachen nutzen. Hierfür muss lediglich eine Modellabstraktionsschicht geschrieben werden, die in der Lage ist, Informationen über die aktuelle Objektstruktur und deren zugrunde liegendes Modell auszulesen (siehe Kapitel 18.2.2). Neben dem Zugriff auf von Fujaba generierte Klassen wurden ja ,wie bereits erwähnt, schon Modellabstraktionsschichten für EMF- und für MOF-Modelle implementiert. Ähnliches gilt für die Flipbook Erweiterung. Diese kann weiterbenutzt werden, wenn die zu analysierende Applikation CoObRA-Protokolle erstellen kann. Hierfür muss gegebenenfalls CoObRA auf die Zielplattform portiert werden.

Für die Design Level Debugging Funktionalität wurde CodeGen2 um die Fähigkeit erweitert, SMAP Dateien zusätzlich zu den Java Dateien zu generieren, wie in Kapitel 18.1.1 erläutert. Diese Funktionalität wurde komplett in Fujaba spezifiziert und umfasst 11 Klassen. Der daraus generierte Code ist ungefähr 1.500 Codezeilen lang. Ungefähr die selbe Menge Quellcode war nötig um die Design Level Debugging Funktionalität auch in Eclipse anzubieten. Hierzu zählen beispielsweise der Quelltext für die Kommandos zum Setzen von Haltepunkten im Modell, sowie der Quelltext, der die Fujaba Nature definiert.

Der eDOBS teilt sich in sieben verschiedene Plugins auf. Neben dem eigentlichen Kern, gibt es zum Beispiel ein Plugin, das die Debugger-Integration realisiert, ein Plugin, das zusätzliche Layout-Mechanismen hinzufügt, sowie ein Plugin für die Fujaba-Integration. Insgesamt umfassen diese Plugins mehr als 20.000 Quelltextzeilen. Für die verwendete Modellabstraktionsschicht und die Implementierungen für EMF und MOF, an denen ich ebenfalls mitgewirkt habe, kommen nochmal ca. 20.000 Zeilen hinzu. Die Flipbook Erweiterung besteht aus zwei Eclipse Plugins, die zusammen in ca. 40 Klassen und Interfaces mit insgesamt 2.500 Zeilen Quellcode implementiert sind.

In diesem Teil wurden also Techniken und Methoden vorgestellt um Fehler in modellbasierten Prozessen zu entdecken und zu lokalisieren. Die Fehlersuche im Modell wird durch diese Techniken wesentlich verbessert und beschleunigt, da nicht mehr auf quelltextbasierte Methoden zurückgegriffen werden muss.

Teil IV.

Zusammenfassung

22. Zusammenfassung

Im Rahmen dieser Arbeit wird ein Konzept zur Fehlersuche in modellbasierter Softwareentwicklung vorgestellt. Hierzu wird zuerst ein komplett modellbasierter Prozess, der Fujaba Process, definiert. In diesem Prozess spielen Szenarien eine entscheidende Rolle. Sie werden zusammen mit dem Kunden oder den Domänenexperten in mehreren Schritten entwickelt und dienen als Diskussions- und Entscheidungsgrundlage. Zusätzlich wird aus den Szenarien die Struktur und das Verhalten systematisch abgeleitet. Der Prozess wird also von den Szenarien getrieben. Es wird gezeigt, dass dieses Vorgehen die Kommunikation zwischen Domänenexperten und Entwicklern verbessert und somit Missverständnisse und frühe Fehler reduzieren kann.

Diese Arbeit stellt weiterhin eine automatische Testgenerierung für die im Fujaba Process verwendeten Szenarien vor. Die generierten Tests überprüfen, ob die Implementierung das entsprechende Szenario korrekt abbildet. Hierdurch werden Szenarien automatisch überprüfbar. Szenarien sind also nicht mehr bloß Dokumentation, sondern tragen in Form von Tests einen Teil zur Applikation bei. Es konnte gezeigt werden, dass durch dieses Vorgehen Fehler in Szenarien früh entdeckt werden können und dass das Problem des „Auseinanderlaufens“ von Dokumentation in Form von Szenarien und der eigentlichen Implementierung verhindert werden kann.

Zusätzlich stellt diese Arbeit verschiedene Ansätze zur Fehlersuche im Fujaba Process mittels Debugging dar. Es wird gezeigt, wie die Ideen des klassischen Debuggens auf Quelltextebene (wie Haltepunkt setzen, schrittweises Ausführen des Quelltextes, Variablen inspizieren usw.) auf modellbasierte Ansätze übertragen werden können. Hierzu wurden unterschiedliche Techniken und Werkzeuge entwickelt. Anhand von Anwendungsbeispielen wird gezeigt, dass diese Debugging-Techniken und -Werkzeuge die Produktivität bei der Fehlersuche enorm erhöhen können. Zusätzlich erleichtern sie das Verstehen von unbekannten Programmen und Algorithmen, weshalb sich die Debugging-Werkzeuge auch speziell für den Einsatz in der Lehre eignen.

Insgesamt wurden Wege aufgezeigt, die das Auffinden von Fehlern und Fehlerursachen mithilfe von Tests und Debugging in modellbasierten Ansätzen erleichtern. So leistet diese Arbeit einen wertvollen Beitrag dazu, das Ziel der Softwaretechnik, Programme fehlerfreier zu machen, zu erreichen.

Sämtliche entwickelten Techniken wurden prototypisch in der Eclipse Entwicklungsumgebung implementiert. Es entstanden ein Codegenerator, ein Modellabstraktionsframework und zahlreiche Eclipse Plugins. Insgesamt wurden zur Realisierung dieser Arbeit über 100.000 Zeilen Quellcode verfasst, die teilweise ausschließlich von mir, teilweise im Team mit anderen wissenschaftlichen Mitarbeitern oder studentischen Hilfskräften geschrieben wurden. Viele der Forschungsergebnisse wurden auf

renommierten Konferenzen oder Workshops präsentiert. Insgesamt habe ich 32 Veröffentlichungen im Bereich Fujaba Process, modellbasiertes Testen und modellbasiertes Debuggen verfasst bzw. an diesen mitgeschrieben, siehe [DGZ02, DGZ03b, GZ02a, GZ03, DGZ03a, DGZ05b, DGZ05a, GZ06c, GZ06a, GGZ+05, SZG07, GZ06b, GBD07, GS07, GSR05, GSZ03, Gei08, DGMZ02, DGZ04, GZ05, GZ04, GZ02b, Gei02b, DGSZ04a, DGSZ04b, DGZ05c, DGSZ06, GSZ05a, GSZ05b, DGZ08, BGSZ08, WGM06].

23. Ausblick

In diesem Kapitel werden offene Forschungsfragen gezeigt, die sich an diese Arbeit anschließen und Grundlage für aufbauende Arbeiten sein können.

Im Fujaba Process gibt es einige Punkte, für die noch zusätzliche Werkzeugunterstützung umgesetzt werden kann. So ist beispielsweise das Erstellen von Storyboards aus textuellen Szenariobeschreibungen ein Vorgang, der aktuell vom Entwickler ohne Toolunterstützung durchgeführt wird. Da hier wichtige Designentscheidungen getroffen werden müssen, kann dieser Vorgang vermutlich nicht komplett automatisiert werden. Es wäre jedoch vorstellbar, durch Textanalyse wichtige Objekte, Attribute und Beziehungen für einen Schritt in einer Szenariobeschreibung zu identifizieren und einen ersten Vorschlag für ein Objektdiagramm zu generieren, das diese Elemente enthält. Der Entwickler kann dann ausgehend von diesen Vorschlägen, das fertige Szenario entwickeln. Am Lehrstuhl sind auf diesem Gebiet bereits erste, erfolgversprechende Versuche unternommen worden. Aktuell bearbeitet Jörn Dreyer das Thema in seiner Doktorarbeit.

Das bereits erwähnte CoObRA Framework wird auch als Persistenzframework von Fujaba selbst eingesetzt. CoObRA ermöglicht es, in Fujaba Projekte zu versionieren und erlaubt es, mit mehreren Entwickler an einem Projekt zu arbeiten, siehe [SZN04]. Da CoObRA die Änderungen an einem Projekt protokolliert und diese Änderungen auch mit Zeitstempeln versehen werden können, ist es leicht möglich herauszufinden, welcher Entwickler wann und wie lange an bestimmten Teilen des Projekts gearbeitet hat. Da die verschiedenen Diagrammarten, an denen der Entwickler arbeitet, zu verschiedenen Phasen im Fujaba Process gehören und normalerweise pro Iteration ein Szenario bearbeitet wird, das wiederum eindeutig einem Usecase zugeordnet werden kann, kann man herausfinden, wie lange ein Entwickler ungefähr an einem bestimmten Usecase in einer bestimmten Phase des FUPs gearbeitet hat und wie groß sein Beitrag zu diesem Usecase war. Eine Iteration des FUP beginnt hier immer mit dem Anlegen eines Storyboards und endet sobald der Test, der aus diesem Storyboard generiert wurde, erfolgreich durchläuft. Die statistischen Daten, wie lange ein Entwickler für einen bestimmtes Szenario braucht, können vollautomatisch erhoben, gesammelt und als Grundlage zur Größen- und Zeitschätzung genutzt werden. Zur Schätzung können die Szenarien und Storyboards der neu zu entwickelnden Funktionalität mit denen der Datenbasis verglichen und daraus eine Vorhersage getroffen werden, wie viel Zeit notwendig ist, um diese Funktionalität umzusetzen. Dadurch würde der FUP um eine Planungskomponente für das Projektmanagement erweitert werden.

Neben dem in dieser Arbeit vorgestellten Testen und Debuggen, fehlt Fujaba noch Toolunterstützung für zwei weitere Verfahren zum Auffinden von Fehlern: Reviewverfahren und Modellanalysen. Um Reviewverfahren zu unterstützen wäre ein mäch-

tiger Diff-Algorithmus nötig, der es erlaubt, die Änderungen, die ein Entwickler am Modell vorgenommen hat, übersichtlich und gut verständlich darzustellen. Einen solchen Algorithmus implementiert beispielsweise das SiDiff Tool der Universität Siegen [TBWK07] oder EMF-Compare für EMF-Modelle. Wird ein solcher Mechanismus in Fujaba integriert, wären modellbasierte Reviews einfacher möglich. Modellanalysen untersuchen das Modell auf Inkonsistenzen noch bevor Quelltext generiert wird. Fujaba bietet mit dem an der Universität Kassel entwickelten Inspections Plugin bereits ein Framework, das in der Lage ist, das Modell auf Inkonsistenzen zu überprüfen. Hierzu müssen Regeln definiert werden, die Inkonsistenzen erkennen und gegebenenfalls kleine Kommandos (sog. Quick-Fixes), die diese Inkonsistenzen beheben können. Momentan existieren aber nur wenige solcher Regeln. Um tatsächlich viele Fehler mit diesem Mechanismus finden zu können, muss ein mächtiger Regelkatalog entwickelt werden, wie ihn beispielsweise FindBugs [HP04] für Quelltextanalysen bietet.

Neben szenariobasierten Tests werden in der Qualitätssicherung heute vor allem Black- und Whitebox Testverfahren eingesetzt. Für Blackbox Tests können existierende Testgeneratoren automatisch Testfälle für Grenzfälle erstellen. So wird beispielsweise eine zu testenden Methode mit einem ganzzahligen Parameter einmal mit 0, einmal mit der kleinsten möglichen negativen Zahl und einmal mit der größten möglichen positiven Zahl aufgerufen. Eine solche Grenzfallbetrachtung ist bei komplexen Szenarien ungleich komplizierter, da diese komplexe Objektstrukturen enthalten. Folgearbeiten könnten untersuchen, inwieweit sich dennoch solche „Grenzszenarien“ automatisch generieren lassen.

Außerdem könnten die in den Szenarien enthaltenen Verhaltensbeschreibungen zusätzlich dazu genutzt werden, automatisch Prototypen zu generieren, die das in den Szenarien beschriebene Verhalten implementieren. So ein Prototyp ermöglicht es dem Kunden, die Szenarien selbst durchzuspielen und zu überprüfen, ob die Applikation das gewünschte Verhalten zeigt. Ist dies nicht der Fall, kann er den Entwickler darauf hinweisen, dass das Szenario berichtigt werden muss. Dies hat zusätzlich den Effekt, dass Fehler früh entdeckt werden. Die Umsetzung dieser Prototypgenerierung und der zuvor erwähnten Grenzfallgenerierung ist bereits in einem Verbundprojekt mit zwei mittelständischen Unternehmen geplant.

Im Debugging Kontext könnten nachfolgende Arbeiten beispielsweise den eDOBS erweitern. Der eDOBS kann bislang nur aus einer normalen Java Anwendungen auf dem lokalen Rechner gestartet werden, da er Funktionalität benutzt, die beispielsweise einer Applikation auf einem Enterprise Application Server nicht zu Verfügung steht. Im Rahmen einer weiterführenden Arbeit könnte untersucht werden, wie es möglich ist, die Funktionalität des eDOBS trotzdem für Enterpriseanwendungen nutzbar zu machen. Hierzu kann beispielsweise eine Weboberfläche erstellt werden, die komplexe Objektstrukturen von Webanwendungen darstellen kann. Ein solcher *WebDOBS* ermöglicht es Entwicklern und Domänenexperten die erstellten Szenarien anhand eines Web-Prototyps durchzuspielen und auf Korrektheit zu überprüfen.

Die Durchführung eines Objektspiels im eDOBS ist bislang nur möglich, wenn die Klassenstruktur bereits vorliegt. Das ist aber bei neu entstehender Software norma-

lerweise nicht der Fall. Um das Objektspiel auch ohne Klassenmodell zu ermöglichen, müsste der eDOBS in der Lage sein, generische Graphstrukturen darstellen zu können. Er müsste es also ermöglichen, Objekte und Links ohne vorliegende Klasseninformationen zu erzeugen. Die Grundlage hierzu wurde in dieser Arbeit durch die eDOBS Cache Layer in Abschnitt 18.2.8 geschaffen. Weiterführende Arbeiten können nun ein Cache Feature Abstraction Module implementieren, das sich durch neue, bisher unbekannte Klassen und Felder erweitern lässt. Auf diese Weise ließen sich Objektspiele auf beliebigen Objektstrukturen im eDOBS durchführen.

Zusätzlich könnten sowohl die Testgenerierung als auch das Debugging für andere Zielsprachen, wie beispielsweise C# bereitgestellt werden. Für die Testgenerierung würde dies im Wesentlichen das Schreiben von Codegenerierungstemplates für die neue Sprache bedeuten. Beim Debugging könnte der SMAP-Mechanismus wiederverwendet werden, es müsste jedoch eine Integration in den Debugger der entsprechenden Zielsprache erfolgen, was vermutlich mit erheblichen Aufwand verbunden ist.

Zusammenfassend hat diese Arbeit also einen wichtigen Beitrag geleistet, die modellbasierte Softwareentwicklung durch einen Prozessvorschlag sowie durch verbesserte Test- und Debuggingtechniken voranzubringen. An diese Arbeit schließen sich viele weitere spannende Forschungsfragen an, mit deren Beantwortung teilweise am Lehrstuhl für Software Engineering der Universität Kassel schon begonnen wurde.

A. Anhang

A.1. Veranstaltungsverzeichnis

Viele, der in dieser Arbeit entwickelten Methoden und Werkzeuge, wurden im Rahmen der Programmiermethodik Vorlesung des Fachgebiets Software Engineering an der Universität Kassel eingesetzt und erprobt. Lernziele dieser Veranstaltung sind

- objektorientierte Modellierung
- objektorientierte Szenarios
- objektorientierte Programmierung
- Test-First Prinzip
- Unified Modeling Language
- Story Driven Modeling
- methodische Vorgehensweise zur Entwicklung größerer Programme ($>$ 10000 LOC)

Die Programmiermethodik Vorlesung wurde seit dem Wintersemester 2002/2003 insgesamt sieben Mal gehalten. Die einzelnen Vorlesungen sind in Tabelle A.1 aufgeführt. Die Teilnehmeranzahl ist in Spalte *TN* angegeben. Zusätzlich zu den in Tabelle A.1 aufgeführten Veranstaltungen, gab es in den Semestern, in denen diese Vorlesung nicht angeboten wurde, noch Veranstaltungen für Nachschreiber. An diesen nahmen insgesamt 19 Studenten teil. So wurde die Programmiermethodik Vorlesung insgesamt von ca. 350 Studenten besucht.

Bis zum Sommersemester 2007 gehörte zu dieser Vorlesung auch immer ein begleitendes Projekt. In diesem entwickelten die Studenten einen Prototypen für eine rechnergestützte Brettspielsimulation. In 2007 wurde die Vorlesung von Grund auf umgestellt und das Projekt durch kleinere Übungen ersetzt. In der Zeile *Beispiel* in Tabelle A.1 ist das jeweilige Brettspiel des Projekts beziehungsweise die Beispiele der Übungen aufgeführt.

Im Rahmen dieser Vorlesung wurde immer das Objektspiel zum Erstellen von Objektdiagrammen eingeführt. Ebenfalls wurden hieraus immer Klassendiagramme abgeleitet. In allen Vorlesungen kam auch der Dobs beziehungsweise nach Fertigstellung dann auch der eDOBS zum Einsatz. Weitere in den Übungen eingesetzte Werkzeuge aus dieser Arbeit zeigt die Spalte *Einsatz in Übungen und Hausaufgaben*. Für den Einsatz der Testgenerierung aus Storydiagrammen steht in dieser Spalte abkürzend *Tests*. Für diese Arbeit relevante Themen, die in der Vorlesung besprochen wurden, sind in Spalte *Vorlesung* aufgeführt. Man beachte, dass Fujaba4Eclipse und somit

auch einige der in dieser Arbeit vorgestellten Integrationen in Fujaba4Eclipse erst in der Vorlesung 2009 eingeführt wurde. Diese Werkzeuge wurden also erst in einer Veranstaltung erprobt. Für die nächsten Jahre ist allerdings weiterhin der Einsatz von Fujaba4Eclipse im Rahmen dieser Vorlesung geplant.

Zusammenfassend haben wir also bereits eine Menge Erfahrungen mit dem Einsatz der Methoden und Werkzeuge dieser Arbeit in der Lehre sammeln können. Unsere Beobachtungen waren, wie in den Kapiteln 8, 13 und 19 beschrieben, sehr positiv. Die Werkzeuge und Methoden helfen den Studenten beim Lernen und beim Erstellen guter Modelle.

Semester	TN	Beispiel	Einsatz in Übungen und Hausaufgaben	Vorlesung	URL
WS 02/03	26	Missisipi Queen	Fujaba, Storyboards, Tests, Dobs		http://www.se.eecs.uni-kassel.de/typo3/index.php?id=519
WS 03/04	39	RoboRally	Fujaba, Storyboards, Tests, Dobs		http://www.se.eecs.uni-kassel.de/typo3/index.php?pmlecture
WS 04/05	80	Kahuna	Fujaba, Storyboards, Tests, Dobs		http://www.se.eecs.uni-kassel.de/typo3/index.php?pmlecture0405
SS 06	30	Tikal	Fujaba, Storyboards, Tests, Dobs	Fujaba Machine	http://www.se.eecs.uni-kassel.de/typo3/index.php?pmlecture04050
SS 07	71	Klausurenverwaltungssystem, Ada, Hanoi	Fujaba, eDOBS, Zetteltest, JUnit manuell	Storyboards, Tests, Fujaba Machine	http://www.se.eecs.uni-kassel.de/typo3/index.php?pmss2007
SS 08	62	Raumreservierung, Ludo, Ada	Fujaba, eDOBS, Zetteltest, JUnit manuell	Storyboards, Tests, Dreisprung, Fujaba Machine	http://www.se.eecs.uni-kassel.de/typo3/index.php?pm08
SS 09	24	Risiko, TopTrump, Ada	Fujaba4Eclipse, Storyboards, Tests, eDOBS, Zetteltest	Dreisprung, Fujaba Machine	http://www.se.eecs.uni-kassel.de/typo3/index.php?ss09pm

Tabelle A.1.: Vorlesungen Programmiermethodik

A.2. Template für die zu-n Suche

```
#set( $return = $imports.addToImports("java.util.*") )
#set( $postfix = "$utility.upFirstChar($sourceName)To$utility.
upFirstChar($targetName)" )
#set( $iter = "fujaba__Iter$postfix" )
#set( $children = $utility.indent("      ",$!children) )
#set( $localVars = $engine.getFromInformation("localVars") )
#set( $return = $localVars.addLocalVariable($iter, "Iterator", $link) )
#set( $searchOpComment = "iterate to-many link $link.InstanceOf.Name
 from $sourceName to $targetName" )
// $searchOpComment
#if ( $targetSet )
#set( $return = $imports.addToImports("de.upb.tools.fca.*") )
#set( $return = $localVars.addLocalVariable($collectionName,
 "FHashSet", $link) )
$collectionName = new FHashSet ();
#else
fujaba__Success = false;
#end
#if( $path )
$iter = new de.uni_kassel.sdm.Path ($sourceName, "$path");
##$iter = new Path ($sourceName, "$path");
#else
$iter = #parse("$!{style}storyPattern/link/iterator.vm" )

#end
#if ( $upperbound )
fujaba__UpperBound_$postfix = false;
#set( $return = $localVars.addLocalVariable(
"fujaba__UpperBound_$postfix","boolean","false", $link) )
#set( $upperadd = " && !(fujaba__UpperBound_$postfix)" )
#end
#set( $debug = $debug || $elem.firstFromDiagrams.parentElement.
hasKeyInStereotypes("assert") )
#if( $debug )
List causes$postfix = new ArrayList<JavaSDMException>();
#end
while ( #if(!$forEach && !$targetSet || $targetNegative || $check)
!(fujaba__Success) && #end${iter}.hasNext ()$!upperadd )
{
    try
    {
#if($check)
      JavaSDM.ensure (${targetName}.equals (${iter}.next ())
#parse("$lang/default:storyPattern/sdmComment.vm"));
#else
#set( $typeCast = (!$role.getTarget().
equals($target.getInstanceOf())) )
#set( $source = "#if(!$typeCast)($targetType)#end ${iter}.next ()" )
#set( $toMany = true )
#set( $targetToken = $dlrTool.createDLRToken([$target]) )
$!targetToken.createStartTag()##
#set( $template = "$lang/default:storyPattern/object/assign.vm" )
```

```
53 $utility.indent("        ", "#parse($template)" )##
54 $!targetToken.createEndTag()##
55 #if($targetToken)#set( $targetToken.comment = $opComment )#end
56 #if ( $upperbound )
57       if ( ${targetName}.equals ($upperbound) )
58       {
59          fujaba__UpperBound_$postfix = true;
60          throw new JavaSDMException () ;
61       }
62 #end
63 #end
64 #set( $opComment = $searchOpComment )
65 $!children
66 #if ( $targetSet )
67       ${collectionName}.add ($targetName);
68 #else
69       fujaba__Success = true;
70 #end
71    }
72    catch ( JavaSDMException fujaba__InternalException )
73    {
74 #if( $debug )
75       causes${postfix}.add (fujaba__InternalException);
76 #end
77       fujaba__Success = false;
78    }
79 }
80 #if ( $targetOptional )
81 if (!fujaba__Success)
82 {
83    fujaba__Success = true;
84    $targetName = null;
85 }
86 #elseif ( !$targetSet )
87 JavaSDM.ensure (##
88 #if( $path && $linkNegative )!#end
89 fujaba__Success#if( $debug ), "$opComment", causes$postfix#end);
90 #end
```

Listing A.1: Template toMany.vm

Literaturverzeichnis

[ADH+09] Aschenbrenner, Nina ; Dreyer, Jorn ; Hahn, Marcel ; Jubeh, Ruben ; Schneider, Christian ; Zundorf, Albert: Building distributed web applications based on model versioning with CoObRa: An experience report. In: *CVSM '09: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*. Washington, DC, USA : IEEE Computer Society, 2009. – ISBN 978–1–4244–3714–6, S. 19–24

[AKRS06] Amelunxen, C. ; Königs, A. ; Rötschke, T. ; Schürr, A.: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In: *A. Rensink, J. Warmer (eds.), Model Driven Architecture - Foundations and Applications: Second European Conference, Heidelberg, Lecture Notes in Computer Science (LNCS), Vol. 4066, 361–375* Springer Verlag, 2006

[BA04] Beck, Kent ; Andres, Cynthia: *Extreme Programming Explained : Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004 `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0321278658`. – ISBN 0321278658

[Bal96] Balzert, Helmut: *Lehrbuch der Software-Technik: Teil 1: Software-Entwicklung*. Heidelberg, Germany : Spektrum Akademischer Verlag, 1996. – ISBN 3–8274–0042–2

[Bal05] Balzert, Heide: *UML 2 in 5 Tagen: Der schnelle Einstieg in die Objektorientierung*. Witten, Germany : W3L. GmbH, 2005. – ISBN 3937137610

[Bal08] Balzert, Helmut: *Lehrbuch der Software-Technik: Softwaremanagement*. 2. Heidelberg : Spektrum, 2008. – ISBN 978–3–8274–1161–7

[Böc06] Böckers, Philipp: *Automatisierte Tests für Zutrittskontrollsysteme*. Aachen, Germany, RWTH Aachen, Diplomarbeit, 2006. `http://ulno.net/advising/`. – Diplomarbeit

[BCSW03] Baresel, A. ; Conrad, M. ; Sadeghipour, S. ; Wegener, J.: The interplay between model coverage and code coverage. In: *Proceedings of the 11th European International Conference on Software Testing, Analysis and Review (EuroSTAR '03)*. Amsterdam, The Netherlands, 2003

[BD04] Bruegge, Bernd ; Dutoit, Allen H.: *Objektorientierte Softwaretechnik*. Pearson Studium, 2004 `http://www.worldcat.org/isbn/3827370825`. – ISBN 3827370825

[BDG+04] Baker, Paul ; Dai, Zhen R. ; Grabowski, Jens ; Haugen, Oystein ; Samuelsson, Eric ; Schieferdecker, Ina ; Williams, Clay E.: The

UML 2.0 Testing Profile. In: *Proceedings of the '8th Conference on Quality Engineering in Software Technology 2004' (CONQUEST 2004) in Nuremberg (Germany), September 22-24, 2004, ISBN 3-9809145-1-8*, ASQF e.V., Erlangen, September 2004, S. 181–189

[BE05] Bontemps, Yves ; Egyed, Alexander: Scenarios and state machines: models, algorithms, and tools: a summary of the 4th workshop. In: *SIGSOFT Softw. Eng. Notes* 30 (2005), Nr. 5, S. 1–4. `http://dx.doi.org/http://doi.acm.org/10.1145/1095430.1095437`. – DOI http://doi.acm.org/10.1145/1095430.1095437. – ISSN 0163–5948

[BGSZ08] Bork, Manuel ; Geiger, Leif ; Schneider, Christian ; Zündorf, Albert: Towards Roundtrip Engineering - A Template-Based Reverse Engineering Approach. In: *ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture*. Berlin, Heidelberg : Springer-Verlag, 2008. – ISBN 978–3–540–69095–5, S. 33–47

[BL01] Briand, L. ; Labiche, Y.: A UML-Based Approach to System Testing. In: *4th International Conference on the Unified Modeling Language (UML)*. Toronto, Canada, 2001, S. 194–208

[BMM05] Bednarik, Roman ; Moreno, Andres ; Myller, Niko: Jeliot 3, an Extensible Tool for Program Visualization. In: *Proceedings of the Fifth Annual Finnish / Baltic Sea Conference on Computer Science Education (Koli Calling 2005)*, 2005, S. 183–184

[Boo90] Booch, Grady: *Object-Oriented Design with Applications*. Benjamin/-Cummings, 1990

[BRJ98] Booch, Grady ; Rumbaugh, James ; Jacobson, Ivar: *The Unified Modeling Language User Guide*. Addison-Wesley Professional, 1998. – ISBN 0201571684

[CJ07] Czyz, Jeffrey K. ; Jayaraman, Bharat: Declarative and visual debugging in Eclipse. In: *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*. New York, NY, USA : ACM, 2007. – ISBN 978–1–60558–015–9, S. 31–35

[Czo04] Czok, Matthias: *UML basierende Analyse, Design und softwaretechnische Realisierung einer Funkkommunikation mit dem ASCA Protokoll*. Kassel, Germany, Kassel University, Diplomarbeit, 2004. – Diplomarbeit

[DGMZ02] Diethelm, Ira ; Geiger, Leif ; Maier, Thomas ; Zündorf, Albert: Turning Collaboration Diagram Strips into Storycharts. In: *SCESM '02: Proceedings of the first international workshop on Scenarios and state machines: models, algorithms and tools*. Florida, Orlando, USA, May 2002

[DGSZ04a] Diethelm, Ira ; Geiger, Leif ; Schneider, Christian ; Zündorf, Albert: Automatic Time Measurement for UML Modeling Activities. In: *Informatics and Student Assessment - Concepts of Empirical Research and Standardisation of Measurement in the Area of Didactics of Informatics*

1 (2004), 39-50. http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/DGSZ04a.pdf

[DGSZ04b] Diethelm, Ira ; Geiger, Leif ; Schneider, Christian ; Zündorf, Albert: Measurement of modeling abilities. In: *Informatics and Student Assessment - Concepts of Empirical Research and Standardisation of Measurement in the Area of Didactics of Informatics* 1 (2004), 51-64. http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/DGSZ04b.pdf

[DGSZ06] Diethelm, Ira ; Geiger, Leif ; Schneider, Christian ; Zündorf, Albert: An Execution Model for teaching Story Diagrams. In: *Workshop Modellierung in Lehre und Weiterbildung (Modellierung 2006)*. Innsbruck, Austria, March 2006

[DGZ02] Diethelm, Ira ; Geiger, Leif ; Zündorf, Albert: UML im Unterricht: Systematische objektorientierte Problemlösung mit Hilfe von Szenarien am Beispiel der Türme von Hanoi. In: Schubert, Sigrid E. (Hrsg.) ; Magenheim, Johannes (Hrsg.) ; Hubwieser, Peter (Hrsg.) ; Brinda, Torsten (Hrsg.): *DDI* Bd. 22, GI, 2002 (LNI). – ISBN 3–88579–351–2, 33-42

[DGZ03a] Diethelm, Ira ; Geiger, Leif ; Zündorf, Albert: Applying Story Driven Modeling to the Paderborn Shuttle System Case Study. In: Leue, Stefan (Hrsg.) ; Systä, Tarja (Hrsg.): *Scenarios: Models, Transformations and Tools* Bd. 3466, Springer, 2003 (Lecture Notes in Computer Science). – ISBN 3–540–26189–3, 109-133

[DGZ03b] Diethelm, Ira ; Geiger, Leif ; Zündorf, Albert: Fujaba goes Mindstorms: Objektorientierte Modellierung zum Anfassen. In: Hubwieser, Peter (Hrsg.): *INFOS* Bd. 32, GI, 2003 (LNI). – ISBN 3–88579–361–X, 225-235

[DGZ04] Diethelm, Ira ; Geiger, Leif ; Zündorf, Albert: Systematic Story Driven Modeling, a case study. In: *SCESM '04: Proceedings of the third international workshop on Scenarios and state machines: models, algorithms and tools*. Edinburgh, Scottland, May 2004

[DGZ05a] Diethelm, Ira ; Geiger, Leif ; Zündorf, Albert: Mit Klebezettel und Augenbinde durch die Objektwelt. In: [Fri05], 149-159

[DGZ05b] Diethelm, Ira ; Geiger, Leif ; Zündorf, Albert: Rettet Prinzessin Ada: Am leichtesten objektorientiert. In: [Fri05], 161-172

[DGZ05c] Diethelm, Ira ; Geiger, Leif ; Zündorf, Albert: Teaching Modeling with Objects First. In: *8th World Conference on Computers in Education (WCCE)*. Cape Town, South Africa, July 2005

[DGZ08] Diethelm, Ira ; Geiger, Leif ; Zündorf, Albert: What's a Good Model and How to Teach It? – Introducing object oriented modeling by using scenarios. In: *ICT and Learning for the Net Generation - LYICT 2008*. Kuala Lumpur, Malaysia, July 2008

[Die02] DIEHL, Stephan (Hrsg.): *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures*. Bd. *2269*. Springer, 2002 (Lecture Notes in Computer Science). – ISBN 3–540–43323–6

[Die07] DIETHELM, Ira: *SStrictly models and objects first Unterrichtskonzept und -methodik für objektorientierte Modellierung im Informatikunterricht*, Kassel University, Diss., 2007. `http://kobra.bibliothek.uni-kassel.de/handle/urn:nbn:de:hebis:34-2007101119340`

[Dre08] DREYER, Jörn: *Analyse von Programmabläufen durch zeitliche Darstellung der Modifikation relevanter Laufzeit-Objektstrukturen*. Kassel, Germany, Kassel University, Diplomarbeit, 2008. – Diplom II Arbeit

[DW98] D'SOUZA, Desmond ; WILLS, Alan C.: *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998. – ISBN 0201310120

[ECM06] ECMA: *Eiffel: Analysis, Design and Programming Language*. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-367.pdf`, zuletzt besucht 19.01.2010, 2006

[EGK+03] EGYED, Alexander ; GLINZ, Martin ; KRUEGER, Ingolf ; SYSTÄ, Tarja ; UCHITEL, Sebastian ; ZÜNDORF, Albert: Second Workshop on Scenarios and State Machines: Models, Algorithms, and Tools. In: *Second Workshop on Scenarios and State Machines*, 2003

[Fag02] FAGAN, Michael: Reviews and Inspections. In: *Software Pioneers - Contributions to Software Engineering*, Springer Verlag, 0 2002, S. 562–573

[FNTZ98] FISCHER, T. ; NIERE, J. ; TORUNSKI, L. ; ZÜNDORF, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation* Paderborn, Germany, 1998

[Fri05] FRIEDRICH, Steffen (Hrsg.): *Unterrichtskonzepte für informatische Bildung, INFOS 2005, 11. GI-Fachtagung Informatik und Schule, 28.-30. September 2005 an der TU Dresden*. Bd. *60*. GI, 2005 (LNI). – ISBN 3–88579–389–X

[GB03] GAMMA, Erich ; BECK, Kent: *Contributing to Eclipse: Principles, Patterns, and Plugins*. Redwood City, CA, USA : Addison Wesley Longman Publishing Co., Inc., 2003. – ISBN 0321205758

[GBD07] GEIGER, Leif ; BUCHMANN, Thomas ; DOTOR, Alexander: EMF Code Generation with Fujaba. In: [GGZ07], 25-29

[Gei02a] GEIGER, Leif: *Design Level Debugging mit Fujaba*. Braunschweig, Germany, Corolo Wilhelmina zu Braunschweig, Diplomarbeit, 2002. `http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/DLD.pdf`. – Bachelor Thesis (Studienarbeit)

[Gei02b] GEIGER, Leif: Design Level Debugging mit Fujaba. In: *Informatiktage*

2002 der Gesellschaft für Informatik (GI). Bad Schussenried, Germany, November 2002

[Gei04] Geiger, Leif: *Automatische JUnit Testgenerierung aus UML-Szenarien mit Fujaba*. Braunschweig, Germany, Corolo Wilhelmina zu Braunschweig, Diplomarbeit, 2004. `http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/TestGen.pdf`. – Diploma Thesis

[Gei08] Geiger, Leif: Model Level Debugging with Fujaba. In: *Proc. of the sixth International Fujaba Days 2008, Dresden, Germany*. Dresden, Germany, September 2008

[Gem08] Gemmerich, Ralf: *Eine Methode zur Generierung einer kostenoptimalen Bordnetzarchitektur*, Kassel University, Diss., 2008. `http://www.uni-kassel.de/upress/publi/abstract.php?978-3-89958-427-1`

[Gen10] Gentleware AG: *gentleware homepage*. `http://www.gentleware.com/`, zuletzt besucht 19.01.2010, 2010

[GGZ+05] Grunske, Lars ; Geiger, Leif ; Zündorf, Albert ; Eetvelde, Niels V. ; Gorp, Pieter V. ; Varró, Dániel: Using Graph Transformation for Practical Model-Driven Software Engineering. In: Beydeda, Sami (Hrsg.) ; Book, Matthias (Hrsg.) ; Gruhn, Volker (Hrsg.): *Model-Driven Software Development*, Springer, 2005. – ISBN 978–3–540–25613–7, 91-117

[GGZ07] Geiger, Leif (Hrsg.) ; Giese, Holger (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proc. of the Fifth International Fujaba Days 2007*. Bd. *tr-ri-07-289*. University of Paderborn, 2007 (Technical Report)

[GHJV05] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns. Elements of Reusable Objekt-Oriented Software*. Addison-Wesley Professional, 2005. – ISBN 0201633612

[GK04] Giese, Holger ; Krüger, Ingolf: Third Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM'04). In: *Proc. of the 26th International Conference on Software Engineering*, IEEE Computer Society Press, 0 2004, S. 756–757

[GS07] Geiger, Leif ; Schneider, Christian: Copy & Paste concept and realization in Fujaba. In: [GGZ07], 21-24

[GSR05] Geiger, Leif ; Schneider, Christian ; Reckord, Carsten: Template- and modelbased code generation for MDA-Tools. In: Giese, Holger (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proc. of the third International Fujaba Days 2005, Paderborn, Germany* Bd. tr-ri-05-259. Paderborn, Germany : University of Paderborn, September 2005 (Technical Report), S. 57–62

[GSZ03] Geiger, Leif ; Schneider, Christian ; Zündorf, Albert: Integrated, Document Centered Modelling in Fujaba. In: Giese, Holger (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proc. of the first International Fujaba Days 2003, Kassel, Germany* Bd. tr-ri-04-247. Kassel, Germany : University of Paderborn, October 2003 (Technical Report), S. 25–28

[GSZ05a] Geiger, Leif ; Siedhof, Jörg ; Zündorf, Albert: μFUP: A Software Development Process for Embedded Systems. In: *Modellbasierte Entwicklung eingebetteter Systeme (MBEES)* (2005), January. `http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/GSZ05b.pdf`

[GSZ05b] Geiger, Leif ; Siedhof, Jörg ; Zündorf, Albert: OO Model based programming of PCLs. In: *Modellbasierte Entwicklung eingebetteter Systeme (MBEES)* (2005), January. `http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/GSZ05a.pdf`

[GSZ+05c] Gemmerich, R. ; Semmelrodt, S. ; Zündorf, A. ; Reckord, C. ; Leohold, J. ; Trippler, J. ; Brabetz, L. ; Müller, D. ; Schrey, U. ; Weil, H.-G.: Ein ganzheitlicher Ansatz zur Generierung und Optimierung von Fahrzeugbordnetzen. In: *12th International Conference and Exhibition Electronic Systems for Vehicles Baden-Baden (VDI Berichte Nr. 1907), pp. 597-608* VDI Verein Deutscher Ingenieure (Hrsg.), Germany), 2005

[GZ02a] Geiger, Leif ; Zündorf, Albert: Graph Based Debugging with Fujaba. In: *Electr. Notes Theor. Comput. Sci.* 72 (2002), Nr. 2. `http://dblp.uni-trier.de/db/journals/entcs/entcs72.html#GeigerZ02`

[GZ02b] Geiger, Leif ; Zündorf, Albert: Graph Based Debugging with Fujaba. In: *International Workshop on Graph-Based Tools (GraBaTs); ICGT Workshop*. Barcelona, Spain, October 2002

[GZ03] Geiger, Leif ; Zündorf, Albert: Transforming Graph Based Scenarios into Graph Transformation Based JUnit Tests. In: Pfaltz, John L. (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Böhlen, Boris (Hrsg.): *AGTIVE* Bd. 3062, Springer, 2003 (Lecture Notes in Computer Science). – ISBN 3–540–22120–4, 61-74

[GZ04] Geiger, Leif ; Zündorf, Albert: Statechart Modeling with Fujaba. In: *2nd International Workshop on Graph-Based Tools (GraBaTs); ICGT Workshop*. Rom, Italy, September 2004

[GZ05] Geiger, Leif ; Zündorf, Albert: Story driven testing - SDT. In: *SCESM '05: Proceedings of the fourth international workshop on Scenarios and state machines: models, algorithms and tools*. New York, NY, USA : ACM, May 2005. – ISBN 1–58113–963–2, 1–6

[GZ06a] Geiger, Leif ; Zündorf, Albert: Developing Tools with Fujaba XProM. In: Lämmel, Ralf (Hrsg.) ; Saraiva, João (Hrsg.) ; Visser, Joost (Hrsg.): *GTTSE* Bd. 4143, Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–45778–X, 344-356

[GZ06b] Geiger, Leif ; Zündorf, Albert: eDOBS - Graphical Debugging for eclipse. In: *Electronic Communications of the EASST* 1 (2006). `http://eceasst.cs.tu-berlin.de/index.php/eceasst/issue/view/9`. – ISSN 1863–2122

[GZ06c] Geiger, Leif ; Zündorf, Albert: Tool Modeling with Fujaba. In: *Electr.*

Notes Theor. Comput. Sci. 148 (2006), Nr. 1, 173-186. `http://dblp.uni-trier.de/db/journals/entcs/entcs148.html#GeigerZ06`

[Hag99] Hager, Oliver: *Die Dobs Architektur und die Java Runtime Bibliothek.* Paderborn, Germany, University of Paderborn, Diplomarbeit, 1999. `http://wwwcs.uni-paderborn.de/cs/jevox/Seminar/DobsArchitektur.pdf`. – Seminararbeit im Rahmen der Projektgruppe JEVOX

[Hof09] Hoffmann, Marc R.: *Java Code Coverage for Eclipse.* `http://www.eclemma.org/`, zuletzt besucht 19.01.2010, 2009

[HP04] Hovemeyer, David ; Pugh, William: Finding bugs is easy. In: *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications.* New York, NY, USA : ACM, 2004. – ISBN 1–58113–833–4, S. 132–136

[IBM10a] IBM: *IBM Rational Software.* `http://www-01.ibm.com/software/de/rational/`, zuletzt besucht 19.01.2010, 2010

[IBM10b] IBM: *Rational DOORS.* `http://www-01.ibm.com/software/awdtools/doors/`, zuletzt besucht 19.01.2010, 2010

[IHJB07] II, James H. C. ; Hendrix, T. D. ; Jain, Jhilmil ; Barowski, Larry A.: Dynamic Object Viewers for Data Structures. In: *Proceedings of SIGCSE 2007*, 2007

[JBR99] Jacobson, Ivar ; Booch, Grady ; Rumbaugh, James: *The Unified Software Development Process.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999. – ISBN 0–201–57169–2

[JK01] J. Koskinen, T. S. E. Mäkinen M. E. Mäkinen: Minimally Adequate Synthesizer Tolerates Inaccurate Information during Behavioral Modeing. In: *SCASE 2001.* Enschede, Netherlands, 2001

[Jue06] Juergeleit, Torsten: *ANTLR plugin for Eclipse.* `http://antlreclipse.sourceforge.net/`, zuletzt besucht 19.01.2010, 2006

[JW03] J. Whittle, J. S. R. Kwan K. R. Kwan: From Scenarios to Code: An Air Traffic Control Case Study. In: *ICSE2003.* Portland, USA, 2003

[KDV07] Ko, Andrew J. ; DeLine, Robert ; Venolia, Gina: Information Needs in Collocated Software Development Teams. In: *ICSE '07: Proceedings of the 29th international conference on Software Engineering.* Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2828–7, S. 344–353

[KGSB99] Krüger, I. ; Grosu, R. ; Scholz, P. ; Broy, M.: From MSCs to Statecharts. In: Rammig, Franz J. (Hrsg.): *Distributed and Parallel Embedded Systems*, Kluwer Academic Publishers, 1999

[KM08] Ko, Andrew J. ; Myers, Brad A.: Debugging reinvented: asking and answering why and why not questions about program behavior. In: *ICSE*

'08: Proceedings of the 30th international conference on Software engineering. New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–079–1, S. 301–310

[KMCA06] Ko, Andrew J. ; Myers, Brad A. ; Coblenz, Michael J. ; Aung, Htet H.: An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. In: *IEEE Trans. Softw. Eng.* 32 (2006), Nr. 12, S. 971–987. http://dx.doi.org/http://dx.doi.org/10.1109/TSE.2006.116. – DOI http://dx.doi.org/10.1109/TSE.2006.116. – ISSN 0098–5589

[KNNZ00] Köhler, Hans J. ; Nickel, Ulrich ; Niere, Jörg ; Zündorf, Albert: Integrating UML diagrams for production control systems. In: *ICSE '00: Proceedings of the 22nd international conference on Software engineering*. New York, NY, USA : ACM, 2000. – ISBN 1–58113–206–9, S. 241–251

[KQPR03] Kölling, Michael ; Quig, Bruce ; Patterson, Andrew ; Rosenberg, John: The BlueJ system and its pedagogy. In: *Journal of Computer Science Education* 13 (2003), December, Nr. 4. http://www.bluej.org/papers/2003-12-CSEd-bluej.pdf

[KS06] Königs, Alexander ; Schürr, Andy: Tool Integration with Triple Graph Grammars - A Survey. In: *Electronic Notes in Theoretical Computer Science* 148 (2006), February, Nr. 1, 113–150. http://dx.doi.org/10.1016/j.entcs.2005.12.015. – DOI 10.1016/j.entcs.2005.12.015

[Lab08] Laboratories, Serbian O.: *SOL UML Debugger*. http://www.sol.rs/index.php?page=umldebugger&pic=city, zuletzt besucht 19.01.2010, 2008

[Lew03] Lewis, Bil: Debugging Backwards in Time. In: *CoRR* cs.SE/0310016 (2003). http://dblp.uni-trier.de/db/journals/corr/corr0310.html#cs-SE-0310016. – informal publication

[LLQC07] Li, Bao-Lin ; Li, Zhi shu ; Qing, Li ; Chen, Yan-Hong: Test Case Automate Generation from UML Sequence Diagram and OCL Expression. In: *CIS '07: Proceedings of the 2007 International Conference on Computational Intelligence and Security*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–3072–9, S. 1048–1052

[LRRM03] Leroux, Hugo ; Réquilé-Romanczuk, Annya ; Mingins, Christine: JACOT: a tool to dynamically visualise the execution of concurrent Java programs. In: *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*. New York, NY, USA : Computer Science Press, Inc., 2003. – ISBN 0–9544145–1–9, S. 201–206

[Meh01] Mehner, Katharina: JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. In: [Die02], S. 163–175

[Men10] Mentor, Object: *JUnit homepage*. http://junit.org/, zuletzt besucht 19.01.2010, 2010

[Mor07] Morley, Liam: *e-bob - Eclipse-Based Object Bench.* `http://ebob.sourceforge.net/`, zuletzt besucht 19.01.2010, 2007

[MST04] Mens, Tom (Hrsg.) ; Schürr, Andy (Hrsg.) ; Taentzer, Gabriele (Hrsg.): *GraBaTs '04: Proceedings of the 2nd International Workshop on Graph-Based Tools, Rome, Italy, October 2, 2004.* 2004

[Net10] NetBeans Community: *NetBeans homepage.* `http://netbeans.org/index.html`, zuletzt besucht 19.01.2010, 2010

[NM06] Norbisrath, Ulrich ; Mosler, Christof: Functionality configuration for eHome systems. In: *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research.* New York, NY, USA : ACM, 2006, S. 8

[No 10] No Magic, Inc.: *MagicDraw homepage.* `http://www.magicdraw.com/`, zuletzt besucht 19.01.2010, 2010

[Nor10] Norbisrath, Ulrich: *Ulrich Norbisrath's Fujaba page.* `http://ulno.net/fujaba/`, zuletzt besucht 19.01.2010, 2010

[NSZ03] Nickel, U. ; Schäfer, W. ; Zündorf, A.: Integrative Specification of Distributed Production Control Systems for Flexible Automated Manufacturing. In: Nagl, M. (Hrsg.) ; Westfechtel, B. (Hrsg.): *Modelle Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen*, Wiley-VCH, 2003

[NZ00] Niere, Jörg ; Zündorf, Albert: Testing and Simulating Production Control Systems Using the Fujaba Environment. In: *AGTIVE '99: Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance.* London, UK : Springer-Verlag, 2000. – ISBN 3–540–67658–9, S. 449–456

[OA00] Offutt, Jeff ; Abdurazik, Aynur: Using UML Collaboration Diagrams for Static Checking and Test Generation. In: *3th International Conference on the Unified Modeling Language (UML).* York, UK, 2000, S. 383–395

[Obj10] Object Management Group, Inc.: *Object Management Group.* `http://www.omg.org/`, zuletzt besucht 19.01.2010, 2010

[OMG03] OMG: *MDA Guide Version 1.0.1.* `http://www.omg.org/docs/omg/03-06-01.pdf`. Version: June 2003

[OMG08] OMG: *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification.* `http://www.omg.org/cgi-bin/doc?formal/08-04-03.pdf`. Version: 2008

[OS02] Oechsle, Rainer ; Schmitt, Thomas: JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). In: *Revised Lectures on Software Visualization, International Seminar.* London, UK : Springer-Verlag, 2002. – ISBN 3–540–43323–6, S. 176–190

[Pro04] PROGRES GROUP: *PROGRES: Programmed Graph Rewriting System.* `http://www-i3.informatik.rwth-aachen.de/research/projects/progres/`, zuletzt besucht 19.01.2010, 2004

[PTP07] POTHIER, Guillaume ; TANTER Éric ; PIQUER, José M.: Scalable omniscient debugging. In: GABRIEL, Richard P. (Hrsg.) ; BACON, David F. (Hrsg.) ; LOPES, Cristina V. (Hrsg.) ; JR., Guy L. S. (Hrsg.): *OOPSLA*, ACM, 2007. – ISBN 978–1–59593–786–5, 535-552

[RG00] RYSER, J. ; GLINZ, M.: Using Dependency Charts to Improve Scenario-Based Testing. In: *17th International Conference on Testing Computer Software (TCS2000)*. Washington D.C., USA, 2000

[Rou05] ROUBTSOV, Vlad: *EMMA: a free Java code coverage tool.* `http://emma.sourceforge.net/`, zuletzt besucht 19.01.2010, 2005

[RQA02] *Rational Software Corporation: Rational Quality Architect Realtime Edition User's Guide.* `http://publibfp.boulder.ibm.com/epubs/pdf/12656660.pdf`, 2002

[Sch07] SCHNEIDER, Christian: *CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrkonzepten*, Kassel University, Diss., 2007. `http://kobra.bibliothek.uni-kassel.de/handle/urn:nbn:de:hebis:34-2007121319874`

[SGW94] SELIC, Bran ; GULLEKSON, Garth ; WARD, Paul T.: *Real-Time Object-Oriented Modeling.* John Wiley & Sons, Inc., 1994. – ISBN 0471599174

[Sih05] SIHLING, Marc: *Modellbasierte Dokumentation mit dem V-Modell® XT - ein Erfahrungsbericht.* `http://www.software-kompetenz.de/servlet/is/29287/ModellbasierteDokumentation_2005.pdf?command=downloadContent&filename=ModellbasierteDokumentation_2005.pdf`. Version: 2005

[STSN02] STEIMANN, Friedrich ; THADEN, Uwe ; SIBERSKI, Wolf ; NEJDL, Wolfgang: Animiertes UML als Medium für die Didaktik der objektorientierten Programmierung. In: *Modellierung 2002: Modellierung in der Praxis - Modellierung für die Praxis*, GI, 2002. – ISBN 3–88579–342–3, S. 159–170

[Sun04] SUN MICROSYSTEMS, INC.: *Java Platform Debugger Architecture.* `http://java.sun.com/j2se/1.5.0/docs/guide/jpda/`, zuletzt besucht 19.01.2010, 2004

[Sun09] SUN MICROSYSTEMS, INC.: *JSR-000045 Debugging Support for Other Languages.* `http://jcp.org/aboutJava/communityprocess/final/jsr045/index.html`, zuletzt besucht 19.01.2010, 2009

[Sun10] SUN MICROSYSTEMS, INC.: *JavaServer Pages Technology.* `http://java.sun.com/products/jsp/`, zuletzt besucht 19.01.2010, 2010

[SWZ95] SCHÜRR, Andy ; WINTER, Andreas ; ZÜNDORF, Albert: Graph Grammar Engineering with PROGRES. In: SCHÄFER, Wilhelm (Hrsg.): *Software Engineering - ESEC '95 (LNCS 989)*, Springer, 1995, S. 219–234

[SZG07] Stölzel, Mirco ; Zschaler, Steffen ; Geiger, Leif: Integrating OCL and Model Transformations in Fujaba. In: *Electronic Communications of the EASST* 5 (2007). `http://eceasst.cs.tu-berlin.de/index.php/eceasst/issue/view/12`. – ISSN 1863–2122

[SZN04] Schneider, Christian ; Zündorf, Albert ; Niere, Jörg: CoObRA - a small step for development tools to collaborative environments. In: *Workshop on Directions in Software Engineering Environments in 26th international conference on software engineering*. Edinburgh, Scotland, UK, May 2004

[TBWK07] Treude, Christoph ; Berlik, Stefan ; Wenzel, Sven ; Kelter, Udo: Difference computation of large models. In: *ESEC-FSE '07: Foundations of Software Engineering*. Dubrovnik, Croatia : ACM, September 2007, 295–304

[The09] The Apache Software Foundation: *Velocity Homepage*. `http://velocity.apache.org/`, zuletzt besucht 19.01.2010, 2009

[USZ02] Uchitel, Sebastian ; Systä, Tarja ; Zündorf, Albert: Scenarios and state machines: models, algorithms, and tools. In: *Software Engineering, International Conference on* 0 (2002), S. 659. `http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/ICSE.2002.1008016`. – DOI http://doi.ieeecomputersociety.org/10.1109/ICSE.2002.1008016. – ISSN 0570–5257

[WGM06] Whittle, Jon (Hrsg.) ; Geiger, Leif (Hrsg.) ; Meisinger, Michael (Hrsg.): *SCESM '06: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, Shanghai, China, May 27, 2006*. ACM, 2006 . – ISBN 1–59593–394–8

[WS00] Whittle, J. ; Schumann, J.: Generating Statechart Designs From Scenarios. In: *ICSE2000*. Limerick, Ireland, 2000

[ZL96] Zeller, Andreas ; Lütkehaus, Dorothea: DDD—a free graphical front-end for UNIX debuggers. In: *SIGPLAN Not.* 31 (1996), Nr. 1, S. 22–27. `http://dx.doi.org/http://doi.acm.org/10.1145/249094.249108`. – DOI http://doi.acm.org/10.1145/249094.249108. – ISSN 0362–1340

[ZLM+06] Zündorf, Albert ; Leohold, Jürgen ; Müller, Dieter ; Gemmerich, Ralf ; Reckord, Carsten ; Schneider, Christian ; Semmelrodt, Sven: Using object scenarios for requirements analysis - an experience report. In: *Modellierung* Bd. 82, GI, 2006 (LNI). – ISBN 3–88579–176–5, S. 269–278

Abbildungsverzeichnis

3.1. Schematische Darstellung des Fujaba Process 11

4.1. Beispielstatechart für den Statecharteditor 13

5.1. Usecasediagramm für den Statecharteditor 15
5.2. Objektspiel 18
5.3. Storyboard Teil 1 21
5.4. Storyboard Teil 2 22
5.5. Klassendiagramm in Fujaba 26
5.6. Storydiagramm für das Umleiten eingehender Transitionen 30
5.7. Ausführung des JUnit Test für die Statechartkonvertierung 32
5.8. Objektstruktur im eDOBS 34

6.1. Teilabschnitte der Codegenerierung mit eingehenden und resultierenden Daten 36
6.2. Modell mit zugehörigem Tokenbaum 36
6.3. Übersichtsklassendiagramm der Codegenerierung 37
6.4. Beispiel für die template-basierte Codegenerierung 38

12.1. Storyboard Execution Teil 1 56
12.2. Storyboard Execution Teil 2 57
12.3. Storydiagramm der JUnit `setUp()` Methode 59
12.4. Storydiagramm der Testmethode 60
12.5. Storydiagramm der Methode `assertStep1()` 60
12.6. Testmethode des Alternativszenarios 64
12.7. Storyboard des Alternativszenarios 65
12.8. Beispielmethode für die Testcodegenerierung 66
12.9. Klassendiagramm der Klassen `JavaSDM` und `JavaSDMException` . . . 68
12.10. Testausführung in Eclipse 72
12.11. Klassendiagramm des Coverage Mechanismus 76
12.12. Abdeckungsreport für die Beispiel-Tests 78
12.13. Storydiagramm der Methode `handleEvent()` 79

18.1. Klassendiagramm des Code-Modell Mapping Baums 102
18.2. Klassendiagramm der zusätzlichen `CodeWriter` 104
18.3. Mappingbaum 105
18.4. Fehlermeldung in Fujaba4Eclipse 108
18.5. Modellbasiertes Debuggen 110
18.6. eDOBS 112
18.7. Editieroperationen in eDOBS 114
18.8. Schnittstelle der Java Feature Abstraction (aus [Sch07]) 116
18.9. eDOBS mit Icons 117
18.10. Fabrikbeispiel im eDOBS 119
18.11. eDOBS mit Abstraktionen 119

18.12. Aus eDOBS Diagramm generierte Methode 121
18.13. Klebezettelmethode beim schrittweisen Durchspielen einer Methode 122
18.14. Klebezettel in eDOBS 123
18.15. Nebel des Vergessens 124
18.16. Nebel des Vergessens in eDOBS 125
18.17. eDOBS nach fehlgeschlagenem Test 127
18.18. Das CoObRA Framework 129
18.19. Klassendiagramm der Caching-Schicht 132
18.20. Klassendiagramm des *Cache Feature Abstraction Module* 134
18.21. Der *Flipbook View* 136
18.22. Flipbook nach fehlschlagendem Test 138
18.23. Debuggingsession nach Wiederaufsetzen aus dem Flipbook 140

19.1. Blinkerkabelbaum im eDOBS 144

Tabellenverzeichnis

12.1. Alternativszenario für den Usecase Execution 63

A.1. Vorlesungen Programmiermethodik 165

Listingsverzeichnis

12.1. Quelltext für die Beispieltestmethode 67
12.2. Quelltext für die Beispieltestmethode unter Anwendung der Testtemplates 69
12.3. Template bound.vm 70
12.4. Template sdmComment.vm 71
12.5. Template storyPattern.vm 71

14.1. OCL Ausdruck zum Prüfen der Endsituation 86

18.1. Beispielquelltext mit Kommentaren 104
18.2. SMAP Datei des Quelltexts aus Listing 18.1 107
18.3. Stacktrace einer CoObRA Änderungsoperation 131

A.1. Template toMany.vm 166