Institut für Informatik
Fachgebiet Softwaretechnik
Warburger Straße 100
33098 Paderborn

Struktur- und verhaltensbasierte Entwurfsmustererkennung

Schriftliche Arbeit
zur Erlangung des Grades
„Doktor der Naturwissenschaften“

vorgelegt von

Dipl.-Inform. Lothar Wendehals
Rosenstraße 42
44289 Dortmund

Paderborn, im September 2007

Zusammenfassung

Die Wartung von Softwaresystemen ist heute eine zeit- und damit kostenintensive Aufgabe. Die Systeme sind ständigen Änderungen unterworfen und über Jahre hinweg gewachsen. Die Dokumentation solcher Systeme wird kaum oder gar nicht gepflegt. Bei einer Größe von hunderttausenden oder sogar mehreren Millionen Zeilen Quelltext ist die schwerste Aufgabe des Softwareentwicklers, die bestehende Software zu verstehen, bevor Änderungen daran vorgenommen werden können. Die einzige verlässliche Grundlage für das Verstehen der Software bildet aber nur der Quelltext.

In der Softwareentwicklung werden weit verbreitete Lösungen für immer wiederkehrende Probleme als Entwurfsmuster bezeichnet. Sie sind vielfach dokumentiert und bilden ein gemeinsames Vokabular unter Entwicklern. Angewendete Entwurfsmuster, so genannte *Entwurfsmusterimplementierungen*, im Quelltext existierender Software zu identifizieren, hilft, das inhärente Design der Software explizit zu dokumentieren und so die Entwickler beim Verstehen der Software zu unterstützen.

In den letzten Jahren wurde eine Reihe von Werkzeugen entwickelt, die Entwurfsmusterimplementierungen (semi-)automatisch im Quelltext erkennen. Bis auf einige wenige Ausnahmen basieren alle Werkzeuge auf einer rein statischen Analyse des Quelltextes, ohne Eigenschaften der Software zur Laufzeit zu untersuchen. Diese Analysen sind gut dazu geeignet, strukturelle Eigenschaften der Entwurfsmuster zu erkennen. Allerdings werden Entwurfsmuster nicht nur durch ihre Struktur, sondern auch durch ihr Verhalten definiert. Verhalten kann jedoch durch statische Analysen nur sehr eingeschränkt untersucht werden. Die bisher entwickelten Werkzeuge erzeugen daher sehr unpräzise Ergebnisse.

Dynamische Analysen bieten eine Lösung für dieses Problem. Sie analysieren Software zur Laufzeit, indem sie ihr Verhalten beobachten. Ausschließlich dynamische Analysen sind jedoch kaum praktisch durchführbar, da zur Laufzeit riesige Datenmengen anfallen, die nur sehr schwer handhabbar sind.

Diese Arbeit stellt eine *struktur- und verhaltensbasierte Entwurfsmustererkennung* vor, die eine existierende, statische Entwurfsmustererkennung mit ei-

ner neu entwickelten, dynamischen Entwurfsmustererkennung kombiniert. Die statische Analyse identifiziert Kandidaten von Entwurfsmusterimplementierungen auf Basis struktureller Informationen. Der Anteil der dynamisch zu untersuchenden Software wird auf diese Kandidaten eingeschränkt, die anfallende Datenmenge wird reduziert und somit problemlos handhabbar. Zur Laufzeit wird dann das Verhalten der Kandidaten mit vorgegebenem Verhalten verglichen. Die durch die statische Analyse identifizierten und durch die dynamische Analyse bestätigten Entwurfsmusterimplementierungen stellen schließlich ein fundiertes und präzises Ergebnis dar.

Danksagung

Eine Arbeit wie diese entsteht niemals ohne Einflüsse von außen. Durch viele Diskussionen mit anderen Wissenschaftlern, Kollegen, Studenten und Freunden wurde sie zu dem, was nun vorliegt. Diesen Personen möchte ich meinen Dank aussprechen. Besonders hervorzuheben ist Wilhelm Schäfer, der die Arbeit wissenschaftlich betreut hat und dessen wissenschaftlicher Mitarbeiter ich fünf Jahre lang war. Aber auch Ekkart Kindler möchte ich danken für seine immer wieder hilfreiche und konstruktive Kritik.

Besonderer Dank gilt meinen Kollegen Matthias Meyer und Robert Wagner, mit denen ich eng zusammengearbeitet habe und die ich durch so manche fruchtbare Diskussion und themenfremde Unterhaltung von ihren eigenen Dissertationen abgehalten habe. Aber auch alle anderen Kollegen trugen mit ihrer Kritik zu dieser Arbeit bei, dies sind: Björn Axenath, Sven Burmester, Matthias Gehrke, Stefan Henkler, Martin Hirsch, Florian Klein, Jörg Niere, Vladimir Rubin, Daniela Schilling, Matthias Tichy, Dietrich Travkin und Jörg Wadsack.

Jörg Niere, Daniela Schilling und Martin Hirsch möchte ich dafür danken, dass sie mich jeweils eine Zeit lang als ihren Bürokollegen ausgehalten haben, bis ich schließlich mein eigenes Büro bekam. Vielen Dank auch an Jutta Haupt, die mir geholfen hat, die bürokratischen Klippen einer Universität zu umschiffen, und an Jürgen Maniera für die technische Unterstützung.

Danke auch an die vielen Studenten, die als studentische Hilfskräfte oder durch ihre Diplom- oder Studienarbeiten an der Umsetzung meiner Ideen mitgearbeitet haben.

Im Herbst 2005 verbrachte ich drei Monate am Georgia Instistute of Technology in Atlanta, USA. Ich sammelte in dieser Zeit viele Erfahrungen und konnte meine Ideen dort mal in einem vollkommen anderen Kontext betrachten. Vielen Dank an Mary Jean Harrold und an Alessandro Orso, dass sie mir diesen Aufenthalt ermöglicht haben.

Vor allem aber möchte ich meinen Eltern Gundi und Josef Wendehals danken, die mir meine Ausbildung erst ermöglicht haben, und meinen Geschwistern Jutta, Martin und Marion, die mich immer wieder während des Studiums und der Promotion unterstützt haben.

Inhalt

Kapitel 1

Einleitung

Das Jahr-2000-Problem verdeutlichte der breiten Öffentlichkeit, dass sich Software häufig nicht nur über Jahrzehnte hinweg im Einsatz befindet, sondern auch immer wieder während ihrer gesamten Lebensdauer gewartet werden muss. Die Ursache für das Jahr-2000-Problem reicht bis in die 1970er Jahre zurück. Jahreszahlen wurden in Algorithmen und Datenbanken nur durch ihre letzten beiden Ziffern repräsentiert. Man ging davon aus, dass die Software nicht bis zum Jahr 2000 im Einsatz sei, und interpretierte die ersten beiden fehlenden Ziffern grundsätzlich als „19“. So wäre beim Jahreswechsel vom Jahr 1999 auf das Jahr 2000 eine „00“ nicht als Jahr 2000, sondern als Jahr 1900 interpretiert worden. Vor der Jahrtausendwende mussten deshalb praktisch alle so genannten *Legacy*-Systeme – das ist Software, deren Algorithmen und Strukturen sowie zugehörige Datenbestände über Jahre hinweg gewachsen sind – auf dieses Problem hin überprüft und unter großem Aufwand korrigiert werden.

Das Jahr-2000-Problem ist zwar ein prominentes Beispiel, stellt aber bei weitem keinen Einzelfall dar. Softwaresysteme sind über Jahre oder sogar Jahrzehnte hinweg im Einsatz und müssen kontinuierlich an neue Anforderungen angepasst werden. Diese Systeme bestehen dabei nicht selten aus Software mit einem Umfang von mehreren Millionen Zeilen Quelltext. Des Weiteren wurden die Softwaresysteme üblicherweise von mehreren Generationen von Entwicklern geschaffen, die unter Umständen nicht mehr zur Weiterentwicklung des Softwaresystems zur Verfügung stehen.

Entwickler, die mit dem *Re-Engineering*, also dem Anpassen bereits bestehender Software an neue Anforderungen, beauftragt werden, stehen häufig unter enormen Kosten- und Zeitdruck. Die neue Version der Software soll schnell auf dem Markt verfügbar sein und möglichst früh produktiv eingesetzt werden. Änderungen an den Softwaresystemen werden daher weitestgehend nur

im Quelltext durchgeführt, wobei die Dokumentation der Änderungen vernachlässigt wird. Die vorhandene Dokumentation spiegelt daher meist nicht den aktuellen Stand der Softwaresysteme wieder.

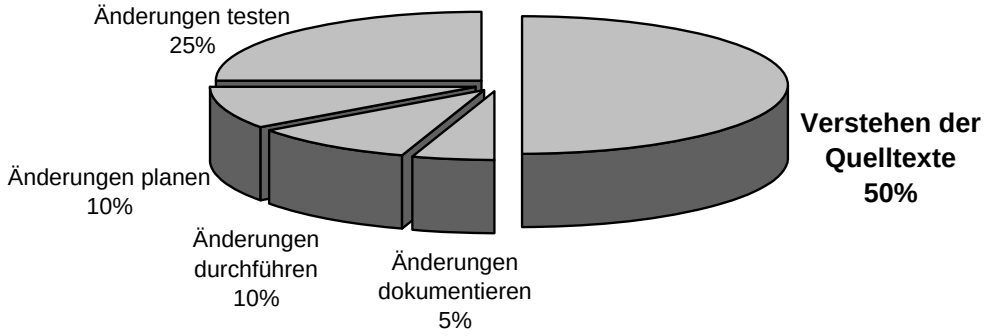

Abbildung 1.1: Kostenverteilung im Re-Engineering nach A. Frazer [Fra92]

Bevor die eigentlichen Änderungen durchgeführt werden können, müssen die Entwickler das Softwaresystem zunächst jedoch verstehen. Die Größe der Softwaresysteme, die Generationen von Entwicklern, die an dem Softwaresystem gearbeitet haben, und die ungenügende oder gar nicht vorhandene Dokumentation erschweren aber zusätzlich diese Aufgabe. Nach einer Studie aus dem Jahr 1992 von A. Frazer [Fra92] beträgt der Anteil der Kosten für das Verstehen des Quelltextes 50% an den Gesamtkosten für das Re-Engineering (Abbildung 1.1). Es ist sogar davon auszugehen, dass sich dieser Anteil seit 1992 eher noch erhöht hat, da die Softwaresysteme zunehmend komplexer geworden sind.

1.1 Reverse-Engineering

Eine Unterdisziplin des Re-Engineerings, das so genannte *Reverse-Engineering*, befasst sich mit der Analyse und dem Verständnis von Softwaresystemen [CC90]. Ziel des Reverse-Engineerings ist es, Softwaresysteme auf ihrem aktuellen Stand zu dokumentieren und zu verstehen. Es wird vor allem versucht, die Systeme auf abstrakterem Niveau zu dokumentieren, um die Kernpunkte der Softwaresysteme herauszustellen, unwesentliche Details auszublenden und so das Verständnis zu fördern.

Als Ausgangspunkt für das Reverse-Engineering können verschiedenste Quellen dienen. Zunächst sind dies die bereits vorhandenen Dokumente, die jedoch unter Umständen nicht mehr aktuell sind. Als weitere Quelle kommen

die früheren Entwickler in Frage, die aber nicht immer zur Verfügung stehen. So bleibt als einzige verlässliche Grundlage für Informationen über das Softwaresystem nur der Quelltext.

Das Verstehen des Quelltextes ist wie bereits oben erwähnt sehr zeit- und damit kostenintensiv. Eine Unterstützung des Menschen durch Werkzeuge, die vom Quelltext abstrahieren, ist somit von großem Vorteil. Zur Zeit sind bereits viele Werkzeuge zum Reverse-Engineering am Markt erhältlich oder werden in aktuellen Forschungsarbeiten entwickelt.

Zur Repräsentation der Softwaresysteme auf abstrakterem Niveau werden unterschiedlichste Arten von Visualisierungen verwendet. Sollen zum Beispiel Schwachstellen in Softwaresystemen gefunden und entfernt werden, sind *Metriken* hilfreich [FP96]. Sie drücken bestimmte Merkmale der Software in Zahlen aus. Aufgrund dieser Werte können dann statistische Ausreißer gefunden werden, die meist gute Hinweise auf Schwachstellen liefern. Will der Softwareentwickler dagegen Abhängigkeiten innerhalb eines Softwaresystems verstehen, ist zum Beispiel das Werkzeug Rigi nützlich [MWT95, Rig]. Es stellt unter anderem stark zusammenhängende Komponenten oder Verwendungsbeziehungen zwischen Klassen eines Softwaresystems in Graphen dar.

Eine sehr weit verbreitete Sprache zur Modellierung und Repräsentation von Software ist die *Unified Modeling Language* (UML) [Obj]. Mit der UML lassen sich unterschiedliche Sichten auf das Modell eines Softwaresystems realisieren. So stellen Paket- und Klassendiagramme statische Informationen wie Organisation und Beziehungen von Einzelteilen der Software dar, während zum Beispiel Aktivitäten- und Sequenzdiagramme dynamische Anteile wie das Verhalten der Software zur Laufzeit beschreiben. Ein großer Vorteil der UML ist, dass sie im kompletten Softwarelebenszyklus eingesetzt werden kann. Sie wird sowohl zum Forward-Engineering, also zur erstmaligen Erstellung der Modelle und der Software, als auch zum Re-Engineering, der Weiterentwicklung bestehender Softwaresysteme, verwendet. Die UML ist also eine ideale Grundlage zur Dokumentation der Ergebnisse des Reverse-Engineerings.

1.2 Entwurfsmustererkennung

> „*All well-structured software-intensive systems are full of patterns.*“
> – Grady Booch, November 2005, [Boo05]

Bei der Softwareentwicklung stoßen Software-Designer immer wieder auf gleichartige Probleme. Im Laufe der Jahre haben sich für diese Probleme

Lösungen herauskristallisiert, die sich unter den Entwicklern als so genannte *Entwurfsmuster* (engl. *Design Patterns*) etabliert haben. Seit Mitte der 1990er Jahre werden diese Entwurfsmuster verstärkt in Büchern und wissenschaftlichen Berichten festgehalten und dokumentiert. Ein Standardwerk ist das Buch von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides „Design Patterns—Elements of Reusable Object-Oriented Software“ [GHJV95] aus dem Jahre 1995, welches 23 weit verbreitete Entwurfsmuster vorstellt. Das Buch „The Pattern Almanac 2000“ [Ris00] wiederum enthält Referenzen auf hunderte verschiedener Entwurfsmuster aus den unterschiedlichsten Anwendungsgebieten.

Das oben genannte Zitat von Grady Booch – einer der ursprünglichen Entwickler der UML – bringt zum Ausdruck, dass sich Muster eigentlich in allen Softwaresystemen wiederfinden, sofern die Software nicht vollkommen chaotisch entwickelt worden ist. Entwurfsmuster wurden auch schon in der Softwareentwicklung eingesetzt, bevor sie in Büchern dokumentiert wurden. Durch ihre explizite Beschreibung wird lediglich ein gemeinsames Vokabular für die Entwickler geschaffen.

Ein Entwurfsmuster besteht in der Regel aus vier Teilen. Der *Name* des Entwurfsmusters bildet die Grundlage für das gemeinsame Vokabular und beschreibt in sehr wenigen Worten das Muster. Das *Problem* beschreibt, wann und in welchem Kontext das Entwurfsmuster angewendet wird. Die *Lösung* des Problems wird meist nicht im Detail, sondern eher auf einem abstrakteren Niveau vorgestellt, um die allgemeine Anwendbarkeit nicht einzuschränken. Dazu werden zum Beispiel UML-Diagramme verwendet, die zur Erläuterung der Struktur und des Verhaltens des Entwurfsmusters dienen. Die *Konsequenzen*, die die Verwendung des jeweiligen Entwurfsmusters impliziert, werden häufig anhand der Zeit- und Platzkomplexität oder der Erweiterbarkeit und der Wiederverwendbarkeit der Lösung diskutiert.

Werden Implementierungen der Entwurfsmuster, so genannte *Entwurfsmusterimplementierungen*, in bestehender Software identifiziert, so lässt sich nicht nur auf das zugrunde liegende Problem und den Kontext, in dem sie angewendet wurden, schließen, sondern auch abschätzen, wie beispielsweise Erweiterungen vorgenommen werden können. Die Identifizierung von Entwurfsmusterimplementierungen durch eine Entwurfsmustererkennung hilft also, das inhärente Design der Software explizit zu dokumentieren und die Entwickler bei ihrer Arbeit zu unterstützen. Eine manuelle Erkennung der Entwurfsmusterimplementierungen ist jedoch aus den bereits genannten Gründen Zeit und Kosten kaum durchführbar. Deshalb werden immer mehr Reverse-Engineering-Werkzeuge entwickelt, die (semi-)automatisch nach Entwurfsmusterimplemen-

tierungen im Quelltext suchen. Zur Dokumentation der Ergebnisse werden zum Beispiel die im Reverse-Engineering entstehenden UML-Dokumente durch die Kennzeichnung von Entwurfsmusterimplementierungen angereichert.

Die informelle Beschreibung der Entwurfsmuster bietet dem Entwickler während des Forward-Engineerings weitreichende Freiheiten bei ihrer Implementierung. Zum einen lässt sich das gleiche Verhalten auf unterschiedliche Weise in Programmiersprachen umsetzen. Eine Schleife kann zum Beispiel als `for`- oder als `while`-Schleife implementiert werden. Zum anderen lassen sich Lösungselemente auf höherem Abstraktionsniveau – wie zum Beispiel Assoziationen – auf unterschiedlichste Weise realisieren. Dadurch entstehen praktisch unendlich viele Implementierungsvarianten eines einzelnen Entwurfsmusters.

Für eine automatische Erkennung der Entwurfsmuster ist die Vielfalt der Implementierungsvarianten eines der größten Probleme. Diesem Problem kann man im Wesentlichen durch zwei Strategien begegnen. Auf der einen Seite können unterschiedliche Implementierungsvarianten durch unterschiedliche Regeln erkannt werden. Dies führt jedoch zu einer theoretisch beliebig großen Zahl an Regeln. Selbst eine Beschränkung auf einige Implementierungsvarianten für viele verschiedene Entwurfsmuster erzeugt eine zu große Menge Regeln, die zu einem Laufzeitproblem bei der Erkennung führt. Beschränkt man auf der anderen Seite aber die Zahl der Regeln, können nur wenige Implementierungsvarianten erkannt werden. Das führt schließlich zu einer ungenauen Erkennung, bei der viele existierende Entwurfsmusterimplementierungen nicht identifiziert werden.

Durch unscharfe Regeln lässt sich diese Situation etwas verbessern. Eine Regel deckt dabei verschiedene Implementierungsvarianten eines Entwurfsmusters ab. Als Nebeneffekt werden jedoch mehr so genannter *False-Positives* erkannt. False-Positives sind Konstrukte, die zwar als Entwurfsmusterimplementierungen identifiziert wurden, aber keine sind. Die Reduzierung des Anteils der False-Positives am Gesamtergebnis der Entwurfsmustererkennung ist ein entscheidender Faktor für die Präzision der Erkennung. False-Positives müssen vom Reverse-Engineer manuell als solche identifiziert werden und stellen damit einen Mehraufwand dar, der möglichst gering gehalten werden sollte.

1.3 Statische und Dynamische Analyse

In den letzten Jahren wurde eine Reihe von Werkzeugen entwickelt, die Entwurfsmusterimplementierungen (semi-)automatisch im Quelltext erkennen. Bis auf einige wenige Ausnahmen basieren alle Werkzeuge auf einer rein stati-

schen Analyse[1], bei der der Quelltext untersucht wird, ohne die Software auszuführen. Diese Analysen sind sehr gut dazu geeignet, strukturelle Eigenschaften der Entwurfsmuster zu erkennen.

Entwurfsmuster werden allerdings nicht nur durch ihre Struktur, sondern auch durch ihr Verhalten definiert. Konstrukte, die zwar in ihrer Struktur mit Entwurfsmustern übereinstimmen und als Implementierungen solcher erkannt werden, sich aber zur Laufzeit anders verhalten, sind mit sehr großer Wahrscheinlichkeit False-Positives. Durch rein statische Analysen kann das Verhalten aber nur sehr eingeschränkt erkannt werden.

Das Verhalten von Software wird bei imperativen Programmiersprachen im Wesentlichen durch Sequenzen von Prozedur- beziehungsweise Methodenaufrufen bestimmt. Statische Analysen erkennen zwar potentielle Methodenaufrufe, ob sie jedoch tatsächlich zur Laufzeit ausgeführt werden, ist nicht sicher festzustellen. Objektorientierte Programmiersprachen mit Polymorphie und dynamischer Methodenbindung verschärfen dieses Problem sogar noch, da die konkreten, aufzurufenden Methoden erst zur Laufzeit festgelegt werden. Die Ermittlung konkreter Sequenzen von Methodenaufrufen durch statische Analysen ist daher sehr ungenau und für die präzise Erkennung von Entwurfsmusterimplementierungen nicht geeignet.

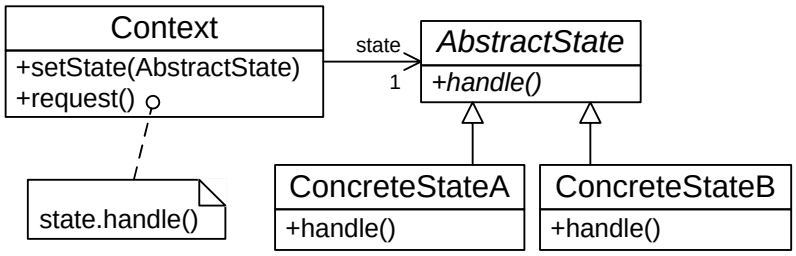

Abbildung 1.2: Die Struktur des *State*-Entwurfsmusters

In [GHJV95] werden einige Entwurfsmuster vorgestellt, die paarweise große Ähnlichkeiten in ihrer Struktur aufweisen. Zu diesen Paaren gehören zum Beispiel *Decorator* und *Chain of Responsibility*, *Strategy* und *Bridge* oder auch *State* und *Strategy*. Wie in den Abbildungen 1.2 und 1.3 zu sehen, sind die beiden Entwurfsmuster *State* und *Strategy* in ihrer Struktur sogar vollkommen identisch. Sie unterscheiden sich ausschließlich durch ihr Verhalten. Bei

[1] siehe: [KP96, AFC98, SK98, SG98, Wuy98, KSRP99, TA99, BP00, KB00, ACGJ01, AG01, SS03, PSRN04, TCHS05, KGH06, SO06]

solchen Entwurfsmustern führen also strukturelle Ähnlichkeiten bei der statischen Analyse zu nicht eindeutigen Ergebnissen.

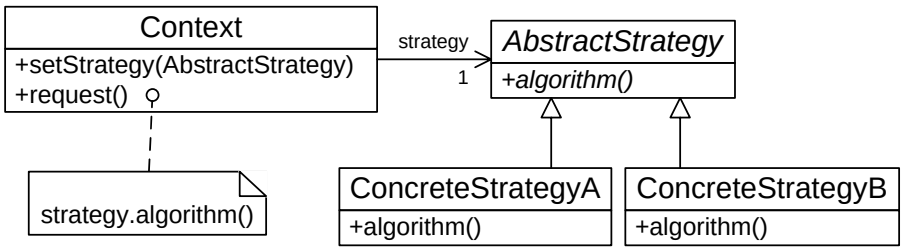

Abbildung 1.3: Die Struktur des *Strategy*-Entwurfsmusters

Dynamische Analysen bieten eine Lösung sowohl für die Ermittlung möglicher Sequenzen von Methodenaufrufen, als auch zur Unterscheidung von Entwurfsmustern gleicher Struktur. Sie analysieren Software zur Laufzeit, indem sie ihr Verhalten beobachten. Dazu wird zum Beispiel durch Instrumentierung [SO05] der Programmcode durch zusätzliche Anweisungen angereichert, die Methodenaufrufe protokollieren. Dadurch wird es in der automatischen Entwurfsmustererkennung möglich, vorgegebenes Verhalten von Entwurfsmustern mit tatsächlich beobachtetem Verhalten von potentiellen Entwurfsmusterimplementierungen zu vergleichen. Die Datenmengen, die bei solchen Analysen anfallen, sind allerdings relativ groß und dadurch nur sehr schwer handhabbar.

Ein weiteres Problem dynamischer Analysen ist die angemessene Auswahl der Eingabedaten, die zur Ausführung der zu analysierenden Software benötigt werden. Das in der Praxis durch dynamische Analysen beobachtete Verhalten eines Softwaresystems stellt immer nur einen kleinen Teil des theoretisch möglichen Verhaltens dar. Um also verwertbare Ergebnisse zu erhalten, sollten die Eingabedaten möglichst repräsentativ für die in der Praxis auftretenden Daten ausgewählt werden. Die Ausführung der zu untersuchenden Software erfolgt dann durch automatische Tests oder durch manuelle Bedienung.

Statische und dynamische Analysen wurden lange Zeit in unterschiedlichen, voneinander unabhängigen Forschungsgebieten entwickelt. Deshalb wurden meist ausschließlich entweder statische oder dynamische Analysen genutzt. Die jeweils andere Analysetechnik wurde sogar häufig für den aktuellen Anwendungsbereich als unpassend dargestellt. Michael Ernst geht in seinem Artikel *„Static and dynamic analysis: synergy and duality“* [Ern03] auf diese Problematik ein und plädiert dafür, die Vorteile beider Analysetechniken zu kombinieren, um so bessere Analyseergebnisse zu produzieren.

1.4 Ergebnisse der Arbeit

Die vorliegende Arbeit stellt eine struktur- und verhaltensbasierte Entwurfsmustererkennung vor, bei der eine bereits existierende, statische Entwurfsmustererkennung um eine dynamische Entwurfsmustererkennung ergänzt wurde. Die statische Analyse untersucht das Softwaresystem auf strukturelle Eigenschaften von Entwurfsmustern. Das Ergebnis dieses Analyseschrittes sind potentielle Entwurfsmusterimplementierungen, so genannte *Kandidaten*, die in UML-Klassendiagrammen des zu untersuchenden Softwaresystems dokumentiert werden. Die strukturbasierte Entwurfsmustererkennung wurde in dieser Arbeit um einen Algorithmus zur Bewertung der Kandidaten erweitert. Die Bewertung eines Kandidaten gibt an, inwieweit der Kandidat mit der vorgegebenen Struktur des Entwurfsmusters übereinstimmt.

Des Weiteren dienen die Kandidaten als Eingabedaten der anschließenden dynamischen Analyse. Die Kandidaten schränken den Suchraum der dynamischen Analyse erheblich ein und reduzieren so die zur Laufzeit anfallende und zu analysierende Datenmenge. Zur Spezifikation des Verhaltens der Entwurfsmuster wurde eine Sprache syntaktisch und semantisch formal definiert, die auf UML-Sequenzdiagramme aufbaut. Aus den Spezifikationen werden Automaten für die dynamische Analyse generiert, die zur Laufzeit beobachtete Sequenzen von Methodenaufrufen mit dem spezifizierten Verhalten der Entwurfsmuster vergleichen.

Die verhaltensbasierte Entwurfsmustererkennung wird in der Praxis mit Hilfe der *Software-Tomographie* [BOH02] durchgeführt. Dabei wird das zu untersuchende Softwaresystem während der dynamischen Analyse in realen, produktiven Umgebungen eingesetzt, so dass das Problem der repräsentativen Eingabedaten gelöst ist. Zur dynamischen Analyse wird das System instrumentiert, das heißt, es wird zusätzlicher Code in die Software eingeführt, der Methodenaufrufe überwacht und protokolliert. Um die Performanz des Softwaresystems so wenig wie möglich zu beeinträchtigen, wird die dynamische Analyse in viele unabhängige Teilanalysen aufgeteilt. Mehrere Instanzen des Softwaresystems werden dann jeweils für eine oder einige wenige Teilanalysen instrumentiert und in einer produktiven Umgebung eingesetzt. So werden repräsentative Daten über die Software gesammelt und der dynamischen Analyse zugeführt.

Als Ergebnis der dynamischen Analyse erhält der Reverse-Engineer Sequenzen von Methodenaufrufen zu den Kandidaten. Die Sequenzen werden von der dynamischen Analyse als konform beziehungsweise nicht-konform zum vorgegeben Verhalten gekennzeichnet. So kann der Reverse-Engineer nicht nur er-

kennen, ob der Kandidat sich wie ein Entwurfsmuster verhält, sondern auch feststellen, warum der Kandidat eventuell gegen das vorgegebene Verhalten verstößt. Das hilft beim Erkennen von Fehlern oder Design-Defekten und gibt Hinweise auf mögliche Korrekturen.

Das in dieser Arbeit entwickelte Verfahren wurde prototypisch in einem Werkzeug umgesetzt und in die Entwicklungsumgebung Eclipse integriert. Außerdem wurde das Verfahren auf ein reales Softwaresystem angewendet, zu dem Entwurfsmusterimplementierungen dokumentiert sind. Die Ergebnisse dieser Anwendung konnten so mit der Dokumentation verglichen und das Verfahren beurteilt werden.

1.5 Aufbau der Arbeit

Die vorliegende Arbeit wird im zweiten Kapitel mit einigen Grundlagen, die zum Verständnis der struktur- und verhaltensbasierten Entwurfsmustererkennung notwendig sind, fortgeführt. Zu den Grundlagen gehört unter anderem die bereits existierende, strukturbasierte Entwurfsmustererkennung, auf der das in dieser Arbeit entwickelte Verfahren aufbaut.

Um darauf aufbauen zu können, werden im dritten Kapitel zunächst einige Erweiterungen an der strukturbasierten Entwurfsmustererkennung behandelt. Des Weiteren gibt das Kapitel einen Überblick über den Prozess der kombinierten, struktur- und verhaltensbasierten Entwurfsmustererkennung und führt ein Verhaltensmodell für Softwaresysteme ein.

Das Thema des vierten Kapitels ist die formale Spezifikation des Verhaltens eines Entwurfsmusters durch Verhaltensmuster. Es wird eine Spezifikationssprache vorgestellt, die an UML-Sequenzdiagramme angelehnt ist und die sowohl syntaktisch als auch semantisch formal definiert wird.

Im fünften Kapitel wird die Verhaltensanalyse behandelt. Zunächst wird diskutiert, wie Sequenzen von Methodenaufrufen zur Laufzeit des zu untersuchenden Softwaresystems gewonnen werden können. Zur Erkennung von Verhaltensmustern in diesen Sequenzen werden Automaten verwendet, die aus den formalen Spezifikationen der Verhaltensmuster automatisch generiert werden.

Die praktische Anwendung des Verfahrens wird im sechsten Kapitel vorgestellt. Es wird ein Szenario diskutiert, wie die verhaltensbasierte Entwurfsmustererkennung in produktiven Umgebungen eingesetzt werden kann, um eine praxisnahe Datenbasis zur Analyse zu erhalten. Außerdem werden einige Ergebnisse des Einsatzes der Entwurfsmustererkennung auf ein reales, umfangreiches Softwaresystem vorgestellt.

Im Rahmen dieser Arbeit ist ein prototypisches Werkzeug entstanden, das das Verfahren der struktur- und verhaltensbasierten Entwurfsmustererkennung umsetzt. Das Werkzeug wurde in die in der Industrie weit verbreitete Entwicklungsumgebung ECLIPSE integriert. Die Architektur und die Benutzungsschnittstelle dieses Werkzeugs werden im siebten Kapitel erläutert.

Das achte Kapitel behandelt verwandte Arbeiten. Darin werden Arbeiten anderer Wissenschaftler diskutiert, die einen starken Bezug zu der vorliegenden Arbeit haben.

Die Arbeit schließt im neunten Kapitel mit einer Zusammenfassung und einem Ausblick, in dem auf mögliche Erweiterungen und weitere Anwendungen der in dieser Arbeit entwickelten Techniken hingewiesen wird.

Im Anhang der Arbeit sind die in der Evaluation verwendeten Struktur- und Verhaltensmuster zu finden. Des Weiteren enthält der Anhang ein Handbuch und die technische Dokumentation des entwickelten Werkzeugs.

Kapitel 2

Grundlagen

Das folgende Kapitel gliedert sich in drei Teile. Der erste Teil behandelt notwendige Grundlagen zu Entwurfsmustern. Im zweiten Teil werden allgemeine Anforderungen an eine automatische Entwurfsmustererkennung formuliert.

Den Schwerpunkt des Kapitels bildet der dritte Teil, in dem die strukturbasierte Entwurfsmustererkennung, die am Fachgebiet Softwaretechnik der Universität Paderborn entwickelt wurde [Pal01, Wen01, Nie04], im Detail erläutert wird. Das in der vorliegenden Arbeit vorgestellte Verfahren verwendet diese Technik zur strukturbasierten Entwurfsmustererkennung, um sie durch eine verhaltensbasierte Erkennung zu vervollständigen.

2.1 Entwurfsmuster

Software-Designer stoßen in ihrer täglichen Arbeit immer wieder auf gleichartige Probleme beim Design und bei der Implementierung großer Softwaresysteme. Im Laufe der Zeit kristallisierten sich bewährte Muster zur Lösung dieser Probleme heraus. Diese Muster geben keine konkrete Lösung für ein Problem, sondern nur eine Lösungsidee vor. Die Implementierung der Lösungsidee erfolgt dann angepasst an die jeweilige Situation. Solche Muster werden in der Softwaretechnik *Entwurfsmuster* (engl. *Design Patterns*) genannt.

Entwurfsmuster werden zwar schon lange eingesetzt, wie zum Beispiel das *Model/View/Controller*-Muster in der Smalltalk-Programmierung, wurden anfangs allerdings kaum dokumentiert. Des Weiteren existierte keine allgemeine Vorgehensweise zur Dokumentation von Entwurfsmustern. Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides veröffentlichten im Jahre 1995 ihr Buch „Design Patterns—Elements of Reusable Object-Oriented Software“ [GHJV95], in dem sie 23, in der objektorientierten Programmierung weit verbreitete Entwurfsmuster vorstellen.

Seit dem Erscheinen dieses Buches sind weitere Sammlungen von Entwurfsmustern veröffentlicht worden. Diese Sammlungen sind meist anwendungsspezifisch, wie zum Beispiel die Entwurfsmuster aus [Lea97] zur Programmierung nebenläufiger Systeme. Das Buch „The Pattern Almanac 2000“ [Ris00] dient als Nachschlagewerk mit Referenzen auf hunderte verschiedener Entwurfsmuster aus den unterschiedlichsten Anwendungsgebieten.

Das Buch von Gamma et al. stellt heute praktisch einen Quasi-Standard zur Dokumentation von Entwurfsmustern dar. In dem Buch werden Entwurfsmuster nach zwei Kriterien klassifiziert: ihrem *Zweck* und ihrem *Anwendungsbereich*. Der Zweck wird in drei Kategorien aufgeteilt: Entwurfsmuster, die zum Erzeugen von Objekten dienen, die eine Struktur beschreiben oder die Verhalten beschreiben. Der Anwendungsbereich wird auf Klassen und Objekte aufgeteilt. Die Tabelle 2.1 ist [GHJV95] entnommen und enthält alle darin vorgestellten Entwurfsmuster. Die Einordnung eines Entwurfsmusters in eine Kategorie erfolgt nach der Hauptintention des Musters. Das Entwurfsmuster *State* zum Beispiel beschreibt im Wesentlichen das Verhalten einer Gruppe von Objekten. Zur Implementierung wird jedoch auch eine Struktur der an dem Muster beteiligten Klassen vorgeschlagen.

		Zweck		
		Erzeugend	**Struktur**	**Verhalten**
Anwendungs-bereich	**Klasse**	Factory Method	Adapter	Interpreter Template Method
	Objekt	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Tabelle 2.1: Kategorisierung der Entwurfsmuster nach Gamma et al. [GHJV95]

Ein Entwurfsmuster nach [GHJV95] besteht aus vier Teilen: Der *Name* des Entwurfsmusters bildet die Grundlage für ein gemeinsames Vokabular unter den Softwareentwicklern und beschreibt in knappen Worten das Entwurfsmu-

ster. Das *Problem* beschreibt den Kontext und mögliche Voraussetzungen zur Anwendung des Entwurfsmusters. In der *Lösung* werden die Struktur sowie Beziehungen und Verhalten der beteiligten Elemente des Entwurfsmusters vorgestellt. Im letzten Teil werden die *Konsequenzen* genannt, die die Anwendung des Entwurfsmusters impliziert. Dazu gehören unter anderem die Erweiterbarkeit und Flexibilität, aber auch die Platz- und Zeitkomplexität des Entwurfsmusters.

Die Dokumentation der Lösung wird unterteilt in Anwendbarkeit, Struktur, teilnehmende Klassen und Objekte, Verhalten sowie Implementierung und beispielhafte Quelltextfragmente. Die Beschreibung dieser Punkte ist nicht formal, sondern erfolgt zu einem großen Teil mit Hilfe einfacher Texte. Die Struktur eines Entwurfsmusters wird in [GHJV95] zwar durch OMT-Diagramme beschrieben, die Diagramme sind jedoch eher als Vorschlag zum Entwurf einer Entwurfsmusterimplementierung zu verstehen. In neuerer Literatur werden anstatt der OMT-Diagramme üblicherweise UML-Klassendiagramme verwendet. Das Verhalten wird in einigen Fällen durch Kollaborations- oder Interaktionsdiagramme, meist aber ebenfalls nur durch Text beschrieben.

Die informelle Art der Dokumentation von Entwurfsmustern ist ideal für das Forward-Engineering. Der Softwareentwickler hat weit gehende Freiheiten, die Implementierung des Entwurfsmusters an die gegebene Situation anzupassen. Zum einen lässt sich das gleiche Verhalten auf unterschiedliche Weise implementieren. Eine Wiederholung kann zum Beispiel als `for`- oder als `while`-Schleife programmiert werden. Zum anderen lassen sich Lösungselemente auf höherem Abstraktionsniveau, wie zum Beipsiel Assoziationen, unterschiedlich realisieren. Dadurch entstehen praktisch unendlich viele Implementierungsvarianten eines einzelnen Entwurfsmusters.

2.2 Automatische Entwurfsmustererkennung

Die Identifizierung von Entwurfsmusterimplementierungen durch eine Entwurfsmustererkennung hilft, das inhärente Design der Software explizit zu dokumentieren und die Entwickler bei ihrer Arbeit zu unterstützen. Entwurfsmusterimplementierungen lassen nicht nur auf das zugrunde liegende Problem und den Kontext, in dem sie angewendet wurden, schließen. Das Wissen um Entwurfsmusterimplementierungen hilft auch dabei, Erweiterungen an den Softwaresystemen vorzunehmen. Eine manuelle Erkennung der Entwurfsmusterimplementierungen ist in sehr kleinen Softwaresystemen noch möglich, in praxisnahen, meist sehr umfangreichen Softwaresystemen jedoch kaum durchführbar.

Eine automatische Entwurfsmustererkennung, die solch umfangreiche Softwaresysteme effizient handhaben kann, ist daher sehr wünschenswert.

Die informelle Beschreibung der Entwurfsmuster führt zu einer Vielfalt von Implementierungsvarianten, die für eine automatische Entwurfsmustererkennung das größte Problem darstellt. Die unterschiedlichen Implementierungsvarianten können zum Beispiel durch unterschiedliche Regeln erkannt werden. Dies führt jedoch zu einer theoretisch beliebig großen Zahl an Regeln, die die Dauer der Entwurfsmustererkennung drastisch steigert. Beschränkt man aber die Zahl der Regeln, können nur wenige Implementierungsvarianten erkannt werden. Das führt schließlich zu einer ungenauen Erkennung, bei der viele existierende Entwurfsmusterimplementierungen nicht identifiziert werden.

Durch unscharfe Regeln lässt sich diese Situation etwas verbessern. Eine Regel deckt dabei verschiedene Implementierungsvarianten eines Entwurfsmusters ab. Als Nebeneffekt werden jedoch mehr *False-Positives* erkannt. Das sind Konstrukte, die zwar als Entwurfsmusterimplementierungen identifiziert wurden, aber keine sind. Die Reduzierung des Anteils der False-Positives am Gesamtergebnis der Entwurfsmustererkennung ist ein entscheidender Faktor für die Präzision der Erkennung. False-Positives müssen vom Reverse-Engineer manuell als solche identifiziert werden und stellen damit einen Mehraufwand dar, der möglichst gering gehalten werden sollte. Hinweise auf die Güte der erkannten Entwurfsmusterimplementierungen können die manuelle Identifikation jedoch erleichtern.

2.2.1 Anforderungen an eine Entwurfsmustererkennung

Aus diesen Überlegungen lassen sich vier allgemeine Anforderungen an eine automatische Entwurfsmustererkennung ableiten.

Skalierbarkeit

Softwaresysteme bestehen nicht selten aus mehreren hunderttausend bis Millionen Zeilen Quelltext. Erst diese Größe macht eine automatische Entwurfsmustererkennung notwendig; kleine Systeme aus nur wenigen tausend Zeilen sind noch manuell analysierbar. Die Skalierbarkeit des Analysealgorithmus ist daher das wichtigste Kriterium einer automatischen Entwurfsmustererkennung. Die automatische Analyse großer Softwaresysteme muss mit einem zeitlich vertretbaren Aufwand möglich sein.

Präzision

Eine automatische Entwurfsmustererkennung ist nur dann für einen Reverse-Engineer sinnvoll einsetzbar, wenn die Ergebnisse der Analyse möglichst präzise sind. Das bedeutet zum einen, dass möglichst alle in einem Softwaresystem vorhandenen Entwurfsmusterimplementierungen von der automatischen Erkennung identifiziert werden sollten. Zum anderen sollten fälschlicherweise erkannte Entwurfsmusterimplementierungen vermieden werden. Eine hundertprozentig korrekte Erkennung ist allerdings nicht zu erreichen.

Anpassbarkeit

Softwaresysteme werden unter den verschiedensten Bedingungen hergestellt. Häufig gibt es spezifische Richtlinien bei den Herstellern, wie bestimmte Architekturdetails oder Eigenschaften der Software umgesetzt werden müssen. Dazu gehören zum Beispiel Richtlinien, ob und wie Zugriffsmethoden für Attribute einer Klasse verwendet werden, wie Assoziationen zwischen Klassen implementiert werden oder nach welchem Schema Klassen und Methoden benannt werden. Aber auch jeder Entwickler hat einen persönlichen Programmierstil, nach dem er gleiche Aufgaben immer wieder auf sehr ähnliche Art und Weise umsetzt. Solche Informationen können zu einer präziseren Erkennung von Entwurfsmusterimplementierungen beitragen. Die Spezifikation eines Entwurfsmusters für die automatische Erkennung sollte deshalb leicht vom Reverse-Engineer an solche Richtlinien und Programmierstile angepasst werden können.

Bewertung

Wegen der sehr vielen Implementierungsvarianten eines Entwurfsmusters ist eine hundertprozentig sichere Aussage, ein bestimmtes Konstrukt sei eine Entwurfsmusterimplementierung, durch eine automatische Entwurfsmustererkennung nicht möglich. False-Positives können nicht immer vermieden werden. Bestimmte Details eines Konstrukts können auf eine tatsächliche Entwurfsmusterimplementierung oder auch ein False-Positive hindeuten, während andere Details wiederum das Gegenteil nahe legen. Wird zum Beispiel beim *State*-Entwurfsmuster eine mehrwertige Referenz anstatt einer einfachen Referenz zwischen der Kontext-Klasse und der abstrakten Zustandsklasse verwendet, so deutet dies zunächst einmal nicht auf eine *State*-Entwurfsmusterimplementierung hin. Es kann allerdings sein, dass trotzdem zur Laufzeit immer nur auf einen konkreten Zustand verwiesen wird. Um den

Reverse-Engineer bei der Sichtung und Einschätzung der Ergebnisse zu unterstützen, ist es deshalb sinnvoll, die Güte der gefundenen Entwurfsmusterimplementierungen zu bewerten.

2.3 Strukturbasierte Entwurfsmustererkennung in Fujaba

Seit dem Jahr 1998 wird an dem Fachgebiet Softwaretechnik an der Universität Paderborn das CASE[1]-Werkzeug Fujaba entwickelt [FNT98]. Fujaba ist ein Akronym und steht für „*From UML to Java And Back Again*“. Ziel des Projekts war es, ein so genanntes *Round-Trip-Engineering*, also die Verschmelzung von Forward- und Reverse-Engineering, zu ermöglichen. Das bedeutet, Änderungen am Modell werden in den bereits vorhandenen Quelltext eines Softwaresystems übernommen und Änderungen am Quelltext werden in ein bereits vorhandenes Modell übertragen. Dadurch ist gleichzeitiges Arbeiten am Modell und am Quelltext eines Softwaresystems möglich.

Mittlerweile ist Fujaba zu einer allgemeinen, modellbasierten Entwicklungsplattform ausgebaut worden, die durch Plug-Ins beliebig erweitert werden kann. Es existieren zur Zeit verschiedene Erweiterungen für Fujaba, zum Beispiel zur Entwicklung mechatronischer Echtzeitsysteme, zur Meta-Modellierung und Modelltransformation oder auch zum Re-Engineering [Fuj].

Im Zuge der Entwicklung der Re-Engineering-Techniken von Fujaba wurde ein erster Ansatz zur Entwurfsmustererkennung 1998/1999 in einer studentischen Projektgruppe erarbeitet. Die Erkennung wurde durch manuell programmierte Algorithmen realisiert, die auf dem abstrakten Syntaxgraphen (ASG) des zu untersuchenden Softwaresystems strukturelle Analysen durchführten. Im Jahre 2001 wurde in einer Diplomarbeit eine formale Sprache auf Basis von Graphgrammatiken zur Beschreibung von Erkennungsregeln für Entwurfsmusterimplementierungen spezifiziert, aus der automatisch Erkennungsmaschinen generiert werden können [Pal01]. In einer weiteren Diplomarbeit wurde außerdem ein Algorithmus zur Anwendung dieser Erkennungsmaschinen entwickelt, der eine inkrementelle Analyse großer Softwareysteme ermöglicht [Wen01]. Die Spezifikationssprache und der inkrementelle Erkennungsalgorithmus werden in [NSW+02] vorgestellt. Um der Variantenvielfalt der Entwurfsmusterimplementierungen zu begegnen, wird in [NWW03] die Verwendung von Unschärfe in der Beschreibung von Erkennungsregeln vorgeschlagen und eine Implementie-

[1] Computer Aided Software Engineering

rung sowie Evaluation vorgestellt. Der gesamte Prozess der strukturbasierten Entwurfsmustererkennung wird im Detail in der Dissertation von Jörg Niere [Nie04] behandelt.

Im Folgenden wird ein Überblick über die strukturbasierte Entwurfsmustererkennung in Fujaba gegeben. Darin werden die wichtigsten Grundlagen, auf denen diese Arbeit aufbaut, vorgestellt. Zunächst wird erläutert, wie der Quelltext eines Softwaresystems als Vorbereitung zur strukturbasierten Entwurfsmustererkennung repräsentiert wird. Anschließend wird die Spezifikationssprache zur Beschreibung der Erkennungsregeln für Entwurfsmusterimplementierungen vorgestellt. Es folgt die Beschreibung des so genannten *Regelkatalogs*, in dem die Abhängigkeiten zwischen den Erkennungsregeln festgehalten werden und der vom Erkennungsalgorithmus verwendet wird. Der Erkennungsprozess und die Bewertung der gefundenen, potentiellen Entwurfsmusterimplementierungen werden am Ende des Kapitels beschrieben.

2.3.1 Strukturmodell eines Softwaresystems

Die Struktur eines Softwaresystems ist spezifiziert durch seinen Quelltext. Der Quelltext ist jedoch zur algorithmischen Analyse meist ungeeignet. Deshalb wird ein anderes Modell für die Struktur eines Softwaresystems benötigt. Im Folgenden wird dieses Modell *Strukturmodell* genannt.

Um Modelle zu beschreiben, werden Modellierungssprachen eingesetzt, deren Syntax durch *Metamodelle* definiert wird. In der Softwaretechnik hat sich die UML [Obj] als Modellierungssprache etabliert. Strukturmodelle werden durch UML-Klassendiagramme beschrieben. Das Metamodell für UML-Klassendiagramme ist daher als Metamodell für Strukturmodelle nahe liegend.

Abbildung 2.1 stellt das in dieser Arbeit verwendete Metamodell für Strukturmodelle dar, das ein vereinfachtes Metamodell für UML-Klassendiagramme ist und an das originale UML-Metamodell angelehnt ist. Dieses Metamodell definiert die Syntax für Strukturmodelle. Zur besseren Übersicht werden alle Metamodelle und Modelle, die im Folgenden eingeführt werden, Paketen zugeordnet, die später referenziert werden. Das Metamodell der Strukturmodelle gehört zum Paket ClassDiagrams.

Die Struktur eines objektorientierten Softwaresystems besteht im Wesentlichen aus Klassen und Vererbungen zwischen Klassen. Des Weiteren besitzen Klassen Attribute und Methoden, die aus einer Methodensignatur und dem Methodenrumpf zusammengesetzt sind. Es bietet sich also an, als Strukturmodell für objektorientierte Softwaresysteme ein Modell aus Klassen, Attributen, Methoden und Vererbungen zwischen Klassen zu verwenden.

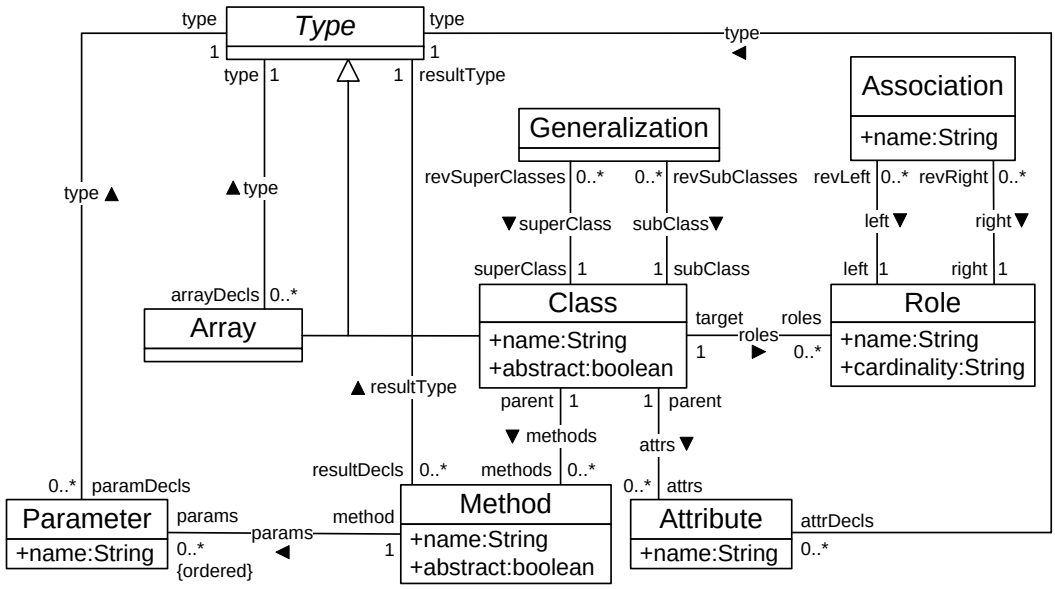

Abbildung 2.1: Das Metamodell für Strukturmodelle, Paket ClassDiagrams

In Abbildung 2.2 ist ein Ausschnitt aus dem Strukturmodell für Quelltexte objektorientierter Sprachen zu sehen. Es ist in großen Teilen identisch mit dem Metamodell für Strukturmodelle aus Abbildung 2.1. Das ist insofern logisch, da hier ein Modell für Klassendiagramme mit dem Metamodell für Klassendiagramme definiert wird.

Es sind jedoch auch Unterschiede zwischen dem Metamodell für Strukturmodelle und dem Strukturmodell für objektorientierte Softwaresysteme festzustellen. Zum einen kann das Strukturmodell keine Assoziationen zwischen Klassen repräsentieren. Heutige, objektorientierte Programmiersprachen enthalten keine Sprachelemente für Assoziationen. Assoziationen können also nicht direkt aus dem Quelltext eines Softwaresystems gewonnen werden. Assoziationen stellen vielmehr bereits eine abstraktere Sicht auf das Softwaresystem dar, die durch explizite Analysen auf der Struktur – zum Beispiel durch die Entwurfsmustererkennung – hergestellt werden muss.

Zum anderen müssen die Methodenrümpfe im Strukturmodell repräsentiert werden. Methodenrümpfe werden in abstrakte Syntaxbäume (engl. *Abstract Syntax Tree* (AST)) transformiert. Da das Modell des abstrakten Syntaxbaums für Methodenrümpfe sehr komplex ist und nicht weiter in dieser Arbeit benötigt wird, ist es im Strukturmodell in Abbildung 2.2 mit der Klasse ASTRootNode nur ausschnittsweise dargestellt. Die Klasse ASTRootNode

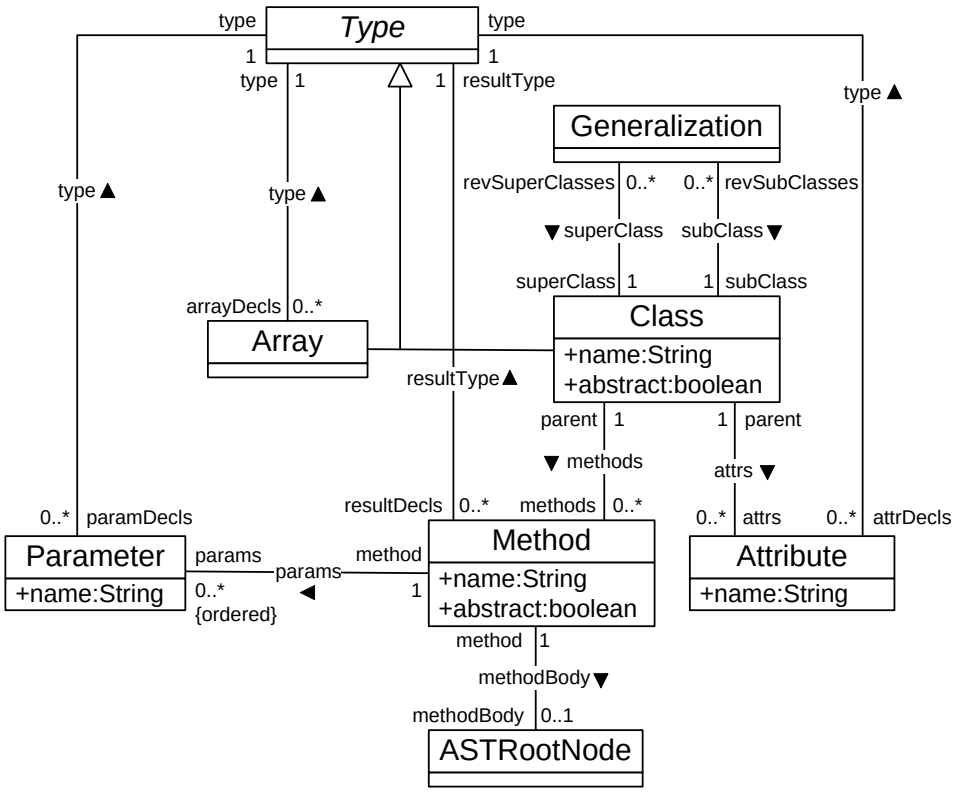

Abbildung 2.2: Ausschnitt aus dem Strukturmodell für Quelltexte objektorientierter Sprachen, Paket Structure

repräsentiert die Wurzel eines Methodenrumpfes. Alle weiteren Klassen des abstrakten Syntaxbaums werden wegen Platzmangels nicht abgebildet. Die Klassen des Strukturmodells gehören zum Paket Structure.

Auf das Verhalten eines Softwaresystems kann durch seine Struktur nur indirekt geschlossen werden. Dazu müssen potentielle Methodenaufrufe in den abstrakten Syntaxbäumen der Methodenrümpfe identifiziert werden. Tatsächlich zur Laufzeit des Softwaresystems ausgeführte Methodenaufrufe oder Sequenzen von ausgeführten Methodenaufrufen können aber der Struktur nicht entnommen werden.

Beispiel

Als durchgängiges Beispiel eines zu untersuchenden Softwaresystems wird in dieser Arbeit ein Mediaplayer verwendet. Der Quelltext dieses Softwaresystems

wird auf Basis des Strukturmodells aufbereitet und dem Reverse-Engineer als Klassendiagramm präsentiert (Abbildung 2.3).

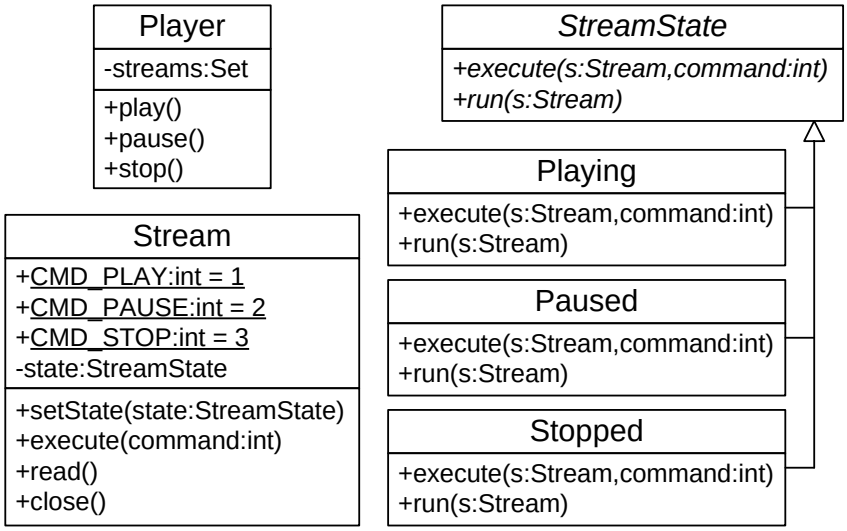

Abbildung 2.3: Klassendiagramm eines Mediaplayers

Das Beispiel stellt einen Ausschnitt eines Softwaresystems zum Abspielen von Multimediadaten wie Liedern oder Filmen dar. Die Daten werden von dem Programm in Form von Datenströmen verarbeitet. Der Benutzer kann die Multimediadaten mit dem Mediaplayer abspielen, ihre Wiedergabe pausieren oder auch beenden. Diese Befehle werden von der zentralen Klasse Player entgegengenommen und an den Datenstrom (Stream) weitergereicht. Der Datenstrom speichert seinen aktuellen Zustand in einem Zustandsobjekt (vom Typ StreamState), das die Ausführung des Befehls durchführt und gegebenenfalls den Zustand des Datenstroms ändert.

Das Klassendiagramm enthält keine Assoziationen, da sie wie bereits erläutert nicht direkt aus den Quelltexten extrahiert, sondern nur durch eine erweiterte Analyse der Struktur gewonnen werden können. Die Beziehungen zwischen den Klassen werden im Quelltext und somit auch in der Struktur nur durch Attribute modelliert. Das Attribut state:StreamState der Klasse Stream ist leicht als eine einfache Referenz von Stream auf StreamState erkennbar. Das Attribut streams:Set der Klasse Player dagegen kann jedoch erst durch Analyse der in der Menge gespeicherten Typen als eine mengenwertige Referenz von Player auf Stream identifiziert werden.

2.3.2 Spezifikation von Strukturmustern

In der strukturbasierten Entwurfsmustererkennung von FUJABA werden Muster spezifiziert, die den strukturellen Anteil eines Entwurfsmusters beschreiben. Mit Hilfe dieser Muster werden in der Struktur eines Softwaresystems Entwurfsmusterimplementierungen identifiziert. Diese Muster werden im Folgenden als *Strukturmuster* bezeichnet.

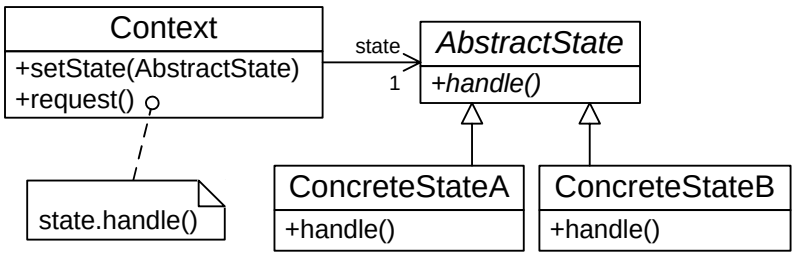

Abbildung 2.4: Die Struktur des *State* Entwurfsmusters

Abbildung 2.4 zeigt die Struktur des *State*-Entwurfsmusters als Klassendiagramm. Die Klasse Context referenziert eine abstrakte Oberklasse AbstractState. Diese Klasse gibt eine gemeinsame Schnittstelle für konkrete Klassen vor, die verschiedene Zustände der Klasse Context implementieren. In dieser Abbildung sind beispielhaft zwei konkrete Zustände ConcreteStateA und ConcreteStateB vorgegeben, es können aber prinzipiell beliebig viele verschiedene Zustände existieren. Zur Laufzeit hat ein Objekt der Klasse Context einen aktuellen Zustand, an den es Anfragen – Aufrufe der Methode request() – delegiert. Durch Zustandswechsel kann das Context-Objekt unterschiedlich auf diese Anfragen reagieren.

In Abbildung 2.4 ist die Struktur des *State*-Entwurfsmusters in konkreter Syntax – einem Klassendiagramm – zu sehen. Strukturmuster werden dagegen auf Basis der abstrakten Syntax der Struktur spezifiziert und beschreiben jeweils einen Teilgraphen innerhalb der Struktur eines Softwaresystems. Die Struktur ist eine Instanz des Strukturmodells, sie ist also durch das Strukturmodell typisiert. Da Strukturmuster Ausschnitte aus der Struktur beschreiben, sind auch sie durch das Strukturmodell typisiert.

Abbildung 2.5 zeigt das *State*-Strukturmuster. Die Syntax der Strukturmuster ist an UML-Objektdiagramme angelehnt. Strukturmuster sind Graphgrammatikregeln [Roz97], die aus einer linken und einer rechten Regelseite bestehen. Die linke Regelseite ist ein Teilgraph, der in einem Wirtsgraphen gesucht wird. Die rechte Regelseite enthält den gesuchten Teilgraphen und

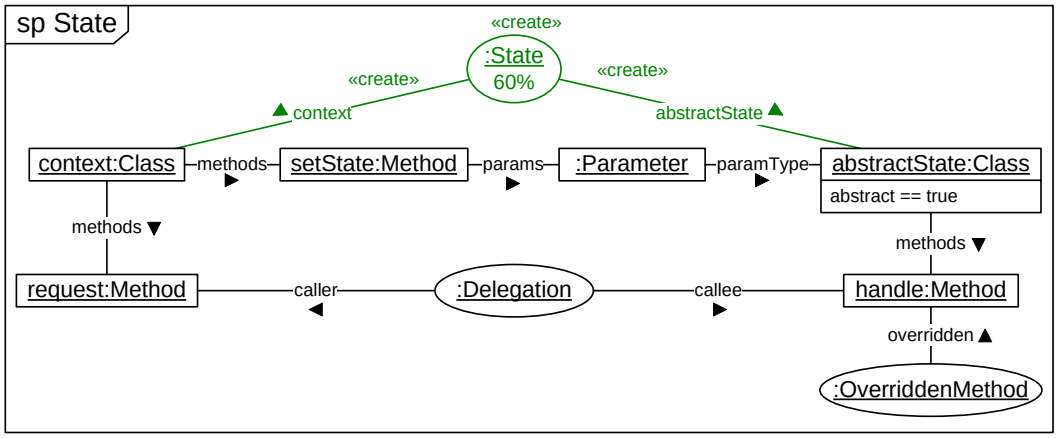

Abbildung 2.5: Das Strukturmuster des *State*-Entwurfsmusters

beschreibt die Modifikationen an diesem Teilgraphen. Modifikationen können das Erzeugen und Löschen von Knoten und Kanten oder auch das Ändern von Knotenattributen sein. Ist eine homomorphe Abbildung der Knoten der linken Regelseite auf Knoten des Wirtsgraphen möglich, so ist der Teilgraph gefunden und die Graphgrammatikregel kann angewendet werden. Das bedeutet, die Modifikationen der rechten Regelseite werden an dem gefundenen Teilgraphen durchgeführt.

Strukturmuster sind eingeschränkte Graphgrammatikregeln. In einem Strukturmuster werden die linke und die rechte Regelseite gemeinsam in einem Graphen dargestellt. Die einzige Modifikation, die ein Strukturmuster durchführt, ist die Erzeugung eines Knotens, der so genannten *Annotation*[2], und einiger Kanten, die die Annotation mit Knoten des gefundenen Teilgraphen verbinden. Dieser Annotationsknoten und die zu erzeugenden Kanten sind mit dem Stereotyp «create» gekennzeichnet. Die linke Regelseite besteht also aus den Knoten und Kanten, die nicht den Stereotyp «create» tragen. Die rechte Regelseite besteht aus allen Knoten und Kanten des Strukturmusters. Der Prozentwert innerhalb des mit «create» markierten Annotationsknotens wird später in Abschnitt 2.3.5 erläutert.

Durch das Anwenden von Strukturmustern auf die Struktur eines Softwaresystems wird die Struktur durch Annotationen angereichert. Die Annotationen markieren die Fundstellen von potentiellen Entwurfsmusterimplementierungen.

[2] visualisiert als Oval

Das *State*-Strukturmuster beschreibt nur einige Aspekte der Struktur des *State*-Entwurfsmusters. So fehlen die überschreibenden Methoden der konkreten Zustände und die Referenz der Klasse Context auf die Klasse AbstractState. Diese Aspekte werden durch so genannte *Hilfsmuster* abgedeckt. Annotationen können in anderen Strukturmustern wiederverwendet werden. In Abbildung 2.5 sind zwei ovale Objekte mit den Namen :Delegation und :OverriddenMethod zu finden. Das sind Annotationen, die durch andere Strukturmuster erzeugt wurden und Hilfsmuster repräsentieren.

Hilfsmuster sind Teile von Entwurfsmustern, die immer wieder in unterschiedlichen Entwurfsmustern vorkommen. Die Verwendung von Annotationen in den Strukturmustern erlaubt es, Strukturmuster verschiedener Hilfsmuster zu kombinieren und so komplexere Strukturmuster für Entwurfsmuster zu schaffen. So müssen gleiche Teile nicht mehrfach spezifiziert werden. Ein Beispiel für ein solches Hilfsmuster ist die *Delegation*. Bei einer *Delegation* wird ein Methodenaufruf entlang einer Referenz von einem Objekt an ein anderes weiter gegeben, der Methodenaufruf wird delegiert.

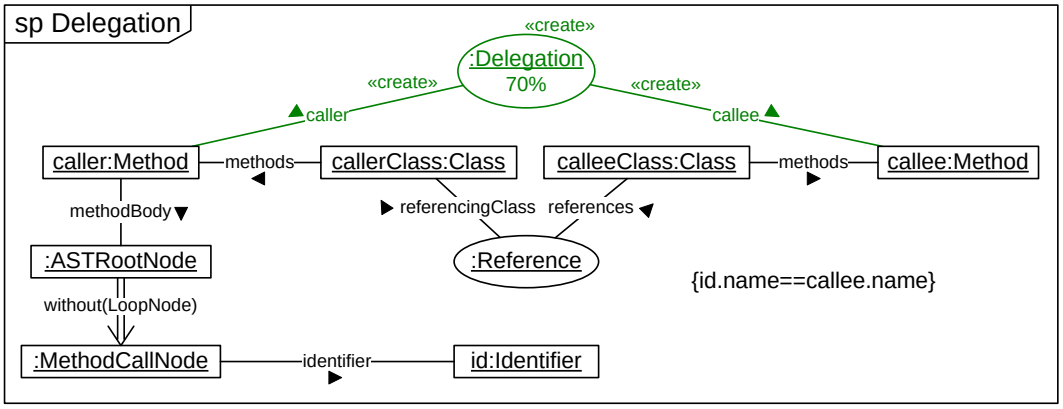

Abbildung 2.6: Das Strukturmuster des *Delegation*-Hilfsmusters

Abbildung 2.6 zeigt das *Delegation*-Strukturmuster. Es beschreibt zwei Methoden, deren Klassen durch ein weiteres Hilfsmuster, eine *Reference*, verbunden sind. Innerhalb des Methodenrumpfs der Methode caller:Method existiert ein Methodenaufruf, wobei der Name der aufgerufenen Methode mit dem Namen der zweiten Methode, callee:Method, übereinstimmt. Der Methodenaufruf darf an fast beliebiger Stelle im Rumpf der Methode stattfinden, nicht jedoch innerhalb einer Schleife. Um dies im Strukturmuster auszudrücken, wird ein Pfad verwendet. Während bei Kanten zwischen zwei Objekten diese unmit-

telbar miteinander verbunden sein müssen, schreiben Pfade vor, dass die zwei über den Pfad verbundenen Objekte nur mittelbar über beliebige Kanten und andere Objekte verbunden sind. In diesem Fall sind sogar die Typen der Objekte, die auf dem Pfad liegen, eingeschränkt. Es darf kein Objekt des Typs LoopNode auf dem Pfad liegen, das bedeutet, der Methodenaufruf darf sich nicht innerhalb einer Schleife befinden.

Der Methodenaufruf wird in diesem Strukturmuster nur über den Namen identifiziert. Eine Überprüfung des Typs des Objekts, auf dem der Methodenaufruf stattfindet, wird nicht durchgeführt. In dem abstrakten Syntaxbaum des Methodenrumpfes, der direkt aus dem Quelltext erzeugt wurde, können die Typen der Variablen im Methodenrumpf nur in wenigen Spezialfällen zweifelsfrei ermittelt werden. Für eine eindeutige Ermittlung der Typen ist ein Übersetzer notwendig, der den gesamten Quelltext und alle notwendigen Bibliotheken kennt. In der strukturbasierten Entwurfsmustererkennung in FUJABA kommt jedoch kein Übersetzer zum Einsatz, da nicht immer der gesamte Quelltext analysiert werden soll oder zur Verfügung steht. An dieser Stelle wird also nur eine Heuristik verwendet, die zu False-Positives bei der Entwurfsmustererkennung führt.

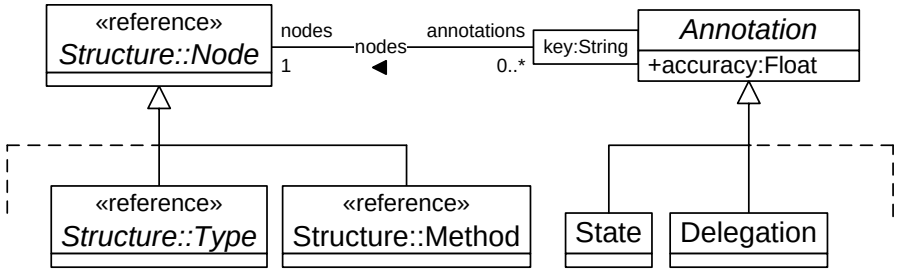

Abbildung 2.7: Modell der Annotationen, Paket Annotations

In Abbildung 2.7 ist das Modell der Annotationen dargestellt. Die Klassen dieses Modells gehören zum Paket Annotations. Annotationen, repräsentiert durch die abstrakte Klasse Annotation, können mit allen Elementen der Struktur verbunden werden. Die Klasse Node ist die Oberklasse aller Klassen des Strukturmodells aus Abbildung 2.2. Von der abstrakten Oberklasse der Annotationen erben konkrete Annotationen wie die des *State*- oder des *Delegation*-Strukturmusters. Als Schlüssel für die Referenz dienen die Namen, die an den Kanten, die im Strukturmuster von den Annotationen ausgehen, stehen. Eine Annotation vom Typ State referenziert zum Beispiel zwei Objekte von Typ Class unter den Schlüsseln context und abstractState.

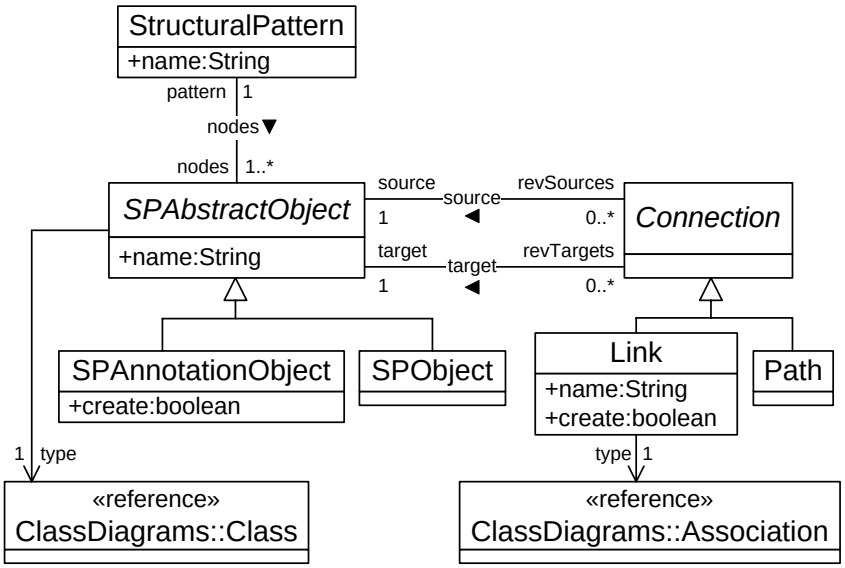

Abbildung 2.8: Metamodell der Strukturmuster, Paket StructuralPatterns

Das Metamodell der Strukturmuster ist in 2.8 abgebildet. Ein Strukturmuster (StructuralPattern) besteht aus Knoten (SPAbstractObject), die durch Kanten (Connection) verbunden sind. Die Knoten sind entweder Annotationen (SPAnnotationObject) oder normale Objekte (SPObject). Die Kanten sind normale Verbindungen (Link) oder Pfade (Path).

Das Metamodell der Strukturmuster ist mit dem Metamodell der Strukturmodelle aus Abbildung 2.1 verbunden, da, wie bereits erwähnt, das Strukturmodell den Strukturmustern zur Typisierung dient. Die Klasse SPAbstractObject referenziert die Klasse Class aus dem Metamodell der Strukturmodelle. So kann der Typ eines Strukturmusterobjekts oder einer Annotation festgelegt werden. Die Klasse Link für die Verbindungen der Objekte im Strukturmuster referenziert die Klasse Association.

2.3.3 Regelkatalog

Zusammengehörige Strukturmuster werden in einem *Regelkatalog* organisiert. Im Folgenden werden Strukturmuster auch synonym als Regeln bezeichnet, da Strukturmuster wie bereits erwähnt Graphgrammatikregeln sind. Die Wiederverwendung von Strukturmustern durch Annotationen in anderen Strukturmustern erzeugt Abhängigkeiten. Im Regelkatalog sind unter anderem diese Abhängigkeiten zwischen den Strukturmustern festgehalten. Abbildung 2.9

zeigt einen Regelkatalog, der auch die vorgestellten *State*- und *Delegation*-Strukturmuster enthält.

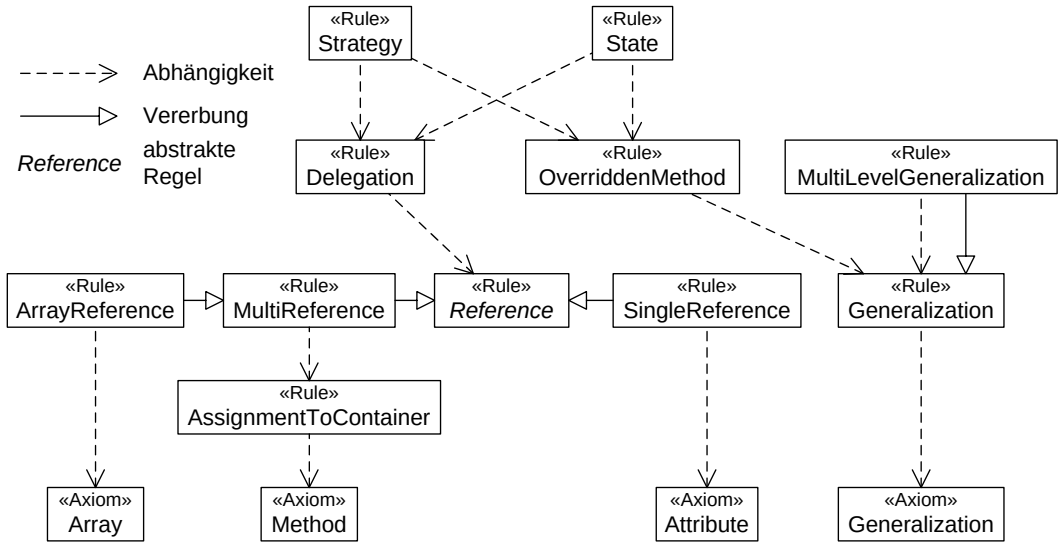

Abbildung 2.9: Regelkatalog

Auf unterster Ebene des Regelkatalogs liegen die *Axiome*. Axiome sind feststehende Fakten aus der Struktur, also Klassen, Methoden, Attribute oder andere Knoten. Über den Axiomen sind die Regeln der Hilfs- und Entwurfsmuster angeordnet, die auf den Axiomen und anderen Regeln aufbauen.

Damit die Regeln im Erkennungsprozess in einer korrekten Reihenfolge angewendet werden, wird jeder Regel des Regelkatalogs ein Rang zugeordnet. Die Regeln, die nur von Axiomen, nicht aber von anderen Regel abhängen, erhalten den Rang 0. Alle weiteren Regeln erhalten einen Rang gemäß ihrer topologischen Sortierung.

Im Regelkatalog werden des Weiteren Vererbungen zwischen den Strukturmustern festgelegt. Es kann Hilfsmuster oder Entwurfsmuster geben, die die gleiche Semantik haben, aber durch unterschiedliche Syntax implementiert werden. Das Hilfsmuster *Reference* zum Beispiel sagt aus, dass eine Klasse eine andere Klasse referenziert. Es kann allerdings verschiedene Implementierungen für solche Referenzierungen geben. Zum einen kann zur Laufzeit ein Objekt immer nur ein einzelnes Objekt referenzieren. In diesem Fall würde eine *SingleReference* vorliegen. Zum anderen kann jedoch zur Laufzeit ein Objekt auch mehrere Objekte gleichzeitig referenzieren, dies wäre eine *MultiReference*. Eine *MultiReference* kann aber wiederum auf unterschiedliche Art implementiert

werden. Eine Möglichkeit wäre die Verwendung eines Container-Objekts zur Speicherung der Referenzen, eine andere Möglichkeit die Verwendung eines Arrays, wie im Hilfsmuster *ArrayReference*.

Bei Verwendung der Vererbung wird zunächst in abstrakten Regeln die gemeinsame Schnittstelle der von den konkreten Regeln erzeugten Annotationen festgelegt. Im Falle der *Reference* wären dies zwei Klassen aus Struktur, die unter den Schlüsseln referencingClass und references annotiert werden. Diese Schnittstelle wurde im Strukturmuster des *Delegation*-Hilfsmusters in Abbildung 2.6 verwendet. In den konkreten Strukturmustern *SingleReference*, *MultiReference* und *ArrayReference* wird dann beschrieben, welche Teilgraphen vorliegen müssen, damit zwei Klassen, wie in der Schnittstelle vorgegeben, annotiert werden.

Wird eine Regel angewendet, so kann für eine darin wiederverwendete Annotation eines abstrakten Strukturmusters eine Annotation eines erbenden, konkreten Strukturmusters polymorph eingesetzt werden. So würde im Beispiel des *Delegation*-Hilfsmusters eine Delegation erkannt werden, egal ob die beiden Klassen durch eine *SingleReference*, *MultiReference* oder *ArrayReference* verbunden sind. Die Vererbung von Strukturmustern ist somit eine von vielen Strategien, dem Problem der Variantenvielfalt von Entwurfsmusterimplementierungen in der Entwurfsmustererkennung zu begegnen.

2.3.4 Strukturbasierter Erkennungsprozess

Jedes Softwaresystem weist individuelle Besonderheiten in seiner Implementierung auf. Das ist auf individuelle Implementierungsstile der Programmierer und unterschiedliche Firmenkulturen zurückzuführen. So entstehen viele Implementierungsvarianten von Hilfsmustern und Entwurfsmustern. Die Strukturmuster müssen jeweils an solche Besonderheiten angepasst werden, um ein gutes Ergebnis bei der Entwurfsmustererkennung zu erzielen. In der strukturbasierten Entwurfsmustererkennung in Fujaba wird daher ein iterativer Prozess vorgeschlagen [NSW+02].

Die Eingaben des Erkennungsprozesses sind der Quelltext und der Regelkatalog. Der Erkennungsprozess startet entweder mit einem neuen Regelkatalog, der sukzessive durch neue Strukturmuster erweitert wird, oder mit einem bereits vorhandenen Regelkatalog, dessen Strukturmuster an die Besonderheiten des zu untersuchenden Softwaresystems angepasst werden. In jedem Iterationsschritt werden jeweils eine Analyse des Softwaresystems oder eines Teils des Softwaresystems durchgeführt und auf Basis der dabei erzielten Ergebnisse die Strukturmuster angepasst. Das Ergebnis der statischen Analyse sind poten-

tielle Entwurfsmusterimplementierungen, so genannte *Kandidaten*. Abbildung 2.10 zeigt eine schematische Darstellung des strukturbasierten Erkennungsprozesses.

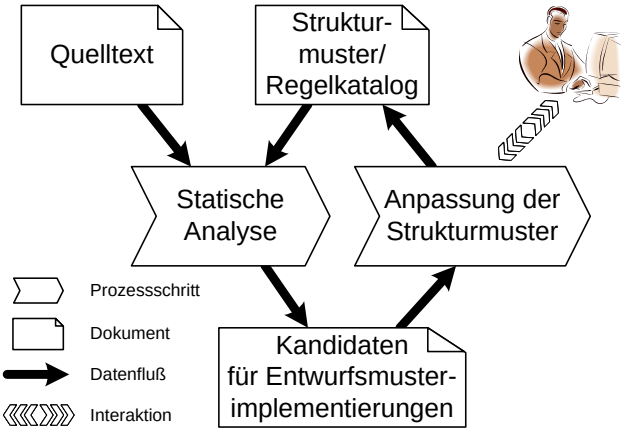

Abbildung 2.10: Der strukturbasierte Erkennungsprozess

Die Abhängigkeiten der Strukturmuster im Regelkatalog geben eine bestimmte Reihenfolge vor, in der die Regeln angewendet werden müssen. Bevor zum Beispiel die Regel des *State*-Entwurfsmusters angewendet werden kann, müssen die Regeln der *Delegation* und der *OverriddenMethod* ausgeführt worden sein. Üblich sind bei solchen Abhängigkeiten Algorithmen, die die Regeln gemäß ihrer topologischen Sortierung im Regelkatalog *bottom-up*, also von unten nach oben, anwenden.

Regeln, die Entwurfsmusterimplementierungen beschreiben, bauen typischerweise auf einer Vielzahl anderer Regeln für Hilfsmuster auf. Beim bottom-up-Algorithmus werden diese Regeln daher erst spät ausgeführt, so dass vorhandene Entwurfsmusterimplementierungen und damit aussagekräftige Ergebnisse erst nach Durchführung einer vollständigen Analyse erkannt werden. Diese Ansätze bieten also keine optimale Unterstützung des Reverse-Engineers, da bei der Analyse großer Systeme jeder Iterationsschritt sehr zeitaufwändig ist und außerdem oft eine Vielzahl von Entwurfsmustern erkannt werden, die dann manuell ausgewertet werden müssen.

Die strukturbasierte Entwurfsmustererkennung in Fujaba verwendet daher einen inkrementellen Algorithmus, der möglichst frühzeitig aussagekräftige Ergebnisse produziert und durch den Reverse-Engineer unterbrochen werden kann. Dadurch kann der Reverse-Engineer frühzeitig und gezielt Ergebnisse untersuchen und daraufhin den Regelkatalog anpassen.

Die frühzeitige Produktion aussagekräftiger Ergebnisse wird durch die Angabe eines Kontextes, in dem eine Regel angewendet wird, unterstützt. Der Kontext ist ein Knoten der Struktur, von dem angenommen wird, dass er auch ein Knoten des durch das Strukturmuster beschriebenen Teilgraphen ist. So kann ausgehend von dem Kontext der Rest des Teilgraphen gesucht werden. Die Angabe des Kontextes ermöglicht die Ausführung einer Regel in polynomieller Laufzeit.

Um frühzeitig Ergebnisse zu produzieren, wird in diesem Ansatz eine Kombination aus *bottom-up-* und *top-down-Algorithmus* eingesetzt. Zu Beginn der Analyse werden die Regeln des Ranges 0 zusammen mit Axiomen aus der Struktur als *Kontext-Regel-Paare* in eine bottom-up-Prioritätswarteschlange eingefügt. Die Warteschlange ist absteigend nach dem Rang der Regel sortiert. Dadurch werden Regeln hohen Ranges, die aussagekräftige Ergebnisse versprechen, möglichst frühzeitig ausgeführt.

Im bottom-up-Modus wird jeweils das vorderste Kontext-Regel-Paar aus der Schlange entfernt. Es wird versucht, die Regel im Kontext anzuwenden. Dazu wird zunächst nach dem von der Regel spezifizierten Teilgraphen gesucht. Wird der Teilgraph gefunden, wird die Regel angewendet und eine Annotation in der Struktur erzeugt. Sind weitere Regeln von der erfolgreich angewendeten Regel abhängig, so werden diese zusammen mit der erzeugten Annotation als Kontext-Regel-Paar in die Schlange einsortiert.

Kann der Teilgraph jedoch nicht gefunden werden, wird die auszuführende Regel nicht sofort verworfen. Stattdessen wird überprüft, ob im Teilgraphen geforderte Annotationen in der Struktur fehlen. In diesem Fall wird versucht, zuerst die fehlenden Annotationen durch Ausführung der entsprechenden Regeln zu erzeugen und dann die auszuführende Regel anzuwenden. Dazu schaltet der Algorithmus vom bottom-up- in den top-down-Modus um.

Im top-down-Modus werden zunächst das aktuelle Kontext-Regel-Paar und rekursiv alle Regeln, von denen die aktuelle Regel abhängig ist, mit entsprechendem Kontext in einen anfangs leeren Stack gelegt. Dann wird ein Kontext-Regel-Paar vom Stack entfernt und der Teilgraph der Regel gesucht. Wird der Teilgraph gefunden, wird die Regel angewendet und eine Annotation erzeugt. Sind andere Regeln von der gerade ausgeführten abhängig, so werden wiederum die abhängigen Regeln mit der Annotation als Kontext-Regel-Paare in die Warteschlange für den bottom-up-Modus einsortiert. Der top-down-Modus endet, wenn der Stack leer ist.

Nach beendetem top-down-Modus fährt der Algorithmus im bottom-up-Modus fort. Der bottom-up-Modus terminiert, wenn die Schlange leer ist. Der Reverse-Engineer kann jedoch den Algorithmus auch jederzeit unterbrechen,

um sich erste Ergebnisse präsentieren zu lassen.

Die Skalierbarkeit des Verfahrens wurde in [NSW+02] belegt. Das Verfahren wurde erfolgreich auf die JAVA-Bibliotheken JAVA Generic Library (JGL) mit 36.500 LOC und JAVA Abstract Window Toolkit (AWT) mit 114.000 LOC angewendet. Die Gesamtdauer der Analyse bleibt mit zum Beispiel ca. 22 Minuten für die Bibliothek JAVA AWT in einer vertretbaren Größenordnung.

2.3.5 Bewertung der Ergebnisse

Die Bewertung der Kandidaten basiert auf einer Bewertung der Strukturmuster. Bei der Spezifikation eines Strukturmusters muss der Reverse-Engineer einen so genannten *Vertrauenswert* angeben. Er wird in den mit dem Stereotyp ≪create≫ gekennzeichneten Annotationsknoten eingetragen. Der Vertrauenswert drückt die Güte des Strukturmusters durch das Verhältnis der Anzahl der durch das Strukturmuster identifizierten, tatsächlichen Entwurfsmusterimplementierungen zu der Anzahl aller durch das Strukturmuster identifizierten Kandidaten aus. Der Vertrauenswert des *State*-Strukturmusters in Abbildung 2.5 sagt zum Beispiel aus, dass 60% aller Kandidaten tatsächliche Entwurfsmusterimplementierungen sind, während 40% False-Positives sind. Dieser Wert wird durch den Reverse-Engineer aufgrund seiner Erfahrung geschätzt.

Die Güte eines Kandidaten wird durch den einen *Genauigkeitswert* ausgedrückt, der sich aus den Vertrauenswerten der an einer Annotation beteiligten Strukturmuster berechnet. Die Berechnung der Genauigkeitswerte findet in einem *Fuzzy-Petri-Netz* (FPN) [Jah99] statt, in dem die Abhängigkeiten aller erzeugten Annotationen festgehalten sind.

In Abbildung 2.11 ist ein FPN vor (links) und nach (rechts) der Berechnung zu sehen. Jede Annotation wird im FPN durch eine Stelle repräsentiert. Abhängigkeiten zwischen Annotationen werden durch Transitionen modelliert. In dem Beispiel ist eine Annotation :State dargestellt, die von einer Annotation :Delegation und einer Annotation :OverriddenMethod abhängt, die wiederum von weiteren Annotationen abhängen. Der Vertrauenswert eines Strukturmusters wird in die Transition aus dem Vorbereich der Annotation des Strukturmusters eingetragen. Für das *State*-Strukturmuster wurde zum Beispiel der Vertrauenswert von 60% eingetragen.

In der Anfangsmarkierung des Fuzzy-Petri-Netzes erhalten die Stellen der Annotationen, die von keinen anderen Annotationen abhängig sind, zunächst den Vertrauenswert ihres Strukturmusters als Markierung. Die Annotation :Generalization erhält zum Beispiel die Markierung 100%. Alle anderen Stellen erhalten die Markierung 0%. Das Fuzzy-Petri-Netz wird nun solange ausgeführt,

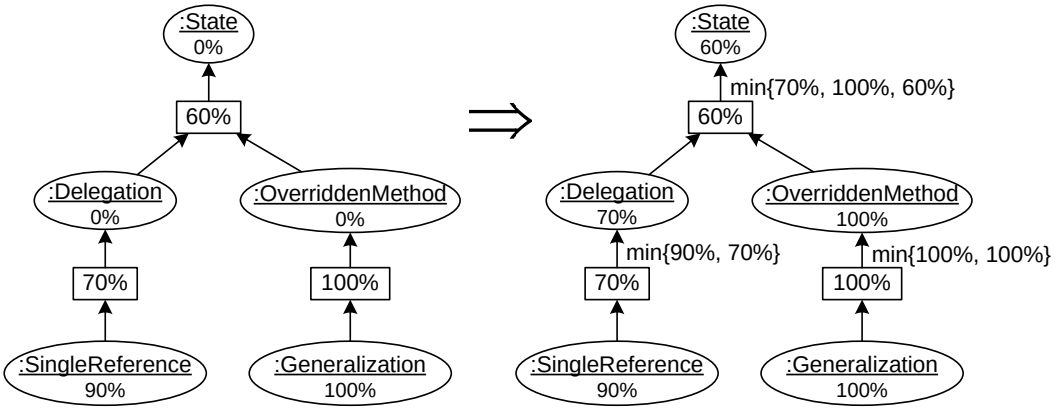

Abbildung 2.11: Fuzzy-Petri-Netz zur Bewertung eines *State*-Kandidaten

bis es stabil ist. Die neue Markierung einer Stelle berechnet sich dabei aus dem Minimum der Markierungen der Stellen aus dem Vorbereich und dem Vertrauenswert der Transition. Die rechte Seite in Abbildung 2.11 zeigt das stabile FPN. Die Markierung einer Stelle wird schließlich als Genauigkeitswert ihrer Annotationen interpretiert. Details zur Berechnung der Genauigkeitswerte sind in [Wen01] und [Nie04] zu finden.

Die Annotationen werden zusammen mit ihren Genauigkeitswerten dem Reverse-Engineer als Ergebnis der strukturbasierten Entwurfsmustererkennung in Form eines annotierten Klassendiagramms präsentiert. Abbildung 2.12 zeigt den Kandidaten einer *State*-Entwurfsmusterimplementierung am Beispiel des Mediaplayers aus Abbildung 2.3, Seite 20. Die Annotation wird als Oval mit dem Namen des Entwurfsmusters und ihrem Genauigkeitswert dargestellt. Die annotierten Klassen der Struktur werden durch Linien mit der Annotation verbunden. An den Linien ist jeweils der Name der Rolle verzeichnet, die die Klasse in der Entwurfsmusterimplementierung spielt.

Die Vertrauenswerte der Strukturmuster werden bei der Spezifikation durch den Reverse-Engineer geschätzt. Da dieser Wert nicht auf tatsächlichen Daten basiert, wurde eine automatische Adaption der Vertrauenswerte entwickelt [Rec04], die Eingaben des Reverse-Engineers verwendet. Nach der Entwurfsmustererkennung kann der Reverse-Engineer den Genauigkeitswert jeder Annotation ändern. Der Vertrauenswert kann zum Beispiel auf 0% gesenkt werden, wenn der entsprechende Kandidat ein False-Positive ist. Der Vertrauenswert kann aber auch auf einen beliebigen Wert zwischen 0% und 100% gesetzt werden, um die vom Reverse-Engineer beurteilte Genauigkeit einer Annotati-

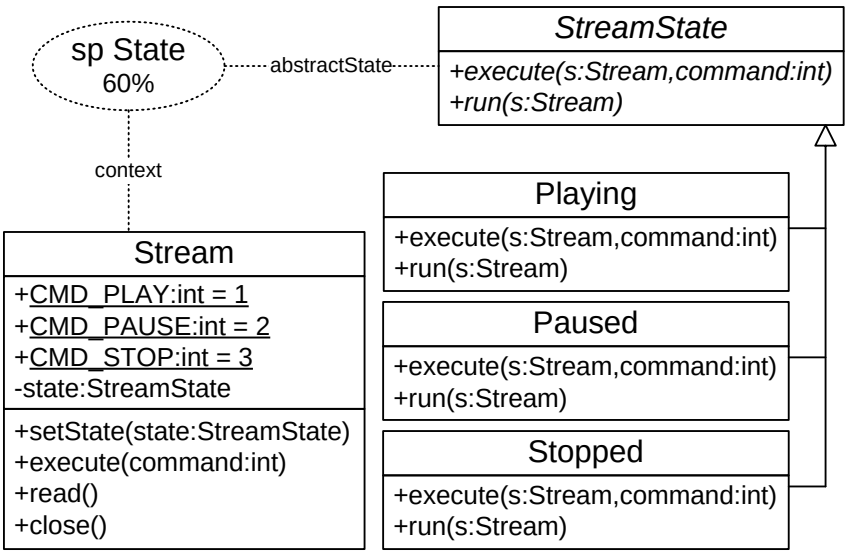

Abbildung 2.12: Ein Kandidat einer *State*-Entwurfsmusterimplementierung

on auszudrücken. Die korrigierten Genauigkeitswerte aller Annotationen eines Strukturmusters fließen bei der automatischen Adaption in die Berechnung eines neuen Vertrauenswertes des Strukturmusters ein. Der neue Vertrauenswert wird dann bei der nächsten Entwurfsmustererkennung verwendet.

2.3.6 Einsatzgebiete

Die strukturbasierte Entwurfsmustererkennung lässt sich nicht nur zur Identifikation von Entwurfsmusterimplementierungen einsetzen. Sie wird auch zur Suche nach Anti-Patterns [BMMM98] verwendet. Anti-Patterns sind im Gegensatz zu Entwurfsmustern schlechte, ungeeignete Implementierungen für immer wiederkehrende Probleme. Ihre Identifikation in Softwaresystemen lässt auf Designschwächen oder Probleme mit der Wartbarkeit der Systeme schließen und kann zu ihrer Verbesserung eingesetzt werden [Mey06].

Das in diesem Kapitel vorgestellte Strukturmodell ist nur zur Repräsentation objektorientierter Softwaresysteme geeignet. Mit dem Metamodell für Strukturmodelle lassen sich jedoch beliebige Strukturmodelle beschreiben. Die strukturbasierte Entwurfsmustererkennung ist daher in ihrer Anwendungsdomäne nicht auf objektorientierte Softwaresysteme beschränkt. Sie wird auch auf andere Strukturmodelle angewendet, so zum Beispiel auf Modelle des Werkzeugs MATLAB/SIMULINK [GMW06].

MATLAB/SIMULINK ist eine Entwicklungs- und Simulationsumgebung für Algorithmen zu numerischen Berechnungen, zur Datenanalyse und -visualisierung und bietet zu diesem Zweck eine eigene Hochsprache [Mat]. In diesem Fall wird das Strukturmodell für objektorientierte Sprachen gegen ein Strukturmodell für die Hochsprache von MATLAB/SIMULINK ausgetauscht. Die strukturbasierte Entwurfsmustererkennung kann dadurch zur Erkennung von Musterimplementierungen in MATLAB/SIMULINK eingesetzt werden. In dem in dieser Arbeit vorgestellten Verfahren wird die strukturbasierte Entwurfsmustererkennung allerdings nur auf Softwaresysteme angewendet, die in einer objektorientierten Sprache geschrieben sind.

2.3.7 Überblick

Im Folgenden wird ein Überblick über die Abstraktionsschichten der strukturbasierten Entwurfsmustererkennung gegeben. In Abbildung 2.13 werden die Diagramme der letzten Abschnitte in ein Schema aus Metamodellen, Modellen und Instanzen beziehungsweise Implementierungen eingeordnet. Die linke Hälfte der Tabelle zeigt die Abstraktionsschichten der Struktur eines Softwaresystems, die rechte Hälfte die Abstraktionsschichten der Strukturmuster.

Das Strukturmodell in der mittleren Abstraktionsschicht der linken Hälfte legt fest, wie die Struktur eines Softwaresystems repräsentiert wird. Das Strukturmodell ist je nach Anwendungsdomäne der strukturbasierten Entwurfsmustererkennung austauschbar. In dieser Arbeit wird ein Strukturmodell für Quelltexte objektorientierter Sprachen verwendet. Ein Beispiel für eine Instanz des Strukturmodells, also die Struktur eines konkreten Softwaresystems, ist in der untersten Abstraktionsschicht zu sehen. Das Metamodell für Strukturmodelle in der obersten Abstraktionsschicht legt dagegen fest, wie die verschiedenen Strukturmodelle zu beschreiben sind.

Auf der rechten Seite ist in der mittleren Abstraktionsschicht das Strukturmuster für das *State*-Entwurfsmuster abgebildet. In dieser Schicht liegen jedoch alle Strukturmuster eines Regelkatalogs, das *State*-Strukturmuster ist nur ein Repräsentant. Ein Strukturmuster beschreibt in einer Art UML-Objektdiagramm Teilgraphen in der Struktur eines Softwaresystems. Aus diesem Grund verwenden die Strukturmuster das Strukturmodell als semantischen Typgraphen. In der untersten Ebene ist eine konkrete Implementierung eines *State*-Entwurfsmusters abgebildet, die in der Struktur der linken Seite gefunden wurde. Das Metamodell der Strukturmuster referenziert wiederum das Metamodell der Strukturmodelle, um so die semantische Typbeziehung zwischen Strukturmuster und Strukturmodell herzustellen.

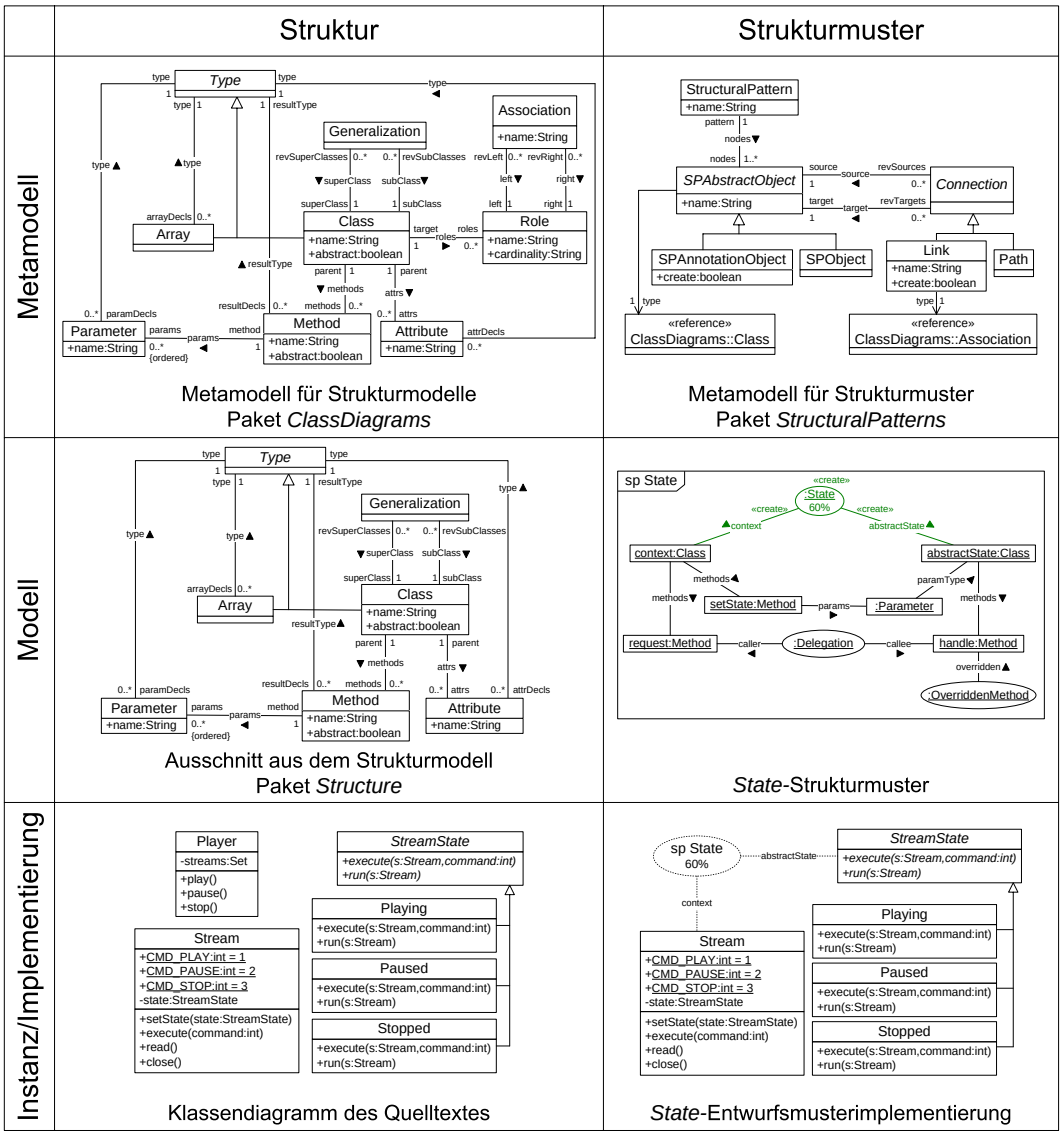

Abbildung 2.13: Die Abstraktionsschichten der strukturbasierten Entwurfsmustererkennung

In dieser Tabelle bestehen in der Vertikalen von oben nach unten jeweils Modell-Instanz-Beziehungen zwischen den Diagrammen. In der Horizontalen stehen die Diagramme dagegen in einer semantischen Beziehung zueinander, die in der Schicht der Metamodelle explizit durch Referenzen zwischen den Metamodellen festgelegt wurde.

2.4 Zusammenfassung

Die strukturbasierte Entwurfsmustererkennung in Fujaba bietet eine gute Grundlage für eine Erweiterung zu einem Verfahren, das alle der in Abschnitt 2.2 genannten Anforderungen erfüllt. Die Skalierbarkeit des Verfahrens wurde in [NSW+02] belegt. Die leichte Anpassbarkeit der Entwurfsmusterspezifikationen ist durch die UML-Objektdiagramme-ähnliche Spezifikationssprache gegeben. Die Spezifikationen sind durch die graphische Notation von dem Reverse-Engineer schnell zu erfassen, da die UML in der Softwaretechnik eine sehr weit verbreitete Sprache ist. Des Weiteren ist das Strukturmodell leicht austauschbar, so dass die Entwurfsmustererkennung auch auf andere Strukturmodelle anwendbar ist [GMW06].

Die vorgestellte Bewertung der Ergebnisse ist allerdings mangelhaft. Die Bewertung beruht nicht auf der Beurteilung der Güte einer einzelnen, potentiellen Entwurfsmusterimplementierung, sondern auf der Beurteilung der Güte des Strukturmusters. Die Bewertung einer Annotation wird über die Typen der Annotationen, von denen sie abhängt, berechnet. Zwei Annotationen eines Strukturmusters hängen jedoch in der Regel von Annotation der gleichen Typen ab. Sie können sich nur dann unterscheiden, wenn durch Polymorphie eine Annotation eines Subtypen verwendet wird. Im Falle zweier *Delegation*-Annotationen zum Beispiel unterscheiden sich ihre Bewertungen nicht, wenn beide von einer *SingleReference* abhängen. Erst wenn zum Beispiel eine Annotation von einer *SingleReference*-Annotation, und die andere von einer *MultiReference*-Annotation abhängt[3], erhalten sie unterschiedliche Bewertungen. Das gegebene Verfahren ist daher nicht zur Bewertung der potentiellen Entwurfsmusterimplementierungen geeignet. In Kombination mit der automatischen Adaption kann es allerdings zur Bewertung der Strukturmuster genutzt werden.

Die Präzision der gegebenen Entwurfsmustererkennung in Fujaba leidet

[3]Die *SingleReference*- und *MultiReference*-Strukturmuster erben beide vom abstrakten *Reference*-Strukturmuster, siehe Abbildung 2.9. Das *Delegation*-Strukturmuster fordert nur eine Annotation vom Typ *Reference* (Abbildung 2.6).

unter der ausschließlichen Analyse der Struktur eines Softwaresystems. Die Zahl der False-Positives ist deshalb bei der strukturbasierten Entwurfsmustererkennung hoch.

Die vorliegende Arbeit stellt aus den zuvor genannten Gründen zunächst eine neue Bewertung der Kandidaten der Strukturanalyse vor. Des Weiteren wird die strukturbasierte Entwurfsmustererkennung durch eine Verhaltensanalyse ergänzt. Durch die Verhaltensanalyse können die Kandidaten der Strukturanalyse nochmals gefiltert werden, um so die Präzision des Gesamtprozesses zu erhöhen und verlässliche Ergebnisse zu produzieren.

Kapitel 3

Erweiterung der strukturbasierten Entwurfsmustererkennung

Dieses Kapitel gibt einen Überblick über das im Zuge dieser Arbeit entwickelte Verfahren der struktur- und verhaltensbasierten Entwurfsmustererkennung. Das in Abschnitt 2.3 vorgestellte Verfahren der strukturbasierten Entwurfsmustererkennung in FUJABA wird erweitert, um allen in Abschnitt 2.2 genannten Anforderungen zu entsprechen. Die konzeptionelle Umsetzung der bisher nicht erfüllten Anforderungen wird in diesem und in den nächsten beiden Kapiteln präsentiert.

Dazu werden im ersten Teil dieses Kapitels zunächst unscharfe Regeln und die Bewertung der Ergebnisse in der strukturbasierten Entwurfsmustererkennung motiviert. Des Weiteren werden die dadurch bedingten, syntaktischen Erweiterungen der Strukturmuster und das Verfahren zur Bewertung der Ergebnisse vorgestellt.

Der zweite Teil des Kapitels befasst sich mit der Ergänzung der strukturbasierten Entwurfsmustererkennung um eine Verhaltensanalyse. Nach der Motivation zur Verhaltensanalyse gibt der Abschnitt einen Überblick über den Prozess der kombinierten struktur- und verhaltensbasierten Entwurfsmustererkennung und erläutert das erweiterte Struktur- und Verhaltensmodell eines Softwaresystems. Die Details zur Spezifikation und Erkennung von Verhalten werden in den Kapiteln 4 und 5 behandelt.

3.1 Unscharfe Regeln und Bewertung

Eine Anforderung an eine automatische Entwurfsmustererkennung ist die Bewertung der Ergebnisse einer Entwurfsmustererkennung. Eine automatische Entwurfsmustererkennung kann wegen der unendlich vielen Implementierungs-

varianten eines Entwurfsmusters keine absoluten, präzisen Ergebnisse erzielen. Um den Reverse-Engineer jedoch bei der Sichtung und Beurteilung der Ergebnisse zu unterstützen, ist es sinnvoll, die Güte der Kandidaten, also der durch die Entwurfsmustererkennung identifizierten, potentiellen Entwurfsmusterimplementierungen zu bewerten.

Wie bereits in Abschnitt 2.4 erläutert wurde, ist das vorhandene Verfahren zur Bewertung der Kandidaten der Strukturanalyse ungeeignet. Es zieht zur Berechnung der Güte nicht die individuellen Eigenschaften des Kandidaten heran, sondern basiert auf der Beurteilung der Güte des Strukturmusters. Es wird deshalb ein Verfahren vorgestellt, das die Kandidaten individuell bewertet. Die Vertrauenswerte der Strukturmuster werden im Folgenden nicht mehr angegeben, da sie in dem neuen Verfahren nicht mehr berücksichtigt werden.

3.1.1 Motivation und Lösungsidee

In [NWW03] wird vorgestellt, wie man dem Problem der Variantenvielfalt von Entwurfsmusterimplementierungen begegnen kann. Dort wird vorgeschlagen, für verschiedene Varianten einer Entwurfsmusterimplementierung die Teile zu identifizieren, die allen Varianten gemeinsam sind. Nur dieser gemeinsame Teil wird dann zur Spezifikation des Strukturmusters verwendet. Es konnte gezeigt werden, dass dadurch die Präzision der Erkennung zwar verringert wird, die Geschwindigkeit der Analyse und damit ihre Skalierbarkeit jedoch erheblich gesteigert werden, da die Anzahl der Regeln sinkt.

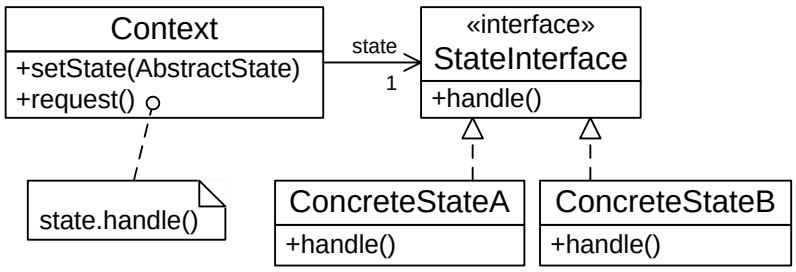

Abbildung 3.1: Eine Variante des *State*-Entwurfsmusters

Allerdings werden bei diesem Verfahren Informationen häufig nicht berücksichtigt, die gute Hinweise auf eine Entwurfsmusterimplementierung liefern. Im Falle des *State*-Entwurfsmusters existieren unter anderem die folgenden zwei Varianten. In der ersten Variante ist eine abstrakte Klasse die gemeinsame

Oberklasse aller konkreten Zustände. Dies ist die Variante, die in den bisherigen Beispielen verwendet wurde. In einer zweiten Variante nach Abbildung 3.1 wird anstatt einer abstrakten Oberklasse eine Schnittstelle vorgegeben, die von allen konkreten Zuständen implementiert werden muss. Abbildung 3.2 zeigt das zugehörige Strukturmuster[1] dieser Variante. Beide Varianten sind gültige *State*-Implementierungen.

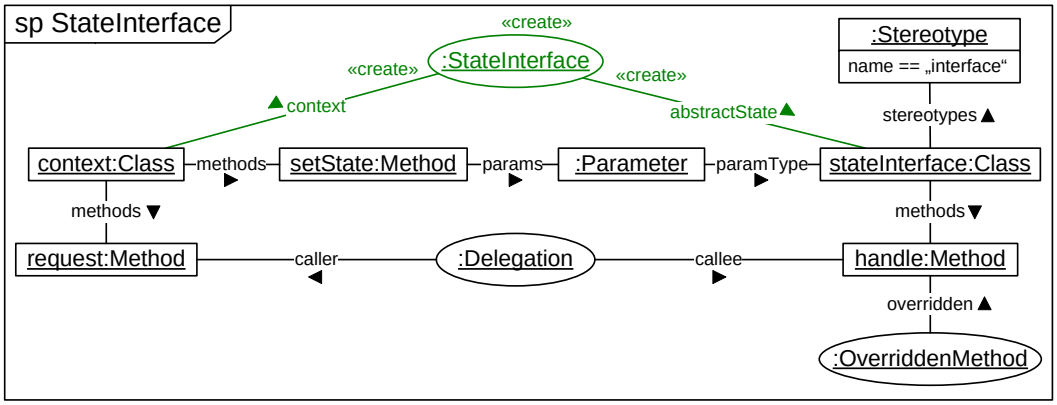

Abbildung 3.2: Das Strukturmuster des *StateInterface*-Entwurfsmusters

Wenn nach dem Ansatz aus [NWW03] die Strukturmuster der beiden Varianten zu einem gemeinsamen Strukturmuster verschmolzen werden, wird in dem daraus entstandenen Strukturmuster weder gefordert, dass die Oberklasse der konkreten Zustände abstrakt sein muss, noch dass diese Oberklasse eine Schnittstelle sein muss. Eine nicht-abstrakte Oberklasse für die konkreten Zustände ist aber in einer tatsächlichen *State*-Entwurfsmusterimplementierung eher unwahrscheinlich. Die Anzahl der mit diesem Strukturmuster erkannten False-Positives steigt gegenüber den beiden ursprünglichen, präziseren Strukturmustern.

Um diesen Nachteil wieder auszugleichen, können so genannte *unscharfe* Regeln verwendet und die durch diese Regeln identifizierten Kandidaten bewertet werden. Die Bedingung, dass zum Beispiel die Oberklasse der konkreten Zustände im *State*-Entwurfsmuster abstrakt ist, ist ein starker, zusätzlicher Hinweis auf eine *State*-Entwurfsmusterimplementierung und sollte deshalb berücksichtigt werden. Bedingungen in einem Strukturmuster sind unter anderem die Existenz von Objekten und Annotationen oder Anforderungen an

[1] Im Strukturmodell von FUJABA wird eine Schnittstelle ebenfalls als Class modelliert, ihr wird jedoch zusätzlich ein Stereotyp «interface» zugeordnet.

Attribute der Objekte.

Unscharfe Regeln enthalten zum einen die notwendigen Bedingungen, die von allen zu erkennenden Varianten eines Entwurfsmusters erfüllt werden müssen. Zum anderen enthalten unscharfe Regeln noch weitere, nicht notwendige Bedingungen, die nicht in jedem Fall durch eine potentielle Entwurfsmusterimplementierung erfüllt sein müssen, aber gute Hinweise auf eine tatsächliche Implementierung liefern. Erfüllt ein Kandidat solche nicht notwendigen Bedingungen, wird die Bewertung des Kandidaten positiv beeinflusst. Je höher die Bewertung eines Kandidaten, desto wahrscheinlicher stellt dieser Kandidat eine tatsächliche Entwurfsmusterimplementierung dar.

3.1.2 Erweiterte Syntax der Strukturmuster

In der strukturbasierten Entwurfsmustererkennung in FUJABA wurde im Zuge der vorliegenden Arbeit eine Bewertung der Kandidaten für Entwurfsmusterimplementierungen eingeführt, die den Grad der Übereinstimmung eines Kandidaten zu einem Strukturmuster beschreibt [Tra06]. Dazu wurde die Syntax der Strukturmuster erweitert, um unscharfe Regeln spezifizieren zu können.

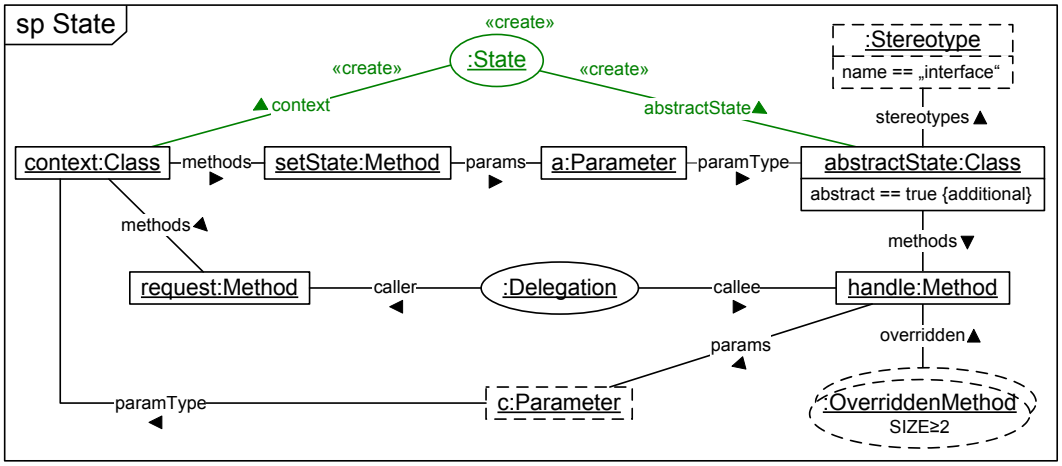

Abbildung 3.3: Das unscharfe *State*-Strukturmuster

Abbildung 3.3 zeigt das unscharfe *State*-Strukturmuster, das als Ergebnis der Verschmelzung der beiden zuvor genannten Varianten entstanden ist. In einem Strukturmuster können unter anderem Knoten oder auch Bedingungen an Attribute von Knoten als nicht notwendig gekennzeichnet werden. Nicht notwendige Knoten werden durch gestrichelte Umrandungen visualisiert. Nicht

notwendige Attributbedingungen werden durch den Zusatz {additional} gekennzeichnet.

Im unscharfen *State*-Strukturmuster wird die Attributbedingung abstract ==true des Objekts abstractState:Class als nicht notwendig gekennzeichnet. Des Weiteren wird in das neue Strukturmuster ein nicht notwendiger Knoten eingefügt, der verdeutlicht, dass es sich bei der Oberklasse der konkreten Zustände auch um eine Schnittstelle handeln kann. So werden sowohl Implementierungen gefunden, bei denen die konkreten Zustände entweder von einer abstrakten Oberklasse erben oder eine Schnittstelle implementieren. Allerdings werden auch Implementierungen mit nicht-abstrakter Oberklasse als Kandidaten identifiziert, die mit höherer Wahrscheinlichkeit False-Positives sind.

Eine weitere Eigenschaft, die nicht alle Implementierungen des *State*-Entwurfsmusters betrifft, kann nun durch die unscharfe Spezifikation modelliert werden. In [GHJV95] wird über die Delegation zwischen dem Kontext und dem aktuellen Zustand gesagt: *„A context may pass itself as an argument to the State object handling the request.“*. Das bedeutet, die handle-Methode der Zustandsklasse kann in einigen Implementierungen einen Parameter vom Typ context haben. Das Parameter-Objekt wird also im unscharfen Strukturmuster als nicht notwendig gekennzeichnet.

In Strukturmustern können Mengen von Objekten spezifiziert werden, sowohl Mengen von normalen Objekten, als auch Mengen von Annotationen. Solche Mengenknoten werden durch einen doppelten Rahmen gekennzeichnet. Während der Erkennung des Strukturmusters können beliebig viele Objekte aus der Struktur des zu untersuchenden Softwaresystems an einen Mengenknoten gebunden werden. Da Mengen grundsätzlich auch leer sein können, wird ein Mengenknoten ebenfalls durch eine gestrichelte Umrandung als nicht notwendig gekennzeichnet. Ist eine bestimmte Größe für eine Menge gefordert, so kann dies mit der Bedingung SIZE festgelegt werden. In dem Beispiel des *State*-Strukturmusters aus Abbildung 3.3 wird gefordert, dass die Methode handle:Method von mindestens zwei Methoden überschrieben wird. Das bedeutet, es existieren mindestens zwei konkrete Zustände, die von abstractState:Class erben beziehungsweise die Schnittstelle implementieren.

Die strukturbasierte Entwurfsmustererkennung wird wie in Abschnitt 2.3.6 erläutert auch zur Suche nach Implementierungen von Anti-Patterns eingesetzt. Anti-Patterns werden häufig durch unscharfe Formulierungen wie „eine Klasse mit *vielen* Methoden“ oder „eine *große* Klasse“ beschrieben. Um Strukturmuster mit solchen Eigenschaften spezifizieren zu können, wurden Metriken in die Syntax der Strukturmuster eingeführt. Mit Hilfe der Metrik LOC (*Lines Of Code*) kann zum Beispiel eine Klasse als *groß* definiert werden, wenn sie aus

mehr als 500 Zeilen Quelltext besteht. Meistens sind Begriffe wie *groß* jedoch nicht in Zahlen zu fassen. Um also keine absoluten Werte für solche Metriken verwenden zu müssen, sind so genannte *Fuzzy-Bedingungen* eingeführt worden, die stattdessen unscharfe Grenzen vorgeben.

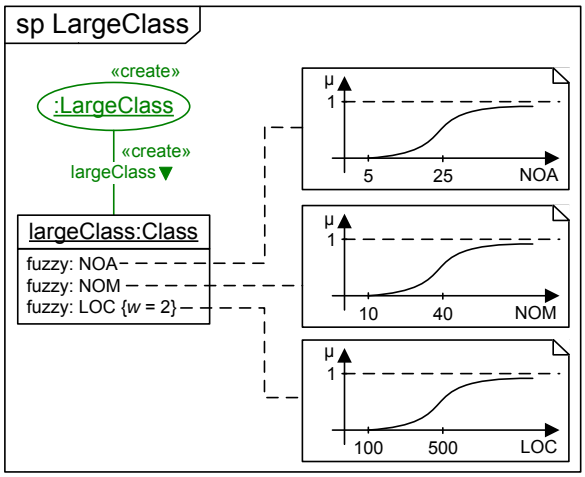

Abbildung 3.4: Das unscharfe Strukturmuster des Anti-Patterns *Large Class*

Abbildung 3.4 zeigt das unscharfe Strukturmuster des Anti-Patterns *Large Class*. Eine Klasse wird als groß definiert, wenn sie viele Attribute (*Number of Attributes* - NOA) und viele Methoden (*Number Of Methods* - NOM) hat, sowie aus sehr vielen Zeilen Quelltext besteht. Für jede der drei Metriken wird jeweils eine Fuzzy-Funktion definiert, die jeder ermittelten Metrik einen Wert zwischen 0 und 1 zuordnet. Durch die Verbindung der Metrik NOM mit einer Fuzzy-Funktion kann zum Beispiel die Bedingung „eine Klasse hat *viele* Methoden“ spezifiziert werden. Je näher der Fuzzy-Wert an 1 liegt, desto eher trifft die Aussage zu. Liegen die Fuzzy-Werte aller drei Metrik-Bedingungen nahe bei 1, so kann davon ausgegangen werden, dass eine Implementierung des Anti-Patterns Large Class vorliegt. Die ermittelten Fuzzy-Werte fließen zudem in die Bewertung des Kandidaten ein.

Bestimmte Bedingungen in Strukturmustern sind wichtiger als andere. Um dies bei der Bewertung eines Kandidaten zu berücksichtigen, sind Gewichte für Bedingungen eingeführt worden. In Abbildung 3.4 ist zum Beispiel die Metrik-Bedingung LOC mit dem Gewicht 2 (w=2) versehen. Wird das Gewicht nicht explizit angegeben, so erhält die Bedingung das Gewicht 1.

3.1.3 Bewertung der Ergebnisse

Um eine potentielle Entwurfsmusterimplementierung zu bewerten, wird der Grad der Übereinstimmung des Kandidaten zum Strukturmuster berechnet. Die verwendete Bewertungsfunktion lässt sich vereinfacht folgendermaßen darstellen:

$$Bewertung(a, sp) = \frac{\sum_{b \in Bedingungen(sp)} Erfülltheitsgrad(b, a) \cdot w_b}{\sum_{b \in Bedingungen(sp)} w_b}$$

Die Bewertung einer Annotation a, die einen Kandidaten repräsentiert, und dem zugehörigen Strukturmuster sp errechnet sich aus dem Verhältnis der Summe der gewichteten Erfülltheitsgrade jeder Bedingung zur Summe der Gewichte w_b aller Bedingungen. Der Erfülltheitsgrad ist für jedes Syntaxelement der Strukturmuster, also jeder Art von Bedingung, separat definiert.

Der Erfülltheitsgrad einer notwendigen Bedingung wie dem Objekt context:Class aus Abbildung 3.3 ist 1. Ein einfaches, nicht notwendiges Objekt wie das Objekt :Stereotype erhält den Erfülltheitsgrad 1, wenn es in dem Kandidaten vorhanden ist, sonst 0. Eine nicht notwendige Attributbedingung wie abstract==true hat ebenfalls den Erfülltheitsgrad 1, wenn sie erfüllt ist, sonst 0. Wird eine Annotation, wie zum Beispiel :Delegation im *State*-Strukturmuster, verwendet, so ist ihr Erfülltheitsgrad durch ihre Bewertung definiert. Die Bewertung einer Annotation wird also rekursiv über die Annotationen berechnet, von denen sie abhängig ist.

Bei Mengenknoten ist der Erfülltheitsgrad abhängig von der Anzahl der Objekte, die der Menge zugeordnet sind. Der Erfülltheitsgrad wird berechnet, indem die Anzahl der Objekte auf eine streng monoton steigende, asymptotisch gegen 1 strebende Funktion abgebildet wird. Je mehr Objekte einer Menge zugeordnet sind, desto höher ist der Erfülltheitsgrad und somit die Bewertung einer Annotation. Bei Mengenknoten, die aus Annotationen bestehen wie :OverriddenMethod, werden zusätzlich zur Anzahl die Bewertungen der einzelnen Annotationen berücksichtigt.

Die streng monoton steigende Funktion für Mengenknoten ist nur eine von vielen möglichen Funktionen. Durch die separate Definition einer Funktion für jedes Syntaxelement zur Berechnung des Erfülltheitsgrades lassen sich die Funktionen leicht austauschen, sollten sie sich in der Praxis als nicht tauglich erweisen. Vorstellbar wäre für Mengenknoten zum Beispiel auch die Gauß-Funktion.

Für Fuzzy-Bedingungen wie die Metrik *Number Of Methods* im Beispiel des Anti-Patterns *Large Class* in Abbildung 3.4 wird die zugehörige Fuzzy-

Funktion als Erfülltheitsgrad verwendet. Auf eine detailliertere Beschreibung der Bewertungsfunktion wird in diesem Kontext allerdings verzichtet. Für weitere Informationen sei auf [Tra06] verwiesen.

Die aus der strukturbasierten Entwurfsmustererkennung resultierenden Annotationen werden mit ihrer Bewertung in einem Klassendiagramm visualisiert. Annotationen, deren Bewertung einen vorgegebenen Schwellwert nicht überschreiten, können aus dem Gesamtergebnis ausgeblendet werden, um die Übersichtlichkeit zu erhöhen.

3.2 Verhaltensbasierte Entwurfsmustererkennung

Eine weitere, bisher nur sehr ungenügend erfüllte Anforderung an eine automatische Entwurfsmustererkennung ist die Produktion möglichst präziser Ergebnisse. Zur Analyse sollten deshalb möglichst viele, charakteristische Eigenschaften der zu suchenden Entwurfsmuster heran gezogen werden. Zu diesen Eigenschaften zählt jedoch nicht nur ihre Struktur, sondern auch ihr Verhalten. Im Folgenden wird deshalb ein Verfahren skizziert, mit dem auch das Verhalten eines Entwurfsmusters spezifiziert und die Übereinstimmung eines Kandidaten mit dieser Spezifikation zur Laufzeit des Softwaresystems überprüft werden kann.

3.2.1 Motivation und Lösungsidee

Das in Abschnitt 2.3 vorgestellte Verfahren zur Entwurfsmustererkennung beschränkt sich auf strukturelle Informationen. Viele Entwurfsmuster definieren sich jedoch nicht nur durch ihre Struktur, sondern in hohem Maße auch durch ihr Verhalten. Das *State*-Entwurfsmuster ist ein gutes Beispiel dafür. Es erlaubt einem Objekt, sein Verhalten abhängig von seinem Zustand zu verändern.

In Abbildung 3.5 ist die Struktur des *State*-Entwurfsmusters zu sehen. Das Objekt, das zur Laufzeit sein Verhalten ändern kann, ist vom Typ Context. Die Klasse Context assoziiert eine abstrakte Oberklasse AbstractState. Diese Klasse gibt eine gemeinsame Schnittstelle für konkrete Klassen vor, die verschiedene Zustände implementieren. Die hier verwendeten konkreten Zustände ConcreteStateA und ConcreteStateB sind nur Beispiele. Das *State*-Entwurfsmuster ist nicht nur auf zwei konkrete Zustände beschränkt, sondern kann prinzipiell beliebig viele, verschiedene Zustände verwalten.

Das Verhalten des *State*-Entwurfsmusters ist in [GHJV95] wie folgt beschrieben:

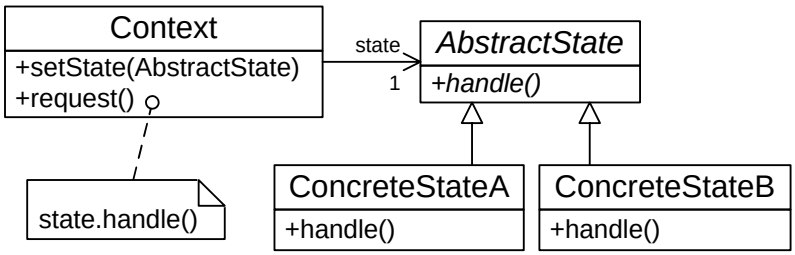

Abbildung 3.5: Die Struktur des *State*-Entwurfsmusters

> „*Context delegates state-specific requests to the current ConcreteState object. [...] Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly. Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.*“.

Zur Laufzeit referenziert also ein Objekt vom Typ Context ein konkretes Zustandsobjekt. Wird eine Anfrage an das Context-Objekt mit Hilfe der Methode request() gestellt, so gibt es die Anfrage an sein aktuelles Zustandsobjekt durch Aufruf der Methode handle() weiter. Die Anfrage kann so abhängig vom Zustand des Objekts bearbeitet werden. Zustandsänderungen werden durch Austausch des aktuellen Zustandsobjekts entweder vom Context-Objekt selber oder von dem aktuellen Zustandsobjekt vorgenommen.

Der Vorteil dieser Struktur ist, dass sie sehr wartungsfreundlich ist. Sie kann relativ einfach durch neue Zustände erweitert werden, ohne große Änderungen an den anderen beteiligten Klassen durchführen zu müssen. Des Weiteren sind die Algorithmen, die in einem bestimmten Zustand ausgeführt werden, jeweils in einer Zustandsklasse gekapselt, was die Übersichtlichkeit erhöht und damit auch die Wartung erleichtert.

Die Struktur des *State*-Entwurfsmusters ist allerdings ein Beispiel für eine in objektorientierten Architekturen sehr häufig vorkommende Konstruktion. Die Grundstruktur besteht aus einer Klasse, die eine abstrakte Klasse assoziiert, von der mehrere konkrete Klassen erben. Das Implementieren oder Überschreiben einer Methode aus einer abstrakten Oberklasse durch konkrete Klassen ist eines der Hauptmerkmale objektorientierter Programmierung. Auch die Delegation von Methodenaufrufen an andere referenzierte Objekte ist häufig anzutreffen. Die Wahrscheinlichkeit, eine solche Struktur in einer objektorientierten

Architektur vorzufinden, ist also relativ hoch.

Sollten aber solche Konstrukte, die zwar in ihrer Struktur mit einem Entwurfsmuster übereinstimmen, sich aber zur Laufzeit anders verhalten, als Implementierungen dieser Entwurfsmuster erkannt werden? Ist eine potentielle *State*-Implementierung, die zur Laufzeit niemals ihr mutmaßliches Zustandsobjekt austauscht, wirklich eine Implementierung eines *State*-Entwurfsmusters? Die zentrale These, die also mit dieser Arbeit belegt werden soll, lautet:

> Ein Konstrukt, welches die Struktur eines bestimmten Entwurfsmusters hat und sich auch wie ein solches Entwurfsmuster verhält, ist mit sehr hoher Wahrscheinlichkeit eine tatsächliche Implementierung dieses Entwurfsmusters.

Es sollte also eine Laufzeitanalyse des Verhaltens durchgeführt werden, um so möglichst viele *False-Positives*, also fälschlicherweise erkannte Entwurfsmusterimplementierungen, auszuschließen.

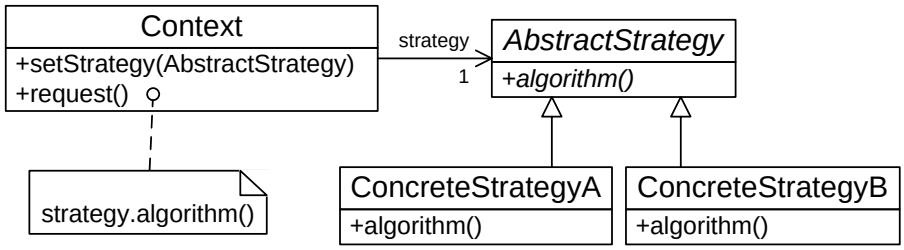

Abbildung 3.6: Die Struktur des *Strategy*-Entwurfsmusters

Es gibt noch einen weiteren Grund, der zeigt, dass eine Strukturanalyse alleine nicht ausreichend ist. Es existieren Paare von Entwurfsmustern, die sich in ihrer Struktur sehr ähnlich sind oder sogar vollständig übereinstimmen. Die Struktur des *Strategy*-Entwurfsmusters (Abbildung 3.6) stimmt zum Beispiel vollständig mit der Struktur des *State*-Entwurfsmusters überein. Das führt bei der strukturbasierten Erkennung dazu, dass dasselbe Konstrukt sowohl als *State*-, als auch als *Strategy*-Entwurfsmusterimplementierung identifiziert wird. Das Verhalten des *Strategy*-Entwurfsmusters ist jedoch im Gegensatz zum *State*-Entwurfsmuster in [GHJV95] beschrieben durch:

> „*A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively.*“.

Auch hier kann ein Objekt zur Laufzeit Algorithmen durch unterschiedliche Strategien ausführen lassen. Der Austausch einer Strategie wird jedoch nicht von dem Objekt selber oder der aktuellen Strategie veranlasst, sondern von außen. Zur Laufzeit kann also durch Beobachtung des Verhaltens zwischen den Entwurfsmustern *State* und *Strategy* unterschieden werden. Weitere Beispiele für Paare ähnlicher Entwurfsmuster sind *Decorator* und *Chain of Responsibility*, *Strategy* und *Bridge*, *State* und *Composite* oder *Composite* und *Chain of Responsibility*.

Das Ignorieren des Verhaltens bei der Entwurfsmustererkennung führt zu höheren Fehlerraten, wie an dem vorangehenden Beispiel zu erkennen ist. Zur präziseren Identifikation von Entwurfsmusterimplementierungen sollte also auch ihr Verhalten untersucht werden.

Das Verhalten von Software wird bei imperativen Programmiersprachen durch Variablenbelegungen und deren Änderungen sowie durch Prozedur- oder Methodenaufrufe bestimmt. In der bisherigen Entwurfsmustererkennung in FUJABA können auf Basis der abstrakten Syntaxbäume der Methoden sehr rudimentäre Daten- und Kontrollflussanalysen durchgeführt oder potentielle Methodenaufrufe identifiziert werden. Statische Analysen können jedoch nicht feststellen, ob potentielle Methodenaufrufe tatsächlich zur Laufzeit ausgeführt werden. Die konkreten, aufzurufenden Methoden werden in objektorientierten Programmiersprachen mit Polymorphismus und dynamischer Methodenbindung sogar erst zur Laufzeit festgelegt.

Die Überwachung von Variablenbelegungen und Methodenaufrufen zur Laufzeit des zu untersuchenden Softwaresystems gestaltet sich jedoch sehr schwierig aufgrund der zu großen Datenmenge[2], die dabei anfällt. Entwurfsmuster beschreiben allerdings in der Regel Kollaborationen mehrerer Objekte. Das Verhalten eines Entwurfsmusters wird also im Wesentlichen bestimmt durch die Art und Reihenfolge von Nachrichten in Form von Methodenaufrufen, die zwischen diesen Objekten ausgetauscht werden. Zur Analyse des Verhaltens eines Programms ist also eine Beschränkung auf die Methodenaufrufe zur Reduzierung der anfallenden Datenmenge möglich.

Der Reverse-Engineer sollte durch die automatische Erkennung auf potentielle Entwurfsmusterimplementierungen hingewiesen werden. Trotzdem sollten aber Konstrukte, deren Struktur und deren Verhalten mit dem des Entwurfsmusters übereinstimmen, deutlich von anderen Konstrukten, die nur in ihrer Struktur mit dem Entwurfsmuster übereinstimmen, unterschieden werden

[2]Bill Lewis berichtet in seiner Arbeit, bei der sowohl Variablenbelegungen als auch Methodenaufrufe aufgezeichnet werden, von 100MB Daten pro Sekunde [Lew03].

können. Rein strukturbasierte Analysen sind nicht in der Lage, solche Unterscheidungen durchzuführen. Um sicherere Aussagen machen zu können, ist auch eine Analyse des Verhaltens notwendig.

Da strukturelle Eigenschaften von Entwurfsmustern mit Hilfe statischer Analysen sehr effizient erkennbar sind, ist eine Kombination aus statischer und dynamischer Analyse sinnvoll [Wen03]. Die bei der dynamischen Analyse anfallende Datenmenge kann durch diesen Ansatz sogar noch weiter reduziert werden. Eine vorher durchgeführte strukturbasierte Entwurfsmustererkennung identifiziert potentielle Entwurfsmusterimplementierungen. In der anschließenden dynamischen Analyse kann dadurch die Überwachung von Methodenaufrufen auf Objekte der potentiellen Entwurfsmusterimplementierungen eingeschränkt werden.

3.2.2 Struktur- und verhaltensbasierter Erkennungsprozess

Das bisherige Verfahren der Entwurfsmustererkennung in FUJABA basiert im Wesentlichen auf der Struktur der Entwurfsmuster. Ihr Verhalten wird nur sehr rudimentär durch Identifikation potentieller Methodenaufrufe auf den abstrakten Syntaxbäumen der Methoden analysiert, wie an dem Beispiel des *Delegation*-Hilfsmusters aus Abschnitt 2.3.2 zu sehen ist. Allerdings ist das Verfahren sehr gut dazu geeignet, durch eine dynamische Analyse des Verhaltens ergänzt zu werden. In Abbildung 3.7 ist der in [Wen03] vorgestellte, erweiterte Prozess der struktur- und verhaltensbasierten Entwurfsmustererkennung zu sehen.

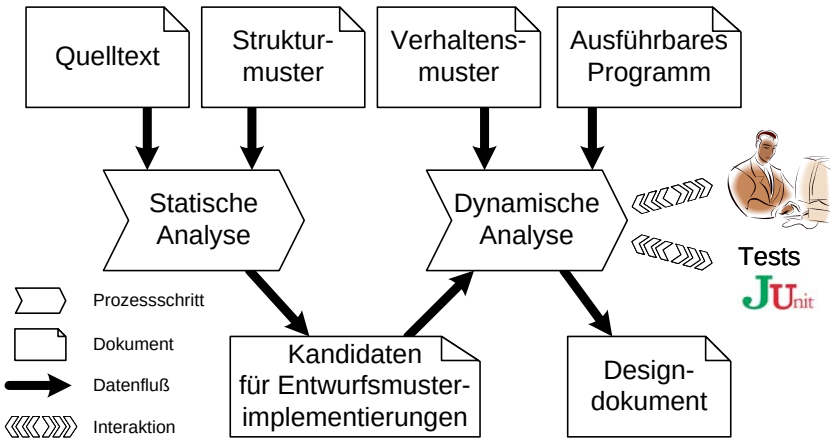

Abbildung 3.7: Der kombinierte Erkennungsprozess

Die statische Analyse des in Abbildung 3.7 gezeigten Prozesses ist eine vereinfachte Darstellung des Prozesses aus Abschnitt 2.3.4. Dieser Prozess der strukturbasierten Entwurfsmustererkennung wird unverändert als Teilprozess übernommen. Eingaben dieses Teilprozesses sind der Quelltext des zu untersuchenden Softwaresystems und ein Katalog von Strukturmustern. Das Ergebnis des Teilprozesses sind potentielle Entwurfsmusterimplementierungen, die so genannten *Kandidaten*.

Diese Kandidaten können in der dynamischen Analyse dazu verwendet werden, den Suchraum einzuschränken. Es müssen nur Methoden der an den Kandidaten beteiligten Klassen zur Laufzeit des Programms überwacht werden. Weitere Eingaben der dynamischen Analyse sind das zu untersuchende, ausführbare Programm und so genannte *Verhaltensmuster*. Das Programm wird in der dynamischen Analyse entweder durch automatische Tests oder manuell durch einen Benutzer ausgeführt. Die beobachteten Methodenaufrufe der Kandidaten können entweder aufgezeichnet und nach Beendigung des Programms mit den Verhaltensmustern verglichen werden, oder bereits zur Laufzeit ausgewertet werden.

Der Begriff Verhaltensmuster wird im Folgenden analog zu dem Begriff des Strukturmusters verwendet. Während Strukturmuster zur Spezifikation der strukturellen Anteile eines Entwurfsmusters dienen, werden Verhaltensmuster zur Spezifikation des Verhaltens eines Entwurfsmusters verwendet. In diesem Zusammenhang sei besonders darauf hingewiesen, dass der Begriff Verhaltensmuster nicht die Kategorie[3] der Entwurfsmuster bezeichnet, die hauptsächlich Kollaborationen und Verhalten von Objekten beschreiben und in der englischen Literatur häufig mit *Behavioral Design Patterns* bezeichnet werden.

Als Ergebnis des Gesamtprozesses erhält der Reverse-Engineer ein Designdokument in Form von annotierten UML-Klassendiagrammen. Die Klassendiagramme repräsentieren das untersuchte Softwaresystem und sind angereichert mit Informationen über die gefundenen Entwurfsmusterimplementierungen. In den Annotationen der Entwurfsmusterimplementierungen werden Bewertungen angegeben, die ausdrücken, inwieweit die Entwurfsmusterimplementierung mit dem Strukturmuster beziehungsweise dem Verhaltensmuster des Entwurfsmusters übereinstimmt. Die Bewertungen stammen aus der statischen (siehe Abschnitt 3.1.3) und der dynamischen Analyse. Hohe Bewertungen bedeuten dabei eine hohe Übereinstimmung der Entwurfsmusterimplementierung mit dem Strukturmuster beziehungsweise dem Verhaltensmuster. Der Reverse-Engineer kann so zum Beispiel Annotationen mit geringer Bewertung aus sei-

[3] siehe Tabelle 2.1, Seite 12

ner Sicht auf das zu untersuchende Softwaresystem entfernen und sich auf die wahrscheinlichsten Entwurfsmusterimplementierungen konzentrieren.

Die in dieser Arbeit entwickelte automatische struktur- und verhaltensbasierte Entwurfsmustererkennung erhebt nicht den Anspruch, absolute Aussagen über das zu untersuchende Softwaresystem zu machen. Der Reverse-Engineer soll durch dieses Verfahren in seinen Bemühungen, das Softwaresystem zu verstehen, deutlich unterstützt werden. Die letztendliche Entscheidung, ob eine Entwurfsmusterimplementierung tatsächlich vorliegt, kann und soll dem Reverse-Engineer jedoch nicht abgenommen werden.

3.2.3 Verhaltensmodell eines Softwaresystems

Das in der strukturbasierten Entwurfsmustererkennung verwendete Strukturmodell kann nicht zur Analyse des Verhaltens herangezogen werden. Aus diesem Grund wird analog zum Strukturmodell ein Verhaltensmodell für die verhaltensbasierte Entwurfsmustererkennung eingeführt.

Das Verhaltensmodell hängt genau wie das Strukturmodell von der Anwendungsdomäne der Entwurfsmustererkennung ab. Das Verhaltensmodell für objektorientierte Softwaresysteme beschreibt eine Sequenz von Methodenaufrufen, die zur Laufzeit beobachtet und aufgezeichnet wurden. Für andere Softwaresysteme, die in funktionalen oder regelbasierten Sprachen geschrieben wurden, könnte das Verhaltensmodell auch eine Sequenz von Funktionsaufrufen oder Regelausführungen beschreiben.

Eine Sequenz von Methodenaufrufen wird auch als *Trace* bezeichnet. Die Ordnung innerhalb eines Traces ist durch die Reihenfolge der Methodenaufrufe zur Laufzeit vorgegeben. Traces müssen nicht die Gesamtheit aller zur Laufzeit ausgeführten Methodenaufrufe umfassen. Sie können beliebige Sequenzen zwischen zweier Zeitpunkte ausgeführte Methodenaufrufe sein. Dabei muss ein Trace nicht einmal alle zwischen den beiden Zeitpunkten ausgeführten Methodenaufrufe enthalten, er kann auch aus einer Teilmenge bestehen. Die einzige Voraussetzung ist, dass die Ordnung der Methodenaufrufe erhalten bleibt.

Im Gegensatz zur Struktur repräsentiert ein Trace daher nicht das gesamte mögliche Verhalten eines Softwaresystems. Es handelt sich immer nur um eine einzelne Sequenz aus theoretisch unendlich vielen möglichen Sequenzen. Traces sind abhängig von der Eingabe, mit der das Softwaresystem gestartet wird, und von den Methoden, die beobachtet werden.

In Abbildung 3.8 ist ein Trace, der zur Laufzeit eines Softwaresystems aufgezeichnet wurde, als Sequenzdiagramm visualisiert. Das Beispiel zeigt einen Ausschnitt aus der Ausführung des Mediaplayers aus Abbildung 2.3, Seite 20.

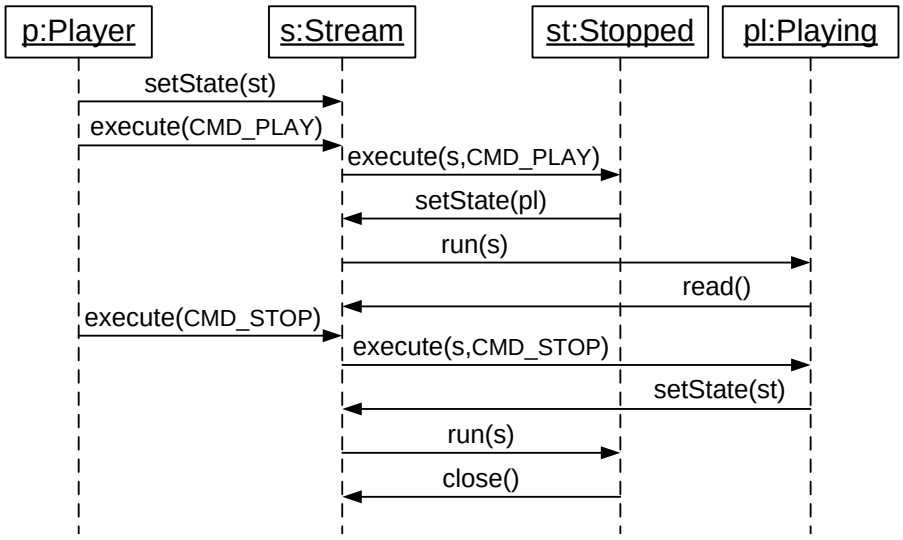

Abbildung 3.8: Trace eines Programmlaufs

Die Gesamtheit der zur Laufzeit aufgezeichneten Methodenaufrufe werden in einem *Tracegraphen* zusammengefasst. Abbildung 3.9 zeigt das Modell eines Tracegraphen. Die Wurzel eines Tracegraphen ist ein Prozess, repräsentiert durch die Klasse Process. Dieser enthält eine Sequenz von Methodenaufrufen (MethodCall). Zu jedem Methodenaufruf gibt es eine Instanz, die den Aufruf ausführt (caller), und eine Instanz, die den Methodenaufruf empfängt (callee). Beide Instanzen sind vom Typ Instance. Des Weiteren können zu jedem Methodenaufruf beliebig viele Argumente (Argument) gehören. Die Klassen des Tracegraphen gehören zum Paket Behavior.

In einem Tracegraphen ist das Verhalten eines Softwaresystems während eines konkreten Programmlaufs aufgezeichnet. Die Instanzen, die in diesem Tracegraphen vorkommen, sind Instanzen der Klassen, die in der Struktur des Softwaresystems beschrieben werden. Des Weiteren sind die Methodenaufrufe des Tracegraphen Aufrufe der Methoden, die ebenfalls in der Struktur beschrieben werden. Diese semantische Typbeziehung wird zwischen dem Verhaltensmodell und dem Strukturmodell durch Referenzen ausgedrückt. Die Klassen der Instanzen, Methodenaufrufe und Argumente des Tracegraphen besitzen jeweils eine Referenz zu Klassen des Strukturmodells aus Abbildung 2.2 (Seite 19). So referenziert Instance die Klasse Structure::Class, MethodCall referenziert die Klasse Structure::Method und Argument die Klasse Structure::Parameter.

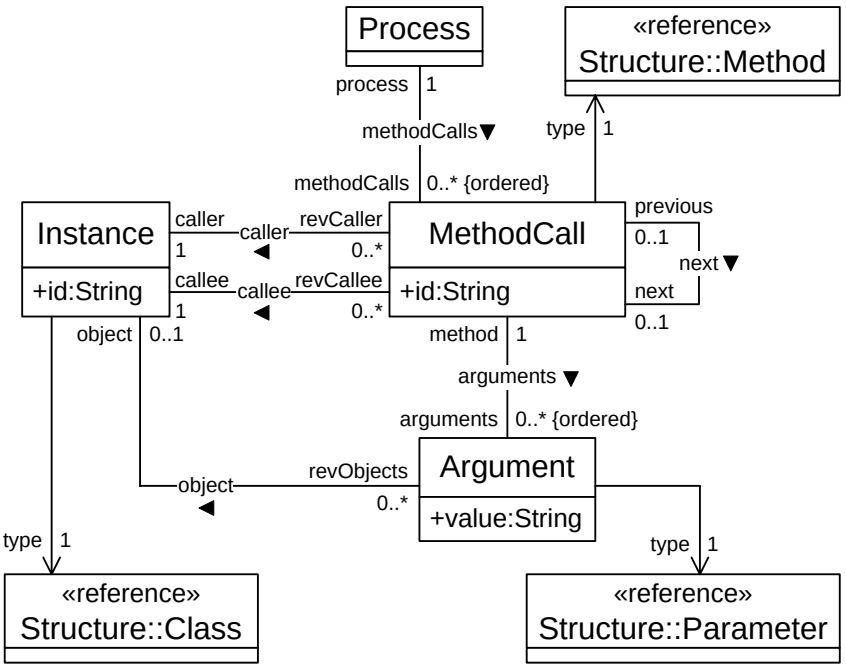

Abbildung 3.9: Das Modell des Tracegraphen, Paket Behavior

3.2.4 Überblick

Um einen besseren Überblick über das erweiterte Struktur- und Verhaltensmodell eines Softwaresystems zu bekommen, sind die beiden, im vorigen Abschnitt eingeführten Diagramme in das Schema aus Metamodellen, Modellen und Instanzen aus Abbildung 2.13 eingeordnet worden. Das Ergebnis ist in Abbildung 3.10 dargestellt. Das Verhaltensmodell, also das Modell des Tracegraphen gehört zur mittleren Abstraktionsschicht, zu der auch das Strukturmodell gehört. Die semantische Beziehung zwischen Struktur- und Verhaltensmodell ist durch Referenzen zwischen den beiden Modellen spezifiziert.

Der Tracegraph, also das konkrete, zur Laufzeit beobachtete Verhalten, gehört zur Instanzschicht auf unterster Ebene. Da Verhaltensmodelle genau wie Strukturmodelle als Klassendiagramme spezifiziert werden, nutzen sie ein gemeinsames Metamodell.

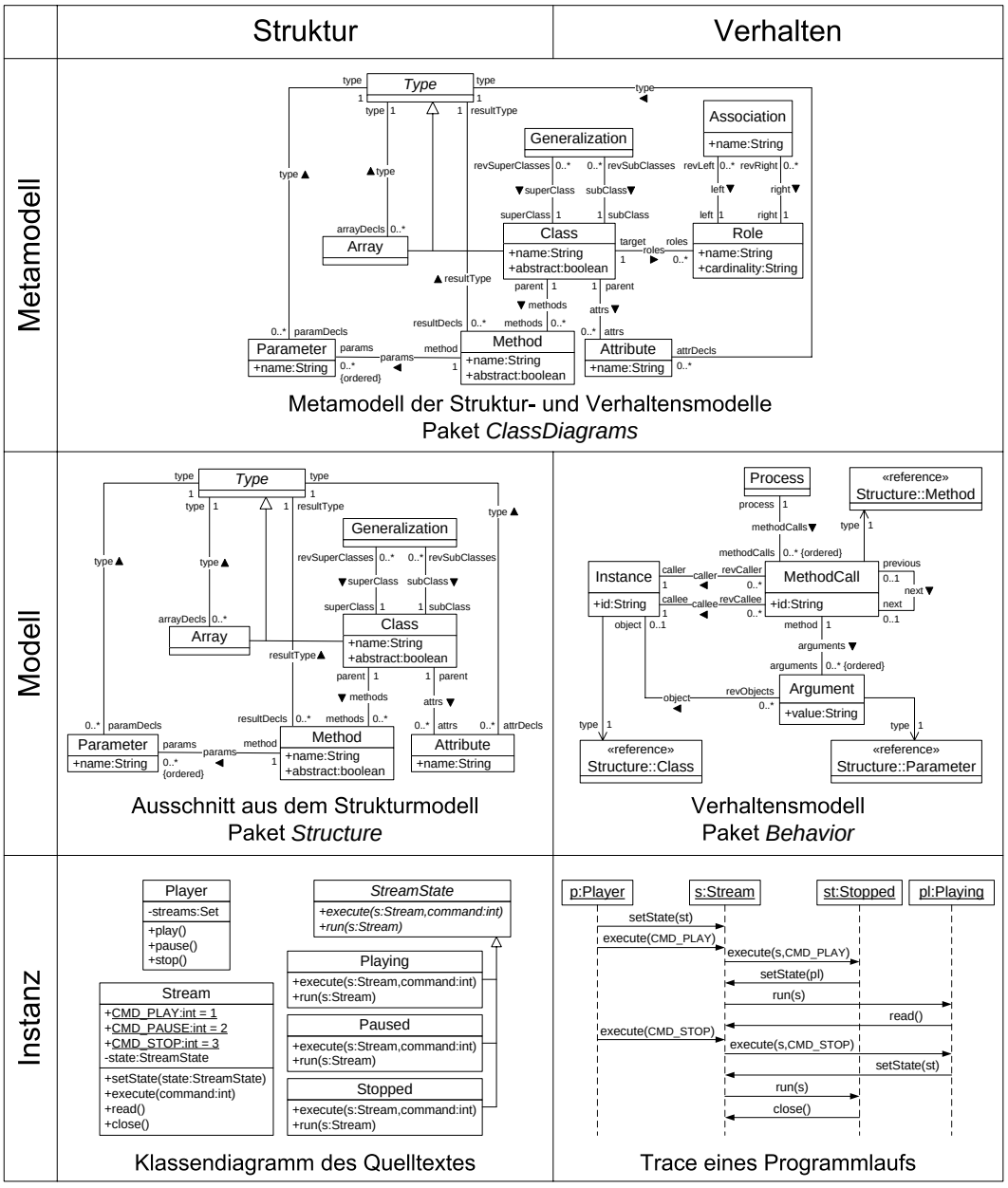

Abbildung 3.10: Die Modellierung von Struktur und Verhalten eines Softwaresystems

3.3 Zusammenfassung

Die in Kapitel 2.3 vorgestellte strukturbasierte Entwurfsmustererkennung in FUJABA erfüllt von den in Abschnitt 2.2 definierten Anforderungen bisher nur die Skalierbarkeit und die Anpassbarkeit vollständig. In Kapitel 2 konnte dies gezeigt werden. Die Bewertung der Ergebnisse dagegen war in der bisherigen Umsetzung nicht praxistauglich. Auch die Präzision des Ansatzes war wegen der fehlenden Analyse des Verhaltens der potentiellen Entwurfsmusterimplementierungen gering.

In diesem Kapitel wurde eine Bewertung der Ergebnisse der strukturbasierten Erkennung vorgestellt, die im Gegensatz zur bisherigen Lösung die individuellen Entwurfsmusterimplementierungen berücksichtigt. Dazu ist die Syntax der Strukturmuster erweitert worden. Es ist nun möglich, Bedingungen in die Strukturmuster aufzunehmen, die nicht von allen Entwurfsmusterimplementierungen erfüllt werden müssen. Sollten sie jedoch von einem Kandidaten erfüllt werden, so liefern sie zusätzliche Hinweise darauf, dass eine tatsächliche Entwurfsmusterimplementierung vorliegt. Diese Hinweise resultieren in einer höheren und damit besseren Bewertung des Kandidaten. Kandidaten, die nur die notwendigen Bedingungen erfüllen, erhalten eine niedrigere und damit schlechtere Bewertung.

Baut ein Strukturmuster auf bereits gefundenen Annotationen auf, so fließen die Bewertungen der verwendeten Annotationen in die Bewertung der Annotation des Strukturmusters ein. Somit können Annotationen, die „schlechte“ Informationen verwenden, keine guten Bewertungen erhalten.

Die Bewertungen der Ergebnisse können vom Reverse-Engineer dazu verwendet werden, um unsichere Informationen, das heißt Annotationen mit geringer Bewertung, aus dem Gesamtergebnis auszublenden. Der Benutzer kann sich so leichter auf die relevanten Informationen konzentrieren.

Die Praxistauglichkeit der Bewertung wurde allerdings bisher nicht überprüft. Kritisch ist vor allem die Bewertung der Mengen in Abhängigkeit von ihrer Größe. Bei Mengenknoten aus Annotationen kann es vorkommen, dass eine Menge von vielen, niedrig bewerteten Annotationen eine höhere Bewertung erhält, als eine Menge von wenigen, aber sehr hoch bewerteten Annotationen. Aus diesem Grund ist es notwendig, die Bewertung in einer größeren Studie in der Praxis zu überprüfen. Das kann jedoch im Rahmen dieser Arbeit nicht geleistet werden. Das Konzept, die Bewertungsfunktion eines Strukturmusters auf die Bewertungsfunktionen der verschiedenen Syntaxelemente eines Strukturmusters zurückzuführen, erleichtert jedoch zum Beispiel einen Austausch der Bewertungsfunktion der Mengenknoten, sollte sich diese Bewertungsfunk-

tion in der Praxis als nicht tauglich erweisen.

Des Weiteren wurde in diesem Kapitel ein Konzept vorgestellt, um die Präzision des bisherigen Ansatzes zu verbessern. Entwurfsmuster werden nicht nur durch ihre Struktur, sondern auch durch ihr Verhalten zur Laufzeit charakterisiert. Das Verhalten wurde bisher nicht berücksichtigt. Die Struktur sehr vieler Entwurfsmuster basiert jedoch auf in der objektorientierten Programmierung häufig verwendeten Konzepten, wie zum Beispiel Vererbung und dynamische Methodenbindung sowie Delegation von Methodenaufrufen. Viele Entwurfsmuster ähneln sich zudem in ihrer Struktur, was eine Unterscheidung durch eine statische Analyse erschwert, wenn nicht sogar unmöglich macht. Somit führt eine rein strukturbasierte Analyse zu einer hohen Zahl von False-Positives.

Das in dieser Arbeit vorgestellte Konzept besteht deshalb aus einer Kombination aus strukturbasierter und verhaltensbasierter Entwurfsmustererkennung, um so die Vorteile beider Techniken nutzen zu können. Die Datenmengen, die zur Laufzeit eines Softwaresystems anfallen und von einer verhaltensbasierten Entwurfsmustererkennung analysiert werden müssen, sind enorm. Eine vorangehende Strukturanalyse kann jedoch diese Datenmengen erheblich verringern, indem die Verhaltensanalyse auf die in der Strukturanalyse identifizierten Kandidaten eingeschränkt wird. In dem kombinierten Prozess aus Struktur- und Verhaltensanalyse wird also das Ergebnis der Strukturanalyse durch die Verhaltensanalyse verfeinert, um so eine möglichst hohe Präzision zu erreichen.

Das bereits vorhandene Strukturmodell eines zu untersuchenden Softwaresystems ist nicht ausreichend zur struktur- und verhaltensbasierten Entwurfsmustererkennung. Es wird um ein Verhaltensmodell erweitert, das Sequenzen von Methodenaufrufen repräsentiert. Sowohl das Struktur- als auch das Verhaltensmodell ist für die Analyse objektorientierter Software vorgesehen. Diese Modelle können jedoch je nach Anwendungsdomäne gegen speziellere Modelle ausgetauscht werden.

Kapitel 4

Verhaltensspezifikation

Das Thema dieses Kapitels ist die formale Spezifikation des Verhaltens der Entwurfsmuster. Dazu werden im ersten Teil des Kapitels zunächst informell Sequenzdiagramme nach UML 2.0 als Spezifikationssprache eingeführt, auf deren Basis so genannte *Verhaltensmuster* definiert werden. Die formale Spezifikation der Syntax der Verhaltensmuster ist Inhalt des zweiten Teils des Kapitels. Im dritten Teil folgt eine informelle Beschreibung der Semantik der Verhaltensmuster. Die formale Spezifikation der Semantik ist Inhalt des vierten Teils des Kapitels. Die Semantik wird durch eine Transformation der Sequenzdiagramme in endliche Automaten definiert. Am Ende des Kapitels werden schließlich die Verhaltensmuster in das bereits bekannte Schema aus Metamodellen, Modellen und Implementierungen eingeordnet.

4.1 Verhaltensmuster

Das Verhalten der Entwurfsmuster ist üblicherweise nicht formal spezifiziert. Wie bereits in Abschnitt 3.2.1 zitiert, wird zum Beispiel zur Spezifikation des Verhaltens des *State*-Entwurfsmusters in [GHJV95] einfacher Text gewählt. Zur automatischen Erkennung durch Algorithmen ist diese Form der Spezifikation jedoch nicht ausreichend. Der Reverse-Engineer muss also das Verhalten eines Entwurfsmusters ebenso wie die Struktur formal spezifizieren. Erst dann kann ein Algorithmus die Spezifikation verarbeiten. Analog zu den Strukturmustern aus 2.3.2 werden deshalb Verhaltensmuster eingeführt.

In Abschnitt 3.2.1 wurde erläutert, dass das Verhalten von Entwurfsmustern mit Hilfe von Sequenzen von Methodenaufrufen beschrieben werden kann. In der Softwaretechnik haben sich dazu die so genannten *Sequenzdiagramme* etabliert. Sequenzdiagramme gibt es in verschiedenen Ausprägungen. Die beiden gebräuchlichsten Spezifikationen sind die *Message Sequence Charts* (MSCs)

der International Telecommunication Union [Int99], und die Sequenzdiagramme nach UML [Obj].

Die Spezifikationssprache für Verhaltensmuster soll im Wesentlichen zwei Kriterien genügen. Zum einen soll mit ihr das Verhalten der Entwurfsmuster formal spezifiziert werden können, so dass die Spezifikationen algorithmisch verarbeitet werden können. Zum anderen soll sie für den Reverse-Engineer intuitiv zugänglich und leicht zu erlernen sein.

Sowohl MSCs, als auch Sequenzdiagramme nach UML 2.0 erfüllen beide Kriterien. Die Verwendung der UML in der strukturbasierten Entwurfsmustererkennung spricht jedoch aus Gründen der Konsistenz für UML-Sequenzdiagramme als Spezifikationssprache für Verhaltensmuster [Wen04]. Aus den Sequenzdiagrammen können automatisch Formalismen gewonnen werden, die Verhaltensmuster präzise beschreiben und zur algorithmischen Verarbeitung geeignet sind. Dazu werden die Sequenzdiagramme zum Beispiel in endliche Automaten übersetzt. Da Sequenzdiagramme häufig im Forward Engineering zur Quelltextgenerierung oder auch zu Dokumentationszwecken eingesetzt werden, sind sie unter Softwareentwicklern allgemein bekannt. Ihre graphische Form macht sie außerdem leicht zugänglich und schnell lesbar.

4.1.1 Formalisierung durch Sequenzdiagramme

Sequenzdiagramme spezifizieren Sequenzen von Nachrichten, die synchron oder asynchron zwischen zwei Objekten versendet werden. Seit der Version 2.0 der UML stehen jedoch nicht nur einfache Sequenzen von Nachrichten zur Verfügung, sondern auch verschiedene Kontrollstrukturen wie Wiederholungen oder Alternativen, in denen wiederum Kontrollstrukturen und Sequenzen von Nachrichten eingebettet sein können.

Verhaltensmuster sind in ihrer Syntax eingeschränkte Sequenzdiagramme. Zum einen stehen nur synchrone Nachrichten zur Verfügung, die Methodenaufrufe zwischen zwei Objekten modellieren. Zum anderen können nur einige der in Sequenzdiagrammen möglichen Kontrollstrukturen in Verhaltensmustern verwendet werden.

Die informelle Beschreibung des Verhaltens eines Entwurfsmusters muss vom Reverse-Engineer in ein Verhaltensmuster umgesetzt werden. Abbildung 4.1 zeigt das Verhaltensmuster des *State*-Entwurfsmusters. Es wurde direkt aus der informellen Beschreibung[1] aus [GHJV95] abgeleitet. In der linken, oberen Ecke des Verhaltensmusters steht neben der Abkürzung bp für *Behavioral Pat-*

[1] siehe Zitat in Abschnitt 3.2.1, Seite 44

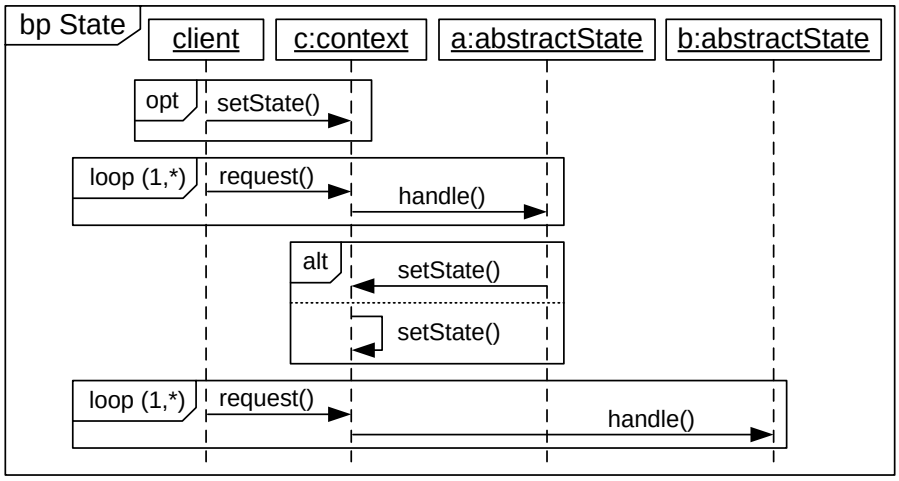

Abbildung 4.1: Das Verhaltensmuster des *State*-Entwurfsmusters

tern – die englische Bezeichnung für Verhaltensmuster – auch der Name des Entwurfsmusters, für das das Verhaltensmuster spezifiziert wurde.

Aus der Beschreibung „*Clients can configure a context with State objects.*“ ist die erste Kontrollstruktur des Verhaltensmusters aus Abbildung 4.1 abgeleitet. Der Klient kann den Kontext mit Zustandsobjekten konfigurieren, das bedeutet auch, dass er unter anderem den Startzustand wählen kann. Die erste Kontrollstruktur ist deshalb optional und enthält die Nachricht setState() des Objekts client an das Objekt c:context. Nach dieser Initialisierung werden Anfragen des clients vom Objekt c:context an den aktuellen Zustand a:abstractState delegiert, wie es durch den Satz „*Context delegates state-specific requests to the current ConcreteState object.*“ beschrieben wird. Die Kombination aus Anfrage (request()) und Delegation (handle()) findet mindestens einmal statt, kann aber beliebig häufig wiederholt werden. Dies ist durch die Wiederholung loop (1,*) ausgedrückt. Aus dem Satz „*Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.*“ ist die folgende Alternative abgeleitet. Der Folgezustand wird entweder durch den aktuellen Zustand a:abstractState oder durch das Objekt c:context ausgewählt. Die weiteren Anfragen werden nun an den Zustand b:abstractState delegiert, wiederum eingebettet in eine Schleife zur beliebig häufigen Wiederholung.

Das Verhaltensmuster des *Strategy*-Entwurfsmusters in Abbildung 4.2 ist analog zum *State*-Verhaltensmuster aus der informellen Beschreibung[2] in

[2]siehe Zitat in Abschnitt 3.2.1, Seite 46

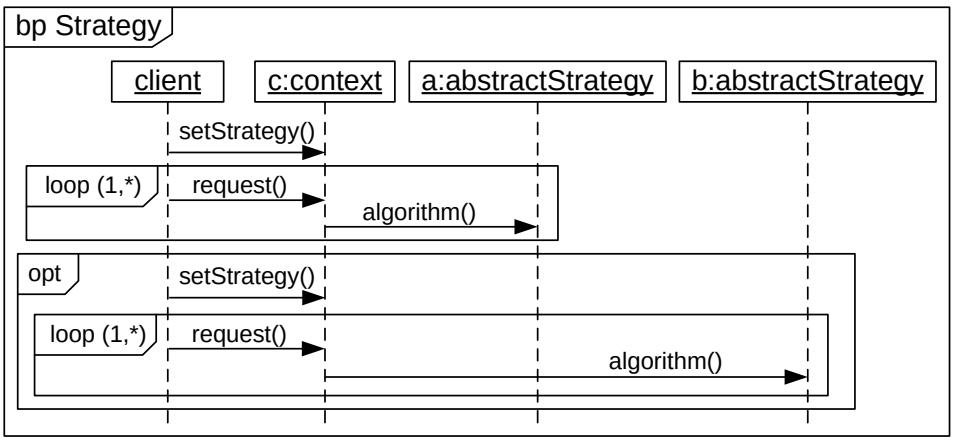

Abbildung 4.2: Das Verhaltensmuster des *Strategy*-Entwurfsmusters

[GHJV95] abgeleitet worden. Im Unterschied zum *State*-Entwurfsmuster muss hier vom client zunächst eine Strategie vorgegeben werden, bevor der client Anfragen an das Objekt c:context stellen kann. Die Anfragen (request()) werden dann wiederum an die aktuelle Strategie a:abstractStrategy delegiert (algorithm()). Ein Strategiewechsel zwischen den Anfragen ist optional vom client durchzuführen. Sollte ein Wechsel erfolgen, so sind alle weiteren Anfragen von c:context an die neue Strategie b:abstractStrategy zu delegieren.

Ein Verhaltensmuster muss nicht das gesamte mögliche Verhalten einer Entwurfsmusterimplementierung zur Laufzeit beschreiben. Bei der Spezifikation sollte ein Verhaltensmuster ähnlich wie ein Strukturmuster auf die wesentlichen Eigenschaften des Entwurfsmusters beschränkt werden. Die wesentlichen Eigenschaften im Verhalten des *State*-Entwurfsmusters sind die Delegation von Anfragen an den aktuellen Zustand und der Zustandswechsel, ausgelöst durch den Kontext oder den aktuellen Zustand. Die letztgenannte Eigenschaft ist zudem ein Unterscheidungsmerkmal zum strukturell identischen *Strategy*-Entwurfsmuster. Beim *Strategy*-Entwurfsmuster werden zwar auch alle Anfragen an die aktuelle Strategie delegiert, der Strategiewechsel wird aber vom Klienten ausgelöst.

Da es für das *State*-Entwurfsmuster ausreicht, einen einzelnen Zustandswechsel zu modellieren, werden nur zwei Zustandsobjekte in dem Verhaltensmuster verwendet. Zur Laufzeit können jedoch beliebig viele Zustandsobjekte existieren. Die Anzahl der Zustandsobjekte zur Laufzeit wird durch das Verhaltensmuster nicht festgelegt. Das gleiche gilt für die Strategieobjekte im *Strategy*-Verhaltensmuster.

4.1.2 Negative Verhaltensmuster

Bisher wurde durch Verhaltensmuster beschrieben, wie sich eine Entwurfsmusterimplementierung zur Laufzeit des Programms verhalten soll. Allerdings können Verhaltensmuster auch beschreiben, wie sich eine Entwurfsmusterimplementierung auf keinen Fall verhalten darf. Solche Verhaltensmuster werden *negative Verhaltensmuster* genannt. Im Gegensatz dazu werden die zuvor beschriebenen Verhaltensmuster auch als *positive Verhaltensmuster* bezeichnet. Zu einem Entwurfsmuster können ein positives und beliebig viele negative Verhaltensmuster spezifiziert werden.

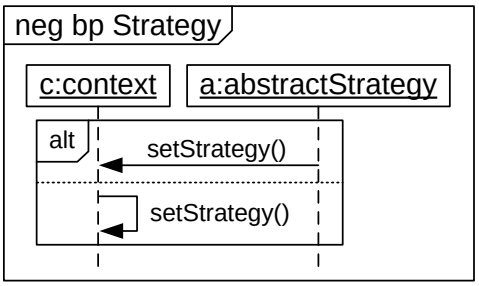

Abbildung 4.3: Ein negatives Verhaltensmuster des *Strategy*-Entwurfsmusters

Abbildung 4.3 zeigt ein negatives Verhaltensmuster des *Strategy*-Entwurfsmusters. Es verbietet einen Strategiewechsel durch die aktuelle Strategie oder durch den Kontext. Dieses Verhalten wird von einem *State*-Entwurfsmuster erwartet, nicht jedoch von einem *Strategy*-Entwurfsmuster. Gekennzeichnet werden negative Verhaltensmuster durch den Zusatz `neg` im Kopf des Verhaltensmusters.

4.1.3 Verbindung zu Strukturmustern

Soll ein Verhaltensmuster zu einem Entwurfsmuster definiert werden, muss immer auch ein Strukturmuster für dieses Entwurfsmuster vorhanden sein. Wie bereits in Abschnitt 3.2.2 erläutert, wird im neuen, struktur- und verhaltensbasierten Erkennungsprozess zunächst die Strukturanalyse auf dem Quelltext mit Hilfe der Strukturmuster durchgeführt. Die verhaltensbasierte Entwurfsmustererkennung erhält dann als Eingabe unter anderem die Kandidaten, die in der Strukturanalyse identifiziert wurden. Um die während der Verhaltensanalyse anfallenden Datenmengen einzuschränken, werden wie bereits beschrieben nur Methodenaufrufe der Kandidaten beobachtet. Aus diesem Grund sind die für

die Verhaltensmusterobjekte verwendbaren Typen und die für die Nachrichten des Verhaltensmusters verwendbaren Methoden eingeschränkt.

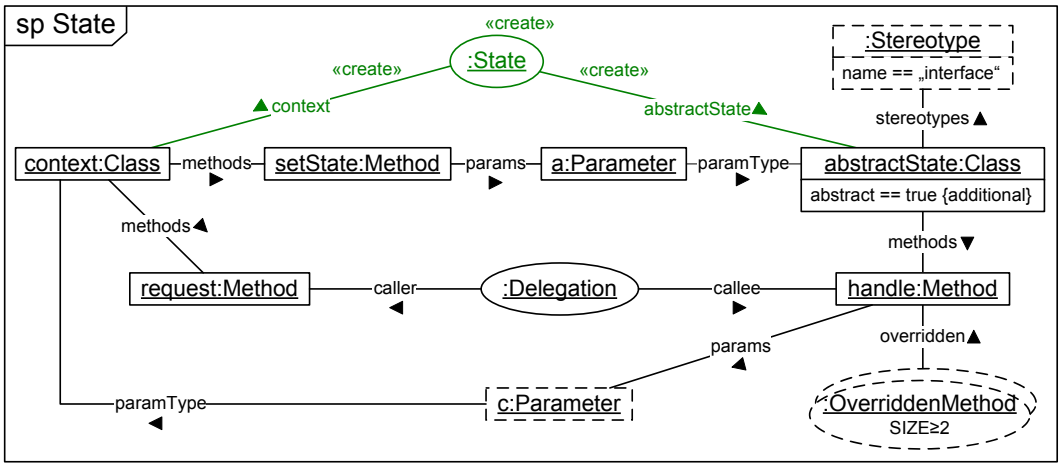

Abbildung 4.4: Das Strukturmuster des *State*-Entwurfsmusters

Im Strukturmuster des *State*-Entwurfsmusters (Abbildung 4.4) sind Objekte für zwei Klassen, context:Class und abstractState:Class, sowie Objekte für drei Methoden, setState:Method, request:Method und handle:Method, spezifiziert. Diese Strukturmusterobjekte sind Variablen, die während der Strukturanalyse gebunden werden. Wird ein Kandidat für eine Entwurfsmusterimplementierung gefunden, so wird, wie bereits in Kapitel 2.3.4 beschrieben, für diesen Kandidaten eine Annotation erzeugt, die die Strukturmusterobjekte an konkrete Elemente aus der Struktur des zu untersuchenden Softwaresystems bindet. Die oben genannten Objekte des Strukturmusters werden an Elemente des ASG gebunden, die Klassen beziehungsweise Methoden repräsentieren.

Ein Verhaltensmuster beschreibt das Verhalten von Instanzen der gebundenen Klassen anhand von Nachrichten, die durch Methodenaufrufe zwischen den Instanzen ausgetauscht werden. Zur Typisierung der Objekte eines Verhaltensmusters stehen also die Objekte des Strukturmusters zur Verfügung, die Variablen für konkrete Klassen darstellen. Als Typnamen werden im Verhaltensmuster stellvertretend die Namen der Strukturmusterobjekte verwendet. Im Falle des *State*-Strukturmusters stehen context und abstractState als Typnamen zur Verfügung, die im *State*-Verhaltensmuster in Abbildung 4.1 verwendet werden. Bei der Verhaltensanalyse werden dann die stellvertretenden Typnamen durch die konkreten Klassennamen der Kandidaten ersetzt.

Analog müssen die Methoden, die in den Nachrichten des Verhaltensmusters

verwendet werden, im Strukturmuster spezifiziert sein. Für eine Nachricht im Verhaltensmuster wird ebenfalls der Name des Strukturmusterobjekts verwendet, das die entsprechende Methode repräsentiert. Während der Verhaltensanalyse wird dann die Variable durch die konkrete Methode ersetzt.

Die Objekte eines Verhaltensmusters müssen nicht grundsätzlich typisiert sein. Im Falle des *State*-Verhaltensmusters ist zum Beispiel zu dem Objekt client kein Typname angegeben. Das bedeutet, bei der Verhaltensanalyse ist die Klasse dieses Verhaltensmusterobjekts nicht festgelegt. Das Verhaltensmusterobjekt kann also während der Verhaltensanalyse an eine beliebige Instanz gebunden werden.

4.2 Syntax

Die Basis zur Spezifikation der Syntax der Verhaltensmuster bildet das Metamodell der Verhaltensmuster. In Form eines UML-Klassendiagramms werden zunächst die möglichen Elemente eines Verhaltensmusters und ihre Beziehungen untereinander beschrieben. Die Anteile, die nicht in einem Klassendiagramm ausgedrückt werden können, wie Invarianten, werden durch die *Object Constraint Language* (OCL), einem Teil der UML, spezifiziert [Obj]. Des Weiteren wird das Metamodell der Strukturmuster erweitert, um die Verbindung zwischen Struktur- und Verhaltensmustern zu definieren.

4.2.1 Metamodell der Verhaltensmuster

Die Syntax der Verhaltensmuster ist angelehnt an die Syntax der Sequenzdiagramme nach UML 2.0. Allerdings gibt es gegenüber den Sequenzdiagrammen einige Einschränkungen und spezielle Eigenschaften der Verhaltensmuster, die im Folgenden spezifiziert werden.

Die Abbildung 4.5 zeigt das Metamodell der Verhaltensmuster. Zentraler Bestandteil dieses Metamodells ist die Klasse BehavioralPattern, die Verhaltensmuster repräsentiert. Sie enthält den Namen des Verhaltensmusters und ein boolsches Attribut, das definiert, ob es ein negatives Verhaltensmuster ist. Des Weiteren referenziert ein Verhaltensmuster eine Menge von Verhaltensmusterobjekten vom Typ BPAbstractObject, die Teil des Verhaltensmusters sind. Zu jedem Verhaltensmusterobjekt gehört eine Lebenslinie vom Typ LifeLine. Eine Lebenslinie repräsentiert den zeitlichen Ablauf von Nachrichten (Message), die durch das jeweilige Verhaltensmusterobjekt in einer festgelegten Reihenfolge gesendet oder von ihm empfangen werden. Die Nachrichten

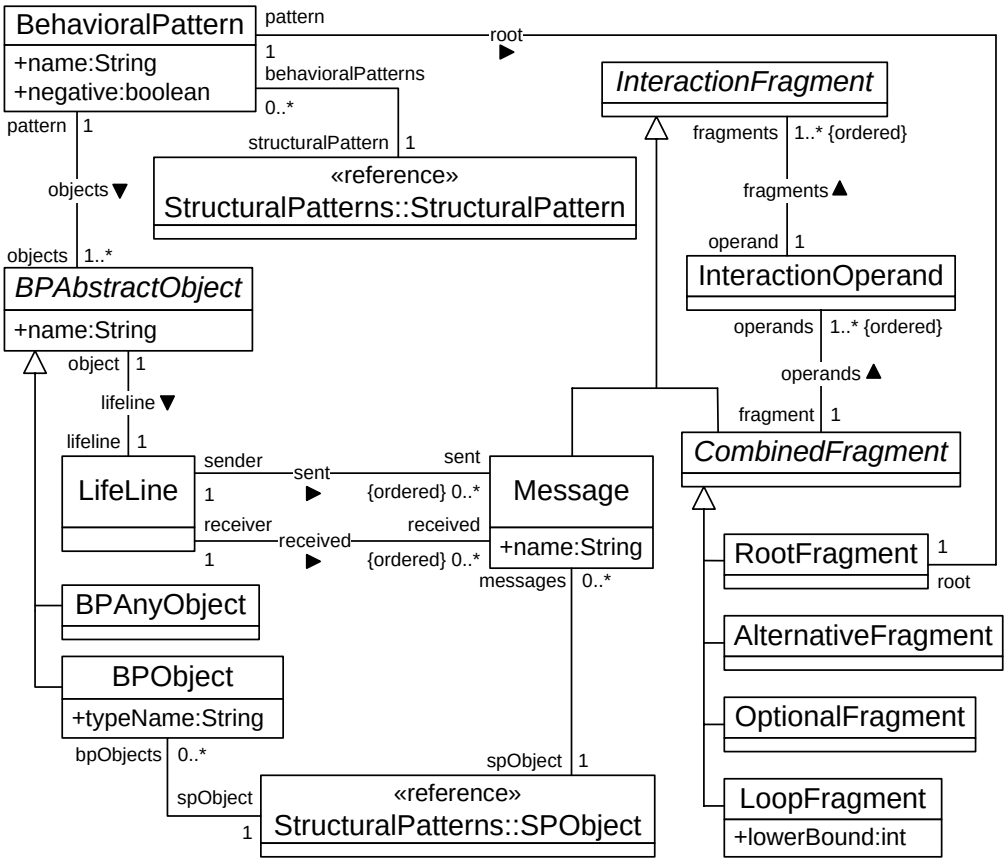

Abbildung 4.5: Metamodell der Verhaltensmuster, Paket BehavioralPatterns

eines Verhaltensmusters sind grundsätzlich synchron, da sie atomare Methodenaufrufe darstellen.

Zur Modellierung des Kontrollflusses in einem Sequenzdiagramm werden in der UML 2.0 die so genannten *kombinierten Fragmente* (engl. *Combined Fragments*) eingesetzt. Kombinierte Fragmente besitzen einen Operator und enthalten mindestens einen Operanden. Operanden enthalten wiederum *Interaktionsfragmente* (engl. *Interaction Fragments*). Interaktionsfragmente können sowohl kombinierte Fragmente, als auch Nachrichten sein. Die wichtigsten, in Sequenzdiagrammen zur Verfügung stehenden kombinierten Fragmente sind: *alternatives Fragment* (Operator: alt), *relevante Nachrichten* (Operator: consider), *irrelevante Nachrichten* (Operator: ignore), *kritischer Bereich* (Operator: critical), *Negation* (Operator: neg), *optionales Fragment* (Operator: opt), *paralleles Fragment* (Operator: par) und *Schleife* (Operator: loop).

Im Metamodell der Verhaltensmuster werden Interaktionsfragmente durch die abstrakte Klasse InteractionFragment und kombinierte Fragmente durch die abstrakte Klasse CombinedFragment repräsentiert. In Verhaltensmustern stehen jedoch nicht alle kombinierten Fragmente zur Verfügung. Es können nur alternative Fragmente (AlternativeFragment), optionale Fragmente (OptionalFragment) und Schleifen (LoopFragment) verwendet werden. Schleifen besitzen in Verhaltensmustern nur eine untere Grenze, die 0 oder 1 sein kann. Bei einer Untergrenze von 0 muss die Schleife nicht durchlaufen werden, bei einer Untergrenze von 1 muss sie mindestens einmal durchlaufen werden. Eine Obergrenze kann nicht festgelegt werden. Eine Schleife kann grundsätzlich mit Einschränkung der Untergrenze beliebig häufig durchlaufen werden.

Ein spezielles, kombiniertes Fragment ist das Wurzel-Fragment (RootFragment), das in jedem Verhaltensmuster nur genau einmal vorkommt und direkt vom Verhaltensmuster referenziert wird. Ein RootFragment ist die Wurzel aller weiteren kombinierten Fragmente und Nachrichten des Verhaltensmusters. Die Klassen des Metamodells der Verhaltensmuster gehören zum Paket BehavioralPatterns.

Als Beispiel ist in Abbildung 4.6 der abstrakte Syntaxgraph des negativen *Strategy*-Verhaltensmusters aus Abbildung 4.3 (Seite 61) dargestellt. Das Verhaltensmuster referenziert ein Objekt vom Typ RootFragment als Wurzel der gesamten Sequenz. Der Operand der Wurzel enthält wiederum eine Alternative mit zwei Operanden, die untereinander geordnet sind. Im ersten Operanden wird die Nachricht setStrategy() vom Verhaltensmusterobjekt a:abstractState an c:context gesendet, im zweiten Operanden schickt sich das Objekt c:context die Nachricht setStrategy() selber.

Im Folgenden werden einige Invarianten definiert, die ein syntaktisch korrektes Verhaltensmuster einhalten muss und die nicht in einem Klassendiagramm ausgedrückt werden können. Damit ein Verhaltensmuster ein sinnvolles Verhalten durch Sequenzen von Nachrichten spezifizieren kann, muss es mindestens ein Objekt vom abstrakten Typ BPAbstractObject enthalten. Jedes Objekt in einem Verhaltensmuster muss außerdem einen Namen besitzen und über diesen eindeutig zu identifizieren sein:

Invariante 4.1 *Jedes Verhaltensmusterobjekt hat einen Namen:*

```
package BehavioralPatterns
context BPAbstractObject inv:
  self.name<>OclVoid
endpackage
```

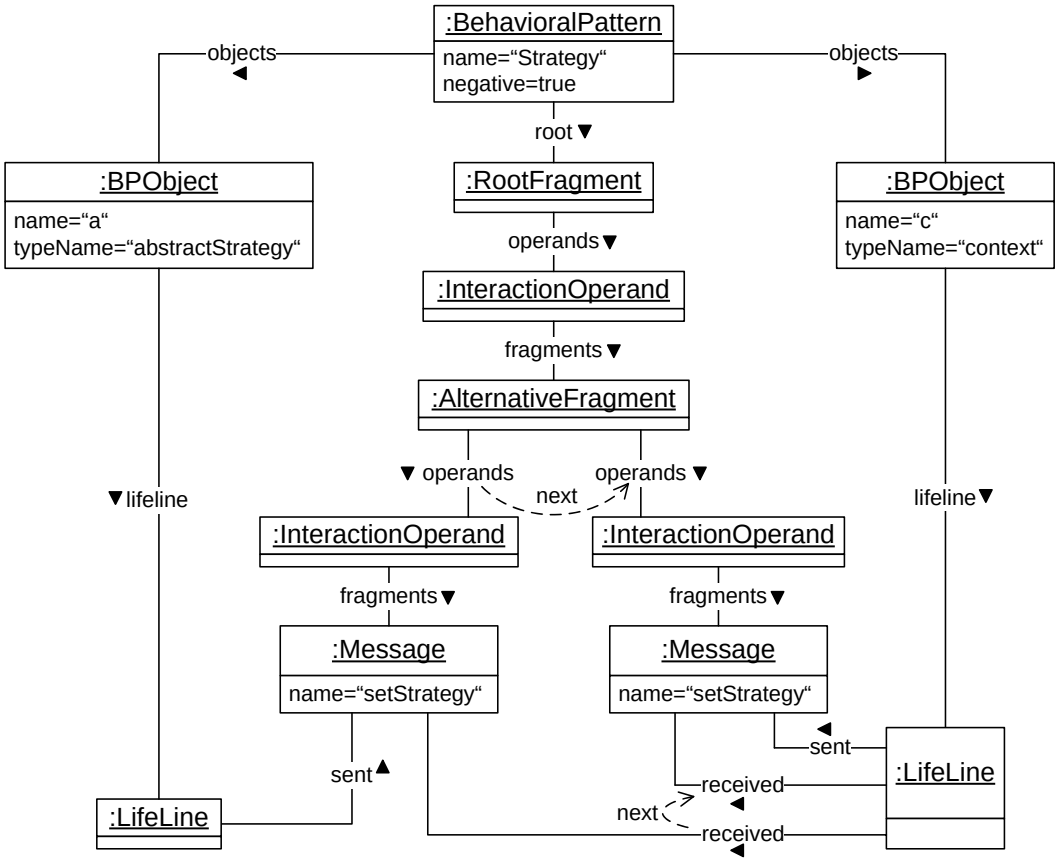

Abbildung 4.6: Abstrakter Syntaxgraph des negativen *Strategy*-Verhaltensmusters

Invariante 4.2 *Die Namen der Objekte eines Verhaltensmusters sind eindeutig:*

package BehavioralPatterns
context BehavioralPattern **inv**:
self.objects→**forAll**(b1:BPAbstractObject, b2:BPAbstractObject|
b1<>b2 **implies** b1.name<>b2.name)
endpackage

Für die kombinierten Fragmente gelten in einem syntaktisch korrekten Verhaltensmuster die folgenden Invarianten:

Invariante 4.3 *Die kombinierten Fragmente Wurzel-Fragment, optionales*

Fragment und Schleife besitzen jeweils genau einen Operanden:

package BehavioralPatterns

context RootFragment **inv**:
self.operands→size()=1

context OptionalFragment **inv**:
self.operands→size()=1

context LoopFragment **inv**:
self.operands→size()=1

endpackage

Invariante 4.4 *Das alternative Fragment enthält mindestens zwei, aber beliebig viele Operanden:*

package BehavioralPatterns
context AlternativeFragment **inv**:
self.operands→size()≥2
endpackage

4.2.2 Erweiterung des Metamodells der Strukturmuster

Um die Verbindung zwischen Verhaltens- und Strukturmustern herzustellen, wurde das Metamodell der Strukturmuster aus Abbildung 2.8, Seite 25, erweitert. In Abbildung 4.7 ist das neue, erweiterte Metamodell der Strukturmuster dargestellt. Die referenzierten Klassen StructuralPattern und SPObject im Metamodell der Verhaltensmuster aus Abbildung 4.5 verweisen auf die entsprechenden Klassen des erweiterten Metamodells der Strukturmuster.

Strukturmusterobjekte werden während der Strukturanalyse an konkrete Elemente eines Kandidaten, also an Elemente der Struktur des Softwaresystems, gebunden. Strukturmusterobjekte sind also Variablen, deren Typen dem Strukturmodell entnommen sind. Die Typnamen der Verhaltensmusterobjekte und die Methodennamen der Nachrichten werden wiederum den Strukturmusterobjekten entnommen. Die Strukturmusterobjekte dienen somit der Typisierung der Elemente eines Verhaltensmusters. Strukturmusterobjekte, die Variablen für konkrete Klassen aus der Struktur darstellen, werden zur Typisierung der Verhaltensmusterobjekte verwendet. Strukturmusterobjekte, die

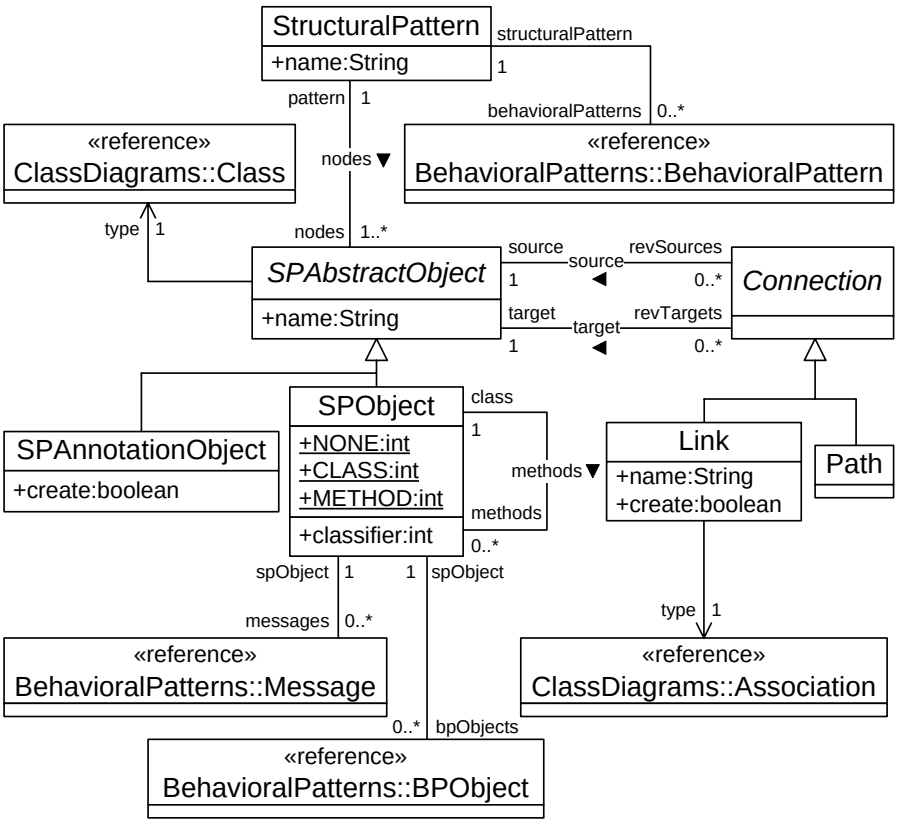

Abbildung 4.7: Erweitertes Metamodell der Strukturmuster, Paket Structural-Patterns

Variablen für Methoden aus der Struktur sind, werden zur Typisierung von Nachrichten in Verhaltensmustern verwendet. Dazu sind der Klasse SPObject aus dem Metamodell der Strukturmuster Referenzen zu den Klassen Message und BPObject aus dem Metamodell der Verhaltensmuster hinzugefügt worden.

Die Strukturmusterobjekte müssen für die Typisierung der Verhaltensmusterobjekte in eine Klassifizierung aus Klassen, Methoden oder sonstigen Objekten der Struktur eingeordnet werden. Das Strukturmodell ist jedoch austauschbar. Es ist also nicht allgemein feststellbar, welche Klassen des Strukturmodells Klassen oder Methoden in der Struktur eines Softwaresystems repräsentieren. Aus diesem Grund wurde das Metamodell der Strukturmuster um die Klassifizierung der Strukturmusterobjekte erweitert. Der Klasse SPObject für Strukturmusterobjekte sind also ein Attribut zur Klassifizierung sowie drei Konstanten hinzugefügt worden, die die drei Klassifizierungen Klasse

(CLASS), Methode (METHOD) und sonstiges Element (NONE) repräsentieren. Die Klassifizierung eines Strukturmusterobjekts findet bei der Spezifikation eines Strukturmusters statt.

Stellt ein Strukturmusterobjekt eine Methode aus der Struktur dar, so muss zusätzlich angegeben werden, welches Strukturmusterobjekt die Klasse darstellt, zu der die Methode gehört. Dies wird durch die Selbstassoziation methods von SPObject ermöglicht. Allerdings sind die folgenden zwei Invarianten einzuhalten:

Invariante 4.5 *Einem Strukturmusterobjekt, das als METHOD klassifiziert ist, ist über die Assoziation methods immer ein Strukturmusterobjekt der Klassifizierung CLASS zugeordnet:*

```
package StructuralPatterns
context SPObject inv:
  self.classifier=SPObject.METHOD implies
    (self.class<>OclVoid and self.class.classifier=SPObject.CLASS)
endpackage
```

Invariante 4.6 *Einem Strukturmusterobjekt der Klassifizierung CLASS sind über die Assoziation methods nur Strukturmusterobjekte der Klassifizierung METHOD zugeordnet:*

```
package StructuralPatterns
context SPObject inv:
  self.classifier=SPObject.CLASS implies
    (self.methods→forAll(m:SPObject|
      m.classifier=SPObject.METHOD))
endpackage
```

4.2.3 Verbindung zwischen Struktur- und Verhaltensmustern

Wie bereits in Kapitel 4.1.3 erklärt, ist jedem Verhaltensmuster ein Strukturmuster zugeordnet. Umgekehrt kann ein Strukturmuster durch ein positives Verhaltensmuster und beliebig viele negative Verhaltensmuster ergänzt werden. Die Verbindung zwischen Verhaltens- und Strukturmuster wird in den beiden Metamodellen durch eine Assoziation zwischen den Klassen BehavioralPattern und StructuralPattern hergestellt. Spezifiziert wird die genannte Einschränkung durch die folgende Invariante:

Invariante 4.7 *Zu jedem Strukturmuster darf maximal ein positives Verhaltensmuster existieren:*

```
package StructuralPatterns
context StructuralPattern inv:
  self.behavioralPatterns→
    collect(bp:BehavioralPatterns::BehavioralPattern|
      bp.negative=false)→size()≤1
endpackage
```

Des Weiteren wird die Zugehörigkeit eines Verhaltensmusters zu einem Strukturmuster durch die Namensgebung ausgedrückt. Invariante 4.8 beschreibt die Namensgleichheit zwischen Struktur- und Verhaltensmustern:

Invariante 4.8 *Die Namen eines Strukturmusters und aller zugeordneten Verhaltensmuster stimmen überein:*

```
package StructuralPatterns
context StructuralPattern inv:
  self.behavioralPatterns→
    forAll(bp:BehavioralPatterns::BehavioralPattern|
      self.name=bp.name)
endpackage
```

Es existieren zwei konkrete Unterklassen für Verhaltensmusterobjekte: BPAnyObject und BPObject. BPAnyObject repräsentiert alle untypisierten Verhaltensmusterobjekte, also Verhaltensmusterobjekte, deren Typ nicht spezifiziert ist und die während der Verhaltensanalyse an beliebige Instanzen gebunden werden können. BPObject repräsentiert dagegen die typisierten Verhaltensmusterobjekte. Der Typ wird durch das dem Verhaltensmusterobjekt zugeordnete Strukturmusterobjekt festgelegt, das wiederum eine Variable für eine Klasse aus der Struktur eines Softwaresystems darstellt. Der Typname des Verhaltensmusterobjekts ist der Objektname des Strukturmusterobjekts. Diese Einschränkungen sind in den beiden folgenden Invarianten definiert:

Invariante 4.9 *Einem typisierten Verhaltensmusterobjekt ist immer ein Strukturmusterobjekt mit der Klassifizierung CLASS zugeordnet:*

```
package BehavioralPatterns
context BPObject inv:
```

```
  self.spObject.classifier=StructuralPatterns::SPObject.CLASS
endpackage
```

Invariante 4.10 *Der Typname eines typisierten Verhaltensmusterobjekts stimmt mit dem Namen des zugeordneten Strukturmusterobjekts überein:*

```
package BehavioralPatterns
context BPObject inv:
  self.typeName=self.spObject.name
endpackage
```

Analog zu den typisierten Verhaltensmusterobjekten ist eine Nachricht im Verhaltensmuster immer mit einem Strukturmusterobjekt verbunden, das eine Variable für eine Methode aus der Struktur eines Softwaresystems darstellt. Der Name der Nachricht im Verhaltensmuster entspricht dem Objektnamen des Strukturmusterobjekts.

Invariante 4.11 *Einer Nachricht eines Verhaltensmusters ist immer ein Strukturmusterobjekt mit der Klassifizierung* METHOD *zugeordnet:*

```
package BehavioralPatterns
context Message inv:
  self.spObject.classifier=StructuralPatterns::SPObject.METHOD
endpackage
```

Invariante 4.12 *Der Name einer Nachricht im Verhaltensmuster stimmt mit dem Namen des zugeordneten Strukturmusterobjekts überein:*

```
package BehavioralPatterns
context Message inv:
  self.name=self.spObject.name
endpackage
```

Die Art der Nachrichten, die an ein Verhaltensmusterobjekt gesendet werden dürfen, sind eingeschränkt. An ein typisiertes Verhaltensmusterobjekt dürfen nur Nachrichten gesendet werden, deren Methoden zu der entsprechenden Klasse des Verhaltensmusterobjekts gehören. Auf untypisierten Verhaltensmusterobjekten dürfen dagegen keine Methodenaufrufe erfolgen.

Invariante 4.13 *An ein typisiertes Verhaltensmusterobjekt* bpObject *dürfen nur Nachrichten gesendet werden, deren Methoden dem zugehörigen Strukturmusterobjekt* spObject *zugeordnet sind:*

```
package BehavioralPatterns
context BPObject inv:
  self.lifeline.received→forAll(m:Message|
    self.spObject.methods→includes(m.spObject)))
endpackage
```

Invariante 4.14 *An ein nicht typisiertes Verhaltensmusterobjekt darf keine Nachricht gesendet werden:*

```
package BehavioralPatterns
context BPAnyObject inv:
  self.lifeline.received→size()=0
endpackage
```

4.2.4 Überblick

Das in Abschnitt 2.3.7 eingeführte Schema aus Metamodell, Modell und Instanz beziehungsweise Implementierung wird hier wieder aufgegriffen, um die in den vorherigen Abschnitten eingeführten Diagramme darin einzuordnen.

In Abbildung 4.8 sind die Abstraktionsschichten der Strukturmuster auf der linken Seite denen der Verhaltensmuster auf der rechten Seite gegenübergestellt. Auf der obersten Schicht der Metamodelle wurde das bisherige Metamodell der Strukturmuster aus Abbildung 2.13 durch das erweiterte Metamodell der Strukturmuster ersetzt. Das Strukturmuster des *State*-Entwurfsmusters wurde durch die Variante mit der unscharfen Spezifikation ausgetauscht.

Auf der rechten Seite sind die Diagramme der Verhaltensmuster eingeordnet. In der obersten Schicht liegt das Metamodell der Verhaltensmuster. In der Mitte ist stellvertretend für alle möglichen Verhaltensmuster das *State*-Verhaltensmuster abgebildet. In der untersten Schicht ist ein möglicher, zum *State*-Verhaltensmuster konformer Trace dargestellt.

Die semantische Beziehung zwischen der linken und der rechten Seite ist wiederum durch Referenzen zwischen den beiden Metamodellen der Struktur- und Verhaltensmuster festgelegt.

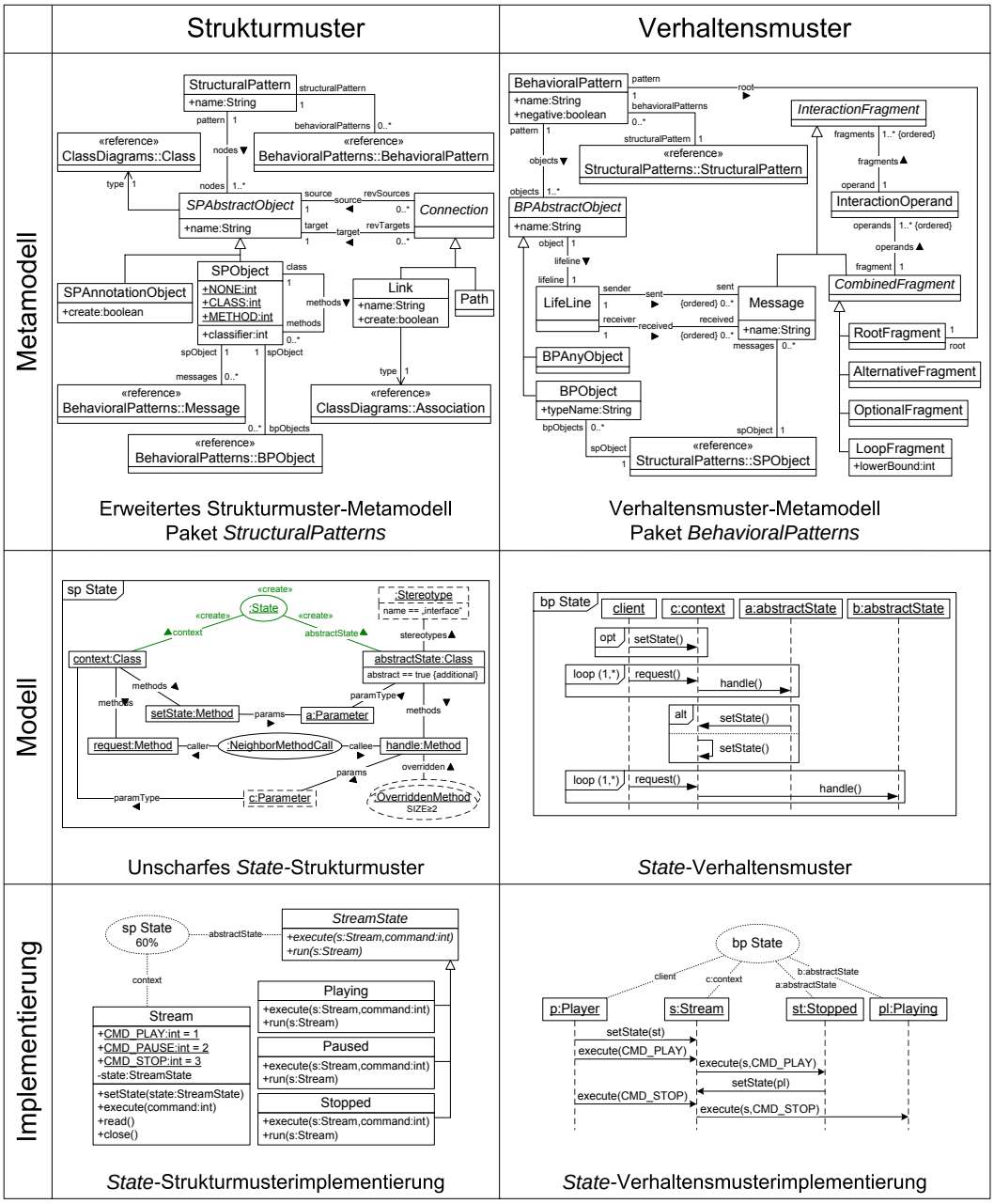

Abbildung 4.8: Die Abstraktionsschichten der Struktur- und Verhaltensmuster

4.3 Semantik

In diesem Abschnitt wird die Semantik der Verhaltensmuster informell erläutert. Als Beispiel dient dazu der Mediaplayer von Seite 19. In der Strukturanalyse wurden einige Klassen des Mediaplayers als *State*-Kandidat identifiziert (Abbildung 4.9).

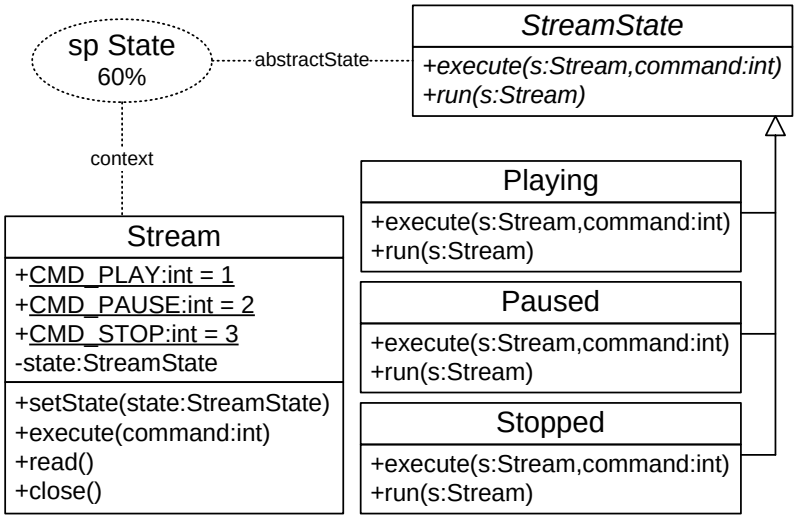

Abbildung 4.9: Der *State*-Kandidat des Mediaplayers

Im Folgenden wird zunächst diskutiert, warum es zur Laufzeit zu einem Kandidaten mehrere Traces geben kann, die auf ihre Konformität zum Verhaltensmuster überprüft werden müssen. Dann wird das Mediaplayer-Beispiel dazu verwendet, die Bindung der Variablen des *State*-Verhaltensmusters zu erklären. Anhand verschiedener Traces des Mediaplayers wird gezeigt, wann ein einzelner Methodenaufruf des Traces zu einer Nachricht konform ist und wann ein kompletter Trace zu einem Verhaltensmuster konform oder nicht konform ist. Zum Abschluss wird die Bewertung konformer und nicht-konformer Traces erläutert.

4.3.1 Mehrfache Überprüfung der Traces

Wie bereits in Abschnitt 4.1.1 erläutert, deckt ein Verhaltensmuster nicht das gesamte mögliche Verhalten einer Entwurfsmusterimplementierung zur Laufzeit ab. Vielmehr soll mit Verhaltensmustern lokales Verhalten einer Entwurfsmusterimplementierung durch typische Sequenzen von Methodenaufrufen aus-

gedrückt werden. Aus diesem Grund werden Traces, die nur Ausschnitte der gesamten Sequenz beobachteter Methodenaufrufe sind, auf ihre Konformität zu einem Verhaltensmuster untersucht.

Die Instanz einer Entwurfsmusterimplementierung kann zur Laufzeit das durch das Verhaltensmuster beschriebene lokale Verhalten mehrfach durchlaufen. Das führt dazu, dass beliebig viele zu einem Verhaltensmuster konforme Traces dieser Instanz in einem Tracegraphen identifiziert werden können.

Des Weiteren können aber auch beliebig viele Instanzen einer Entwurfsmusterimplementierung zur Laufzeit existieren. Zu jeder dieser Instanzen werden Traces beobachtet, die auf ihre Konformität zum Verhaltensmuster untersucht werden müssen. Es ist also sehr wahrscheinlich, dass es zu einem Kandidaten zur Laufzeit verschiedene Traces derselben oder auch unterschiedlicher Instanzen gibt, die konform zum Verhaltensmuster sind. Ebenso kann es aber auch Traces von Instanzen geben, die nicht konform zum Verhaltensmuster sind.

4.3.2 Bindung der Variablen

Die Annotation, die zu einem in der Strukturanalyse identifizierten Kandidaten erzeugt wurde, enthält eine Bindung der Variablen des Strukturmusters an die Elemente des Kandidaten. Die Bindung der *State*-Annotation aus Abbildung 4.9 ist in Tabelle 4.1 aufgelistet.

Variable des Strukturmusters	Element des Kandidaten
context	Stream
abstractState	StreamState
setState	Stream::setState(StreamState)
request	Stream::execute(int)
handle	*StreamState::execute(Stream, int)*

Tabelle 4.1: Initiale Variablenbindung der *State*-Annotation zum Mediaplayer

Die Annotation wird ebenfalls zur Bindung der Variablen des Verhaltensmusters verwendet. Die Typnamen der Verhaltensmusterobjekte und die Methodennamen der Nachrichten wurden bei der Spezifikation des Verhaltensmusters dem Strukturmuster entnommen, wie im vorherigen Abschnitt bereits erläutert wurde. Dadurch werden die Variablen des Verhaltensmusters an dieselben Elemente des Kandidaten gebunden wie die Variablen des Strukturmusters. Es ist dadurch festgelegt, welche Typen die Verhaltensmusterobjekte haben und welche Methoden dieser Typen im Verhaltensmuster vorkommen.

Die Bindung der Typnamen der Verhaltensmusterobjekte und der Methodennamen der Nachrichten steht bereits zu Beginn der Verhaltensanalyse fest. Die Verhaltensmusterobjekte werden dagegen erst während der Verhaltensanalyse bei der Überprüfung der Konformität von beobachteten Methodenaufrufen an konkrete Instanzen aus der Laufzeitumgebung des zu untersuchenden Softwaresystems gebunden.

Zur Vereinfachung der Beschreibung der Semantik sei aber im Folgenden davon ausgegangen, dass bereits zu Beginn der Verhaltensanalyse alle Instanzen aus der Laufzeitumgebung bekannt sind. Aus dieser Menge werden nichtdeterministisch alle möglichen Tupel von Instanzen, die als eine Instanz einer potentiellen Entwurfsmusterimplementierung kollaborieren, ausgewählt. Mit jedem dieser Tupel wird wiederum die Variablenbindung eines Verhaltensmusters initialisiert. Die Verhaltensmusterobjekte werden nichtdeterministisch an die passenden Elemente des Tupels gebunden. Das bedeutet, für jedes dieser Tupel existiert ein konkretes Verhaltensmuster, bei dem alle Variablen gebunden sind und das beschreibt, wie die Elemente des Tupels interagieren müssen, um als eine Instanz einer tatsächlichen Entwurfsmusterimplementierung zu gelten.

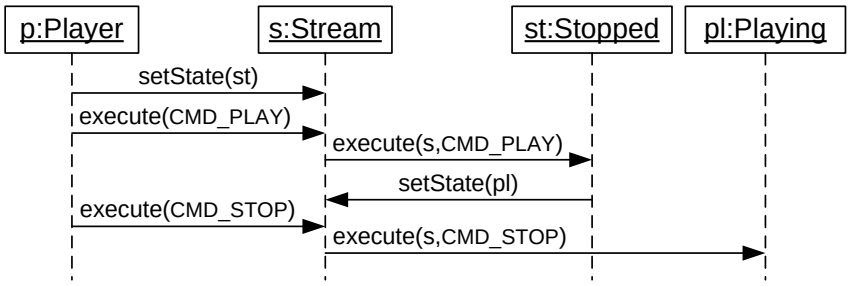

Abbildung 4.10: Beobachteter Trace des Mediaplayers

Werden nun während der Verhaltensanalyse Traces dieser Tupel beobachtet, können sie mit ihrem konkreten Verhaltensmuster verglichen werden. In Abbildung 4.10 ist ein Trace zu sehen, der für einige Instanzen von Klassen des Mediaplayers beobachtet wurde. Diese Instanzen wurden zu Beginn der Verhaltensanalyse nichtdeterministisch ausgewählt. Zusammen mit den Elementen aus der Struktur des *State*-Kandidaten bilden sie nun eine Variablenbindung für ein konkretes *State*-Verhaltensmuster. Diese Variablenbindung ist in Tabelle 4.2 aufgelistet.

Abbildung 4.11 zeigt das zu der Variablenbindung gehörige *State*-Verhal-

Variable des Verhaltensmusters	Element der Struktur/Instanz
context	Stream
abstractState	StreamState
setState	Stream::setState(StreamState)
request	Stream::execute(int)
handle	*StreamState::execute(Stream, int)*
client	p
c	s
a	st
b	pl

Tabelle 4.2: Variablenbindung eines *State*-Verhaltensmusters zum Mediaplayer

tensmuster des Mediaplayer-Kandidaten, bei dem die Typnamen der Verhaltensmusterobjekte und die Methodennamen der Nachrichten durch die konkreten Elemente des Kandidaten und die Verhaltensmusterobjekte durch die Instanzen aus der Laufzeitumgebung ersetzt wurden, wie es in Tabelle 4.2 vorgegeben ist. Das Verhaltensmuster beschreibt, wie diese Instanzen des Mediaplayers sich verhalten müssen, um als tatsächliche *State*-Entwurfsmusterinstanz identifiziert zu werden.

Die Typen der Verhaltensmusterobjekte stimmen teilweise nicht mit den Typen der Instanzen aus dem Trace überein. Es sind jedoch polymorphe Typbindungen erlaubt. So ist zum Beispiel die Typvariable abstractState des Verhaltensmusterobjekts a an den konkreten Typ StreamState gebunden. Die Instanz st aus dem Trace, an die das Verhaltensmusterobjekt a gebunden ist, ist allerdings vom Typ Stopped. Sie ist aber auch vom Typ StreamState, da der Typ Stopped von StreamState erbt. Es liegt also eine korrekte, polymorphe Bindung vor.

4.3.3 Konformität von Methodenaufrufen

Um festzustellen, ob ein Trace wie in Abbildung 4.10 zu dem Verhaltensmuster in Abbildung 4.11 konform ist, muss zunächst einmal festgelegt werden, wann ein Methodenaufruf des Traces zu einer Nachricht des Verhaltensmusters konform ist. Eine Nachricht besteht aus bis zu fünf verschiedenen Variablen. Diese fünf Variablen – namentlich das aufrufende Verhaltensmusterobjekt und sein Typ, das aufgerufene Verhaltensmusterobjekt und sein Typ sowie die aufgerufene Methode – müssen mit dem Methodenaufruf verglichen werden. Passen

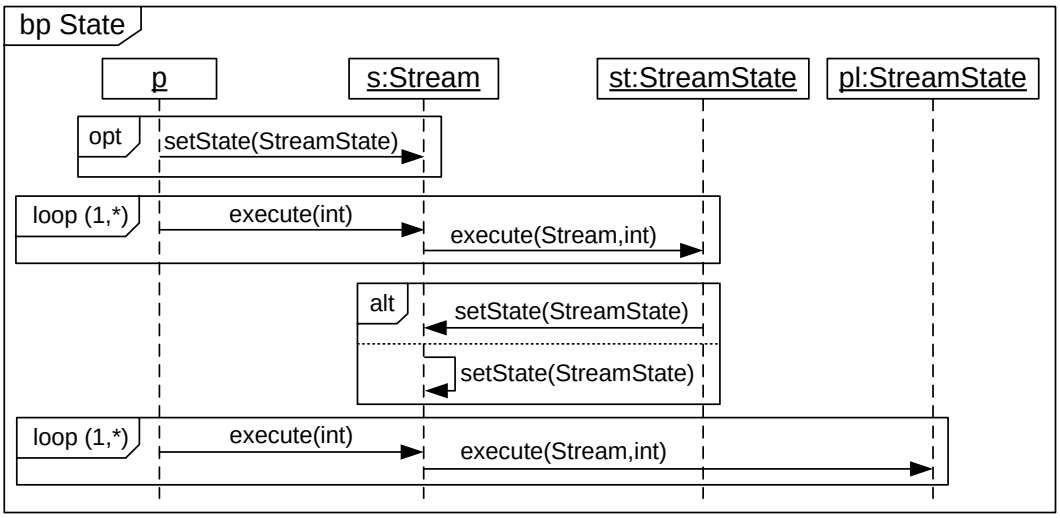

Abbildung 4.11: Ein *State*-Verhaltensmuster zu Instanzen des Mediaplayer-Kandidaten

die Elemente des Methodenaufrufs zu den Elementen, an die die Variablen gebunden sind, so ist der Methodenaufruf konform zu der Nachricht.

Wie bereits oben erwähnt, muss der Typ der an einem Methodenaufruf beteiligten Instanz nicht mit dem Typ des Verhaltensmusterobjekts identisch sein, um das Verhaltensmusterobjekt an die Instanz zu binden. Es dürfen auch polymorphe Bindungen vorliegen. Um formal zu definieren, wann eine polymorphe Bindung erlaubt ist, wird im Folgenden die Konformitätsfunktion über zwei Typen eingeführt. Ein Verhaltensmusterobjekt darf an eine Instanz des Traces gebunden werden, wenn der Typ der Instanz konform ist zu dem Typ, an den die Typvariable des Verhaltensmusterobjekts gebunden ist.

Die Instanzen der Klasse Class des Strukturmodells repräsentieren die Typen in der Struktur eines Softwaresystems. Die Konformitätsfunktion wird daher als Methode der Klasse Class definiert. Eine Instanz c1 von Class ist konform zu einer zweiten Instanz c2 von Class, wenn c1 identisch zu c2 ist oder unmittelbar oder mittelbar von c2 erbt.

Definition 4.1 *Die Konformitätsfunktion der Klasse Class des Strukturmodells ist definiert durch:*

package Structure
context Class::conformsTo(c:Class):Boolean
post: **result** = (self = c)

```
        or (self.revSubClasses→exists(g:Generalization|g.superClass=c))
        or (self.revSubClasses→exists(g:Generalization|
            g.superClass.conformsTo(c)))
endpackage
```

Auch die aufgerufene Methode muss nicht identisch sein zu der Methode, die in der Nachricht spezifiziert wurde. In Entwurfsmustern wird sehr häufig von der polymorphen Methodenbindung Gebrauch gemacht. Im *State*-Entwurfsmuster wird zum Beispiel die handle-Methode durch den abstrakten Zustands vorgegeben, und durch die konkreten Zustände implementiert. Der Aufruf der handle-Methode geschieht aber über die Schnittstelle des abstrakten Zustands. Die polymorphe Methodenbindung wird daher auch in Verhaltensmustern verwendet. Bei einem Methodenaufruf muss also nicht nur überprüft werden, ob die aufgerufene Methode zu der Methode der Nachricht identisch ist, sondern auch, ob eventuell die aufgerufene Methode die Methode der Nachricht implementiert oder überschreibt und damit polymorph gebunden wurde. Aus diesem Grund wird auch eine Konformitätsfunktion für die Klasse Method des Strukturmodells definiert.

Eine Instanz m2 des Typs Method ist zu einer Instanz m1 des Typs Method konform, wenn entweder m2 identisch zu m1 ist, oder m2 die Methode m1 überschreibt oder implementiert. Im zweiten Fall muss der Typ, zu dem m2 gehört, von dem Typ, zu dem m1 gehört, erben und die Signaturen von m1 und m2 müssen identisch sein. Das bedeutet, sowohl die Namen der Methoden m1 und m2, als auch die Reihenfolge und Typen ihrer Parameter müssen übereinstimmen[3].

Definition 4.2 *Die Konformitätsfunktion der Klasse Method des Strukturmodells ist definiert durch:*

```
package Structure
context Method::conformsTo(m:Method):Boolean
post: result = (self = m)
        or (self.parent.conformsTo(m.parent)
            and self.name = m.name
```

[3] In einigen Programmiersprachen müssen die Typen der Parameter nicht übereinstimmen, damit eine Methode m2 eine Methode m1 überschreibt. Eine kovariante (z.B. in Eiffel) oder kontravariante Redefinition der Parametertypen ist ebenfalls möglich. In diesen Fällen muss also eventuell in der Konformitätsfunktion die Überprüfung der Identität der Parametertypen durch eine Konformitätsprüfung ersetzt werden.

and self.params→**forAll**(p:Parameter|p.type =
m.params→**at**(self.params→**indexof**(p)).type))
endpackage

Im Folgenden werden nun fünf Bedingungen formuliert, die gelten müssen, damit ein Methodenaufruf mc des Typs Behavior::MethodCall zu einer Nachricht eines Verhaltensmusters konform ist. Die Nachricht hat die Form (a:A)→(b:B).m(), das bedeutet, das Verhaltensmusterobjekt a:A ruft auf dem Verhaltensmusterobjekt b:B die Methode m auf.

1. Die Methode des Methodenaufrufs mc ist konform zu der Methode, die an die Variable m der Nachricht gebunden wurde.
2. Der Typ der aufrufenden Instanz ist konform zu dem Typ, der an die Variable A der Nachricht gebunden wurde.
3. Der Typ der aufgerufenen Instanz ist konform zu dem Typ, der an die Variable B der Nachricht gebunden wurde.
4. Die Variable a der Nachricht ist an die aufrufende Instanz des Methodenaufrufs mc gebunden.
5. Die Variable b der Nachricht ist an die aufgerufene Instanz des Methodenaufrufs mc gebunden.

Hat die Nachricht dagegen die Form a→(b:B).m(), das bedeutet, ein untypisiertes Verhaltensmusterobjekt a ruft die Methode auf, dann ist der Methodenaufruf mc zu der Nachricht konform, wenn die Bedingungen 1, 3, 4 und 5 gelten. Der Typ der aufrufenden Instanz wird also ignoriert.

4.3.4 Konformität von Traces

Um zu überprüfen, ob ein Trace konform zu einem Verhaltensmuster ist, müssen alle Methodenaufrufe des Traces mit den Vorgaben des Verhaltensmusters verglichen werden. Zu den Vorgaben gehören nicht nur die Art der Nachrichten, die gesendet werden dürfen, sondern vor allem auch die Reihenfolge, in der sie gesendet werden dürfen. Die Methodenaufrufe werden also auf ihre Konformität zu den Nachrichten überprüft, die zum Zeitpunkt ihrer Beobachtung erlaubt sind. Im Folgenden werden mehrere hypothetische Traces des Mediaplayers untersucht, die zur Laufzeit beobachtet werden könnten und anhand derer zum Verhaltensmuster konforme und nicht-konforme Traces erläutert werden.

Konforme Traces

Es wird zunächst die Konformität des Traces aus Abbildung 4.10 (Seite 76) zum konkreten *State*-Verhaltensmuster des Mediaplayer-Kandidaten aus Abbildung 4.11 (Seite 78) gezeigt. Die Methodenaufrufe des Traces werden der Reihe nach auf ihre Konformität zu den Nachrichten des Verhaltensmusters geprüft.

Zu Beginn der Verhaltensanalyse darf entweder die optionale Nachricht p→(s:Stream).setState(StreamState) oder die Nachricht p→(s:Stream).execute (int) gesendet werden. Der erste, beobachtete Methodenaufruf des Mediaplayer-Traces ist setState(st) auf der Instanz s:Stream durch die Instanz p:Player. Die Instanzen des Methodenaufrufs sind identisch mit den Instanzen, die an die Verhaltensmusterobjekte der optionalen Nachricht p→(s:Stream).setState(StreamState) gebunden wurden. Der Typ der aufgerufenen Instanz stimmt mit dem Typ des aufgerufenen Verhaltensmusterobjekts überein. Der Typ der aufrufenden Instanz ist in der Nachricht nicht spezifiziert, er wird also ignoriert. Die Methode des Aufrufs ist ebenfalls zu der Methode der Nachricht identisch. Der erste Methodenaufruf des Traces ist also konform zu der ersten, optionalen Nachricht des Verhaltensmusters.

Zum Beobachtungszeitpunkt des zweiten Methodenaufrufs darf nur die Nachricht p→(s:Stream).execute(int) gesendet werden. Der zweite Methodenaufruf ist konform zu dieser Nachricht. Danach ist nur die Nachricht (s:Stream)→(st:StreamState).execute(Stream, int) erlaubt. Der Typ Stopped der Instanz, auf der der dritte Methodenaufruf stattfindet, gleicht zwar nicht dem im Verhaltensmuster geforderten Typ StreamState, erbt jedoch von StreamState und ist deshalb erlaubt. Der dritte Methodenaufruf ist also konform zur dritten Nachricht des Verhaltensmusters.

Zum Zeitpunkt des vierten Methodenaufrufs dürfen drei verschiedene Nachrichten gesendet werden. Entweder kann die zuvor schon einmal durchlaufene Schleife ein weiteres Mal durchlaufen werden. Dann müsste ein Methodenaufruf erfolgen, der zur Nachricht p→(s:Stream).execute(int) konform ist. Es kann jedoch auch ein Methodenaufruf erfolgen, der zu einer der beiden alternativen Nachrichten konform ist. Der tatsächlich beobachtete Methodenaufruf des Traces ist zu der Nachricht (st:StreamState)→(s:Stream).setState(StreamState) der ersten Alternative konform.

Auch die verbleibenden Methodenaufrufe des Traces sind konform zu den jeweils zu ihrem Beobachtungszeitpunkt erlaubten Nachrichten. Der gesamte Trace ist deshalb konform zu dem konkreten *State*-Verhaltensmuster des Mediaplayer-Kandidaten.

Berücksichtigte und ignorierte Methodenaufrufe

In der Beschreibung eines Entwurfsmusters werden Methoden genannt, die eine ganz bestimmte Rolle spielen, wie zum Beispiel die Methoden setState(), request() und handle() im Entwurfsmuster *State*. Diese Methoden werden durch die Strukturanalyse identifiziert und in der Verhaltensanalyse beobachtet.

Die Klassen eines Kandidaten besitzen in der Regel aber noch weitere Methoden, die in dem Entwurfsmuster keine Rolle spielen. Möglicherweise nehmen die Klassen eines Kandidaten an einer weiteren Entwurfsmusterimplementierung teil, die ebenfalls analysiert werden soll. Dann müssen weitere Methoden der Klassen beobachtet werden. Das kann dazu führen, dass Methoden des einen Entwurfsmusters zur Laufzeit unter Umständen verschränkt mit den Methoden eines anderen Entwurfsmusters ausgeführt und beobachtet werden. Das bedeutet, zwischen Aufrufen der Methoden eines Entwurfsmusters finden weitere Aufrufe anderer Methoden statt. Diese Methodenaufrufe sollen jedoch weder bei der Spezifikation eines Verhaltensmusters, noch bei der Analyse des Verhaltens berücksichtigt werden.

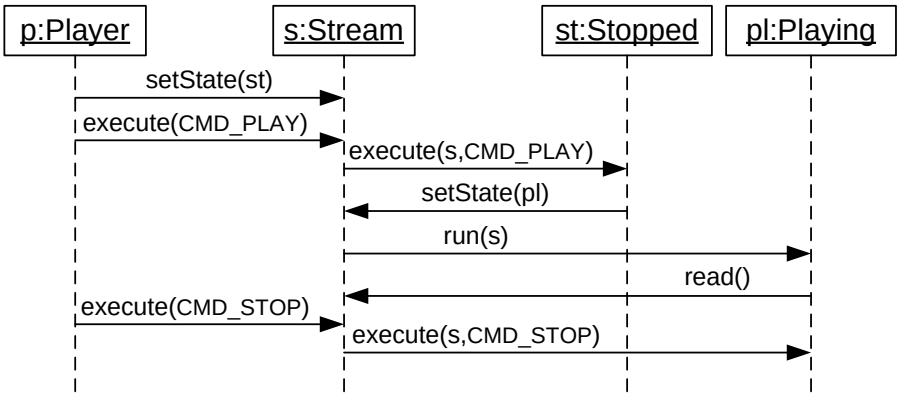

Abbildung 4.12: Beobachteter Trace des Mediaplayers

Der Trace aus Abbildung 4.12 entspricht im Wesentlichen dem zum *State*-Verhaltensmuster konformen Trace aus Abbildung 4.10 (Seite 76). Im Unterschied zum letzteren wurden bei diesem aber zwei zusätzliche Methodenaufrufe zwischen den Instanzen der Entwurfsmusterimplementierung beobachtet. Die beiden Methodenaufrufe run(s) und read() zwischen den Instanzen s und pl werden jedoch im Verhaltensmuster nicht genannt. Trotz dieser zusätzlichen Methodenaufrufe soll der beobachtete Trace konform zum Verhaltensmuster sein.

Aus diesem Grund werden in Traces nur solche Methodenaufrufe berücksichtigt, deren Methoden in Nachrichten des Verhaltensmusters verwendet werden. Im Beispiel des *State*-Verhaltensmuster werden also nur Aufrufe der Methoden Stream::setState(StreamState), Stream::execute(int) und StreamState::execute(Stream, int) berücksichtigt, wohingegen zum Beispiel Aufrufe der Methoden StreamState::run(Stream) oder auch Stream::read() bei der Überprüfung der Konformität eines Traces ignoriert werden.

In Sequenzdiagrammen nach UML 2.0 gibt es zwei spezielle kombinierte Fragmente namens *relevante Nachrichten* (Operator: *consider*) und *irrelevante Nachrichten* (Operator: *ignore*), mit denen gezielt Nachrichten in einer Teilsequenz berücksichtigt beziehungsweise ignoriert werden können. Mit Hilfe des kombinierten Fragments *relevante Nachrichten* werden Teilsequenzen von Nachrichten bestimmter, vorgegebener Methoden spezifiziert. Die Methoden der zu berücksichtigenden, relevanten Nachrichten werden in einer Menge hinter dem Operator *consider* des Fragments angegeben. Nachrichten dieser Methoden dürfen innerhalb der Teilsequenz nur in der spezifizierten Weise gesendet werden. Über Nachrichten aller anderen, nicht genannten Methoden wird in der innerhalb des Fragments spezifizierten Teilsequenz keine Aussage getroffen, sie werden deshalb ignoriert.

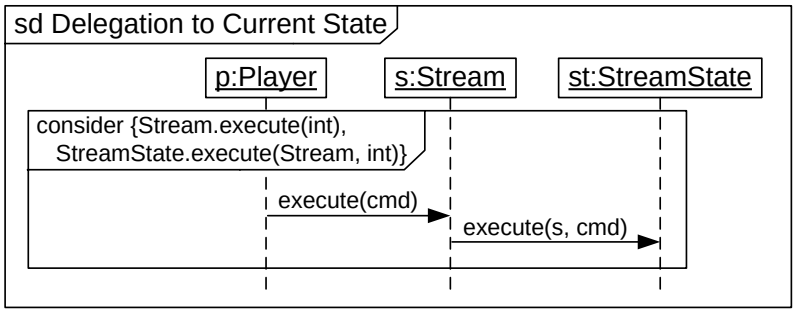

Abbildung 4.13: Sequenzdiagramm des Mediaplayers

In Abbildung 4.13 ist ein Sequenzdiagramm des Mediaplayers zu sehen, das die Delegation eines Aufrufs von execute(int) an die Methode execute(Stream, int) innerhalb eines *consider*-Fragments spezifiziert. In dem Fragment werden Nachrichten genau dieser beiden Methoden berücksichtigt. Durch das relevante-Nachrichten-Fragment wird festgelegt, dass zuerst eine Nachricht execute(int) vom Objekt p:Player an das Objekt s:Stream gesendet wird. Anschließend muss die Nachricht execute(Stream, int) von s:Stream an st:StreamState gesendet werden. Innerhalb dieser Teilsequenz darf kein andersartiger Aufruf

der beiden genannten Methoden erfolgen. Zum Beispiel darf keine Nachricht execute(Stream, int) an das Objekt st:StreamState durch p:Player geschickt werden. Nachrichten anderer, im *consider*-Operator nicht genannter Methoden dürfen jedoch an beliebiger Stelle in dieser Teilsequenz erfolgen, sie werden in dieser Spezifikation ignoriert.

Beim irrelevante-Nachrichten-Fragment sind im Gegensatz dazu explizit die zu ignorierenden Methoden festgelegt. Die in der Teilsequenz spezifizierten Nachrichten müssen exakt in der gegebenen Form erfolgen. Nachrichten, die nicht explizit in der Menge des *ignore*-Operators genannt werden, dürfen zu keinem Zeitpunkt innerhalb der gegebenen Teilsequenz gesendet werden. Nur Nachrichten der explizit genannten Methoden dürfen an beliebiger Stelle versendet werden.

Die Semantik des *consider*-Fragments entspricht also genau der Semantik, die implizit für Verhaltensmuster gilt. Die gewünschte Semantik der Verhaltensmuster erhielte man daher auch, wenn man die gesamte Sequenz des Verhaltensmuster durch ein relevante-Nachrichten–Fragment umschließe. Aus Gründen der Vereinfachung wird jedoch darauf verzichtet und die *consider*-Semantik implizit auf das gesamte Verhaltensmuster angewendet.

Nicht-konforme Traces

Ein nicht-konformer Trace ist dagegen in Abbildung 4.14 dargestellt. Die ersten drei Methodenaufrufe sind konform zu den im Verhaltensmuster geforderten Nachrichten. Dann wird allerdings der Zustandswechsel durch die Instanz p ausgelöst. Dieser Methodenaufruf ist zu keiner Nachricht, die zu diesem Zeitpunkt erlaubt ist, konform. Der Trace ist deshalb nicht konform zum *State*-Verhaltensmuster und steht somit im Widerspruch zum *State*-Entwurfsmuster.

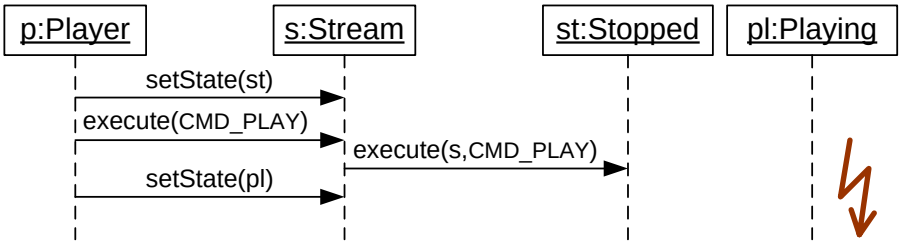

Abbildung 4.14: Zum *State*-Verhaltensmuster nicht-konformer Trace

Auch der in Abbildung 4.15 gezeigte Trace ist nicht konform zum *State*-Verhaltensmuster. In diesem Fall wird aber der Zustandswechsel von einem

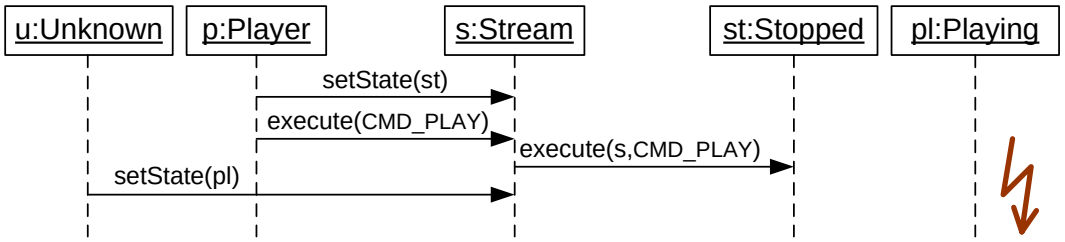

Abbildung 4.15: Unerlaubter Aufrufer der Nachricht setState

unbekannten Objekt ausgelöst, das nicht im Verhaltensmuster spezifiziert wurde und demnach nicht erlaubt ist.

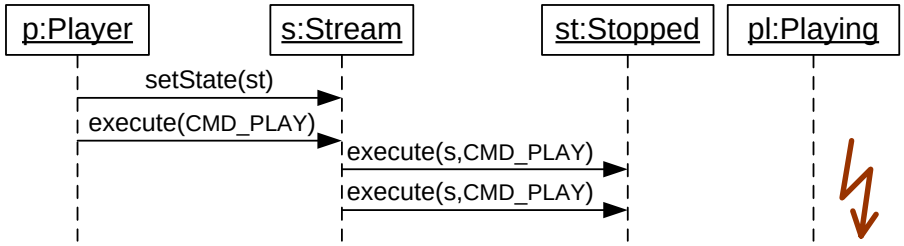

Abbildung 4.16: Unerlaubte Nachricht

Abbildung 4.16 zeigt einen Trace, bei dem die handle-Methode des *State*-Verhaltensmuster zu einem Zeitpunkt aufgerufen wird, zu dem sie nicht aufgerufen werden darf. Der Trace ist ebenfalls nicht konform zum Verhaltensmuster.

4.3.5 Wertung konformer und nicht-konformer Traces

Ist ein Trace zu einem positiven Verhaltensmuster konform, so wird dieser Trace als positives Indiz für den Kandidaten gewertet. Wird jedoch ein Trace beobachtet, der nicht konform zum positiven Verhaltensmuster ist, so wird er als negatives Indiz gewertet. Indizien sind nur Hinweise für oder gegen die Vermutung, ein Kandidat sei eine tatsächliche Entwurfsmusterimplementierung. Indizien sind keine absoluten Aussagen.

Bei einem negativen Verhaltensmuster werden nur Traces gewertet, die konform zum negativen Verhaltensmuster sind. Solche Traces werden dann als negatives Indiz angesehen. Verstößt ein Trace gegen ein negatives Verhaltensmuster, so kann man daraus nicht schließen, dass der Kandidat sich dem Ent-

wurfsmuster entsprechend verhält. Daher können aus einem solchen Trace keine Rückschlüsse auf den Kandidaten gezogen werden. Ein Trace, der zu einem negativen Verhaltensmuster nicht-konform ist, wird deshalb ignoriert.

Die Interpretation der positiven und negativen Indizien, die zu einem Kandidaten durch die Verhaltensanalyse gesammelt wurden, wird dem Reverse-Engineer überlassen. Bei einem deutlichen Überwiegen der positiven Indizien ist die Wahrscheinlichkeit, dass der Kandidat eine tatsächliche Entwurfsmusterimplementierung ist, sehr hoch. Überwiegen dagegen die negativen Indizien, so ist die Wahrscheinlichkeit, dass der Kandidat ein False-Positive ist, höher.

4.4 Erzeugung eines Automaten

Die formale Definition der Semantik der Verhaltensmuster erfolgt durch die Spezifikation einer inkrementellen Transformation eines Verhaltensmusters in einen *endlichen Automaten*. Zunächst wird das Verhaltensmuster in einen nichtdeterministischen, endlichen Automaten transformiert. Dieser Automat wird anschließend in einen deterministischen, endlichen Automaten umgewandelt und um zusätzliche Transitionen erweitert. Die Erweiterung gleicht einer Vervollständigung des endlichen Automaten, wird aber nur teilweise durchgeführt. Das Ergebnis der Transformation ist ein deterministischer Automat, der zur algorithmischen Erkennung des Verhaltensmusters verwendet werden kann, indem er als Eingabe die Methodenaufrufe eines Traces erhält [WO06]. Der Automat kann nach Verarbeitung der Methodenaufrufe feststellen, ob der Trace konform oder nicht-konform zu dem Verhaltensmuster ist, das er repräsentiert.

4.4.1 Nichtdeterministischer Automat

Bei der inkrementellen Transformation wird für jedes Interaktionsfragment der Verhaltensmuster eine Transformation in Elemente eines nichtdeterministischen, endlichen Automaten (NFA) angegeben. Zur Übersetzung eines vollständigen Verhaltensmusters in einen NFA werden alle Interaktionsfragmente des Verhaltensmusters, also alle Nachrichten und kombinierten Fragmente, nach ihrer gegebenen Ordnung in Zustände, Transitionen und Symbole des NFA transformiert und daraus ein vollständiger NFA erzeugt.

Ein *nichtdeterministischer, endlicher Automat* ist definiert durch eine endliche Menge Q von *Zuständen*, eine Menge Σ von *Eingabesymbolen* (auch *Alphabet* genannt), einen *Startzustand* q_0, eine Menge $F \subseteq Q$ von *akzeptierenden*

Zuständen sowie durch die *Transitionsfunktion* δ, die einen Zustand aus Q und ein Symbol aus Σ auf eine Teilmenge aus Q abbildet.

Interaktionsfragment

Jedes Interaktionsfragment des Verhaltensmusters wird in ein Teilkonstrukt des letztendlich resultierenden NFA transformiert, das zwei Zustände enthält: einen Anfangszustand $s_0 \in Q$ und einen Endzustand $s_e \in Q$.

Konkatenation der Interaktionsfragmente

Die aus den Interaktionsfragmenten entstehenden Teilkonstrukte werden anhand der Reihenfolge der Interaktionsfragmente im Verhaltensmuster über ϵ-Transitionen konkateniert. ϵ ist das *leere Wort*. Die Transformation ist in Abbildung 4.17 anschaulich dargestellt.

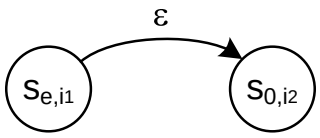

Abbildung 4.17: Konkatenation zweier Interaktionsfragmente

Definition 4.3 *Die Reihenfolge der Interaktionsfragmente innerhalb eines Operanden ist im Metamodell der Verhaltensmuster über die geordnete Assoziation* fragments *zwischen* InteractionOperand *und* InteractionFragment *definiert.*

Transformationsregel 4.1 *Ist ein Interaktionsfragment $i_1 \in I$ der direkte Vorgänger des Interaktionsfragments $i_2 \in I$ in einem Operanden, so wird der Endzustand $s_{e,i_1} \in Q$ von i_1 durch eine ϵ-Transition mit dem Anfangszustand $s_{0,i_2} \in Q$ von i_2 verbunden:*
$s_{0,i_2} \in \delta(s_{e,i_1}, \epsilon)$

Nachricht

Das wichtigste Interaktionsfragment der Verhaltensmuster ist die Nachricht. Die Nachrichten eines Verhaltensmusters definieren nicht nur maßgeblich die Menge Σ der Eingabesymbole des NFA, sondern auch die Transitionsfunktion

δ. Eine Nachricht wird in zwei Zustände übersetzt, die durch eine Transition verbunden sind. Das Symbol, das von der Transition akzeptiert wird, repräsentiert die Nachricht. In Abbildung 4.18 ist die Transformation wiederum anschaulich dargestellt.

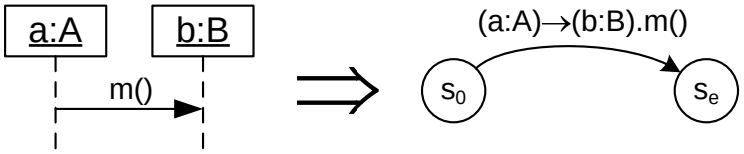

Abbildung 4.18: Transformation einer Nachricht

Transformationsregel 4.2 *Eine Nachricht, in der das typisierte Verhaltensmusterobjekt* a:A *die Methode* m *auf dem typisierten Verhaltensmusterobjekt* b:B *aufruft, wird in zwei Zustände $s_0 \in Q$ und $s_e \in Q$ transformiert, die über eine Transition mit dem Symbol $(a : A) \rightarrow (b : B).m() \in \Sigma$ verbunden werden:*
$\delta(s_0, (a : A) \rightarrow (b : B).m()) = \{s_e\}$

Das Symbol des Automaten ist also aus denselben Variablen zusammengesetzt wie die Nachricht des Verhaltensmusters. Die Bindung der Variablen des Verhaltensmusters gilt somit auch für das Symbol des Automaten. Befindet sich ein Automat während der Verhaltensanalyse in dem Zustand s_0 und erhält er einen Methodenaufruf mc des Typs Behavior::MethodCall als Eingabe, so kann anhand der Bindung der Variablen geprüft werden, ob der Methodenaufruf konform zu dem Symbol ist. Die Konformität eines Methodenaufrufs zu einem Symbol ist analog zu der Konformität eines Methodenaufrufs zu einer Nachricht nach den Bedingungen 1 bis 5 aus Abschnitt 4.3.3 definiert, da das Symbol aus denselben Variablen besteht wie die zugehörige Nachricht. Ist ein Methodenaufruf also konform zu dem Symbol, so kann der Automat über die zum Symbol gehörige Transition in den Zustand s_e wechseln.

Analog gelten diese Aussagen auch für Nachrichten, die von einem untypisierten Verhaltensmusterobjekt gesendet werden. Die folgende Definition gibt die Transformation für eine solche Nachricht an.

Transformationsregel 4.3 *Eine Nachricht, in der das untypisierte Verhaltensmusterobjekt* a *die Methode* m *auf dem typisierten Verhaltensmusterobjekt* b:B *aufruft, wird in zwei Zustände $s_0 \in Q$ und $s_e \in Q$ transformiert, die über eine Transition mit dem Symbol $a \rightarrow (b : B).m() \in \Sigma$ verbunden werden:*
$\delta(s_0, a \rightarrow (b : B).m()) = \{s_e\}$

Operand

Jedes kombinierte Fragment eines Verhaltensmusters enthält mindestens einen Operanden, der wiederum mindestens ein Interaktionsfragment, aber beliebig viele, untereinander geordnete Interaktionsfragmente enthält.

Definition 4.4 *Der Anfangszustand $s_{0,o}$ eines Operanden ist definiert als der Anfangszustand s_{0,i_f} seines ersten Interaktionsfragments i_f. Der Endzustand $s_{e,o} \in Q$ eines Operanden ist definiert als der Endzustand $s_{e,i_l} \in Q$ seines letzten Interaktionsfragments i_l:*

$s_{0,o}{=}s_{0,i_f}$
$s_{e,o}{=}s_{e,i_l}$

Alternatives Fragment

Ein alternatives Fragment wird in zwei Zustände s_0 und s_e transformiert, die über ϵ-Transitionen mit den Anfangs- und Endzuständen der Operanden des alternativen Fragments verbunden werden. Als Beispiel ist die Transformation eines alternativen Fragments mit zwei Operanden in Abbildung 4.19 dargestellt. Die gestrichelten Transitionen in der Abbildung sind Platzhalter für die Teilkonstrukte des NFA, die aus den Operanden transformiert und eingesetzt werden.

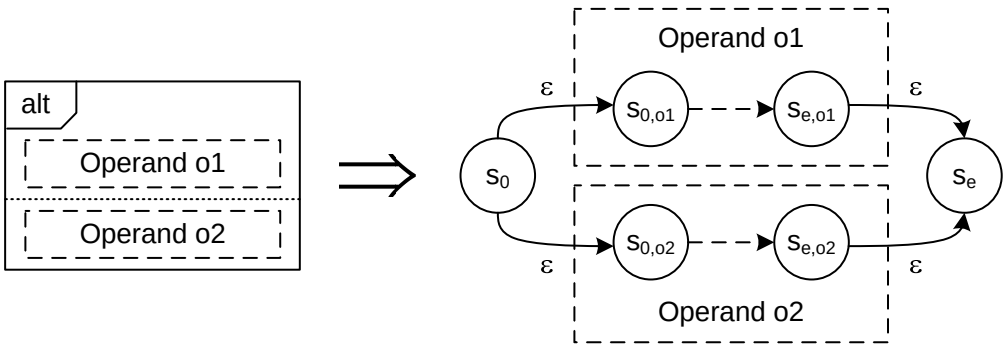

Abbildung 4.19: Transformation eines alternativen Fragments mit zwei Operanden

Definition 4.5 *Die Menge der Operanden $\mathcal{O}(a)$ eines alternativen Fragments a ist im Metamodell der Verhaltensmuster definiert durch die Assoziation operands zwischen CombinedFragment und InteractionOperand.*

Transformationsregel 4.4 *Sei $\mathcal{O}(a)$ die Menge der Operanden eines alternativen Fragments a. Der Anfangszustand $s_0 \in Q$ des alternativen Fragments wird jeweils über eine ϵ-Transition mit dem Anfangszustand $s_{0,o} \in Q$ jedes seiner Operanden $o \in \mathcal{O}(a)$ verbunden. Der Endzustand $s_{e,o} \in Q$ jedes Operanden $o \in \mathcal{O}(a)$ wird ebenfalls über eine ϵ-Transition mit dem Endzustand $s_e \in Q$ des alternativen Fragments verbunden:*

$\forall o \in \mathcal{O} : s_{0,o} \in \delta(s_0, \epsilon), s_{0,o} \in Q$

$\forall o \in \mathcal{O} : s_e \in \delta(s_{e,o}, \epsilon), s_{e,o} \in Q$

Optionales Fragment

Aus einem optionalen Fragment werden ein Anfangs- und ein Endzustand erzeugt, die über eine ϵ-Transition miteinander verbunden werden. Die ϵ-Transition ermöglicht, das optionale Fragment im Automaten zu überspringen. Des Weiteren wird das durch den Operanden des optionalen Fragments erzeugte Teilkonstrukt wie bei dem alternativen Fragment zwischen Anfangs- und Endzustand des optionalen Fragments eingefügt. In Abbildung 4.20 ist die gestrichelte Transition ebenfalls ein Platzhalter für das Teilkonstrukt des Operanden.

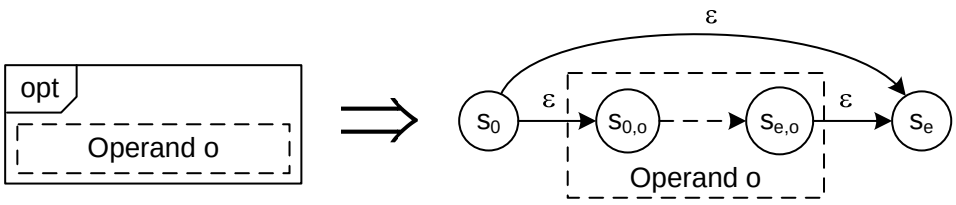

Abbildung 4.20: Transformation eines optionalen Fragments

Transformationsregel 4.5 *Der Anfangszustand $s_0 \in Q$ des optionalen Fragments wird über eine ϵ-Transition mit dem Endzustand $s_e \in Q$ verbunden. Des Weiteren wird der Anfangszustand s_0 des optionalen Fragments über eine ϵ-Transition mit dem Anfangszustand $s_{0,o} \in Q$ seines Operanden o verbunden. Ebenso wird der Endzustand $s_{e,o} \in Q$ des Operanden o über eine ϵ-Transition mit dem Endzustand s_e des optionalen Fragments verbunden:*

$\delta(s_0, \epsilon) = \{s_e, s_{0,o}\}$

$s_e \in \delta(s_{e,o}, \epsilon)$

Schleife

Bei der Transformation einer Schleife werden zwei Varianten unterschieden, Schleifen mit Untergrenze 0 und Schleifen mit Untergrenze 1. Bei Schleifen mit Untergrenze 0 wird eine ϵ-Transition vom Anfangs- zum Endzustand erzeugt, die bei Schleifen mit Untergrenze 1 fehlt. Der Operand der Schleife wird, wie in den beiden Abbildungen 4.21 und 4.22 dargestellt, analog zu den anderen kombinierten Fragmenten zwischen Anfangs- und Endzustand der Schleife eingefügt.

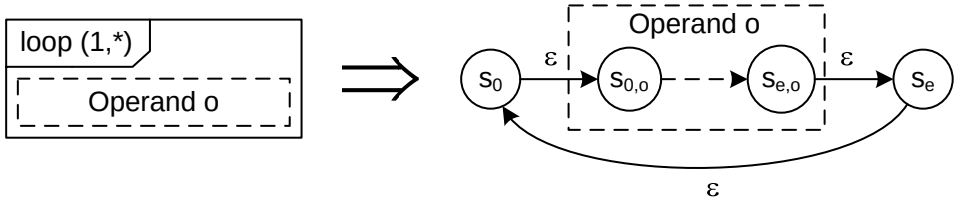

Abbildung 4.21: Transformation einer Schleife mit mindestens einem aber beliebig vielen Durchläufen

Transformationsregel 4.6 *Eine Schleife wird in einen Anfangszustand $s_0 \in Q$ und einen Endzustand $s_e \in Q$ transformiert. Zur Wiederholung der Schleife wird eine ϵ-Transition vom Endzustand zum Anfangszustand der Schleife erzeugt. Der Anfangszustand s_0 der Schleife wird über eine ϵ-Transition mit dem Anfangszustand $s_{0,o} \in Q$ ihres Operanden o verbunden. Der Endzustand $s_{e,o} \in Q$ des Operanden o wird ebenfalls über eine ϵ-Transition mit dem Endzustand s_e der Schleife verbunden:*

$s_0 \in \delta(s_e, \epsilon)$
$\delta(s_0, \epsilon) = \{s_{0,o}\}$
$s_e \in \delta(s_{e,o}, \epsilon)$

Transformationsregel 4.7 *Bei einer Schleife mit der Untergrenze 0 wird zusätzlich der Anfangszustand s_0 der Schleife über eine ϵ-Transition mit dem Endzustand s_e der Schleife verbunden:*

$s_e \in \delta(s_0, \epsilon)$

Startzustand und akzeptierender Zustand des NFA

Zur vollständigen Spezifikation des NFA fehlt noch die Spezifikation des Startzustands q_0 und der Menge der akzeptierenden Zustände $F \subseteq Q$.

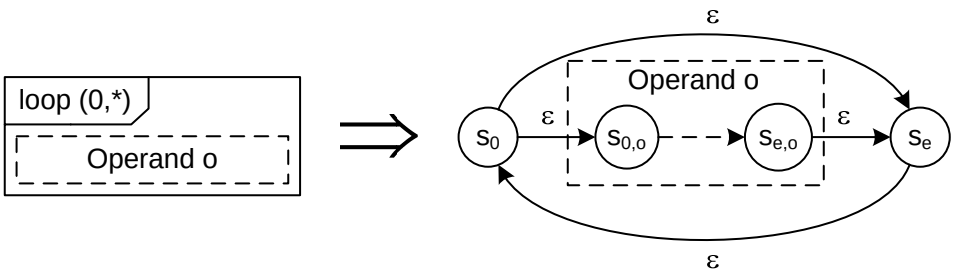

Abbildung 4.22: Transformation einer Schleife mit beliebig vielen Durchläufen

Definition 4.6 *Das erste Interaktionsfragment eines Verhaltensmusters ist definiert als das erste Interaktionsfragment des Operanden des Wurzel-Fragments. Das letzte Interaktionsfragment eines Verhaltensmusters ist analog definiert als das letzte Interaktionsfragment des Operanden des Wurzel-Fragments.*

Transformationsregel 4.8 *Der Anfangszustand $s_{0,i_f} \in Q$ des ersten Interaktionsfragments des Verhaltensmusters wird zum Startzustand q_0 des aus dem Verhaltensmuster transformierten NFA. Der einzige akzeptierende Zustand des NFA ist der Endzustand $s_{e,i_l} \in Q$ des letzten Interaktionsfragments des Verhaltensmusters:*

$q_0 = s_{0,i_f}$
$F = \{s_{e,i_l}\}$

Transformationsalgorithmus

In Abbildung 4.23 ist der Algorithmus zur Transformation eines Verhaltensmusters in einen NFA mit Hilfe einer Pseudocode-Syntax angegeben. Der Algorithmus definiert, in welcher Reihenfolge die Elemente des Verhaltensmusters anhand der Transformationsregeln 4.1 bis 4.8 in Konstrukte des NFA transformiert werden.

Der Algorithmus wird auf dem Wurzel-Fragment des Verhaltensmusters gestartet (Zeile 2). Jedes kombinierte Fragment transformiert zunächst seine Operanden (Zeilen 5 und 6), erst dann wird das kombinierte Fragment selber anhand seiner spezifischen Regel 4.4, 4.5, 4.6 beziehungsweise 4.7 transformiert (Zeile 7).

Die Interaktionsfragmente werden anhand ihrer gegebenen Ordnung innerhalb ihres Operanden transformiert (Zeile 10). Durch polymorphe Methodenbindung werden kombinierte Fragmente und Nachrichten unterschieden. Eine

```
 1: BehavioralPattern::constructNFA()
 2:   self.root.constructNFA()
 3:   transform initial state and accepting state by rule 4.8

 4: CombinedFragment::constructNFA()
 5:   forEach operand:InteractionOperand in self.operands do
 6:     operand.constructNFA()
 7:   transform self by rule 4.4, 4.5, 4.6 or 4.7

 8: InteractionOperand::constructNFA()
 9:   let previous:InteractionFragment = OCLVoid
10:   forEach current:InteractionFragment in self.fragments do
11:     current.constructNFA()
12:     if previous<>OCLVoid then
13:       concatenate previous with current by rule 4.1
14:     previous = current

15: Message::constructNFA()
16:   transform self by rule 4.2 or 4.3
```

Abbildung 4.23: Algorithmus zur Transformation eines Verhaltensmusters in einen NFA

Nachricht wird nach den Regeln 4.2 beziehungsweise 4.3 transformiert (Zeile 16), ein kombiniertes Fragment wird wie oben erläutert transformiert. Die bereits transformierten Interaktionsfragmente werden anhand ihrer Ordnung nach Regel 4.1 konkateniert (Zeile 13).

Nach der Transformation des Wurzel-Fragments werden schließlich der Startzustand und der akzeptierende Zustand des Automaten nach Regel 4.8 bestimmt (Zeile 3). Nach Ausführung dieses Algorithmus erhält man den aus einem Verhaltensmuster erzeugten NFA.

Beispiel

Mit Hilfe der oben genannten Transformationsregeln wurde das *State*-Verhaltensmuster aus Abbildung 4.1 in einen NFA umgewandelt. Das Ergebnis ist in Abbildung 4.24 zu sehen.

Die Zustände 0 bis 3 wurden aus dem optionalen Fragment am Beginn des

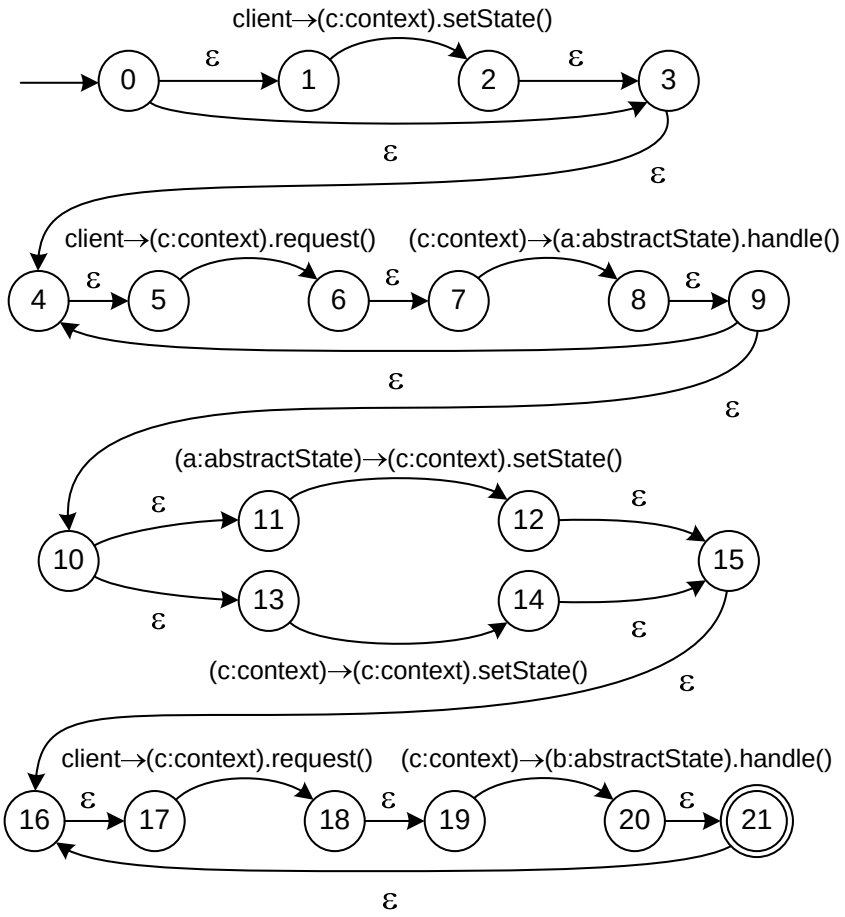

Abbildung 4.24: Nichtdeterministischer, endlicher Automat für das *State*-Verhaltensmuster

Verhaltensmusters erzeugt. Die Nachricht innerhalb des optionalen Fragments wurde in die Zustände 1 und 2 transformiert. In den Zuständen 4 bis 9 wurde die erste Schleife mit zwei hintereinander gesendeten Nachrichten kodiert. Die Zustände 10 bis 15 bilden das alternative Fragment mit den zwei Operanden. Die zweite Schleife wurde in den Zuständen 16 bis 21 analog zu der ersten Schleife kodiert.

Das optionale Fragment ist das erste Interaktionsfragment des *State*-Verhaltensmusters. Sein Anfangszustand ist der Startzustand des NFA. Das letzte Interaktionsfragment ist die zweite Schleife. Der Endzustand der zweiten Schleife ist deshalb der akzeptierende Zustand des Automaten. Das Alphabet des NFA

besteht aus dem leeren Wort ϵ und den Symbolen, die aus den Nachrichten erzeugt wurden.

4.4.2 Deterministischer Automat

Der NFA kann wegen des Nichtdeterminismus noch nicht zur algorithmischen Überprüfung der Verhaltensmuster verwendet werden. Er lässt sich allerdings mit polynomiellen Aufwand in einen äquivalenten, *deterministischen, endlichen Automaten* (DFA) umwandeln.

Ein *deterministischer, endlicher Automat* ist wie ein NFA definiert durch eine endliche Menge Q von *Zuständen*, eine Menge Σ von *Eingabesymbolen*, einen *Startzustand* q_0, eine Menge $F \subseteq Q$ von *akzeptierenden Zuständen* sowie durch die *Transitionsfunktion* δ, die jedoch im Gegensatz zum NFA einen Zustand aus Q und ein Symbol aus Σ auf einen einzelnen Zustand aus Q abbildet.

Ein DFA, der aus einem NFA für ein Verhaltensmuster entstanden ist, ist in der Lage, einen zu diesem Verhaltensmuster konformen Trace zu akzeptieren. Es wird im Folgenden davon ausgegangen, dass Methodenaufrufe, die nicht in der Menge der Eingabesymbole enthalten sind, vom DFA ignoriert werden. So werden alle Methodenaufrufe vom DFA ignoriert, die auch von dem Verhaltensmuster nicht berücksichtigt werden. Im Folgenden wird nun untersucht, wie ein solcher DFA Traces verarbeitet, die nicht konform zum Verhaltensmuster sind.

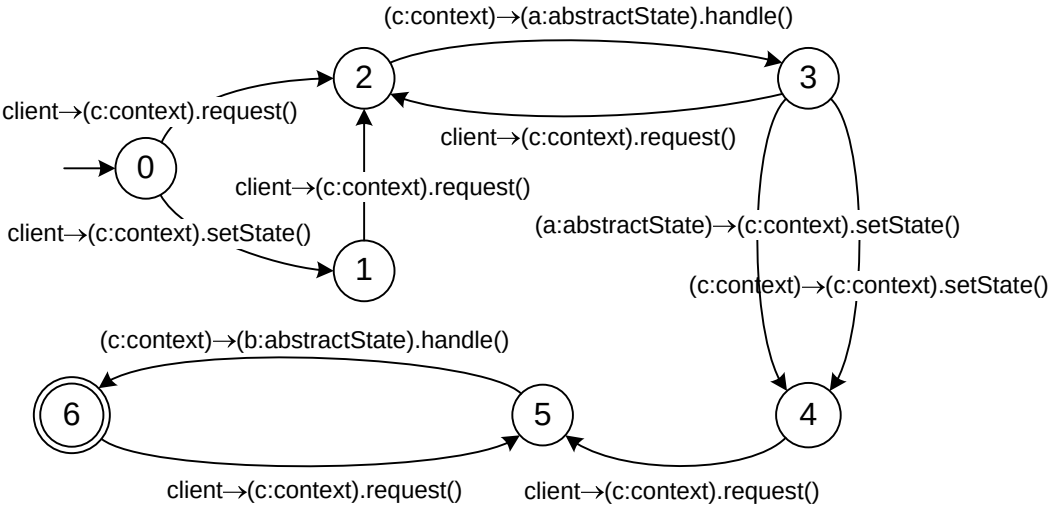

Abbildung 4.25: Deterministischer Automat für das *State*-Verhaltensmuster

In Abbildung 4.25 ist der DFA dargestellt, der aus dem NFA aus Abbildung

4.24 für das *State*-Verhaltensmuster entstanden ist. Anhand dieses Beispiels werden nun zwei zum *State*-Verhaltensmuster nicht-konforme Traces untersucht.

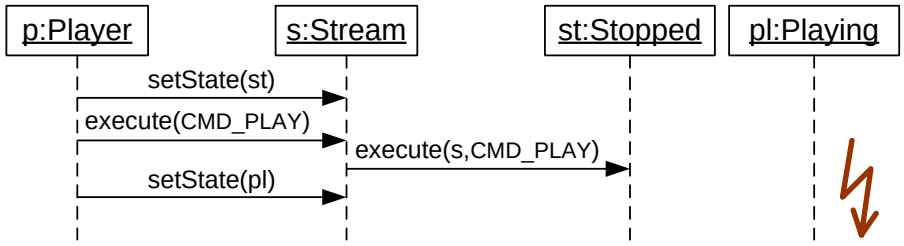

Abbildung 4.26: Zum *State*-Verhaltensmuster nicht-konformer Trace

Der in Abbildung 4.26 dargestellte Trace ist bereits in Abschnitt 4.3 vorgestellt worden. Zur Verhaltensanalyse werden zunächst die Variablen der Typ- und Methodennamen des DFA aus Abbildung 4.25 an die Klassen und Methoden des zum Trace gehörenden Kandidaten gebunden, wie in Abschnitt 4.3.2 erläutert. Zur Vereinfachung sei auch hier davon ausgegangen, dass die Verhaltensmusterobjekte bereits zu Beginn der Verhaltensanalyse nichtdeterministisch an Instanzen aus der Laufzeitumgebung gebunden werden. Bei der tatsächlichen Verhaltensanalyse werden die Verhaltensmusterobjekte erst sukzessive während der Analyse gebunden. Die Bindung in diesem Beispiel ist die gleiche, wie in Tabelle 4.2 auf Seite 77 aufgelistet.

Erhält der Automat nun die ersten drei Methodenaufrufe des Traces als Eingabe, so wechselt er vom Startzustand 0 in den Zustand 3. Der vierte Methodenaufruf (p:Player)→(s.Stream).setState(pl) kann jedoch im Zustand 3 nicht von dem DFA verarbeitet werden, da es in diesem Zustand keine Transition mit einem Symbol gibt, zu dem der Methodenaufruf konform ist. Das Symbol, zu dem der Methodenaufruf konform wäre, ist $client \rightarrow (c : context).setState()$. Da dieses Symbol zum Alphabet des DFA gehört, verharrt der DFA aber in diesem Fall in einem nicht akzeptierenden Zustand. Der Trace wird also nicht vom DFA akzeptiert.

Der zweite Trace aus Abbildung 4.27 ist ebenfalls nicht konform zum *State*-Verhaltensmuster. Der DFA verarbeitet, wie beim ersten Beispiel, die ersten drei Methodenaufrufe und befindet sich dann im Zustand 3. Für den nun folgenden Methodenaufruf (u:Unknown)→(s:Stream).setState(pl) existiert allerdings im Alphabet des DFA kein Symbol. Für einen Aufruf der Methode Stream.setState(StreamState) existieren nur die beiden Symbole (c :

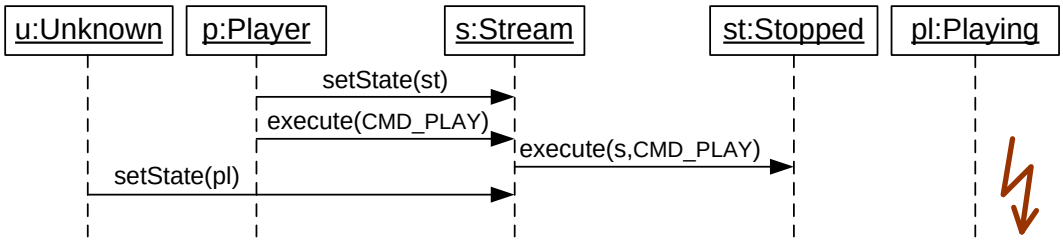

Abbildung 4.27: Alternativer, zum *State*-Verhaltensmuster nicht-konformer Trace

context) → (*c* : *context*).*setState*() und (*a* : *abstractState*) → (*c* : *context*) .*setState*(). Die aufrufenden Objekte der beiden Symbole passen jedoch nicht zu der aufrufenden Instanz u:Unknown, da die Typen nicht konform zueinander sind. Zu diesem Methodenaufruf existiert also kein passendes Symbol des Alphabets des DFA, er würde daher einfach vom DFA ignoriert. Der DFA kann also weitere Methodenaufrufe entgegen nehmen und später den Trace eventuell akzeptieren.

Um dies zu verhindern, müssen das Alphabet des DFA erweitert und zusätzliche Transitionen hinzugefügt werden, die verhindern, dass zum Verhaltensmuster nicht-konforme Traces vom DFA akzeptiert werden.

Erweiterung des deterministischen Automaten

Zu diesem Zweck wird in den aus dem NFA entstandenen DFA ein zusätzlicher, nicht akzeptierender Zustand eingefügt, der eine Senke innerhalb des DFA ist. Der DFA soll diesen Zustand erreichen, wenn der von ihm untersuchte Trace nicht konform zum Verhaltensmuster ist. Da der Zustand eine Senke ist, kann der Automat den Zustand nicht wieder verlassen. Ein einmal verworfener Trace kann also niemals akzeptiert werden. Im Folgenden wird diese Senke deshalb auch als *verwerfender Zustand* bezeichnet[4]. In Abbildungen wird die Senke mit einem R (Reject) versehen. Die folgende Regel führt die Senke ein.

Transformationsregel 4.9 *Sei D ein DFA, der aus dem NFA eines Verhaltensmusters entstanden ist. Der DFA D wird ergänzt um einen nicht-akzeptierenden Zustand* $r \in Q/F$*, der eine Senke innerhalb des DFA D ist:*

$\forall \sigma \in \Sigma : \delta(r, \sigma) = r$

[4]In der Literatur findet man auch die Bezeichnung *Fangzustand* (engl. trap state).

Des Weiteren wird das Alphabet des DFA um zusätzliche Symbole ergänzt. Diese Symbole haben die Form $* \rightarrow (b : B).m()$ beziehungsweise $* \notin \mathcal{C} \rightarrow (b : B).m()$. Ein Symbol der Form $* \rightarrow (b : B).m()$ repräsentiert einen Aufruf der Methode m() auf dem Verhaltensmusterobjekt b:B durch einen beliebigen Aufrufer. Demgegenüber repräsentiert ein Symbol der Form $* \notin \mathcal{C} \rightarrow (b : B).m()$ einen Aufruf der Methode m() auf dem Verhaltensmusterobjekt b:B durch einen beliebigen Aufrufer, der nicht Element der Menge $\mathcal{C}$ ist, die Verhaltensmusterobjekte enthält.

Dem DFA werden Transitionen hinzugefügt, die diese Symbole akzeptieren und die nicht akzeptierende Zustände des DFA mit Ausnahme des Startzustands mit dem neuen, verwerfenden Zustand r verbinden. So führen nicht erlaubte Methodenaufrufe oder Methodenaufrufe, die durch einen nicht erlaubten Aufrufer ausgelöst wurden, zum Verwerfen des Traces durch den Automaten.

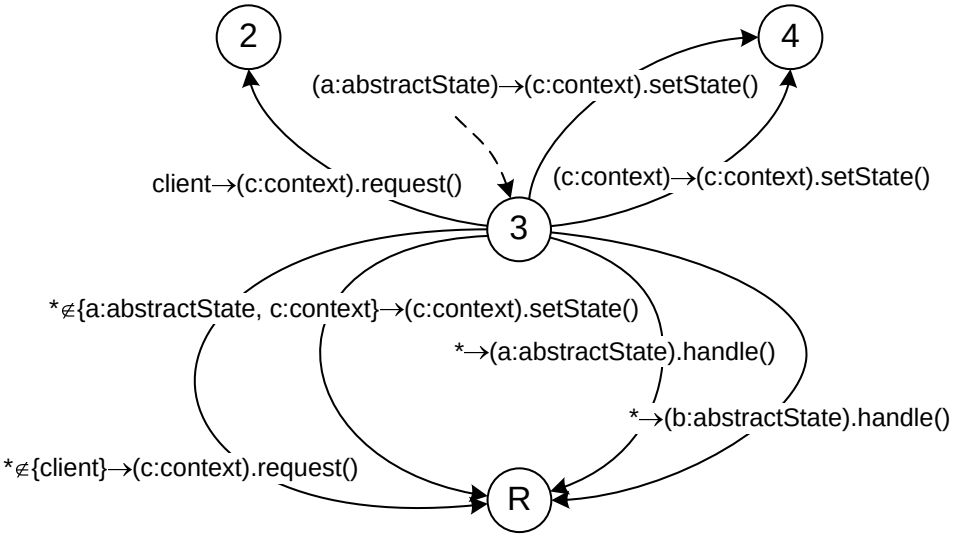

Abbildung 4.28: Ausschnitt aus dem um einen verwerfenden Zustand erweiterten DFA

Abbildung 4.28 zeigt nur einen kleinen Ausschnitt aus dem um einen verwerfenden Zustand und zusätzlichen Transitionen erweiterten DFA des *State*-Verhaltensmusters. Der vollständige DFA ist zu komplex, um ihn an dieser Stelle abzubilden. In dieser Abbildung ist zu sehen, welche Transitionen, ausgehend vom Zustand 3, dem Automaten hinzugefügt wurden. In der bisherigen Version konnte der DFA im Zustand 3 nur die drei Symbole $client \rightarrow (c : context).request()$, $(c : context) \rightarrow (c : context).setState()$

und $(a : abstractState) \rightarrow (c : context).setState()$ akzeptieren. Alle anderen relevanten Methodenaufrufe müssen jedoch zu einem Verwerfen des Traces führen. Aus diesem Grund werden zusätzliche Transitionen eingeführt, die den Zustand 3 mit der Senke verbinden und die das „Komplement“ der drei Symbole akzeptieren.

Zunächst einmal darf die Nachricht request() auf dem Verhaltensmusterobjekt c:context in diesem Zustand nur durch das Verhaltensmusterobjekt client gesendet werden. Alle anderen Aufrufer sind für diese Nachricht nicht erlaubt. Deshalb wird eine Transition vom Zustand 3 zum verwerfenden Zustand mit dem Symbol $* \notin \{client\} \rightarrow (c : context).request() \in \Sigma$ hinzugefügt.

Des Weiteren dürfen nur die Verhaltensmusterobjekte a:abstractState und c:context die Nachricht setState() an das Verhaltensmusterobjekt c:context senden. Daher wird der DFA um eine Transition vom Zustand 3 zum verwerfenden Zustand mit dem Symbol $* \notin \{a : abstractState, c : context\} \rightarrow (c : context)$ $.setState() \in \Sigma$ erweitert.

Es muss außerdem sichergestellt werden, dass Methodenaufrufe, die in diesem Zustand nicht erlaubt sind, zu einem Verwerfen des Traces führen. Dies sind alle Aufrufe von Methoden, die von dem Verhaltensmuster berücksichtigt werden, und auf den beteiligten Verhaltensmusterobjekte aufgerufen werden. Im Zustand 3 ist dies nur die Methode handle(), die auf den Verhaltensmusterobjekten a:abstractState und b:abstractState aufgerufen wird. Dem DFA werden deshalb zwei weitere Transitionen vom Zustand 3 zum verwerfenden Zustand mit den Symbolen $* \rightarrow (a : abstractState).handle() \in \Sigma$ und $* \rightarrow (b : abstractState).handle() \in \Sigma$ hinzugefügt.

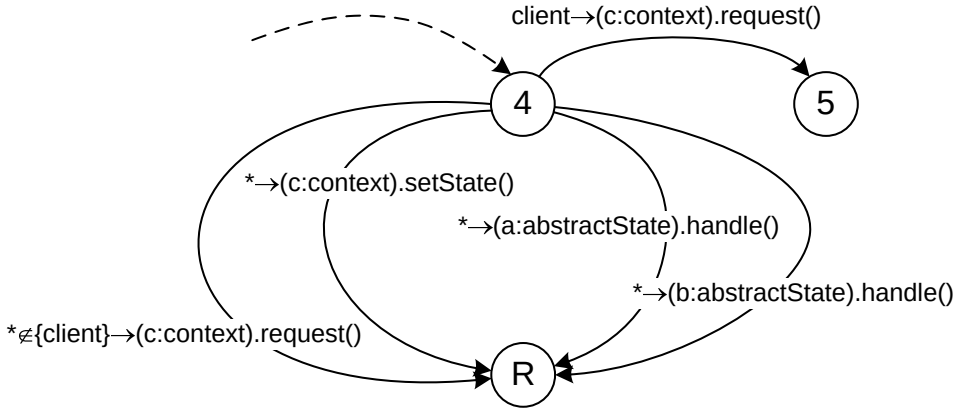

Abbildung 4.29: Erweiterung des Zustands 4 um zusätzliche Transitionen

Dieses Verfahren muss für alle nicht-akzeptierenden Zustände mit Ausnahme des Startzustands durchgeführt werden. In Abbildung 4.29 sind die zusätzlichen Transitionen zu sehen, die vom Zustand 4 zum verwerfenden Zustand führen.

Im Folgenden wird dieses Verfahren im Detail erläutert. Zunächst wird ein Algorithmus vorgestellt, der alle Nachrichten, die von den Verhaltensmusterobjekten eines Verhaltensmusters empfangen werden können, in einer Menge zusammen fasst. Die Menge enthält als Elemente Tupel, die aus dem Objekt- und dem Typnamen des aufgerufenen Verhaltensmusterobjekts sowie dem Methodennamen der Nachricht bestehen. Der Algorithmus ist in Abbildung 4.30 in Pseudocode-Syntax angegeben.

```
1: BehavioralPattern::receivedMessages():
       Set(TupleType(object:String, type:String, method:String))
2:   let result:Set(TupleType(object:String, type:String,
       method:String))
3:   forEach o:BPObject in self.objects do
4:     forEach m:Message in o.lifeline.received do
5:       result→add(Tuple(o.name, o.typeName, m.name))
6:   return result
```

Abbildung 4.30: Algorithmus zur Berechnung der aufgerufenen Nachrichten eines Verhaltensmusters

Der Algorithmus liefert zum Beispiel für die Instanz state:BehavioralPattern des *State*-Verhaltensmusters die folgende Menge:

state.receivedMessages() ={(a,abstractState,handle), (b,abstractState,handle), (c,context,request), (c,context,setState)}

Ein Tupel dieser Menge repräsentiert alle Nachrichten, die als Empfänger das im Tupel genannte Verhaltensmusterobjekt haben, und die die im Tupel genannte Methode aufrufen. Das Tupel (a,abstractState,handle) zum Beispiel repräsentiert alle Nachrichten, in denen die Methode handle() auf dem Verhaltensmusterobjekt a:abstractState aufgerufen wird.

Die nun folgenden Definitionen werden für die Transformationsregel benötigt, die den DFA um Transitionen ergänzt, die in einem Zustand unerlaubte Methodenaufrufe akzeptieren und den DFA damit in den verwerfenden Zustand führen. Als erstes wird die Menge $\mathcal{AS}_D(s)$ (*Accepted Symbols*) definiert, die als Elemente alle in einem Zustand s des DFAs D akzeptierten Symbole enthält.

Definition 4.7 *Die Menge $\mathcal{AS}_D(s)$ der in einem Zustand s eines DFA D akzeptierten Symbole ist definiert durch:*
$\mathcal{AS}_D(s) = \{\sigma \in \Sigma | \delta(s, \sigma) = s', s \in Q, s' \in Q/\{r\}\}$,
wobei r der verwerfende Zustand des DFA D ist.

Das bedeutet, $\mathcal{AS}_D(s)$ enthält alle Symbole, die die vom Zustand s ausgehenden Transitionen akzeptieren. Im DFA des *State*-Verhaltensmusters gilt zum Beispiel für den Zustand 3:
$\mathcal{AS}_{State}(3) = \{(client) \rightarrow (c : context).request(),$
$(c : context) \rightarrow (c : context).setState(),$
$(a : abstractState) \rightarrow (c : context).setState()\}$
Die nächste Definition ermöglicht den Zugriff auf die Variablen eines Symbols.

Definition 4.8 *Sei $\sigma \in \Sigma$ ein Symbol eines DFA D. Dann bezeichne σ_{caller} den Objektnamen und $\sigma_{callerType}$ den Typnamen des aufrufenden Verhaltensmusterobjekts des Symbols σ. Analog bezeichne σ_{callee} den Objektnamen und $\sigma_{calleeType}$ den Typnamen des aufgerufenen Verhaltensmusterobjekts des Symbols σ. σ_{method} bezeichne den Methodennamen des Symbols σ.*

Die Menge $\mathcal{RM}_D(s)$ der in einem Zustand s eines DFA D empfangbaren Nachrichten (*Receivable Messages*) enthält Tupel, die aus dem Objekt- und dem Typnamen des aufgerufenen Verhaltensmusterobjekts sowie dem Methodennamen des Symbols bestehen.

Definition 4.9 *Die Menge $\mathcal{RM}_D(s)$ wird definiert für einen Zustand s des DFA D. Die Elemente der Menge sind alle Tupel aus dem Objekt- und dem Typnamen des aufgerufenen Verhaltensmusterobjekts sowie den aufgerufenen Methodennamen, die in einem Symbol $\sigma \in \mathcal{AS}_D(s)$ enthalten sind:*
$\forall \sigma \in \mathcal{AS}_D(s) : (\sigma_{callee}, \sigma_{calleeType}, \sigma_{method}) \in \mathcal{RM}_D(s)$

Für den Zustand 3 des *State*-DFA gilt:
$\mathcal{RM}_{State}(3) = \{(c, context, request), (c, context, setState)\}$
Die folgende Transformationsregel fügt nun dem DFA Symbole der Form $* \rightarrow (b : B).m()$ und Transitionen, die diese Symbole akzeptieren und den DFA damit in den verwerfenden Zustand führen, hinzu.

Transformationsregel 4.10 *Sei* bp *des Typs* BehavioralPattern *das zu einem DFA D gehörende Verhaltensmuster. Jeder nicht-akzeptierende Zustand des*

DFA D mit Ausnahme des Startzustands wird mit Transitionen zum verwerfenden Zustand $r \in Q$ verbunden, die in dem jeweiligen Zustand unerlaubte Methodenaufrufe akzeptieren:

$\forall s \in Q/(F \cup \{q_0, r\}) : \forall rm \in bp.receivedMessages/\mathcal{RM}(s) :$
$\delta(s, * \rightarrow (rm.object : rm.type).rm.method()) = r,$
$* \rightarrow (rm.object : rm.type).rm.method() \in \Sigma$

Dem Zustand 3 des *State*-DFA werden also die folgenden beiden Transitionen hinzugefügt:

$\delta(3, * \rightarrow (a : abstractState).handle()) = r$ mit
$* \rightarrow (a : abstractState).handle() \in \Sigma$

und

$\delta(3, * \rightarrow (b : abstractState).handle()) = r$ mit
$* \rightarrow (b : abstractState).handle() \in \Sigma$

Zuletzt muss noch eine Transformationsregel definiert werden, die den DFA um Transitionen ergänzt, die in einem Zustand Methodenaufrufe unerlaubter Aufrufer akzeptieren und den DFA damit in den verwerfenden Zustand führen. Dazu werden weitere Definitionen benötigt. Die nächste Definition ermöglicht den Zugriff auf die Elemente eines Tupels der Menge $\mathcal{RM}_D(s)$.

Definition 4.10 *Sei rm ein Tupel der Menge $\mathcal{RM}_D(s)$. Dann bezeichne rm_{callee} den Objektnamen und $rm_{calleeType}$ den Typnamen des aufgerufenen Verhaltensmusterobjekts des Tupels rm. rm_{method} bezeichne den Methodennamen des Tupels rm.*

Nun muss noch die Menge $\mathcal{C}_D(s, rm)$ definiert werden, die aus allen Verhaltensmusterobjekten besteht, die in einem Zustand s des DFA D die Methode rm_{method} auf dem Verhaltensmusterobjekt rm_{callee}:$rm_{calleeType}$ aufrufen dürfen, mit $rm \in \mathcal{RM}_D(s)$.

Definition 4.11 *Sei s ein Zustand des DFA D und rm ein Element der Menge $\mathcal{RM}_D(s)$. Dann ist die Menge $\mathcal{C}_D(s, rm)$ definiert durch:*

$\mathcal{C}_D(s, rm) = \{(\sigma_{caller} : \sigma_{callerType}) | \delta(s, \sigma) \neq r \wedge$
$\sigma_{callee} = rm_{callee} \wedge \sigma_{calleeType} = rm_{calleeType} \wedge \sigma_{method} = rm_{method}\}$

Für den Zustand 3 des *State*-DFA und $rm = (c, context, setState)$ gilt:

$\mathcal{C}_{State}(3, (c, context, setState)) = \{(a : abstractState), (c : context)\}$

Die folgende Transformationsregel fügt nun dem DFA Symbole der Form $* \notin \mathcal{C} \rightarrow (b : B).m()$ sowie Transitionen, die diese Symbole akzeptieren, hinzu.

Transformationsregel 4.11 *Sei* bp *des Typs* BehavioralPattern *das zu einem DFA D gehörende Verhaltensmuster. Jeder nicht-akzeptierende Zustand des DFA D mit Ausnahme des Startzustands wird mit Transitionen zum verwerfenden Zustand $r \in Q$ verbunden, die Methodenaufrufe von unerlaubten Aufrufern akzeptieren:*

$\forall s \in Q/F \cup \{q_0, r\} : \forall rm \in \mathcal{RM}_D(s) :$
$\delta(s, * \notin \mathcal{C}_D(s, rm) \rightarrow (rm_{callee} : rm_{calleeType}).rm_{method}()) = r,$
$* \notin \mathcal{C}_D(s, rm) \rightarrow (rm_{callee} : rm_{calleeType}).rm_{method}() \in \Sigma$

Mit dieser Transformationsregel werden dem Zustand 3 des *State*-DFA die folgenden beiden Transitionen hinzugefügt:

$\delta(3, * \notin \{(a : abstractState), (c : context)\} \rightarrow (c : context).setState()) = r$
mit
$* \notin \{(a : abstractState), (c : context)\} \rightarrow (c : context).setState() \in \Sigma$
und
$\delta(3, * \notin \{(client)\} \rightarrow (c : context).request()) = r$ mit
$* \notin \{(client)\} \rightarrow (c : context).request() \in \Sigma$

Ein DFA, der zunächst aus einen NFA eines Verhaltensmusters entstanden ist, und nun durch die Transformationsregeln 4.10 und 4.11 erweitert wurde, akzeptiert alle zum Verhaltensmuster konformen Traces und verwirft alle zum Verhaltensmuster nicht-konformen Traces. Dieser DFA kann in der Verhaltensanalyse zur Erkennung des Verhaltensmusters eingesetzt werden.

Die vorgenommene Erweiterung gleicht einer Vervollständigung des Automaten. Da das Verfahren aber nur für die nicht-akzeptierenden Zustände ohne dem Startzustand angewendet wurde, fehlen einige Transitionen für einen vollständigen Automaten.

4.5 Zusammenfassung

Dieses Kapitel hat die formale Spezifikation der Verhaltensmuster zum Inhalt. Wie bereits in Kapitel 3.2 erläutert, lässt sich das Verhalten von Entwurfsmuster durch Sequenzen von Methodenaufrufen spezifizieren. In der Softwaretechnik haben sich dafür Sequenzdiagramme nach UML 2.0 etabliert. Da sich UML-Sequenzdiagramme auch sehr gut in den bisherigen, ebenfalls UML-unterstützten Erkennungsprozess integrieren, wurden sie als Basis zur Spezifikation von Verhaltensmustern gewählt.

Verhaltensmuster sind allerdings in ihrer Syntax gegenüber allgemeinen UML-Sequenzdiagrammen eingeschränkt. Des Weiteren sind die Verhaltensmuster syntaktisch eng verzahnt mit den Strukturmustern. Aus diesem Grund

wurde für die Verhaltensmuster auf Basis der Sequenzdiagramme nach UML 2.0 eine eigene Syntax definiert. In diesem Zusammenhang wurde auch das Metamodell der Strukturmuster erweitert, um die Verbindung zwischen Struktur- und Verhaltensmustern zu schaffen.

Die Semantik der Verhaltensmuster wurde in diesem Kapitel zunächst nur informell anhand eines Beispiels beschrieben. Dazu wurden einige hypothetische Traces des Beispiels vorgestellt, zu denen erläutert wurde, ob sie konform oder nicht-konform zu einem gegebenen Verhaltensmuster sind.

Formal ist die Semantik der Verhaltensmuster durch eine allgemeine Transformation von Verhaltensmustern in deterministische, endliche Automaten definiert worden. Dazu wurden Transformationsregeln für jedes Syntaxelement eines Verhaltensmusters in Elemente eines nichtdeterministischen, endlichen Automaten angegeben. Nach der Transformation erfolgt eine Umwandlung des nichtdeterministischen in einen deterministischen, endlichen Automaten, der jedoch nur einen Teil der durch das Verhaltensmuster beschriebenen Traces erkennt. Daher wurden weitere Transformationsregeln angegeben, die den deterministischen Automaten so erweitern, dass er alle zu einem Verhaltensmuster konformen Traces akzeptiert, und alle nicht-konformen verwirft. Dieser Automat wird nun im folgenden Kapitel zur Erkennung von Verhaltensmustern verwendet.

Kapitel 5

Verhaltensanalyse

In diesem Kapitel wird der Prozess der Verhaltensanalyse erläutert. Der erste Teil des Kapitels gibt einen Überblick über den Prozess. In den folgenden Abschnitten werden die Teilschritte der Verhaltensanalyse vorgestellt. Zunächst werden zwei verschiedene Verfahren zur Gewinnung der Traces diskutiert. Anschließend wird im Detail die Erkennung von Verhaltensmustern in den Traces mit Hilfe der im letzten Kapitel eingeführten deterministischen Automaten erklärt. Den Abschluss des Kapitels bildet die Bewertung der Ergebnisse der Verhaltensanalyse.

5.1 Verhaltensbasierter Erkennungsprozess

Der in Abbildung 5.1 dargestellte verhaltensbasierte Erkennungsprozess ist ein detaillierterer Ausschnitt aus dem Gesamtprozess der struktur- und verhaltensbasierten Entwurfsmustererkennung aus Abbildung 3.7 auf Seite 48. Der Prozess der verhaltensbasierten Entwurfsmustererkennung wurde bereits in [WO06] veröffentlicht.

Eingabe der dynamischen Analyse sind unter anderem die Kandidaten für Entwurfsmusterimplementierungen, die als Ergebnis der Strukturanalyse gewonnen wurden. Des Weiteren wird der Katalog der Verhaltensmuster benötigt. Dabei hängen die zu verwendenden Verhaltensmuster von den Strukturmustern ab, die zuvor in der Strukturanalyse verwendet wurden. Im Gegensatz zur Strukturanalyse wird bei der Verhaltensanalyse nicht der Quelltext des zu untersuchenden Softwaresystems benötigt, sondern der übersetzte Programmcode, damit das Programm ausgeführt werden kann.

Bei der Ausführung gibt es grundsätzlich zwei verschiedene Vorgehensweisen, um das Programm zu überwachen und Traces zu beobachten. Zum einen kann der Programmcode in unveränderter Form ausgeführt und die

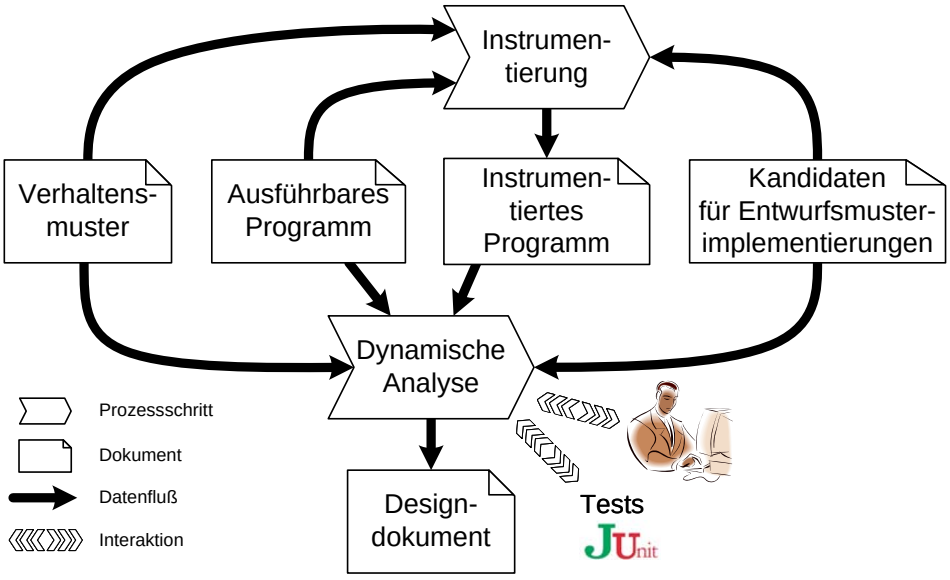

Abbildung 5.1: Der verhaltensbasierte Erkennungsprozess

Ausführung des Programms von außen überwacht werden, zum Beispiel durch einen Debugger. Zum anderen kann aber auch der Programmcode manipuliert werden. In diesem Fall wird zusätzlicher Code in den vorhandenen Programmcode eingefügt, um Methodenaufrufe aus dem zu untersuchenden Programm heraus zu protokollieren. Das Einfügen zusätzlichen Codes wird *Instrumentierung* genannt. Die zu beobachtenden Methoden werden den Verhaltensmustern in Kombination mit den Kandidaten entnommen.

In der dynamischen Analyse werden diese Eingaben verwendet, um das Programm auszuführen und in den beobachteten Traces nach Verhaltensmustern zu suchen. Die Suche nach Verhaltensmustern kann ebenfalls auf zwei verschiedene Weisen durchgeführt werden. Entweder werden die beobachteten Methodenaufrufe parallel zur Programmausführung analysiert, oder die Traces werden zunächst gespeichert und erst im Anschluss an die Programmausführung analysiert. Die erste Methode, die so genannte *Online-Analyse*, hat den Vorteil, dass nur eine geringe Datenmenge gespeichert werden muss. Die Informationen über die beobachteten Methodenaufrufe werden direkt verarbeitet und danach verworfen. Übrig bleiben nur die Ergebnisse der dynamischen Analyse. Jedoch wird unter Umständen zur Verhaltensanalyse Rechenzeit verbraucht, die dem zu untersuchenden Programm nicht zur Verfügung steht. Die zweite Methode, die so genannte *Offline-Analyse*, benötigt nur sehr wenig Rechenzeit, um

die beobachteten Methodenaufrufe in einem Protokoll zu speichern. Allerdings werden die Protokolle sehr groß, so dass sie viel Speicherplatz verbrauchen und unter Umständen schwer zu handhaben sind.

Das zentrale Problem dynamischer Analysen ist die angemessene Auswahl der Eingabedaten, die zur Ausführung des zu analysierenden Softwaresystems benötigt werden. Um verwertbare Ergebnisse zu erhalten, sollten die Eingabedaten möglichst repräsentativ für die in der Praxis auftretenden Daten ausgewählt werden. Die Ausführung der zu untersuchenden Software kann entweder durch automatische Tests erfolgen oder durch manuelle Bedienung durch einen Benutzer. Im Idealfall wird das Softwaresystem in einer produktiven Umgebung eingesetzt, bei der reale Daten als Eingabe verwendet werden.

Das Ergebnis der dynamischen Analyse stellt das Gesamtergebnis der struktur- und verhaltensbasierten Entwurfsmustererkennung dar. Der Reverse-Engineer erhält ein Design-Dokument in Form von UML-Klassendiagrammen, in denen die Kandidaten annotiert sind. Die Annotationen enthalten Bewertungen, die den Grad der Übereinstimmung der Entwurfsmusterimplementierung mit dem Struktur- und den Verhaltensmustern des Entwurfsmusters angeben.

5.2 Gewinnung der Traces

Wie bereits erwähnt, gibt es zwei grundsätzliche Vorgehensweisen, um Methodenaufrufe in einem zu untersuchenden Softwaresystem zu überwachen. Die erste besteht darin, das unveränderte Programm von außen zu beobachten, die zweite darin, den Programmcode so zu verändern, dass die Methodenaufrufe aus dem Programmcode heraus überwacht werden. In der vorliegenden Arbeit wurde für beide Vorgehensweisen je eine Lösung umgesetzt. Dabei wurde in beiden Lösungen sowohl eine Online-, als auch eine Offline-Analyse ermöglicht. Im Folgenden werden diese beiden Lösungen vorgestellt und ihre Vor- und Nachteile einander gegenübergestellt. Zunächst werden aber einige Voraussetzungen für die Gewinnung der Traces erläutert.

5.2.1 Voraussetzungen

In Abschnitt 3.2.3 wird beschrieben, wie der Trace eines zu untersuchenden Softwaresystems als Tracegraph repräsentiert wird. Die Informationen, die der Tracegraph zu einem einzelnen Methodenaufruf enthält, sind die aufrufende Instanz und ihr Typ, die aufgerufene Instanz und ihr Typ, sowie der Name der Methode und die Argumente. Diese Informationen müssen bei der Überwa-

chung des Softwaresystems gesammelt und in einem Tracegraphen aufbereitet werden.

Bei einer Online-Analyse werden die Methodenaufrufe in Form einer Instanz der Klasse Behavior::MethodCall aus dem Modell des Tracegraphen direkt an die Verhaltensanalyse weitergereicht. Bei einer Offline-Analyse wird der Tracegraph dagegen in einer Datei gespeichert. Der Tracegraph kann dann später aus der Datei rekonstruiert und die Methodenaufrufe an die Verhaltensanalyse weitergereicht werden.

In Entwurfsmustern wird sehr häufig von der polymorphen Methodenbindung Gebrauch gemacht. Sie wird daher auch in den Verhaltensmustern verwendet. Für die Verhaltensanalyse müssen also alle Methoden überwacht werden, die polymorph für die in den Nachrichten der Verhaltensmuster verwendeten Methoden gebunden werden können. Die zu beobachtenden Methoden sind also nicht nur die Methoden, die in den Nachrichten eines Verhaltensmusters verwendet werden, sondern auch alle Methoden, die die Methoden der Nachrichten überschreiben oder implementieren.

5.2.2 Überwachung durch Debugging

In der vorliegenden Arbeit wurde ein Verfahren entwickelt, bei dem der Programmcode des zu untersuchenden Softwaresystems unverändert mit Hilfe eines Debugger ausgeführt wird und Aufrufe zu beobachtender Methoden überwacht werden [MW05]. Dazu setzt der Debugger am Anfang des Rumpfes einer zu beobachtenden Methode einen Breakpoint. Das bedeutet, es wird eine Stelle im Programmcode der Methode markiert, an dem die Ausführung des Programms unterbrochen wird. Der Debugger greift bei einer Unterbrechung auf den Methodenaufrufstack zu und ermittelt daraus alle für die Verhaltensanalyse relevanten Informationen dieses Methodenaufrufs. Anschließend wird die Ausführung des Programms fortgesetzt.

Das Verfahren hat den Vorteil, dass es auf einem universellem Konzept aufbaut. Debugger mit der Fähigkeit, Breakpoints zu setzen, existieren für nahezu alle Programmiersprachen. Das Verfahren ist damit auf die meisten Programmiersprachen sehr leicht übertragbar. Der Nachteil dieses Verfahrens besteht jedoch darin, dass ein Debugger die Ausführung des zu untersuchenden Programms zum Teil erheblich verlangsamt. Der Performanzverlust hängt dabei nur von der Zahl der zu überwachenden Methoden ab. Je nach Programmiersprache und eingesetztem Debugger wird ein nicht unerheblicher Performanzverlust allein durch die notwendige Ausführung des Programms im Debugging-Modus verursacht. Der Einsatz der dynamischen Analyse in einer produktiven

Umgebung ist damit kaum möglich.

Die in der vorliegenden Arbeit umgesetzte Lösung dieses Verfahrens wurde für die Überwachung Java-basierter Programme entwickelt [WME04, MW05, MW05]. Die Ausführung der Programme wird durch das Debugging um den Faktor 2,5 bis 8 je nach Zahl der zu beobachtenden Methoden verlangsamt. In einer anderen Arbeit, in der ebenfalls über einen Debugger Methodenaufrufe überwacht werden, wird sogar von einer Verlangsamung um einen Faktor von bis zu 300.000 berichtet [Meh01, Meh03]. Es wurde eine künstliche Messung durchgeführt, bei der eine Methode etwa 10.000 Mal aufgerufen wurde. Zum Debuggen wurde jedoch eine spezielle Schnittstelle der Java-Virtual-Machine verwendet.

5.2.3 Überwachung durch Instrumentierung

Durch Instrumentierung wird der Programmcode eines Softwaresystems manipuliert. Häufig wird Instrumentierung zur Analyse von dynamischen Eigenschaften der Software verwendet [SO05]. Die Techniken zur Instrumentierung werden grundsätzlich in zwei Kategorien eingeteilt, in statische und dynamische Instrumentierung. Bei der statischen Instrumentierung wird der Programmcode vor der Ausführung des Programms verändert. Hier gibt es wiederum zwei verschiedene Möglichkeiten, die Instrumentierung durchzuführen. Zum einen wird der zusätzliche Programmcode dem ursprünglichen Quelltext hinzugefügt und der Quelltext neu übersetzt, um anschließend das Programm auszuführen. Zum anderen wird der bereits übersetzte Programmcode vor der Ausführung verändert. Bei der dynamischen Instrumentierung wird der Programmcode dagegen erst beim Laden in den Speicher während der Ausführung des zu untersuchenden Softwaresystems verändert.

Zur Überwachung von Methodenaufrufen für die Verhaltensanalyse wird dem ursprünglichen Programmcode zusätzlicher Programmcode hinzugefügt. Der eingefügte Programmcode ermittelt beim Aufruf einer zu beobachtenden Methode alle notwendigen Informationen und protokolliert diese oder gibt sie direkt an die Verhaltensanalyse weiter. Prinzipiell kann der zusätzliche Programmcode wie beim Debugging jeweils am Anfang des Methodenrumpfes eingefügt werden. Er wird dadurch immer dann ausgeführt, wenn die zu beobachtende Methode aufgerufen wird.

Im Gegensatz zum Debugging führt die Instrumentierung nur zu einem geringen Performanzverlust, der nur von der Zahl der überwachten Methoden abhängt. Die Instrumentierung ist damit zum Einsatz in produktiven Umgebungen durchaus geeignet. Ein Nachteil dieser Methode ist aber der höhere

Aufwand für den Reverse-Engineer gegenüber dem Debugging. Bei der statischen Instrumentierung muss der Programmcode vor der Ausführung des Programms erst manipuliert und unter Umständen sogar der Quelltext neu übersetzt werden.

Wird der bereits übersetzte Programmcode verändert, so ist die Umsetzung der Instrumentierung sehr stark abhängig von der Programmiersprache, in der das zu untersuchende Softwaresystem geschrieben wurde. Die Quelltexte einiger Programmiersprachen wie C++ oder PASCAL werden direkt in Maschinensprache übersetzt. Dieser Programmcode ist nur mit sehr hohem Aufwand zu manipulieren, da zum Beispiel alle Sprungadressen im Programmcode neu berechnet werden müssen. Bei Programmiersprachen wie JAVA, SMALLTALK oder C#, bei denen der Quelltext in eine Zwischensprache übersetzt wird, ist die Instrumentierung dagegen einfacher. Die Zwischensprache ist ebenfalls eine Hochsprache, die zur Ausführung meist interpretiert wird. Daher müssen zum Beispiel Sprungadressen nur lokal innerhalb eines Methodenrumpfes neu berechnet werden. Damit ist die Technik der Instrumentierung nicht so leicht übertragbar auf andere Programmiersprachen wie das Debugging.

In der vorliegenden Arbeit wurde ein Werkzeug entwickelt, mit dem JAVA-Bytecode instrumentiert werden kann. JAVA-Bytecode ist in eine Zwischensprache übersetzter JAVA-Quelltext. Allerdings kann das Prinzip, am Anfang eines Methodenrumpfes Programmcode einzufügen, der die Aufrufe der Methode überwacht, nicht auf JAVA-Programmcode übertragen werden. In JAVA-Programmcode ist es nicht möglich, innerhalb eines Methodenrumpfes auf die Instanz zuzugreifen, die die Methode aufgerufen hat. Der Methodenaufrufstack, in dem diese Information enthalten ist, ist nicht abrufbar. Die für die Verhaltensanalyse benötigten Informationen können also auf diese Weise nicht gewonnen werden.

Stattdessen müssen die Stellen im JAVA-Programmcode ergänzt werden, an denen die zu beobachtenden Methoden aufgerufen werden. Das bedeutet aber, dass der gesamte Programmcode auf solche Methodenaufrufe untersucht und gegebenenfalls instrumentiert werden muss. Dieses Vorgehen birgt weitere Nachteile. Werden Teile des Softwaresystems nicht instrumentiert, können Aufrufe von zu beobachtenden Methoden aus diesen Teilen des Softwaresystems nicht überwacht werden. In JAVA ist es zudem möglich, durch Introspektion (JAVA-Reflection API) Methoden aufzurufen, ohne diesen Aufruf explizit im Quelltext ausdrücken zu müssen. Bei solchen Methodenaufrufen ist es daher prinzipiell nicht möglich, durch Instrumentierung die für die Verhaltensanalyse notwendigen Informationen zu gewinnen.

5.3 Verhaltenserkennung

Im Folgenden wird der Prozess der Verhaltenserkennung im Detail erläutert. In Abbildung 5.2 ist das Modell der Verhaltenserkennung dargestellt. Die Klasse BehavioralAnalysis ist der Einstiegspunkt für die Verhaltensanalyse. Als Eingabe erhält die Verhaltensanalyse die Kandidaten für Entwurfsmusterimplementierungen aus der Strukturanalyse. Diese werden durch die Klasse Annotations::Annotation (Abbildung 2.7, Seite 24) repräsentiert. Die Klasse BehavioralAnalysis bildet durch die Referenz annotations einen Schlüssel auf eine Menge von Annotationen ab. Als Schlüssel dient der Typname der Annotationen. Der Schlüssel „State“ bildet zum Beispiel auf alle Annotationen des Typs Annotations::State ab, also alle Annotationen des *State*-Strukturmusters.

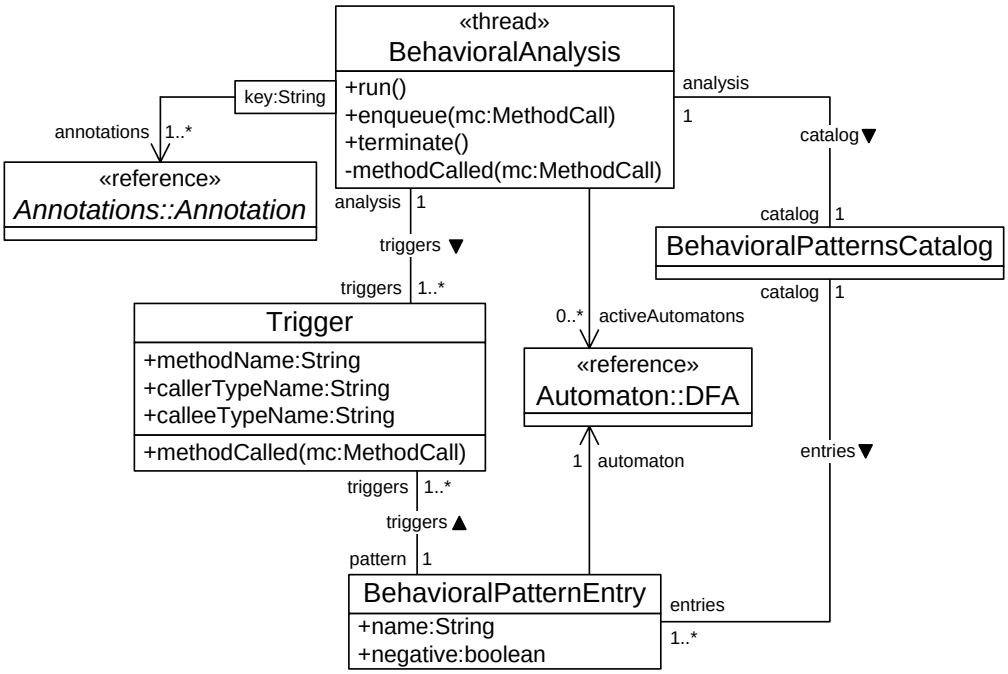

Abbildung 5.2: Modell der Verhaltenserkennung, Paket BehaviorAnalysis

Des Weiteren gehört zur Eingabe ein Katalog von Verhaltensmustern, repräsentiert durch die Klasse BehavioralPatternsCatalog. Ein Katalog enthält beliebig viele Einträge für Verhaltensmuster (BehavioralPatternEntry), von denen jeder wiederum einen deterministischen, endlichen Automaten (Automaton::DFA) zur Erkennung des Verhaltensmusters referenziert.

Die Verhaltenserkennung wird als eigenständiger Thread gestartet. Zur Laufzeit der Analyse werden die beobachteten Methodenaufrufe als Instanz der Klasse Behavior::MethodCall (Abbildung 3.9, Seite 52) des Tracegraphen einer FIFO-Queue (*First In, First Out*) mit der Methode BehavioralAnalysis::enqueue(MethodCall) hinzugefügt. Bei einer Offline-Analyse werden die Methodenaufrufe einer Datei entnommen. Bei einer Online-Analyse werden die Methodenaufrufe durch den Debugger oder den instrumentierten Programmcode an die Verhaltensanalyse weitergereicht. Die Verhaltensanalyse wird beendet, wenn alle Methodenaufrufe aus der Datei gelesen wurden respektive wenn das zu untersuchende Softwaresystem terminiert.

Die Erkennung eines Verhaltensmusters durch einen Automaten wird durch so genannte *Trigger* ausgelöst. Trigger repräsentieren besondere Methodenaufrufe. Jedes Verhaltensmuster hat mindestens einen, aber beliebig viele Trigger. Die Verhaltenserkennung referenziert alle Trigger der Verhaltensmuster eines Kataloges. Die aktiven Automaten, die durch die Trigger ausgelöst wurden, referenziert die Verhaltenserkennung durch die Assoziation activeAutomatons.

5.3.1 Erweiterter Automat

In Abbildung 5.3 ist das Modell der deterministischen, endlichen Automaten dargestellt, die nach dem Algorithmus in Abschnitt 4.4 erzeugt und durch die Verhaltenserkennung verwendet werden.

Der Automat, repräsentiert durch die Klasse DFA, enthält eine Menge von Zuständen. Zwei ausgezeichnete Zustände dieser Menge sind der Startzustand und der verwerfende Zustand des Automaten. Die Zustände sind entweder nicht-akzeptierend (NON_ACCEPTING), akzeptierend (ACCEPTING) oder verwerfend (REJECTING). Die Zustandsübergänge werden durch Transitionen (Transition) modelliert, von denen wiederum jede ein Symbol vom Typ der abstrakten Klasse AbstractSymbol referenziert.

Symbole

Jedes Symbol hat ein Attribut für den Methodennamen und referenziert ein Objekt (MethodCallObject), auf dem die Methode aufgerufen wird, durch die Assoziation callee. Es gibt drei Klassen für konkrete Symbole, PermittedMethodCall, ProhibitedMethodCall und ProhibitedCaller.

Die Klasse PermittedMethodCall repräsentiert Symbole der Formen $(a : A) \rightarrow (b : B).m()$ und $a \rightarrow (b : B).m()$. Dazu referenziert sie zusätzlich ein Objekt, das den Methodenaufruf ausführt, durch die Assoziation caller. Dabei gilt für

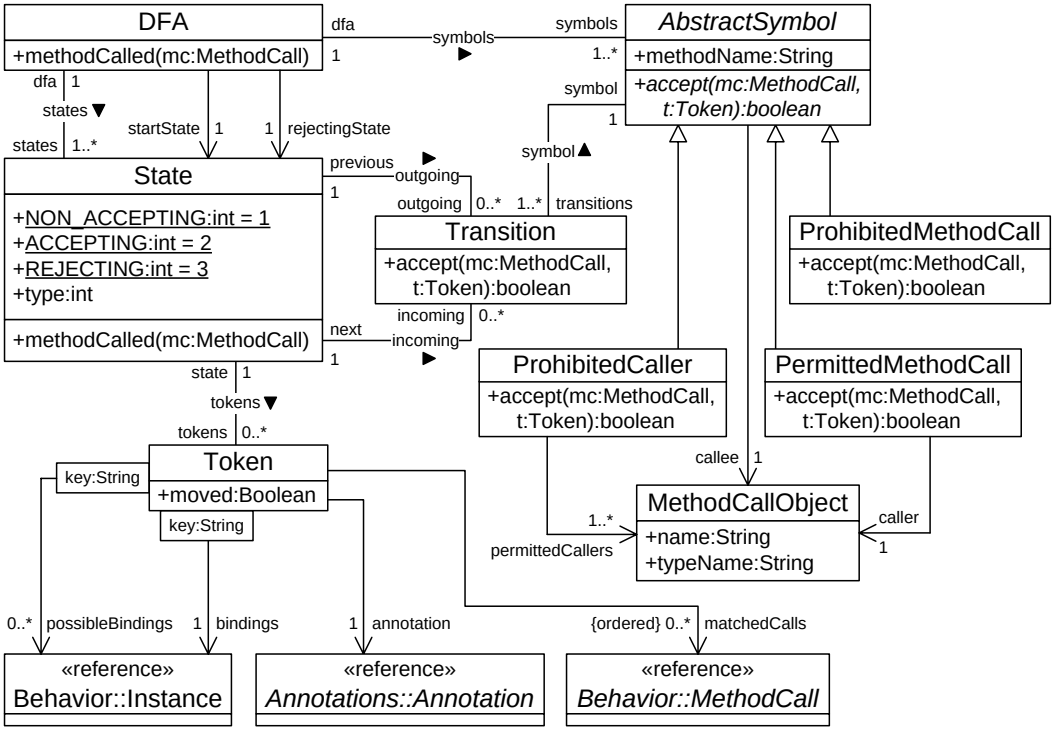

Abbildung 5.3: Modell des deterministischen Automaten, Paket Automaton

eine Instanz symbol:PermittedMethodCall und ein Symbol $\sigma \in \Sigma$ der genannten Formen:

symbol.methodName=σ_{method},
symbol.caller.name=σ_{caller},
symbol.caller.typeName=$\sigma_{callerType}$,
symbol.callee.name=σ_{callee} und
symbol.callee.typeName=$\sigma_{calleeType}$.

Die Klasse ProhibitedMethodCall repräsentiert Symbole der Form $* \rightarrow (b : B).m()$. Dabei gilt für eine Instanz symbol:ProhibitedMethodCall und ein Symbol $\sigma \in \Sigma$ der genannten Form:

symbol.methodName=σ_{method},
symbol.callee.name=σ_{callee} und
symbol.callee.typeName=$\sigma_{calleeType}$.

Die Klasse ProhibitedCaller repräsentiert Symbole der Form $* \notin \mathcal{C} \rightarrow (b : B).m()$. Dazu referenziert sie eine Menge von Objekten, denen es ausschließlich erlaubt ist, die Methode aufzurufen. Diese Menge repräsentiert $\mathcal{C}$. Dabei gilt

für eine Instanz symbol:ProhibitedCaller und ein Symbol $\sigma \in \Sigma$ der genannten Form:

symbol.methodName=σ_{method},
symbol.permittedCallers→**size**()=$|\mathcal{C}|$ **and**
$\forall c \in \mathcal{C}$: symbol.permittedCallers→**exists**(mco:MethodCallObject|
mco.name=c_{caller} **and** mco.typeName=$c_{callerType}$),
symbol.callee.name=σ_{callee} und
symbol.callee.typeName=$\sigma_{calleeType}$.

Die Symbole und die Transitionsfunktion eines Automaten sind so konstruiert, dass es in jedem Zustand maximal eine ausgehende Transition gibt, zu dessen Symbol ein beobachteter Methodenaufruf konform ist.

Tokens

Das hier verwendete Modell eines Automaten entspricht nicht der allgemeinen Definition von deterministischen, endlichen Automaten. Es wurde um ein bekanntes Konzept der Petri-Netze erweitert, den so genannten *Tokens*.

Wie bereits in Abschnitt 4.3.1 erläutert, ist es wahrscheinlich, dass während der Verhaltensanalyse sehr viele Traces auf Konformität zu einem Verhaltensmuster untersucht werden müssen. Nach der klassischen Theorie müsste für jeden dieser Traces ein eigener Automat verwendet werden. Die Variablenbindungen des Automaten werden mit der Annotation eines Kandidaten initialisiert. Bei der Verhaltenserkennung ändert der Automat seinen Zustand durch Konsumieren der Methodenaufrufe und endet in einem nicht-akzeptierenden, akzeptierenden oder dem verwerfenden Zustand. Ein klassischer Automat wäre daher nur einmal „verwendbar“.

Um bei der Verhaltensanalyse die Automaten wiederverwenden zu können, werden im Modell Token durch die Klasse Token eingeführt. Ein Token repräsentiert den aktuellen Zustand eines Automaten, indem es einen der möglichen Zustände des Automaten referenziert. Die Variablenbindungen des Automaten werden durch die Referenzen annotation und bindings repräsentiert. Die Annotation enthält die Bindung der Typ- und Methodennamen an die Elemente des untersuchten Kandidaten. Die qualifizierte Referenz bindings bildet dagegen einen Schlüssel auf eine Instanz aus dem Verhaltensmodell ab. Als Schlüssel wird der Name eines Verhaltensmusterobjekts benutzt. So wird die Bindung der Verhaltensmusterobjekte an die Instanzen, die die Methoden aufrufen beziehungsweise auf denen die Methoden aufgerufen werden, repräsentiert. Des Weiteren enthält das Token eine geordnete Liste von Methodenaufrufen des Traces, die konform zum Verhaltensmuster sind.

Jeder Zustand eines Automaten vom Typ State kann beliebig viele Tokens referenzieren, so dass mit einem Automaten beliebig viele Traces gleichzeitig auf Konformität zum Verhaltensmuster untersucht werden können. Für jeden Trace, der untersucht werden soll, wird ein Token in den Startzustand des Automaten gelegt und mit der Annotation des Kandidaten initialisiert. Dies wird durch die Trigger eines Verhaltensmusters durchgeführt.

5.3.2 Trigger

Die Trigger eines Verhaltensmusters repräsentieren solche Methodenaufrufe, die am Anfang eines zum Verhaltensmuster konformen Traces stehen dürfen. Trigger sind also alle Nachrichten eines Verhaltensmusters, die chronologisch als erstes aufgerufen werden dürfen. Im *State*-Verhaltensmuster sind dies die Nachrichten client→(c:context).setState() und client→(c:context).request(). Da client→(c:context).setState() eine optionale Nachricht ist, darf auch client→(c:context).request() als erstes aufgerufen werden. Im *Strategy*-Verhaltensmuster gibt es dagegen nur einen Trigger, nämlich client→(c:context).setStrategy(). Formal definiert werden die Trigger über den deterministischen Automaten des Verhaltensmusters.

Definition 5.1 *Die Menge $\mathcal{T}$ der Trigger eines Verhaltensmusters ist definiert über den zugehörigen DFA D:*

$$\begin{aligned}\mathcal{T} = \{&trigger : Trigger | \delta(q_0, \sigma) \in Q/\{r\} \wedge \\ &trigger.method = \sigma_{method} \wedge \\ &trigger.callerTypeName = \sigma_{callerType} \wedge \\ &trigger.calleeTypeName = \sigma_{calleeType}\},\end{aligned}$$

wobei q_0 der Startzustand und r der verwerfende Zustand des DFA D ist.

Die Verwendung eines Triggers stellt eine Optimierung der Verhaltenserkennung dar. Es müssen nicht alle möglichen Traces auf Konformität überprüft werden, sondern nur solche, die mit einem Methodenaufruf beginnen, der zu einem Trigger konform ist. Es stellt sich jedoch die Frage, ob dadurch konforme oder nicht-konforme Traces von der Verhaltenserkennung „übersehen“ werden?

Zu der Menge der Trigger gehört mindestens eine Nachricht, die nicht optional ist. In einem konformen Trace muss also ein zu der Nachricht konformer Methodenaufruf vorhanden sein. Wird also während der Ausführung des zu untersuchenden Softwaresystems kein entsprechender Methodenaufruf beobachtet und damit die Erkennung des Verhaltensmusters getriggert, so kann

auch kein zum Verhaltensmuster konformer Trace gefunden werden. Es können also durch Verwendung der Trigger keine konformen Traces von der Verhaltenserkennung „übersehen“ werden.

Es bleibt die Frage, ob nicht-konforme Traces, die negative Indizien für die Existenz einer Entwurfsmusterimplementierung liefern, „übersehen“ werden? Theoretisch stellen alle Traces, die zwar Methodenaufrufe der zum Verhaltensmuster gehörenden Nachrichten enthalten, aber nicht als konforme Traces erkannt werden, solche negativen Indizien dar. Ein Verhaltensmuster deckt nicht das gesamte mögliche Verhalten eines Entwurfsmusters ab, sondern spezifiziert nur eine Schlüsselsequenz, anhand derer das Entwurfsmuster erkannt werden kann. Somit sind alle anderen Traces, die nicht aus dieser Schlüsselsequenz bestehen, nicht konform. Im Gegensatz zu konformen Traces kommen solche Traces aber weit häufiger vor. Startet ein nicht-konformer Traces mit dem Aufruf einer zum Trigger konformen Methode, so wird der Trace als nicht-konform erkannt. Alle anderen nicht-konformen Traces werden „übersehen“. Allerdings wäre das Wissen um solche Traces wegen der weit höheren Wahrscheinlichkeit kein echter Mehrwert für den Reverse-Engineer. Die erkannten nicht-konformen Traces sind dagegen eher interessant, da sie eben gegen die Schlüsselsequenz verstoßen. Trigger sind also ein legitimer Weg zur Optimierung der Verhaltenserkennung.

Im Folgenden wird erläutert, wie ein Trigger anhand eines gegebenen Methodenaufrufs entscheidet, ob die Verhaltenserkennung eines Verhaltensmusters durch die Erzeugung eines Tokens gestartet werden muss. Der Algorithmus ist in Pseudocode-Syntax in Abbildung 5.4 zu sehen.

Ein Trigger wird durch eine Annotation auf eine konkrete Methode, einen konkreten, aufrufenden Typ und einen konkreten, aufgerufenen Typ des Kandidaten abgebildet. Für jede Annotation (Zeile 3) muss nun der Trigger diese Elemente des Kandidaten mit dem beobachteten Methodenaufruf vergleichen (Zeile 4). Sowohl die Typen der am Methodenaufruf beteiligten Instanzen, als auch die aufgerufene Methode müssen zu den konkreten Typen respektive der konkreten Methode konform sein. Treffen diese Bedingungen zu, gehört der Methodenaufruf zum Kandidaten und die nachfolgenden Methodenaufrufe müssen auf ihre Konformität zum Verhaltensmuster des Kandidaten untersucht werden.

Dazu prüft der Trigger zunächst, ob bereits der Automat des Verhaltensmusters zu der Menge der aktiven Automaten der Verhaltensanalyse hinzugefügt wurde (Zeile 5). Ist dies nicht der Fall, fügt der Trigger den Automaten dieser Menge hinzu (Zeile 6). Anschließend wird ein Token erzeugt (Zeile 7), unter anderem mit der Annotation initialisiert (Zeilen 8 und 9) und dem Startzustand

```
 1: Trigger::methodCalled(mc:MethodCall)
 2:   let token:Token
 3:   forEach annotation:Annotation in
          self.analysis.annotations[self.pattern.name] do
 4:     if (self.callerTypeName<>OCLVoid implies
               mc.caller.type.conformsTo(
                 annotation.nodes[self.callerTypeName]))
            and mc.callee.type.conformsTo(
              annotation.nodes[self.calleeTypeName])
            and mc.type.conformsTo(
              annotation.nodes[self.methodName]) then
 5:       if not self.analysis.activeAutomatons→
              includes(self.pattern.automaton) then
 6:         self.analysis.activeAutomatons→
              add(self.pattern.automaton)
 7:       token = new Token
 8:       token.annotation = annotation
 9:       token.moved = false
10:       self.pattern.automaton.startState.tokens→add(token)
```

Abbildung 5.4: Die Methode methodCalled der Klasse Trigger

des Automaten hinzugefügt (Zeile 10).

Ein Trigger enthält keine Verhaltensmusterobjekte, die mit den Instanzen eines Methodenaufrufs verglichen werden müssten. In Abschnitt 4.3.2 wurden die Verhaltensmusterobjekte zu Beginn der Verhaltensmustererkennung nicht-deterministisch an Instanzen aus der Laufzeitumbegung des zu untersuchenden Programms gebunden. Dies ist in der Verhaltensanalyse natürlich nicht möglich. Die Verhaltensmusterobjekte werden erst während der Verhaltenserkennung gebunden. Da zum Zeitpunkt, zu dem der Trigger mit dem Methodenaufruf verglichen wird, die Bindung der Verhaltensmusterobjekte noch nicht feststeht, werden die Verhaltensmusterobjekte ignoriert.

5.3.3 Verarbeitung der beobachteten Methodenaufrufe

Die Methodenaufrufe werden aus einer Datei gelesen oder vom Debugger beziehungsweise dem instrumentierten Programm beobachtet und der Verhaltenserkennung asynchron durch eine FIFO-Queue übergeben. Solange die Queue noch

nicht leer ist und das beobachtete Programm noch nicht terminiert beziehungsweise die Datei nicht vollständig gelesen wurde, entnimmt die Verhaltenserkennung jeweils einen Methodenaufruf der Queue und verarbeitet ihn.

```
1: BehavioralAnalysis::methodCalled(mc:MethodCall)
2:   forEach trigger:Trigger in self.triggers do
3:     trigger.methodCalled(mc)
4:   forEach dfa:DFA in self.activeAutomatons do
5:     dfa.methodCalled(mc)
```

Abbildung 5.5: Die Methode methodCalled der Klasse BehavioralAnalysis

In Abbildung 5.5 ist der Algorithmus zur Verarbeitung eines Methodenaufrufs innerhalb der Klasse BehavioralAnalysis zu sehen. Sie beschränkt sich darauf, den Methodenaufruf zunächst an alle Trigger (Zeilen 2 und 3) und dann an alle aktiven Automaten (Zeilen 4 und 5) weiter zu reichen. Die Verarbeitung der Methodenaufrufe durch die Trigger wurde im vorigen Abschnitt behandelt. Sie fügen gegebenenfalls neue Automaten der Menge der aktiven Automaten hinzu und legen ein neues Token in den Startzustand der zuständigen Automaten.

```
1:  DFA::methodCalled(mc:MethodCall)
2:    forEach state:State in self.states do
3:      state.methodCalled(mc)
4:    self.checkBindings()
```

Abbildung 5.6: Die Methode methodCalled der Klasse DFA

Die weitere Verarbeitung eines Methodenaufrufs durch einen Automaten findet nach den drei Algorithmen in den Abbildungen 5.6, 5.7 und 5.8 statt. Ein Automat gibt einen Methodenaufruf lediglich an alle seine Zustände weiter (Abbildung 5.6). Der Methodenaufruf in Zeile 4 wird in Abschnitt 5.3.6 erläutert.

Ein Zustand iteriert über alle Tokens, die in dem Zustand liegen (Abbildung 5.7, Zeile 3). Zunächst prüft der Zustand, ob das Token bereits verschoben wurde (Zeile 4). Die Klasse Token besitzt ein boolsches Attribut moved, das angibt, ob das Token bei der Überprüfung des aktuellen Methodenaufrufs bereits von einem Zustand in einen Nachfolgezustand verschoben wurde. Es verhindert, dass ein Methodenaufruf mehrfach durch den klassischen Automaten, repräsentiert durch das Token, konsumiert wird.

```
 1: State::methodCalled(mc:MethodCall)
 2:   let accepted:Boolean
 3:   forEach token:Token in self.tokens do
 4:     if not token.moved then
 5:       accepted = false
 6:       forEach transition:Transition in self.outgoing do
 7:         if not accepted then
 8:           accepted = transition.accept(mc,token)
 9:   forEach token:Token in self.tokens do
10:     token.moved=false
```

Abbildung 5.7: Die Methode methodCalled der Klasse State

Wurde das Token noch nicht verschoben, das bedeutet, der Methodenaufruf wurde für dieses Token noch nicht verarbeitet, so iteriert der Zustand nun über alle Transitionen, die von ihm ausgehen, (Zeilen 5-8) und reicht den Methodenaufruf und das Token über die Methode Transition::accept(MethodCall, Token) an die Transition weiter. Diese Iteration endet, sobald eine Transition den Methodenaufruf akzeptiert, also den boolschen Wert true zurück gibt (Zeilen 7 und 8). Akzeptiert eine Transition den Methodenaufruf, wird das Token vom aktuellen Zustand in den Nachfolgezustand der Transition verschoben. Akzeptiert keine der Transitionen den Methodenaufruf, so bedeutet dies, dass der Automat, dessen Zustand das Token repräsentiert, den gegebenen Methodenaufruf ignoriert und im aktuellen Zustand verbleibt.

Nachdem der Zustand die Iteration über die Tokens abgeschlossen hat, iteriert er ein zweites Mal über die im Zustand verbliebenen Tokens und weist jeweils ihrem Attribut moved den boolschen Wert false zu. Dadurch werden die Tokens für den nächsten Methodenaufruf vorbereitet.

Die Methode accept(MethodCall, Token) der Klasse Transition (Abbildung 5.8) akzeptiert einen Methodenaufruf, wenn sein zugehöriges Symbol den Methodenaufruf akzeptiert (Zeile 2). In diesem Fall verschiebt die Transition das Token vom aktuellen Zustand (Zeile 3) in den nachfolgenden Zustand (Zeile 4) und markiert das Token als verschoben (Zeile 5). Anschließend fügt sie den Methodenaufruf der Liste der zum Verhaltensmuster konformen Methodenaufrufe hinzu (Zeile 6) und gibt den boolschen Wert true zurück (Zeile 7). Akzeptiert die Transition den Methodenaufruf nicht, gibt sie den boolschen Wert false zurück.

```
1:  Transition::accept(mc:MethodCall,token:Token):Boolean
2:    if self.symbol.accept(mc,token) then
3:      self.previous.tokens→remove(token)
4:      self.next.tokens→add(token)
5:      token.moved = true
6:      token.matchedCalls→add(mc)
7:      return true
8:    else
9:      return false
```

Abbildung 5.8: Die Methode accept der Klasse Transition

5.3.4 Konforme Methodenaufrufe und Variablenbindung

Im Folgenden wird die Verarbeitung des Methodenaufrufs durch ein Symbol erklärt. Dabei wird nicht nur überprüft, ob ein Methodenaufruf konform zu dem Symbol ist. Bei Konformität werden gegebenenfalls auch Verhaltensmusterobjekte an die am Methodenaufruf beteiligten Instanzen gebunden.

Die Bindung der Typ- und Methodennamen eines Symbols ist durch die Annotation eines Kandidaten gegeben. Die Bindung der Variablen an konkrete Elemente des Kandidaten wird in der Klasse Annotations::Annotation durch die qualifizierte Assoziation nodes (siehe Abbildung 2.7, Seite 24) hergestellt. Als Schlüssel für die qualifizierte Assoziation dient der Variablenname.

Die Verhaltensmusterobjekte werden dagegen durch die Assoziation bindings der Klasse Token an Instanzen der Laufzeitumgebung gebunden (Abbildung 5.3). Als Schlüssel für die qualifizierte Assoziation dient auch hier der Variablenname der Verhaltensmusterobjekte. Die Bindung steht allerdings zu Beginn der Verhaltensanalyse noch nicht fest.

Es existieren drei konkrete Klassen für Symbole, PermittedMethodCall, ProhibitedMethodCall und ProhibitedCaller. Alle drei Klassen implementieren jeweils die Methode accept(MethodCall,Token), die von ihrer abstrakten Oberklasse AbstractSymbol deklariert wird. Für jede der drei Implementierungen dieser Methode werden Nachbedingungen definiert. Die Nachbedingungen spezifizieren nicht nur, unter welchen Bedingungen das Symbol den Methodenaufruf akzeptiert, sondern auch, ob und gegebenenfalls wie Verhaltensmusterobjekte an Instanzen gebunden werden.

Symbole des Typs PermittedMethodCall

Die Klasse PermittedMethodCall repräsentiert Symbole der Formen $(a : A) \rightarrow (b : B).m()$ und $a \rightarrow (b : B).m()$. Für diese Symbole wurde bereits in Abschnitt 4.3.3 durch fünf Bedingungen informell erläutert, wann ein Methodenaufruf zu der Nachricht, die durch das Symbol repräsentiert wird, konform ist. Jedoch wurde in Abschnitt 4.3.3 vorausgesetzt, dass die Verhaltensmusterobjekte bereits zu Beginn der Verhaltenserkennung nichtdeterministisch gebunden werden. Hier gilt diese Voraussetzung nicht, daher werden die in Abschnitt 4.3.3 genannten Bedingungen erweitert und anschließend formal definiert.

Ein Methodenaufruf mc des Typs Behavior::MethodCall wird von einem Symbol der Form $(a : A) \rightarrow (b : B).m()$ akzeptiert, wenn folgende Bedingungen gelten:

1. Die Methode des Methodenaufrufs mc ist konform zu der Methode, die an die Variable m des Symbols gebunden wurde.

2. Der Typ der aufrufenden Instanz ist konform zu dem Typ, der an die Variable A des Symbols gebunden wurde.

3. Der Typ der aufgerufenen Instanz ist konform zu dem Typ, der an die Variable B des Symbols gebunden wurde.

4. Ist die Variable a des Symbols bereits gebunden, so ist sie an die aufrufende Instanz des Methodenaufrufs mc gebunden.

5. Ist die Variable b des Symbols bereits gebunden, so ist sie an die aufgerufene Instanz des Methodenaufrufs mc gebunden.

Hat das Symbol dagegen die Form $a \rightarrow (b : B).m()$, dann akzeptiert es den Methodenaufruf mc, wenn die Bedingungen 1, 3, 4 und 5 gelten.

Die Verhaltensmusterobjekte des Symbols werden unter folgenden Bedingungen gebunden:

- Ist die Variable a des Symbols vor Aufruf der Methode accept(MethodCall,Token) ungebunden, so wird sie an die aufrufende Instanz des Methodenaufrufs mc gebunden.

- Ist die Variable b des Symbols vor Aufruf der Methode accept(MethodCall, Token) ungebunden, so wird sie an die aufgerufene Instanz des Methodenaufrufs mc gebunden.

In Definition 5.2 sind diese Bedingungen in einer OCL-Nachbedingung formal beschrieben. Zur einfacheren Formulierung der Nachbedingung wird zunächst die boolsche Variable conform definiert, die angibt, ob der gegebene Methodenaufruf mc konform zu dem Symbol ist. Ist der Methodenaufruf mc konform, so gilt nach dem Aufruf der Methode accept(MethodCall,Token), dass die Verhaltensmusterobjekte an die zugehörigen Instanzen des Methodenaufrufs mc gebunden sind, falls sie vor Aufruf der Methode accept(MethodCall,Token) noch nicht gebunden waren. Des Weiteren ist der Wert der Variable conforms das Ergebnis der Methode accept(MethodCall,Token).

Definition 5.2 *Ein Symbol des Typs PermittedMethodCall akzeptiert einen Methodenaufruf des Typs MethodCall, wenn gilt:*

```
package Automaton
context PermittedMethodCall::accept(mc:Behavior::MethodCall,
          t:Token):Boolean
post: let conform:Boolean =
            mc.type.conformsTo(
              t@pre.annotation.nodes[self.methodName]) and
            (self.caller.typeName<>OCLVoid implies
              mc.caller.type.conformsTo(
                t@pre.annotation.nodes[self.caller.typeName])) and
            mc.callee.type.conformsTo(
              t@pre.annotation.nodes[self.callee.typeName]) and
            (t@pre.bindings[self.caller.name]<>OCLVoid
              implies t@pre.bindings[self.caller.name]=mc.caller) and
            (t@pre.bindings[self.callee.name]<>OCLVoid
              implies t@pre.bindings[self.callee.name]=mc.callee)
          in
        (conform and t@pre.bindings[self.caller.name]=OCLVoid
          implies t.bindings[self.caller.name]=mc.caller) and
        (conform and t@pre.bindings[self.callee.name]=OCLVoid
          implies t.bindings[self.callee.name]=mc.callee)) and
        result = conform
endpackage
```

Der Ausdruck t@pre beschreibt das Token t im Zustand vor dem Aufruf der Methode accept(MethodCall,Token).

Symbole des Typs ProhibitedMethodCall

Die Klasse ProhibitedMethodCall repräsentiert Symbole der Form $* \rightarrow (b : B).m()$, die zum Verwerfen des Traces genutzt werden. Ein solches Symbol akzeptiert einen Methodenaufruf mc des Typs MethodCall, wenn gilt:

1. Die Methode des Methodenaufrufs mc ist konform zu der Methode, die an die Variable m des Symbols gebunden wurde.
2. Der Typ der aufgerufenen Instanz ist konform zu dem Typ, der an die Variable B des Symbols gebunden wurde.
3. Die Variable b des Symbols ist an die aufgerufene Instanz des Methodenaufrufs mc gebunden.

Wenn Symbole des Typs ProhibitedMethodCall einen Methodenaufruf mc akzeptieren, wird das Token in einen verwerfenden Zustand verschoben und damit der gesamte Trace verworfen. Das ist allerdings nur korrekt, wenn zweifelsfrei feststeht, dass der Methodenaufruf mc zu der untersuchten Instanz des Kandidaten gehört, für die der Automat beziehungsweise das Token erzeugt wurde. Im Falle des ProhibitedMethodCall-Symbols kann diese Bedingung nur dann zweifelsfrei festgestellt werden, wenn das aufgerufene Verhaltensmusterobjekt des Symbols bereits gebunden ist. Ist es an die aufgerufene Instanz des Methodenaufrufs mc gebunden, verstößt der Methodenaufruf mc gegen das Verhaltensmuster und das Verwerfen des Traces ist korrekt. Ist es an eine andere Instanz gebunden, so gehört der Methodenaufruf mc nicht zu der untersuchten Instanz des Kandidaten und wird ignoriert.

Ist das aufgerufene Verhaltensmusterobjekt dagegen noch ungebunden, kann keine Aussage darüber getroffen werden, ob der Methodenaufruf mc zu der Instanz des Kandidaten gehört. In diesem Fall akzeptiert das Symbol den Methodenaufruf mc nicht, so dass er vom Automaten ignoriert wird. Allerdings wird die aufgerufene Instanz des Methodenaufrufs mc einer Menge von möglichen Bindungen für das aufgerufene Verhaltensmusterobjekt hinzugefügt. Diese Menge wird durch die Assoziation possibleBindings der Klasse Token (Abbildung 5.3) repräsentiert. In Abschnitt 5.3.6 wird erläutert, was die möglichen Bindungen bedeuten und wozu sie verwendet werden.

Definition 5.3 *Ein Symbol des Typs ProhibitedMethodCall akzeptiert einen Methodenaufruf des Typs MethodCall, wenn gilt:*

package Automaton

```
context ProhibitedMethodCall::accept(mc:Behavior::MethodCall,
            t:Token):Boolean
post: let typeConform:Boolean =
            mc.type.conformsTo(
              t@pre.annotation.nodes[self.methodName]) and
            mc.callee.type.conformsTo(
              t@pre.annotation.nodes[self.callee.typeName])
          in
        ((typeConform and
          t@pre.bindings[self.callee.name]=OCLVoid) implies
            t.possibleBindings[self.callee.name]→includes(mc.callee)) and
        result = typeConform and
          t@pre.bindings[self.callee.name]=mc.callee
endpackage
```

In Definition 5.3 sind die genannten Bedingungen in einer OCL-Nachbedingung zur Methode ProhibitedMethodCall::accept(MethodCall,Token) formal spezifiziert.

Symbole des Typs ProhibitedCaller

Die Klasse ProhibitedCaller repräsentiert Symbole der Form $* \notin \mathcal{C} \rightarrow (b : B).m()$, die ebenfalls zum Verwerfen des Traces verwendet werden. Ein solches Symbol akzeptiert einen Methodenaufruf mc des Typs MethodCall, wenn gilt:

1. Die Methode des Methodenaufrufs mc ist konform zu der Methode, die an die Variable m des Symbols gebunden wurde.

2. Der Typ der aufgerufenen Instanz ist konform zu dem Typ, der an die Variable B des Symbols gebunden wurde.

3. Die Variable b des Symbols ist an die aufgerufene Instanz des Methodenaufrufs mc gebunden.

4. Keines der Verhaltensmusterobjekte der Menge $\mathcal{C}$ ist an die aufrufende Instanz des Methodenaufrufs mc gebunden. Ist ein Verhaltensmusterobjekt o der Menge $\mathcal{C}$ ungebunden, so darf der Typ der aufrufenden Instanz nicht konform sein zu dem Typ, der an die Typvariable des Verhaltensmusterobjekts o gebunden wurde.

Auch bei einem Symbol des Typs ProhibitedCaller muss zweifelsfrei feststehen, dass der Methodenaufruf mc zur beobachteten Instanz des Kandidaten gehört, damit das Symbol den Methodenaufruf akzeptieren darf. Das bedeutet, das aufgerufene Verhaltensmusterobjekt muss an die aufgerufene Instanz des Methodenaufrufs mc gebunden sein. Ist das aufgerufene Verhaltensmusterobjekt nicht gebunden und sind alle anderen Bedingungen erfüllt, dann wird wie bei Symbolen des Typs ProhibitedMethodCall die aufgerufene Instanz des Methodenaufrufs mc der Menge von möglichen Bindungen für das aufgerufene Verhaltensmusterobjekt hinzugefügt. Der Methodenaufruf mc wird in diesem Fall vom Symbol nicht akzeptiert und damit vom Automaten ignoriert.

In Definition 5.4 sind die genannten Bedingungen in einer OCL-Nachbedingung zur Methode ProhibitedCaller::accept(MethodCall,Token) formal spezifiziert.

Definition 5.4 *Ein Symbol des Typs ProhibitedCaller akzeptiert einen Methodenaufruf des Typs MethodCall, wenn gilt:*

```
package Automaton
context ProhibitedCaller::accept(mc:Behavior::MethodCall,
          t:Token):Boolean
post: let typeConform:Boolean =
          mc.type.conformsTo(
            t@pre.annotation.nodes[self.methodName]) and
          mc.callee.type.conformsTo(
            t@pre.annotation.nodes[self.callee.typeName]) and
          self.permittedCallers→forAll(mco:MethodCallObject|
            (t@pre.bindings[mco.name]=OCLVoid implies
              not mc.caller.type.conformsTo(
                t@pre.bindings[mco.typeName])) or
            t@pre.bindings[mco.name]<>mc.callee)
        in
      ((typeConform and
        t@pre.bindings[self.callee.name]=OCLVoid) implies
          t.possibleBindings[self.callee.name]→includes(mc.callee)) and
      result = typeConform and
        t@pre.bindings[self.callee.name]=mc.callee
endpackage
```

5.3.5 Beispiel

Im Folgenden wird zur Erläuterung der Verhaltenserkennung der Trace eines *State*-Kandidaten (Abbildung 4.9, Seite 74) herangezogen, der während der Ausführung des Mediaplayers beobachtet wurde. Abbildung 5.9 zeigt diesen Trace. Zuvor wurde der *State*-Kandidat in der Strukturanalyse identifiziert und annotiert. Die Bindungen der Annotation sind in Tabelle 4.1 auf Seite 75 festgehalten.

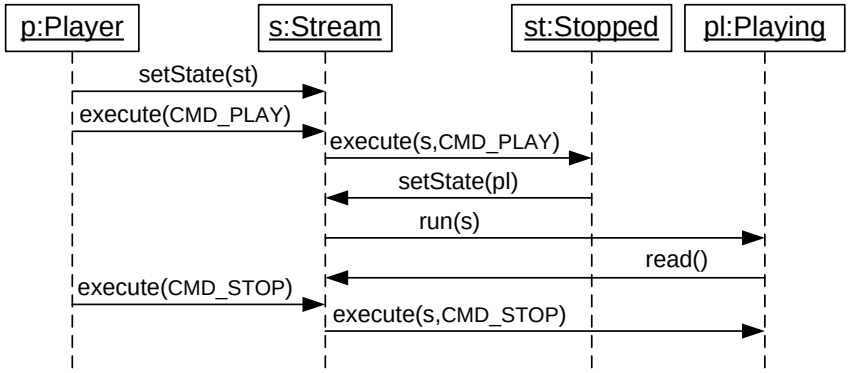

Abbildung 5.9: Beobachteter Trace des Mediaplayers

Der erste beobachtete Methodenaufruf des Traces triggert die Erkennung des *State*-Verhaltensmusters. Es wird ein Token erzeugt und in den Startzustand `0` des Automaten zum *State*-Verhaltensmuster (Abbildung 5.10) gelegt. Das Token wird mit der Annotation des *State*-Kandidaten initialisiert. Darin sind nur die Bindungen der Typ- und Methodennamen an Elemente des Kandidaten enthalten. Zu diesem Zeitpunkt ist noch keines der Verhaltensmusterobjekte gebunden.

Der erste Methodenaufruf des Traces wird nun vom Automaten verarbeitet, indem er an alle Zustände weitergereicht wird. Da es nur im Startzustand ein Token gibt, reicht der Startzustand den Methodenaufruf und das Token an die von ihm ausgehenden Transitionen weiter. Die Transition mit dem Symbol $client \rightarrow (c : context).setState()$ akzeptiert schließlich den Methodenaufruf. Dabei werden die beiden Verhaltensmusterobjekte *client* und c an die Instanzen `p` beziehungsweise `s` gebunden, der Methodenaufruf der Liste der konformen Methodenaufrufe hinzugefügt und das Token in den nachfolgenden Zustand verschoben. Abbildung 5.11 zeigt den Automaten, nachdem das Token verschoben wurde. In Tabelle 5.1 ist die Bindung der Verhaltensmusterobjekte nach dem ersten Zustandsübergang abgebildet.

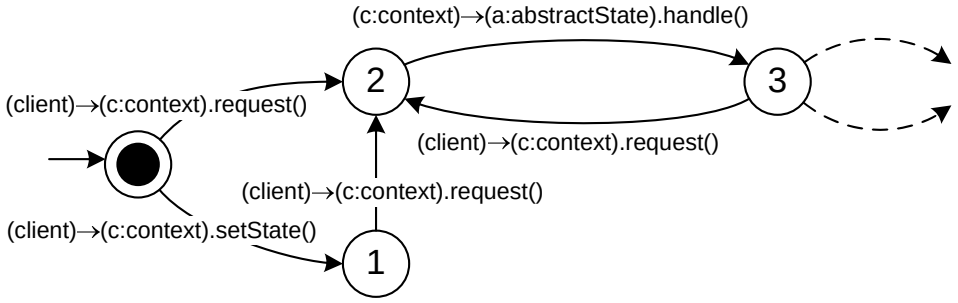

Abbildung 5.10: Ausschnitt aus dem Automaten des *State*-Verhaltensmusters mit Token

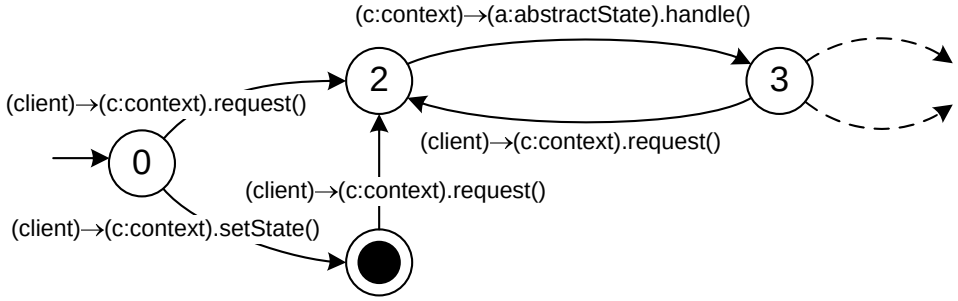

Abbildung 5.11: Automat des *State*-Verhaltensmusters nach dem Zustandsübergang

Die nächsten Methodenaufrufe werden ähnlich verarbeitet. Beim dritten Methodenaufruf des Traces wird das Verhaltensmusterobjekt a an die Instanz st gebunden. Der fünfte und der sechste Methodenaufruf werden ignoriert, da es keine Transitionen mit passenden Symbolen im Automaten gibt.

Mit dem letzten Methodenaufruf des Traces endet das Token in einem akzeptierenden Zustand des Automaten. Damit akzeptiert der Automat den Trace als konform zum Verhaltensmuster. In Abbildung 5.12 ist schließlich der Trace zu sehen, der dem Verhaltensmuster entspricht. Gegenüber dem beobachteten Trace fehlen die Methodenaufrufe, die vom Automaten ignoriert wurden. Dargestellt sind nur die Methodenaufrufe, die das Token in der Liste der konformen Methodenaufrufe gespeichert hat. Des Weiteren sind die Bindungen der Verhaltensmusterobjekte an die an den Methodenaufrufen beteiligten Instanzen dargestellt.

Verhaltensmusterobjekt	*Instanz*
client	p
c	s
a	-
b	-

Tabelle 5.1: Variablenbindung des Tokens nach dem ersten Zustandsübergang

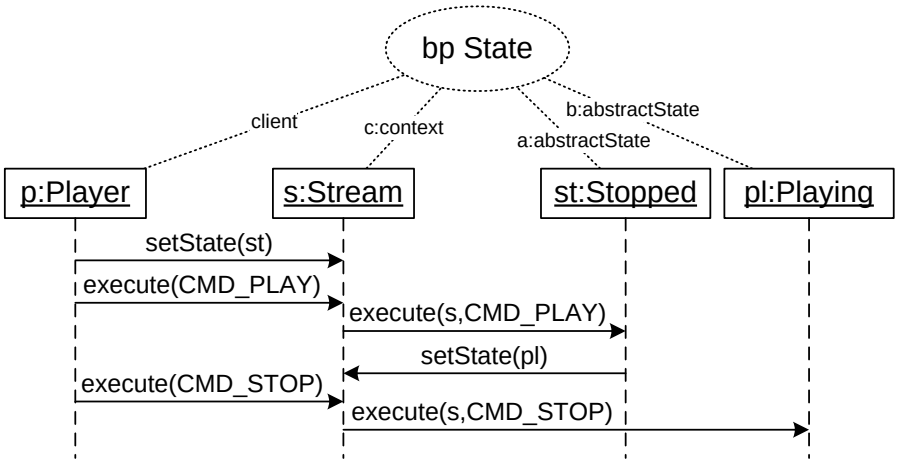

Abbildung 5.12: Erkannte Verhaltensmusterimplementierung des *State*-Kandidaten

5.3.6 Nachträgliches Verwerfen eines Traces

Das späte Binden der Verhaltensmusterobjekte während der Verhaltenserkennung führt zu Situationen, in denen nicht zweifelsfrei entschieden werden kann, ob ein beobachteter Methodenaufruf zu einer untersuchten Instanz eines Kandidaten gehört. Abbildung 5.13 zeigt einen Trace, der nicht konform zum *State*-Verhaltensmuster ist und zu solch einer Situation führt. Der Trace stammt vom *State*-Kandidaten des Mediaplayer-Beispiels (Abbildung 4.9, Seite 74).

Abbildung 5.14 zeigt einen Ausschnitt aus dem DFA des *State*-Verhaltensmusters. Es sind nur die Transitionen dargestellt, die bei der Verhaltenserkennung des Traces aus Abbildung 5.13 eine Rolle spielen.

Der erste Methodenaufruf des Traces triggert die Erkennung des *State*-Verhaltensmusters. Es wird ein Token erzeugt und in den Startzustand 0 des Automaten aus Abbildung 5.14 gelegt. Das Token wird mit der Annotation

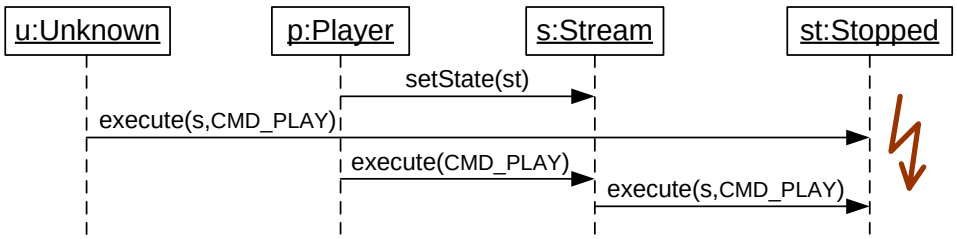

Abbildung 5.13: Zum *State*-Verhaltensmuster nicht-konformer Trace

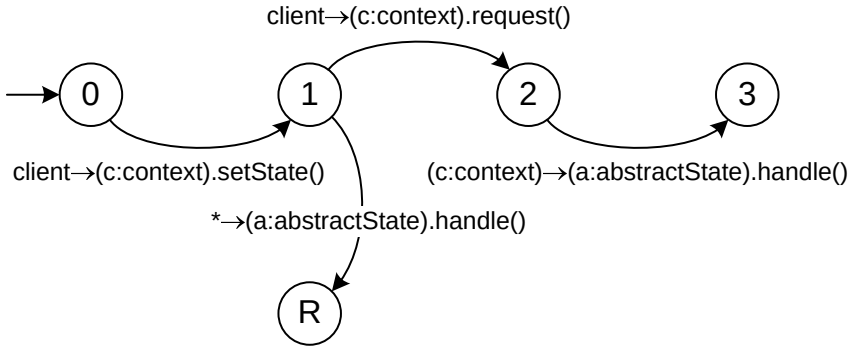

Abbildung 5.14: Ausschnitt aus dem Automaten des *State*-Verhaltensmusters

des *State*-Kandidaten initialisiert (siehe Tabelle 4.1, Seite 75).

Der erste Methodenaufruf wird nun von der Transition mit dem Symbol $client \rightarrow (c : context).setState()$ akzeptiert, so dass das Token in den Zustand 1 verschoben wird. In diesem Zustand sind die Verhaltensmusterobjekte *client* an die Instanz p und c an die Instanz s gebunden. Alle weiteren Verhaltensmusterobjekte sind noch ungebunden.

Der zweite Methodenaufruf, der beobachtet wird, ist (u:Unknown)→(st: Stopped).execute(s,CMD_PLAY). Dieser Methodenaufruf ist nicht konform zum Verhaltensmuster, da die handle()-Methode des *State*-Entwurfsmusters an dieser Stelle nicht aufgerufen werden darf. Außerdem darf sie nur vom Kontext-Objekt aufgerufen werden. Der Trace müsste also hier verworfen werden. Die Transition mit dem Symbol $* \rightarrow (a : abstractState).handle()$, die den Zustand 1 mit dem verwerfenden Zustand R verbindet, kann jedoch den Methodenaufruf nicht akzeptieren. Die Methode und der Typ der aufgerufenen Instanz sind zwar konform zu dem Symbol, das Verhaltensmusterobjekt a ist jedoch noch nicht gebunden. Es kann in dieser Situation also nicht zweifelsfrei festgestellt werden, ob der beobachtete Methodenaufruf zu der untersuchten Instanz des

State-Kandidaten gehört. Er könnte auch zu einer anderen Instanz des *State*-Kandidaten gehören. Nach Definition 5.3 darf deshalb der Methodenaufruf von der Transition nicht akzeptiert werden. Allerdings wird die Instanz st zu der Menge der möglichen Bindungen für das Verhaltensmusterobjekt a hinzugefügt. Da keine andere Transition den Methodenaufruf in diesem Zustand akzeptiert, wird er vom Automaten ignoriert.

Der dritte Methodenaufruf des Traces wird akzeptiert, so dass das Token in den Zustand 2 verschoben wird. Auch der vierte Methodenaufruf wird akzeptiert. Das Token befindet sich nun im Zustand 3. Bei der Verarbeitung des Methodenaufrufs wurde das Verhaltensmusterobjekt a an die aufgerufene Instanz st des Methodenaufrufs gebunden.

Erst zu diesem Zeitpunkt steht zweifelsfrei fest, dass die Instanz st zu der untersuchten Instanz des *State*-Kandidaten gehört. Es stellt sich heraus, das der zweite Methodenaufruf nicht hätte ignoriert werden dürfen, sondern zum Verwerfen des Traces hätte führen müssen.

Aus diesem Grund sind die möglichen Bindungen in das Modell der Automaten aufgenommen worden. Mögliche Bindungen werden nur durch Symbole hergestellt, deren Transitionen in den verwerfenden Zustand führen. Sie werden hergestellt, wenn nicht festgestellt werden kann, ob eine am beobachteten Methodenaufruf beteiligte Instanz zu der untersuchten Instanz eines Kandidaten gehört. Wird zu einem späteren Zeitpunkt festgestellt, dass die fragliche Instanz dazu gehört, kann der Trace nachträglich verworfen werden.

Dieses Konzept ist auf die klassischen Automaten nur sehr schwer abzubilden. Es wurde daher eine pragmatische Lösung gewählt. Nach jeder Verarbeitung eines Methodenaufrufs wird geprüft, ob ein Verhaltensmusterobjekt an eine Instanz gebunden wurden, die bereits in der Menge der möglichen Bindungen des Verhaltensmusterobjekts enthalten ist. Wird eine solche Bindung gefunden, wird der Trace durch das Verschieben des Tokens in den verwerfenden Zustand nachträglich verworfen.

Dieser Algorithmus ist in Abbildung 5.15 zu sehen. Er wird am Ende der Methode methodCalled der Klasse DFA (Abbildung 5.6, Seite 118) aufgerufen.

5.4 Bewertung der Ergebnisse

In Abschnitt 4.3.1 wurde bereits erläutert, dass zu jedem Kandidaten zur Laufzeit des zu untersuchenden Softwaresystems beliebig viele Traces beobachtet werden können. Das liegt zum einen daran, dass die Verhaltensmuster in der Regel nicht das globale Verhalten des Entwurfsmusters beschreiben, sondern

```
1: DFA::checkBindings()
2:   forEach state:State in self.states do
3:     forEach token:Token in state.tokens do
4:       forEach key:String in token.bindings→keys() do
5:         if token.possibleBindings[key]→
               includes(token.bindings[key]) then
6:           token.state.tokens→remove(token)
7:           self.rejectingState.tokens→add(token)
```

Abbildung 5.15: Die Methode checkBindings der Klasse DFA

Ausschnitte typischen lokalen Verhaltens. Eine Instanz des Kandidaten kann also dieses lokale Verhalten zur Laufzeit mehrfach durchlaufen. Des Weiteren kann der Kandidat auch mehrfach instanziiert werden, so dass in diesem Fall mehrere Instanzen des Kandidaten beobachtet werden.

Diese Traces werden auf Konformität zu dem positiven Verhaltensmuster, aber auch auf Konformität zu den möglichen negativen Verhaltensmustern des zum Kandidaten gehörenden Entwurfsmusters untersucht. Die Traces können sowohl konform, als auch nicht konform zu den Verhaltensmustern sein. All dies führt dazu, dass zu einem Kandidaten am Ende der Verhaltensanalyse mehrere Traces vorliegen, aus denen sich unter Umständen widersprüchliche Rückschlüsse auf das Verhalten des Kandidaten ziehen lassen.

Die Token, die von den Traces getriggert wurden, enthalten die Analyseergebnisse. Es gibt verschiedene Szenarien von Zustandsübergängen, die ein Token während der Verhaltenserkennung durchlaufen kann. Ein Token kann am Ende in einem nicht-akzeptierenden, einem akzeptierenden oder dem verwerfenden Zustand liegen. Ein akzeptierender Zustand kann von einem Token mehrfach durchlaufen werden. Wie an dem Automaten des *State*-Verhaltensmusters (Abbildung 4.25, Seite 95) zu sehen ist, führt eine Schleife am Ende des Verhaltensmusters dazu, dass das Token einen akzeptierenden Zustand wieder verlassen kann. Es ist sogar möglich, dass ein Token nach dem Verlassen eines akzeptierenden Zustands irgendwann in den verwerfenden Zustand gelangt. Das bedeutet, der beobachtete Trace war zu einem bestimmten Zeitpunkt konform zum Verhaltensmuster. Dann folgte aber irgendwann ein Methodenaufruf, der nicht konform war. Den verwerfenden Zustand kann ein Token jedoch nicht mehr verlassen, da der verwerfende Zustand eine Senke ist.

Am Ende werden die Tokens aller aktiven Automaten ausgewertet. Jedes Token wurde durch einen Trace getriggert und gehört deshalb zu einem bestimm-

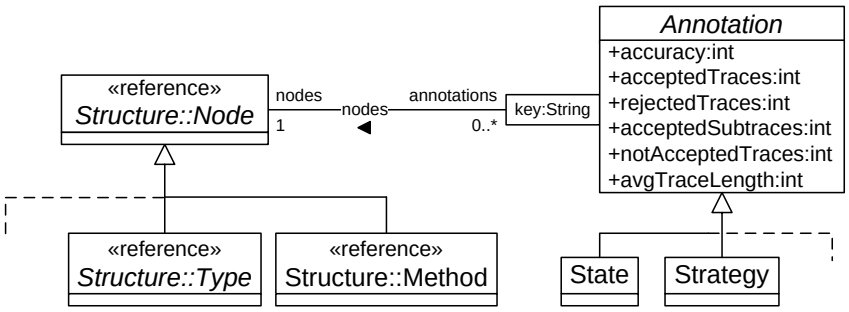

Abbildung 5.16: Erweitertes Modell der Annotationen, Paket Annotations

ten Kandidaten. Zu jedem Kandidaten werden fünf Messwerte festgehalten, die aus den Token berechnet und in der Annotation des Kandidaten gespeichert werden. Dazu wurde die abstrakte Klasse Annotation, von der alle konkreten Annotationen der Entwurfsmuster erben, um einige Attribute ergänzt. Das erweiterte Modell der Annotationen ist in Abbildung 5.16 dargestellt.

Die folgenden Messwerte werden für jeden Kandidaten aus seinen Token berechnet. Die Messwerte werden jedoch nur für Token aus solchen Automaten berechnet, die zu positiven Verhaltensmustern gehören.

- **acceptedTraces**: Anzahl der akzeptierten Traces. Diese Anzahl setzt sich unter anderem aus der Anzahl aller Tokens zusammen, die in einem akzeptierenden Zustand liegen. Des Weiteren werden aber auch alle Tokens hinzu gezählt, die zwar in einem nicht-akzeptierenden oder in dem verwerfenden Zustand liegen, aber mindestens einmal während der Analyse einen akzeptierenden Zustand durchlaufen haben.

- **rejectedTraces**: Anzahl der verworfenen Traces. Dies ist die Anzahl der Tokens, die in dem verwerfenden Zustand liegen.

- **notAcceptedTraces**: Anzahl der nicht akzeptierten Traces. Dies ist die Anzahl der Tokens, die in einem nicht-akzeptierenden Zustand liegen.

- **acceptedSubtraces**: Anzahl der verworfenen und nicht akzeptierten Traces, die aber einen Subtrace enthalten, der akzeptiert wurde. Die Traces waren bis zu einem bestimmten Zeitpunkt konform zum Verhaltensmuster, verließen aber den akzeptierenden Zustand wieder.

- **avgTraceLength**: Durchschnittliche Länge aller akzeptierten Traces. Der Durchschnitt wird über die Anzahl der konformen Methodenauf-

rufe aller akzeptierten Traces gebildet. Bei Tokens, die in einem nicht-akzeptierenden oder in dem verwerfenden Zustand liegen und mehrfach einen akzeptierenden Zustand durchliefen, wird für den Durchschnitt die Anzahl der konformen Methodenaufrufe zu dem Zeitpunkt gewählt, zu dem das Token zum letzten Mal in einem akzeptierenden Zustand lag.

Wie in Abschnitt 4.3.5 erläutert, werden bei einem negativen Verhaltensmuster nur Traces gewertet, die konform zu dem negativen Verhaltensmuster sind. Diese Traces gehen dann als negatives Indiz in die Gesamtbewertung des Kandidaten ein. Deshalb wird die Anzahl der Tokens, die in einem akzeptierenden Zustand eines Automaten eines negativen Verhaltensmusters liegen, zu der Anzahl der verworfenen Traces hinzu gefügt.

Diese Messwerte werden dem Reverse-Engineer zusammen mit der Bewertung aus der Strukturanalyse als Gesamtergebnis der struktur- und verhaltensbasierten Entwurfsmustererkennung präsentiert. Auf eine Verrechnung der Einzelwerte zu einem einzigen Wert wird bewusst verzichtet. Die Einzelwerte sind sehr viel aussagekräftiger als ein einzelner Wert, der als eine absolute Aussage interpretiert werden könnte. Das in dieser Arbeit vorgestellte Verfahren erhebt keinen Anspruch, absolute Aussagen über das zu untersuchende Softwaresystem zu treffen. Die Interpretation der Ergebnisse soll vielmehr dem Reverse-Engineer überlassen werden.

Aus den ermittelten Messwerten der Verhaltensanalyse kann der Reverse-Engineer verschiedene Rückschlüsse auf den Kandidaten ziehen. Das Verhältnis der Anzahl akzeptierter Traces zu den verworfenen Traces ist sicher ein guter Hinweis darauf, ob ein Kandidat eine tatsächliche Entwurfsmusterimplementierung darstellt. Ein hoher Wert für dieses Verhältnis deutet eher auf eine tatsächliche Entwurfsmustererkennung hin, als ein niedriger. Eine hohe durchschnittliche Länge der akzeptierten Traces ist ein weiterer Indikator für eine tatsächliche Entwurfsmusterimplementierung. Die Wahrscheinlichkeit, dass kurze Traces zufällig zu einem Verhaltensmuster konform sind, ist wesentlich höher, als bei sehr langen Traces. Basiert eine hohe durchschnittliche Länge der Traces zudem auf sehr vielen akzeptierten Traces, ist die Wahrscheinlichkeit für ein False Positive gering.

5.5 Zusammenfassung

In diesem Kapitel ist die Verhaltensanalyse durch die im Kapitel 4 eingeführten Verhaltensmuster und den daraus generierten Automaten vorgestellt worden.

Der Prozess der Verhaltensanalyse wird auf Basis des Verhaltensmusterkatalogs, der Ergebnisse aus der Strukturanalyse und dem zu untersuchenden, ausführbaren Programm gestartet.

Zur Gewinnung der Traces zur Laufzeit des zu untersuchenden Softwaresystems wurden zwei Verfahren vorgestellt, die auf zwei grundsätzlich unterschiedlichen Prinzipien aufbauen. Beide Verfahren haben Vor- und Nachteile. Das Debugging ist ohne weitere Prozessschritte direkt mit den Eingaben der Verhaltensanalyse einsetzbar. Die im Rahmen dieser Arbeit implementierte Lösung ermöglicht die Überwachung von Softwaresystemen, die in der Programmiersprache Java erstellt wurden. Das Verfahren ist jedoch sehr leicht auf andere Programmiersprachen übertragbar. Allerdings ist die Performanz des Verfahrens gering. Es ist daher in produktiven Umgebungen nur sehr bedingt einsetzbar.

Die Instrumentierung bietet dagegen eine sehr performante Überwachung des Softwaresystems. Die Übertragung des Verfahrens auf andere Programmiersprachen ist jedoch sehr aufwendig. Des Weiteren ist ein zusätzlicher Prozessschritt vor der eigentlichen Verhaltenserkennung notwendig. Das Debugging eignet sich somit eher für kurze Tests der Verhaltensanalyse, in denen sie zum Beispiel iterativ eingesetzt wird, um die Eignung von Verhaltensmustern zu überprüfen. Die Instrumentierung ist dagegen eher geeignet, um letztendlich relevante Ergebnisse zu generieren, die eventuell sogar aus einem produktivem Einsatz des Softwaresystems stammen.

Für die Verhaltenserkennung ist ein Modell vorgestellt worden, das auf die im letzten Kapitel vorgestellten Automaten aufbaut. Die Automaten wurden um so genannte Tokens erweitert, um den Speicheraufwand der Verhaltensanalyse zu verringern. Es wurde formal definiert, wann die Erkennung eines Verhaltensmusters ausgelöst wird und wie die beobachteten Methodenaufrufe durch die Automaten verarbeitet werden.

Im Gegensatz zur theoretischen Betrachtung in Kapitel 4.26 ist es in der realen Umsetzung der Verhaltenserkennung nicht möglich, die Verhaltensmusterobjekte zu Beginn nichtdeterministisch zu binden. Aus diesem Grund wurde formal definiert, wie die Verhaltensmusterobjekte während der Verhaltenserkennung gebunden werden.

Zum Abschluss des Kapitels wurde vorgestellt, welche Daten dem Reverse-Engineer als Ergebnis der gesamten struktur- und verhaltensbasierten Entwurfsmustererkennung präsentiert werden und wie im Besonderen die Ergebnisse der Verhaltensanalyse zu interpretieren sind.

Kapitel 6

Praktische Anwendung

Dieses Kapitel befasst sich mit der Anwendung der struktur- und verhaltensbasierten Entwurfsmustererkennung auf ein praxisnahes Softwaresystem. Zunächst werden im ersten Teil des Kapitels einige Grundlagen zur dynamischen Analyse eines produktiv eingesetzten Softwaresystems erläutert. Im zweiten Teil wird ein Szenario beschrieben, das im Rahmen dieser Arbeit für einen Praxistest des Ansatzes herangezogen wurde. Schließlich werden die Ergebnisse präsentiert und diskutiert. Aus der Diskussion der Ergebnisse werden mögliche Verbesserungen abgeleitet.

6.1 Software-Tomographie

Ein zentrales Problem dynamischer Analysen ist die angemessene Auswahl der Eingabedaten, die zur Ausführung des zu analysierenden Softwaresystems benötigt werden. Das in der Praxis durch dynamische Analysen beobachtete Verhalten eines Softwaresystems stellt immer nur einen kleinen Teil des theoretisch möglichen Verhaltens dar. Um also verwertbare Ergebnisse zu erhalten, sollten die Eingabedaten möglichst repräsentativ für die in der Praxis auftretenden Daten ausgewählt werden.

Das Problem der repräsentativen Eingabedaten lässt sich sehr elegant lösen, indem das zu untersuchende Softwaresystem während der dynamischen Analyse in einer realen, produktiven Umgebung eingesetzt wird. Das setzt allerdings voraus, dass der Einfluss der dynamischen Analyse auf das produktive Softwaresystem so gering wie möglich gehalten wird.

Jim Bowring, Alessandro Orso und Mary Jean Harrold [BOH02] schlagen daher die *Software-Tomographie* vor. Software-Tomographie wird eingesetzt, um mit minimalem Einfluss produktiv eingesetzte Software während ihrer Ausführung zu beobachten. Voraussetzung für den Einsatz der Software-

Tomographie ist, dass die Aufgabe der dynamischen Analyse in sehr viele, voneinander unabhängige Teilaufgaben aufgeteilt werden kann. Die Teilaufgaben müssen sich wiederum durch eine möglichst minimale Instrumentierung des zu untersuchenden Softwaresystems lösen lassen.

Eine weitere Voraussetzung für Software-Tomographie ist, dass das Softwaresystem in der Praxis in mehreren Instanzen eingesetzt wird. In diesem Fall kann die dynamische Analyse in viele Teilanalysen aufgeteilt werden, indem eine Instanz jeweils für eine oder einige wenige Teilaufgaben instrumentiert wird. Die Performanz einer einzelnen Instanz wird dadurch nur sehr wenig verringert, so dass sie in der produktiven Umgebung eingesetzt werden kann. Die weitere Aufgabe der Software-Tomographie besteht darin, die so gewonnenen Daten der einzelnen Teilanalysen zu sammeln und wieder zu einem Gesamtergebnis zusammenzufassen.

In der struktur- und verhaltensbasierten Entwurfsmustererkennung ist die Aufteilung der dynamischen Analyse in Teilanalysen bereits vorgegeben. Jeder in der Strukturanalyse identifizierte Kandidat stellt eine Teilanalyse dar, die unabhängig von den anderen Kandidaten ausgeführt werden kann. Die Instanzen des zu untersuchenden Softwaresystems werden also jeweils für einen oder einige wenige Kandidaten instrumentiert. Auch die Ergebnisse der einzelnen Teilanalysen sind unabhängig voneinander, so dass sie zur Auswertung nur gesammelt werden müssen.

6.2 Szenario

Das in der vorliegenden Arbeit entwickelte Verfahren wurde auf ein praxisnahes Softwaresystem angewendet. Voraussetzungen für das zu untersuchende Softwaresystem waren eine hinreichende Größe, die Verwendung von Entwurfsmustern bei der Entwicklung des Softwaresystems und deren Dokumentation. Das Softwaresystem sollte zu groß sein, um von einem Reverse-Engineer noch manuell analysiert werden zu können. Kleine Systeme enthalten außerdem entsprechend weniger Entwurfsmusterimplementierungen als große. Eine automatische Entwurfsmustererkennung macht also nur Sinn bei größeren Softwaresystemen, die bei mehreren zehntausend, besser noch mehreren hunderttausend Zeilen Quelltext liegen.

Wurden bei der Entwicklung des Softwaresystems Entwurfsmuster explizit zum Design verwendet und dokumentiert, erleichtert dies den Praxistest des Verfahrens. Die explizite Verwendung ist natürlich keine Voraussetzung zur Anwendung der Entwurfsmustererkennung. Wie bereits in der Einleitung

erläutert, können Entwurfsmusterimplementierungen auch in Softwaresystemen gefunden werden, bei denen Entwurfsmuster nicht explizit im Design verwendet wurden. In dem Praxistest ist die Dokumentation der Entwurfsmusterimplementierungen allerdings sehr hilfreich zur Beurteilung der Qualität der automatischen Entwurfsmustererkennung.

Zur Entwurfsmustererkennung bietet sich ECLIPSE an. ECLIPSE erfüllt die zuvor genannten Voraussetzungen. Es ist ein relativ großes, in JAVA geschriebenes Softwaresystem, bei dem während der Entwicklung Entwurfsmuster eingesetzt wurden. Erich Gamma und Kent Beck, die zu den Entwicklern von ECLIPSE gehören, haben in ihrem Buch „Contributing to eclipse - Principles, Patterns, and Plug-Ins“ einige der in ECLIPSE enthaltenen Entwurfsmusterimplementierungen dokumentiert [GB04]. Eine Auswahl der dokumentierten Entwurfsmusterimplementierungen mit Seitenreferenzen auf [GB04] ist in Tabelle 6.1 zu finden.

Erich Gamma und Kent Beck haben als Grundlage für ihr Buch die Version 2.1 von ECLIPSE verwendet. In dieser Version besteht ECLIPSE aus etwa 66 Plug-Ins. Die dokumentierten Entwurfsmusterimplementierungen konzentrieren sich aber auf einige wenige Plug-Ins, die im Folgenden der Entwurfsmustererkennung unterzogen wurden. In Tabelle 6.2 sind die untersuchten Plug-Ins aufgelistet. Außerdem ist zu jedem Plug-In angegeben, aus wie vielen Klassen es besteht. Insgesamt wurden also mehr als 2400 Klassen untersucht, so dass ein praxisnahes Szenario gegeben ist.

Die Klassen der Plug-Ins aus Tabelle 6.2 wurden zunächst mit Hilfe der strukturbasierten Entwurfsmustererkennung untersucht. Auf Grundlage der erkannten Kandidaten wurde anschließend die verhaltensbasierte Entwurfsmustererkennung durchgeführt. Dazu wurde während der Ausführung mit ECLIPSE ein kurzes Beispielprogramm implementiert. Des weiteren waren bereits zuvor umfangreiche Projekte in die Arbeitsumgebung von ECLIPSE importiert worden, in denen unter anderem Suchen durchgeführt wurden. So entstand ein realistisches Szenario wie in einer produktiven Umgebung. Die Traces wurden aufgezeichnet und offline analysiert. Die Struktur- und Verhaltensmuster des verwendeten Kataloges sind in Anhang A zu finden.

6.3 Ergebnisse

Ein Auszug der Ergebnisse des Praxistests der Struktur- und Verhaltensanalyse wird im Folgenden präsentiert. Im Anschluss werden einige konzeptionelle Schwächen, die während des Praxistests erkannt wurden, erläutert. Es werden

Entwurfs-musterimpl.	Rolle	Klasse/Methode
Observer 1 S. 303	Subject register Observer notify	org.eclipse.core.resources.IWorkspace addResourceChangeListener() org.eclipse.core.resources.IResourceChangeListener resourceChanged(IResourceChangeEvent)
Observer 2 S. 333	Subject register Observer notify	org.eclipse.swt.widgets.Button addSelectionListener() org.eclipse.swt.events.SelectionListener widgetSelected(SelectionEvent)
Observer 3 S. 343	Subject register Observer notify	org.eclipse.jface.action.IAction addPropertyChangeListener() org.eclipse.jface.util.IPropertyChangeListener propertyChange()
Strategy 1 S. 332	Context setStrategy Strategy algorithm	org.eclipse.swt.widgets.Composite setLayout() org.eclipse.swt.widgets.Layout layout()
Strategy 2 S. 341	Context setStrategy Strategy algorithm	org.eclipse.jface.viewers.StructuredViewer addFilter() org.eclipse.jface.viewers.ViewerFilter filter()
Strategy 3 S. 341	Context setStrategy Strategy algorithm	org.eclipse.jface.viewers.StructuredViewer setSorter() org.eclipse.jface.viewers.ViewerSorter sort()

Tabelle 6.1: Entwurfsmusterimplementierungen in Eclipse [GB04]

jedoch Ideen präsentiert, mit denen diese Schwächen behoben werden können, die aber aus Zeitgründen in dieser Arbeit nicht mehr umgesetzt werden konnten.

6.3.1 Strukturanalyse

In Tabelle 6.3 sind die Ergebnisse zu den in der Strukturanalyse erkannten Kandidaten der oben genannten Entwurfsmusterimplementierungen zu finden. Alle Entwurfsmusterimplementierungen wurden korrekt als Kandidaten der jeweiligen Entwurfsmuster identifiziert. Die drei *Strategy*-Implementierungen wurden jedoch aufgrund der sehr ähnlichen Strukturmuster sowohl als *Stra-*

Plug-In	Anzahl Klassen
org.eclipse.core.resources_2.1.3	242
org.eclipse.jdt.core_2.1.3/model	295
org.eclipse.jdt.ui_2.1.3	690
org.eclipse.jface.text_2.1.0	145
org.eclipse.jface_2.1.3	165
org.eclipse.swt_2.1.3/common	107
org.eclipse.swt_2.1.3/win32	54
org.eclipse.ui.workbench_2.1.3	710
Summe	2408

Tabelle 6.2: Analysierte Plug-Ins von ECLIPSE

tegy- als auch als *State*-Kandidaten identifiziert. Die *State*-Kandidaten sind False-Positives.

Entwurfsmusterimpl.	Kandidat	Bewertung
Observer 1	Observer	16,56%
Observer 2	Observer	11,48%
Observer 3	Observer	60,82%
Strategy 1	Strategy	73,27%
	State	73,27%
Strategy 2	Strategy	88,88%
	State	88,88%
Strategy 3	Strategy	67,29%
	State	67,29%

Tabelle 6.3: Ergebnisse der Strukturanalyse

Die Bewertung der Kandidaten auf Basis ihrer Übereinstimmung mit den Strukturmustern ist sehr unterschiedlich. Das *Observer*-Strukturmuster enthält mehr nicht notwendige Anteile als die *Strategy*- und *State*-Strukturmuster. Daher sind hier zwischen den Kandidaten größere Unterschiede in der Bewertung zu finden. Auffällig ist, dass bei allen drei *Strategy*-Implementierungen die erkannten *Strategy*- und *State*-Kandidaten jeweils die gleiche Bewertung erhalten haben. Allerdings lässt sich diese Tatsache durch die fast identischen Strukturmuster erklären.

6.3.2 Verhaltensanalyse

In der Verhaltensanalyse konnten zwei der drei *Strategy*-Implementierungen klar bestätigt werden. In Tabelle 6.4 sind die Ergebnisse zu den drei *Strategy*-Implementierungen zu sehen. Die erste *Strategy*-Implementierung ist Teil des Layout-Algorithmus der Benutzungsschnittstelle und wurde sehr häufig ausgeführt. Das Verhältnis der akzeptierten zu den nicht-akzeptierten Traces bestätigt mit 195 zu 14 deutlich den *Strategy*-Kandidaten. Dagegen wurde beim *State*-Kandidaten kein einziger Trace akzeptiert, dafür aber über 1200 Traces verworfen.

Entwurfs-musterimpl.	Kandidat	akzept. Traces	verworfene Traces	nicht akzept. Traces	akzept. Subtraces	durchschnittl. Tracelänge
Strategy 1	Strategy	195	14	69	7	5,6
	State	0	1236	431	0	0,0
Strategy 2	Strategy	2	10	4	0	48,5
	State	0	22	6	0	0,0
Strategy 3	Strategy	12	0	0	0	16,5
	State	0	156	22	0	0,0

Tabelle 6.4: Ergebnisse der Verhaltensanalyse

Die dritte *Strategy*-Implementierung wurde nur sehr selten ausgeführt. Die Strategie wird zur Sortierung von Elementen einer Sicht wie zum Beispiel eines Projekt-Baumes verwendet. Allerdings ist klar zu erkennen, dass auch hier der *Strategy*-Kandidat bestätigt wurde. Es wurde kein Trace verworfen und die durchschnittliche Länge der akzeptierten Traces ist zudem relativ hoch. Der *State*-Kandidat wurde auch hier klar verworfen.

Bei der zweiten *Strategy*-Implementierung ist das Ergebnis nicht so deutlich. Diese Strategie wird ebenfalls in Sichten verwendet, um Elemente aus der Sicht heraus zu filtern. Nur zwei Traces des *Strategy*-Kandidaten wurden akzeptiert, wobei sie jedoch eine sehr hohe durchschnittliche Länge von 48,5 Methodenaufrufen aufweisen. Dagegen wurden aber zehn Traces verworfen. Bei näherer Betrachtung des Kandidaten und der verworfenen Traces stellte sich heraus, dass in einigen Fällen mehrere Filter gleichzeitig in einem Kontext benutzt werden. Soll ein Element in der Sicht angezeigt werden, wird eine Anfrage an den Kontext gestellt. Diese Anfrage wird vom Kontext an die Filter weiter gegeben. Dabei entscheidet zunächst der erste Filter, ob das Element angezeigt wird. Akzeptiert der Filter, wird die Anfrage an den nächsten Filter geschickt. Dies wird solange wiederholt, bis kein Filter mehr übrig ist, oder ein Filter

das Element ablehnt. Lehnt immer der erste Filter das Element ab, verhält sich die Implementierung wie eine *Strategy*. Wird die Anfrage aber an mehrere Filter geschickt, so verhält sie sich wie eine *Chain of Responsibility*. Bei diesem Entwurfsmuster wird eine Anfrage solange in einer Kette von Objekten weitergereicht, bis sich ein Objekt für die Anfrage verantwortlich zeigt und sie bearbeitet. Es handelt sich bei diesem Kandidaten also entweder um eine falsch dokumentierte Entwurfsmusterimplementierung oder um eine sehr frei interpretierte *Strategy*-Implementierung.

Von den *Observer*-Kandidaten konnte keiner durch die Verhaltensanalyse bestätigt werden. Die Gründe hierfür werden im Folgenden erläutert.

6.3.3 Schwächen des Ansatzes

Bei der praktischen Anwendung sind einige Schwächen des Ansatzes zu Tage getreten. Die Spezifikationssprache ist leider in einigen Fällen noch nicht ausdrucksstark genug, um das Verhalten eines Entwurfsmuster passend zu beschreiben.

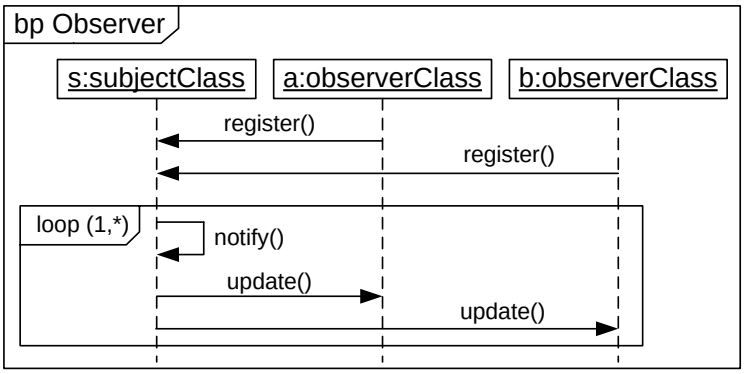

Abbildung 6.1: Das *Observer*-Verhaltensmuster

In Abbildung 6.1 ist das im Praxistest verwendete *Observer*-Verhaltensmuster zu sehen. Hier registrieren sich zwei Objekte des Typs observerClass bei einem Objekt vom Typ subjectClass. Ändert sich am subjectClass-Objekt etwas, ruft es die Methode *notify()* auf, die daraufhin die beiden observerClass-Objekte durch Aufruf der Methode *update()* benachrichtigt. Das Problem bei diesem Verhaltensmuster ist die feste Anzahl der observerClass-Objekte. In Sequenzdiagrammen nach der UML 2.0, an denen die Spezifikationssprache für Verhaltensmuster angelehnt ist, ist immer nur eine festgelegte Anzahl von Ob-

jekten erlaubt. Es ist also nicht möglich, den Nachrichtenaustausch zwischen einer beliebigen Anzahl von Objekten zu spezifizieren.

Zur Laufzeit eines Programms können sich jedoch beliebig viele observerClass-Objekte registrieren. Registrieren sich in einem konkreten Fall also zum Beispiel drei observerClass-Objekte, so wird bei der Registrierung des dritten Objektes der Trace abgelehnt, da ein zu diesem Zeitpunkt nicht erlaubter Methodenaufruf beobachtet wird. Dieser Fall tritt immer dann auf, wenn sich nur ein oder mehr als drei Objekte beim subjectClass-Objekt registrieren. Somit ist die Wahrscheinlichkeit, dass ein Trace akzeptiert wird, gering.

Die Lösung dieses Problems liegt in der Erweiterung der Spezifikationssprache. Abbildung 6.2 zeigt eine modifizierte Version des *Observer*-Verhaltensmusters in einer erweiterten Syntax. Hier wird ein mengenwertiges Verhaltensmusterobjekt für die *Observer* eingesetzt. Dem Verhaltensmusterobjekt wird nun bei der Verhaltensanalyse nicht mehr nur eine einzige Instanz zugeordnet, sondern eine beliebig große Anzahl.

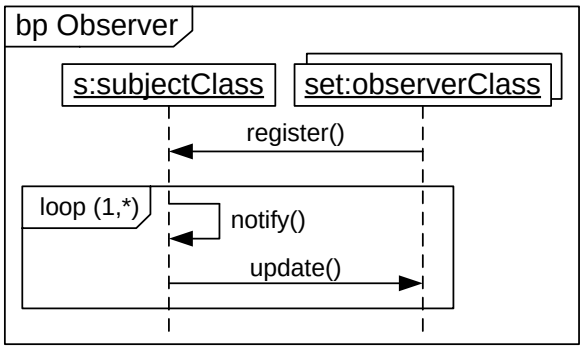

Abbildung 6.2: Das *Observer*-Verhaltensmuster in erweiterter Syntax

Die Nachrichten register() und update() können nun so interpretiert werden, dass es keine einzelnen Nachrichten sind, sondern eine Sequenz von Nachrichten. Jede Instanz der Menge muss nun zur Laufzeit genau einmal die Methode register() aufrufen. Genau so muss die Instanz, die dem subjectClass-Objekt zugeordnet ist, auf jeder Instanz der Menge genau einmal die Methode update() aufrufen. Mit Hilfe eines solchen Verhaltensmusters hätten die *Observer*-Kandidaten aus der Strukturanalyse bestätigt werden können.

Das *State*-Verhaltensmuster (Seite 59, Abbildung 4.1) weist eine ähnliche Schwäche auf. Das *State*-Entwurfsmuster beschreibt den Austausch verschiedener Zustände, um Anfragen an einen Kontext abhängig von dessen Zustand bearbeiten zu können. Die Anzahl der zur Laufzeit verwendeten Zustände ist aber

nicht festgelegt. Das im Praxistest verwendete Verhaltensmuster beschreibt jedoch nur einen einzigen Zustandswechsel zwischen zwei Zuständen. Findet ein weiterer Zustandswechsel statt, wird der Trace fälschlicherweise verworfen. Allerdings wird ein Subtrace dieses Traces akzeptiert.

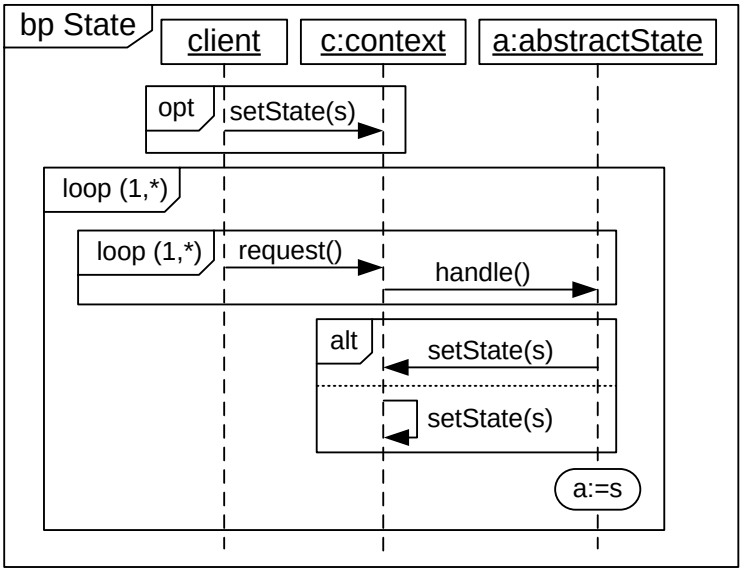

Abbildung 6.3: Das *State*-Verhaltensmuster in erweiterter Syntax

In Abbildung 6.3 ist ein weiterer Vorschlag zur Erweiterung der Syntax zu sehen, mit dem diese Schwäche behoben werden kann. In den Nachrichten werden zusätzlich Argumente spezifiziert. Hier wurde der Methode setState das Argument s hinzugefügt. Nach dem Wechsel des Zustands in der Alternative, ausgelöst durch den Methodenaufruf setState(s) auf dem Kontextobjekt c, wird dem Zustandsobjekt des Verhaltensmusters eine andere Instanz zugewiesen. Semantisch bedeutet dies, dass sich die Identität des Verhaltensmusterobjekts a:abstractState ändert. Wird nun die äußere Schleife wiederholt, so wird der Methodenaufruf handle() auf einer anderen Instanz als zuvor ausgeführt. Die Instanz wird durch das Argument des setState(s)-Aufrufs festgelegt. So kann man beliebig viele Zustandswechsel zwischen beliebig vielen Zuständen in einem Verhaltensmuster spezifizieren. Die Fehlinterpretation mehrfacher Zustandswechsel wie zuvor diskutiert würde damit korrigiert und die Präzision der Verhaltensanalyse erhöht.

6.4 Zusammenfassung

Die Anwendung der struktur- und verhaltensbasierten Entwurfsmustererkennung auf ein praxisnahes Softwaresystem stößt auf einige praktische Probleme. Das Kapitel beschreibt diese Probleme und präsentiert zugleich Lösungen. Zunächst einmal ist ganz allgemein die Auswahl der Eingabedaten für dynamische Analysen ein zentrales Problem. Das in der Praxis beobachtete Verhalten kann immer nur einen Bruchteil des theoretisch möglichen Verhaltens darstellen. Eine möglichst repräsentative Auswahl der Eingabedaten ist durch eine Beobachtung eines produktiv eingesetzten Softwaresystems möglich. Um aber die Beeinflussung durch die dynamische Analyse möglichst gering zu halten, wird die Software-Tomographie nach [BOH02] für die Verhaltenserkennung vorgeschlagen. Dabei wird die Gesamtanalyse auf mehrere Teilanalysen aufgeteilt. Diese werden auf verschiedenen Instanzen des zu untersuchenden Softwaresystems durchgeführt. So bleibt der Einfluss der dynamischen Analyse auf die Laufzeit des produktiven Softwaresystems gering und man erhält repräsentative Daten.

Für einen Praxistest der Entwurfsmustererkennung wurde ein Szenario präsentiert, für das zunächst einige Voraussetzungen festgelegt wurden. Das zu untersuchende System soll groß sein und zudem möglichst auf Basis von Entwurfsmustern implementiert worden sein, die auch dokumentiert sind. So können die Ergebnisse der Entwurfsmustererkennung mit der Realität verglichen werden. Als Softwaresystem wurde Eclipse 2.1 ausgewählt.

Auf dieses System wurde die Entwurfsmustererkennung angewendet. Die Strukturanalyse konnte einige der dokumentierten Entwurfsmusterimplementierungen korrekt identifizieren, fand jedoch auch False-Positives. Von den in der Strukturanalyse korrekt erkannten Kandidaten konnte die Verhaltensanalyse einige bestätigen und die False-Positives widerlegen. Andere korrekte Kandidaten konnten jedoch aufgrund ungenügender Verhaltensmuster nicht bestätigt werden.

Die ungenügende Spezifikation der Verhaltensmuster konnte auf eine in bestimmten Fällen ungenügend ausdrucksstarke Spezifikationssprache zurückgeführt werden. Gleichzeitig wurden jedoch Ideen zu einer möglichen Erweiterung der Spezifikationssprache vorgestellt, mit denen die Einschränkungen aufgehoben werden können. Aus Zeitgründen konnten diese Ideen aber nicht mehr im Rahmen dieser Arbeit umgesetzt werden.

Kapitel 7

Werkzeugunterstützung

In diesem Kapitel wird die technische Umsetzung der in den vorhergehenden Kapiteln vorgestellten Konzepte durch das Werkzeug RECLIPSE behandelt. Zunächst wird im ersten Teil des Kapitels die Einbettung des Werkzeugs in die Entwicklungsumgebung ECLIPSE erläutert. Im zweiten Teil des Kapitels werden die Architektur und die wichtigsten Komponenten der Entwurfsmustererkennung beschrieben, bevor im dritten Teil die Benutzungsschnittstelle mit Hilfe eines Beispiels vorgestellt wird.

7.1 Entwicklungsumgebung

Die im Rahmen dieser Arbeit entwickelte struktur- und verhaltensbasierte Entwurfsmustererkennung baut auf der Entwicklungsumgebung FUJABA [Fuj] auf. FUJABA wird seit 1998 am Fachgebiet Softwaretechnik der Universität Paderborn entwickelt. FUJABA ist eine modellbasierte Entwicklungsumgebung auf Basis von UML, Story Driven Modelling[1] (SDM) und Graphtransformationen, die von Dritten durch Plug-Ins beliebig erweitert werden kann [BGN+04]. Zu den Grundfunktionen gehört eine JAVA-Codegenerierung aus den mit FUJABA spezifizierten Modellen.

In den letzten Jahren fand das Werkzeug ECLIPSE [Ecl] in der Industrie eine immer größere Verbreitung. ECLIPSE stellt in der Grundversion lediglich ein durch Plug-Ins erweiterbares Framework dar. Auf Basis von ECLIPSE können aber beliebige Anwendungen durch unterschiedliche Konfigurationen von Plug-Ins geschaffen werden. Eine weit verbreitete Konfiguration von ECLIPSE ist die JAVA-Entwicklungsumgebung JDT (Java Development Tooling).

[1] Story Driven Modelling ist eine besondere Form der Modellierung von Algorithmen durch UML-Objektdiagramme und Graphtransformationen

Fujaba wurde als Plug-In in Eclipse integriert. Die so entstandene Entwicklungsumgebung aus Eclipse, JDT und Fujaba wird Fujaba4Eclipse genannt. Das Werkzeug Fujaba4Eclipse vereint somit eine modellbasierte Entwicklung mit Java-Codegenerierung und einer Java-Entwicklungsumgebung.

In der vorliegenden Arbeit wurde Fujaba4Eclipse durch die struktur- und verhaltensbasierte Entwurfsmustererkennung zu einem Reverse-Engineering-Werkzeug erweitert. Die bereits existierende strukturbasierte Entwurfsmustererkennung [Wen01, Nie04] ist, wie in Kapitel 3 beschrieben, in dieser Arbeit erweitert und auf Fujaba4Eclipse portiert worden. Die neu entwickelte, verhaltensbasierte Entwurfsmustererkennung ist ebenfalls auf Basis von Fujaba4Eclipse in Form von Eclipse-Plug-Ins realisiert worden. Die Konfiguration aus Fujaba4Eclipse und den Plug-Ins zur struktur- und verhaltensbasierten Entwurfsmustererkennung wird Reclipse genannt.

7.2 Architektur

Die struktur- und verhaltensbasierte Entwurfsmustererkennung ist modular auf Basis verschiedener Komponenten aufgebaut. Abbildung 7.1 zeigt die wichtigsten Komponenten und deren Abhängigkeiten untereinander. Die Komponenten sind als Eclipse-Plug-Ins entwickelt worden. Die im Diagramm genannten Komponenten enthalten nur die Modelle und die Logik. Zu fast jeder der Komponenten gibt es jeweils noch eine zusätzliche Komponente, die ihre Benutzungsschnittstelle enthält. Im Folgenden wird jede der im Diagramm genannten Komponenten kurz erläutert. Details zu den Komponenten, wie Abhängigkeiten, Ein- und Ausgaben sowie Versionen, können im Anhang C.1 nachgeschlagen werden.

- **de.uni_paderborn.fujaba**: Dies ist die Hauptkomponente, in der die integrierte Entwicklungsumgebung Fujaba enthalten ist. Sie stellt unter anderem die Pakete ClassDiagrams (Abbildung 2.1, Seite 18) zur Spezifikation von Klassendiagrammen und Structure zur Repräsentation der Struktur (Abbildung 2.2, Seite 19) zur Verfügung. Allerdings sind diese beiden Pakete in Fujaba aus dem in Abschnitt 2.3.1 genannten Grund identisch.

- **org.reclipse.javaast**: Stellt das Paket JavaAST als Modell des abstrakten Syntaxbaums für Java-Methodenrümpfe zur Verfügung.

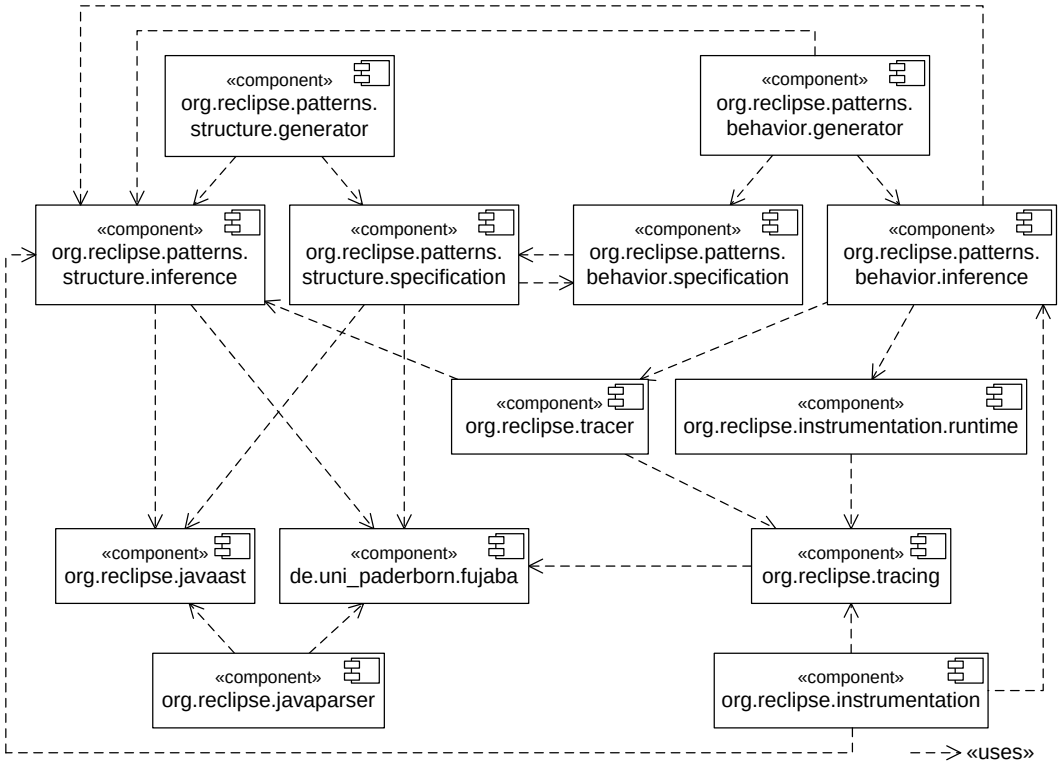

Abbildung 7.1: Die Komponenten der struktur- und verhaltensbasierten Entwurfsmustererkennung

- **org.reclipse.javaparser**: Stellt einen Parser zur Verfügung, der den Java-Quelltext des zu untersuchenden Softwaresystems in eine Struktur auf Basis des Strukturmodells umwandelt.

- **org.reclipse.tracing**: Stellt das Paket Behavior (Abbildung 3.9, Seite 52) zur Repräsentation von Tracegraphen bereit. Stellt außerdem das Paket TraceDefinition zur Spezifikation der zu überwachenden Methoden eines Softwaresystems zur Verfügung.

- **org.reclipse.tracer**: Stellt ein Werkzeug zum Debugging von Java-Programmcode zur Verfügung. Das Werkzeug erzeugt Breakpoints zur Überwachung der Methodenaufrufe während der Ausführung des Programms. Die beobachteten Methodenaufrufe können entweder in eine Datei zur Offline-Analyse gespeichert werden, oder direkt in einer Online-Analyse der Verhaltensanalyse übergeben werden.

- **org.reclipse.instrumentation**: Stellt ein Werkzeug zur Instrumentierung von Java-Bytecode bereit. Das Werkzeug fügt zusätzlichen Programmcode ein, der zur Überwachung von Methodenaufrufen dient.
- **org.reclipse.instrumentation.runtime**: Stellt die Laufzeitumgebung für den instrumentierten Programmcode zur Verfügung. Wird als Bibliothek dem instrumentierten Programmcode hinzugefügt. Die beobachteten Methodenaufrufe können entweder in eine Datei zur Offline-Analyse gespeichert werden, oder direkt in einer Online-Analyse der Verhaltensanalyse übergeben werden.
- **org.reclipse.patterns.structure.specification**: Stellt zur Spezifikation von Strukturmustern das Paket StructuralPatterns (Abbildung 4.7, Seite 68) zur Verfügung.
- **org.reclipse.patterns.structure.inference**: Enthält den Algorithmus zur Strukturanalyse (Abbildung 2.10, Seite 28). Dazu verwendet die Komponente Erkennungsmaschinen für Strukturmuster, um in der Struktur des zu untersuchenden Softwaresystems nach Kandidaten für Entwurfsmusterimplementierungen zu suchen. Sie stellt außerdem das Paket Annotations (Abbildung 5.16, Seite 132) zur Annotation der Kandidaten bereit. Für die Erkennungsmaschinen wird eine Schnittstelle definiert.
- **org.reclipse.patterns.structure.generator**: Stellt einen Algorithmus zur Verfügung, um aus den Strukturmustern die Erkennungsmaschinen zu generieren.
- **org.reclipse.patterns.behavior.specification**: Stellt das Paket BehavioralPatterns (Abbildung 4.5, Seite 64) zur Spezifikation der Verhaltensmuster zur Verfügung.
- **org.reclipse.patterns.behavior.inference**: Enthält den Algorithmus zur Verhaltensanalyse (Abbildung 5.1, Seite 106). Die Komponente verwendet endliche Automaten, um im Tracegraphen nach Verhaltensmustern zu suchen. Sie stellt außerdem die Pakete BehaviorAnalysis (Abbildung 5.2, Seite 111) und Automaton (Abbildung 5.3, Seite 113) zur Verfügung.
- **org.reclipse.patterns.behavior.generator**: Stellt den Algorithmus aus Abschnitt 4.4 zur Verfügung, um aus den Verhaltensmustern endliche Automaten zu generieren. Erzeugt außerdem aus dem Ergebnis der Strukturanalyse die Trace-Definition zur Überwachung des Programms.

7.3 Benutzungsschnittstelle

Im Folgenden wird die Benutzungsschnittstelle von RECLIPSE vorgestellt. Zunächst werden einige allgemeine Elemente der Benutzungsschnittstelle erläutert. Es folgen die Benutzungsschnittstellen zur Spezifikation der Struktur- und der Verhaltensmuster. Anschließend wird der Prozess der struktur- und verhaltensbasierten Entwurfsmustererkennung anhand des Beispiels des Mediaplayers aus Abschnitt 2.3.1 durchgeführt und daran die Benutzungsschnittstelle der Entwurfsmustererkennung präsentiert. Ein detailliertes Handbuch zu RECLIPSE findet sich im Anhang B.

7.3.1 Elemente der Benutzungsschittstelle

Die Oberfläche der Entwicklungsumgebung ECLIPSE ist aus einer Reihe von verschiedenen Sichten und Editoren konfigurierbar, die von Plug-Ins zur Verfügung gestellt werden. Solche Konfigurationen werden in *Perspektiven* zusammengefasst. Die Sichten und Editoren einer Perspektive sind üblicherweise so zusammengestellt, dass mit ihnen eine bestimmte Aufgabe durchgeführt werden kann. Sie lassen sich aber auch beliebig anordnen, verkleinern oder vergrößern, so dass der Benutzer die Oberfläche individuell nach seinen Bedürfnissen anpassen kann.

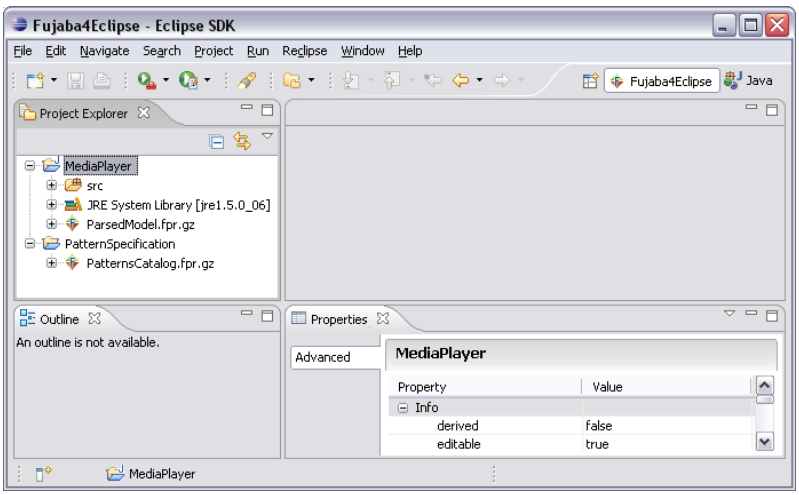

Abbildung 7.2: Die Benutzeroberfläche von RECLIPSE

Abbildung 7.2 zeigt die Perspektive, die von FUJABA4ECLIPSE zur Verfügung gestellt wird. Die Oberfläche ist in verschiedene Bereiche für die Sichten

und Editoren aufgeteilt. Im linken, oberen Teil der Oberfläche ist der *Project Explorer* zu sehen. FUJABA4ECLIPSE ist eine modellbasierte Entwicklungsumgebung, die Modelle in Modelldateien organisiert. Mit dem *Project Explorer* können die Modelle einer Modelldatei durchsucht und zur Ansicht oder zum Bearbeiten ausgewählt werden.

Der rechte, obere Teil der Oberfläche ist dem jeweiligen Editor vorbehalten, der zum Beispiel zum Bearbeiten eines Modells oder JAVA-Quelltextes verwendet wird. Links unten ist die *Outline*-Sicht angeordnet, die eine Übersicht über das aktuell bearbeite Modell oder den Quelltext bietet. Rechts unten ist der *Properties*-Editor zu sehen, mit dem zum Beispiel die Eigenschaften einzelner Modellelemente geändert werden können.

Die Funktionen von RECLIPSE sind in die Perspektive von FUJABA4ECLIPSE integriert und lassen sich über das Menü *Reclipse* oder über Kontext-Menüs des Modells aufrufen.

7.3.2 Spezifikation der Struktur- und Verhaltensmuster

Im Prozess der struktur- und verhaltensbasierten Entwurfsmustererkennung geht der Reverse-Engineer üblicherweise von einem bereits existierenden Katalog von Struktur- und Verhaltensmustern aus. Dieser Katalog wird dann an die Eigenheiten des zu untersuchenden Softwaresystems angepasst.

Strukturmuster

Abbildung 7.3 zeigt die Entwicklungsumgebung RECLIPSE. Im Project Explorer auf der rechten Seite sind die in der aktuell geöffneten Modelldatei vorhandenen Strukturmuster aufgelistet. Eines dieser Strukturmuster, das *State*-Strukturmuster, wird im Editor rechts oben angezeigt und bearbeitet.

Auf der rechten Seite des Editors befindet sich eine *Palette* – hier aus Platzgründen eingeklappt – die eine Reihe von Werkzeugen für den jeweiligen Editor zur Verfügung stellt. Die Palette des Strukturmuster-Editors bietet zum Beispiel Werkzeuge zum Hinzufügen von Strukturmusterobjekten, von Annotationen oder auch von Verbindungen. Mit dem Properties-Editor werden die Eigenschaften des Strukturmusters oder einzelner Elemente des Strukturmusters geändert. Links unten in der Outline-Sicht ist eine verkleinerte Ansicht des Editors zum Überblick abgebildet.

Nach der Bearbeitung werden aus den Strukturmustern die Erkennungsmaschinen für die Strukturanalyse generiert. Aus den Strukturmustern werden Graphtransformationsalgorithmen erzeugt, die auf dem Strukturmodell des zu

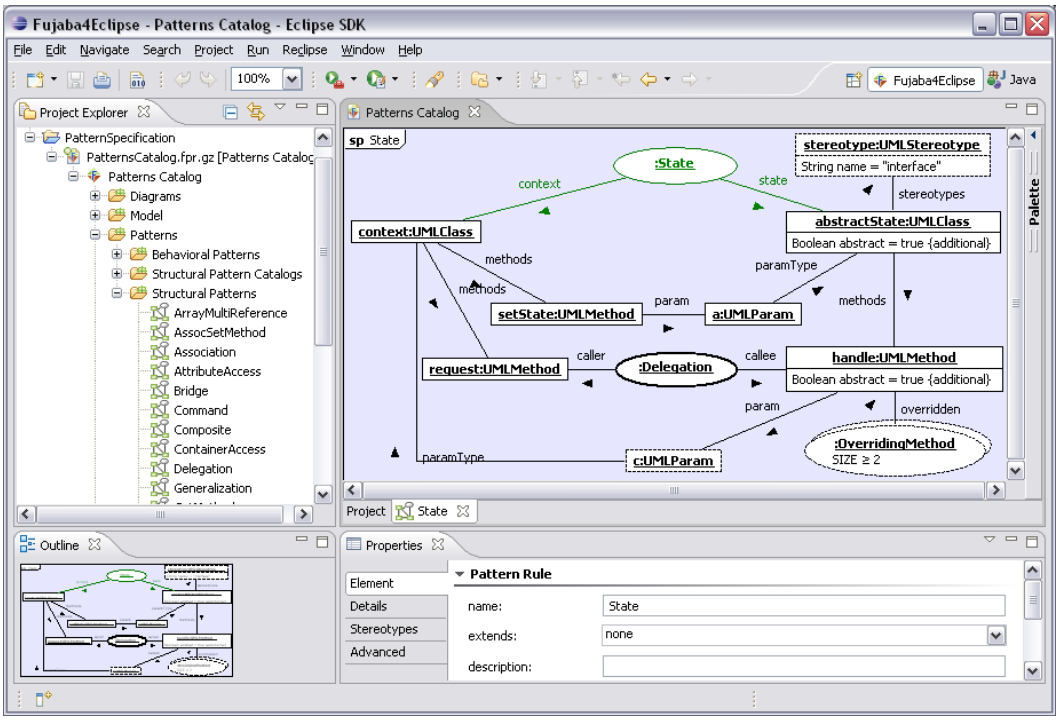

Abbildung 7.3: Die Spezifikation des *State*-Strukturmusters in RECLIPSE

untersuchenden Softwaresystems arbeiten. Die Graphtransformationsalgorithmen werden in JAVA-Klassen übersetzt und stehen dann der Strukturanalyse zur Verfügung. Details zur Generierung der Graphtransformationsalgorithmen finden sich in [Nie04].

Verhaltensmuster

Die Spezifikationen der zu den Strukturmustern gehörigen Verhaltensmuster befinden sich in derselben Modelldatei wie die Strukturmuster. In Abbildung 7.4 ist die Spezifikation des *State*-Verhaltensmusters zu sehen. Über die Palette auf der rechten Seite des Editors lassen sich neue Verhaltensmusterobjekte, Nachrichten oder auch kombinierte Fragmente dem Verhaltensmuster hinzufügen. Die Eigenschaften der Elemente eines Verhaltensmusters werden ebenfalls über den Properties-Editor geändert, der hier aber aus Platzgründen ausgeblendet ist. Die Typnamen der Verhaltensmusterobjekte und die Methodennamen der Nachrichten wurden der Spezifikation des *State*-Strukturmusters aus Abbildung 7.3 entnommen.

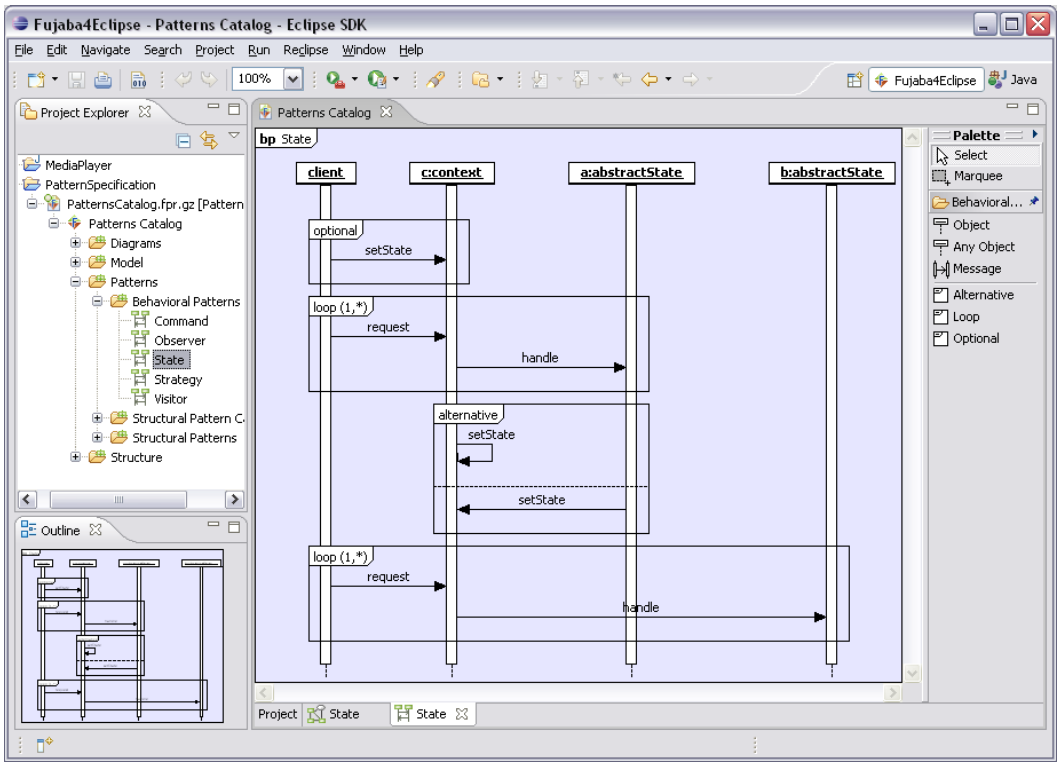

Abbildung 7.4: Die Spezifikation des *State*-Verhaltensmusters in Reclipse

Für die Verhaltensanalyse werden aus den Verhaltensmustern, wie in Abschnitt 4.4 spezifiziert, Automaten generiert, denen das Modell aus Abbildung 5.3 (Seite 113) zugrunde liegt. Die Automaten eines Kataloges werden deskriptiv in einer Datei gespeichert. Die formale Definition des Dateiformats findet sich in der technischen Dokumentation im Anhang C.2.4.

7.3.3 Strukturbasierte Entwurfsmustererkennung

Als Voraussetzung zur strukturbasierten Entwurfsmustererkennung muss zunächst der Quelltext des zu untersuchenden Softwaresystems unter Reclipse in eine Modelldatei importiert werden. Die Struktur des Softwaresystems steht nun den Analysen zu Verfügung. Anschließend wird die strukturbasierte Entwurfsmustererkennung unter Angabe eines Katalogs von Strukturmustern gestartet. In Abbildung 7.5 ist das Ergebnis der Analyse zu sehen.

Der Editor auf der rechten Seite zeigt einen Ausschnitt des Klassendia-

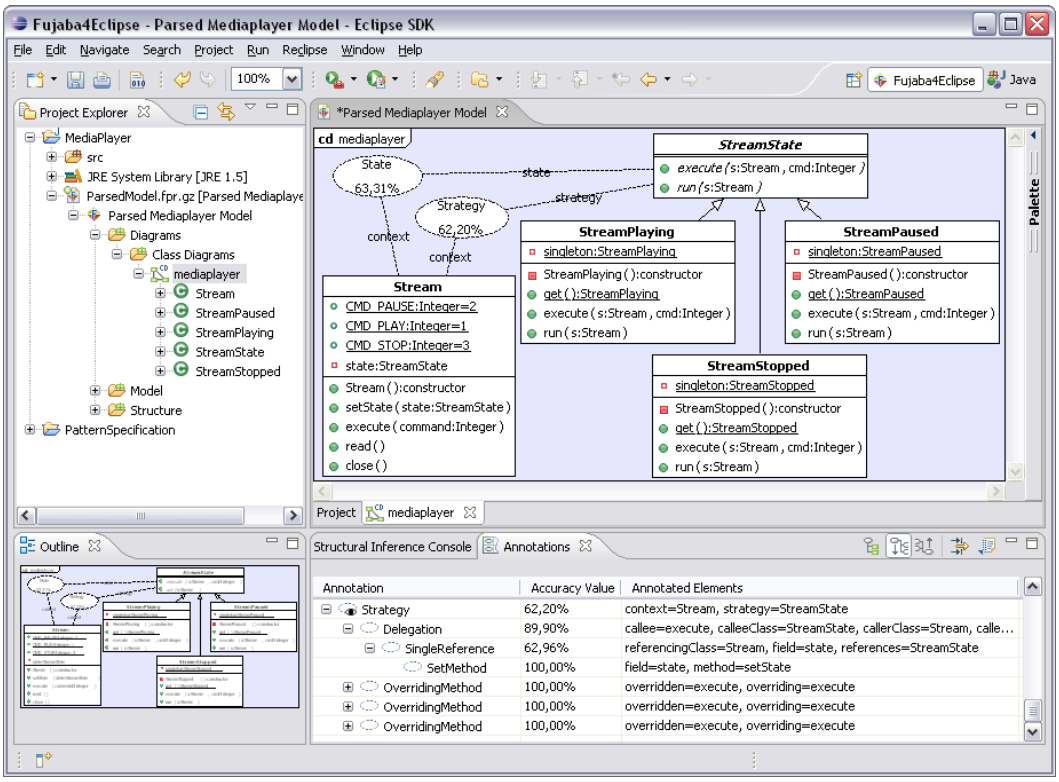

Abbildung 7.5: Das Ergebnis der Strukturanalyse

gramms zum Mediaplayer. Im Klassendiagramm werden außerdem die Annotationen visualisiert, die im Laufe der strukturbasierten Entwurfsmustererkennung erzeugt wurden. Der besseren Übersichtlichkeit wegen sind nur die wichtigsten beiden Annotationen eingeblendet worden. Das Gesamtergebnis der strukturbasierten Entwurfsmustererkennung ist in der Sicht *Annotations* rechts unten einzusehen. Darin sind alle Annotationen mit weiteren Informationen enthalten. Meldungen, die während des strukturbasierten Erkennungsprozesses ausgegeben werden, wie Angaben über Prozessschritte oder identifizierte Strukturmuster, sind in der Sicht *Structural Inference Console* zu finden.

Die Annotationen werden nach der Strukturanalyse zur Vorbereitung der verhaltensbasierten Entwurfsmustererkennung in eine Datei exportiert. Die Spezifikation des Dateiformats ist in Anhang C.2.1 zu finden. Des Weiteren muss eine *Trace-Definition* exportiert werden, die die Informationen enthält, welche Methoden des Strukturmodells zur Laufzeit des Programms überwacht werden müssen. Das Dateiformat ist in Anhang C.2.2 spezifiziert.

7.3.4 Verhaltensbasierte Entwurfsmustererkennung

Ein wesentlicher Bestandteil der verhaltensbasierten Entwurfsmustererkennung ist die Gewinnung der Traces. Wie in Abschnitt 5.2 erläutert, stehen dazu zwei verschiedene Verfahren zur Verfügung, das Debugging und die Instrumentierung. Im Folgenden wird die Realisierung der beiden Verfahren in Reclipse vorgestellt. Des Weiteren wird erläutert, wie die verhaltensbasierte Entwurfsmustererkennung im Offline-Modus ausgeführt wird. Die Ergebnisse werden anschließend in einer Sicht visualisiert und können vom Reverse-Engineer untersucht werden.

Debugging

Zum Debugging wurde der Reclipse Tracer entwickelt [WME04, MW05]. Das zu untersuchende Softwaresystem wird durch den Reclipse Tracer gestartet und im Hintergrund durch den Debugger überwacht. Das zu untersuchende Programm wird wie gewohnt im Vordergrund ausgeführt. Die Trace-Definition bestimmt, welche Methoden während der Ausführung überwacht werden.

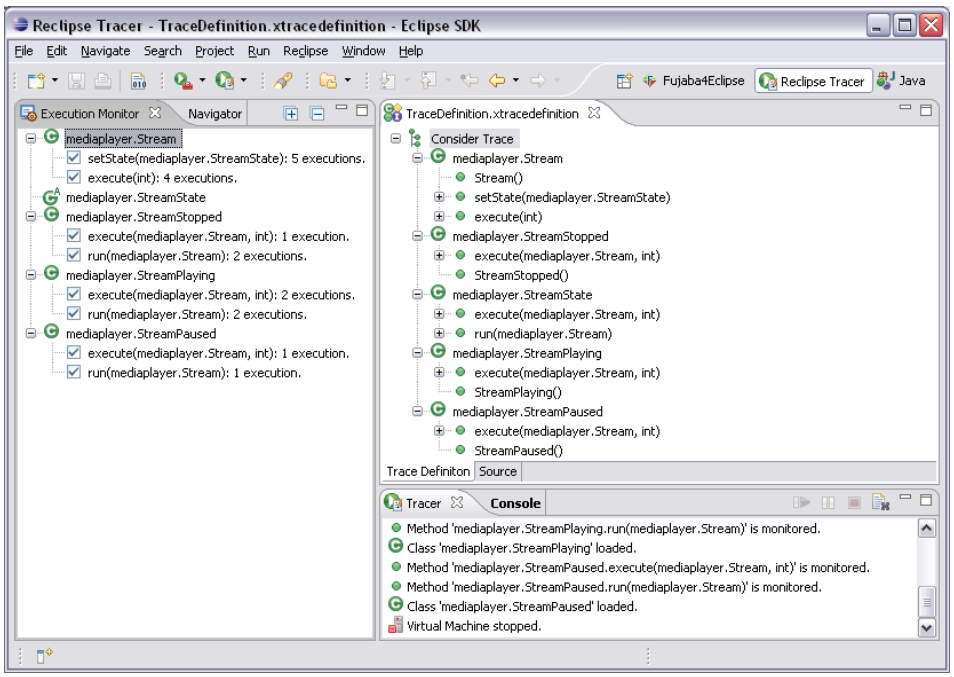

Abbildung 7.6: Der Reclipse Tracer nach dem Debuggen des Mediaplayers

Abbildung 7.6 zeigt den Reclipse Tracer nach dem Debuggen des Mediaplayers. Der Reclipse Tracer stellt eine weitere Perspektive in Eclipse zur Verfügung, in der die Trace-Definition angesehen und geändert werden kann und in der die Ausgaben des Reclipse Tracers angezeigt werden. Im Editor auf der rechten Seite wird die Trace-Definition als ein Baum dargestellt, in dem die zu überwachenden Methoden jeweils unterhalb ihrer Klasse aufgelistet sind. Rechts unten ist die Sicht *Tracer* geöffnet, die Informationen über überwachte Klassen, Methoden und den Zustand des überwachten Programms während des Debuggens ausgibt.

Auf der linken Seite zeigt die Sicht *Execution Monitor* an, welche der zu überwachenden Methoden während des Debuggens bereits ausgeführt wurden und gegebenenfalls, wie häufig sie ausgeführt wurden. Wird das zu untersuchende Programm durch den Reverse-Engineer ausgeführt, kann er hier ablesen, ob genügend Daten für die Verhaltensanalyse gesammelt wurden.

Der Reclipse Tracer kann den Trace direkt an die verhaltensbasierte Entwurfsmustererkennung weiter reichen, oder in eine Datei zur späteren Analyse protokollieren. Auch eine gleichzeitige Analyse und Protokollierung ist möglich. Das Dateiformat des aufgezeichneten Traces ist in Anhang C.2.3 spezifiziert.

Instrumentierung

Zur Instrumentierung des zu untersuchenden Softwaresystems steht in Reclipse ein so genannter *Wizard* zur Verfügung. Ein Wizard besteht aus einer Reihe von Dialogen, die nacheinander verschiedene Eingaben vom Benutzer abfragen und anschließend eine Funktion auf diesen Eingabedaten ausführen. Im Falle der Instrumentierung fragt der Wizard nach dem übersetzten Programmcode des Softwaresystems, der Trace-Definition sowie einigen weiteren Informationen und erzeugt daraus den instrumentierten Programmcode.

In Abbildung 7.7 sind zwei der Dialoge des Instrumentierung-Wizards zu sehen. Wie beim Debuggen kann der Reverse-Engineer auswählen, ob der durch den instrumentierten Programmcode gewonnene Trace direkt an die verhaltensbasierte Entwurfsmustererkennung weiter gegeben und in eine Datei protokolliert werden soll. Die Instrumentierung fügt dann zusätzlichen Programmcode in das zu untersuchende Softwaresystem ein. Zusätzlich werden weitere für die Protokollierung und die verhaltensbasierte Entwurfsmustererkennung erforderlichen Bibliotheken dem zu untersuchenden Softwaresystem hinzugefügt.

Die Ausführung des instrumentierten Programms erfolgt wie gewohnt. Während der Ausführung werden je nach Konfiguration der Instrumentierung

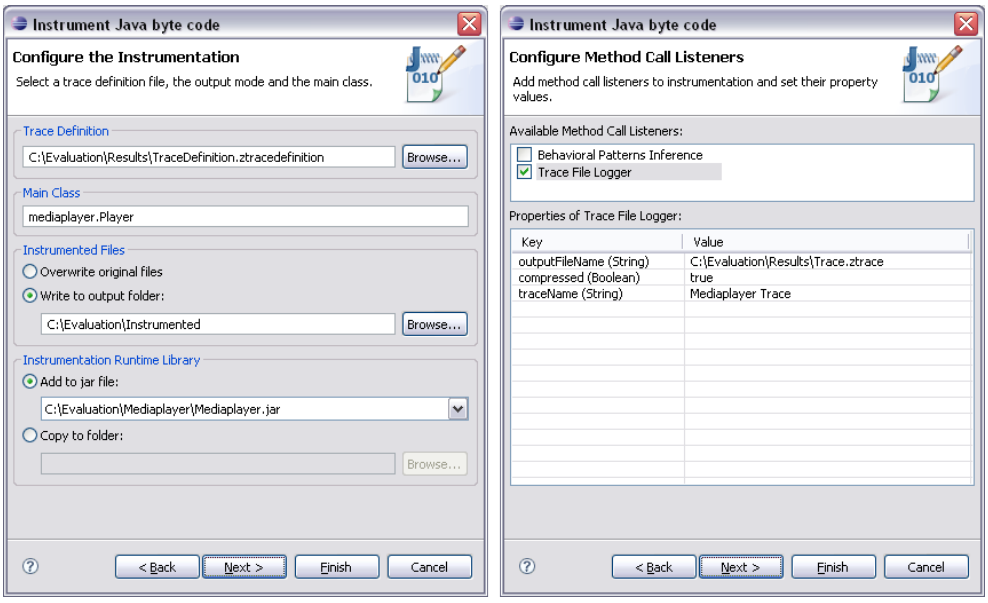

Abbildung 7.7: Der Instrumentierungs-Wizard

die beobachteten Methodenaufrufe in eine Trace-Datei protokolliert und durch die verhaltensbasierte Entwurfsmustererkennung verarbeitet.

Verhaltenserkennung

Die verhaltensbasierte Entwurfsmustererkennung wird entweder, wie in den vorherigen Abschnitten beschrieben, in einem Online-Modus während der Ausführung des zu untersuchenden Softwaresystems, oder im Offline-Modus nach der Ausführung durchgeführt.

In Abbildung 7.8 ist der Dialog zum Starten der verhaltensbasierten Entwurfsmustererkennung im Offline-Modus zu sehen. Der Reverse-Engineer muss als Eingabe die Datei angeben, die die Annotationen der Strukturanalyse enthält, also die Kandidaten. Des Weiteren wird der durch Debugging oder Instrumentierung aufgezeichnete Trace, sowie der Katalog mit den Verhaltensmustern benötigt. Das Ergebnis der verhaltensbasierten Entwurfsmustererkennung wird in einer Datei festgehalten. Das Dateiformat ist in Anhang C.2.5 spezifiziert.

Das Ergebnis wird durch die Sicht *Behavioral Analysis Result* visualisiert. Abbildung 7.9 zeigt diese Sicht mit dem Ergebnis der verhaltensbasierten Entwurfsmustererkennung zum Mediaplayer-Beispiel. Am oberen Rand der Sicht

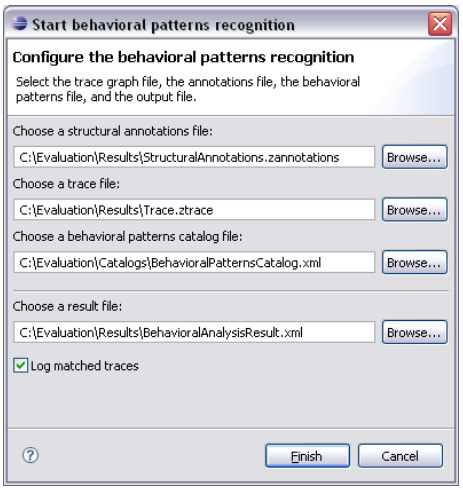

Abbildung 7.8: Starten der Verhaltensanalyse im Offline-Modus

kann ein Kandidat aus der Menge aller Kandidaten ausgewählt werden. In diesem Beispiel wurde der *State*-Kandidat des Mediaplayers gewählt. In der linken oberen Hälfte wird das Ergebnis der strukturbasierten Entwurfsmustererkennung angezeigt. In einer Tabelle ist die Abbildung der Verhaltensmusterobjekte des *State*-Verhaltensmusters auf die Elemente der Struktur angegeben. In der Mitte der Sicht ist eine Zusammenfassung der Ergebnisse der verhaltensbasierten Entwurfsmustererkennung zu diesem Kandidaten zu sehen. Es wird die Anzahl der insgesamt getriggerten Traces, der daraus akzeptierten, verworfenen und nicht akzeptierten Traces angegeben. Ferner wird die Anzahl der verworfenen und nicht akzeptierten Traces genannt, die einen akzeptierten Subtrace enthalten, sowie die durchschnittliche Länge der akzeptierten Traces.

In diesem Fall wurde insgesamt acht Mal die Überprüfung eines Traces durch einen Automaten ausgelöst. Ein Trace wurde akzeptiert, fünf verworfen und zwei weitere nicht akzeptiert. Allerdings wurden in zwei der verworfenen und nicht akzeptierten Traces Subtraces beobachtet, die akzeptiert wurden. Die durchschnittliche Länge der akzeptierten Traces und Subtraces betrug fünf Methodenaufrufe.

Aus den beobachteten Traces zu einem Kandidaten kann auf der rechten Seite der Sicht ein Trace ausgewählt werden, der genauer untersucht werden kann. Es wird zu dem ausgewählten Trace angegeben, ob er akzeptiert, verworfen oder nicht akzeptiert wurde und ob ein Subtrace dieses Traces akzeptiert wurde. Außerdem wird die Länge des akzeptierten Traces beziehungsweise Sub-

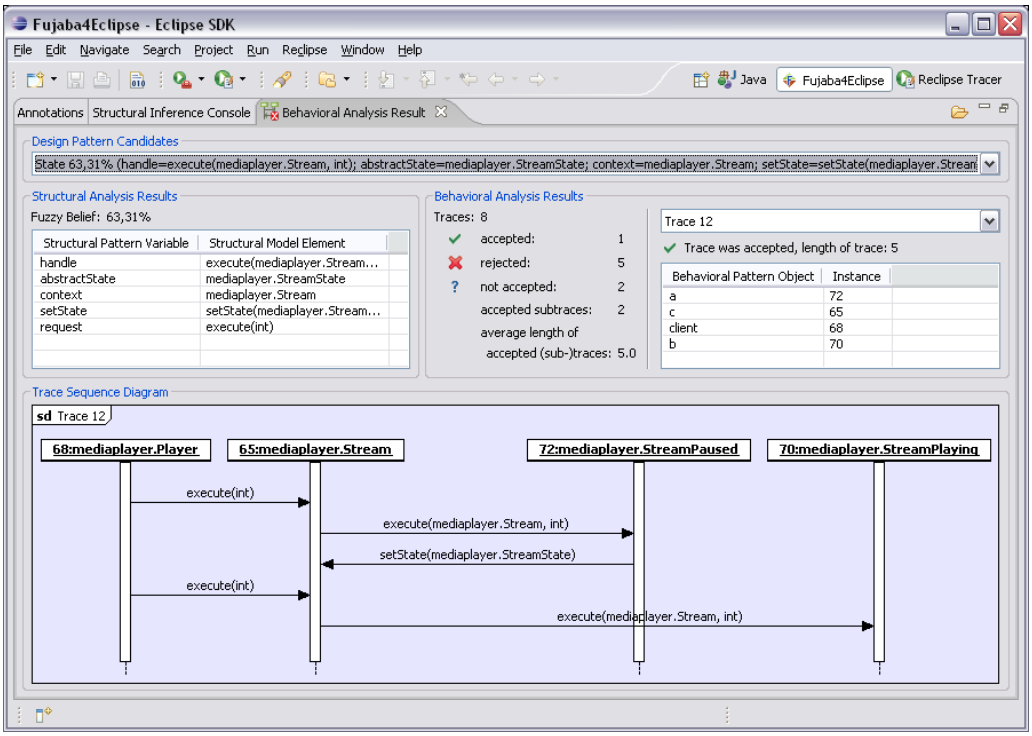

Abbildung 7.9: Das Ergebnis der Verhaltensanalyse

traces genannt. Die Bindung der Verhaltensmusterobjekte an die Instanzen zur Laufzeit des untersuchten Programms wird in einer Tabelle angegeben.

In der unteren Hälfte wird der ausgewählte Trace als Sequenzdiagramm dargestellt. Das Sequenzdiagramm enthält die vom Automaten konsumierten Methodenaufrufe. Methodenaufrufe, die vom Automaten ignoriert wurden, werden nicht dargestellt.

Analyse der Ergebnisse

In Abbildung 7.10 ist ein verworfener Trace ausgewählt worden. Die Visualisierung des Traces hilft zum Beispiel festzustellen, warum er verworfen wurde. In diesem Fall besteht der Trace nur aus zwei Methodenaufrufen. Der erste Methodenaufruf löste die Überprüfung des Traces durch den Automaten für das *State*-Verhaltensmuster (Abbildung 4.1, Seite 59) aus. Der zugehörige Trigger aus dem Verhaltensmuster ist die Nachricht client→(c:context).setState(). Diese Nachricht geht vom untypisierten Verhaltensmusterobjekt client aus. Hier

wurde die Instanz 72 vom Typ StreamPlaying an client gebunden, wie der Tabelle in der Mitte rechts zu entnehmen ist. Nach dem Verhaltensmuster muss nun die nächste Nachricht, der request-Aufruf auf dem Verhaltensmusterobjekt c:context von client ausgeführt werden. Allerdings wird hier der request-Aufruf durch eine andere Instanz ausgeführt, was zum Verwerfen des Traces führt.

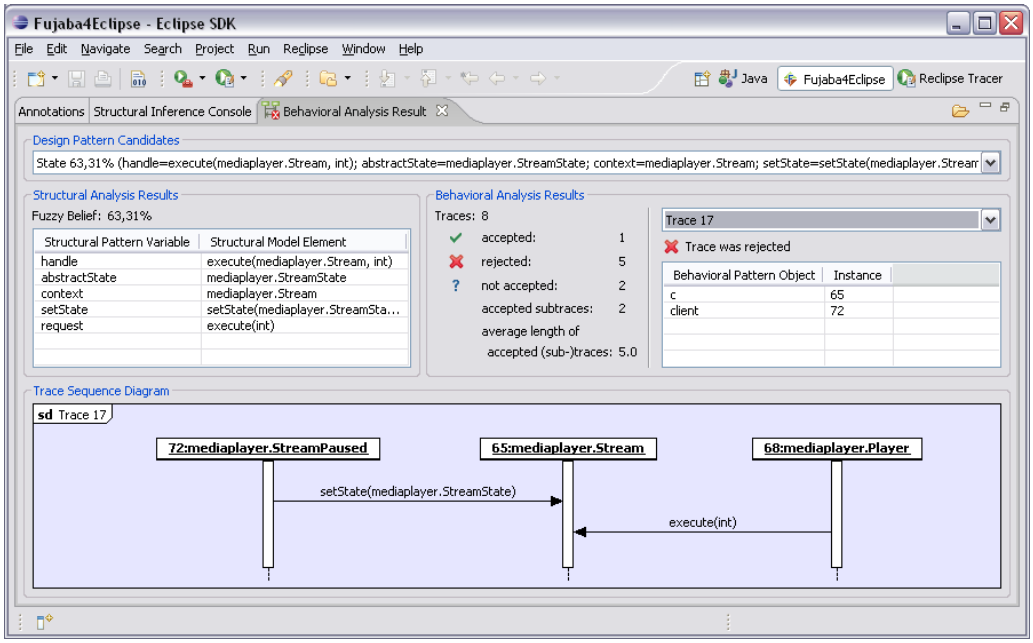

Abbildung 7.10: Ein verworfener Trace

Wenn man den Trace jedoch genauer betrachtet, stellt man fest, dass er das *State*-Verhaltensmuster nicht verletzt. Es ist ein Subtrace des akzeptierten Traces aus Abbildung 7.9. Der Methodenaufruf setState(mediaplayer.StreamState) wird aber leider durch den Automaten fälschlicherweise als die triggernde Nachricht client→(c:context).setState() des *State*-Verhaltensmusters interpretiert.

Dieses False-Positive ließe sich verhindern, wenn das *State*-Verhaltensmuster optimiert wird. Die erste, optionale Nachricht client→(c:context).setState() ist nicht entscheidend zum Erkennen eines *State*-Entwurfsmusters. Entscheidend ist vielmehr der Zustandswechsel, der durch den aktuellen Zustand oder dem Kontext durchgeführt werden muss. Daher kann die erste, optionale Nachricht aus dem *State*-Verhaltensmuster entfernt werden. Das führt dazu, dass die Überprüfung des Verhaltensmusters nur noch durch die eindeutige Nachricht

client→(c:context).request() getriggert wird und so die Zahl der False-Positives reduziert wird.

7.4 Zusammenfassung

Die struktur- und verhaltensbasierte Entwurfsmustererkennung ist vollständig implementiert worden. Das entstandene Werkzeug RECLIPSE wurde in die weit verbreitete Entwicklungsumgebung ECLIPSE integriert. Der gesamte Prozess der struktur- und verhaltensbasierten Entwurfsmustererkennung wird von RECLIPSE unterstützt, angefangen bei der graphischen Spezifikation der Struktur- und Verhaltensmuster, über die Struktur- und Verhaltensanalyse, dem Debuggen und der Instrumentierung des Programmcodes, bis hin zur Visualisierung der Ergebnisse sowohl der Struktur- als auch der Verhaltensanalyse. Im Anhang B ist ein Handbuch zur Benutzung von RECLIPSE zu finden, das im Detail alle notwendigen Schritte erklärt, um den Prozess in RECLIPSE vollständig durchzuführen.

Kapitel 8

Verwandte Arbeiten

In diesem Kapitel wird der aktuelle Stand der Forschung in den Bereichen der Entwurfsmustererkennung und der dynamischen Verhaltensanalysen betrachtet. Aus dem Bereich der Entwurfsmustererkennung werden im ersten Teil des Kapitels zunächst Arbeiten vorgestellt, die auf rein statischen Analysen der Struktur eines Softwaresystems aufbauen. Die Arbeiten werden in Bezug auf die in Abschnitt 2.2 genannten Anforderungen beurteilt. Der zweite Teil des Kapitels behandelt verschiedene Arbeiten, die explizit das Verhalten von Softwaresystemen zur Laufzeit analysieren.

Kombinationen aus Struktur- und Verhaltensanalysen werden seit einiger Zeit im Reverse-Engineering eingesetzt. Im dritten Teil werden Arbeiten diskutiert, die diese Kombinationen anwenden. Im Besonderen werden solche Arbeiten diskutiert und mit der vorliegenden Arbeit verglichen, die aus dem Bereich der Entwurfsmustererkennung stammen.

8.1 Strukturbasierte Entwurfsmustererkennung

Frühe Ansätze zur Suche nach allgemeinen Softwaremustern waren das System Pat von Mehdi Harandi und Jim Ning [HN90] aus dem Jahre 1990 und die Arbeit von Linda Wills aus dem Jahre 1992 [Wil92]. Nach dem Erscheinen des Buches von Gamma et al. [GHJV95] im Jahre 1995 wurde dann eine Reihe von Werkzeugen entwickelt, die speziell Entwurfsmusterimplementierungen (semi-)automatisch im Quelltext erkennen. Bis auf einige wenige Ausnahmen[1] basieren jedoch sehr viele dieser Werkzeuge auf einer rein statischen Analyse[2], bei der der Quelltext ausschließlich nach strukturellen Eigenschaften der

[1] siehe [Bro96, GZ05, HHHL03]

[2] siehe [KP96, AFC98, SK98, SG98, Wuy98, KSRP99, TA99, BP00, KB00, ACGJ01, AG01, SS03, PSRN04, TCHS05, KGH06, SO06]

Entwurfsmuster untersucht wird, ohne die Software auszuführen.

Das Verhalten eines Softwaresystems, im Wesentlichen bestimmt durch Methodenaufrufe, lässt sich jedoch durch statische Analysen nur bedingt untersuchen. Statische Analysen erkennen zwar potentielle Methodenaufrufe, ob jedoch diese Methodenaufrufe zur Laufzeit tatsächlich ausgeführt werden, ist nicht sicher festzustellen. Objektorientierte Programmiersprachen mit Polymorphie und dynamischer Methodenbindung verschärfen dieses Problem sogar noch, da die konkreten, aufzurufenden Methoden erst zur Laufzeit festgelegt werden. Die im Folgenden aufgeführten Verfahren zur strukturbasierten Entwurfsmustererkennung lassen sich grob in zwei Kategorien einteilen: Verfahren, die potentielle Methodenaufrufe untersuchen, und solche, die dies nicht tun.

Entwurfsmustererkennung ohne Analyse potentieller Methodenaufrufe

Ohne Analyse potentieller Methodenaufrufe lassen sich Konstrukte wie *Delegation* (siehe dazu auch [GHJV95]) nicht erkennen. Bei einer Delegation wird ein Methodenaufruf von einem Objekt an ein anderes weitergegeben, die Aufgabe wird von einem Objekt an ein anderes Objekt delegiert. Dieses Konstrukt ist in vielen Entwurfsmustern wie zum Beispiel *State*, *Strategy*, *Visitor* oder auch *Mediator* ein zentraler Bestandteil. Eine präzise Erkennung dieser und anderer Entwurfsmuster ist ohne die Analyse potentieller Methodenaufrufe daher kaum möglich.

Zu der Kategorie der Entwurfsmustererkennungen ohne Analyse potentieller Methodenaufrufe zählen die Verfahren von Christian Krämer und Lutz Prechelt [KP96], von Giulio Antoniol et al. [AFC98], von Federico Bergenti und Agostino Poggi [BP00], von Hyoseob Kim und Cornelia Boldyreff [KB00], von Ilka Philippow et al. [PSRN04], sowie von Nija Shi und Ronald A. Olsson [SO06].

Entwurfsmustererkennung mit Analyse potentieller Methodenaufrufe

Zu den Arbeiten, bei denen potentielle Methodenaufrufe bei der Entwurfsmustererkennung berücksichtigt werden, gehören das Projekt Spool von Rudolf Keller und Reinhard Schauer [SK98, KSRP99], das Projekt SPQR von Jason Smith und David Stotts [SS03], sowie die Verfahren von Roel Wuyts [Wuy98], von Jochen Seemann und Jürgen Wolff von Gudenberg [SG98], von Paolo Tonella und Giulio Antoniol [TA99], von Hervé Albin-Amiot und Yann-Gaël Guéhéneuc [AG01, ACGJ01] und von Nikolas Tsantalis et al. [TCHS05].

Neben der fehlenden Präzision bei der Analyse potentieller Methodenaufrufe und der damit verbundenen hohen Zahl von False-Positives erfüllen die meisten dieser Arbeiten weitere der an eine automatische Entwurfsmustererkennung gestellten Anforderungen nicht. So ist die leichte Anpassbarkeit der Entwurfsmusterspezifikationen bei den Arbeiten [SK98, KSRP99], [TA99] und [TCHS05] nicht gegeben. Des Weiteren wurden zu den Verfahren [Wuy98], [AG01, ACGJ01], [SS03] und [TCHS05] keine Ergebnisse einer Anwendung auf praxisnahe Softwaresysteme veröffentlicht, wodurch sich keine Aussagen über die Skalierbarkeit der Verfahren treffen lassen. Bei dem Verfahren von Hervé Albin-Amiot und Yann-Gaël Guéhéneuc deutet eine neuere Veröffentlichung [KGH06] auf eine exponentielle Laufzeit hin.

Allen genannten Verfahren ist gemeinsam, dass die Ergebnisse der Entwurfsmustererkennung nicht bewertet werden. Als Ergebnis der Analyse liefern die Verfahren potentielle Entwurfsmusterimplementierungen, ohne Aussagen über die Sicherheit dieser Kandidaten zu treffen. Unter den Kandidaten befinden sich jedoch auch viele fälschlicherweise erkannte Entwurfsmusterimplementierungen. Bei der Sichtung und Einschätzung der Kandidaten wird der Reverse-Engineer also nicht unterstützt.

8.2 Dynamische Analysen zur Verhaltenserkennung

In den folgenden Ansätzen wird das Verhalten von Softwaresystemen ausschließlich durch dynamische Analysen untersucht. Sie werden jedoch für verschiedene andere Zwecke als zur Entwurfsmustererkennung eingesetzt. Trotzdem sind die verwendeten Techniken vergleichbar mit denen, die in dieser Arbeit für die verhaltensbasierte Entwurfsmustererkennung eingesetzt wurden.

Thomas Kunz und Michiel Seuren zeichnen den Nachrichtenaustausch zwischen Prozessen auf, um nach Kommunikationsmustern in verteilten Anwendungen zu suchen [KS97]. In verteilten Anwendungen kommunizieren mehrere Prozesse miteinander, indem sie durch Nachrichten Informationen austauschen, wobei relativ komplexe Kommunikationsmuster immer wieder wiederholt werden. Ein typisches Kommunikationsmuster ist zum Beispiel das Versenden einer Nachricht eines Prozesses an alle anderen Prozesse. Die Ausführung von verteilten Prozessen wird häufig durch so genannte *Process-Time*-Diagramme dargestellt, die aus parallelen Sequenzen von atomaren Ereignissen wie dem Senden oder Empfangen einer Nachricht bestehen. Kunz

und Seuren repräsentieren solche Sequenzen als Zeichenketten, in denen atomare Ereignisse als einzelnes Zeichen kodiert werden. Kommunikationsmuster werden dagegen als reguläre Ausdrücke über diesen Zeichenketten kodiert und durch Anwendung der regulären Ausdrücke erkannt. In Visualisierungen können so zum Beispiel komplexe Sequenzen von Nachrichten zu abstrakteren Einheiten zusammengefasst werden.

Tamar Richner und Stéphane Ducasse verwenden dynamische Analysen zur Identifikation von Kollaborationen [RD02]. Beim kollaborationsbasierten Entwurf von objektorientierten Softwaresystemen wird das Verhalten der Anwendung durch verschiedene Kollaborationen von Objekten beschrieben. Die Rückgewinnung solcher Kollaborationen hilft dabei, die Funktionsweise des Softwaresystems zu verstehen. In dem Verfahren werden Methoden instrumentiert, um von Methodenaufrufen zur Laufzeit die aufrufende und die aufgerufene Instanz und ihre Typen sowie den Namen der Methode aufzuzeichnen. Ähnliche Sequenzen von Methodenaufrufen werden mit Hilfe von Kollaborationsmustern gesucht. Kollaborationsmuster werden durch Sequenzen oder Mengen von Methodenaufrufen und die beteiligten Instanzen und Typen definiert. Der *Collaboration Browser* dient zur Visualisierung der aufgezeichneten Methodenaufrufe und der identifizierten Kollaborationen. Lei Wu et al. beschreiben ein ähnliches Verfahren [WSV04], das jedoch in nicht-objektorientierten Sprachen geschriebene Softwaresysteme analysiert.

Das von Giovanni Vigna und Richard Kemmerer entwickelte Werkzeug NetSTAT erkennt mit Hilfe von Zustandsautomaten Angriffe auf Netzwerke [VK98]. Verschiedene Angriffsszenarien werden durch Sequenzen von Nachrichten auf einem Netzwerk beschrieben. Die Transitionen eines Zustandsautomaten kodieren, welche Nachrichten Zustandsübergänge auslösen. Endet ein Automat in einem akzeptierenden Endzustand, so wird ein Angriff auf das Netzwerk festgestellt. Mit Hilfe so genannter Proben an bestimmten Knoten des Netzwerks werden die Nachrichten dezentral überwacht. Statische Informationen über das Netzwerk werden dazu verwendet, die Proben im Netzwerk zu verteilen. Jede der Proben enthält einen Teil der Angriffsszenarien. Durch einen Filter melden die Proben nur die relevanten Nachrichten an die Automaten, die sie dann verarbeiten.

Die vorgestellten Verfahren nutzen Techniken, die sich in der Verhaltenserkennung bewährt haben. So wird das Verhalten mit Hilfe von Sequenzen von Methodenaufrufen oder Nachrichten beschrieben. Methodenaufrufe werden durch Instrumentierung überwacht. Statische Informationen dienen dazu, die Überwachung auf relevante Nachrichten einzuschränken. Die Erkennung des Verhaltens geschieht durch reguläre Ausdrücke beziehungsweise durch da-

zu äquivalente, endliche Automaten. Diese Techniken wurden in der vorliegenden Arbeit aufgegriffen und zur verhaltensbasierten Entwurfsmustererkennung eingesetzt.

8.3 Kombinierte statische und dynamische Analysen

Statische und dynamische Analysen wurden lange Zeit in unterschiedlichen, voneinander unabhängigen Forschungsgebieten entwickelt. Dadurch wurden meist ausschließlich entweder statische oder dynamische Analysen genutzt. Die jeweils andere Analysetechnik wurde sogar häufig für den aktuellen Anwendungsbereich als unpassend dargestellt. Michael Ernst plädiert deshalb in seinem Artikel *„Static and dynamic analysis: synergy and duality“* [Ern03] dafür, die Vorteile beider Analysetechniken zu kombinieren. Die Unzulänglichkeiten einer Analysetechnik sollen durch die jeweils andere Analysetechnik ausgeglichen werden, um insgesamt bessere Analyseergebnisse zu produzieren.

Es gab in den letzten Jahren verschiedene Ansätze im Reverse-Engineering, die Einschränkungen der statischen Analysen durch Ergänzung um dynamische Analysen oder umgekehrt aufzuheben. Im Folgenden werden zunächst exemplarisch einige Verfahren vorgestellt, die zwar im Reverse-Engineering angewendet werden, aber nicht aus dem speziellen Gebiet der Entwurfsmustererkennung stammen. Daraufhin folgen Verfahren, die statische und dynamische Analysen speziell zur Entwurfsmustererkennung kombinieren.

8.3.1 Ausgewählte Verfahren im Reverse-Engineering

Tamar Richner und Stéphane Ducasse stellen in [RD99] ein Verfahren vor, das abstrakte Sichten auf objektorientierte Softwaresysteme erzeugt. Dabei verwenden sie sowohl statische als auch dynamische Analysen. Durch die statische Analyse werden strukturelle Informationen über das Softwaresystem gewonnen. Die dynamische Analyse zeichnet Methodenaufrufe zur Laufzeit auf. Diese Informationen werden als Prolog-Fakten repräsentiert.

Zur Analyse werden Prolog-Regeln zur Verfügung gestellt, die Fakten aus der statischen und der dynamischen Analyse kombinieren. Durch die Regeln können abstrakte Sichten auf das Softwaresystem erzeugt werden. Beispiele für solche abstrakten Sichten sind Graphen, die Methodenaufrufe zwischen Klassen oder Komponenten darstellen, oder auch Graphen, die die Erzeugung von Objekten durch Klassen oder Komponenten visualisieren.

Das Verfahren dient nur zur Visualisierung. Allerdings könnte die Datenbasis mit einer Entwurfsmustererkennung kombiniert werden, die PROLOG-Regeln verwendet. Dazu wären zum Beispiel die Ansätze [KP96], [Wuy98] oder auch [BP00] geeignet. Die Erstellung und insbesondere auch Wartung der PROLOG-Regeln ist allerdings sehr mühselig, besonders wenn es sich um große Regelsätze handelt, bei denen der Reverse-Engineer schnell den Überblick verliert.

Im Verfahren von Tarja Systä wird die statische Analyse durch eine dynamische Analyse beeinflusst und umgekehrt [Sys99a, Sys99b]. Die statische Analyse erzeugt aus dem JAVA-Quelltext eines Softwaresystems Abhängigkeitsgraphen, die zum Beispiel Vererbungen zwischen Klassen, potentielle Aufrufe zwischen Methoden oder auch Lese- und Schreibzugriffe auf Attribute beschreiben. Visualisiert werden die Graphen durch das Werkzeug RIGI [MWT95, Rig]. Die dynamische Analyse erzeugt so genannte *Szenariodiagramme*, die sehr ähnlich zu Sequenzdiagrammen sind.

Auf Basis des Abhängigkeitsgraphen der statischen Analyse werden die in der dynamischen Analyse zu untersuchenden Teile des Softwaresystems eingeschränkt. Dadurch kann die in der dynamischen Analyse anfallende Datenmenge drastisch reduziert werden. Die Ergebnisse der dynamischen Analyse werden wiederum zur Verfeinerung des Abhängigkeitsgraphen eingesetzt. Die Teile des Softwaresystems, die nicht ausgeführt wurden, können aus dem Abhängigkeitsgraphen ausgeblendet werden.

Eine zu Tarja Systäs Ansatz ähnliche Arbeit ist von Claudio Riva und Jordi Vidal Rodriguez vorgestellt worden [RR02]. Riva und Rodriguez nutzen ebenfalls das Werkzeug RIGI zur Visualisierung von Komponenten als Ergebnis der statischen Analyse. Die Daten der dynamischen Analyse werden durch Instrumentierung des Softwaresystems gewonnen und als Message Sequence Charts (MSCs) [Int99] dargestellt. Zur besseren Übersichtlichkeit können Objekte oder Methodenaufrufe in den MSCs zusammengefasst werden.

Alle drei vorgestellten Verfahren beschränken sich auf die Erzeugung von abstrakten Sichten auf das zu untersuchende Softwaresystem. Der Reverse-Engineer kann sich dadurch leichter einen Überblick über die Architektur oder auch über die Abhängigkeiten eines Softwaresystems schaffen. Details werden durch die Abstraktion aber ausgeblendet. Die Erkennung von Entwurfsmusterimplementierungen hat dagegen weniger einen Überblick über die Software zum Ziel, als vielmehr die Erkennung von Architekturdetails, die dem Verständnis einzelner Teile der Software dienen.

Thomas Eisenbarth, Rainer Koschke und Daniel Simon kombinieren dynamische und statische Analysen zur Identifizierung von Softwarekomponenten, die bestimmte Eigenschaften eines Softwaresystems implementieren [EKS01]. Die

dynamische Analyse wird dazu genutzt, den Suchraum für die statische Analyse zu reduzieren. Zunächst werden Szenarien ausgesucht, die die zu lokalisierenden Eigenschaften des Softwaresystems ausführen. Anschließend wird eine Konzeptanalyse dazu verwendet, um einen Zusammenhang zwischen Szenarien und Softwarekomponenten herzustellen. In der folgenden statischen Analyse werden dann die Softwarekomponenten durch Slicing und manuelle Inspektion weiterverarbeitet.

Dieser Ansatz hilft dem Reverse-Engineer bei der Identifizierung von Komponenten. Er erfährt, auf welche Komponenten die Funktionen eines Softwaresystems verteilt sind, nicht aber, wie sie im Detail aufgebaut sind und wie sie funktionieren. Der Ansatz kann allerdings helfen, die relevanten Teile eines zu untersuchenden Softwaresystems zu identifizieren und sie zur Weiterverarbeitung zum Beispiel einer Entwurfsmustererkennung zuzuführen. So kann die Entwurfsmustererkennung auf die relevanten Komponenten eingeschränkt werden und damit viel Zeit eingespart werden.

Paolo Tonella und Alessandra Potrich erzeugen sowohl aus einer statischen und einer dynamischen Analyse Objektdiagramme [TP02]. Die durch statische Analyse erzeugten Objektdiagramme beschreiben einen *Objektfluss*. Das Objektflussdiagramm enthält zum Beispiel Informationen, an welchen Stellen im Quelltext Objekte erzeugt und diese den Feldern anderer Objekte zugewiesen werden. Aus der dynamischen Analyse werden Objektdiagramme hergestellt, die Objektstrukturen beschreiben. Der Reverse-Engineer kann nun beide Arten von Diagrammen miteinander vergleichen und Rückschlüsse auf das untersuchte Softwaresystem ziehen.

Ein statischer Methodenaufrufgraph beschreibt potentielle Aufrufe zwischen den Methoden eines Softwaresystems. Ein solcher Graph kann durch statische Analysen erzeugt werden, das Ergebnis beruht dann allerdings auf einer konservativen Schätzung. Atanas Rountev, Scott Kagan und Michael Gibas verfeinern deshalb den statischen Aufrufgraphen nach der statischen Analyse durch eine dynamische Analyse [RKG04].

Trevor Parsons und John Murphy suchen mit ihrem Verfahren in komponentenbasierten Softwaresystemen nach Anti-Patterns [PM04]. Anti-Patterns stellen im Gegensatz zu Entwurfsmustern schlechte Lösungen für immer wiederkehrende Probleme dar, die aber trotzdem häufig verwendet werden. Parsons und Murphy suchen nach Anti-Patterns, die die Performanz einer Anwendung reduzieren. Dazu zeichnen sie durch eine dynamische Analyse Methodenaufrufe zusammen mit ihren Performanzdaten auf. In der folgenden statischen Analyse werden die Aufrufgraphen nach den Anti-Patterns durchsucht.

8.3.2 Verfahren zur Entwurfsmustererkennung

Im Folgenden werden drei Ansätze vorgestellt, die das Verhalten von Entwurfsmusterimplementierungen zur Laufzeit zu ihrer Erkennung nutzen. Dabei werden statische und dynamische Analysen kombiniert.

Ein sehr früher Ansatz, der sowohl statische als auch dynamische Analysen zur Entwurfsmustererkennung verwendet, ist von Kyle Brown in seiner Masterarbeit vorgestellt worden [Bro96]. Die Erkennung findet auf Smalltalk-Quelltext statt, dessen strukturelle Eigenschaften wie Klassen, Attribute und Vererbungen in einem statischen Modell repräsentiert werden. Ein weiteres Modell repräsentiert Methodenaufrufe zwischen Objekten. Diese Methodenaufrufe werden zur Laufzeit des zu untersuchenden Programms durch den Smalltalk-Interpreter aufgezeichnet und in dem dynamischen Modell aufbereitet.

Der Ansatz von Brown erkennt vier der Entwurfsmuster aus [GHJV95]: *Composite*, *Decorator*, *Template Method* und *Chain of Responsibility*. Die Erkennungsalgorithmen nutzen jedoch keine direkte Kombination aus statischer und dynamischer Analyse. Für die ersten drei genannten Entwurfsmuster wird ausschließlich das statische Modell des Softwaresystems genutzt. Nur die Erkennung des *Chain-of-Responsibility*-Musters findet auf dem dynamischen Modell statt. Die Erkennungsalgorithmen sind manuell als Methoden der Klassen des statischen und des dynamischen Modells implementiert worden. Die Umsetzung der Algorithmen lässt also keine Kombination aus statischer und dynamischer Analyse zu. Außerdem ist die Wartung und die Erweiterung der Erkennungsalgorithmen sehr aufwändig.

Yann-Gaël Guéhéneuc et al. stellen in [GDJ02] ein Werkzeug vor, mit dem Ereignisse eines JAVA-Programms während seiner Laufzeit analysiert werden können. Zu den beobachtbaren Ereignissen zählen unter anderem die Erzeugung von Objekten, Methodenaufrufe oder auch Attributzugriffe. In [GZ05] schlagen die Autoren vor, aus den beobachteten Ereignissen und den Daten einer zusätzlichen statischen Analyse des Quelltextes UML-Sequenzdiagramme und Zustandsdiagramme zu synthetisieren. Diese Diagramme sollen dann für weitere Analysen wie dem *Conformance-Checking* oder einer statischen Entwurfsmustererkennung, die die Autoren bereits in [AG01, ACGJ01] veröffentlicht haben, verwendet werden. Eine Umsetzung dieser Idee ist bisher jedoch nicht vorgestellt worden.

In dem Verfahren von Welf Löwe und Dirk Heuzeroth werden die Ergebnisse aus der statischen Analyse des Quelltextes mit Hilfe der dynamischen Analyse verbessert [HHL02, HHHL03]. Die statische Analyse arbeitet auf dem abstrakten Syntaxgraphen des Quelltextes. Die Struktur eines zu suchenden

Entwurfsmusters wird als Relation über den Elementen des abstrakten Syntaxgraphen definiert. Das Ergebnis der statischen Analyse sind Tupel, die die Relationen erfüllen. Diese Tupel bilden Kandidaten für Entwurfsmusterimplementierungen und sind Eingabe der dynamischen Analyse. Die dynamische Analyse beobachtet nur die Kandidaten während der Laufzeit, der Suchraum wird also durch die statische Analyse eingeschränkt.

Das Verhalten eines Entwurfsmusters wird durch Zustände und Zustandsübergänge spezifiziert. Zu Methoden, die in der statischen Analyse identifiziert werden, können Vor- und Nachbedingungen angegeben werden, die vor beziehungsweise nach Aufruf der Methode gelten müssen. Die Methodenaufrufe werden während der Laufzeit durch einen Debugger erfasst. Bei Ausführung einer Methode werden die Vor- und Nachbedingungen der Methode geprüft. Wird eine Bedingung nicht eingehalten, so verletzt der Kandidat das vorgegebene Verhalten des Entwurfsmusters. In diesem Fall wird der Kandidat verworfen. Erfüllt der Kandidat bei allen Methodenaufrufen die Vor- und Nachbedingungen, so wird er als tatsächliche Entwurfsmusterimplementierung bestätigt.

Die Relationen zur Definition der Struktur eines Entwurfsmusters sowie die Vor- und Nachbedingungen zur Definition des Verhaltens können in zwei unterschiedlichen Sprachen, SanD-Prolog und SanD, spezifiziert werden [HML03]. SanD-Prolog ist eine Sammlung von Prolog-Prädikaten, die grundlegende Relationen definiert. Diese werden zur Spezifikation weiterer Prolog-Regeln genutzt, mit denen die Relationen für die statische Analyse und die Zustände und Zustandsübergänge für die dynamische Analyse eines Entwurfsmusters spezifiziert werden. SanD nutzt dagegen eine Notation ähnlich der objektorientierter Programmiersprachen, bei der die Spezifikationen der Struktur und des Verhaltens eines Entwurfsmusters ineinander integriert sind. Aus den SanD-Spezifikationen werden Prolog-Regeln generiert.

Nach Aussage der Autoren tendieren die Spezifikationen in SanD-Prolog dazu, kompliziert und lang zu werden, so dass sie schlecht zu warten sind. SanD ist dagegen intuitiver und leichter zu warten. Es ist allerdings nicht so ausdrucksstark wie SanD-Prolog. So können zum Beispiel keine Bedingung wie „eine Klasse C darf keine Methode m enthalten“ formuliert werden. Des Weiteren werden die Ergebnisse der Entwurfsmustererkennung nicht bewertet.

8.4 Transformation von Sequenzdiagrammen

Es existieren einige Arbeiten zur Transformation von Sequenzdiagrammen in Automaten. Im Reverse-Engineering wird diese Transformation häufig einge-

setzt, um aus unübersichtlichen Traces, die aus der Software gewonnen werden, in eine abstraktere und auch meist viel kompaktere Form zu bringen. Tarja Systä zum Beispiel stellt ein Werkzeug vor, dass aus Traces Zustandsdiagramme generiert [Sys99a, Sys99b]. Die Traces, die zum Beispiel von Klassen gewonnen werden, stellen nur Ausschnitte aus dem Verhalten eines Programms dar. Aus diesen Traces werden Zustandsdiagramme generiert, die dagegen das gesamte mögliche Verhalten der Klasse abbilden. So wird dem Reverse-Engineer das Verständnis einer Klasse erheblich erleichtert.

In einigen Arbeiten werden Transformationen von Sequenzdiagrammen in Automaten ebenfalls zur Formalisierung der Semantik der Sequenzdiagramme verwendet. Zu diesen Arbeiten gehören unter anderem die Verfahren von Thomas Firley et al. [FHD+99] sowie Jochen Klose und Hartmut Wittke [KW01]. In diesen Arbeiten werden um Zeitinformationen angereicherte UML Sequenzdiagramme beziehungsweise *Live Sequence Charts* (LSCs) in Timed Automata übersetzt. Die Timed Automata dienen wiederum als Eingaben für Modelchecker. So können Spezifikationen auf Basis von Sequenzdiagrammen formal verifiziert werden.

Sebastian Uchitel, Jeff Kramer und Jeff Magee nutzen *Labelled Transistion Systems* (LTS), eine besondere Form der Zustandsautomaten, um während der Systementwicklung das geforderte Verhalten von Komponenten mit tatsächlich implementierten Verhalten abzugleichen [UKM03]. Aus Sequenzdiagrammen, die verschiedene Szenarien beschreiben und einer Reihe von Einschränkungen, die in OCL spezifiziert sind, werden LTS generiert. Aus den LTS kann dann Verhalten abgeleitet werden, das nicht spezifiziert worden ist, und eventuell auch nicht gewollt ist. Unterspezifiziertes Verhaltens einer Komponente wird somit frühzeitig im Entwicklungsprozess erkannt.

8.5 Zusammenfassung

Keines der in diesem Kapitel vorgestellten Verfahren zur Entwurfsmustererkennung erfüllt alle in Abschnitt 2.2 definierten Anforderungen an eine automatische Entwurfsmustererkennung. Die meisten Verfahren verwenden nur eine strukturbasierte Analyse. Das Verhalten wird bei einigen gar nicht, bei anderen nur auf Basis der statischen Analyse potentieller Methodenaufrufe berücksichtigt. Dabei werden die Methodenaufrufe entweder ausschließlich aus dem Quelltext extrahiert oder als in einem UML-Verhaltensmodell gegeben gefordert. Es werden weder Reihenfolgen von Methodenaufrufen, noch Zustände der beteiligten Objekte berücksichtigt. Allen diesen Ansätzen ist gemeinsam, dass die

tatsächlichen Methodenaufrufe nicht zur Laufzeit des zu untersuchenden Softwaresystems ermittelt werden. Somit kann keines dieser Verfahren Kandidaten heraus filtern, die strukturell passen, sich jedoch nicht wie ein Entwurfsmuster verhalten, oder strukturell ähnliche Entwurfsmuster anhand ihres Verhaltens unterscheiden. Die Präzision der rein strukturbasierten Verfahren ist damit prinzipiell bedingt sehr gering.

Die Notwendigkeit, dynamische Analysen zur Erkennung von Entwurfsmustern einzusetzen, wurde schon früh in [Bro96] erkannt. Allerdings wurden in dieser Arbeit die Vorteile einer echten Kombination aus statischer und dynamischer Analyse nicht genutzt. Es existiert nur ein Verfahren von Dirk Heuzeroth und Welf Löwe [HHL02, HHHL03], das konsequent statische und dynamische Analysen zur Entwurfsmustererkennung kombiniert. Es wurde parallel zu der vorliegenden Arbeit entwickelt. Allerdings ist die Anpassbarkeit der Entwurfsmusterspezifikationen wegen der sehr komplexen Spezifikationssprache eingeschränkt und die Ergebnisse der Entwurfsmustererkennung werden nicht bewertet.

Kapitel 9

Zusammenfassung und Ausblick

Im letzten Kapitel werden zunächst das entwickelte Verfahren anhand der in Abschnitt 2.2 genannten Anforderungen diskutiert und die Ergebnisse der Arbeit zusammengefasst. Der zweite Teil des Kapitels gibt einen Ausblick auf mögliche Erweiterungen und andere Anwendungsgebiete der struktur- und verhaltensbasierten Entwurfsmustererkennung.

9.1 Zusammenfassung

In der vorliegenden Arbeit wurde eine automatische Entwurfsmustererkennung konzipiert und realisiert, die sowohl die Struktur als auch das Verhalten der Entwurfsmuster berücksichtigt. Im Folgenden wird diskutiert, ob das Verfahren den in Abschnitt 2.2 definierten Anforderungen Skalierbarkeit, Präzision, Anpassbarkeit und Bewertung gerecht wird.

Die wichtigste Anforderung ist die Skalierbarkeit des Algorithmus. Große, praxisnahe Softwaresysteme müssen in einem zeitlich vertretbarem Zeitraum analysierbar sein. Die Skalierbarkeit der strukturbasierten Entwurfsmustererkennung wurde bereits in [Nie04] und [NSW+02] nachgewiesen. Dort wurden reale Anwendungen mit einem zeitlich vertretbaren Aufwand untersucht. Bei der verhaltensbasierten Entwurfsmustererkennung ist der zeitliche Aufwand im Wesentlichen durch das Sammeln der Traces bestimmt. Zur Ausführung können entweder automatische Tests herangezogen werden, oder aber die Software wird auf Basis der Software-Tomographie in realen Umgebungen eingesetzt. Automatische Tests sind relativ schnell durchzuführen, erzeugen aber künstliche Daten. Der Einsatz in produktiven Bedingungen erzeugt praxisnahe Daten, erfordert aber auch einen größeren zeitlichen Aufwand. Allerdings muss die Ausführung des zu untersuchenden Softwaresystem weder bei automatischen Tests, noch beim Einsatz in produktiven Umgebungen vom

Reverse-Engineer überwacht werden, er kann in dieser Zeit andere Tätigkeiten ausführen. Der Aufwand zur Instrumentierung ist dagegen sehr gering. Der Aufwand der Offline-Verhaltensanalyse der gesammelten Daten bleibt wie die Strukturanalyse in einem zeitlich vertretbarem Maß.

Die Präzision der bereits vorhandenen, strukturbasierten Entwurfsmustererkennung konnte durch die Kombination mit einer Verhaltensanalyse erheblich gesteigert werden. Zum einen können nun Implementierungen von strukturell ähnlichen oder sogar identischen Entwurfsmustern anhand ihres Verhaltens unterschieden werden. Dies konnte in der Anwendung des Verfahrens auf die Entwicklungsumgebung Eclipse gezeigt werden (Abschnitt 6.3). Des Weiteren wurden viele der in der strukturbasierten Entwurfsmustererkennung identifizierten False-Positives mit Hilfe der Verhaltensanalyse aussortiert.

Die Anpassung bereits existierender Struktur- und Verhaltensmuster an individuelle Eigenschaften des zu untersuchenden Softwaresystems kann die Präzision der Entwurfsmustererkennung erheblich steigern. Das in dieser Arbeit vorgestellte Verfahren macht es dem Reverse-Engineer relativ leicht, Struktur- und Verhaltensmuster an individuelle Eigenschaften anzupassen. Die für die Struktur- und Verhaltensmuster entwickelten Spezifikationssprachen sind an UML-Objektdiagramme beziehungsweise -Sequenzdiagramme angelehnt und somit leicht und intuitiv erlernbar für Softwareentwickler. Die Sprachen ermöglichen relativ kompakte und schnell zu erfassende Spezifikationen. Zudem werden die Spezifikationen automatisch in eine Form übersetzt, die von den Erkennungsalgorithmen verarbeitet werden kann. So wird ein iterativer Prozess, bei dem die Entwurfsmustererkennung zunächst wiederholt auf einen Teil des gesamten Softwaresystems angewendet wird, um die Struktur- und Verhaltensmuster sukzessive zu verbessern, optimal unterstützt.

Die struktur- und verhaltensbasierte Entwurfsmustererkennung erhebt nicht den Anspruch, absolut sichere Ergebnisse zu produzieren. Daher werden die Ergebnisse sowohl der Strukturanalyse, als auch der Verhaltensanalyse bewertet, um ihre Güte zu beschreiben. Um viele Implementierungsvarianten abzudecken, beschreiben Strukturmuster zum einen notwendige Strukturen eines Entwurfsmusters. Zum anderen enthalten sie zusätzliche, zur Identifikation einer Entwurfsmusterimplementierung nicht notwendige Strukturen, die aber gute Hinweise auf tatsächliche Entwurfsmusterimplementierungen sind. Die Bewertung der Strukturanalyse gibt an, inwieweit die Struktur eines Kandidaten mit dem Strukturmuster übereinstimmt. Je höher die Übereinstimmung ist, desto wahrscheinlicher ist der Kandidat eine tatsächliche Entwurfsmusterimplementierung. In der Verhaltensanalyse werden viele Traces eines Kandidaten mit dem Verhaltensmuster verglichen, um festzustellen, ob sie konform oder

nicht-konform zum Verhaltensmuster sind. Zudem wird zu jedem Kandidaten die durchschnittliche Zahl der Methodenaufrufe seiner konformen Traces berechnet. Der Reverse-Engineer kann aus diesen Daten relativ schnell schließen, ob der Kandidat eine tatsächliche Entwurfsmusterimplementierung ist.

Neben der Erfüllung der zuvor genannten Anforderungen war ein weiteres Ziel der Arbeit die formale syntaktische wie semantische Definition einer Spezifikationssprache für Verhaltensmuster. Die Syntax der Sprache wurde durch UML-Klassendiagramme und OCL-Invarianten definiert, während die Semantik durch die Angabe einer Transformation der Verhaltensmuster auf deterministische, endliche Automaten festgelegt wurde. Die endlichen Automaten werden zudem von der Verhaltensanalyse verwendet, um die Konformität der beobachteten Traces zu untersuchen.

Die struktur- und verhaltensbasierte Entwurfsmustererkennung ist prototypisch in dem Werkzeug Reclipse umgesetzt worden. Dazu wurde die weit verbreitete Entwicklungsumgebung Eclipse zusammen mit dem Werkzeug Fujaba4Eclipse um die Algorithmen zur Struktur- und Verhaltensanalyse, sowie einer Visualisierung der Ergebnisse ergänzt. Eclipse bietet unter anderem eine Java-basierte Softwareentwicklung. Fujaba4Eclipse ergänzt Eclipse um eine modellbasierte Entwicklung auf Basis der UML und eine automatische Java-Codegenerierung. Somit umfasst das so entstandene Werkzeug Reclipse in erheblichem Umfang den Softwareentwicklungszyklus im Forward-, als auch Reverse-Engineering.

Die praktische Anwendung des Werkzeugs wurde ebenfalls anhand von Eclipse vorgeführt. Einige der in Eclipse verwendeten Entwurfsmusterimplementierungen sind dokumentiert. So konnten die Ergebnisse der Entwurfsmustererkennung mit der Dokumentation verglichen und beurteilt werden.

9.2 Ausblick

Die Entwicklung der struktur- und verhaltensbasierte Entwurfsmustererkennung ist mit der vorliegenden Arbeit nicht abgeschlossen. Wegen des erheblichen Aufwands konnte leider keine umfangreiche Evaluation des Verfahrens durchgeführt werden. In einer solchen Evaluation sollte besonderes Augenmerk auf die Untersuchung der Praxistauglichkeit der Bewertungen gelegt werden. In der Strukturanalyse kann es zum Beispiel vorkommen, dass der Bewertungsalgorithmus bei Mengenknoten aus Annotationen eine Menge von vielen, niedrig bewerteten Annotationen höher bewertet, als eine Menge von wenigen, aber sehr hoch bewerteten Annotationen. Sollte sich der Bewertungsalgorithmus

als noch nicht ausgereift herausstellen, lässt er sich aufgrund seiner modularen Architektur durch den Austausch einzelner Funktionen leicht korrigieren.

In der Verhaltensanalyse wurde bisher auf eine Verrechnung der Anzahl konformer und nicht-konformer Traces, der durchschnittlichen Länge der Traces und der Anzahl der akzeptierten Subtraces zu einem einzigen Wert bewusst verzichtet. Es wird davon ausgegangen, dass die Einzelwerte sehr viel aussagekräftiger sind als ein einzelner Wert, der schwer zu interpretieren ist. Dies muss allerdings erst in der Praxis überprüft werden.

In diesem Zusammenhang steht auch die Frage, wie aussagekräftig die erkannten nicht-konformen Traces sind. Wie bereits in Abschnitt 5.3.2 diskutiert, kommen nicht-konforme Traces mit weit höherer Wahrscheinlichkeit vor, als konforme. Wurden nur einige konforme, aber sehr viele nicht-konforme Traces erkannt, so bedeutet das nicht zwangsläufig, dass der Kandidat ein False-Positive ist. Werden dagegen zu einem negativen Verhaltensmuster konforme Traces erkannt, so sprechen diese Traces sehr viel stärker für ein False-Positive, als nicht-konforme Traces zu einem positiven Verhaltensmuster. In einer Evaluation sollte also auch untersucht werden, inwieweit negative Verhaltensmuster besser geeignet sind zur Erkennung von False-Positives als nicht-konforme Traces.

Um die Präzision der Verhaltensanalyse deutlich zu erhöhen, sollte vor einer Evaluierung die Spezifikationssprache, wie in Abschnitt 6.3.3 erläutert, erweitert werden. Aus Zeitgründen konnte diese Erweiterung leider nicht mehr durchgeführt werden. Durch die erweiterte Syntax wird es möglich, das Verhalten einer Vielzahl weiterer Entwurfsmuster zu beschreiben, was bisher wegen der Einschränkungen nicht möglich war.

Nicht immer steht eine lauffähige Version des zu untersuchenden Softwaresystems zur Verfügung. In solchen Fällen, in denen eine dynamische Analyse nicht möglich ist, wäre es denkbar, die Verhaltensanalyse auf Basis von statischen Analysen durchzuführen. Thomas Eisenbarth, Rainer Koschke und Gunther Vogel stellen in [EKV02] ein Verfahren vor, um aus C-Code statisch Traces zu extrahieren. Aufgrund der in Abschnitt 1.3 genannten Gründe können statische Analysen aber keine exakten Ergebnisse, sondern nur Schätzungen liefern. Es müsste daher untersucht werden, ob die Genauigkeit einer statischen Trace-Extrahierung für die Verhaltensanalyse der Entwurfsmustererkennung ausreichend ist.

Die verhaltensbasierte Entwurfsmustererkennung kann auch zum Forward-Engineering eingesetzt werden. Im Abschnitt 6.3 wurde von der Entdeckung einer Entwurfsmusterimplementierung berichtet, die als eine *Strategy*-Implementierung dokumentiert wurde. Allerdings stellte sich heraus, dass dieser Kandi-

dat zwar die Struktur einer *Strategy* aufweist, sich aber eher wie eine *Chain of Responsibility* verhält. Solche Implementierungen mögen beabsichtigt sein, möglicherweise stellen sie aber Design-Defekte dar, die später behoben werden müssen. Im Forward-Engineering ließen sich solche Design-Defekte verhindern. Mit Hilfe der Verhaltensmuster könnte das beabsichtigte Verhalten als Protokolle festgeschrieben werden. In Regressionstests kann dann während der Entwicklung die Einhaltung dieser Protokolle überprüft werden. Nicht-konforme Traces liefern Hinweise auf Verletzung der Protokolle. Design-Defekte aufgrund falschen Verhaltens werden so frühzeitig im Entwicklungsprozess verhindert. Solche Spezifikationen und Überprüfungen von Protokollen durch Verhaltensmuster ließen sich sogar auf ganze Komponenten ausweiten, mit denen andere Komponenten in einer vorgegeben Form kommunizieren müssen. Die Verhaltensmuster könnten zusammen mit den Komponenten ausgeliefert werden, um die Entwicklung darauf aufbauender Software zu unterstützen.

Literatur

[ACGJ01] Albin-Amiot, Hervé; Cointe, Pierre; Guéhéneuc, Yann-Gaël; Jussien, Narendra: Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together. In: Richardson, Debra (Hrsg.); Feather, Martin (Hrsg.); Goedicke, Michael (Hrsg.): *Proc. of ASE-2001: The* 16^{th} *IEEE Conference on Automated Software Engineering*. Coronado, CA, USA: IEEE Computer Society Press, November 2001, S. 166–173

[AFC98] Antoniol, G.; Fiutem, R.; Christoforetti, L.: Design Pattern Recovery in Object-Oriented Software. In: *Proc. of the* 6^{th} *International Workshop on Program Comprehension (IWPC)*. Ischia, Italien: IEEE Computer Society Press, Juni 1998, S. 153–160

[AG01] Albin-Amiot, Hervé; Guéhéneuc, Yann-Gaël: Design Patterns: A Round-Trip. In: Ardourel, Gilles (Hrsg.); Haupt, Michael (Hrsg.); Agustin, Jose Luis H. (Hrsg.); Ruggaber, Rainer (Hrsg.); Suscheck, Charles (Hrsg.): *Proc. of the* 11^{th} *ECOOP Workshop for Ph.D. Students in Object-Oriented Systems*, 2001, S. 1–10

[BGN+04] Burmester, Sven; Giese, Holger; Niere, Jörg; Tichy, Matthias; Wadsack, Jörg P.; Wagner, Robert; Wendehals, Lothar; Zündorf, Albert: Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. In: *International Journal on Software Tools for Technology Transfer (STTT)* 6 (2004), August, Nr. 3, S. 203–218

[BMMM98] Brown, W.J.; Malveau, R.C.; McCormick, H.W.; Mombray, T.J.: *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. New York, NY, USA: John Wiley and Sons, Inc., 1998

[BOH02] Bowring, Jim; Orso, Allesandro; Harrold, Mary J.: Monitoring Deployed Software Using Software Tomography. In: *Proc. of the 2002 Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. Charleston, SC, USA: ACM Press, November 2002, S. 2–9

[Boo05] Booch, Grady: On Creating a Handbook of Software Architecture. In: *Keynote of the* 5^{th} *Working IEEE/IFIP Conference on Software Architecture (WICSA)*. Pittsburgh, PA, USA: http://www.booch.com/architecture/blog.jsp?archive=2005-11.html, November 2005. – Stand: März 2006

[BP00] Bergenti, Federico; Poggi, Agostino: Improving UML Designs Using Automatic Design Pattern Detection. In: *Proc.12th International Conference on Software Engineering and Knowlege Engineering (SEKE 2000)*. Chicago, IL, USA, Juni 2000, S. 336–343

[Bro96] Brown, Kyle: *Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk*, North Carolina State University, Diplomarbeit, Juni 1996

[CC90] Chikofsky, Elliot J.; Cross II, James H.: Reverse Engineering and Design Recovery: A Taxonomy. In: *IEEE Software* 7 (1990), Januar, Nr. 1, S. 13–17

[Ecl] Eclipse Foundation Inc.: *Eclipse*. http://www.eclipse.org/. – Stand: April 2007

[EKS01] Eisenbarth, Thomas; Koschke, Rainer; Simon, Daniel: Aiding Program Comprehension by Static and Dynamic Feature Analysis. In: *Proc. of the International Conference on Software Maintenance (ICSM 2001)*. Florenz, Italien: IEEE Computer Society Press, November 2001, S. 602–611

[EKV02] Eisenbarth, Thomas; Koschke, Rainer; Vogel, Gunther: Static Trace Extraction. In: *Proc. of the Ninth Working Conference on Reverse Engineering (WCRE'02)*. Richmond, VA, USA: IEEE Computer Society Press, Oktober 2002, S. 128–137

[Ern03] Ernst, Michael D.: Static and dynamic analysis: synergy and duality. In: *Proc. of the ICSE Workshop on Dynamic Analysis (WODA03)*. Portland, Oregon, USA, Mai 2003, S. 25–28

[FHD+99] Firley, Thomas; Huhn, Michaela; Diethers, Karsten; Gehrke, Thomas; Goltz, Ursula: Timed Sequence Diagrams and Tool-Based Analysis – A Case Study. In: *The Second International Conference on The Unified Modeling Language, Beyond the Standard (UML'99)* Bd. 1723, Springer, Oktober 1999, S. 645–660

[FNT98] Fischer, Thorsten; Niere, Jörg; Torunski, Lars: *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling*, Universität Paderborn, Paderborn, Deutschland, Diplomarbeit, Juli 1998

[FP96] Fenton, Norman E.; Pfleeger, Shari L.: *Software Metrics - A Rigorous & Practical Approach*. Second Edition. International Thompson Computer Press, 1996

[Fra92] Frazer, A.: Reverse Engineering - hype, hope or here? In: Hall, P.A.V. (Hrsg.): *Software Reuse and Reverse Engineering in Practice* Bd. Jgg. 12 der UNICOM Applied Information Technology. London, Großbritannien: Chapman & Hall, 1992, S. 209–243

[Fuj] University of Paderborn, Germany: *Fujaba Tool Suite*. http://www.fujaba.de/. – Stand: April 2007

[GB04] Gamma, Erich; Beck, Kent; Gamma, Erich (Hrsg.); Nackman, Lee (Hrsg.); Wiegand, John (Hrsg.): *Contributing to eclipse - Principles, Patterns, and Plug-Ins*. Boston, MA, USA: Addison-Wesley, 2004 (The Eclipse Series)

[GDJ02] Guéhéneuc, Yann-Gaël; Douencey, Rémi; Jussien, Narendra: No Java without Caffeine A Tool for Dynamic Analysis of Java Programs. In: *17th IEEE International Conference on Automated Software Engineering (ASE 2002)*. Edinburgh, Schottland, Großbritannien: IEEE Computer Society Press, September 2002, S. 117–126

[GHJV95] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA, USA: Addison-Wesley, 1995

[GMW06] Giese, Holger; Meyer, Matthias; Wagner, Robert: A Prototype for Guideline Checking and Model Transformation in Matlab/Simulink. In: Giese, Holger (Hrsg.); Westfechtel, Bernhard (Hrsg.): *Proc. of the* 4^{rd} *International Fujaba Days 2006, Bayreuth, Deutschland* Bd. tr-ri-06-275, Universität Paderborn, September 2006

[GZ05] Guéhéneuc, Yann-Gaël; Ziadi, Tewfik: Automated Reverse-engineering of UML v2.0 Dynamic Models. In: Demeyer, Serge (Hrsg.); Mens, Kim (Hrsg.); Wuyts, Roel (Hrsg.); Ducasse, Stéphane (Hrsg.): *Proc. of the* 6^{th} *ECOOP Workshop on Object-Oriented Reengineering*. Glasgow, Schottland, Großbritannien: Springer-Verlag, Juli 2005, S. 1–5

[HHHL03] Heuzeroth, Dirk; Holl, Thomas; Högström, Gustav; Löwe, Welf: Automatic Design Pattern Detection. In: *Proc. of the* 11^{th} *International Workshop on Program Comprehension (IWPC), Portland, USA*. Portland, OR, USA: IEEE Computer Society Press, Mai 2003, S. 94–103

[HHL02] Heuzeroth, Dirk; Holl, Thomas; Löwe, Welf: Combining Static and Dynamic Analyses to Detect Interaction Patterns. In: *Proc. of the* 6^{th} *International Conference on Integrated Design and Process Technology*. Pasadena, CA, USA, Juni 2002, S. 1–7

[HML03] Heuzeroth, Dirk; Mandel, Stefan; Löwe, Welf: Generating Design Pattern Detectors from Pattern Specifications. In: *Proc. of the* 18^{th} *IEEE International Conference on Automated Software Engineering*. Montreal, Quebec, Kanada: IEEE Computer Society Press, Oktober 2003, S. 245–248

[HN90] Harandi, Mehdi T.; Ning, Jim Q.: Knowledge Based Program Analysis. In: *IEEE Transactions on Software Engineering* 7 (1990), Januar/Februar, Nr. 1, S. 74–81

[Int99] International Telecommunication Union: *Message Sequence Chart (MSC)*. Series Z: Languages and General Software Aspects for Telecommunication Systems. November 1999

[Jah99] Jahnke, Jens H.: *Management of Uncertainty and Inconsistency in Database Reengineering Processes*, Universität Paderborn, Paderborn, Deutschland, Dissertation, August 1999

[KB00] Kim, Hyoseob; Boldyreff, Cornelia: A Method to Recover Design Patterns Using Software Product Metrics. In: *Software Reuse: Advances in Software Reusability: 6th International Conference (ICSR-6)* Bd. 1844. Wien, Österreich: Springer-Verlag, Juni 2000, S. 318–335

[KGH06] Kaczorol, Olivier; Guéhéneuc, Yann-Gaël; Hamel, Sylvie: Efficient Identification of Design Patterns with Bit-vector Algorithm. In: di Lucca, Giuseppe A. (Hrsg.); Gold, Nicolas (Hrsg.): *Proc. of the* 10^{th} *European Conference on Software Maintenance and Reengineering.* Bari, Italien: IEEE Computer Society Press, März 2006, S. 173–182

[KP96] Krämer, Christian; Prechelt, Lutz: Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In: *Proc. of the* 3^{rd} *Working Conference on Reverse Engineering (WCRE).* Monterey, CA, USA: IEEE Computer Society Press, November 1996, S. 208–215

[KS97] Kunz, Thomas; Seuren, Michiel F.: Fast Detection of Communication Patterns in Distributed Executions. In: *Proc. of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research.* Toronto, Ontario, Kanada: IBM Press, 1997, S. 1–13

[KSRP99] Keller, Rudolf K.; Schauer, Reinhard; Robitaille, Sébastien; Pagé, Patrick: Pattern-Based Reverse-Engineering of Design Components. In: *Proc. of the* 21^{st} *International Conference on Software Engineering, Los Angeles, USA*, IEEE Computer Society Press, Mai 1999, S. 226–235

[KW01] Klose, Jochen; Wittke, Hartmut: An Automata Based Interpretation of Live Sequence Charts. In: Margaria, Tiziana (Hrsg.); Yi, Wang (Hrsg.): *Proceedings of the* 7^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* Bd. 2031. London, Großbritannien: Springer-Verlag, April 2001, S. 512–527

[Lea97] Lea, Doug: *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, 1997

[Lew03] Lewis, Bil: Recording Events to Analyze Programs. In: *Object-Oriented Technology. ECOOP 2003 Workshop Reader.* Bd. Lecture notes on computer science (LNCS 3013), Springer-Verlag, Juli 2003, S. 1–5

[Mat] The MathWorks, Inc.: *MATLAB.* http://www.mathworks.com/products/matlab/. – Stand: April 2007

[Meh01] Mehner, Katharina: *LNCS 2269.* Bd. 2269/2002: *JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs.* Software Visualization, International Seminar. Berlin, Deutschland: Springer-Verlag, Mai 2001, S. 163–175

[Meh03] Mehner, Katharina: Zur Performanz der Überwachung von Methodenaufrufen mit der Java Platform Debugger Architecture (JPDA). In: *Java Spektrum, SIGS-DATACOM, Troisdorf, Deutschland* Nov./Dez. (2003), November, S. 1–11

[Mey06] Meyer, Matthias: Pattern-based Reengineering of Software Systems. In: *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006).* Benevento, Italien: IEEE Computer Society, Oktober 2006, S. 305–306

[MW05] Meyer, Matthias; Wendehals, Lothar: Selective Tracing for Dynamic Analyses. In: Zaidman, Andy (Hrsg.); Hamou-Lhadj, Abdelwahab (Hrsg.); Greevy, Orla (Hrsg.): *Proc. of the 1st Workshop on Program Comprehension through Dynamic Analysis (PCODA), co-located with the 12th WCRE, Pittsburgh, Pennsylvania, USA* Bd. 2005-12 Universiteit Antwerpen, Belgien, 2005, S. 33–37

[MWT95] Müller, Hausi A.; Wong, Kenny; Tilley, Scott R.: Understanding Software Systems using Reverse Engineering Technologies. In: Alagar, V. S. (Hrsg.); Missaoui, R. (Hrsg.): *Object-Oriented Technology for Database and Software Systems.* Singapur: World Scientific Publishing, Dezember 1995, S. 240–252

[Nie04] Niere, Jörg: *Inkrementelle Entwurfsmustererkennung*, Universität Paderborn, Paderborn, Deutschland, Dissertation, Juni 2004

[NSW+02] Niere, Jörg; Schäfer, Wilhelm; Wadsack, Jörg P.; Wendehals, Lothar; Welsh, Jim: Towards Pattern-Based Design Recovery. In: *Proc. of the 24th International Conference on Software Engineering (ICSE).* Orlando, FL, USA: ACM Press, Mai 2002, S. 338–348

[NWW03] Niere, Jörg; Wadsack, Jörg P.; Wendehals, Lothar: Handling Large Search Space in Pattern-Based Reverse Engineering. In: *Proc. of the 11th International Workshop on Program Comprehension (IWPC).* Portland, OR, USA: IEEE Computer Society Press, Mai 2003, S. 274–279

[Obj] Object Management Group (OMG): *Unified Modeling Language (UML).* http://www.uml.org/. – Stand: April 2007

[Pal01] Palasdies, Marcus: *Design-Pattern Spezifikation und Erkennung auf Basis von Story-Diagrammen*, Universität Paderborn, Paderborn, Deutschland, Diplomarbeit, Mai 2001

[PM04] Parsons, Trevor; Murphy, John: Data Mining for Performance Antipatterns in Component Based Systems Using Run-Time and Static Analysis. In: *Transactions on Automatic Control and Computer Science* 49 (63) (2004), Mai, Nr. 3, S. 113–118

[PSRN04] Philippow, Ilka; Streitferdt, Detlef; Riebisch, Matthias; Naumann, Sebastian: An approach for reverse engineering of design patterns. In: *Software and Systems Modeling, Springer-Verlag Heidelberg* 4 (2004), April, Nr. 1, S. 55–70

[RD99] Richner, Tamar; Ducasse, Stéphane: Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM).* Oxford, Großbritannien: IEEE Computer Society, August 1999, S. 13–22

[RD02] Richner, Tamar; Ducasse, Stéphane: Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. In: *Proc. of the International Conference on Software Maintenance (ICSM'02).* Montreal, Quebec, Kanada: IEEE Computer Society Press, Oktober 2002, S. 34–43

[Rec04] Reckord, Carsten: *Optimierung von Genauigkeitswerten unscharfer Regeln*, Universität Paderborn, Paderborn, Deutschland, Diplomarbeit, Mai 2004

[Rig] *Rigi: A Visual Tool for Understanding Legacy Systems*. http://www.rigi.csc.uvic.ca/. – Stand: April 2007

[Ris00] Rising, Linda; Vlissides, John M. (Hrsg.): *The Pattern Almanac 2000*. Boston, MA, USA: Addison-Wesley, 2000 (Software Patterns Series)

[RKG04] Rountev, Atanas; Kagan, Scott; Gibas, Michael: Static and Dynamic Analysis of Call Chains in Java. In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. Boston, Massachusetts, USA: ACM Press, Juli 2004, S. 1–11

[Roz97] Rozenberg, Grzegorz (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation*. Bd. 1. Singapur: World Scientific Publishing, 1997

[RR02] Riva, Claudio; Rodriguez, Jordi V.: Combining Static and Dynamic Views for Architecture Recovery. In: *Proc. of the* 6^{th} *European Conference on Software Maintenance and Reengineering*. Budapest, Ungarn: IEEE Computer Society Press, März 2002, S. 47–55

[SG98] Seemann, Jochen; von Gudenberg, Jürgen W.: Pattern-Based Design Recovery of Java Software. In: *Proc. of the* 6^{th} *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Lake Buena Vista, FL, USA: ACM Press, November 1998, S. 10–16

[SK98] Schauer, Reinhard; Keller, Rudolf K.: Pattern Visualization for Software Comprehension. In: *Proc. of the* 6^{th} *International Workshop on Program Comprehension (IWPC)*. Ischia, Italien: IEEE Computer Society Press, Juni 1998, S. 1–9

[SO05] Seesing, Arjan; Orso, Allesandro: InsECTJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse. In: *Proceedings of the eclipse Technology eXchange (eTX)*

Workshop at OOPSLA 2005. San Diego, CA, USA, Oktober 2005, S. 49–53

[SO06] Shi, Nija; Olsson, Ronald A.: Reverse Engineering of Design Patterns from Java Source Code. In: *Proc. of the* 21^{st} *IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. Tokyo, Japan, September 2006, S. 123–134

[SS03] Smith, Jason M.; Stotts, David: SPQR: Flexible Automated Design Pattern Extraction From Source Code. In: *Proc. of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*. Montreal, Kanada: IEEE Computer Society Press, Oktober 2003, S. 215–224

[Sys99a] Systä, Tarja: Dynamic reverse engineering of Java software. In: Ducasse, S. (Hrsg.); Ciupke, O. (Hrsg.): *Proc. of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*. Lissabon, Portugal, Juni 1999 (FZI Report 2-6-6/99), S. 1–7

[Sys99b] Systä, Tarja: On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software. In: *Proc. of the 6th Working Conference on Reverse Engineering (WCRE99)*. Atlanta, GA, USA, Oktober 1999, S. 304–313

[TA99] Tonella, Paolo; Antoniol, Giulio: Object Oriented Design Pattern Inference. In: *Proc. of the* 9^{th} *International Conference on Software Maintenance (ICSM)*. Oxford, Großbritannien: IEEE Computer Society Press, September 1999, S. 230–238

[TCHS05] Tsantalis, Nikolaos; Chatzigeorgiou, Alexander; Halkidis, Spyros T.; Stephanides, George: A Novel Approach to Automated Design Pattern Detection. In: *Proc. of the* 10^{th} *Panhellenic Conference on Informatics (PCI'2005)*. Volos, Griechenland: Springer-Verlag, November 2005 (LNCS), S. 1–15

[TP02] Tonella, Paolo; Potrich, Alessandra: Static and Dynamic C++ Code Analysis for the Recovery of the Object Diagram. In: *Proc. of the* 18^{th} *IEEE International Conference on Software Maintenance (ICSM'02)*. Montreal, Quebec, Kanada: IEEE Computer Society Press, Oktober 2002, S. 54–63

[Tra06] Travkin, Dietrich: *Bewertung automatisch erkannter Instanzen von Software-Mustern*, Universität Paderborn, Paderborn, Deutschland, Diplomarbeit, August 2006

[UKM03] Uchitel, Sebastian; Kramer, Jeff; Magee, Jeff: Behaviour Model Elaboration using Partial Labelled Transition Systems. In: *Proc. of the* 9^{th} *European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* Helsinki, Finnland: ACM Press, September 2003, S. 19–27

[VK98] Vigna, Giovanni; Kemmerer, Richard A.: NetSTAT: A Network-based Intrusion Detection Approach. In: *Proc. of the* 14^{th} *Annual Computer Security Application Conference.* Scottsdale, AZ, USA: IEEE Computer Society Press, Dezember 1998, S. 25–34

[Wen01] Wendehals, Lothar: *Cliché- und Mustererkennung auf Basis von Generic Fuzzy Reasoning Nets*, Universität Paderborn, Paderborn, Deutschland, Diplomarbeit, Oktober 2001

[Wen03] Wendehals, Lothar: Improving Design Pattern Instance Recognition by Dynamic Analysis. In: Cook, Jonathan (Hrsg.); Ernst, Michael (Hrsg.): *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA).* Portland, OR, USA, Mai 2003, S. 29–32

[Wen04] Wendehals, Lothar: Specifying Patterns for Dynamic Pattern Instance Recognition with UML 2.0 Sequence Diagrams. In: Doberkat, E.-E. (Hrsg.); Kelter, U. (Hrsg.): *Softwaretechnik-Trends: Proc. of the* 6^{th} *Workshop Software Reengineering (WSR)* Bd. 24/2. Bad Honnef, Deutschland, Mai 2004, S. 63–64

[Wil92] Wills, Linda M.: *Automated Program Recognition by Graph Parsing.* Cambridge, Mass., USA, Massachussets Institute of Technology, Dissertation, 1992

[WME04] Wendehals, Lothar; Meyer, Matthias; Elsner, Andreas: Selective Tracing of Java Programs. In: Schürr, Andy (Hrsg.); Zündorf, Albert (Hrsg.): *Proc. of the 2nd International Fujaba Days 2004, Darmstadt, Germany* Bd. tr-ri-04-253, Universität Paderborn, Paderborn, Deutschland, September 2004, S. 7–10

[WO06] Wendehals, Lothar; Orso, Alessandro: Recognizing Behavioral Patterns at Runtime using Finite Automata. In: *Proc. of the* 4^{th} *ICSE 2006 Workshop on Dynamic Analysis (WODA)*. Schanghai, China: ACM Press, Mai 2006, S. 33–40

[WSV04] Wu, Lei; Sahraoui, Houari; Valtchev, Petko: Program Comprehension with Dynamic Recovery of Code Collaboration Patterns and Roles. In: *Proc. of the 2004 conference of the Centre for Advanced Studies on Collaborative research*. Markham, Ontario, Kanada: IBM Press, Oktober 2004, S. 56–67

[Wuy98] Wuyts, Roel: Declarative Reasoning about the Structure of Object-Oriented Systems. In: Gil, Joseph (Hrsg.): *Proc. of TOOLS-USA'98*. Santa Barbara, CA, USA: IEEE Computer Society Press, August 1998, S. 112–124

Anhang A

Struktur- und Verhaltensmuster

Der in der praktischen Anwendung in Kapitel 6 verwendete Strukturmuster-Katalog besteht aus 17 Strukturmustern (Abbildung A.1). Die meisten darunter sind Hilfsmuster, die zur Identifikation der Entwurfsmuster benötigt werden. Im Folgenden sind zu den Entwurfsmustern, zu denen auch Verhaltensmuster existieren, jeweils die Struktur- und Verhaltensmuster aufgeführt.

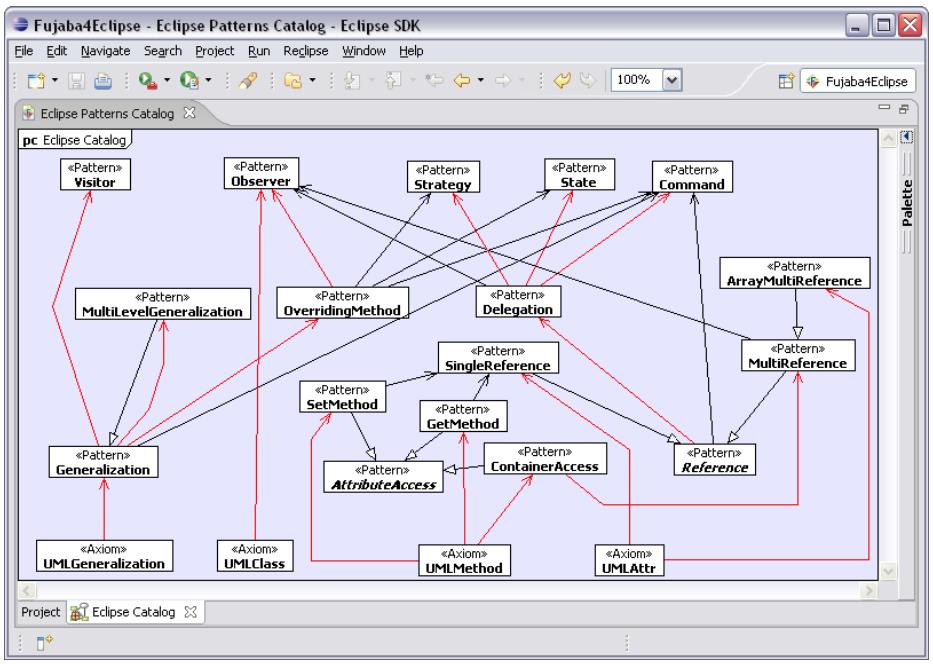

Abbildung A.1: Der Strukturmuster-Katalog für Eclipse

A.1 Command

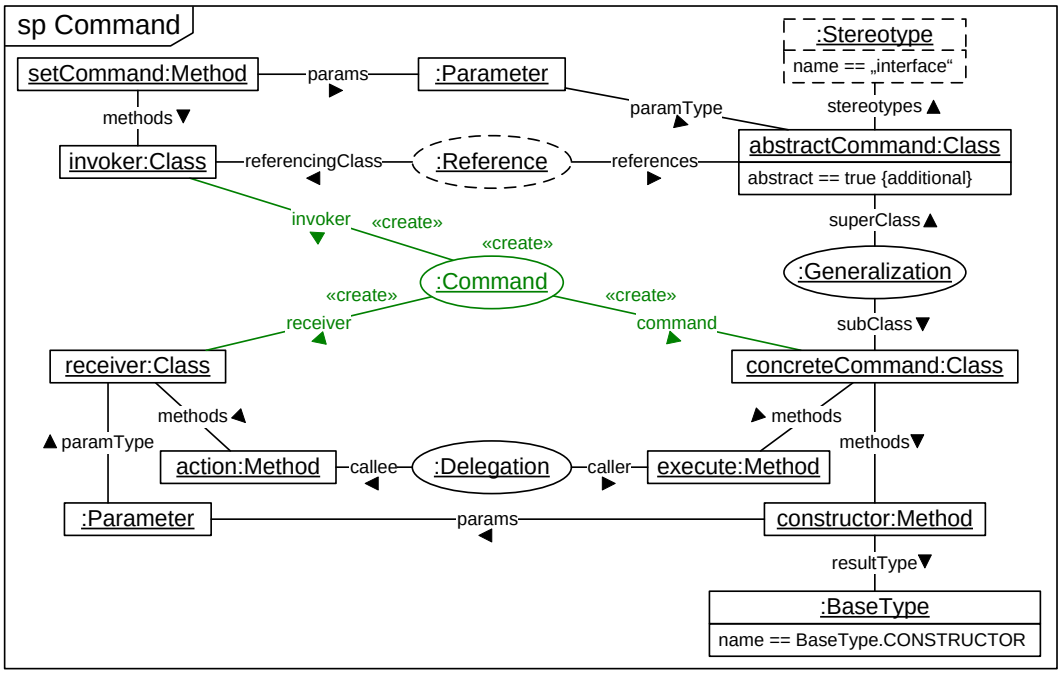

Abbildung A.2: Das *Command*-Strukturmuster

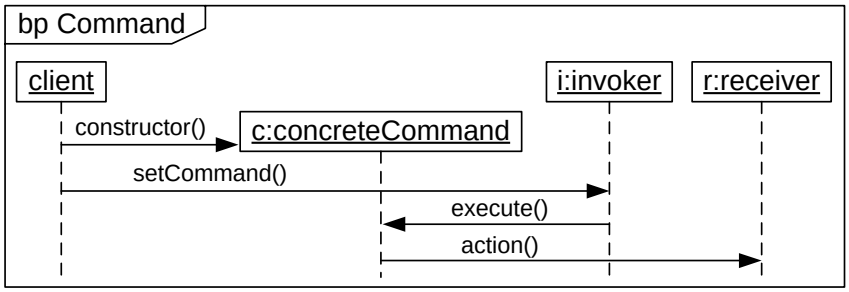

Abbildung A.3: Das *Command*-Verhaltensmuster

A.2 Observer

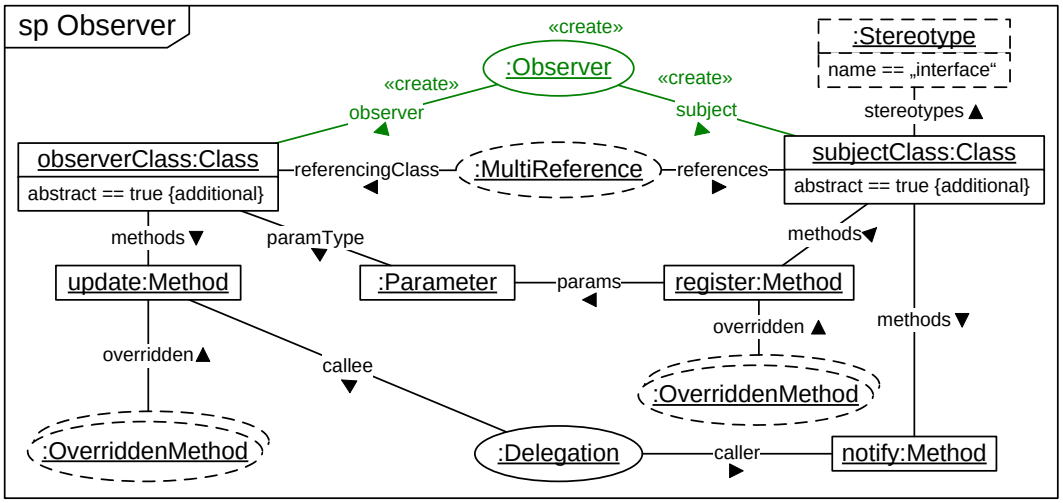

Abbildung A.4: Das *Observer*-Strukturmuster

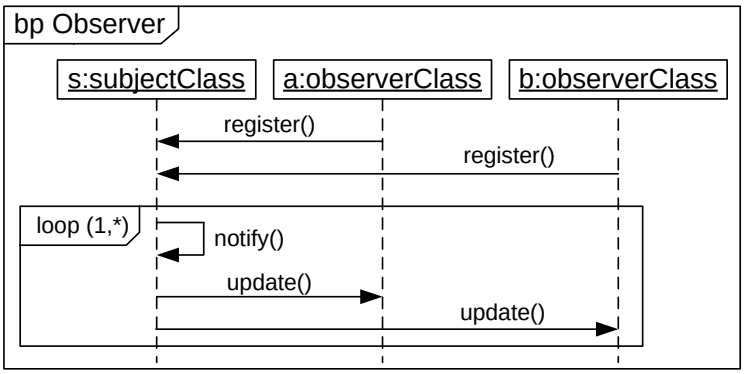

Abbildung A.5: Das *Observer*-Verhaltensmuster

A.3 State

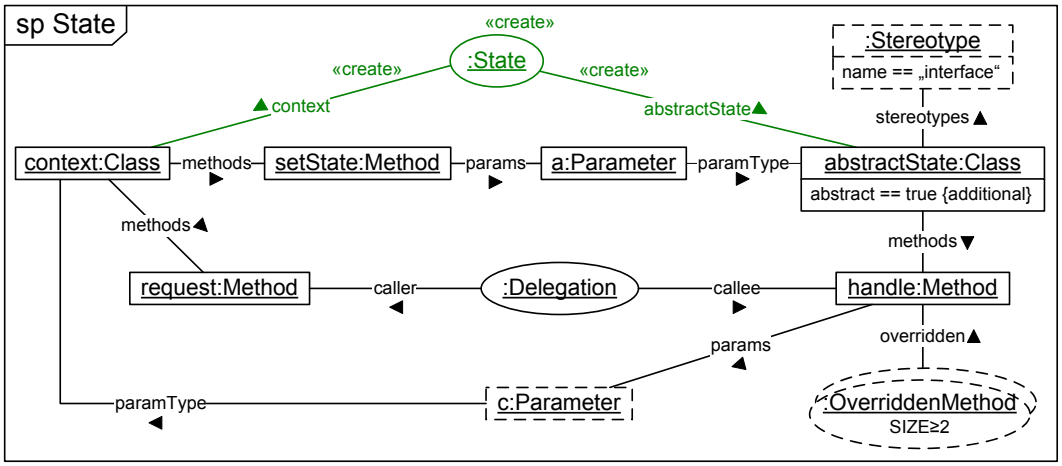

Abbildung A.6: Das *State*-Strukturmuster

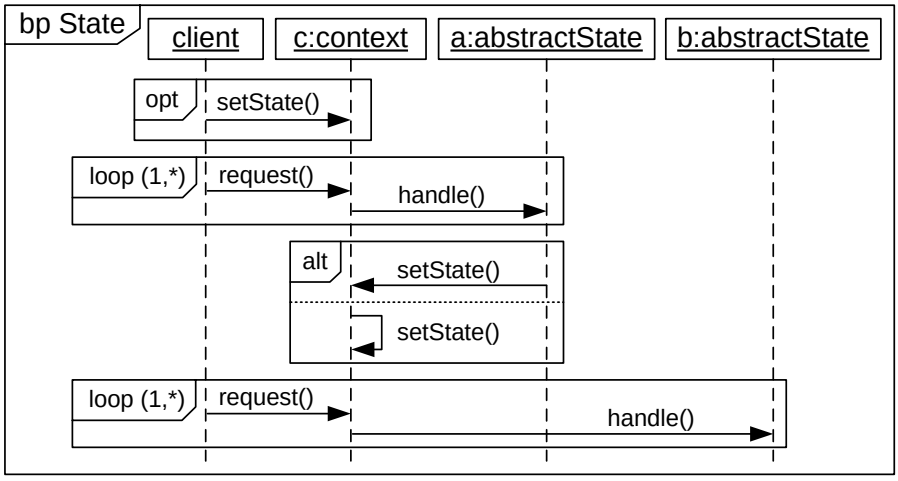

Abbildung A.7: Das *State*-Verhaltensmuster

A.4 Strategy

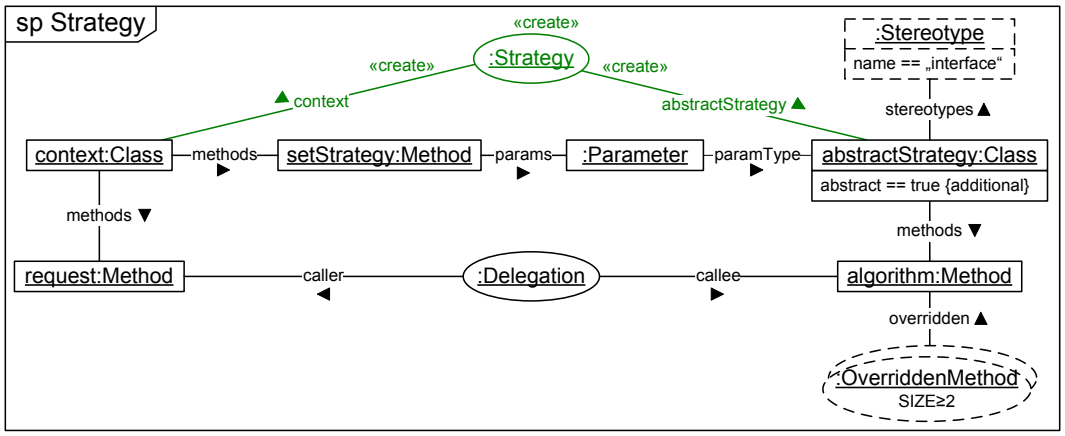

Abbildung A.8: Das *Strategy*-Strukturmuster

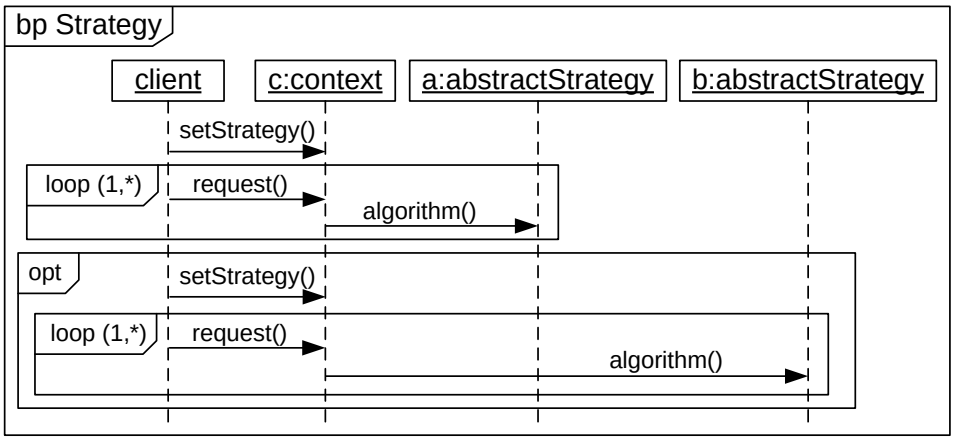

Abbildung A.9: Das *Strategy*-Verhaltensmuster

A.5 Visitor

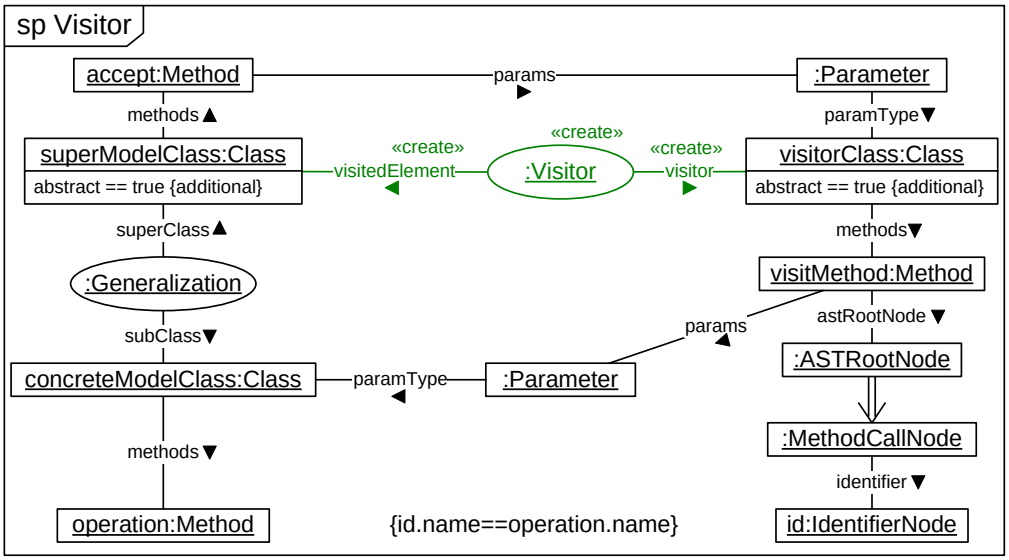

Abbildung A.10: Das *Visitor*-Strukturmuster

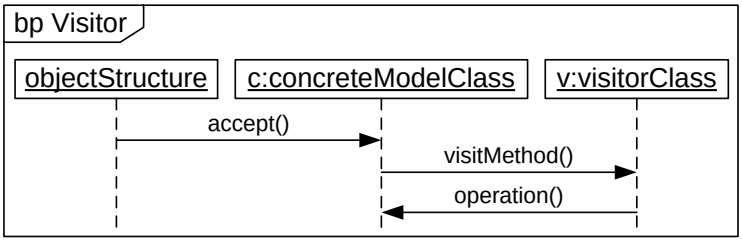

Abbildung A.11: Das *Visitor*-Verhaltensmuster

Anhang B

Reclipse Handbuch

Dieses Handbuch führt anhand der Benutzungsschnittstelle durch den Prozess der struktur- und verhaltensbasierten Entwurfsmustererkennung im Werkzeug Reclipse. Als Beispiel wird der Mediaplayer aus Abschnitt 2.3.1 verwendet. Es wird zunächst erklärt, wie die Spezifikationen der Struktur- und Verhaltensmuster zur Analyse aufbereitet werden. Danach wird dargestellt, wie in Reclipse der Quelltext des Programms in ein Strukturmodell transformiert und die Strukturanalyse durchgeführt wird. Anschließend wird die Benutzung des Reclipse Tracers und des Werkzeugs zur Instrumentierung des ausführbaren Programmcodes erläutert. Zum Abschluss wird die Durchführung der Verhaltensanalyse mit Reclipse behandelt.

B.1 Generierung von Struktur- und Verhaltensmusterkatalogen

Im Folgenden wird von bereits existierenden Spezifikationen von Struktur- und Verhaltensmustern ausgegangen. Es wird nur kurz darauf eingegangen, wie die Struktur- und Verhaltensmuster mit Hilfe der Editoren spezifiziert werden. Ausführlicher wird beschrieben, wie aus den vorhandenen Mustern Kataloge generiert werden, die von der Struktur- und Verhaltensanalyse verwendet werden können.

Die Struktur- und Verhaltensmuster eines Katalogs werden in einem Fujaba-Modell gespeichert. Darin ist auch die Spezifikation des Strukturmodells enthalten. In Reclipse werden die Fujaba-Modelle wiederum Eclipse-Projekten zugeordnet. Nach dem Öffnen eines Modells stehen die Spezifikationen zur Bearbeitung zur Verfügung. Der Reverse-Engineer kann neue Struktur- oder Verhaltensmuster hinzufügen, existierende ändern oder entfer-

nen. In Abbildung B.1 ist das *State*-Strukturmuster zu sehen, das im Editor zur Bearbeitung geöffnet wurde.

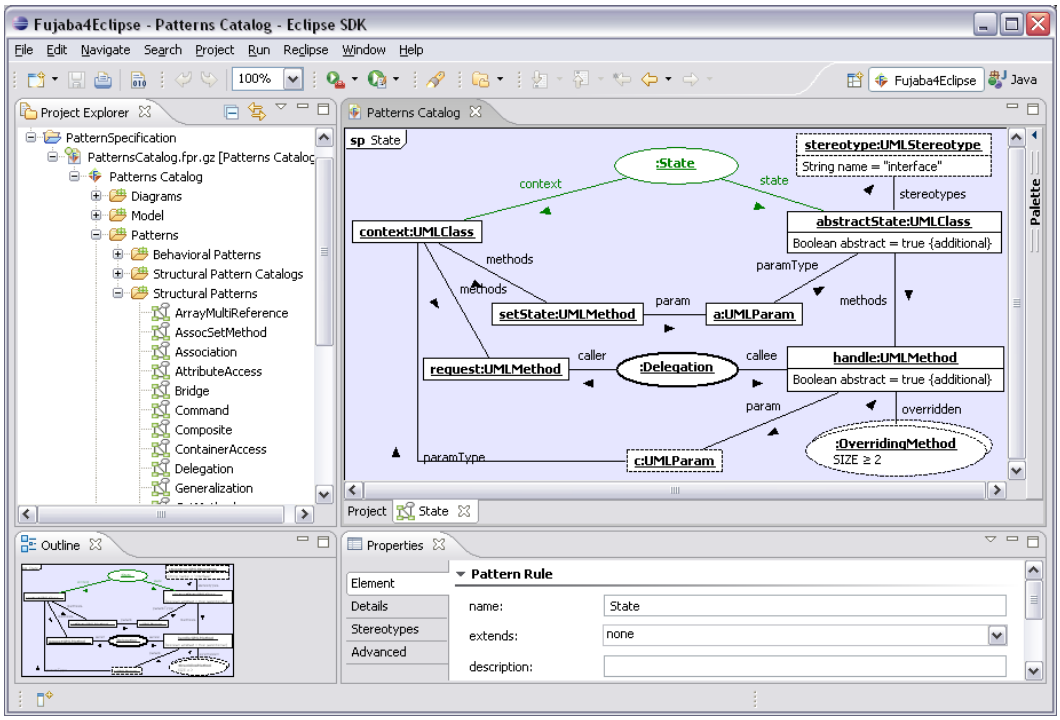

Abbildung B.1: Die Spezifikation des *State*-Strukturmusters in Reclipse

Auf der linken Seite ist der so genannte *Project Explorer* zu sehen, über den man Zugriff auf alle Projekte und die darin enthaltenen Dokumente erhält. Das Fujaba-Modell ist geöffnet und zeigt unter anderem alle darin vorhandenen Strukturmuster an. Links unten ist im *Outline* eine Übersicht über den Editor zu sehen. Rechts oben ist der Editor, mit dem das Modell bearbeitet werden kann. Die Werkzeuge zur Bearbeitung sind über die Palette am rechten Rand des Editors zu erreichen. Dazu gehören zum Beispiel Werkzeuge zum Hinzufügen von Annotationen, Objekten oder auch Verbindungen. Aus Platzgründen wurde die Palette hier jedoch eingeklappt. Rechts unten befindet sich die *Properties*-Sicht, mit der man bestimmte Eigenschaften der im Editor angezeigten Elemente ändern kann, wie zum Beispiel ihre Namen. In diesem Beispiel werden die Eigenschaften des Strukturmusters angezeigt. Man kann den Namen, die Vererbungshierarchie oder die Beschreibung des Strukturmusters ändern.

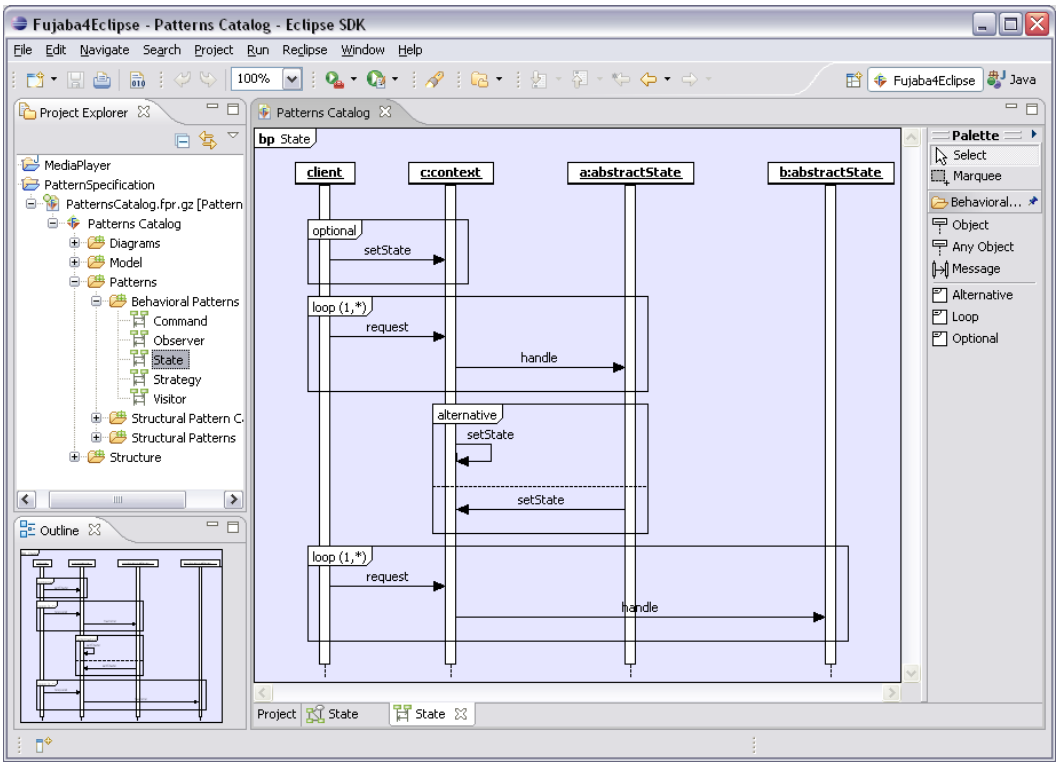

Abbildung B.2: Die Spezifikation des *State*-Verhaltensmusters in Reclipse

In Abbildung B.2 ist die Spezifikation des *State*-Verhaltensmusters zu sehen. Das Verhaltensmuster wurde links aus der Liste der im Modell vorhandenen Verhaltensmuster ausgewählt. In dieser Ansicht ist die Palette am rechten Rand des Editors ausgeklappt. Sie enthält Werkzeuge zum Hinzufügen von Verhaltensmusterobjekten, Nachrichten und Kombinierten Fragmenten oder zum Selektieren von Elementen.

Die Spezifikationen der Struktur- und Verhaltensmuster können in dieser Form nicht von Reclipse zur Analyse verwendet werden. Deshalb werden aus den Spezifikationen Kataloge exportiert, die die Muster in einer für die Analyse aufbereiteten Form enthalten. Dazu wird im *Project Explorer* aus dem Kontext-Menü des Modells der Menüpunkt *Export* aufgerufen. Es wird ein so genannter *Wizard* geöffnet, in dem man den Punkt *Structural Patterns Catalog* im Zweig *Fujaba* auswählt (siehe Abbildung B.3). Ein Wizard besteht meist aus mehreren Dialogen, die nacheinander verschiedene Informationen abfragen. Anschließend wird eine Aufgabe anhand dieser Informationen ausgeführt.

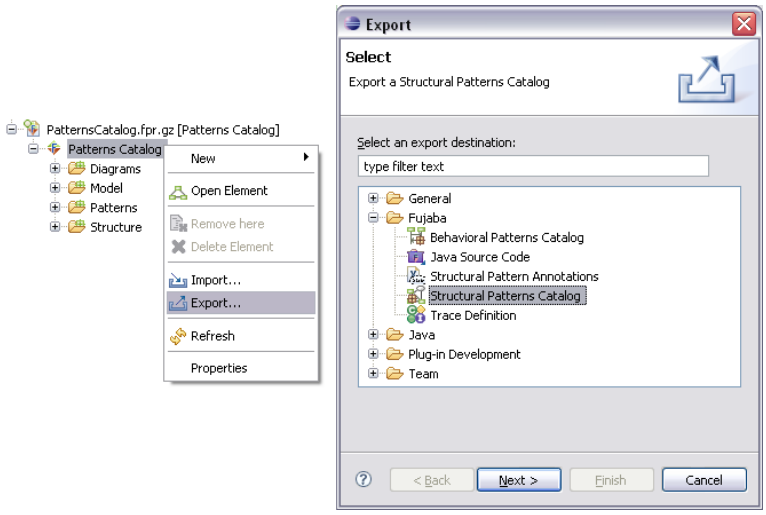

Abbildung B.3: Der Export-Wizard

Der Wizard zum Export eines Strukturmuster-Katalogs besteht aus den Dialogen, die in Abbildung B.4 dargestellt sind. Zunächst wählt man das FUJABA-Modell aus, aus dem der zu exportierende Strukturmuster-Katalog stammt. Im nächsten Dialog wird der Strukturmuster-Katalog ausgewählt. Grundsätzlich existiert immer ein *Main Catalog*, der alle Strukturmuster des Modells enthält. Weitere Strukturmuster-Kataloge können vom Reverse-Engineer im Modell erzeugt werden und enthalten Teilmengen der vorhandenen Strukturmuster. Des Weiteren wird in diesem Dialog angegeben, wo der generierte Strukturmuster-Katalog als Datei abgelegt werden soll.

Aus den Strukturmustern werden JAVA-Klassen generiert, die von der Strukturanalyse zur Suche in der Struktur verwendet werden. Damit diese JAVA-Klassen übersetzt werden können, müssen im folgenden Dialog die Bibliotheken angegeben werden, die zur Übersetzung notwendig sind. Im letzten Dialog muss der Reverse-Engineer entscheiden, ob das Modell mit den Strukturmuster-Spezifikationen im generierten Katalog gespeichert werden soll. Das Modell wird zur Auswertung der identifizierten Kandidaten nach der in Abschnitt 3.1 beschriebenen Methode benötigt. Außerdem können weitere Optionen ausgewählt werden, die zur Suche nach Fehlern während der Generierung nützlich sind. Nach Betätigung der *Finish*-Schaltfläche werden die JAVA-Klassen generiert, übersetzt und in einer Bibliothek zusammengefasst, die später von der Strukturanalyse verwendet wird.

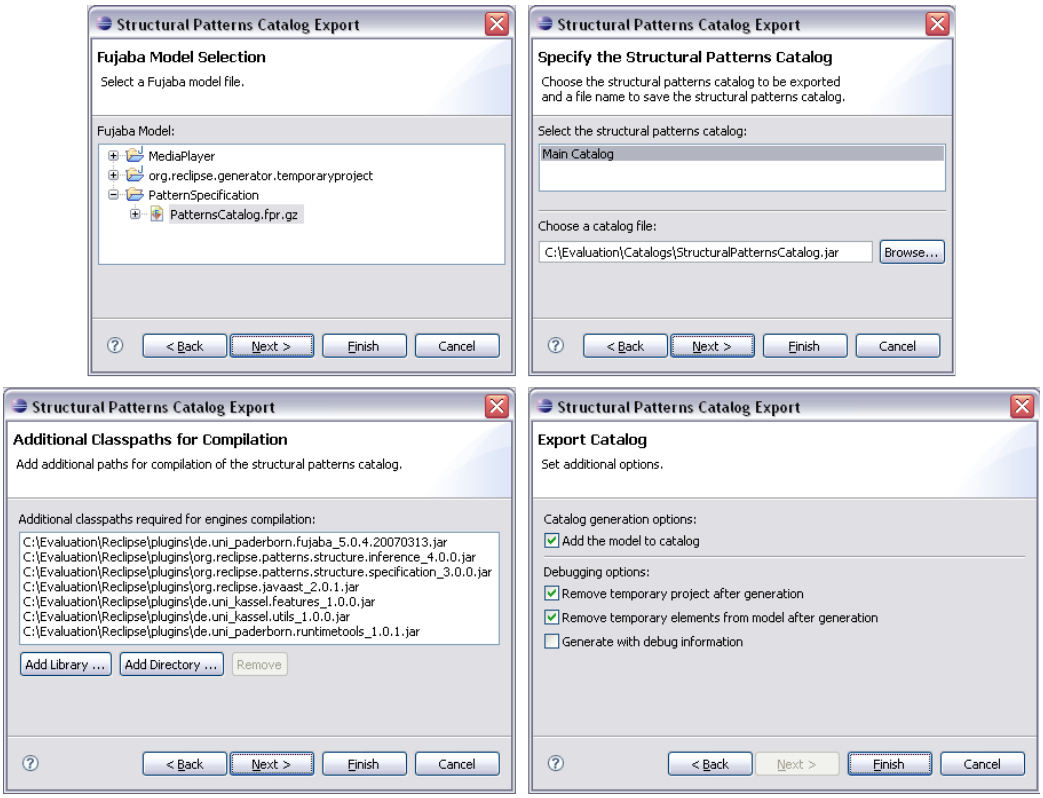

Abbildung B.4: Der Export eines Strukturmuster-Katalogs

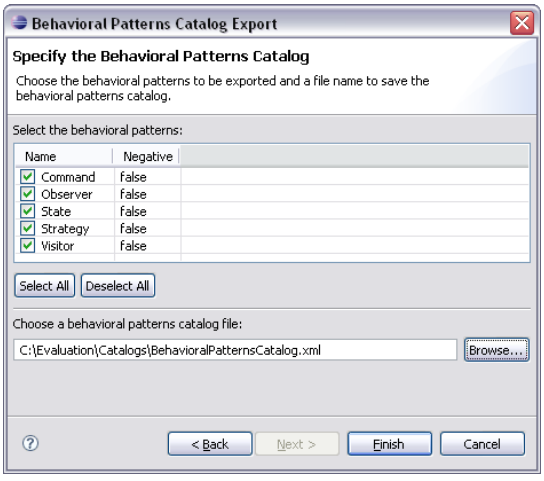

Abbildung B.5: Der Export eines Verhaltensmusterkatalogs

Zum Export der Verhaltensmuster wird wiederum der Export-Wizard geöffnet. Hier wählt man nun den Punkt *Behavioral Patterns Catalog* (Abbildung B.3) aus. Nach Auswahl des Modells mit den Verhaltensmustern wird der Dialog aus Abbildung B.5 angezeigt. Darin gibt man die zu exportierenden Verhaltensmuster und eine Datei für den Katalog an. Die Verhaltensmuster werden dann, wie in Abschnitt 4.4 beschrieben, in endliche Automaten transformiert und in einer generischen Beschreibungssprache gespeichert. Die Spezifikation der Beschreibungssprache findet sich in Anhang C.2.4.

B.2 Strukturbasierte Entwurfsmustererkennung

Zur Strukturanalyse legt man zunächst ein neues FUJABA-Modell an. In diesem Fall wurde ein Modell mit dem Namen *Parsed Mediaplayer Model* erzeugt und in das Projekt mit dem Quelltext des Mediaplayers gelegt. Dann öffnet man den Import-Wizard über das Kontext-Menü des Modells, um den JAVA-Quelltext in das Modell zu importieren. Im Wizard wählt man den Punkt *Fujaba Model from Java Source File(s)* aus (Abbildung B.6).

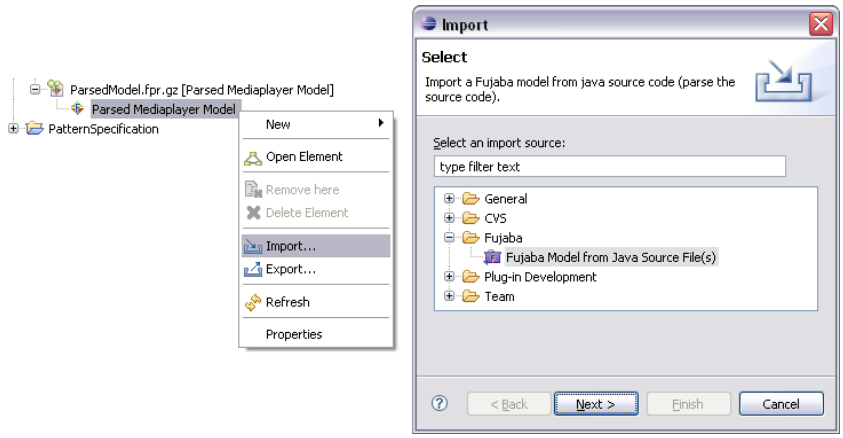

Abbildung B.6: Der Import-Wizard

Nach Auswahl des Modells, in das der JAVA-Quelltext importiert werden soll, gibt der Reverse-Engineer im nächsten Dialog des Wizards (Abbildung B.7) die JAVA-Quelldateien oder auch die Verzeichnisse mit den Quelldateien an. Verzeichnisse können rekursiv nach JAVA-Quelldateien durchsucht werden. Im letzten Dialog kann man einige Optionen auswählen, unter anderem, ob für

jedes JAVA-Paket ein eigenes Klassendiagramm erzeugt werden soll, oder ob alle Klassen in ein einzelnes Klassendiagramm aufgenommen werden sollen.

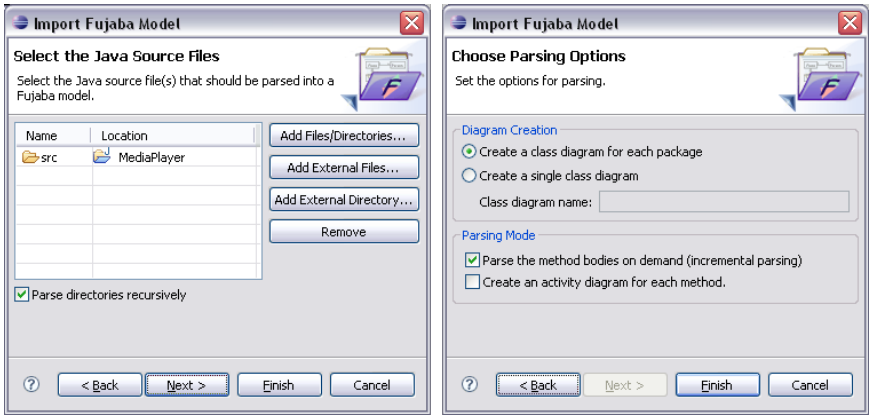

Abbildung B.7: Der Import von JAVA-Quelltext in ein FUJABA-Modell

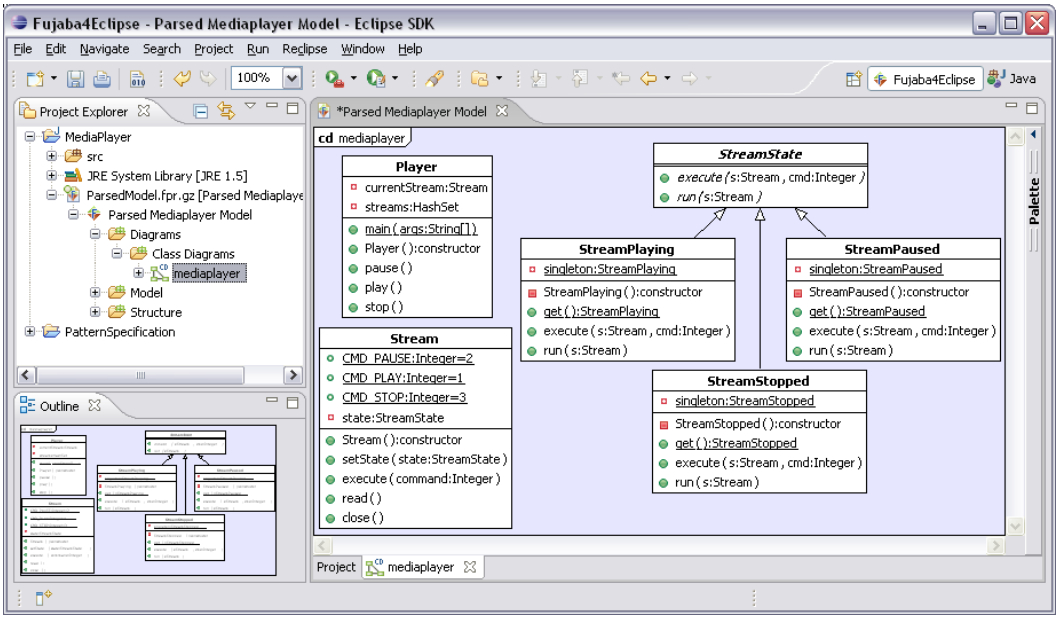

Abbildung B.8: Der Mediaplayer dargestellt im Klassendiagramm

Nach dem Import kann sich der Reverse-Engineer die JAVA-Klassen in Klassendiagrammen darstellen lassen. Wie bereits in Abschnitt 2.3.1 erläutert, werden keine Assoziationen zwischen den Klassen angezeigt, da diese das Ergebnis weiter gehender Analysen sind. In Abbildung B.8 ist das Klassendiagramm des Mediaplayers dargestellt.

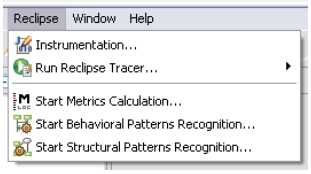

Abbildung B.9: Das *Reclipse*-Menü

Die Strukturanalyse wird über den Menüpunkt *Start Structural Patterns Recognition...* im *Reclipse*-Menü aufgerufen (Abbildung B.9). Es wird ein Wizard geöffnet, in dem man zunächst das Modell auswählt, in dem nach Strukturmustern gesucht werden soll.

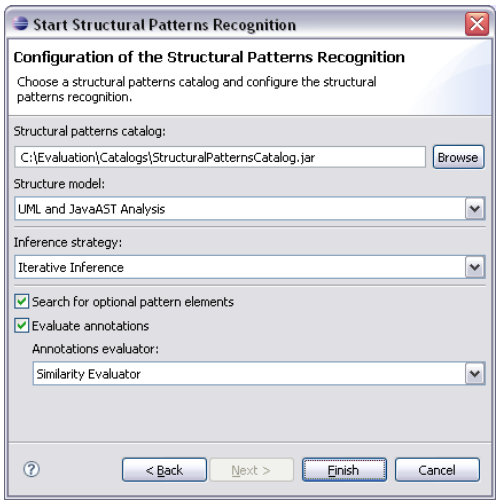

Abbildung B.10: Der Dialog zum Starten der Strukturanalyse

Im zweiten Dialog (Abbildung B.10) wählt man als erstes den Strukturmuster-Katalog aus. Dies ist der Katalog, der zuvor exportiert wurde. Des Weiteren wählt man ein Strukturmodell. Wie in Abschnitt 2.3.6 beschrieben wurde, kann die Strukturanalyse auch auf andere Strukturmodelle wie zum

Beispiel für MATLAB/SIMULINK eingesetzt werden. Das entsprechende Strukturmodell wird hier bestimmt. Zur Strukturanalyse stehen verschiedene Inferenzalgorithmen zur Verfügung. In diesem Fall wird der in Abschnitt 2.3.4 beschriebene Algorithmus verwendet. Im unteren Bereich des Dialogs kann angegeben werden, ob nach optionalen Elementen der Strukturmuster gesucht werden soll und ob die Kandidaten bewertet werden sollen. Soll eine Bewertung durchgeführt werden, muss der Strukturmuster-Katalog das FUJABA-Modell mit den Spezifikationen der Strukturmuster enthalten. Die Strukturanalyse wird schließlich durch Betätigung der *Finish*-Schaltfläche gestartet.

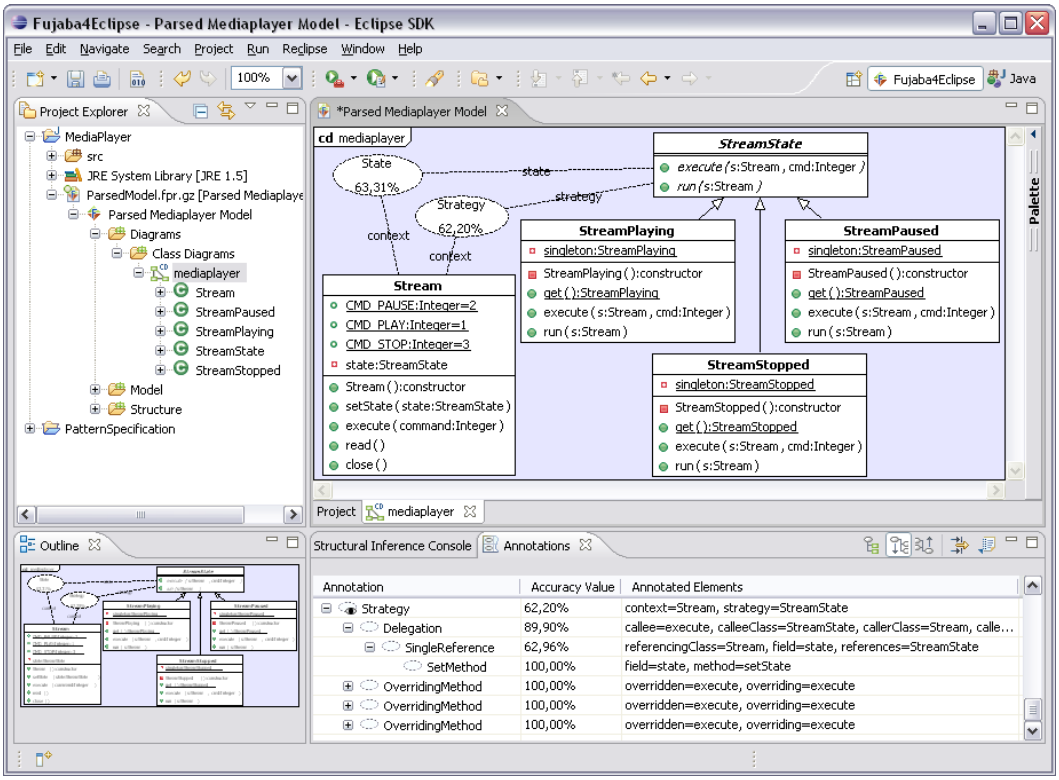

Abbildung B.11: Das Ergebnis der Strukturanalyse

In Abbildung B.11 ist das Ergebnis der Strukturanalyse zu sehen. Im Klassendiagramm werden die Annotationen zweier Kandidaten angezeigt, eine *State*- und eine *Strategy*-Annotation. Diese beiden Kandidaten sind jedoch nur ein kleiner Ausschnitt aus allen gefundenen Kandidaten. Die im Klassendiagramm dargestellten Kandidaten können zur besseren Übersicht gefiltert werden. Im unteren Bereich ist die *Annotations*-Sicht geöffnet, die dagegen

alle identifizierten Kandidaten, sowie ihre Bewertungen und die von ihnen annotierten Elemente auflistet. Des Weiteren werden darin Abhängigkeiten zwischen den Kandidaten angezeigt. Zum Beispiel baut der dargestellte *Strategy*-Kandidat auf einen *Delegation*-Kandidaten und drei *OverridingMethod*-Kandidaten auf, die wiederum von weiteren Kandidaten abhängig sind.

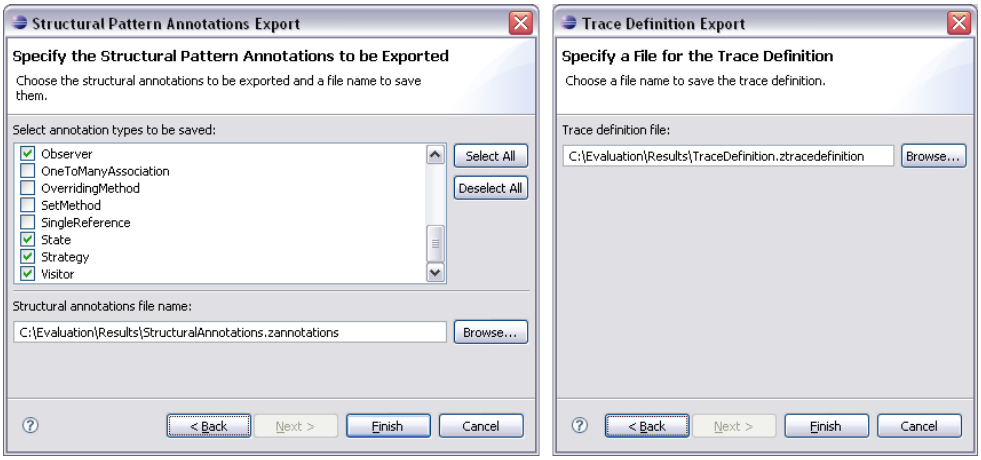

Abbildung B.12: Der Export der Kandidaten und der Trace-Definition

In der Verhaltensanalyse werden die Kandidaten aus der Strukturanalyse überwacht, wie in Abschnitt 3.2 erläutert wurde. Dazu benötigt die Verhaltensanalyse zum einen für jeden Kandidaten die Bindung der Strukturmustervariablen an die Elemente der Struktur und zum anderen die Methoden, die zur Laufzeit des Softwaresystems beobachtet werden müssen. Die Bindungen der Strukturmustervariablen sind in den Annotationen enthalten und werden durch Export der Annotationen zur Verfügung gestellt. Dazu wählt man im Export-Wizard den Punkt *Structural Pattern Annotations* aus (Abbildung B.3, Seite 200). In Abbildung B.12 ist auf der linken Seite der Dialog des Export-Wizards für die Annotationen zu sehen. Hier gibt man die Typen der Annotationen, die exportiert werden sollen, und die Datei für die Annotationen an. Das Datenformat für Annotations-Dateien ist im Anhang C.2.1 zu finden.

Auf der rechten Seite in Abbildung B.12 ist der Dialog zum Export der Trace-Definition dargestellt. Er wird im Export-Wizard über den Punkt *Trace Definition* aufgerufen. Die Trace-Definition enthält die Informationen, welche Methoden zur Laufzeit überwacht werden müssen. Das Datenformat der Trace-Definition ist im Anhang C.2.2 spezifiziert.

B.3 Verhaltensbasierte Entwurfsmustererkennung

Im Folgenden wird beschrieben, wie zunächst die Daten der Gesamtanalyse für die Software-Tomographie in Daten für Teilanalysen aufgetrennt werden. Dann wird erklärt, wie das zu untersuchende Softwaresystem zur Laufzeit mit Hilfe des Reclipse Tracers oder der Instrumentierung beobachtet wird. Bei der Ausführung wird ein Trace aufgezeichnet, der anschließend durch die Verhaltensanalyse untersucht wird.

B.3.1 Software-Tomographie

Zur Anwendung der Software-Tomographie muss die Gesamtanalyse in viele kleine Teilanalysen aufgeteilt werden. Im Falle der Entwurfsmustererkennung bedeutet das, dass die Gesamtheit der durch die Strukturanalyse identifizierten Kandidaten in Form von Annotationen in viele kleine Gruppen von wenigen Annotationen getrennt werden muss. Gleichzeitig muss die Trace-Definition ebenfalls auf die jeweils zu untersuchenden Kandidaten eingeschränkt werden. Zu diesem Zweck gibt es den *Trace Definition Splitting Wizard*. Er wird über das Kontext-Menü einer Trace-Definition aufgerufen.

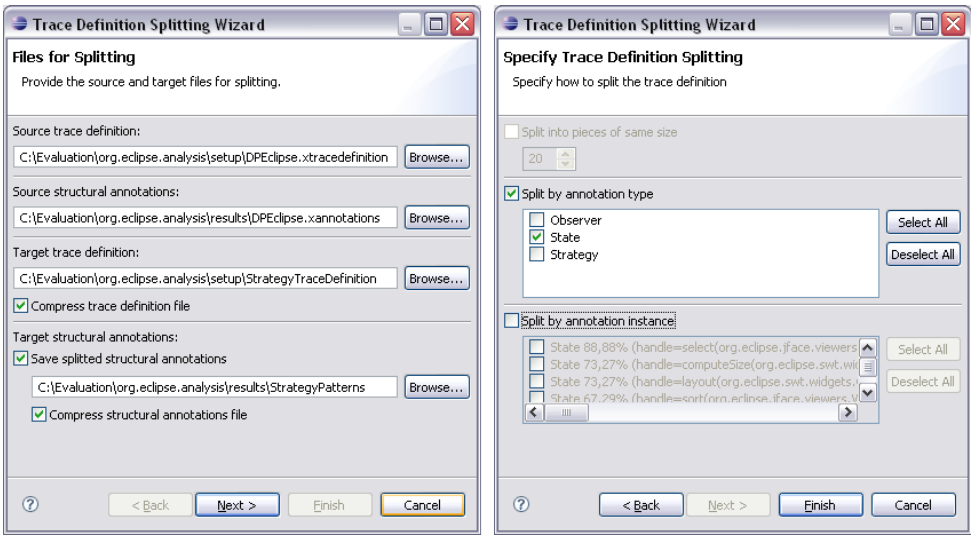

Abbildung B.13: Das Zertrennen der Trace Definition und der Annotationen

Abbildung B.13 zeigt die zwei Dialoge des Wizards. Im ersten Wizard werden die ursprüngliche Trace-Definition und die Datei mit allen von der Strukturana-

lyse erzeugten Annotationen ausgewählt. Gleichzeitig werden die Dateinamen der neu zu erzeugenden, aufgetrennten Trace-Definitionen und Annotationen angegeben.

Im zweiten Dialog können nun aufgrund von drei Kriterien die Daten getrennt werden. Zunächst einmal können die Annotation in Gruppen gleicher Größe aufgeteilt werden. Die Anzahl der Annotationen jeder Teilgruppe kann festgelegt werden. Passend zu jeder Teilgruppe von Annotationen wird jeweils eine Trace-Definition generiert. Die Dateien werden in nummerierter Form abgelegt. Die Annotationen können aber auch nach ihrem Typ und sogar nach einzelnen Annotationen aufgetrennt werden. So können zum Beispiel alle Annotationen des *Strategy*-Entwurfsmusters separiert von allen anderen zusammen mit einer passenden Trace-Definition gespeichert werden.

Die so entstandene Trace-Definition, die nur einen Teil der Gesamt-Trace-Definition enthält, bildet schließlich die Eingabe zum Debugging beziehungsweise zur Instrumentierung. So wird nur ein Teil der Gesamtanalyse durchgeführt und der Einfluss auf das zu untersuchende System gering gehalten.

B.3.2 Debugging

Der Reclipse Tracer wird über die Toolbar gestartet. Abbildung B.14 zeigt das Menü des Reclipse Tracers. Es können mehrere Konfigurationen für Programme, die zur Laufzeit beobachtet werden sollen, abgerufen werden. Zur Erzeugung einer neuen Konfiguration wählt man *Reclipse Tracer...* aus.

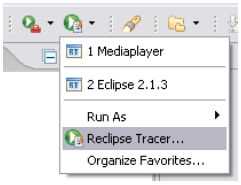

Abbildung B.14: Das Menü zum Aufruf des Reclipse Tracers

Es wird ein Dialog geöffnet, mit dem Konfigurationen verwaltet werden können. In den Abbildungen B.15 und B.16 wird die Konfiguration des Mediaplayers bearbeitet. Auf der *Main*-Seite werden die Hauptklasse, mögliche Programmparameter, das Arbeitsverzeichnis und die Trace-Definition konfiguriert. Die *Classpath*-Seite wird zur Konfiguration des Klassenpfads verwendet. Die hier nicht abgebildeten Seiten *JRE* und *Options* dienen zum Einstellen der Java-Laufzeitumgebung beziehungsweise einiger Optionen.

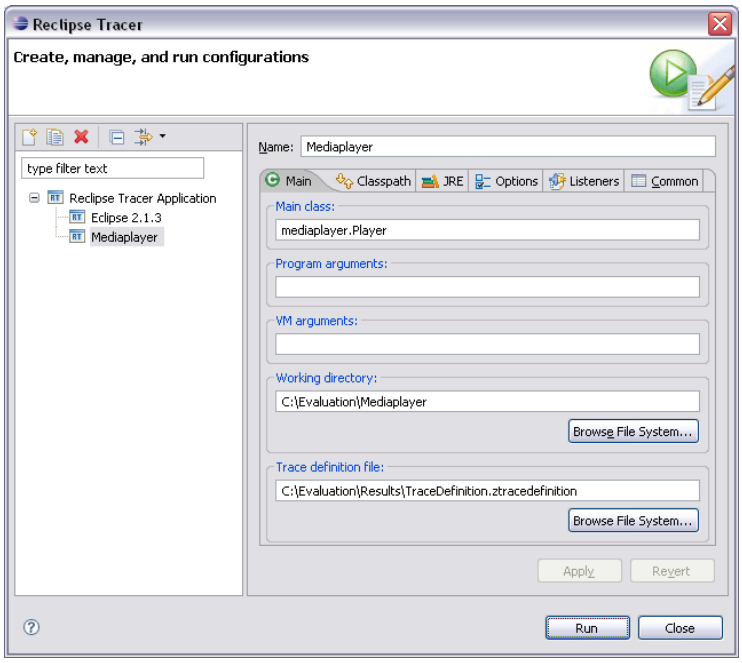

Abbildung B.15: Die Konfiguration des Reclipse Tracers für den Mediaplayer

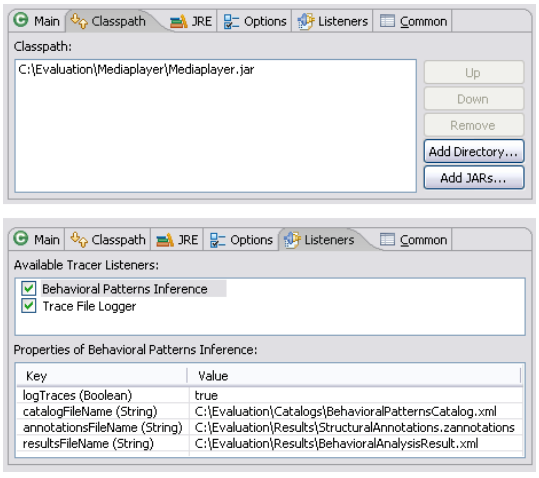

Abbildung B.16: Die Konfiguration des Klassenpfads und der Listener

Auf der Seite *Listener* werden die Module konfiguriert, die die vom Tracer beobachteten Methodenaufrufe verarbeiten. Es stehen zwei Module zur Verfügung. Das Modul *Behavioral Patterns Inference* übergibt die Methodenaufrufe der Verhaltensanalyse. Dazu benötigt das Modul den Verhaltensmuster-Katalog und die Annotationen aus der Strukturanalyse. Des Weiteren muss man angeben, in welche Datei die Ergebnisse der Verhaltensanalyse gespeichert werden sollen. Der Reverse-Engineer kann entscheiden, ob die Traces, die für einen Kandidaten überprüft werden, gespeichert werden sollen. Das Datenformat für die Ergebnisse der Verhaltensanalyse ist im Anhang C.2.5 spezifiziert. Das zweite Modul *Trace File Logger* dient zum Protokollieren des Traces. Das Datenformat ist ebenfalls im Anhang unter C.2.3 zu finden.

Wird das zu untersuchende Programm durch den Reclipse Tracer gestartet, so werden in einer eigenen Perspektive Informationen über den Ablauf angezeigt. Abbildung B.17 zeigt die Ausgaben bei der Ausführung des Mediaplayers.

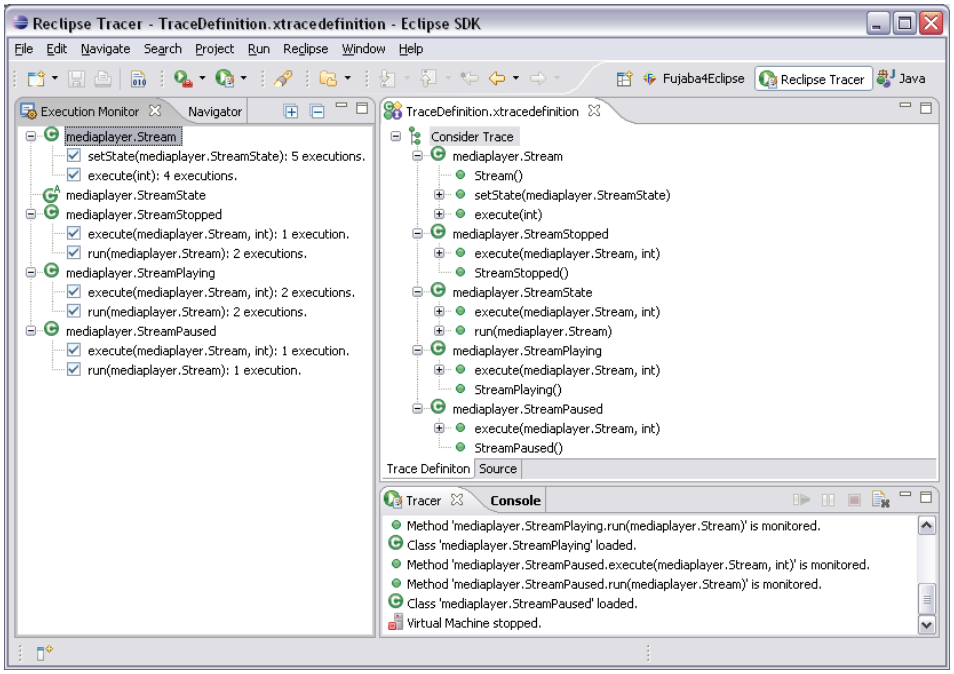

Abbildung B.17: Ausführung des Mediaplayers durch den Reclipse Tracer

Auf der linken Seite der Perspektive ist der *Execution Monitor*. Darin sind alle beobachteten Methoden in einem Baum unterhalb ihrer jeweiligen Klasse

aufgelistet. Zu jeder Methode wird außerdem angegeben, ob und wie oft sie aufgerufen wurde. In der Sicht *Tracer* am unteren Rand werden Meldungen des *Reclipse Tracers* ausgegeben. Sollte das zu untersuchende Programm Fehler oder sonstige Meldungen auf der Konsole ausgeben, so werden diese in der Sicht *Console* angezeigt. Die Trace-Definition kann in dem Editor rechts oben angezeigt und bearbeitet werden.

B.3.3 Instrumentierung

Zur Instrumentierung steht ebenfalls ein Wizard zur Verfügung. Er wird über das *Reclipse*-Menü aufgerufen (Abbildung B.9, Seite 204). Der Instrumentierungs-Wizard besteht aus sechs Dialogen, die im Folgenden erläutert werden.

Im ersten Dialog in Abbildung B.18 kann der Reverse-Engineer entscheiden, ob er eine neue Instrumentierungs-Konfiguration erzeugen will, oder ob er eine existierende bearbeiten oder ausführen möchte. Im zweiten Dialog gibt der Reverse-Engineer die JAVA-Klassen und Bibliotheken an, die instrumentiert werden sollen. Auch die Angabe von Verzeichnissen, die rekursiv nach JAVA-Klassen durchsucht werden können, ist möglich.

Die Instrumentierung benötigt eine Trace-Definition, in der die Methodenaufrufe aufgelistet sind, die überwacht werden sollen. Die Trace-Definition wird im dritten Dialog angegeben. Des Weiteren muss die Hauptklasse des Programms angegeben werden. Die Hauptklasse wird besonders instrumentiert, um den Start und das Beenden des Programms überwachen zu können. Der Reverse-Engineer kann entscheiden, ob die angegebenen Dateien durch die Instrumentierung überschrieben oder in ein anderes Verzeichnis kopiert werden sollen. Die instrumentierten Klassen benötigen zur Ausführung eine Bibliothek, die die Laufzeitumgebung der Instrumentierung enthält. Diese Instrumentierungs-Bibliothek kann in eine der zu instrumentierenden Bibliotheken eingefügt werden oder in ein gegebenes Verzeichnis kopiert werden. In beiden Fällen muss der Reverse-Engineer sicherstellen, dass die Instrumentierungs-Bibliothek zur Laufzeit des zu untersuchenden Programms für alle Klassen im Klassenpfad zur Verfügung steht. Zusätzlich zur Instrumentierungs-Bibliothek wird eine Konfigurationsdatei für die Laufzeitumgebung erzeugt, die ebenfalls entweder in die angegebene, zu instrumentierende Bibliothek eingefügt oder in das gegebene Verzeichnis geschrieben wird. Auch diese Konfigurationsdatei muss im Klassenpfad zu finden sein.

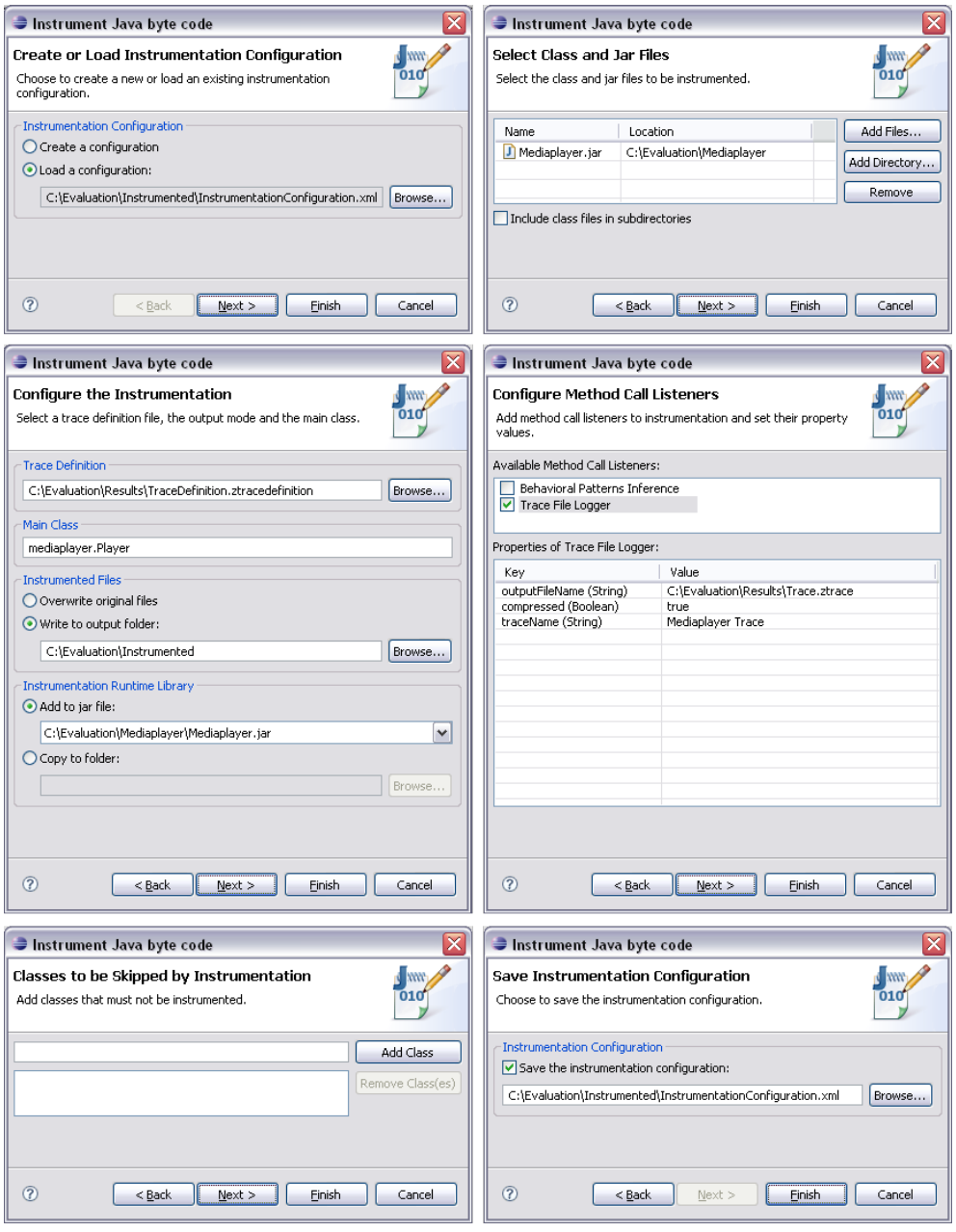

Abbildung B.18: Die Konfiguration der Instrumentierung

Im vierten Dialog werden analog zum Reclipse Tracer Module konfiguriert, die während des Ausführung des Programms über ausgeführte Methodenaufrufe informiert werden. Es stehen die gleichen zwei Module zur Verfügung, wie im Reclipse Tracer. Das Modul *Behavioral Patterns Inference* übergibt die Methodenaufrufe der Verhaltensanalyse. Das Modul *Trace File Logger* dient zum Protokollieren des Traces. Dazu benötigt es die Angabe, in welche Datei der Trace gespeichert werden soll, und einen Namen für den Trace. Da Traces sehr groß werden können, kann die Datei komprimiert werden. Dabei wird eine Datei im Zip-Format angelegt, in die die eigentliche Trace-Datei eingefügt wird.

In einigen Fällen sollen bestimmte Klassen nicht instrumentiert werden. Liegen diese Klassen in zu instrumentierende Bibliotheken oder Verzeichnissen, deren restliche Klassen alle instrumentiert werden sollen, so können im vierten Dialog diese Klassen angegeben werden. Dazu muss nur ihr voll qualifizierter Klassenname in die Liste eingetragen werden. Im letzten Dialog hat der Reverse-Engineer die Gelegenheit, die Instrumentierungs-Konfiguration zu speichern, um sie bei einer späteren Gelegenheit noch einmal zu verwenden.

B.3.4 Verhaltenserkennung

Die Verhaltensanalyse kann im Online- oder Offline-Modus durchgeführt werden. Für den Online-Modus muss das Modul *Behavioral Patterns Inference* im Reclipse Tracer beziehungsweise in der Instrumentierung aktiviert werden. Zur Offline-Analyse muss das Modul *Trace File Logger* aktiviert werden, damit der Trace während der Programmausführung aufgezeichnet wird. Der Trace kann anschließend durch die Verhaltensanalyse im Offline-Modus untersucht werden. Dazu ruft man im *Reclipse*-Menü *Start Behavioral Patterns Recognition...* auf (Abbildung B.9, Seite 204).

Abbildung B.19 zeigt den Dialog, der dadurch geöffnet wird. Hier muss man die Datei angeben, die die Annotationen aus der Strukturanalyse enthält, die Datei, die den Trace enthält, und die Datei, die den Verhaltensmusterkatalog enthält. Außerdem muss der Reverse-Engineer angeben, in welche Datei die Ergebnisse der Verhaltensanalyse geschrieben werden.

Nach der Durchführung der Verhaltensanalyse wird die Sicht *Behavioral Analysis Result* geöffnet (Abbildung B.20). Darin kann der Reverse-Engineer zu jedem Kandidaten, der ausgeführt wurde, die untersuchten Traces ansehen. Die Kandidaten kann man in der Kombobox unter *Design Pattern Candidates* auswählen. In der Mitte links ist die Variablenbindung des Strukturmusters an die Elemente der Struktur aufgelistet. Dies ist das Ergebnis der Strukturana-

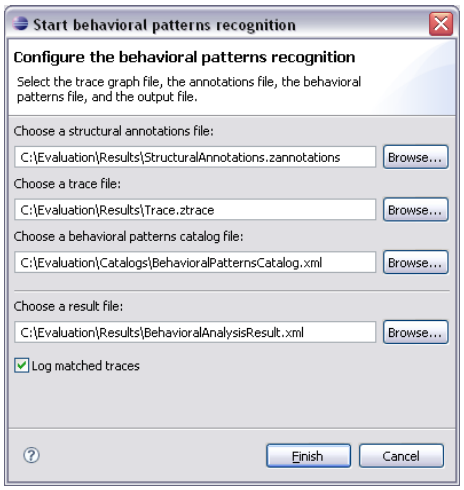

Abbildung B.19: Starten der Verhaltensanalyse im Offline-Modus

lyse. In der Mitte daneben sind die im Abschnitt 5.4 erklärten Messwerte der Verhaltensanalyse zu dem ausgewählten Kandidaten zu finden. Jeder der untersuchten Traces kann auch einzeln untersucht werden. Dazu wählt man rechts einen der Traces aus. Es wird angegeben, ob der Trace akzeptiert wurde, und die Bindung der Verhaltensmusterobjekte an die Instanzen zur Laufzeit wird in einer Tabelle aufgelistet. Im unteren Bereich wird der ausgewählte Trace als Sequenzdiagramm visualisiert. Bei Traces, die verworfen wurde, wird der letzte Methodenaufruf, der zum Verwerfen geführt hat, ebenfalls im Sequenzdiagramm visualisiert. So kann der Reverse-Engineer leicht feststellen, warum der Trace verworfen wurde. Um die Visualisierung der Traces zu ermöglichen, muss bei der Online-Analyse im Modul *Behavioral Patterns Inference* im Reclipse Tracer beziehungsweise bei der Instrumentierung die Eigenschaft *logTraces* auf *true* gesetzt werden. Bei der Offline-Analyse muss im Dialog aus Abbildung B.19 die Option *Log matched traces* aktiviert werden.

Das Ergebnis der Verhaltensanalyse kann auch unabhängig von der vorhergehenden Verhaltensanalyse betrachtet werden. Dazu kann man in der Sicht *Behavioral Analysis Result* das Ergebnis aus einer Datei laden. Dazu betätigt man die Schaltfläche mit dem geöffneten Ordner in der Toolbar der Sicht.

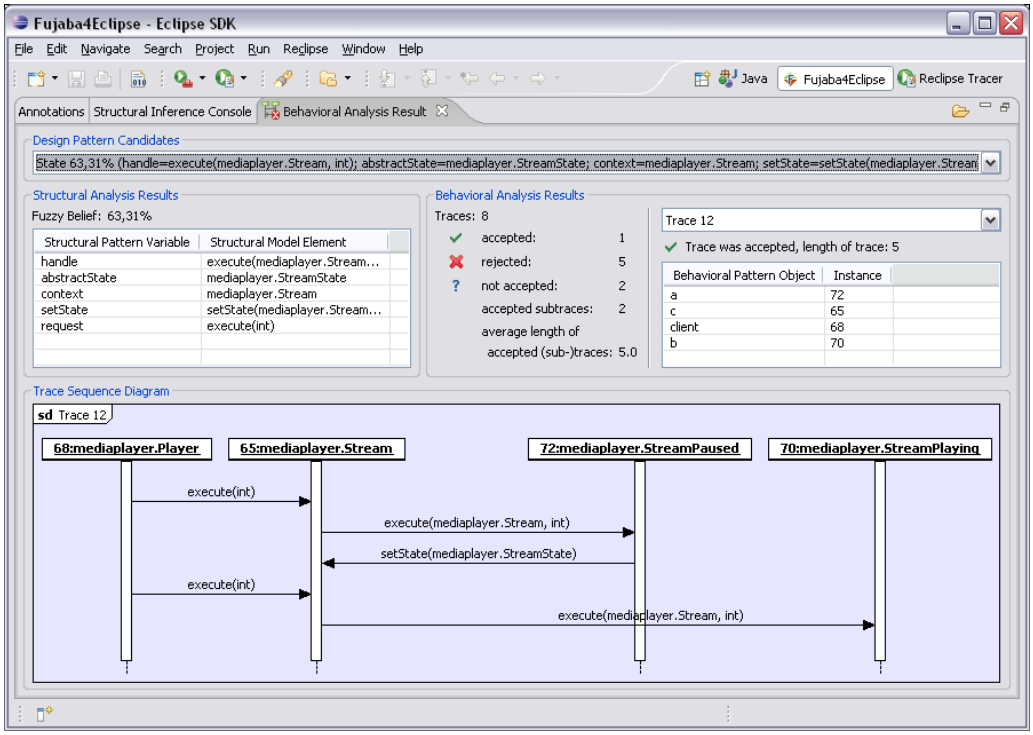

Abbildung B.20: Das Ergebnis der Verhaltensanalyse

Anhang C

Technische Dokumentation

C.1 Komponenten der Entwurfsmustererkennung

In diesem Kapitel werden die Komponenten des Werkzeugs RECLIPSE dokumentiert. Zu jeder Komponente werden ihre Funktion und ihre Version genannt. Optional werden ihre Abhängigkeiten von anderen Komponenten, die Komponente, die ihre Benutzungsschnittstelle implementiert, sowie ihre Ein- und Ausgaben angegeben. Einige der Komponenten sind zusätzlich von Komponenten des ECLIPSE-Frameworks abhängig. Diese Abhängigkeiten werden der besseren Übersichtlichkeit halber nicht genannt.

C.1.1 de.uni_paderborn.fujaba

Funktion

Dies ist die Hauptkomponente, in der die integrierte Entwicklungsumgebung FUJABA enthalten ist. Sie stellt außerdem die Pakete ClassDiagrams (Abbildung 2.1, Seite 18) zur Spezifikation von Klassendiagrammen und Structure zur Repräsentation der Struktur (Abbildung 2.2, Seite 19) bereit. Allerdings sind diese beiden Pakete in FUJABA aus dem in Abschnitt 2.3.1 genannten Grund identisch.

Abhängigkeiten

Benötigt die Komponenten de.uni_paderborn.runtimetools (Version 1.0.1), de.uni_paderborn.appindependent (Version 1.0.2), de.uni_kassel.coobra2 (Version 1.0.0), de.uni_kassel.features (Version 1.0.0) und de.uni_kassel.utils (Version 1.0.0).

Benutzungsschnittstelle

Die Benutzungsschnittstelle wird von der Komponente de.uni_paderborn.fujaba4eclipse (Version 0.7.0) bereitgestellt.

Version

5.0.4.20070313

C.1.2 org.reclipse.javaast

Funktion

Stellt das Paket JavaAST als Modell des abstrakten Syntaxbaums für JAVA-Methodenrümpfe zur Verfügung.

Version

2.0.1

C.1.3 org.reclipse.javaparser

Funktion

Stellt einen Parser zur Verfügung, der den JAVA-Quelltext des zu untersuchenden Softwaresystems in eine Struktur auf Basis des Strukturmodells umwandelt.

Abhängigkeiten

Verwendet das Strukturmodell des Pakets Structure aus der Komponente de.uni_paderborn.fujaba und das Modell des abstrakten Syntaxbaums aus der Komponente org.reclipse.javaast.

Eingabe

JAVA-Quelltext des zu untersuchenden Softwaresystems.

Ausgabe

Strukturmodell des zu untersuchenden Softwaresystems.

Benutzungsschnittstelle

Die Benutzungsschnittstelle wird von der Komponente org.reclipse.javaparser.ui (Version 1.0.0) bereitgestellt.

Version

4.0.2

C.1.4 org.reclipse.tracing

Funktion

Stellt das Paket Behavior (Abbildung 3.9, Seite 52) zur Repräsentation von Tracegraphen bereit. Stellt außerdem das Paket TraceDefinition zur Spezifikation der zu überwachenden Methoden eines Softwaresystems zur Verfügung.

Abhängigkeiten

Verwendet das Paket Structure aus der Komponente de.uni_paderborn.fujaba, um die Verbindung zwischen dem Verhaltens- und dem Strukturmodell herzustellen.

Version

1.0.0

C.1.5 org.reclipse.tracer

Funktion

Stellt ein Werkzeug zum Debugging von JAVA-Programmcode zur Verfügung. Das Werkzeug erzeugt Breakpoints zur Überwachung der Methodenaufrufe während der Ausführung des Programms. Die beobachteten Methodenaufrufe können entweder in eine Datei zur Offline-Analyse gespeichert werden, oder direkt in einer Online-Analyse der Verhaltensanalyse übergeben werden.

Abhängigkeiten

Verwendet das Paket TraceDefinition sowie das Paket Behavior aus der Komponente org.reclipse.tracing. Verwendet außerdem das Paket Annotations aus der Komponente org.reclipse.patterns.structure.inference.

Eingabe

Ausführbarer Programmcode des zu untersuchenden Softwaresystems, Kandidaten der Strukturanalyse und Spezifikation der zu überwachenden Methoden des Softwaresystems.

Ausgabe

Tracegraph der beobachteten Methodenaufrufe.

Benutzungsschnittstelle

Die Benutzungsschnittstelle wird von der Komponente org.reclipse.tracer.ui (Version 1.0.0) bereitgestellt.

Version

1.0.0

C.1.6 org.reclipse.instrumentation

Funktion

Stellt ein Werkzeug zur Instrumentierung von Java-Bytecode bereit. Das Werkzeug fügt zusätzlichen Programmcode ein, der zur Überwachung von Methodenaufrufen dient.

Abhängigkeiten

Verwendet das Paket TraceDefinition aus der Komponente org.reclipse.tracing. Verwendet außerdem das Paket Annotations aus der Komponente org.reclipse.patterns.structure.inference. Verwendet das Paket org.objectweb.asm (Version 3.0.0) zur Instrumentierung des Bytecodes.

Eingabe

Ausführbarer Programmcode des zu untersuchenden Softwaresystems, Kandidaten der Strukturanalyse und Spezifikation der zu überwachenden Methoden des Softwaresystems.

Ausgabe

Instrumentierter Programmcode des zu untersuchenden Softwaresystems.

Benutzungsschnittstelle

Die Benutzungsschnittstelle wird von der Komponente org.reclipse.instrumentation.ui (Version 1.0.0) bereitgestellt.

Version

1.0.0

C.1.7 org.reclipse.instrumentation.runtime

Funktion

Stellt die Laufzeitumgebung für den instrumentierten Programmcode zur Verfügung. Wird als Bibliothek dem instrumentierten Programmcode hinzugefügt. Die beobachteten Methodenaufrufe können entweder in eine Datei zur Offline-Analyse gespeichert werden, oder direkt in einer Online-Analyse der Verhaltensanalyse übergeben werden.

Abhängigkeiten

Verwendet das Paket Behavior aus der Komponente org.reclipse.tracing.

Eingabe

Der instrumentierte Programmcode meldet Methodenaufrufe.

Ausgabe

Tracegraph der beobachteten Methodenaufrufe.

Version

1.0.0

C.1.8 org.reclipse.patterns.structure.specification

Funktion:

Stellt das Paket StructuralPatterns (Abbildung 4.7, Seite 68) zur Spezifikation von Strukturmustern bereit.

Abhängigkeiten

Verwendet das Paket ClassDiagrams aus der Komponente de.uni_paderborn.fujaba, sowie das Paket JavaAST aus der Komponente org.reclipse.javaast, um das Metamodell der Strukturmuster mit dem Metamodell des Strukturmodells zu verbinden. Verwendet außerdem das Paket BehavioralPatterns aus der Komponente org.reclipse.patterns.behavior.specification, um das Metamodell der Strukturmuster mit dem Metamodell der Verhaltensmuster zu verbinden.

Ausgabe

Strukturmuster, spezifiziert durch den Reverse-Engineer.

Benutzungsschnittstelle

Die Benutzungsschnittstelle wird von der Komponente org.reclipse.patterns.structure.specification.ui (Version 2.0.0) bereitgestellt.

Version

3.0.0

C.1.9 org.reclipse.patterns.structure.inference

Funktion

Diese Komponente enthält den Algorithmus zur Strukturanalyse (Abbildung 2.10, Seite 28). Dazu verwendet sie Erkennungsmaschinen für Strukturmuster, um in der Struktur des zu untersuchenden Softwaresystems nach Kandidaten für Entwurfsmusterimplementierungen zu suchen. Sie stellt außerdem das Paket Annotations (Abbildung 5.16, Seite 132) zur Annotation der Kandidaten bereit. Für die Erkennungsmaschinen wird eine Schnittstelle definiert.

Abhängigkeiten

Verwendet das Paket Structure aus der Komponente de.uni_paderborn.fujaba und das Paket JavaAST aus der Komponente org.reclipse.javaast.

Eingabe

Das Strukturmodell und ein Katalog von Erkennungsmaschinen für Strukturmuster.

Ausgabe

Kandidaten für Entwurfsmusterimplementierungen in Form von Annotationen.

Benutzungsschnittstelle

Die Benutzungsschnittstelle wird von der Komponente org.reclipse.patterns.structure.inference.ui (Version 2.0.0) bereitgestellt.

Version

4.0.0

C.1.10 org.reclipse.patterns.structure.generator

Funktion

Stellt einen Algorithmus zur Verfügung, um aus den Strukturmustern Erkennungsmaschinen zu generieren.

Abhängigkeiten

Verwendet das Paket StructuralPatterns aus der Komponente org.reclipse.patterns.structure.specification. Verwendet außerdem die Schnittstellen für die Erkennungsmaschinen aus der Komponente org.reclipse.patterns.structure.inference.

Eingabe

Strukturmuster.

Ausgabe

Erkennungsmaschinen für Strukturmuster.

Benutzungsschnittstelle

Die Benutzungsschnittstelle wird von der Komponente org.reclipse.patterns.structure.generator.ui (Version 1.1.1) bereitgestellt.

Version

2.1.0

C.1.11 org.reclipse.patterns.behavior.specification

Funktion

Stellt das Paket BehavioralPatterns (Abbildung 4.5, Seite 64) zur Spezifikation der Verhaltensmuster bereit.

Abhängigkeiten

Verwendet das Paket StructuralPatterns aus der Komponente org.reclipse.patterns.structure.specification, um das Metamodell der Verhaltensmuster mit dem Metamodell der Strukturmuster zu verbinden. Benötigt außerdem die Komponenten de.uni_paderborn.fujaba.sequencediagrams (Version 1.0.0) und de.uni_paderborn.fujaba.sequencediagrams.ui (Version 1.0.0) .

Ausgabe

Verhaltensmuster, spezifiziert durch den Reverse-Engineer.

Benutzungsschnittstelle

Die Benutzungsschnittstelle wird von der Komponente org.reclipse.patterns.behavior.specification.ui (Version 1.0.0) bereitgestellt.

Version

1.0.0

C.1.12 org.reclipse.patterns.behavior.inference

Funktion

Diese Komponente enthält den Algorithmus zur Verhaltensanalyse (Abbildung 5.1, Seite 106). Sie verwendet endliche Automaten, um im Tracegraphen nach Verhaltensmustern zu suchen. Sie stellt außerdem die Pakete BehaviorAnalysis (Abbildung 5.2, Seite 111) und Automaton (Abbildung 5.3, Seite 113) zur Verfügung.

Abhängigkeiten

Verwendet die durch das Paket Behavior aus der Komponente org.reclipse.tracing repräsentierten und von den Komponenten org.reclipse.tracer oder org.reclipse.instrumentation.runtime erzeugten Tracegraphen. Verwendet außerdem das Paket Annotations aus der Komponente org.reclipse.patterns.structure.inference.

Eingabe

Kandidaten der Strukturanalyse, ein Tracegraph und ein Katalog von Verhaltensmustern.

Ausgabe

Bewertung der Konformität der Kandidaten zu den Verhaltensmustern.

Benutzungsschnittstelle

Die Benutzungsschnittstelle wird von der Komponente org.reclipse.patterns.behavior.inference.ui (Version 1.0.0) bereitgestellt.

Version

1.0.0

C.1.13 org.reclipse.patterns.behavior.generator

Funktion

Stellt den Algorithmus aus Abschnitt 4.4 zur Verfügung, um aus den Verhaltensmustern endliche Automaten zu generieren. Erzeugt außerdem aus dem

Ergebnis der Strukturanalyse die Trace-Definition zur Überwachung des Programms.

Abhängigkeiten

Verwendet das Paket BehavioralPatterns aus der Komponente org.reclipse.patterns.behavior.specification und das Paket Automaton aus der Komponente org.reclipse.patterns.behavior.inference.

Eingabe

Verhaltensmuster.

Ausgabe

Deterministische, endliche Automaten zur Erkennung der Verhaltensmuster.

Benutzungsschnittstelle

Die Benutzungsschnittstelle wird von der Komponente org.reclipse.patterns.behavior.generator.ui (Version 1.0.0) bereitgestellt.

Version

1.0.0

C.2 Datenformate der Komponenten

Im Folgenden werden die wichtigsten Datenformate der zuvor vorgestellten Komponenten spezifiziert. Dazu werden jeweils eine *Document Type Definition* und ein passendes Beispiel jeweils in XML-Syntax angegeben.

C.2.1 Annotationen

Für jeden in der Strukturanalyse identifizierten Kandidaten gibt es eine Variablenbindung des Strukturmusters an Elemente der Struktur. Diese Variablenbindung ist in einer Annotation festgehalten. Annotationen werden entweder als einfache Textdatei mit dem Suffix *.xannotations* oder als komprimierte Datei im ZIP-Format mit dem Suffix *.zannotations* gespeichert.

Die komprimierte Datei enthält einen Eintrag mit dem Namen *StructuralAnnotations.xannotations*. Die Annotationen sind Ausgabe der Komponente org.reclipse.patterns.structure.inference und Eingabe der Komponente org.reclipse.patterns.behavior.inference.

Document Type Definition

Die Variablenbindung ist in den Elementen des Typs BoundObject festgehalten. Das Attribut key ist der Variablenname aus dem Strukturmuster, name der Name des Strukturelements.

```
<!ELEMENT StructuralAnnotations (StructuralAnnotation*)>

<!ELEMENT StructuralAnnotation (BoundObject*)>
<!ATTLIST StructuralAnnotation  name        CDATA #REQUIRED
                                fuzzyBelief CDATA #IMPLIED>

<!ELEMENT BoundObject EMPTY>
<!ATTLIST BoundObject key  CDATA #REQUIRED
                      name CDATA #REQUIRED>
```

Beispiel

In diesem Beispiel sind die Annotationen eines *Observer*-, eines *Strategy*- und eines *State*-Kandidaten aus dem Mediaplayer gespeichert.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE StructuralAnnotations SYSTEM "http://wwwcs.
     uni-paderborn.de/cs/fujaba/DTDs/StructuralAnnotations.dtd">
<StructuralAnnotations>
  <StructuralAnnotation name="Observer" fuzzyBelief="17.12">
    <BoundObject key="update" name="Stream()"/>
    <BoundObject key="register"
                 name="execute(mediaplayer.Stream, int)"/>
    <BoundObject key="subject" name="mediaplayer.StreamState"/>
    <BoundObject key="observerClass" name="mediaplayer.Stream"/>
    <BoundObject key="getState" name="run(mediaplayer.Stream)"/>
  </StructuralAnnotation>

  <StructuralAnnotation name="Strategy" fuzzyBelief="62.20">
```

```
    <BoundObject key="abstractStrategy"
                 name="mediaplayer.StreamState"/>
    <BoundObject key="algorithm"
                 name="execute(mediaplayer.Stream, int)"/>
    <BoundObject key="request" name="execute(int)"/>
    <BoundObject key="context" name="mediaplayer.Stream"/>
    <BoundObject key="setStrategy"
                 name="setState(mediaplayer.StreamState)"/>
  </StructuralAnnotation>

  <StructuralAnnotation name="State" fuzzyBelief="63.31">
    <BoundObject key="handle"
                 name="execute(mediaplayer.Stream, int)"/>
    <BoundObject key="setState"
                 name="setState(mediaplayer.StreamState)"/>
    <BoundObject key="abstractState"
                 name="mediaplayer.StreamState"/>
    <BoundObject key="context" name="mediaplayer.Stream"/>
    <BoundObject key="request" name="execute(int)"/>
  </StructuralAnnotation>
</StructuralAnnotations>
```

C.2.2 Trace-Definition

Die Trace-Definition ist wie die Annotationen ebenfalls Ergebnis der Strukturanalyse. In ihr wird festgehalten, welche Methodenaufrufe welcher Klassen zur Laufzeit beobachtet werden müssen. Die Trace-Definition wird entweder als einfache Textdatei mit dem Suffix *.xtracedefinition* oder als komprimierte Datei im ZIP-Format mit dem Suffix *.ztracedefinition* gespeichert. Die komprimierte Datei enthält einen Eintrag mit dem Namen *TraceDefinition.xtracedefinition*. Die Trace-Definition ist Ausgabe der Komponente org.reclipse.patterns.behavior.generator und Eingabe der Komponenten org.reclipse.tracer und org.reclipse.instrumentation.

Document Type Definition

Eine Trace-Definition besteht aus zwei Bereichen, einem CriticalTrace und einem ConsiderTrace. Im CriticalTrace werden nur Klassen genannt. Von diesen Klassen werden alle Methodenaufrufe überwacht. Im ConsiderTrace sind da-

gegen Klassen aufgeführt, von denen nur die gegebenen Methoden überwacht werden sollen.

```
<!ELEMENT TraceDefinition (CriticalTrace?, ConsiderTrace?)>

<!ELEMENT CriticalTrace (CriticalClass+)>

<!ELEMENT CriticalClass (CallerClass*)>
<!ATTLIST CriticalClass name CDATA #REQUIRED>

<!ELEMENT CallerClass EMPTY>
<!ATTLIST CallerClass name CDATA #REQUIRED>

<!ELEMENT ConsiderTrace (ConsiderClass+)>

<!ELEMENT ConsiderClass (ConsiderMethod+)>
<!ATTLIST ConsiderClass name CDATA #REQUIRED>

<!ELEMENT ConsiderMethod (Parameter*, CallerClass*)>
<!ATTLIST ConsiderMethod name CDATA #REQUIRED>

<!ELEMENT Parameter EMPTY>
<!ATTLIST Parameter type CDATA #REQUIRED>
```

Beispiel

Das Beispiel zeigt die Trace-Definition des Mediaplayers.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE TraceDefinition SYSTEM "http://wwwcs.uni-paderborn.de/
                            cs/fujaba/DTDs/TraceDefinition.dtd">
<TraceDefinition>
  <ConsiderTrace>
    <ConsiderClass name="mediaplayer.Stream">
      <ConsiderMethod name="Stream"/>
      <ConsiderMethod name="setState">
        <Parameter type="mediaplayer.StreamState"/>
      </ConsiderMethod>
      <ConsiderMethod name="execute">
```

```
        <Parameter type="int"/>
      </ConsiderMethod>
    </ConsiderClass>

    <ConsiderClass name="mediaplayer.StreamPlaying">
      <ConsiderMethod name="execute">
        <Parameter type="mediaplayer.Stream"/>
        <Parameter type="int"/>
      </ConsiderMethod>
    </ConsiderClass>

    <ConsiderClass name="mediaplayer.StreamStopped">
      <ConsiderMethod name="execute">
        <Parameter type="mediaplayer.Stream"/>
        <Parameter type="int"/>
      </ConsiderMethod>
    </ConsiderClass>

    <ConsiderClass name="mediaplayer.StreamPaused">
      <ConsiderMethod name="execute">
        <Parameter type="mediaplayer.Stream"/>
        <Parameter type="int"/>
      </ConsiderMethod>
    </ConsiderClass>

    <ConsiderClass name="mediaplayer.StreamState">
      <ConsiderMethod name="execute">
        <Parameter type="mediaplayer.Stream"/>
        <Parameter type="int"/>
      </ConsiderMethod>
      <ConsiderMethod name="run">
        <Parameter type="mediaplayer.Stream"/>
      </ConsiderMethod>
    </ConsiderClass>
  </ConsiderTrace>
</TraceDefinition>
```

C.2.3 Tracegraph

Der Tracegraph enthält alle aufgezeichneten Methodenaufrufe einer Programmausführung. Ein Tracegraph wird entweder als einfache Textdatei mit dem Suffix *.xtrace* oder als komprimierte Datei im ZIP-Format mit dem Suffix *.ztrace* gespeichert. Die komprimierte Datei enthält einen Eintrag mit dem Namen *Trace.xtrace*. Der Tracegraph ist Ausgabe der Komponenten org.reclipse.tracer und org.reclipse.instrumentation und Eingabe der Komponente org.reclipse.patterns.behavior.inference.

Document Type Definition

Im Tracegraph werden der Start des Programms, das Laden einer Klasse, das Starten und Beenden eines Methodenaufrufs, sowie das Beenden des Programms als Ereignisse festgehalten. Wird eine überwachte Klasse geladen, so wird im Element ClassLoaded auch die Vererbungshierarchie der Klasse aufgezeichnet, um bei der Verhaltensanalyse polymorphe Methodenaufrufe erkennen zu können.

```
<!ELEMENT Trace (ProcessStart,
                 (ClassLoaded|MethodEntry|MethodExit)*,
                 ProcessEnd)>
<!ATTLIST Trace mainclass CDATA #IMPLIED
                date      CDATA #IMPLIED>

<!ELEMENT ProcessStart EMPTY>
<!ATTLIST ProcessStart name CDATA #REQUIRED
                       time CDATA #IMPLIED>

<!ELEMENT ClassLoaded (SuperType*)>
<!ATTLIST ClassLoaded name CDATA #REQUIRED>

<!ELEMENT SuperType EMPTY>
<!ATTLIST SuperType name CDATA #REQUIRED>

<!ELEMENT MethodEntry (Caller, Callee, Argument*)>
<!ATTLIST MethodEntry id     CDATA #REQUIRED
                      name   CDATA #REQUIRED
                      thread CDATA #IMPLIED
                      time   CDATA #IMPLIED>
```

```
<!ELEMENT Caller EMPTY>
<!ATTLIST Caller id   CDATA #REQUIRED
                 type CDATA #REQUIRED>

<!ELEMENT Callee EMPTY>
<!ATTLIST Callee id   CDATA #REQUIRED
                 type CDATA #REQUIRED>

<!ELEMENT Argument EMPTY>
<!ATTLIST Argument value CDATA #IMPLIED
                   id    CDATA #IMPLIED
                   type  CDATA #REQUIRED>

<!ELEMENT MethodExit EMPTY>
<!ATTLIST MethodExit id CDATA #REQUIRED>

<!ELEMENT ProcessEnd EMPTY>
<!ATTLIST ProcessEnd time CDATA #IMPLIED>
```

Beispiel

Das Beispiel zeigt einen kleinen Ausschnitt aus dem Tracegraphen, der während der Ausführung des Mediaplayers aufgezeichnet wurde.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE BehavioralPatternsCatalog SYSTEM "http://wwwcs.
                 uni-paderborn.de/cs/fujaba/DTDs/Trace.dtd">

<Trace mainclass="mediaplayer.Player"
       date="Mon Jan 29 17:14:34 CET 2007">

  <ProcessStart name="main"/>

  <ClassLoaded name="mediaplayer.Stream">
  </ClassLoaded>

  <ClassLoaded name="mediaplayer.StreamState">
  </ClassLoaded>
```

```
  <ClassLoaded name="mediaplayer.StreamStopped">
    <SuperType name="mediaplayer.StreamState"/>
  </ClassLoaded>

  <MethodEntry id="1" name="setState" thread="main">
    <Caller id="66" type="mediaplayer.Stream"/>
    <Callee id="66" type="mediaplayer.Stream"/>
    <Argument id="68" type="mediaplayer.StreamState"/>
  </MethodEntry>

  <MethodEntry id="2" name="run" thread="main">
    <Caller id="66" type="mediaplayer.Stream"/>
    <Callee id="68" type="mediaplayer.StreamStopped"/>
    <Argument id="66" type="mediaplayer.Stream"/>
  </MethodEntry>

  <MethodEntry id="3" name="execute" thread="main">
    <Caller id="69" type="mediaplayer.Player"/>
    <Callee id="66" type="mediaplayer.Stream"/>
    <Argument value="1" type="int"/>
  </MethodEntry>

  <MethodEntry id="4" name="execute" thread="main">
    <Caller id="66" type="mediaplayer.Stream"/>
    <Callee id="68" type="mediaplayer.StreamStopped"/>
    <Argument id="66" type="mediaplayer.Stream"/>
    <Argument value="1" type="int"/>
  </MethodEntry>

...

  <ProcessEnd/>

</Trace>
```

C.2.4 Verhaltensmusterkatalog

Im Verhaltensmusterkatalog werden die endlichen Automaten gespeichert, die aus den Verhaltensmustern generiert wurden. Er ist Ausgabe der Komponente org.reclipse.patterns.behavior.generator und Eingabe der Komponente org.reclipse.patterns.behavior.inference.

Document Type Definition

Zu jedem Verhaltensmuster gibt es einen DFA und einen Trigger, also eine Methode, deren Aufruf die Analyse eines Traces auslöst. Der DFA besteht aus den Symbolen des Eingabealphabets, mehreren Zuständen und Transitionen zwischen den Zuständen und einem expliziten Startzustand. Der Typ eines Zustands ist in dem Attribut type festgehalten. Das Attribut type eines nicht-akzeptierenden Zustands hat den Wert 0, eines akzeptierenden Zustands den Wert 1 und eines verwerfenden Zustands den Wert 2.

```
<!ELEMENT BehavioralPatternsCatalog (BehavioralPatternEntry*)>

<!ELEMENT BehavioralPatternEntry (DFA, Trigger+)>
<!ATTLIST BehavioralPatternEntry name     CDATA #REQUIRED
                                 negative CDATA #IMPLIED>

<!ELEMENT DFA ((PermittedMethodCallSymbol|
                ProhibitedMethodCallSymbol|
                ProhibitedCallerSymbol)+, State+, Transition*,
                StartState)>

<!ELEMENT PermittedMethodCallSymbol (Caller, Callee)>
<!ATTLIST PermittedMethodCallSymbol id         CDATA #REQUIRED
                                    methodName CDATA #REQUIRED>

<!ELEMENT ProhibitedMethodCallSymbol (Callee)>
<!ATTLIST ProhibitedMethodCallSymbol id         CDATA #REQUIRED
                                     methodName CDATA #REQUIRED>

<!ELEMENT ProhibitedCallerSymbol (PermittedCaller+, Callee)>
<!ATTLIST ProhibitedCallerSymbol id         CDATA #REQUIRED
                                 methodName CDATA #REQUIRED>
```

```
<!ELEMENT Caller EMPTY>
<!ATTLIST Caller name CDATA #REQUIRED
                 type CDATA #IMPLIED>

<!ELEMENT Callee EMPTY>
<!ATTLIST Callee name CDATA #REQUIRED
                 type CDATA #REQUIRED>

<!ELEMENT PermittedCaller EMPTY>
<!ATTLIST PermittedCaller name CDATA #REQUIRED
                          type CDATA #IMPLIED>

<!ELEMENT State EMPTY>
<!ATTLIST State id     CDATA #REQUIRED
                name   CDATA #IMPLIED
                type   CDATA #REQUIRED>

<!ELEMENT Transition EMPTY>
<!ATTLIST Transition previousStateId CDATA #REQUIRED
                     nextStateId     CDATA #REQUIRED
                     symbolId        CDATA #REQUIRED>

<!ELEMENT StartState EMPTY>
<!ATTLIST StartState id   CDATA #REQUIRED>

<!ELEMENT Trigger EMPTY>
<!ATTLIST Trigger callerType CDATA #IMPLIED
                  calleeType CDATA #REQUIRED
                  methodName CDATA #REQUIRED>
```

Beispiel

Das Beispiel ist nur ein kleiner Ausschnitt aus dem Verhaltensmusterkatalog, der für die Analyse des Mediaplayers verwendet wurde. Es enthält lediglich die Definition des endlichen Automaten für das *State*-Verhaltensmuster.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE BehavioralPatternsCatalog SYSTEM "http://wwwcs.
 uni-paderborn.de/cs/fujaba/DTDs/BehavioralPatternsCatalog.dtd">
```

```
<BehavioralPatternsCatalog>
  <BehavioralPatternEntry name="State" negative="false">
    <DFA>
      <PermittedMethodCallSymbol id="symbol19"
                                 methodName="setState">
        <Caller name="client"/>
        <Callee name="c" type="context"/>
      </PermittedMethodCallSymbol>

      <PermittedMethodCallSymbol id="symbol20"
                                 methodName="request">
        <Caller name="client"/>
        <Callee name="c" type="context"/>
      </PermittedMethodCallSymbol>

      <PermittedMethodCallSymbol id="symbol21"
                                 methodName="handle">
        <Caller name="c" type="context"/>
        <Callee name="a" type="abstractState"/>
      </PermittedMethodCallSymbol>

      <PermittedMethodCallSymbol id="symbol22"
                                 methodName="setState">
        <Caller name="c" type="context"/>
        <Callee name="c" type="context"/>
      </PermittedMethodCallSymbol>

      <PermittedMethodCallSymbol id="symbol23"
                                 methodName="setState">
        <Caller name="a" type="abstractState"/>
        <Callee name="c" type="context"/>
      </PermittedMethodCallSymbol>

      <PermittedMethodCallSymbol id="symbol24"
                                 methodName="handle">
        <Caller name="c" type="context"/>
        <Callee name="b" type="abstractState"/>
      </PermittedMethodCallSymbol>
```

```
<ProhibitedMethodCallSymbol id="symbol25"
                            methodName="setState">
  <Callee name="c" type="context"/>
</ProhibitedMethodCallSymbol>

<ProhibitedMethodCallSymbol id="symbol26"
                            methodName="handle">
  <Callee name="a" type="abstractState"/>
</ProhibitedMethodCallSymbol>

<ProhibitedMethodCallSymbol id="symbol27"
                            methodName="handle">
  <Callee name="b" type="abstractState"/>
</ProhibitedMethodCallSymbol>

<ProhibitedMethodCallSymbol id="symbol28"
                            methodName="request">
  <Callee name="c" type="context"/>
</ProhibitedMethodCallSymbol>

<ProhibitedCallerSymbol id="symbol29"
                        methodName="request">
  <PermittedCaller name="client"/>
  <Callee name="c" type="context"/>
</ProhibitedCallerSymbol>

<ProhibitedCallerSymbol id="symbol30" methodName="handle">
  <PermittedCaller name="c" type="context"/>
  <Callee name="a" type="abstractState"/>
</ProhibitedCallerSymbol>

<ProhibitedCallerSymbol id="symbol31"
                        methodName="setState">
  <PermittedCaller name="c" type="context"/>
  <PermittedCaller name="a" type="abstractState"/>
  <Callee name="c" type="context"/>
</ProhibitedCallerSymbol>
```

```
<ProhibitedCallerSymbol id="symbol32" methodName="handle">
  <PermittedCaller name="c" type="context"/>
  <Callee name="b" type="abstractState"/>
</ProhibitedCallerSymbol>

<State id="state13" name="0,1" type="0"/>
<State id="state14" name="1" type="0"/>
<State id="state15" name="2" type="0"/>
<State id="state16" name="4,3,1" type="0"/>
<State id="state17" name="5,6" type="0"/>
<State id="state18" name="5,7" type="0"/>
<State id="state19" name="8" type="0"/>
<State id="state20" name="5,9,10" type="1"/>
<State id="state21" name="R" type="2"/>

<Transition previousStateId="state17"
            nextStateId="state21" symbolId="symbol29"/>
<Transition previousStateId="state14"
            nextStateId="state21" symbolId="symbol27"/>
<Transition previousStateId="state15"
            nextStateId="state21" symbolId="symbol28"/>
<Transition previousStateId="state16"
            nextStateId="state21" symbolId="symbol29"/>
<Transition previousStateId="state13"
            nextStateId="state15" symbolId="symbol20"/>
<Transition previousStateId="state17"
            nextStateId="state21" symbolId="symbol26"/>
<Transition previousStateId="state19"
            nextStateId="state21" symbolId="symbol32"/>
<Transition previousStateId="state16"
            nextStateId="state21" symbolId="symbol27"/>
<Transition previousStateId="state15"
            nextStateId="state21" symbolId="symbol30"/>
<Transition previousStateId="state18"
            nextStateId="state21" symbolId="symbol29"/>
<Transition previousStateId="state16"
            nextStateId="state18" symbolId="symbol23"/>
<Transition previousStateId="state18"
            nextStateId="state21" symbolId="symbol25"/>
```

```
<Transition previousStateId="state16"
            nextStateId="state21" symbolId="symbol26"/>
<Transition previousStateId="state19"
            nextStateId="state20" symbolId="symbol24"/>
<Transition previousStateId="state14"
            nextStateId="state15" symbolId="symbol20"/>
<Transition previousStateId="state16"
            nextStateId="state21" symbolId="symbol31"/>
<Transition previousStateId="state15"
            nextStateId="state21" symbolId="symbol27"/>
<Transition previousStateId="state18"
            nextStateId="state19" symbolId="symbol20"/>
<Transition previousStateId="state18"
            nextStateId="state21" symbolId="symbol26"/>
<Transition previousStateId="state14"
            nextStateId="state21" symbolId="symbol26"/>
<Transition previousStateId="state15"
            nextStateId="state16" symbolId="symbol21"/>
<Transition previousStateId="state19"
            nextStateId="state21" symbolId="symbol28"/>
<Transition previousStateId="state20"
            nextStateId="state19" symbolId="symbol20"/>
<Transition previousStateId="state14"
            nextStateId="state21" symbolId="symbol25"/>
<Transition previousStateId="state18"
            nextStateId="state21" symbolId="symbol27"/>
<Transition previousStateId="state17"
            nextStateId="state21" symbolId="symbol25"/>
<Transition previousStateId="state17"
            nextStateId="state21" symbolId="symbol27"/>
<Transition previousStateId="state13"
            nextStateId="state14" symbolId="symbol19"/>
<Transition previousStateId="state19"
            nextStateId="state21" symbolId="symbol26"/>
<Transition previousStateId="state17"
            nextStateId="state19" symbolId="symbol20"/>
<Transition previousStateId="state19"
            nextStateId="state21" symbolId="symbol25"/>
<Transition previousStateId="state14"
```

```
                nextStateId="state21" symbolId="symbol29"/>
      <Transition previousStateId="state16"
                nextStateId="state15" symbolId="symbol20"/>
      <Transition previousStateId="state15"
                nextStateId="state21" symbolId="symbol25"/>
      <Transition previousStateId="state16"
                nextStateId="state17" symbolId="symbol22"/>

      <StartState id="state13"/>

    </DFA>

    <Trigger calleeType="context" methodName="setState"/>
    <Trigger calleeType="context" methodName="request"/>

  </BehavioralPatternEntry>

  ...

</BehavioralPatternsCatalog>
```

C.2.5 Ergebnis der struktur- und verhaltensbasierten Entwurfsmustererkennung

Das Ergebnis der struktur- und verhaltensbasierten Entwurfsmustererkennung ist Ausgabe der Komponente org.reclipse.patterns.behavior.inference. Sie kann nachträglich im Werkzeug RECLIPSE in der Sicht *Behavioral Analysis Result* geladen und vom Reverse-Engineer ausgewertet werden.

Document Type Definition

Zu jedem Kandidaten, zu dem Traces analysiert wurden, gibt es ein Element Annotation, das das Ergebnis der Strukturanalyse im Element StructuralAnnotation und das Ergebnis der Verhaltensanalyse im Element BehavioralAnnotation enthält. Zusätzlich können in dem Ergebnis der Verhaltensanalyse auch die untersuchten Traces enthalten sein.

```
<!ELEMENT BehavioralAnalysisResult (Annotation*)>
<!ATTLIST BehavioralAnalysisResult date CDATA #IMPLIED>
```

```
<!ELEMENT Annotation (StructuralAnnotation,
                      BehavioralAnnotation)>
<!ATTLIST Annotation type CDATA #REQUIRED>

<!ELEMENT StructuralAnnotation (BoundObject*)>
<!ATTLIST StructuralAnnotation fuzzyBelief CDATA #IMPLIED>

<!ELEMENT BoundObject EMPTY>
<!ATTLIST BoundObject key  CDATA #REQUIRED
                      name CDATA #REQUIRED>

<!ELEMENT BehavioralAnnotation (Trace*)>
<!ATTLIST BehavioralAnnotation traces           CDATA #REQUIRED
                    acceptedTraces              CDATA #REQUIRED
                    notAcceptedTraces           CDATA #REQUIRED
                    rejectedTraces              CDATA #REQUIRED
                    passedAcceptingStateTraces CDATA #REQUIRED
                    avgLengthAcceptedTraces     CDATA #REQUIRED>

<!ELEMENT Trace (BoundObject+, MethodCall+)>
<!ATTLIST Trace id                    CDATA #REQUIRED
                result                CDATA #REQUIRED
                passedAcceptingState  CDATA #REQUIRED
                lengthOfAcceptedTrace CDATA #REQUIRED>

<!ELEMENT MethodCall (Caller, Callee, Argument*)>
<!ATTLIST MethodCall id     CDATA #REQUIRED
                     name   CDATA #REQUIRED>

<!ELEMENT Caller EMPTY>
<!ATTLIST Caller id   CDATA #REQUIRED
                 type CDATA #REQUIRED>

<!ELEMENT Callee EMPTY>
<!ATTLIST Callee id   CDATA #REQUIRED
                 type CDATA #REQUIRED>

<!ELEMENT Argument EMPTY>
```

```
<!ATTLIST Argument value CDATA #IMPLIED
                   id    CDATA #IMPLIED
                   type  CDATA #REQUIRED>
```

Beispiel

Das Beispiel zeigt einen kleinen Ausschnitt aus dem Ergebnis der Analyse des Mediaplayers.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE BehavioralAnalysisResult SYSTEM "http://wwwcs.
  uni-paderborn.de/cs/fujaba/DTDs/BehavioralAnalysisResult.dtd">

<BehavioralAnalysisResult date="Thu Mar 08 16:52:49 CET 2007">

  <Annotation type="State">

    <StructuralAnnotation fuzzyBelief="63.31360076101584">
      <BoundObject key="handle"
                   name="execute(mediaplayer.Stream, int)"/>
      <BoundObject key="abstractState"
                   name="mediaplayer.StreamState"/>
      <BoundObject key="context" name="mediaplayer.Stream"/>
      <BoundObject key="setState"
                   name="setState(mediaplayer.StreamState)"/>
      <BoundObject key="request" name="execute(int)"/>
    </StructuralAnnotation>

    <BehavioralAnnotation traces="8" acceptedTraces="1"
               notAcceptedTraces="2" rejectedTraces="5"
               passedAcceptingStateTraces="3"
               avgLengthAcceptedTraces="5.0">

      <Trace id="0" result="2" passedAcceptingState="1"
             lengthOfAcceptedTrace="5">

        <BoundObject key="a" name="67"/>
        <BoundObject key="c" name="65"/>
        <BoundObject key="client" name="68"/>
```

```
<BoundObject key="b" name="70"/>

<MethodCall id="2" name="execute">
  <Caller id="68" typeName="mediaplayer.Player"/>
  <Callee id="65" typeName="mediaplayer.Stream"/>
  <Argument typeName="int"/>
</MethodCall>

<MethodCall id="3" name="execute">
  <Caller id="65" typeName="mediaplayer.Stream"/>
  <Callee id="67" typeName="mediaplayer.StreamStopped"/>
  <Argument typeName="mediaplayer.Stream"/>
  <Argument typeName="int"/>
</MethodCall>

<MethodCall id="4" name="setState">
  <Caller id="67" typeName="mediaplayer.StreamStopped"/>
  <Callee id="65" typeName="mediaplayer.Stream"/>
  <Argument typeName="mediaplayer.StreamState"/>
</MethodCall>

<MethodCall id="6" name="execute">
  <Caller id="68" typeName="mediaplayer.Player"/>
  <Callee id="65" typeName="mediaplayer.Stream"/>
  <Argument typeName="int"/>
</MethodCall>

<MethodCall id="7" name="execute">
  <Caller id="65" typeName="mediaplayer.Stream"/>
  <Callee id="70" typeName="mediaplayer.StreamPlaying"/>
  <Argument typeName="mediaplayer.Stream"/>
  <Argument typeName="int"/>
</MethodCall>

<MethodCall id="10" name="execute">
  <Caller id="68" typeName="mediaplayer.Player"/>
  <Callee id="65" typeName="mediaplayer.Stream"/>
  <Argument typeName="int"/>
</MethodCall>
```

```
        <MethodCall id="12" name="setState">
          <Caller id="72" typeName="mediaplayer.StreamPaused"/>
          <Callee id="65" typeName="mediaplayer.Stream"/>
          <Argument typeName="mediaplayer.StreamState"/>
        </MethodCall>

      </Trace>

      ...
    </BehavioralAnnotation>

  </Annotation>

</BehavioralAnalysisResult>
```

Abbildungen

Tabellen

Index

The Power of Procrastination

Während meines Aufenthaltes am Georgia Institute of Technology, Atlanta, USA im Herbst 2005 machte mich ein Freund auf die Internetseite www.phdcomics.com aufmerksam. Auf dieser Seite werden unter dem Motto „The Power of Procrastination“ in regelmäßigen Abständen die „Piled Higher and Deeper“ Comics publiziert. Sie werden von Jorge Cham, einem ehemaligen PHD-Studenten, gezeichnet und handeln vom Alltag einiger fiktiver PHD-Studenten[1]. Die Comics spiegeln jedoch in erstaunlicher Präzision Erfahrungen und Anekdoten wider, die man in seiner Zeit als Doktorand erlebt. Während der letzten beiden Jahre waren sie für mich immer wieder eine willkommene Abwechslung von der Arbeit an dieser Dissertation. Ich möchte deshalb mit einigen ausgewählten Comics, die mir besonders gefallen haben, meine Dissertation schließen.

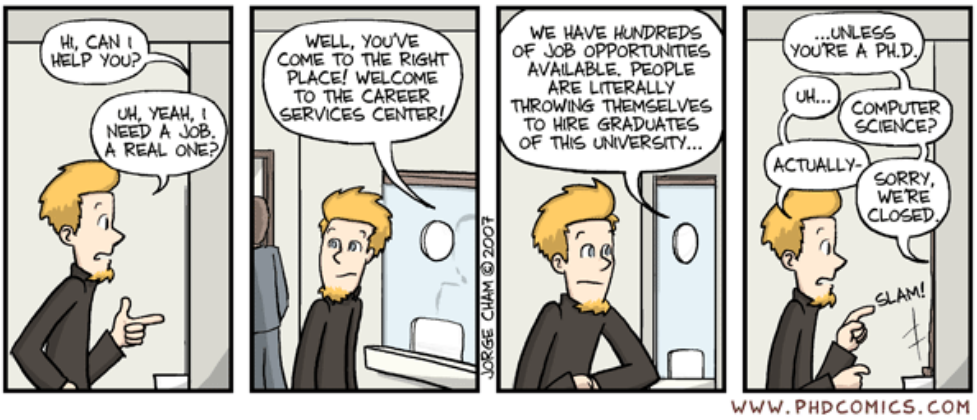

Abbildung C.1: Career Services

Allen derzeitigen Doktoranden und Studenten, die eine Promotion in Informatik in Betracht ziehen, kann ich aus eigener Erfahrung versichern, dass die

[1]Bezeichnung für Promotions-Studenten in den USA

Situation nach der Promotion nicht ganz so schlimm ist, wie in Abbildung C.1 dargestellt[2]. Es hat sich gelohnt.

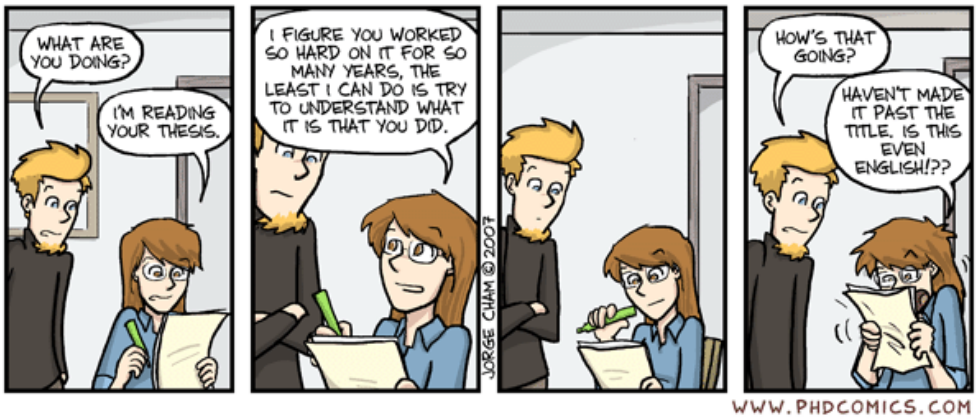

Abbildung C.2: Is This Even English?

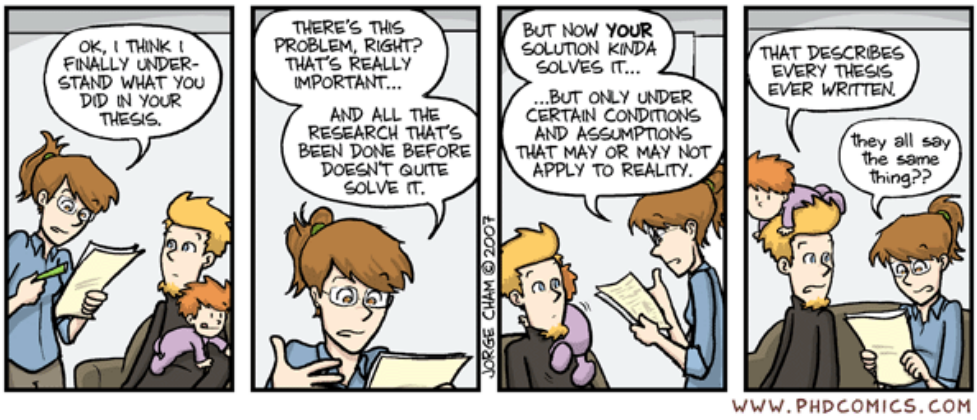

Abbildung C.3: May Or May Not Apply To Reality

[2]Abbildungen C.1, C.2 und C.3: ©„Piled Higher and Deeper“ by Jorge Cham, www.phdcomics.com