



**UNIVERSITÄT PADERBORN**

*Die Universität der Informationsgesellschaft*

Fakultät für Elektrotechnik, Informatik und Mathematik

Institut für Informatik

Fachgebiet Softwaretechnik

Warburger Straße 100

33098 Paderborn

# Berücksichtigung von Objektmengen bei der dynamischen Entwurfsmustererkennung

Bachelor-Arbeit

im Rahmen des Studiengangs Informatik

zur Erlangung des Grades

Bachelor of Science

von

MARIE CHRISTIN PLATENIUS

Im Spiringsfelde 9

33098 Paderborn

vorgelegt bei

Prof. Dr. Wilhelm Schäfer

und

Prof. Dr. Hans Kleine Büning

Paderborn, Oktober 2009



# Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

---

Ort, Datum

---

Unterschrift



# Inhalt

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Problemstellung . . . . .	2
1.2	Vorgehensweise und Lösungsansatz . . . . .	3
1.3	Struktur der Arbeit . . . . .	4
<b>2</b>	<b>Grundlagen</b>	<b>7</b>
2.1	Probleme der Entwurfsmustererkennung . . . . .	7
2.2	Struktur- und verhaltensbasierte Entwurfsmustererkennung in RE- CLIPSE . . . . .	7
2.2.1	Strukturanalyse . . . . .	8
2.2.2	Verhaltensanalyse . . . . .	10
<b>3</b>	<b>Konzept der Erweiterungen</b>	<b>21</b>
3.1	Problemstellung . . . . .	21
3.2	Erweiterung der Spezifikationsprache . . . . .	23
3.2.1	Mengenobjekte . . . . .	23
3.2.2	Each-Fragment . . . . .	24
3.2.3	Selbstaufrufe . . . . .	25
3.2.4	Argumente . . . . .	25
3.2.5	Zuweisungen . . . . .	25
3.3	Beispiel-Verhaltensmuster . . . . .	26
3.3.1	Observer . . . . .	27
3.3.2	Chain of Responsibility . . . . .	28
3.3.3	State . . . . .	28
3.4	Erweiterung der Transformation in Automaten . . . . .	30
3.4.1	Transformationsregeln . . . . .	30
3.4.2	Beispielautomat . . . . .	33
3.5	Erweiterung des Analysevorgangs . . . . .	33
<b>4</b>	<b>Umsetzung der Erweiterung</b>	<b>37</b>
4.1	Technische Umsetzung . . . . .	37
4.1.1	Verhaltensmusterspezifikation . . . . .	37
4.1.2	Automatentransformation . . . . .	39
4.1.3	Analysevorgang . . . . .	41
4.2	Erweiterung der Benutzerschnittstelle . . . . .	47
4.2.1	Spezifikation von Mengenobjekten und Each-Fragmenten . . . . .	47

4.2.2	Spezifikation von Selbstaufrufen . . . . .	49
4.2.3	Spezifikation von Argumenten und Zuweisungen . . . . .	50
<b>5</b>	<b>Praktische Anwendung</b>	<b>51</b>
5.1	Bisherige Ergebnisse . . . . .	51
5.2	Praktische Anwendung des Observer-Entwurfsmusters . . . . .	51
5.2.1	Beispielprogramm . . . . .	52
5.2.2	Analyseergebnisse . . . . .	53
5.2.3	Erweitertes Beispielprogramm . . . . .	54
5.3	Praktische Anwendung des State-Entwurfsmusters . . . . .	55
5.3.1	Beispielprogramm . . . . .	55
5.3.2	Analyseergebnisse . . . . .	57
<b>6</b>	<b>Verwandte Arbeiten</b>	<b>59</b>
6.1	Verwendung von Mengenobjekten . . . . .	59
6.2	Kombinierte statische und dynamische Entwurfsmusteranalysen . . . . .	59
<b>7</b>	<b>Zusammenfassung und Ausblick</b>	<b>63</b>
7.1	Zusammenfassung . . . . .	63
7.2	Ausblick . . . . .	64
 <b>Anhang</b>		
<b>A</b>	<b>Anhang</b>	<b>67</b>
A.1	Verhaltensmuster . . . . .	67
A.1.1	Observer . . . . .	67
A.1.2	Chain Of Responsibility . . . . .	68
A.1.3	State . . . . .	69
A.2	Neue Datenformate . . . . .	70
A.2.1	Verhaltensmusterkatalog . . . . .	70
A.3	Modelle der Beispielprogramme . . . . .	72
A.3.1	Observer-Beispiel . . . . .	72
A.3.2	State-Beispiel . . . . .	72
<b>Literatur</b>		<b>73</b>

# 1 Einleitung

Software befindet sich meist lange Zeit im Einsatz und wird ständig weiterentwickelt. Auf Grund dessen muss sie während ihrer gesamten Lebensdauer gewartet werden. Aus diesem Grund verbringen Softwareentwickler einen großen Teil ihrer Zeit mit dem Warten von Softwaresystemen [Mey06]. Oft sind die Leute, die ein System warten, nicht dieselben, die es entworfen haben. Deswegen müssen sie viel Zeit aufwenden, um die Software zu verstehen. Das Verstehen des Quelltextes ist besonders bei großen Applikationen problematisch und vor allem zeitaufwendig. Der Prozess der Analyse eines Softwaresystems mit der Intention, die Komponenten des Systems sowie ihre Beziehungen zu identifizieren und auf einer abstrakteren Ebene zu dokumentieren, wird als *Reverse Engineering* bezeichnet [CJ90]. Untersuchungen haben ergeben, dass die Wartung eines Softwaresystems 75 Prozent seiner Lebensdauer ausmacht und dass das Verstehen der Software bis zu 80 Prozent der Zeit eines Reverse Engineers in Anspruch nimmt [WQY98].

Reverse Engineering wird durch die Identifikation von so genannten *Entwurfsmustern* (Engl.: *Design Patterns*) innerhalb eines Softwaresystems unterstützt. Entwurfsmuster sind in der Softwareentwicklung inzwischen weit verbreitet und werden insbesondere seit Mitte der 1990er Jahre verstärkt dokumentiert. Sie beschreiben bewährte Lösungen zu immer wieder auftauchenden Problemen in einem bestimmten Kontext [GHJV95]. Booch geht davon aus, dass in jedem Softwaresystem Entwurfsmusterimplementierungen zu finden sind [Boo08]. Vor allem große Softwaresysteme basieren auf Mustern, die aufgedeckt und analysiert werden müssen, um das System verstehen zu können, da derartige Systeme meist sehr komplex sind [KSR99].

Die Identifizierung von Entwurfsmusterimplementierungen hilft zwar beim Verständnis und der Dokumentation des Designs einer Software, jedoch ist die Suche nach ihnen insbesondere in einer großen Menge von Quelltext sehr aufwendig. Daher ist die Unterstützung des Menschen durch Werkzeuge, die über eine automatische Entwurfsmustererkennung verfügen, sehr hilfreich.

In den vergangenen Jahren wurden viele Werkzeuge entwickelt, die Entwurfsmusterimplementierungen in Softwaresystemen finden können (z.B. [KP96], [Ant98], [SG98], [KSR99], [BP00], [TCHS05], [SO06], [KGH06]). Entwurfsmuster beschreiben Struktur und Verhalten von Softwarekomponenten. Bisher beschränkte sich die automatische Entwurfsmustererkennung allerdings häufig auf eine statische Analyse des Quelltextes, um strukturelle Eigenschaften der Entwurfsmuster zu erkennen. Es gibt jedoch Muster, die strukturell ähnlich oder sogar identisch sind. Sie können hinsichtlich der Struktur nicht unterschieden werden. Deshalb reicht die statische Analyse nicht aus.

Entwurfsmuster werden nicht nur durch ihre Struktur, sondern auch durch ihr Laufzeitverhalten definiert. Verhalten kann teilweise auch statisch erkannt werden. Jedoch wird zur Berücksichtigung von Konzepten, wie dynamischer Methodenbindung und Polymorphismus, eine so genannte dynamische Analyse, also eine Analyse des Verhaltens zur Laufzeit, benötigt [Wen07]. Eine Kombination von statischer und dynamischer Analyse führt somit vermutlich zu besseren Ergebnissen bei der Entwurfsmustererkennung als die statische Analyse alleine.

Die Arbeitsgruppe Softwaretechnik der Universität Paderborn entwickelte mit ihrem Reverse-Engineering-Werkzeug RECLIPSE eine solche Kombination. RECLIPSE unterstützt die Erkennung von Entwurfsmustern, indem es dem Benutzer die Möglichkeit gibt, zuerst eine statische Analyse der Struktur und anschließend eine dynamische Analyse des Verhaltens durchzuführen.

Für die Strukturanalyse werden Modelle der Struktur der Entwurfsmuster spezifiziert, sogenannte *Strukturmuster*. Strukturmuster sind bei diesem Ansatz Graphgrammatikregeln. Durch die strukturelle Analyse eines Softwaresystems werden in diesem die Stellen gekennzeichnet, an denen Entwurfsmusterimplementierungen vermutet werden, die so genannten *Kandidaten*. Unter den Kandidaten können jedoch auch noch Konstrukte sein, die fälschlicherweise als Entwurfsmusterimplementierungen erkannt wurden.

In der verhaltensbasierten Analyse wird mit *Verhaltensmustern* gearbeitet. Jedem Strukturmuster kann ein entsprechendes Verhaltensmuster zugeordnet werden. In der Verhaltensanalyse wird die zu untersuchende Software ausgeführt, um das Laufzeitverhalten der Kandidaten mit den spezifizierten Verhaltensmustern vergleichen zu können.

Die Ergebnisse der dynamischen Verhaltensanalyse werden ausgewertet. Anhand der Bewertung kann der Benutzer entscheiden, ob es sich bei den durch die statische Analyse gefundenen Kandidaten wirklich um Entwurfsmusterimplementierungen handelt.

Die Umsetzung dieser Konzepte in RECLIPSE wird in [Wen07] beschrieben.

### 1.1 Problemstellung

Der in RECLIPSE verwendete Mechanismus zur verhaltensbasierten Entwurfsmustererkennung zeigte bei der praktischen Anwendung einige Schwächen ([Wen07], Abschnitt 6.3.3). Das Problem liegt bei den Verhaltensmustern. Diese werden durch Sequenzdiagramme nach der UML 2.0 mit einer eingeschränkten Syntax dargestellt. Diese Spezifikationsprache erlaubt jedoch keinen passenden Umgang mit Objektmengen.

Die Schwächen lassen sich gut anhand des Observer-Verhaltensmusters verdeutlichen. Der Zweck des Observer-Musters ist, dass verschiedene Beobachter-Objekte über Änderungen an einem beobachteten Objekt, genannt *Subjekt*, informiert werden. Zur Laufzeit können sich beliebig viele Observer-Objekte bei einem Subjekt registrieren. Mithilfe der verwendeten Sequenzdiagramme lassen sich jedoch nur

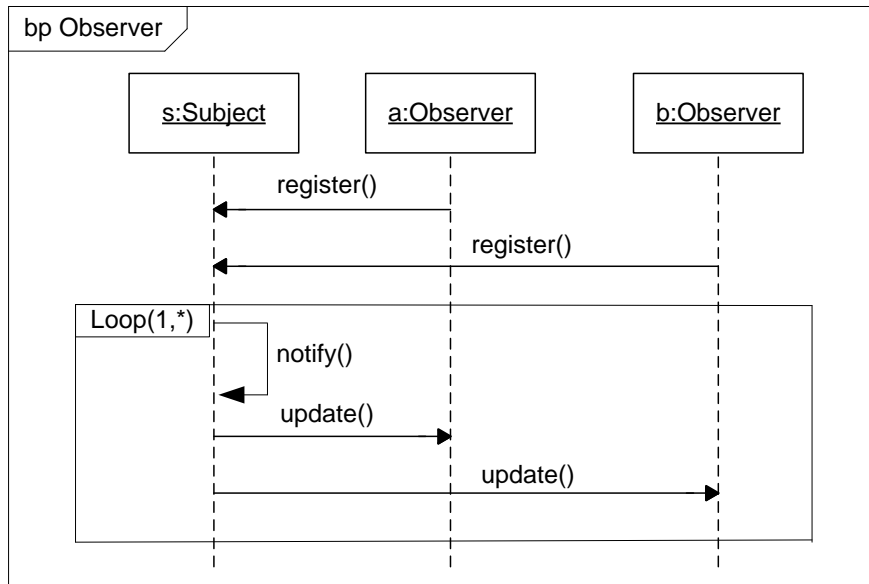


Abbildung 1.1: Observer-Verhaltensmuster

festen Anzahlen von Objekten modellieren. Somit können nur Observer-Muster-Instanzen erkannt werden, deren Anzahl von Observer-Objekten genau der spezifizierten Anzahl im Verhaltensmuster entspricht.

Mit dem Observer-Verhaltensmuster beispielsweise, wie es in Abbildung 1.1 dargestellt ist, würden nur Vorkommen des Observer-Musters mit genau zwei Observer-Objekten erkannt werden. Treten zur Laufzeit mehrere Observer auf, ändert sich die Abfolge der Methodenaufrufe, da die Methoden `register()` und `update()` entsprechend oft aufgerufen werden. Somit bleibt eine solche Entwurfsmusterimplementierung bei der dynamischen Analyse unerkannt.

Ein ähnliches Problem tritt beim State-Verhaltensmuster auf. Das State-Entwurfsmuster beschreibt einen Zustandswechsel mit dem Ziel, Anfragen an einen Kontext abhängig von dessen Zustand bearbeiten zu können. Mit der aktuellen Spezifikationssprache kann man nur eine feste Anzahl Zustände darstellen. Die Anzahl der verwendeten Zustände des State-Musters zur Laufzeit ist jedoch nicht festgelegt, es können beliebig viele Zustände und Zustandswechsel vorkommen.

## 1.2 Vorgehensweise und Lösungsansatz

Das oben beschriebene Problem lässt sich durch eine Erweiterung der Spezifikationssprache lösen. Lothar Wendehals schlug bereits einen Lösungsweg vor, in dem ein mengenwertiges Verhaltensmusterobjekt eingesetzt wird [Wen07]. Mithilfe des Mengenobjektes kann einem Objekt bei der Verhaltensanalyse eine beliebig große Anzahl Instanzen eines gemeinsamen Typs zugeordnet werden.

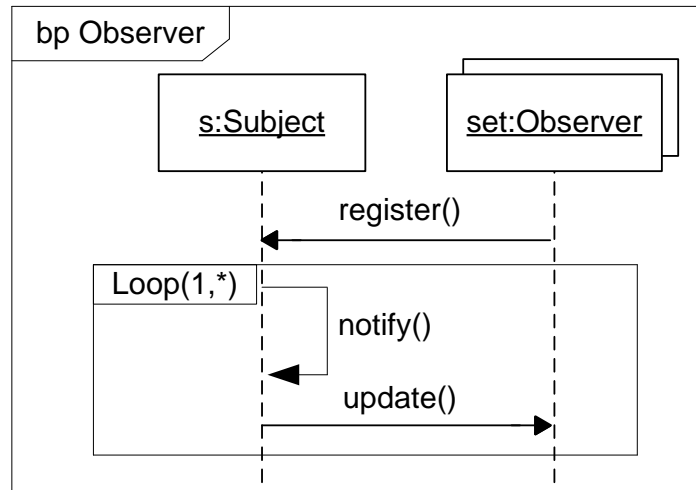


Abbildung 1.2: Observer-Verhaltensmuster mit erweiterter Syntax nach [Wen07]

Eine Möglichkeit für die Darstellung der erweiterten Syntax zeigt Abbildung 1.2. Dort ist ein Mengenobjekt mit dem Namen `set` des Typs `Observer` abgebildet. Das Mengenobjekt ist durch den doppelten Rahmen gekennzeichnet. Das Objekt `s` des Typs `Subject` hingegen ist ein einfaches Objekt.

Für die Lösung des Problems mit dem State-Muster schlägt Wendehals eine Erweiterung der Spezifikationsprache um Argumente und Zuweisungen vor. Mithilfe einer Zuweisung kann die Identität eines Verhaltensmusterobjektes geändert werden. Das daraus resultierende State-Verhaltensmuster wird in Abbildung 1.3 dargestellt. Dort sieht man unten am Ende der Lifeline des `abstractState`-Objekts eine Zuweisung mit der Beschriftung `a:=s`. Das bedeutet, dass dem Objekt des Typs `abstractState` (vorher `a`), nun eine andere Instanz (`s`) zugewiesen wird. Diese wurde vorher in der Methode `setState(s)` übergeben und gebunden.

Auf Grund der Erweiterung der Verhaltensmuster muss auch das Verfahren der Verhaltensanalyse angepasst werden. Dazu gehört eine Weiterentwicklung der zur Analyse verwendeten Automaten.

### 1.3 Struktur der Arbeit

Die vorliegende Arbeit wird im zweiten Kapitel mit einigen Grundlagen, die zum weiteren Verständnis notwendig sind, fortgeführt. Dabei wird zuerst die struktur- und verhaltensbasierte Entwurfsmustererkennung in RECLIPSE, auf denen diese Arbeit aufbaut, genauer erläutert. Dazu zählt auch eine Darstellung der Verhaltensanalyse.

Das Thema des dritten Kapitels ist das Konzept der erarbeiteten Erweiterung der Verhaltenserkennung unter Berücksichtigung von Objektmengen.

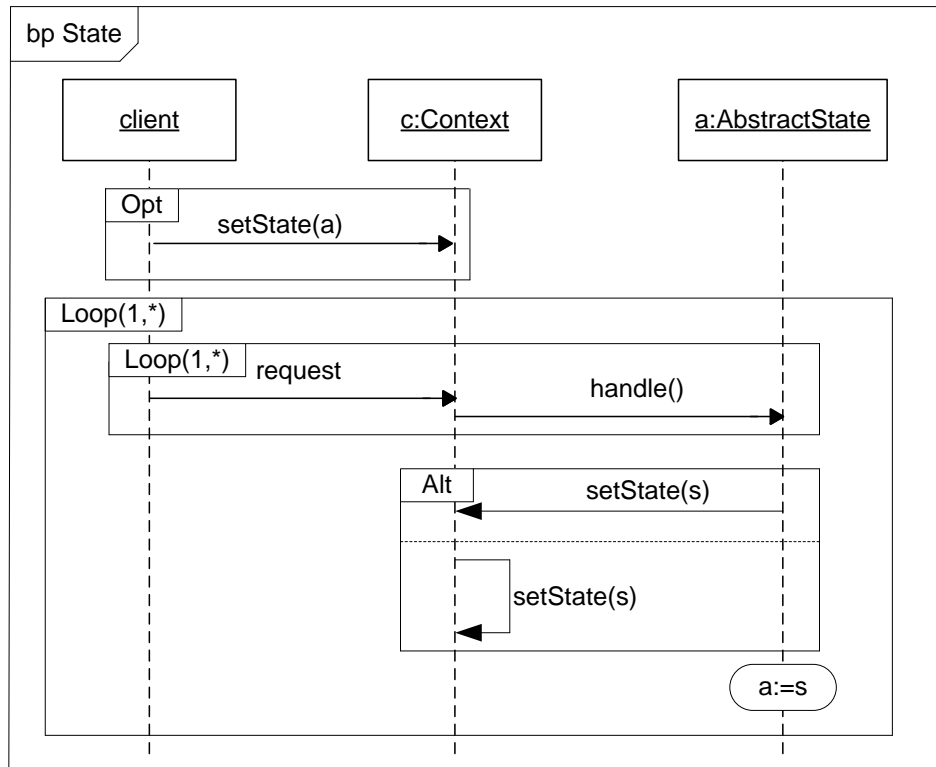


Abbildung 1.3: State-Verhaltensmuster mit erweiterter Syntax [Wen07]

Das vierte Kapitel beschäftigt sich mit der Umsetzung des Konzepts in RECLIPSE.

Im Anschluss wird der Ansatz inklusive der Erweiterung in einer praktischen Anwendung untersucht. Dazu werden einige Entwurfsmuster spezifiziert und an einem Beispielprogramm getestet. Insbesondere werden die Erkennung des Observer- und des State-Musters mithilfe der erweiterten Verhaltensmuster überprüft. Kapitel fünf beschreibt die praktische Anwendung und stellt die Ergebnisse der Untersuchung vor.

Im sechsten Kapitel werden verwandte Arbeiten thematisiert. Dabei werden Arbeiten anderer Wissenschaftler diskutiert, die einen starken Bezug zu der vorliegenden Arbeit haben und dieselbe Thematik behandeln.

Die Arbeit schließt im siebten Kapitel mit einer Zusammenfassung. Dabei wird ein Ausblick gegeben, in dem auf festgestellte Schwächen des entwickelten Verfahrens und weitere mögliche Erweiterungen eingegangen wird.



## 2 Grundlagen

Das folgende Kapitel erläutert zuerst die Probleme, die allgemein bei der Entwurfsmustererkennung auftauchen. Daraufhin wird der Ablauf der struktur- und verhaltensbasierten Entwurfsmustererkennung in RECLIPSE beschrieben. Dabei wird erst auf die statische Strukturanalyse, danach auf die dynamische Verhaltensanalyse genauer eingegangen.

### 2.1 Probleme der Entwurfsmustererkennung

Die Identifikation von Entwurfsmusterimplementierungen in bestehender Software ist zwar hilfreich für den Verstehensprozess beim Reverse-Engineering, jedoch ist die manuelle Identifikation sehr aufwendig. Deswegen ist es notwendig, den Menschen durch geeignete Werkzeuge zu unterstützen.

Entwurfsmuster werden häufig zumindest teilweise informell beschrieben, so auch in [GHJV95], das die in dieser Arbeit verwendeten Muster enthält. Diese Art der Beschreibung erlaubt dem Entwickler, Entwurfsmuster flexibel einzusetzen. Dadurch entstehen jedoch viele verschiedene Implementierungsvarianten im Quelltext [Nie04]. Da es prinzipiell unendlich viele Implementierungsvarianten eines Entwurfsmusters geben kann, kann eine automatische Entwurfsmustererkennung keine absolut präzisen Ergebnisse erzielen. Konstrukte, die von der automatischen Mustererkennung fälschlicherweise als Entwurfsmusterimplementierungen identifiziert wurden, aber in Wirklichkeit keine sind, nennt man *False Positives*. Bestehende Entwurfsmusterimplementierungen, die nicht gefunden wurden, nennt man im Gegensatz dazu *False Negatives*.

### 2.2 Struktur- und verhaltensbasierte Entwurfsmustererkennung in Reclipse

RECLIPSE kombiniert die statische Analyse einer Software mit einer dynamischen Analyse. Bei der statischen Analyse handelt es sich um eine Untersuchung des Quelltextes, ohne die Software auszuführen. Da die Struktur eines Softwaresystems untersucht wird, spricht man hier auch von einer *Strukturanalyse*. Die dynamische Analyse hingegen untersucht die Software zur Laufzeit, indem sie ihr Verhalten beobachtet. Man kann in diesem Fall also auch von einer *Verhaltensanalyse* sprechen.

Es gibt Entwurfsmuster, die strukturell ähnlich oder identisch sind, wie beispielsweise das State- und das Strategy-Muster. Eine Implementierung eines dieser Muster wird von der statischen Analyse doppelt erkannt: einmal als State- und einmal als Strategy-Entwurfsmusterimplementierung. Einer der beiden Kandidaten muss demnach ein False-Positive sein. Die Muster unterscheiden sich jedoch in ihrem Verhalten. Daher ist eine Kombination von struktur- und verhaltensbasierter Entwurfsmustererkennung geeignet, um die Menge der False-Positives so stark wie möglich einzuschränken.

Eine detaillierte Beschreibung der Strukturanalyse findet man in [Nie04]. Die Erweiterung der Strukturanalyse sowie der genaue Ablauf der Verhaltensanalyse werden in [Wen07] spezifiziert. In dieser Arbeit wird lediglich ein kurzer Überblick gegeben.

### 2.2.1 Strukturanalyse

Die Strukturanalyse von RECLIPSE arbeitet nicht direkt auf dem Quelltext des zu untersuchenden Softwaresystems, sondern auf einem *Strukturmodell*, das sich gut zur algorithmischen Analyse eignet. Das Strukturmodell wird bei diesem Ansatz durch einen abstrakten Syntaxgraphen dargestellt. Im Wesentlichen stellt das Modell Klassen, Attribute, Methoden und Vererbungen zwischen Klassen dar. Das Strukturmodell ist leicht austauschbar, sodass die Entwurfsmustererkennung auch auf andere Modelle anwendbar ist [GMW06].

Für die strukturbasierte Entwurfsmustererkennung ist es notwendig, zuerst die Struktur der Entwurfsmuster zu spezifizieren, die man suchen will. Dies geschieht durch *Strukturmuster*. Durch die strukturelle Analyse wird das Strukturmodell durch sogenannte *Annotationen* angereichert. Eine Annotation wird durch einen Knoten dargestellt und durch zusätzliche Kanten mit Knoten des Strukturmodells verbunden. Die Annotationen markieren die Fundstellen von potentiellen Entwurfsmusterimplementierungen. Die annotierten Fundstellen werden als *Kandidaten* bezeichnet und in einem annotierten Klassendiagramm dargestellt.

Die Abbildung 2.1 zeigt das Strukturmuster des Observer-Entwurfsmusters. Die Intention des Observer-Musters ist es, eine Abhängigkeit zwischen mehreren Beobachtern und einem Subjekt zu definieren, sodass sobald das Subjekt seinen Status ändert, all seine Beobachter informiert werden und sich automatisch aktualisieren [GHJV95]. Das Observer-Muster setzt sich unter anderem aus einer Observer-Klasse (`observer:Class`) mit einer Methode `update` und einer Subjekt-Klasse (`subject:Class`) mit den Methoden `register` und `notify` zusammen, jeweils repräsentiert durch die entsprechenden Objekte vom Typ `Method`. Manche Methoden haben Parameter, die durch ein Objekt vom Typ `Parameter` angegeben werden, wie hier `p:Parameter`. Der Typ des Parameters wird durch die Verbindung `paramType` definiert. Der grün markierte Knoten `:Observer`, der zusätzlich mit `<<create>>` beschriftet ist, ist die Annotation für das Entwurfsmustervorkommen.

Muster lassen sich inkrementell spezifizieren. Das heißt, man kann Entwurfsmu-

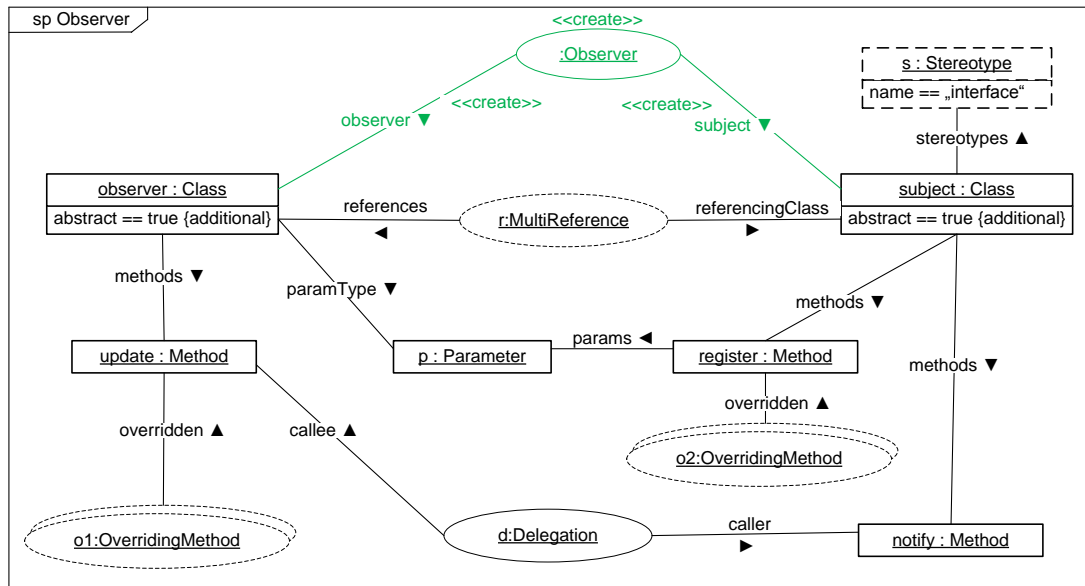


Abbildung 2.1: Strukturmuster des Observer-Entwurfsmusters

stervorkommen anhand von Annotationen auch in anderen Strukturmustern einsetzen. Häufig vorkommende Strukturen, wie z.B. Delegationen können somit wiederverwendet werden. Dies ist auch im Observer-Strukturmuster der Fall. Wie man in Abbildung 2.1 sieht, enthält es die *Hilfsmuster* `Delegation`, `MultiReference` und `OverridingMethod`. Die in der Abbildung gestrichelt dargestellten Knoten sind optional. Optionale Anteile eines Musters werden im weiteren Verlauf dieses Abschnitts näher erläutert.

Lothar Wendehals erweiterte 2007 die Syntax der Strukturmuster, um sogenannte *unscharfe Regeln* miteinbeziehen zu können [Wen07]. Das Ziel war eine bessere Handhabung der Differenzen zwischen verschiedenen Implementierungsvarianten eines Entwurfsmusters.

Unschärfe Regeln bestehen zum Teil aus notwendigen Bedingungen, die von allen zu erkennenden Varianten erfüllt werden müssen. Des Weiteren enthalten sie zusätzlich weitere, nicht notwendige Bedingungen, die nicht von jeder Entwurfsmusterimplementierung erfüllt sein müssen. Diese optionalen Elemente liefern jedoch gute Hinweise auf eine tatsächliche Implementierung. Optionale Knoten werden im Strukturmuster durch gestrichelte Umrandungen visualisiert, optionale Attribute durch den Zusatz `additional`. Das Observer-Strukturmuster auf Abbildung 2.1 enthält beispielsweise das optionale Element `s:Stereotyp` mit dem Wert `name == „interface“`, welches besagt, dass die Klasse mit einem Stereotyp gekennzeichnet sein kann. In diesem Fall kann die Klasse ein Interface sein. Außerdem ist der Ausdruck `abstract == true` in `observer` und `subject` als optional gekennzeichnet worden.

In Strukturmustern können auch Objektmengen spezifiziert werden. Mengen-

knoten werden mithilfe eines doppelten Rahmens dargestellt. In der Abbildung 2.1 findet man beispielsweise zwei mal die Annotations-Mengenknoten `OverridingMethod`. Die Annotationsknoten `OverridingMethod` bedeuten, dass die mit ihnen verbundenen Methoden überschrieben werden müssen, um zu dem Strukturmuster konform zu sein. Während der Erkennung des Strukturmusters können beliebig viele Objekte aus der Struktur des zu untersuchenden Softwaresystems an einen Mengenknoten gebunden werden. Mengenknoten werden immer als optional gekennzeichnet, da Mengen grundsätzlich auch leer sein können.

Auch der folgende Bewertungsvorgang der Kandidaten wurde von Wendehals überarbeitet [Wen07]. Um einen Kandidaten zu bewerten, wird der Grad seiner Übereinstimmung zum Strukturmuster berechnet. Werden optionale Musterteile erfüllt, wird die Bewertung positiv beeinflusst, ansonsten negativ.

Je höher die Bewertung eines Kandidaten, umso wahrscheinlicher handelt es sich bei diesem Kandidaten um eine wirkliche Entwurfsmusterimplementierung. Niedrige Bewertungen hingegen lassen auf False-Positives schließen.

Die Abbildung 2.2 stellt das Metamodell der Strukturmuster dar. Strukturmuster (`StructuralPattern`) bestehen aus Objekten (`SPObject`) und Verbindungen (`Connection`) zwischen Objekten. Im Observer-Strukturmuster aus Abbildung 2.1 gehören die Knoten vom Typ `Class`, wie z.B. `observer` und ihre Methoden, wie z.B. `update:Method`, zu den `SPObjects`. Auch andere im Strukturmuster rechteckig dargestellte Knoten, wie z.B. `p:Parameter` sind `SPObjects`. Des Weiteren gibt es Annotationen (`SPAnnotationObject`). Dazu zählen die runden Knoten im Strukturmuster, wie z.B. `:Observer` oder `r:MultiReference`. In der späteren Verhaltensanalyse werden Referenzen auf die Objekte der Strukturanalyse benötigt (`BehavioralPatterns::Message`, `BehavioralPatterns::BPObject`).

Ein Beispiel für einen Observer-Kandidaten wird in Abbildung 2.3 gezeigt. Die Klasse `WeatherSensor` entspricht einer Subjekt-Klasse, `WeatherStation` ist der Observer. Die Stationen können sich über die Methode `attach` beim Sensor registrieren, welcher alle bei sich registrierten Stationen über seine Aktualisierungen benachrichtigt.

### 2.2.2 Verhaltensanalyse

Nicht nur aufgrund der Tatsache, dass Entwurfsmuster strukturell gleich sein können, können False-Positives bei der Strukturanalyse nicht ausgeschlossen werden. Auch Polymorphismus und dynamische Methodenbindung zur Laufzeit erfordern zusätzlich zur statischen Analyse eine dynamische Untersuchung. Deswegen ist eine auf die strukturelle Analyse folgende dynamische Verhaltensanalyse notwendig. Anhand dieser Laufzeitanalyse des Verhaltens versucht man die erkannten Kandidaten auszuschließen oder zu bestätigen.

Der gesamte Erkennungsprozess der struktur- und verhaltensbasierten Entwurfsmustererkennung in RECLIPSE wird in Abbildung 2.4 dargestellt. Der linke Bereich der Abbildung, der zu der statischen Analyse gehört, wurde in Abschnitt 2.2.1 beschrieben. Dabei bekommt die Analyse als Eingabe den Quelltext des

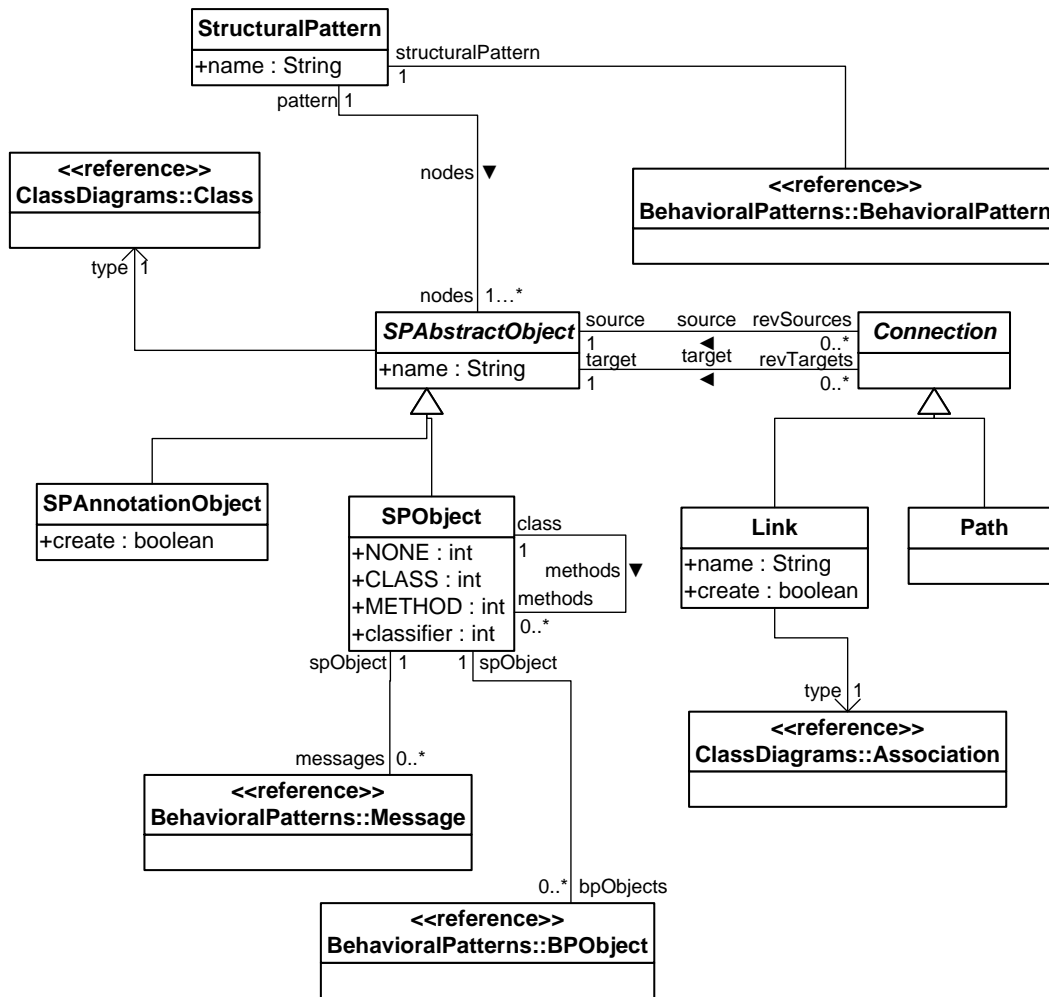


Abbildung 2.2: Metamodell der Strukturmuster nach [Wen07]

zu untersuchenden Programms und Strukturmuster. Das Ergebnis der statischen Analyse sind Kandidaten für Entwurfsmusterimplementierungen. Für die dynamische Analyse werden Laufzeitinformationen gebraucht. Dazu wird das zu untersuchende Programm ausgeführt. Während der Programmausführung werden die Kandidaten der statischen Analyse beobachtet. Die Programmausführung wird in Form von Listen von Methodenaufrufen aufgezeichnet, die so genannten Traces. Nun folgt der Teil der dynamischen Analyse, der auf der rechten Seite gezeigt wird. Dieser bekommt als Eingabe die aus dem Ergebnis der statischen Analyse erzeugten Traces. Die dynamische Analyse arbeitet außerdem mit einem Katalog von Modellen, die das Verhalten von Entwurfsmustern spezifizieren. Diese Modelle werden *Verhaltensmuster* genannt und im Folgenden beschrieben.

Hier reicht es für die Verhaltensanalyse aus, Art und Reihenfolge von Nachrichten zu betrachten. Diese treten in Form von Methodenaufrufen, die zwischen

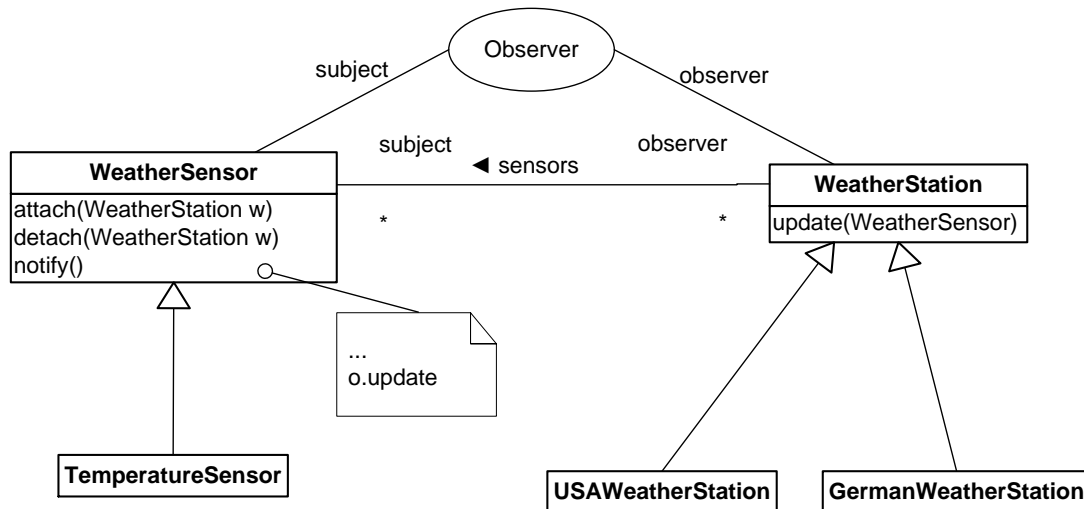


Abbildung 2.3: Observer-Entwurfsmuster-Kandidat

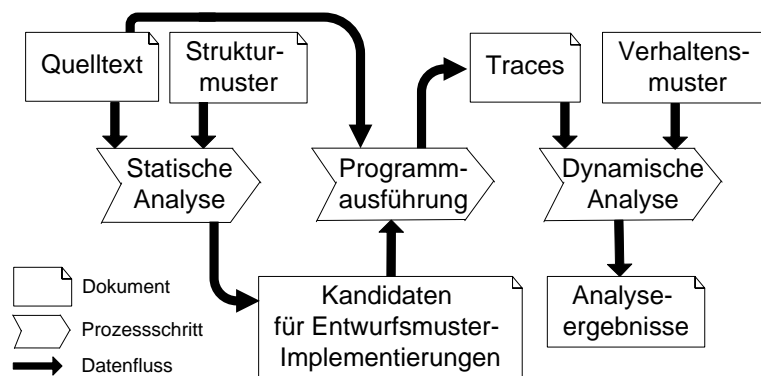


Abbildung 2.4: Der kombinierte Erkennungsprozess

Objekten ausgetauscht werden, auf.

## Verhaltensmuster

Verhaltensmuster sind das Analogon zu Strukturmustern. Während Strukturmuster die Struktur eines Entwurfsmusters spezifizieren, beschreiben Verhaltensmuster sein Verhalten zur Laufzeit. Zu jedem Verhaltensmuster muss es ein zugehöriges Strukturmuster geben<sup>1</sup>.

Verhaltensmuster werden auf Basis von Sequenzdiagrammen nach der UML 2.0 definiert. Sequenzdiagramme haben den Vorteil, dass sie unter Softwareentwicklern bekannt und allgemein leicht zugänglich und schnell lesbar sind [Wen07]. Verhaltensmuster sind im Vergleich zu Sequenzdiagrammen syntaktisch eingeschränkt.

<sup>1</sup>Es muss jedoch nicht zu jedem Strukturmuster ein Verhaltensmuster spezifiziert werden, die Gegenrichtung gilt also nicht.

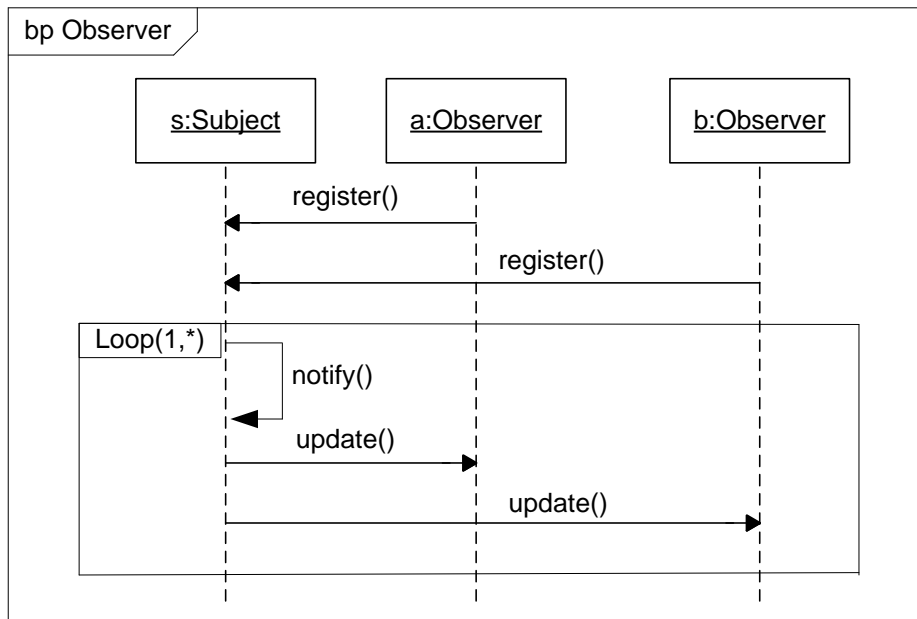


Abbildung 2.5: Observer-Verhaltensmuster

Die Einschränkungen bestehen unter anderem darin, dass nur synchrone Nachrichten zwischen Objekten zur Verfügung stehen, denn die Nachrichten modellieren Methodenaufrufe zwischen zwei Objekten.

In der Literatur gibt es üblicherweise keine formale Spezifikation des Verhaltens von Entwurfsmustern in Musterbeschreibungen. Der Reverse-Engineer hat also die Aufgabe, aus der informellen Beschreibung<sup>2</sup> ein Verhaltensmuster abzuleiten. Laut Wendehals muss ein Verhaltensmuster nicht das gesamte mögliche Verhalten einer Entwurfsmusterimplementierung zur Laufzeit beschreiben. Es sollte sich stattdessen auf die für den Reverse-Engineer wesentlichen Eigenschaften des Entwurfsmusters beschränken [Wen07].

Ein Beispiel ist das Observer-Verhaltensmuster, das in Abbildung 2.5 dargestellt wird. Darin gibt es ein Subjekt (`s:Subject`), bei dem sich zwei Observer registrieren (`a:Observer`, `b:Observer`). Dies geschieht, indem sie jeweils die Methode `register()` des Subjekts aufrufen. Wenn nun das Subjekt seine Methode `notify()` aufruft, weil sich Daten geändert haben, ruft es auf den bei ihm registrierten Observern jeweils `update()` auf, um sie zu aktualisieren. Das Schleifenelement, ein so genanntes *Loop-Fragment* (beschriftet mit `loop(1,*)`), kennzeichnet, dass dies beliebig oft geschehen kann, aber mindestens einmal geschehen muss.

Die Abbildung 2.6 zeigt das Metamodell der Verhaltensmuster. Es können nur Objekte (`BPObject`) von Typen angelegt werden, die im Strukturmuster als Klasse gekennzeichnet wurden. Die Objekte `Subject`, `a:Observer` und `b:Observer`

<sup>2</sup>Solche Beschreibungen findet man zum Beispiel im Standardwerk von Gamma et al. [GHJV95].

aus dem Observer-Verhaltensmuster aus Abbildung 2.5 sind BPObjets. Ihre Typen verweisen auf die gleichnamigen Objekte aus dem Observer-Strukturmuster in Abbildung 2.1. Analog dazu müssen sich alle Methoden, die im Verhaltensmuster verwendet werden, auf Method-Objekte aus dem Strukturmuster beziehen. Ein Objekt stellt nur die Methoden zur Verfügung, mit denen es über einen methods-Link im Strukturmuster verbunden wurde. Beispielsweise kann hier nur auf Objekten vom Typ `Subject` die Methode `register` aufgerufen werden (vgl. Abbildung 2.1). Es gibt außerdem die Möglichkeit, untypisierte Objekte im Ver-

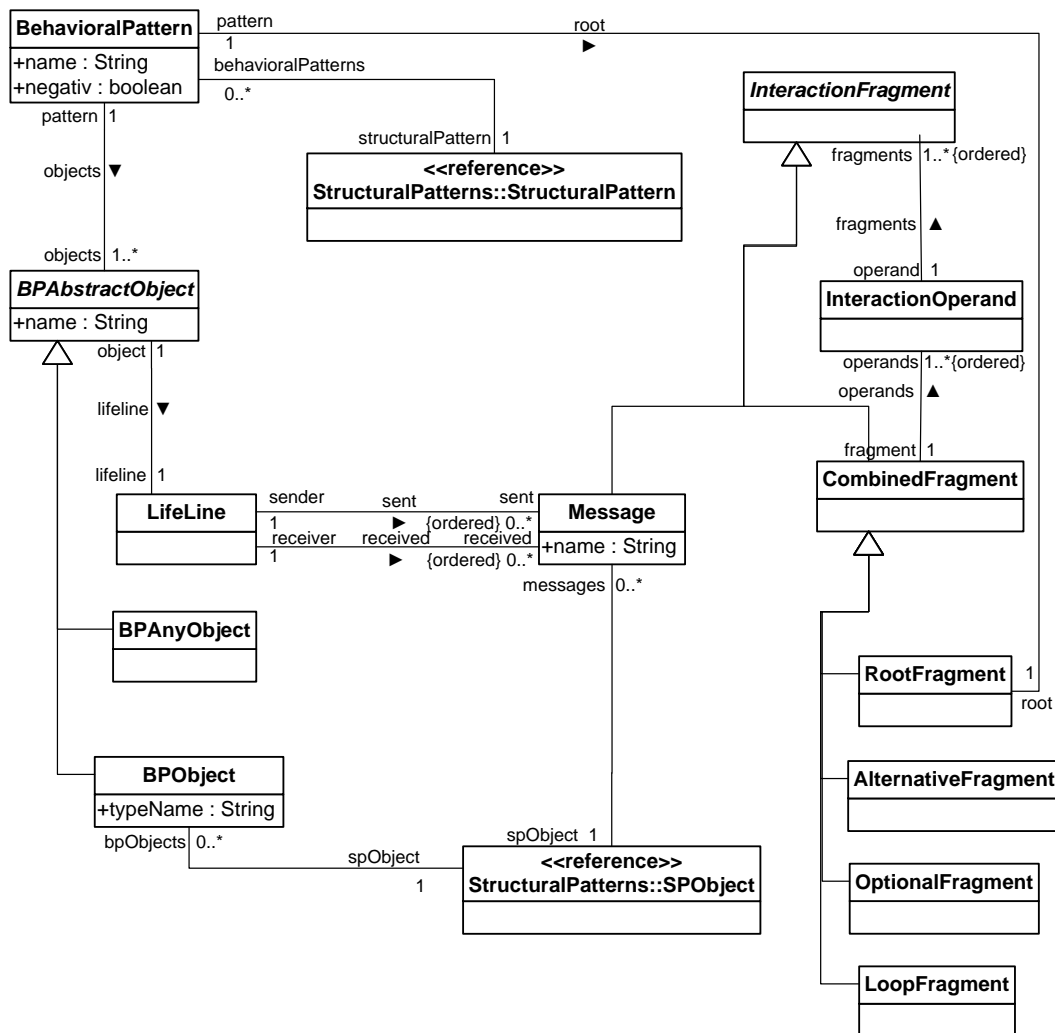


Abbildung 2.6: Metamodell der Verhaltensmuster nach [Wen07]

haltensmuster (`BPAnyObject`) zu benutzen. Diese werden verwendet, wenn man den Typ eines Objektes nicht kennt oder dieser an der Stelle gleichgültig ist. Dargestellt wird ein solches Objekt, indem man es mit einem Namen ohne Typ beschriftet. Beispielsweise kann man damit modellieren, dass irgendein Client eine

bestimmte Methode auf einem bestimmten anderen, typisierten Objekt aufruft. Dies ist zum Beispiel beim State-Muster der Fall, welches in einem späteren Kapitel vorgestellt wird. Auf den nicht-typisierten Verhaltensmusterobjekten dürfen keine Methodenaufrufe erfolgen. Außer den Nachrichten (`Message`) gibt es noch weitere Interaktionsfragmente. Mithilfe dieser besteht die Möglichkeit, Schleifen (`LoopFragment`, siehe Abbildung 2.5) und Alternativen (`AlternativeFragment`) zu definieren, sowie Teile der Interaktion als optional zu kennzeichnen (`OptionalFragment`).

## Traces

Das Verhalten des zu untersuchenden Softwaresystems wird durch ein Verhaltensmodell beschrieben. Es besteht aus einer Sequenz von Methodenaufrufen, die zur Laufzeit beobachtet und aufgezeichnet wurden. Eine solche Sequenz von Methodenaufrufen wird *Trace* genannt. Traces müssen nicht alle zur Laufzeit ausgeführten Methodenaufrufe umfassen, sondern können aus einer beliebigen Teilmenge ausgeführter Methodenaufrufe zwischen zwei Zeitpunkten bestehen, solange die Reihenfolge erhalten bleibt.

Zur Gewinnung von Traces gibt es zwei Möglichkeiten. Die erste ist das Beobachten des unveränderten Programms von außen, das sogenannte *Debugging*. Die zweite Möglichkeit ist die *Instrumentierung*. Dabei wird der Programmcode so verändert, dass die Methodenaufrufe aus dem Code heraus überwacht werden.

Bei der Verhaltensanalyse werden Traces auf ihre Konformität zu den spezifizierten Verhaltensmustern überprüft.

Ein Methodenaufruf des Traces ist konform zu einer Nachricht im Verhaltensmuster, wenn sowohl das aufrufende Verhaltensmusterobjekt und sein Typ, sowie das aufgerufene Verhaltensmusterobjekt und sein Typ, als auch die aufgerufene Methode übereinstimmen. Ein Trace ist konform zu einem Verhaltensmuster, wenn alle seine Methodenaufrufe konform zu den Nachrichten im Verhaltensmuster sind. Methodenaufrufe, deren Methoden nicht im Verhaltensmuster verwendet werden, werden ignoriert und nicht mit in den Trace aufgezeichnet.

Ist ein Trace zu einem Verhaltensmuster konform, so stellt dies einen Hinweis für die Vermutung, der Kandidat sei eine tatsächliche Entwurfsmusterimplementierung, dar. Ein nicht-konformer Trace hingegen ist ein Hinweis auf ein False-Positive. Eine wirkliche Aussage wird jedoch erst durch die Analyse aller Traces zu einem Kandidaten möglich.

## Variablenbindungen

Ein Strukturmuster beschreibt die Struktur eines Entwurfsmusters allgemein. Kandidaten sind Ausschnitte aus dem abstrakten Syntaxgraphen eines konkreten Softwaresystems, die dem Muster strukturell entsprechen und deren Elemente auf Teile des Strukturmusters abgebildet werden.

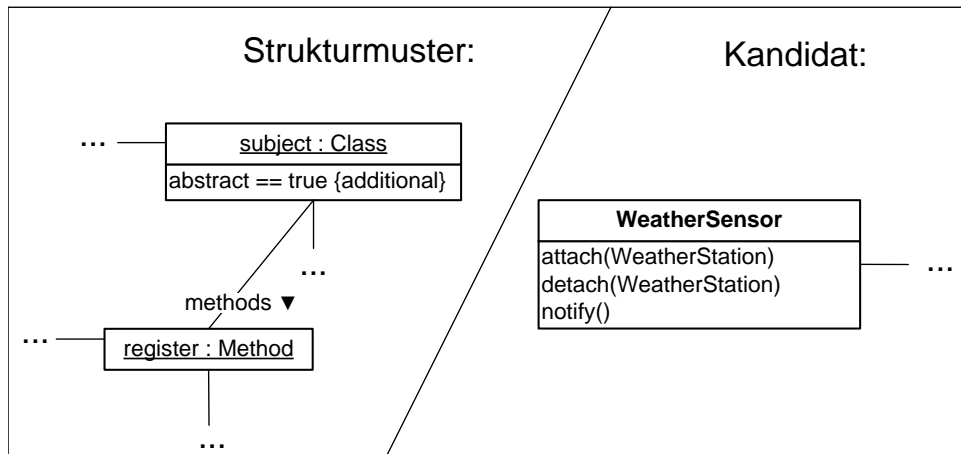


Abbildung 2.7: Ausschnitt aus Observer-Kandidat und -Strukturmuster

Abbildung 2.7 zeigt einen Ausschnitt aus dem Observer-Strukturmuster aus Abbildung 2.1 und dem Observer-Beispielkandidaten aus Kapitel 2.2.1, Abbildung 2.3. In diesem Beispiel werden das Objekt `WeatherSensor` aus dem Kandidaten und das Element `subject:Class` aus dem Strukturmuster aneinander gebunden. Ebenso wird die Methode `attach` aus dem Kandidaten an das Methodenobjekt `register` aus dem Strukturmuster gebunden<sup>3</sup>. Tabelle 2.2.2 zeigt alle Bindungen

Strukturmuster	Kandidat
Observer	WeatherStation
Subject	WeatherSensor
register	attach
notify	notify
update	update

Tabelle 2.1: Variablenbindungen für den Beispiel-Kandidaten

für den Beispielkandidaten.

Verhaltensmuster beschreiben das Verhalten eines Entwurfsmusters allgemein, enthalten jedoch durch die Kandidaten die gleiche Bindung, wie das Strukturmuster. Abbildung 2.8 zeigt denselben Ausschnitt aus dem Observer-Beispielkandidaten, wie in Abbildung 2.7 und einen Ausschnitt aus dem zugehörigen Observer-Verhaltensmuster. Hier gelten dieselben Bindungen wie oben: `WeatherSensor` wird an `s:subject` und `attach` wird an `register` gebunden.

<sup>3</sup>An dieser Stelle ist zu beachten, dass die Strukturanalyse mehrere mögliche Kandidaten vorschlägt: bei dem einen wird die Methode `attach` an die Methode `register` gebunden, während bei dem andern zum Beispiel die Methode `detach` an `register` gebunden wird. In folgenden wird nur der Kandidat betrachtet, der die oben angegebenen Variablenbindungen enthält.

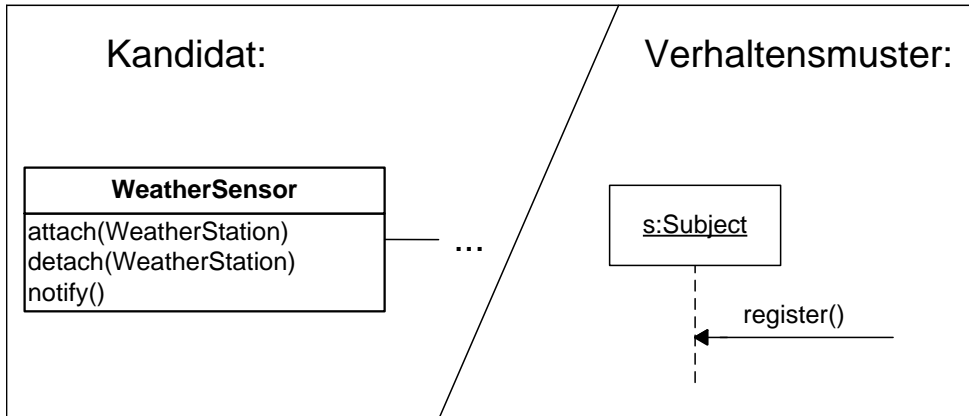


Abbildung 2.8: Ausschnitt aus Observer-Kandidat und -Verhaltensmuster

Die Bindung der Typnamen der Verhaltensmusterobjekte und der Methodennamen der Nachrichten steht also bereits zu Beginn der Verhaltensanalyse fest. Die Bindung der Verhaltensmusterobjekte selbst an konkrete Instanzen in der laufenden Software hingegen geschieht erst während der Verhaltensanalyse bei der Überprüfung der Konformität der Methodenaufrufe. Auch polymorphe Typbindungen sind erlaubt.

### Transformation in Automaten

Um die Erkennung der Verhaltensmuster algorithmisch durchführen zu können, werden sie in *endliche Automaten* transformiert. Dazu wird ein Verhaltensmuster zuerst in einen nichtdeterministischen endlichen Automaten umgewandelt. Jedes Interaktionsfragment des Verhaltensmusters wird in ein Teilkonstrukt des Automaten übersetzt. Abbildung 2.9 zeigt die Transformation einer Beispielnachricht nach diesem Schema. Die resultierenden Teilkonstrukte werden über  $\varepsilon$ -Transitionen

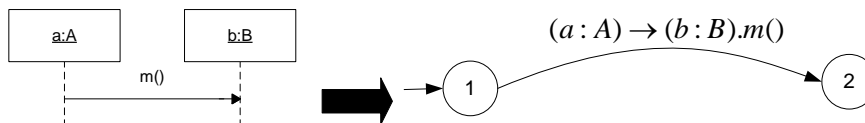


Abbildung 2.9: Transformation einer Nachricht des Verhaltensmusters in ein Automatenfragment

verbunden. Die Reihenfolge der Interaktionsfragmente im Verhaltensmuster bleibt dabei erhalten. Details zur Transformation werden in [Wen07] beschrieben.

Der nichtdeterministische Automat zu dem in 2.5 beschriebenen Observer-Verhaltensmuster wird in Abbildung 2.10 gezeigt. Wie beschrieben entspricht je-

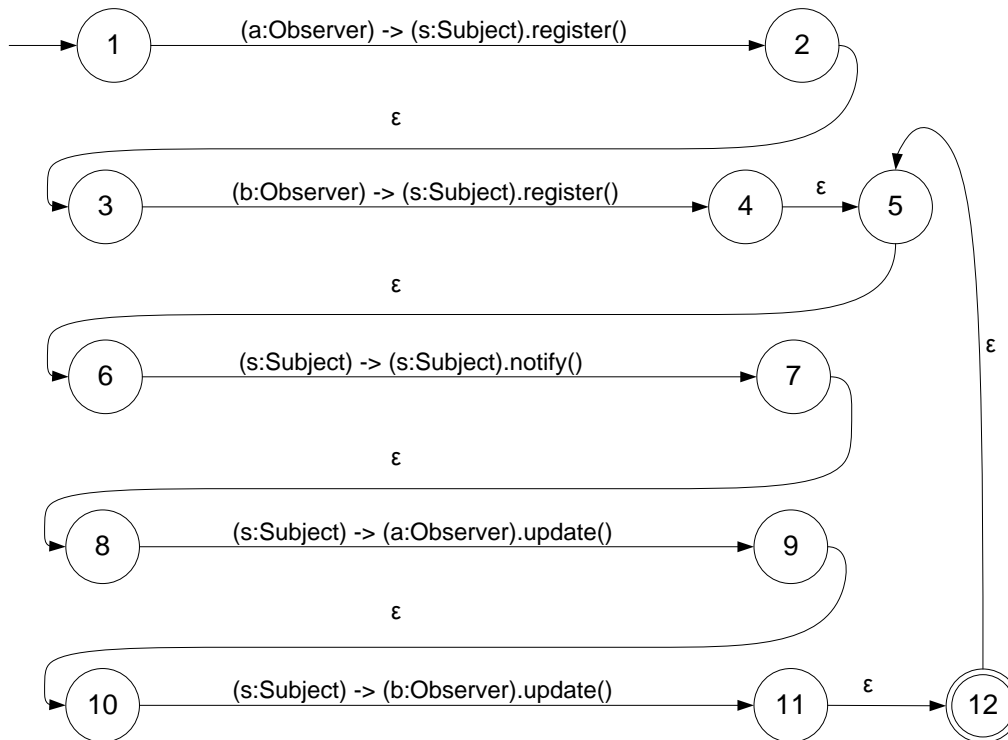


Abbildung 2.10: Automat aus dem Observer-Verhaltensmuster

de Nachricht im Verhaltensmuster einer Transition im Automaten. Das Loop-Fragment wird so übersetzt, dass es eine  $\varepsilon$ -Transition vom Zustand hinter dem zu wiederholenden Abschnitt, bis zum Zustand vor dem zu wiederholenden Abschnitt gibt. In der Abbildung 2.10 ist dies die Transition von Zustand 12 zum Zustand 5. Der Abschnitt muss also mindestens einmal ausgeführt werden, kann aber beliebig oft wiederholt werden, was genau der angegebenen Kardinalität im Verhaltensmuster entspricht. Der Zustand 12 ist akzeptierend. Das bedeutet, dass der Trace akzeptiert wird, wenn der Automat in diesem Zustand endet.

Der fertige nichtdeterministische Automat wird danach in einen deterministischen Automaten umgewandelt<sup>4</sup>. Damit der Automat keine nicht-konformen Traces akzeptiert, wird ein verwerfender Zustand hinzugefügt und mit zusätzlichen Transitionen verbunden, die unerlaubte Methodenaufrufe repräsentieren. Dieser ist aufgrund der Übersichtlichkeit in Abbildung 2.10 nicht dargestellt.

### Analysevorgang

Der endgültige Automat bekommt beim Erkennungsvorgang als Eingabe die Methodenaufrufe des Traces und kann nun entscheiden, ob ein Trace konform zu dem entsprechenden Verhaltensmuster ist, oder nicht.

<sup>4</sup>Dies ist durch Potenzmengenkonstruktion in polynomieller Zeit möglich [HU93].

Dazu werden zuerst die Variablen für die Klassen und Methoden im Automaten gegen die konkreten Klassen und Methoden des Kandidaten ausgetauscht.

Das verwendete Automatenmodell wurde um so genannte *Tokens*, ein bekanntes Konzept der Petri-Netze, erweitert. Ein Token repräsentiert den aktuellen Zustand eines Automaten. Jeder Zustand eines Automaten kann beliebig viele Tokens referenzieren, sodass mit einem Automaten beliebig viele Traces gleichzeitig auf Konformität zum Verhaltensmuster untersucht werden können. Ein Token enthält Referenzen auf die Variablenbindungen des Automaten. Es existiert ein Automat pro Verhaltensmuster und ein Token pro Trace.

Ein Methodenaufwurf im Trace, der im Verhaltensmuster einem der möglichen ersten Methodenaufwufe entspricht, löst die Erkennung durch den Automaten aus (so genannter *Trigger*). Der Trigger fügt den Automaten den aktiven Automaten der Verhaltensanalyse hinzu und erzeugt ein Token.

Zu Beginn der Analyse befindet sich der Automat im Anfangszustand. Der Automat konsumiert den ersten Methodenaufwurf und bindet dabei die Objektvariablen des Kandidaten, die bei diesem Aufruf verwendet werden, an die entsprechenden Variablen im Automaten. Daraufhin wechselt der Automat in den nächsten Zustand. Die folgenden Methodenaufwufe des Traces werden analog konsumiert, dabei werden die restlichen Variablen gebunden. Endet der Automat in einem akzeptierenden Zustand, wird der beobachtete Trace als konform zum Verhaltensmuster erkannt. Ist der Endzustand ein verwerfender Zustand, wird der Trace verworfen.

## Ergebnisse

Die Ergebnisse der Strukturanalyse werden durch UML-Klassendiagramme dargestellt, in denen die Kandidaten annotiert sind, die den Grad der Übereinstimmung der Entwurfsmusterimplementierungen mit den Strukturmustern anzeigen. Abbildung 2.11 veranschaulicht einen Teil der Ergebnisse der Strukturanalyse einer praktischen Anwendung von Wendehals ([Wen07], Abschnitt 6.3.2). In diesem Fall wurde eine Strategy-Implementierung sowohl als Strategy- als auch als State-Kandidat identifiziert.

Die Ergebnisse der Verhaltensanalyse werden in Tabellen über das Verhältnis der akzeptierten zu den nicht-akzeptierten Traces dokumentiert. Eine Beispieldatenbank ist in Tabelle 2.2 zu sehen. Der in Abbildung 2.11 beschriebene Strategy-Kandidat taucht hier unter der Bezeichnung **Strategy 2** auf. Die Zeilen stellen die

Tabelle 2.2: Ergebnisse einer Verhaltensanalyse

Entwurfsmusterimpl.	Kandidat	akzept. Traces	verworfen Traces	nicht akzept. Traces	durchschnittl. Tracelänge
Strategy 1	Strategy	195	14	69	5,6
	State	0	1236	431	0
Strategy 2	Strategy	2	10	4	48,5
	State	0	22	6	0
Strategy 3	Strategy	12	0	0	16,5
	State	0	156	22	0

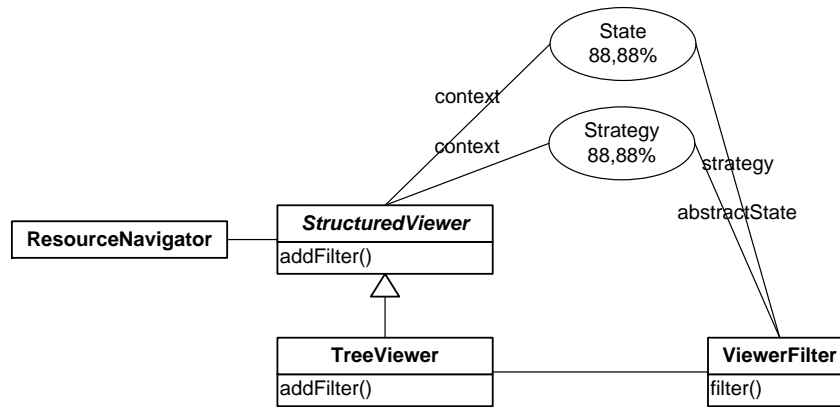


Abbildung 2.11: Ergebnisse einer Strukturanalyse

Kandidaten aus der Strukturanalyse dar. Es gab drei unterschiedliche Entwurfsmustervorkommen, die alle sowohl als Strategy- als auch als State-Kandidaten identifiziert wurden. Hohe Werte in der Spalte für akzeptierte Traces sprechen für eine wirkliche Entwurfsmusterimplementierung, eine hohe Anzahl verworfener Traces spricht für ein False-Positive. In diesem Fall ist beispielsweise der State-Kandidat **Strategy 1** mit sehr hoher Wahrscheinlichkeit ein False-Positive, da die Anzahl der verworfenen Traces sehr hoch ist. Der zugehörige Strategy-Kandidat wurde dagegen deutlich bestätigt, da die Anzahl der akzeptierten Traces vergleichsweise hoch ist. Ein Trace wird als verworfen bezeichnet, wenn der zugehörige Automat in einem verwerfenden Zustand endete. Von einem nicht akzeptierten Trace spricht man hingegen, wenn der Automat weder in einem verwerfenden Zustand noch in einen akzeptierenden Zustand gelangt ist. Das bedeutet, dass man nicht weiß, ob der Trace noch akzeptiert werden würde, wenn die Sequenz der Methodenaufrufe länger wäre.

Der kombinierte Prozess aus Struktur- und Verhaltensanalyse resultiert also in viel präziseren Ergebnissen, als die Strukturanalyse sie zuvor alleine ermöglichte.

# 3 Konzept der Erweiterungen

Dieses Kapitel gibt einen Überblick über die im Rahmen dieser Arbeit umgesetzte Erweiterung der Verhaltensanalyse. Zunächst wird ein weiteres Mal auf die Problemstellung im Detail eingegangen. Danach werden die neuen Notationselemente eingeführt. Als letztes wird erläutert, wie die Transformation der neuen Elemente in die Automaten zur Verhaltensanalyse geschieht und wie die erweiterte Verhaltensanalyse abläuft.

## 3.1 Problemstellung

Die bisher verwendete Spezifikationsprache für Verhaltensmuster, die in Kapitel 2.2.2 beschrieben wurde, zeigte bei der praktischen Anwendung einige Probleme. Die zur Verfügung stehenden Verhaltensmuster-elemente erlauben nicht die Berücksichtigung von Objektmengen, wie sie zum Beispiel beim Observer- oder beim State-Entwurfsmuster auftreten können.

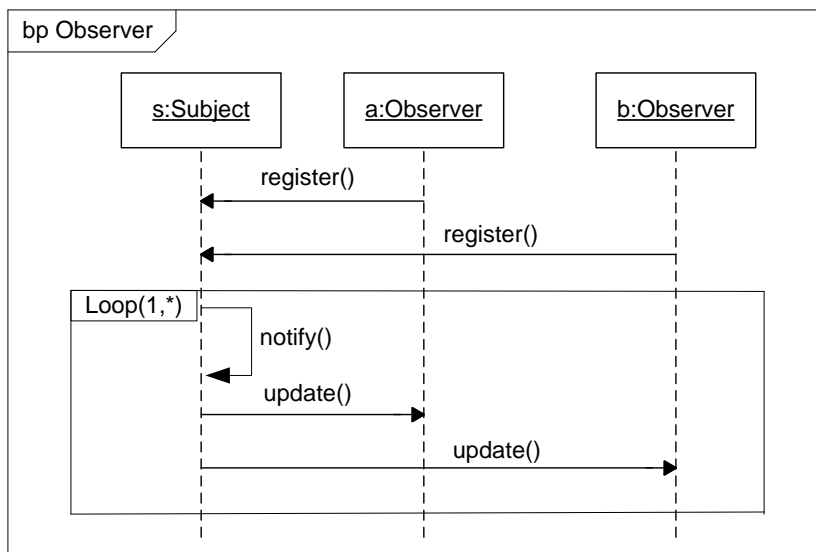


Abbildung 3.1: Observer-Verhaltensmuster in bisheriger Syntax

Abbildung 3.1 zeigt das Observer-Verhaltensmuster, das mit der bisherigen Spezifikationsprache realisiert wurde. Es gibt zwei Observer **a** und **b**, die sich bei dem

Subjekt *s* registrieren. Danach werden jedes mal, nachdem das Subjekt `notify()` aufruft, beide Observer aktualisiert.

Eine Observer-Implementierung muss zur Laufzeit genau zwei Observer-Objekte instanziiert haben, um zu diesem Verhaltensmuster konform zu sein. Werden zur Laufzeit drei Observer verwendet, wird das Muster anhand dieses Verhaltensmusters nicht erkannt, da sich die Sequenz der Methodenaufrufe von einer Sequenz mit weniger Observern unterscheidet: die `register()`-Methode wird drei mal hintereinander aufgerufen, anstatt zwei mal, ebenso wie die `update()`-Methode. Dasselbe Problem tritt auf, wenn man beispielsweise eine Instanzsituation mit nur einen Observer hat.

Um diese Schwachstelle zu beheben, ist es sinnvoll, die Spezifikationsprache um ein Objekt zu erweitern, dem bei der Verhaltensanalyse eine beliebig große Anzahl Instanzen eines Typs zugeordnet werden kann. Mithilfe eines solchen Objektes können Observer-Muster-Instanzen mit beliebig vielen Observern zur Laufzeit erkannt werden. Das derart modifizierte Observer-Verhaltensmuster wird in Abschnitt 3.3.1 vorgestellt.

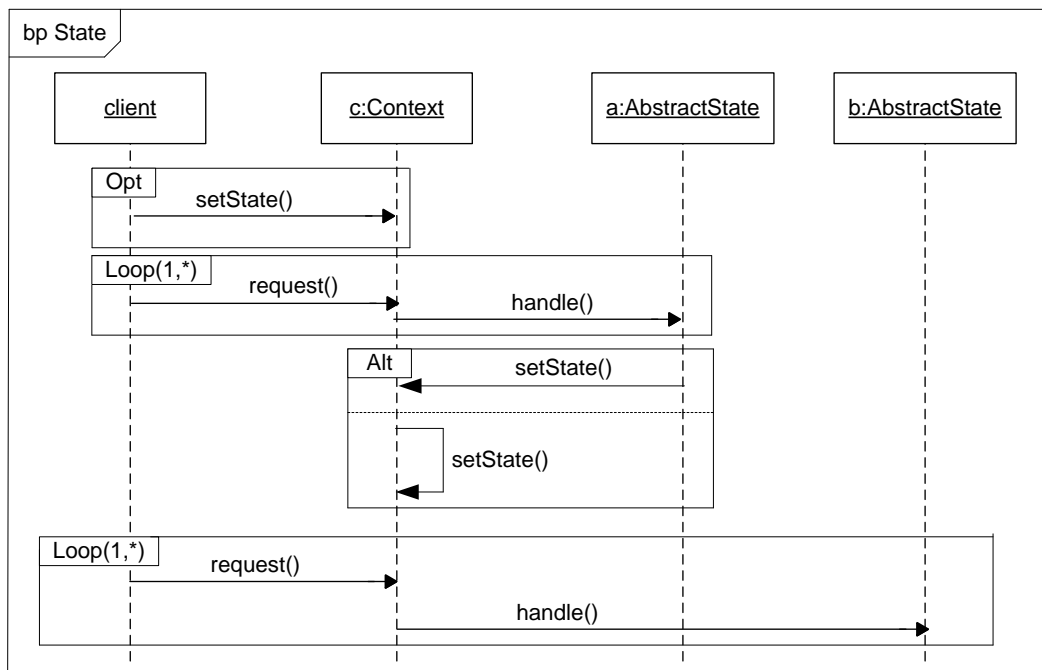


Abbildung 3.2: State-Verhaltensmuster in bisheriger Syntax

Beim State-Verhaltensmuster tritt ein ähnliches Problem auf. Abbildung 3.2 zeigt das State-Verhaltensmuster, in der bisherigen Spezifikationsprache. Es gibt zwei Zustände *a* und *b* mit dem Typ `AbstractState`. Zuerst werden beliebig Anfragen (`request()`) an den Kontext *c* über `handle()` an das Objekt *a* delegiert. Danach wird entweder durch den alten Zustand *a* oder durch den Kontext mit

`setState()` ein neuer Zustand gesetzt. Alle Anfragen, die danach an den Kontext gesendet werden, werden nun an den neuen Zustand `b` delegiert.

Mit diesem Verhaltensmuster lässt sich eine State-Implementierung erkennen, die zur Laufzeit über genau zwei Zustände verfügt. Um eine Umsetzung mit mehr als zwei Zuständen erkennen zu können, muss man das Verhaltensmuster um dieselbe Anzahl Zustände ergänzen. Es können aber immer nur Musterinstanzen mit einer fest vorgegebenen Anzahl von Zuständen erkannt werden. Um dieses Problem zu lösen, muss man eine Möglichkeit finden, um eine beliebige Anzahl an Zustandswechseln modellieren zu können.

## 3.2 Erweiterung der Spezifikationsprache

Zur Lösung der in Abschnitt 3.1 beschriebenen Probleme wurde die Spezifikationsprache für Verhaltensmuster um so genannte *Mengenobjekte* erweitert. Außerdem wurde das neue Element *Each-Fragment* sowie die neue Nachrichteneigenschaft *self call* aufgenommen, die in Kombination mit Mengenobjekten verwendet werden können. Des Weiteren wurde die Spezifikationsprache erweitert, um die Möglichkeit Argumente von Nachrichten zu definieren sowie mit einem *Zuweisungselement* ausgestattet.

### 3.2.1 Mengenobjekte

Als Ergänzung zu den typisierten und nicht-typisierten Objekten wurde der Spezifikationsprache für Verhaltensmuster ein mengenwertiges Verhaltensmusterobjekt hinzugefügt. Mithilfe eines mengenwertigen Verhaltensmusterobjektes kann einem Objekt bei der Verhaltensanalyse eine beliebig große Anzahl konkrete Instanzen von Objekten aus dem Kandidaten zugeordnet werden. Die Instanzen sind dabei alle von demselben Typ. Ein *Mengenobjekt* wird im Verhaltensmuster, wie in Abbildung 3.3, als ein mit einem zweiten Rahmen hinterlegtes Objekt dargestellt.

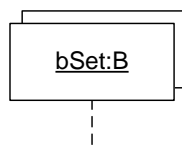


Abbildung 3.3: Mengenwertiges Verhaltensmusterobjekt, sog. Mengenobjekt

Mengenobjekte repräsentieren mindestens ein Objekt. Das heißt entgegen der mathematischen Definition von Mengen, darf ein Mengenobjekt in einem Verhaltensmuster nicht für eine leere Menge stehen.

Eine eingehende Nachricht zu einem Mengenobjekt bedeutet, dass die entsprechende Methode auf einem Objekt der Menge aufgerufen wird. In dem in Abbildung 3.4 auf der linken Seite dargestellten Beispiel ruft das Objekt `a` vom Typ

A die Methode  $m$  auf einem Objekt in der Menge  $bSet$  auf. Dabei sind alle Objekte in der Menge  $bSet$  von Typ B. Eine aus einem Mengenobjekt ausgehende

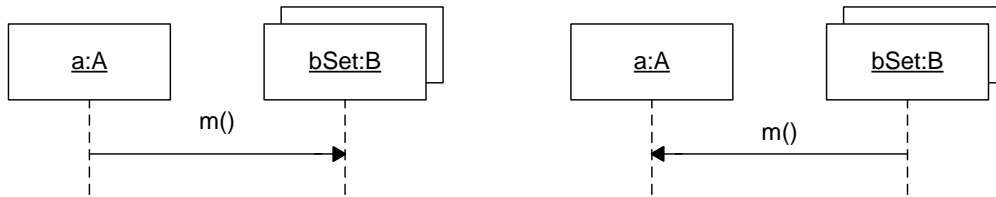


Abbildung 3.4: Nachrichten von und zu Mengenobjekten

Nachricht bedeutet, dass ein Objekt aus der Menge die entsprechende Methode auf dem Zielobjekt aufruft, wie in Abbildung 3.4 rechts.

Die Spezifikationsprache erlaubt es nicht, mehrere Mengen des gleichen Typs zu verwenden. Außerdem besteht nicht die Möglichkeit, einzelne Objekte in den Mengen gezielt zu referenzieren. Dies sind mögliche Erweiterungen, die in Kapitel 7.2 diskutiert werden.

### 3.2.2 Each-Fragment

Um eine Sequenz von Nachrichten zu modellieren, die alle konkreten Objekte eines Mengenobjektes betrifft, wurde ein weiteres Notationselement, das *Each-Fragment*, eingeführt. Ein Each-Fragment ist ein spezielles Loop-Fragment (siehe Abschnitt 2.2.2), das dafür sorgt, dass ein bestimmter Methodenaufruf auf allen Objekten einer Menge erfolgt. Die Abbildung 3.5 zeigt ein Each-Fragment, das bewirkt, dass das Objekt  $a$  die Methode  $m()$  auf allen Objekten der Menge  $bSet$  aufruft. Umgekehrt kann man mithilfe eines Each-Fragments auch die

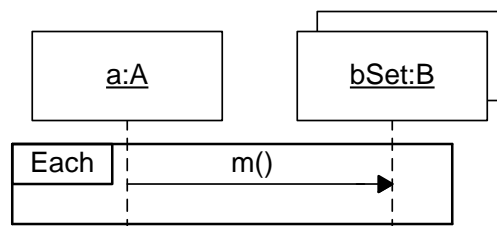


Abbildung 3.5: Each-Fragment

Situation modellieren, in der alle Objekte einer Menge eine bestimmte Methode auf einem anderen Objekt aufrufen. Die Benutzung von Each-Fragmenten ist nur in Kombination mit Mengenobjekten sinnvoll. Die Semantik für den Fall eines Each-Fragmentes mit einer Nachricht von einem Mengenobjekt zu einem anderen Mengenobjekt ist undefiniert.

### 3.2.3 Selbstaufrufe

Objekte innerhalb eines Mengenobjektes können, wie normale Objekte auch, Methoden auf dem eigenen Mengenobjekt aufrufen. Hierbei gibt es die Möglichkeit, zu entscheiden, ob das Objekt die Methode auf sich selbst aufruft, oder auf einem anderen Objekt, das ebenfalls in der Menge enthalten ist. Dies modelliert man mit einer zusätzlichen Nachrichteneigenschaft *self call*. Diese kann entweder den Wert `true` oder `false` annehmen. Ist *self call* `true`, so wird die Nachricht im Verhaltensmuster mit `self`, andernfalls mit `other` gekennzeichnet. Die Abbildung 3.6 zeigt beide Varianten. Der linke Ausschnitt eines Verhaltensmusters drückt aus, dass ein Objekt aus der Menge `bSet:B` die Methode `m()` auf sich selbst aufruft. Der rechte Verhaltensmusterschnitt zeigt einen Fall, in dem ein Objekt aus der Menge `bSet:B` die Methode `m()` auf irgendeinem anderen Objekt derselben Menge aufruft.

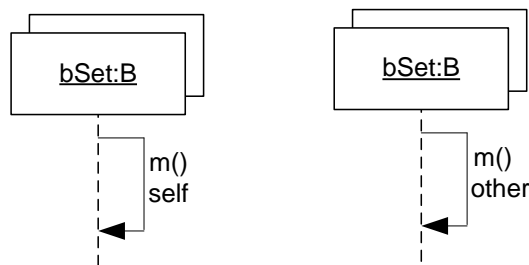


Abbildung 3.6: Nachrichteneigenschaft *self call*

### 3.2.4 Argumente

Methoden können Parameter haben, die bei einem Methodenaufruf mit Argumenten belegt werden. Die bei einem Methodenaufruf benutzten Argumente wurden bisher an den Nachrichten nicht dargestellt, sind aber für die Erweiterung notwendig. Zu diesem Zweck besteht die Möglichkeit, beim Anlegen einer Nachricht jedem Argument einen Namen zu geben. Die Instanz des Parameters wird damit durch den Methodenaufruf gebunden. So entsteht die Möglichkeit, ein Objekt, das als Argument übergeben wurde, später zu referenzieren.

In Abbildung 3.7 wird mit dem Methodenaufruf `m(b1)` das Objekt `b1` als Argument übergeben.

### 3.2.5 Zuweisungen

Üblicherweise ist in Sequenzdiagrammen und damit auch in Verhaltensmustern ein Objekt einer Lifeline zugeordnet. Mithilfe einer Zuweisung kann man die Identität eines Objektes ändern. Eine Zuweisung wird im Verhaltensmuster auf der Lifeline eines Objektes dargestellt. Auf der linken Seite der Zuweisung steht der Name

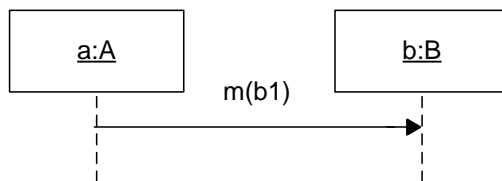


Abbildung 3.7: Methodenaufruf mit Argument

des Objekts, zu dem die Lifeline aktuell gehört, also das Verhaltensmusterobjekt, dem man ein anderes Instanzobjekt aus dem Trace zuweisen möchte. Auf der rechten Seite steht der Name der neuen Instanz. Die neue Instanz muss bereits als Argument in einem vorherigen Methodenaufruf gebunden worden sein, wie in Abschnitt 3.2.4 beschrieben. Außerdem müssen die Typen in der Zuweisung konform sein.

Abbildung 3.8 zeigt ein Beispiel für die Anwendung einer Zuweisung. Dort wird die Methode `m` von dem Objekt `b` vom Typ `B` aufgerufen und das Objekt `b1` als Argument übergeben. Als nächstes folgt das Zuweisungselement `b:=b1`. Die Zuweisung bewirkt, dass der nächste Methodenaufruf `n()` nicht auf dem Objekt `b`, sondern auf dem vorher gebundenen Objekt `b1` vom Typ `B` aufgerufen wird. Der Methodenaufruf, in dem ein Objekt gebunden wird, muss nicht auf dem Objekt erfolgen, dessen Identität später durch eine Zuweisung geändert wird.

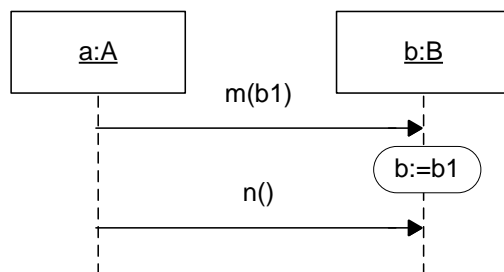


Abbildung 3.8: Zuweisungsobjekt

### 3.3 Beispiel-Verhaltensmuster

Im Folgenden werden die Verhaltensmuster zu den Entwurfsmustern Observer, Chain of Responsibility und State unter Einbeziehung der neuen Notationselemente erläutert.

### 3.3.1 Observer

Beim Observer-Muster geht es darum, eine Abhängigkeit zwischen mehreren Beobachtern und einem Subjekt zu definieren. Dabei sollen, sobald das Subjekt seinen Status ändert, all seine Beobachter informiert werden und sich automatisch aktualisieren [GHJV95].

Das zugehörige Verhaltensmuster wird in Abbildung 3.9 dargestellt. Das Objekt `s:Subject` repräsentiert das zu beobachtende Subjekt, die Beobachter werden durch ein Mengenobjekt `oSet:Observer` dargestellt. Zuerst registrieren sich beliebig viele Observer bei dem Subjekt, indem sie die Methode `register()` aufrufen. Dabei werden die entsprechenden Objekte der Menge hinzugefügt. Von nun an wird jedes Mal, wenn das Subjekt `notify()` aufruft, von allen registrierten Objekten die Methode `update()` aufgerufen. Dabei sorgt das Each-Fragment dafür, dass die Nachricht an alle Objekte geht, die zuvor der Menge `o:Observer` hinzugefügt wurden.

Bei dem alten Observer-Verhaltensmuster aus Abbildung 3.1 musste man für jeden Observer ein eigenes Objekt darstellen, jedes Observer-Objekt beim Subjekt registrieren und vom Subjekt aus aktualisieren. Mithilfe der neuen Spezifikationselemente kann man eine Objektmenge spezifizieren, die für eine beliebige Anzahl von Observern steht und damit Instanzen von Observer-Implementierungen mit beliebig vielen Observern erkennen.

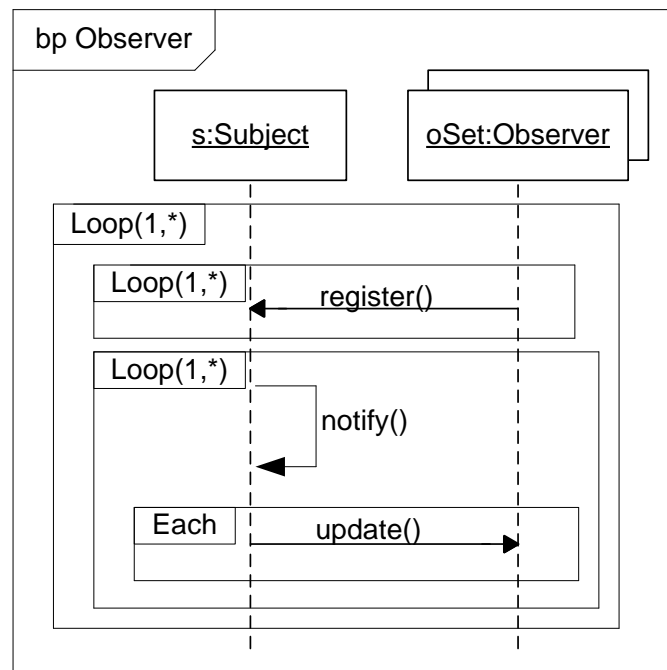


Abbildung 3.9: Neues Observer-Verhaltensmuster

### 3.3.2 Chain of Responsibility

Eine Chain Of Responsibility kann benutzt werden, um den Sender einer Anfrage von ihrem Empfänger zu entkoppeln, indem man mehr als einem Objekt die Möglichkeit gibt, die Anfrage zu bearbeiten [GHJV95]. Dabei werden die Empfänger hintereinander geschaltet und die Anfrage die Kette entlang geleitet, bis ein Objekt sie bearbeiten kann.

Das Verhaltensmuster zur Chain of Responsibility wird in Abbildung 3.10 gezeigt. Der Aufrufer `client` ist ein untypisiertes Objekt. Das Mengenobjekt `h:AbstractHandler` steht für die verketteten Empfänger, die die Anfrage bearbeiten sollen. Nachdem der Client die Methode `handleRequest(r)` auf einem Objekt aus dem Mengenobjekt aufgerufen hat, rufen die Objekte in der Menge nacheinander `handleRequest(r)` auf dem jeweils nächsten Objekt auf. Dabei wird durch die Nachrichteneigenschaft `other` sichergestellt, dass ein Objekt nicht sich selbst, sondern ein anderes Objekt aus der Menge aufruft. Dabei bleibt das Argument `r` gleich. Es kann auch sein, dass direkt das erste Element in der Kette der Handler die Anfrage bearbeiten kann, daher ist die Kardinalität des Loops  $(0,*)$ .

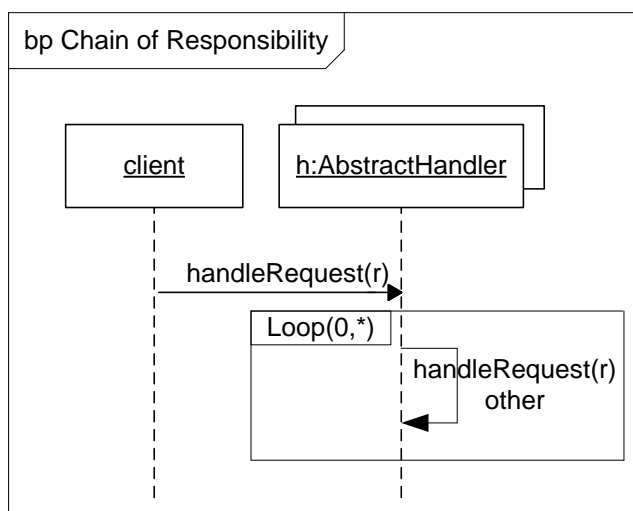


Abbildung 3.10: Chain-of-Responsibility-Verhaltensmuster

### 3.3.3 State

Das State-Muster erlaubt einem Objekt, sein Verhalten zu ändern, wenn sein Zustand wechselt [GHJV95]. Anfragen des Aufrufers werden jeweils an den aktuellen Zustand delegiert. Der aktuelle Zustand kann entweder durch den Kontext, oder durch den Zustand selbst geändert werden.

Abbildung 3.11 zeigt das Verhaltensmuster des State-Entwurfsmusters. Der Startzustand kann durch den Client gewählt werden. Dies wird durch eine op-

tionale Nachricht `setState(a)` an das Objekt `c:Context` spezifiziert. Die Delegation einer Anfrage durch `client` wird von dem Objekt `c:Context` übernommen. Der Client ruft beliebig oft, jedoch mindestens einmal `request` auf `c:Context` auf, woraufhin `c:Context` die Anfrage delegiert, indem es auf dem aktuellen Zustandsobjekt `a:AbstractState` die Methode `handle` aufruft. Tritt nun ein Zustandswechsel ein, wird der Folgezustand entweder durch den aktuellen Zustand oder durch das Kontext-Objekt, durch den Methodenaufruf `setState(s)` ausgewählt. Dies wird durch das Alternative-Fragment symbolisiert. Durch die Zuweisung `a:=s` wird dem Verhaltensmusterobjekt `a:AbstractState` eine andere Instanz `s` zugeordnet. Dadurch werden weitere Anfragen an einen anderen Zustand `s` delegiert. Im Gegensatz zu dem alten State-Verhaltensmuster aus Abbildung 3.2, bietet

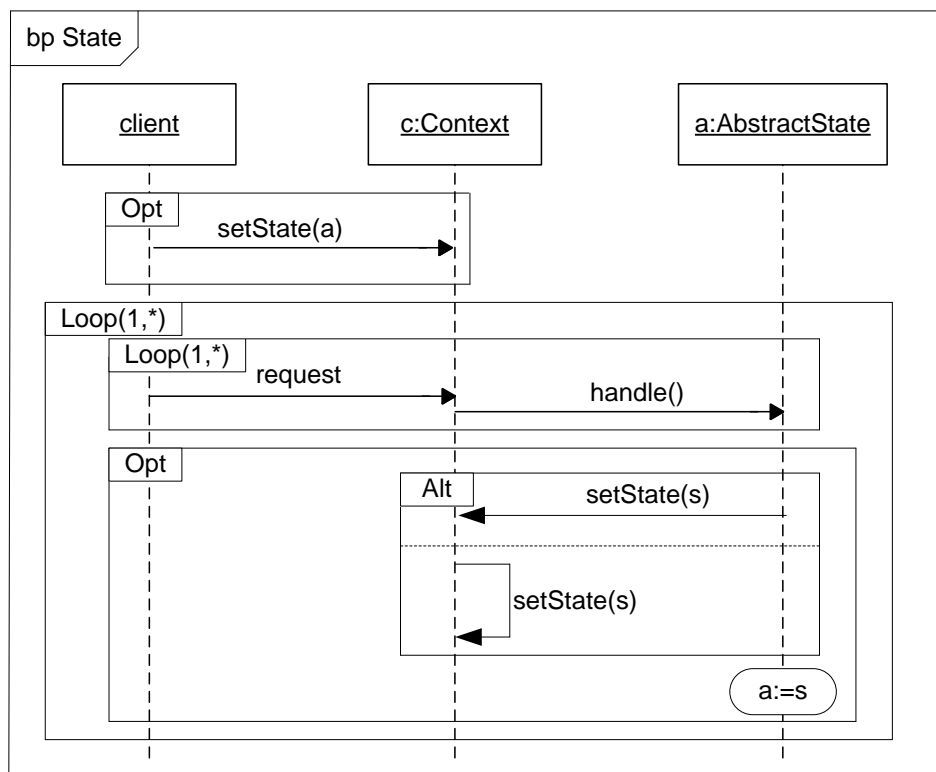


Abbildung 3.11: Neues State-Verhaltensmuster

das mit den neuen Spezifikationselementen dargestellte Verhaltensmuster also die Möglichkeit beliebig viele Zustandswechsel zu modellieren. Dadurch kann man State-Implementierungen mit einer beliebigen Anzahl an Zuständen erkennen.

## 3.4 Erweiterung der Transformation in Automaten

Um eine Verhaltensanalyse für die in Abschnitt 3.2 beschriebenen Erweiterungen durchführen zu können, müssen für die neuen Notationselemente geeignete Transformationsregeln definiert werden, anhand derer man die Elemente in die zur Mustererkennung verwendeten Automaten übersetzen kann<sup>1</sup>.

### 3.4.1 Transformationsregeln

Im Folgenden werden die verschiedenen Transformationsregeln für die Übersetzung der neuen Elemente in Automatenfragmente vorgestellt. Die Regeln für die bisherige Sprache findet man in [Wen07].

#### Transformationsregeln für Mengenobjekte

Abbildung 3.12 zeigt, wie das zugehörige Automatenfragment für eine Nachricht an ein Mengenobjekt aussieht. Die Nachricht wird auf einem Objekt aus der Menge aufgerufen. In der Beschriftung der Transition steht das Schlüsselwort **Set** für eine Menge. Der Typ den die Objekte der Menge haben, wird als Index angegeben. Ist das Element vorher noch nicht in der Menge enthalten, wird es hinzugefügt. Dies wird durch den Ausdruck unter der Transition mithilfe des Vereinigungssymbols  $\cup$  aus der Mengenlehre beschrieben. Die umgekehrte Richtung funktioniert analog.

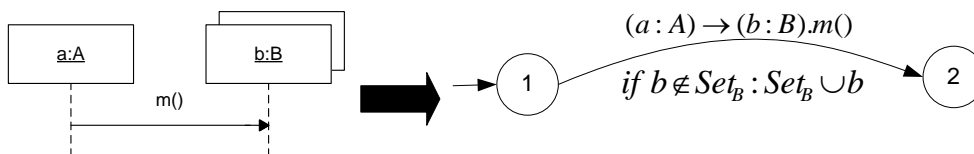


Abbildung 3.12: Transformationsregel für Mengenobjekte

#### Transformationsregeln für Each-Fragmente

Die Transformationsregel für ein Each-Fragment wird in Abbildung 3.13 dargestellt. Die Nachricht, die in dem Each-Fragment enthalten ist, wird auf jedem Objekt aufgerufen, das in der Menge enthalten ist. Deswegen wird hier der Allquantor  $\forall$  aus der Mengenlehre benutzt. Auch hier ist die Funktionsweise in der umgekehrten Richtung analog.

<sup>1</sup>Bei der Notation handelt es sich um eine Veranschaulichung der Konzepte aus Kapitel 3.2 für Automaten. In der Realität geschieht die Analyse algorithmisch, die Automaten sind nur ein Zwischenprodukt.

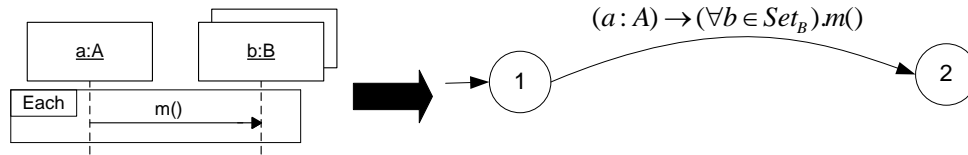


Abbildung 3.13: Transformationsregel für Each-Fragmente

### Transformationsregeln für Selbstaufrufe

Die Abbildungen 3.14 und 3.15 zeigen die Transformation von Nachrichten mit der Property `self call` für die Fälle `self call = true` und `self call = false`. Im ersten Fall ruft ein Objekt sich selbst auf. Dies wird durch die Namensgleichheit der Objekte `b` sichergestellt. Im zweiten Fall, ruft ein Objekt ein anderes Objekt auf, wobei beide Objekte in derselben Menge sind. Die Transition im Automaten für den Fall `self call = false` wird zusätzlich mit einer Ungleichung gekennzeichnet, die sicherstellt, dass es sich nicht um dasselbe Objekt handelt. Im Beispiel wird dies durch die Ungleichung `other ≠ b` hervorgehoben.

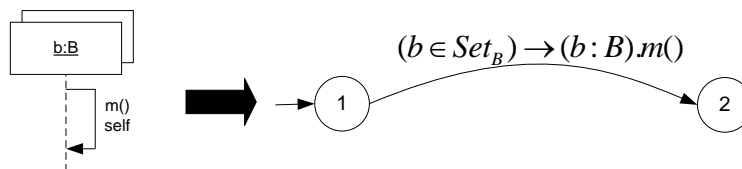


Abbildung 3.14: Transformationsregel für `self call = true`

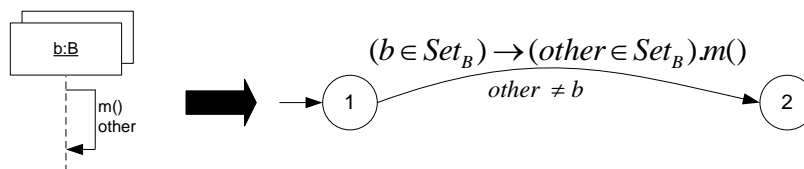


Abbildung 3.15: Transformationsregel für `self call = false`

### Transformationsregeln für Zuweisungen

Eine Zuweisung wird im Automaten unter der Transition der vorherigen Nachricht dargestellt. Das zugehörige Automatenfragment wird in Abbildung 3.16 dargestellt. Der Methodenaufruf, der auf die Zuweisung `b:=b1` folgt, wird anstatt auf der Instanz `b:B` auf der neu zugewiesenen Instanz `b1:B` ausgeführt.

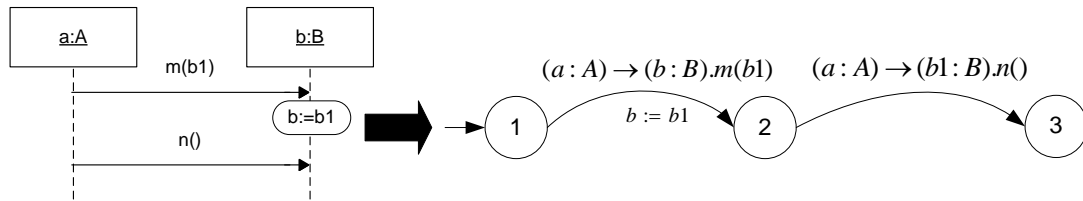


Abbildung 3.16: Transformationsregel für Zuweisungen

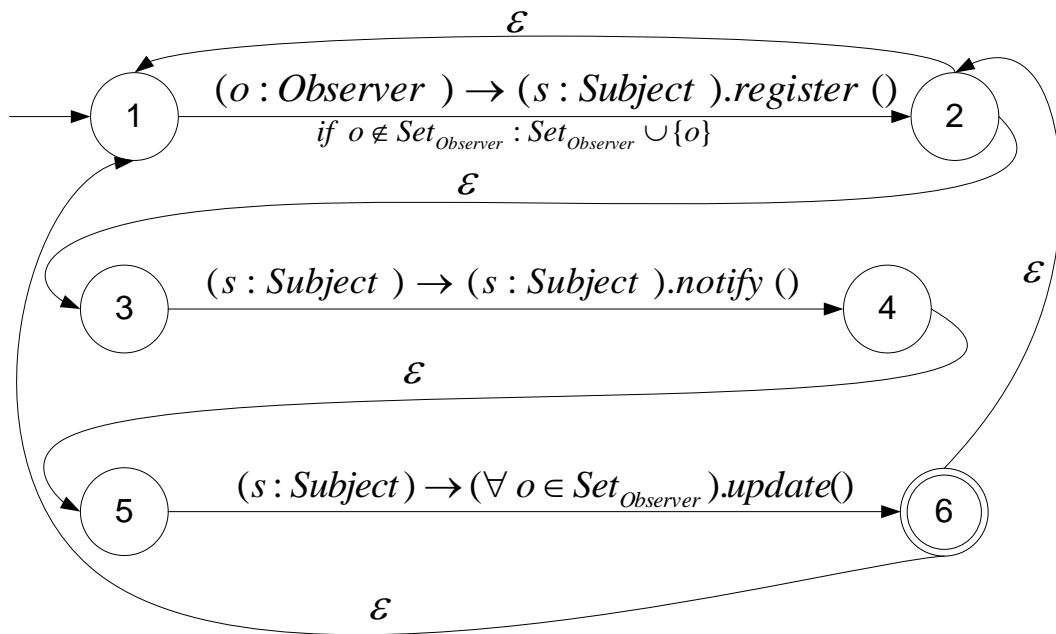


Abbildung 3.17: Automat aus dem neuen Observer-Verhaltensmuster

### 3.4.2 Beispielautomat

Abbildung 3.17 zeigt den vereinfachten nichtdeterministischen Automaten, der aus dem Observer-Verhaltensmuster (siehe Abbildung 3.9) generiert wird. Die Vereinfachung besteht darin, dass im Sinne einer übersichtlicheren Darstellung auf einige  $\varepsilon$ -Transitionen verzichtet wurde, die bei der Konkatenation der Automatenfragmente entstehen. Im Gegensatz zu dem Automaten aus dem alten Verhaltensmuster des Observer-Entwurfsmusters (Abbildung 2.10, Abschnitt 2.2.2), wurden hier die neuen Transformationsregeln aus Abschnitt 3.4 verwendet. Die erste Transition von Zustand 1 zu Zustand 2 beschreibt, wie die Observer sich bei dem Subjekt registrieren. Ein Objekt `o` vom Typ `Observer` ruft die Methode `register()` auf dem Subjekt `s` auf. Befindet sich `o` noch nicht in der Menge `SetObserver`, wird es ihr an dieser Stelle hinzugefügt. Eine  $\varepsilon$ -Transition führt von Zustand 2 zurück in Zustand 1. Dadurch können sich beliebig viele Observer registrieren. Als nächstes ruft `s:Subject` auf sich selbst die Methode `notify` auf. Dies entspricht im Automaten dem Zustandsübergang von Zustand 3 zu Zustand 4. Daraufhin müssen sich alle registrierten Observer aktualisieren. Dies wird durch die Transition von Zustand 5 zu Zustand 6 sichergestellt. Die Transition ist aus dem Each-Fragment des Verhaltensmusters entstanden und drückt durch den Allquantor aus, dass die Methode `update` nacheinander auf allen Objekten der Menge aufgerufen wird. Danach gelangt man in den akzeptierenden Zustand 6. Durch die Loop-Fragmente, kann man über die  $\varepsilon$ -Transitionen entweder wieder in Zustand 1 (Registrierung weiterer Objekte) oder Zustand 2 (erneuter Aufruf von `notify`) gelangen.

## 3.5 Erweiterung des Analysevorgangs

In diesem Abschnitt wird der Analysevorgang der verhaltensbasierten Entwurfsmustererkennung mit den erweiterten Automaten am Beispiel des Observer-Musters und des Kandidaten aus Abbildung 2.3 erläutert.

Bei der Erkennung werden zunächst im Automaten die Variablen für die Klassen und Methoden gegen die konkreten Klassen und Methoden des Kandidaten ausgetauscht. Für das konkrete Beispiel entsteht der Übergangsautomat, der in Abbildung 3.18 dargestellt wird. `Observer` wurde gemäß Tabelle 2.2.2 durch `WeatherStation`, `Subject` durch `WeatherSensor` und `register()` durch `attach()`, ersetzt.

Das zu untersuchende Beispielprogramm stellt eine Situation dar, in der verschiedene Wetterstationen einen Sensor beobachten sollen. Es gibt ein Objekt `ws` vom Typ `WeatherSensor` und zwei Wetterstationen `usa` und `g` vom Typ `WeatherStation`, bzw. von den konkreten Typen `USAWeatherStation` und `GermanWeatherStation`. Der Kandidat wurde bereits in Abbildung 2.3 dargestellt.

Abbildung 3.19 zeigt einen Beispieltrace des Programms mit einer Sequenz von Methodenaufrufen, der zu einer Instanz des Kandidaten aufgezeichnet wurde. Zuerst rufen die beiden Objekte vom Typ `WeatherStation` die `attach`-Methode auf

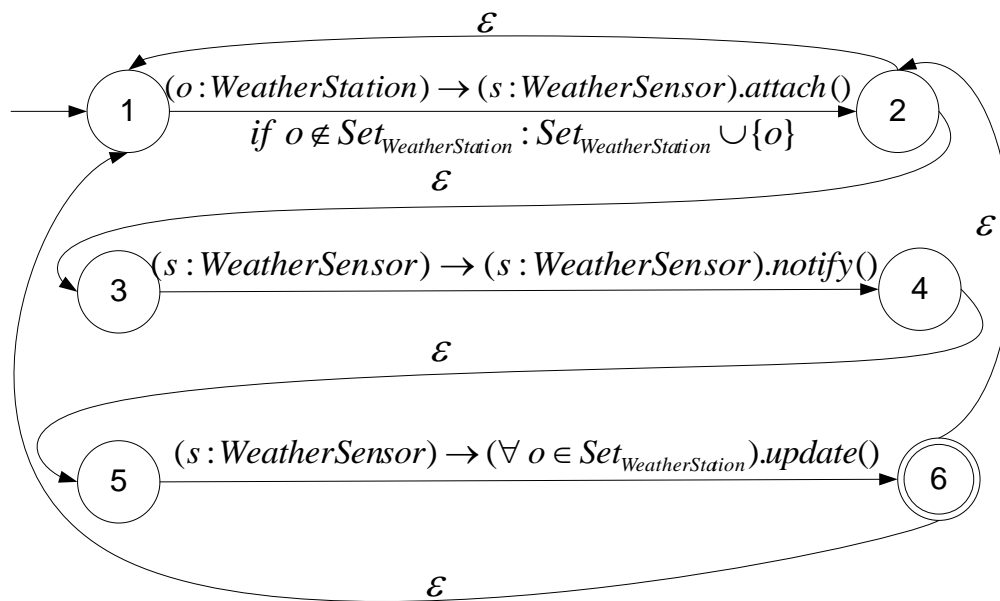


Abbildung 3.18: Übergangsautomat für einen Kandidaten des Observer-Musters

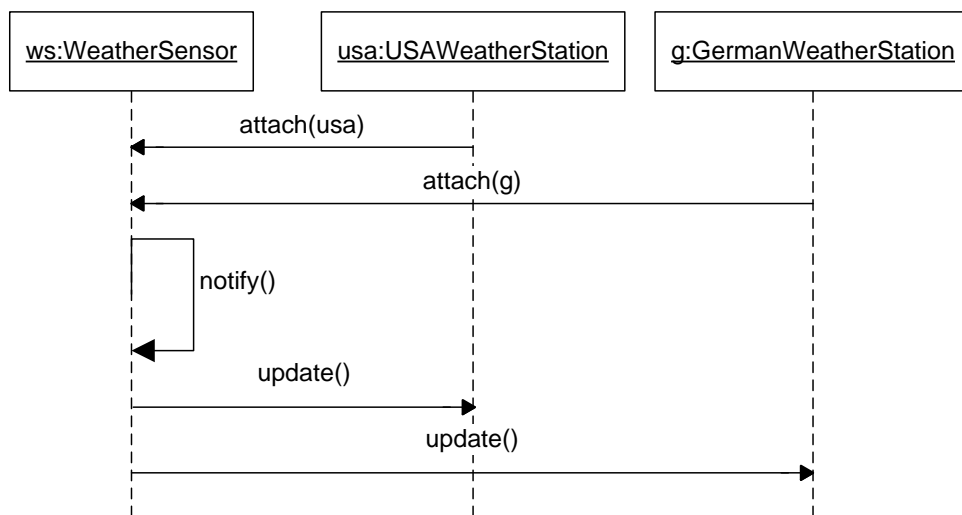


Abbildung 3.19: Beispiel-Trace

dem `WeatherSensor` auf. Nachdem dieser ein `notify()` ausgeführt hat, ruft er die Methode `update` beider Wetterstationen auf.

Der Trace wird zur Programmlaufzeit aufgezeichnet. Der Automat konsumiert die im Trace aufgelisteten Methodenaufrufe und bindet dabei die Variablen. Dabei entsteht aus dem Übergangautomat ein endgültiger Automat. Abbildung 3.20 zeigt eine Aufzeichnung der Zustände, wie sie beim Erkennungsvorgang entsteht, und stellt somit den Ablauf der Erkennung dar. Der Automat beginnt im Anfangszustand 1. Beim Schalten der ersten Transition wird das Objekt `usa` an die Variable `o` sowie das Objekt `ws` an die Variable `s` gebunden. Außerdem wird das Objekt `usa` der Menge  $Set_{WeatherStation}$  hinzugefügt. Die Transition von Zustand 1 nach Zustand 2 schaltet. Danach schaltet nochmal dieselbe Transition, jedoch wird diesmal das Objekt `g` an die Variable `o` gebunden und der Menge hinzugefügt. Dies entspricht ebenfalls der Transition von Zustand 1 in Zustand 2. Durch die  $\varepsilon$ -Transition gelangt der Automat in Zustand 3. Als nächstes folgt der `notify`-Aufruf, durch den der Automat in Zustand 4 gelangt. Nun konsumiert der Automat die beiden Aufrufe der `update()`-Methoden von `usa` und `g`. Da die Menge  $Set_{WeatherStation}$  an dieser Stelle genau aus diesen beiden Objekten besteht, gelangt der Automat in den akzeptierenden Zustand 6.

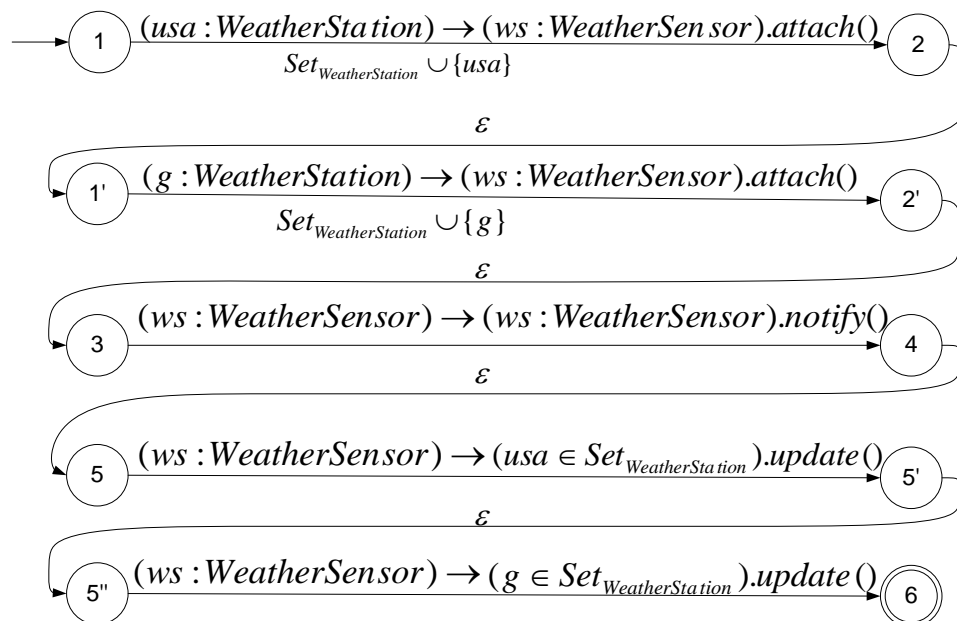


Abbildung 3.20: Aufzeichnung der Zustände beim Erkennungsvorgang

Der Trace ist also konform zum Observer-Verhaltensmuster. Dies ist ein Hinweis darauf, dass sich das Beispielprogramm an dieser Stelle wie eine Instanz des Observer-Entwurfsmusters verhält.



# 4 Umsetzung der Erweiterung

Das im vorherigen Kapitel beschriebene Konzept wurde als Erweiterung für RECLIPSE umgesetzt. Diese Umsetzung wird im Folgenden näher beschrieben. Dabei wird zuerst auf die technische Implementierung eingegangen. Im zweiten Teil wird gezeigt, wie sich die Umsetzung dem Benutzer darstellt.

## 4.1 Technische Umsetzung

In diesem Abschnitt wird zunächst auf die Erweiterung der Verhaltensmusterspezifikation um die neuen Notationselemente eingegangen. Danach werden die Modifikationen in der Transformation der zur Analyse benötigten Automaten beschrieben. Als letztes wird dargelegt, wie der Prozess der Verhaltensanalyse angepasst wurde.

### 4.1.1 Verhaltensmusterspezifikation

Der Editor zur Erstellung der Verhaltensmuster musste um die Möglichkeit der Erstellung der neuen Notationselemente, die in Kapitel 3.2 erläutert wurden, erweitert werden.

Dazu wurde das Metamodell für Verhaltensmuster aus Abbildung 2.6 angepasst. Abbildung 4.1 zeigt das neue Metamodell. Die im Rahmen dieser Arbeit vorgenommenen Änderungen sind blau gekennzeichnet. Das Element `BPObject` wurde um die Eigenschaft `isSet` erweitert, um Mengenobjekte zu realisieren. Ist das Attribut `isSet true`, dann handelt es sich bei dem Objekt um ein Mengenobjekt. Außerdem wurde ein neues Element `EachFragment` eingefügt. Das Each-Fragment ist ebenso wie z.B. das `LoopFragment` ein `CombinedFragment`. Die Nachrichten (`Message`) wurden um das Attribut `selfCall` erweitert. Des Weiteren werden nun bei den Nachrichten auch Argumente dargestellt. Daher hat `Message` nun eine Liste mit Argumenten vom Typ `BPArgument`. Zuletzt wurde ein Zuweisungsobjekt `BPAssignment` eingefügt. Assignments haben eine Variable auf der linken und eine auf der rechten Seite (`leftSide`, `rightSide`). Die linke Seite ist dabei ein `BPObject`, die rechte ein `BPArgument`. Sie werden ebenso wie Nachrichten an der Lifeline dargestellt und müssen sich in die Reihenfolge der Nachrichten einreihen. Deswegen erben sie von `Message`.

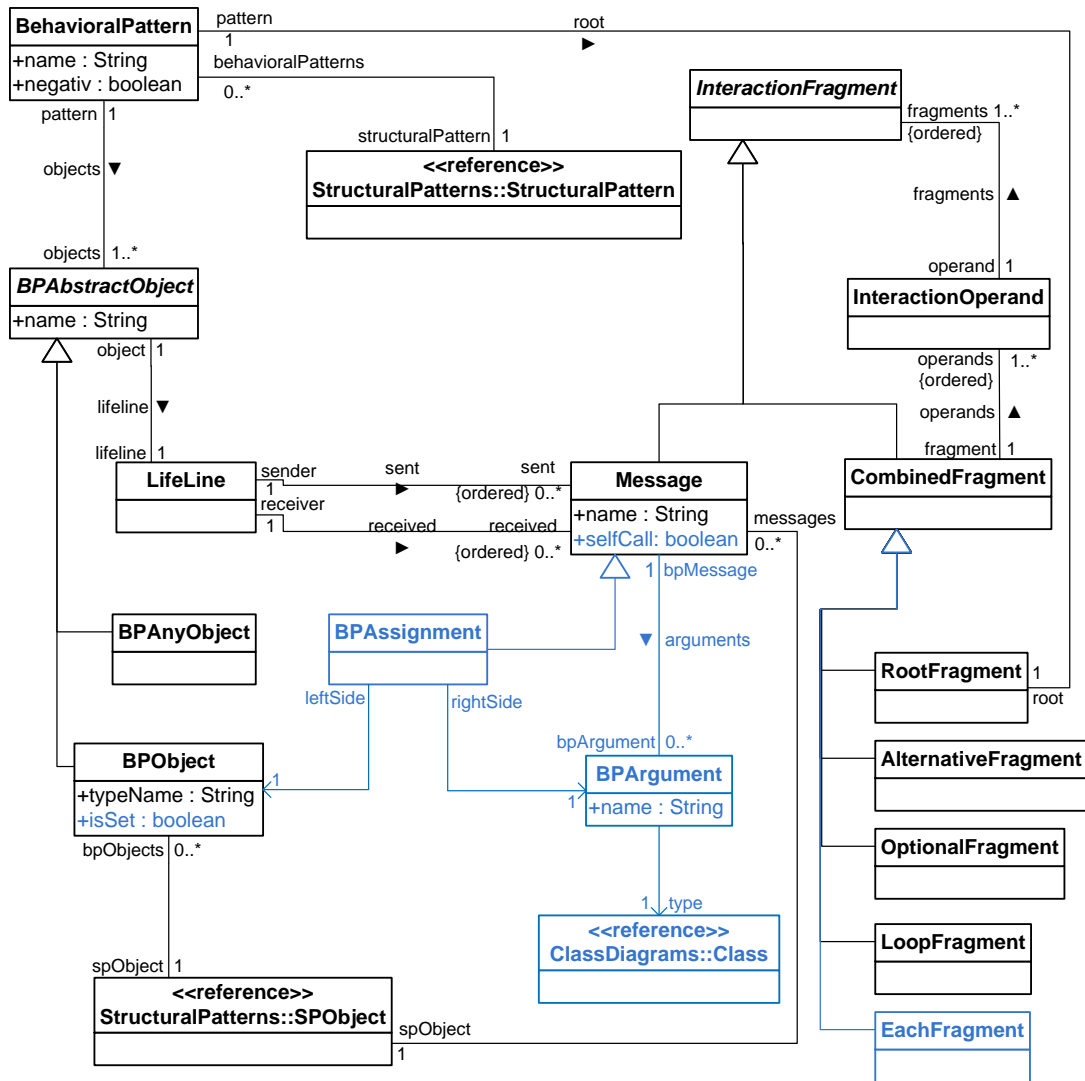


Abbildung 4.1: Erweitertes Metamodell für Verhaltensmuster

## 4.1.2 Automatentransformation

Aus den fertigen Verhaltensmustern lassen sich endliche Automaten generieren. Bei den Automaten handelt es sich um ein Zwischenprodukt, das für die spätere Verhaltensanalyse (siehe Abschnitt 4.1.3) verwendet wird.

Die Automaten werden in einem Verhaltensmusterkatalog im XML-Format gespeichert. Die DTD<sup>1</sup> musste für die erweiterte Spezifikationsprache entsprechend angepasst werden. In der folgenden DTD sind die im Rahmen dieser Arbeit vorgenommenen Erweiterungen ebenfalls blau gekennzeichnet.

```

<!ELEMENT BehavioralPatternsCatalog (BehavioralPatternEntry*)>

<!ELEMENT BehavioralPatternEntry (DFA, Trigger+)>
<!ATTLIST BehavioralPatternEntry name CDATA #REQUIRED
                                negative CDATA #IMPLIED>

<!ELEMENT DFA ((PermittedMethodCallSymbol|
                ProhibitedMethodCallSymbol|
                ProhibitedCallerSymbol)+, State+, Assignment*, Transition*,
                StartState, RejectingState)>

<!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
                typeName CDATA #REQUIRED
                set CDATA #IMPLIED
                forEach CDATA #IMPLIED>

<!ELEMENT Callee EMPTY>
<!ATTLIST Callee name CDATA #REQUIRED
                typeName CDATA #REQUIRED

```

<sup>1</sup>Document Type Definition, Schema-Definition z.B. für XML-Dokumente

```

        set          CDATA #IMPLIED
        forEach     CDATA #IMPLIED>

<!ELEMENT PermittedCaller EMPTY>
<!ATTLIST PermittedCaller name          CDATA #REQUIRED
                    typeName CDATA #IMPLIED
                    set          CDATA #IMPLIED
                    forEach     CDATA #IMPLIED>

<!ELEMENT State EMPTY>
<!ATTLIST State id          CDATA #REQUIRED
                    name     CDATA #IMPLIED
                    type     CDATA #REQUIRED>

<!ELEMENT Assignment EMPTY>
<!ATTLIST Assignment id          CDATA #REQUIRED
                    leftSide  CDATA #REQUIRED
                    rightSide CDATA #REQUIRED>

<!ELEMENT Transition EMPTY>
<!ATTLIST Transition previousStateId CDATA #REQUIRED
                    nextStateId     CDATA #REQUIRED
                    symbolId         CDATA #REQUIRED
                    assignmentId     CDATA #IMPLIED>

<!ELEMENT StartState EMPTY>
<!ATTLIST StartState id          CDATA #REQUIRED>

<!ELEMENT RejectingState EMPTY>
<!ATTLIST RejectingState id          CDATA #REQUIRED>

<!ELEMENT Trigger EMPTY>
<!ATTLIST Trigger callerName        CDATA #IMPLIED
                    callerTypeName  CDATA #IMPLIED
                    calleeName       CDATA #IMPLIED
                    calleeTypeName   CDATA #REQUIRED
                    methodName        CDATA #REQUIRED>
```

Neu hinzugekommen sind die Attribute `set` und `forEach` bei den Elementen `Caller`, `Callee` und `PermittedCaller`. Nimmt `set` den Wert `true` an, ist das entsprechende Element ein Mengenobjekt und die zugehörige Transition im Automaten muss mit Mengenoperationen angereichert werden. Diese sorgen dafür, dass das Element der in der Analyse betrachteten Menge hinzugefügt wird (vgl.

Transformationsregeln in Abschnitt 3.4.1).

Das Attribut `forEach` tritt nur in Verbindung mit Mengenobjekten auf und gibt an, ob der Ausdruck an der entsprechenden Transition einen Allquantor beinhalten muss. Ist bei dem Caller `forEach` `true`, so ist der Sender der Nachricht ein Mengenobjekt und es wird von allen Objekten, die das zugehörige Mengenobjekt enthält, die Nachricht aufgerufen. Ist hingegen bei einem Callee `forEach` `true`, dann ist der Empfänger ein Mengenobjekt und alle in der Menge enthaltenen Objekte empfangen die Nachricht.

Um die Belegung der oben beschriebenen Eigenschaft `self call` in der Automatenbeschreibung auszudrücken, verwendet man den Namen des entsprechenden Callee-Elements. Ist `self call` im Verhaltensmuster `false`, so ist im XML-Dokument der Wert des Attributs `name` „other“. „Other“ ist dabei ein Platzhalter, der für irgendein Objekt steht, das sich von dem im Caller angegebenen unterscheidet.

Für die Zuweisungen wurde die DTD um ein neues Element `Assignment` erweitert. Es besteht aus den Attributen `id`, `leftSide` und `rightSide`. `leftSide` repräsentiert das Objekt, dem die Lifeline aktuell gehört, während `rightSide` für das Objekt steht, das zugewiesen werden soll. Einer Transition kann über das Attribut `assignmentId` ein `Assignment` zugewiesen werden.

### 4.1.3 Analysevorgang

Die Anpassung des Analysevorgangs lässt sich in zwei Schritte untergliedern. Der erste Schritt ist die Erweiterung des Modells der verwendeten Automaten, danach muss der Algorithmus entsprechend angepasst werden.

#### Modell des Automaten

Für die Erweiterung des Analysevorgangs musste zunächst das Modell der Automaten überarbeitet werden. Abbildung 4.2 zeigt das Modell mit den entsprechenden Erweiterungen. Die im Rahmen dieser Arbeit vorgenommenen Änderungen wurden erneut blau gekennzeichnet.

Ein Automat (DFA) hat eine bestimmte Anzahl an Zuständen vom Typ `State`. Zustände sind entweder akzeptierend (`ACCEPTING`), verwerfend (`REJECTING`) oder nichts von beidem (`NON-ACCEPTING`). Ein Automat hat genau einen verwerfenden Zustand sowie einen Startzustand. Ein Zustand hat ein- und ausgehende Transitionen. Eine `Transition` hat immer einen vorherigen Zustand und einen folgenden Zustand. Für die Erweiterung der Analyse um Zuweisungen wurden Transitionen um ein Attribut `assignment` vom Typ `Assignment` erweitert.

Zustände können Referenzen auf ein oder mehrere `Token` haben (vgl. Kapitel 2.2.2). Für ein `Token` wird mit dem Attribut `moved:Boolean` festgehalten, ob es gerade in einen anderen Zustand gewechselt ist hat, oder nicht. Die Klasse `Token` wurde um zwei Mengen `setOfSets` und `setForEach` erweitert. Die Menge

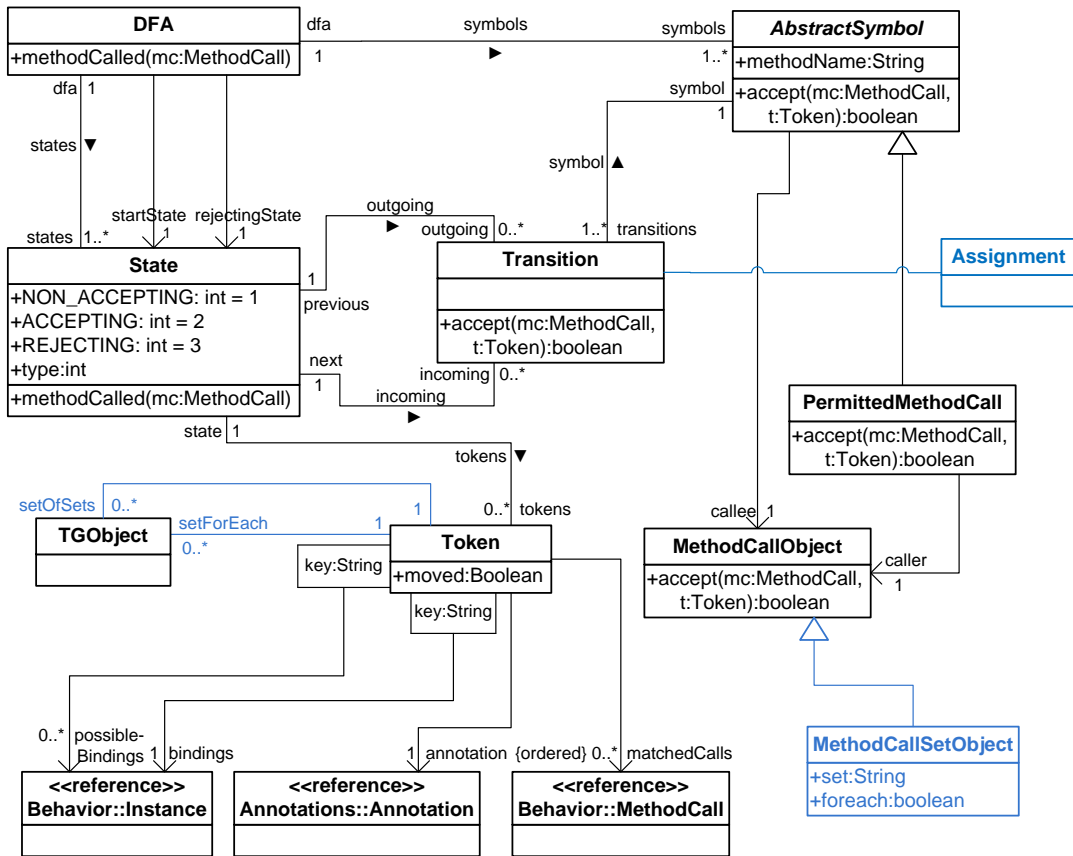


Abbildung 4.2: Erweitertes Modell des deterministischen Automaten

`setOfSets` enthält alle Mengen, die durch die im Verhaltensmuster spezifizierten Mengenobjekte definiert wurden. Diesen Mengen werden `TGObjects` (Trace-Graph-Objects<sup>2</sup>) hinzugefügt. Bei der Erkennung des Observer-Musters, werden beispielsweise alle Objekte, die „register()“ aufrufen in dieser Menge gesammelt. Die `setForEach`-Menge wird verwendet, um die Transitionen zu überprüfen, die für Methodenaufrufe, die innerhalb eines Each-Fragments spezifiziert wurden, generierten wurden. Auch diese Menge besteht aus `TGObjects`. Beim Observer-Muster entspricht dies der Menge der Objekte, auf denen ein „update()“ ausgeführt wurde.

Jede Transition referenziert ein Symbol vom Typ `AbstractSymbol`. Hier gibt es unterschiedliche Arten von Symbolen. Ein `PermittedMethodCall` ist ein Symbol, das einen erlaubten Methodenaufruf kennzeichnet, also einen Aufruf, bei dem eine Transition von einem gültigen Zustand in einen anderen gültigen Zustand schaltet. Aufrufende sowie aufgerufene Objekte innerhalb dieser Symbole heißen `MethodCallObject`. Außerdem wurde eine neue Klasse `MethodCallSetObject` hinzugefügt, die in den Transitionen auftauchende Mengenobjekte darstellt. `MethodCallSetObject` erbt von `MethodCallObject` und hat das zusätzliche Attribut `set:String`, welches den Namen der Menge angibt, sowie das Attribut `foreach`, welches festhält, ob die zugehörige Transition für alle Objekte einer Menge gilt, oder nicht. Verglichen mit den in Abschnitt 3.4.1 dargestellten Automaten, entspricht eine Belegung des Attributs `foreach` mit `true` dem Allquantor  $\forall$ .

## Analyse-Algorithmus

Eine Iteration des von der Verhaltensanalyse verwendeten Erkennungsalgorithmus wird in Abbildung 4.3 als Sequenzdiagramm veranschaulicht.

Der Algorithmus bekommt die Methodenaufrufe des Traces in einer Queue als Eingabe. Für jeden Aufruf wird in der Klasse `BehavioralAnalysis` die Methode `methodCalled(MethodCall)` aufgerufen. In dieser Methode wird der Aufruf an alle Trigger (vgl. Abschnitt 2.2.2) delegiert, die die benötigten Automaten aktivieren und Tokens erzeugen. Daraufhin wird der Methodenaufruf an alle aktiven Automaten weitergereicht.

In der Klasse `DFA` wiederum wird der Methodenaufruf an alle Zustände (`State`) des Automaten delegiert. In der Klasse `State` wird zuerst über alle Tokens iteriert, die dieser Zustand enthält. Dabei wird durch die Abfrage des oben genannten Attributs `moved` überprüft, ob das Token bei der Bearbeitung des aktuellen Methodenaufrufs bereits verschoben wurde. Für alle noch nicht verschobenen Tokens, ruft der Zustand auf allen ausgehenden Transitionen die Methode `accept` auf, um zu überprüfen ob diese Tokens in einen neuen Zustand verschoben werden müssen. Die Methode `accept` der Klasse `Transition` bekommt neben dem Methodenaufruf außerdem das entsprechende Token übergeben. Akzeptiert eine Transition, wird das Token in den neuen Zustand verschoben. Eine Transition akzeptiert, wenn der

<sup>2</sup>Trace-Graph-Objekte repräsentieren die Objekte, die bei der Ausführung des Programms in den Trace aufgenommen wurden.

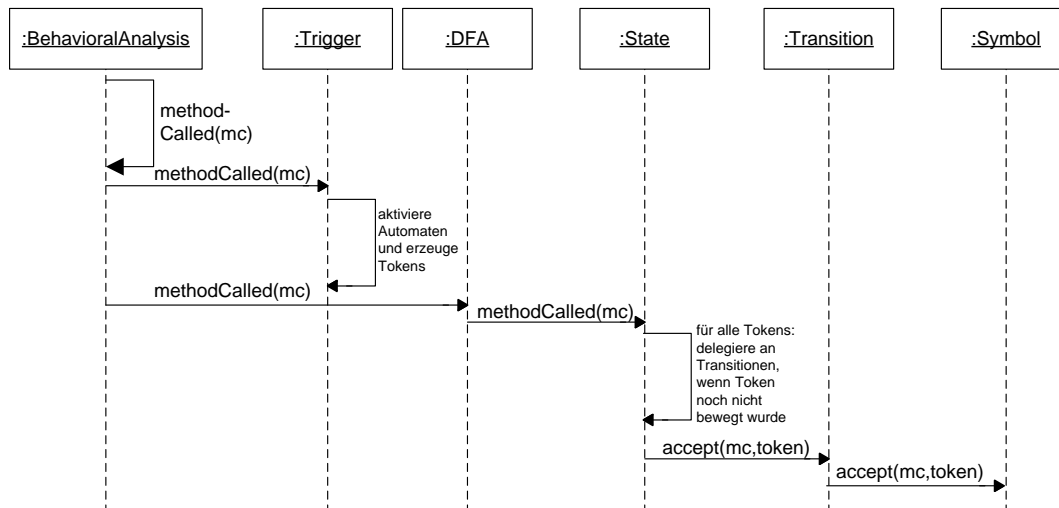


Abbildung 4.3: Veranschaulichung eines Schrittes des Analyse-Algorithmus

konsumierte Methodenaufruf des Traces an der Stelle konform zum Symbol der Transition ist. Der genaue Ablauf wird im Folgenden erklärt.

Die Methode `accept(MethodCall, Token)` der Klasse `Transition` enthält den Großteil der benötigten Erweiterungen für die Berücksichtigung von Mengenobjekten und Each-Fragmenten.

Die neue Methode wird in dem Aktivitätendiagramm auf Abbildung 4.4 dargestellt. Die Erweiterung betrifft nur Transitionen, deren Symbol akzeptiert und ein `PermittedMethodCallSymbol` ist. Trifft beides zu wird abgefragt, ob der `Caller` des Symbols ein Mengenobjekt ist. Ist dies der Fall und handelt es sich nicht um einen Aufruf innerhalb eines Each-Fragmentes, wird der Aufrufer des Methodenaufrufs aus dem Trace einer Menge mit dem entsprechenden Namen (im folgenden „Set-Menge“) hinzugefügt. Dies entspricht im Observer-Beispiel den `register`-Aufrufen. Dabei werden alle Observer-Objekte, die `register` aufrufen der entsprechenden Observer-Menge hinzugefügt. Handelte es sich bei dem aktuellen Methodenaufruf jedoch um einen mit „`foreach`“ gekennzeichneten Aufruf, wird der Aufrufer der speziellen Menge `setForEach` (vgl. Abschnitt 4.1.3) hinzugefügt. Dies entspricht den `update`-Aufrufen des Observer-Beispiels. Dabei muss sich der Algorithmus alle Objekte merken, auf die `update` aufgerufen wurde, um später vergleichen zu können, ob auch wirklich alle registrierten Observer aktualisiert wurden. Die Operationen für den `Callee` des Symbols gehen analog von statten. Der nächste Teil des Algorithmus sorgt dafür, dass im Falle eines `ForEach`-Aufrufs ein Token erst verschoben wird, wenn alle Aufrufe gemacht wurden. Dazu wird überprüft, ob jeweils der nächste Methodenaufruf „ähnlich“ ist. Das heißt, dass er denselben Methodennamen hat und der Aufrufer bzw. der Aufgerufene gleich ist bzw. dieselbe Vererbungshierarchie haben. Ist dies der Fall, werden die folgenden Befehle übersprungen. Beispielsweise bei der Analyse des Observer-Beispiels be-

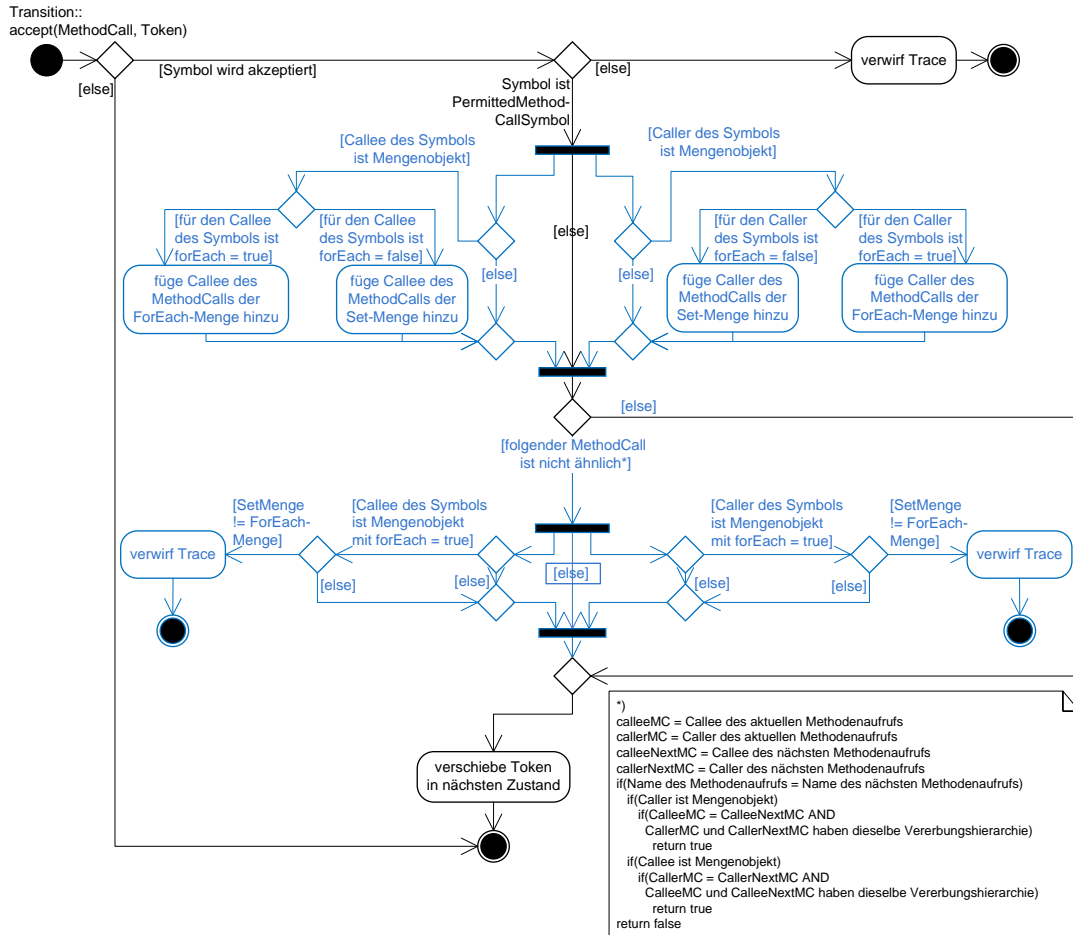


Abbildung 4.4: Veranschaulichung der Methode accept(MethodCall, Token) der Klasse Transition

deutet das, dass nach dem letzten update-Aufruf in der Methodenaufzugssequenz, der zugehörige Analyse-Automat seinen Zustand ändert, weil ein Analyse-Schritt abgearbeitet wurde. Im anderen Fall handelt es sich entweder um einen „normalen“ Methodenaufruf oder um den letzten Aufruf eines ForEach-Aufrufs. Im zweiten Fall ist entweder der **Caller** oder der **Callee** ein Mengenobjekt und `forEach = true`. Dann wird überprüft, ob in der **ForEach-Menge** dieselben Objekte sind, wie in der Menge des Mengenobjektes (**Set-Menge**). Wenn dies so ist, dann sind die entsprechenden Aufrufe im Trace konform zu der ForEach-Nachricht im Verhaltensmuster. Für das Observer-Beispiel würde dies bedeuten, dass alle Observer, die sich zuvor beim Subjekt registriert haben, auch später aktualisiert wurden, also das korrekte Verhalten des Observer-Musters zeigen. Das Token wird in den nächsten Zustand verschoben, ist dieser akzeptierend, wird dies im Token gespeichert. Stimmt die ForEach-Menge nicht mit der Set-Menge überein, muss der Trace verworfen werden. Dies ist im Observer-Beispiel entweder der Fall, wenn nicht auf allen Observern, die sich registriert haben, update aufgerufen wurde, oder wenn auf Objekten update aufgerufen wurde, die nicht in der Menge der registrierten Observer sind.

Außerdem musste ein Teil der Methode `accept(MethodCall, Token)` der Klasse `PermittedMethodCall` angepasst werden. Es geht um den Teil, der prüft, ob die Instanz, auf der der Methodenaufruf erfolgt, bereits gebunden wurde. Die Änderung besteht darin, dass Mengenobjekte als Aufrufer mehrere Instanzen enthalten können und deswegen in dem Fall der Name des Aufrufers unterschiedlich sein muss, um dem Objekt mehrere Bindungen ermöglichen zu können. Daher wird dem Namen der Instanz, wenn sie ein Mengenobjekt ist, eine laufende Nummer angehängt. Für ein aufgerufenes Mengenobjekt wurde dies analog umgesetzt. Bei einer Analyse mit einem Observer-Verhaltensmuster mit dem Mengenobjekt „oSet:Observer“, werden also alle Observer an „oSet“ gebunden.

Für die Erweiterung der Nachrichteneigenschaft `self call` wurde die Methode `accept(MethodCall, Token)` der Klasse `PermittedMethodCallSymbol` ergänzt. Die Methode enthält nun zusätzlich eine Abfrage, die prüft, ob der Name des Aufrufers oder Aufgerufenen des Symbols „other“ lautet. Ist dies der Fall, wird kontrolliert, ob der Aufrufer des Methodenaufrufs im Trace gleich dem Aufgerufenen ist. Ist diese Abfrage positiv, gibt die Methode „false“ zurück, das Symbol wird also nicht akzeptiert.

Um in der Analyse Zuweisungen berücksichtigen zu können, wurde die Methode `accept(MethodCall, Token)` der Klasse `Transition` ein weiteres Mal erweitert. Abbildung 4.5 zeigt das Aktivitätendiagramm der Methode mit der Ergänzung, welche in blau hervorgehoben wird. Sobald eine Transition erreicht wird, die eine Zuweisung enthält, wird die Liste der Argumente des aktuellen Methodenaufrufs nach einem Argument durchsucht, das vom richtigen Typ ist. Dadurch erhält man die ID, die die neue Instanz haben muss, die der Lifeline zugeordnet wird. Es wird immer eine passende ID gefunden, weil nur Objekte auf der rechten Seite der Zuweisung stehen können, die vorher als Argument in einem Methodenaufruf gebunden wurden. Daraufhin wird die bisherige Variablenbindung durch eine neue

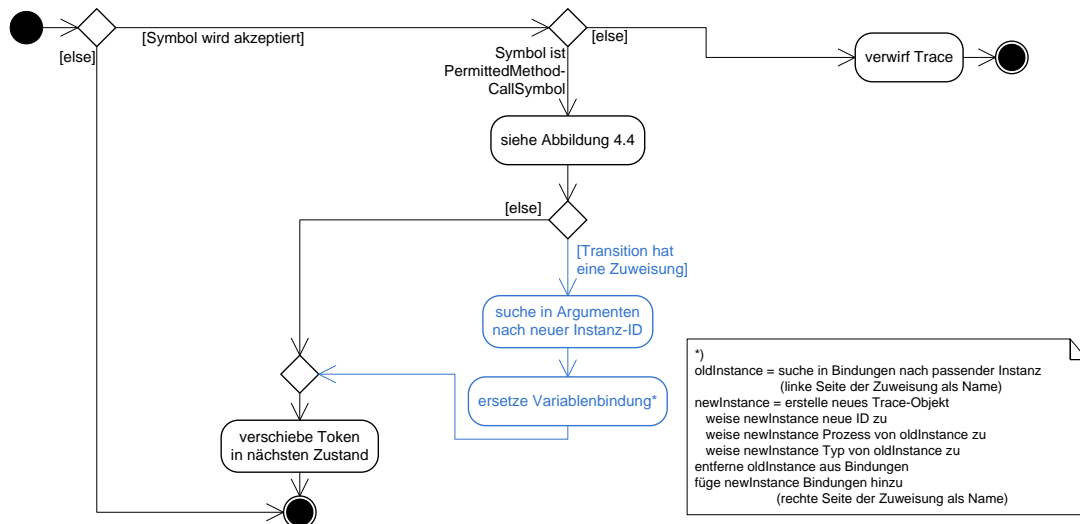


Abbildung 4.5: Erweiterung von `Transition::accept(MethodCall, Token)` für Zuweisungen

ersetzt, die zu der zugewiesenen Instanz passt. Dabei wird das alte Objekt aus der Liste der Bindungen des Tokens entfernt und dafür eine neue Instanz mit der neuen ID und demselben Typ hinzugefügt.

## 4.2 Erweiterung der Benutzerschnittstelle

In diesem Abschnitt wird beschrieben, wie sich die Umsetzung des Konzeptes dem Benutzer darstellt. Dabei wird auf die Änderungen in der Benutzerschnittstelle zur Verhaltensspezifikation eingegangen.

Der erweiterte Verhaltensmuster-Editor bietet nun zusätzlich die Möglichkeit zur Definition von Mengenobjekten, Each-Fragmenten, Selbstaufrufen innerhalb von Mengenobjekten, Argumenten und Zuweisungen.

### 4.2.1 Spezifikation von Mengenobjekten und Each-Fragmenten

Um ein Mengenobjekt zu erstellen, muss man zunächst ein normales Verhaltensmusterobjekt erstellen. Setzt man danach im Properties-Fenster die Eigenschaft `isSet` auf `true`, wird das entsprechende Objekt mit einem zweiten Rahmen hinterlegt, welcher die Mengeneigenschaft kennzeichnet. Abbildung 4.6 zeigt einen Screenshot des Verhaltensmuster-Editors mit dem Observer-Verhaltensmuster. Hier sieht man ein Beispiel für ein Mengenobjekt. Das Mengenobjekt ist das Objekt mit der Beschriftung `oSet:observerClass`. Im Properties-Fenster zu diesem Objekt kann man sehen, dass für die Eigenschaft `isSet` der Wert `true` eingestellt wurde. Für das Each-Fragment wurde die Palette um einen weiteren Eintrag erweitert („Each“). Analog zu der Erzeugung von Loop-, Alternative- und Optional-

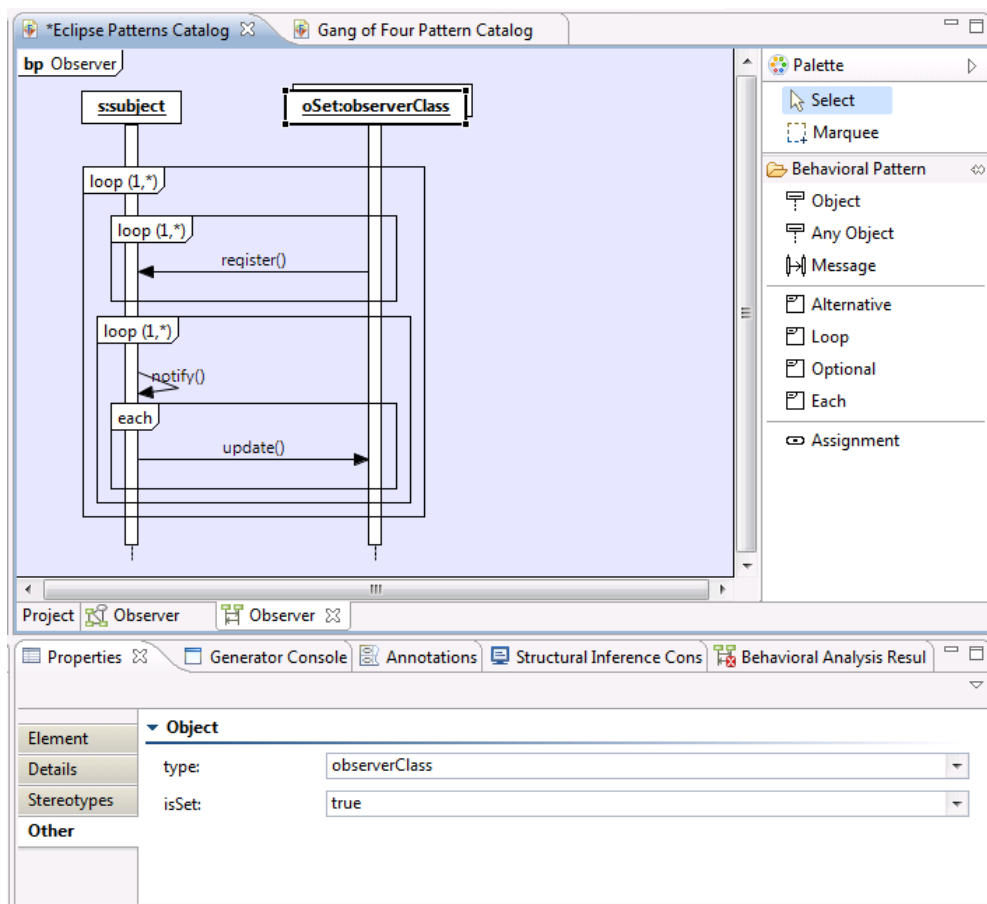
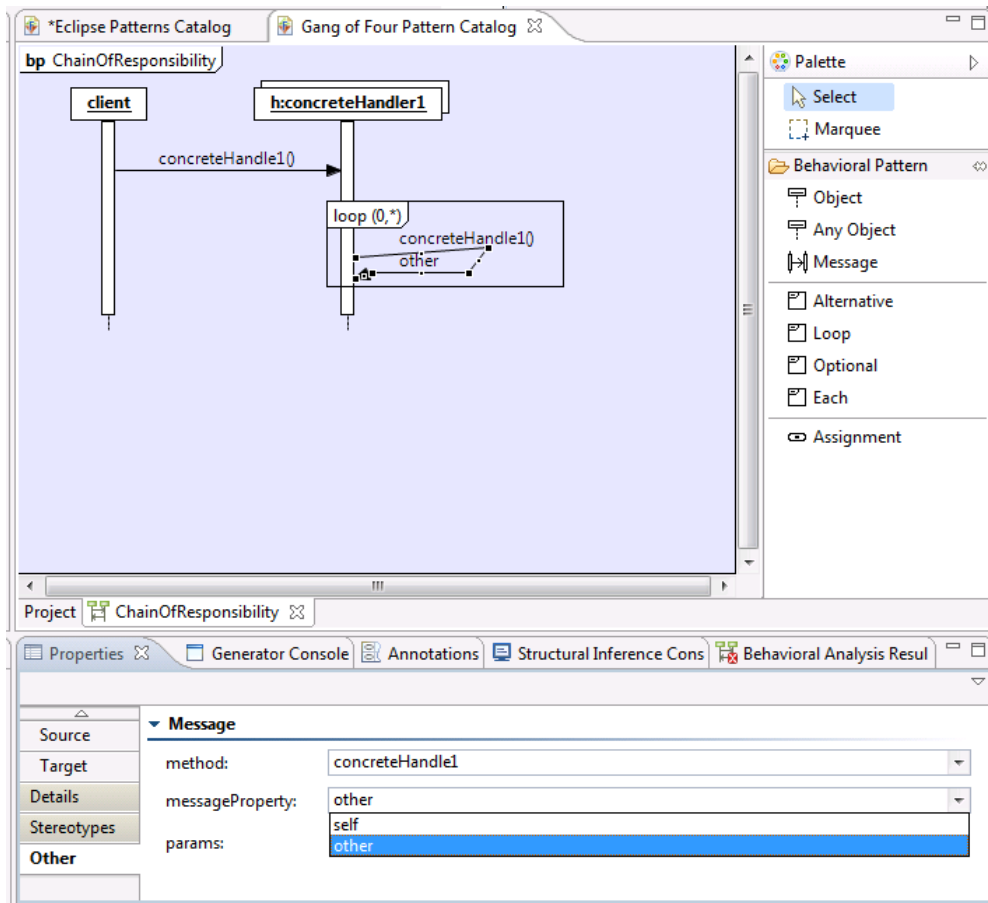


Abbildung 4.6: Screenshot des erweiterten Verhaltensmuster-Editors mit Mengenobjekt und Each-Fragment

Abbildung 4.7: Screenshot des Editors zu Nachrichteneigenschaft `self call`

Fragmente kann man nun mithilfe des entsprechenden Werkzeugs der Palette ein Each-Fragment erstellen. Die Abbildung zeigt neben drei Loop-Fragmenten ein Each-Fragment. Das Each-Fragment beinhaltet die Nachricht `update()`, die auf dem Mengenobjekt `oSet:observerClass` aufgerufen wird. Ein Each-Fragment darf nur mit einer Nachricht verwendet werden, bei der ein „normales“ Objekt eine Methode eines Mengenobjektes aufruft, oder bei der ein Mengenobjekt eine Methode eines normalen Objektes aufruft. Ohne Mengenobjekte sind Each-Fragmente nicht benutzbar.

### 4.2.2 Spezifikation von Selbstaufen

Um Selbstaufe zu modellieren, muss man die Eigenschaften der entsprechenden Nachricht im Properties-Fenster konfigurieren. Abbildung 4.7 zeigt den Editor mit dem Chain-Of-Responsibility-Verhaltensmuster und das Properties-Fenster bei der Auswahl der Eigenschaft `self call = true`. Dadurch wird die Nachricht mit einer Beschriftung `other` gekennzeichnet.

### 4.2.3 Spezifikation von Argumenten und Zuweisungen

Wie man in dem State-Verhaltensmuster-Screenshot in Abbildung 4.8 sehen kann, werden an den Nachrichten nun auch Argumente dargestellt. Der erste Methodenaufruf `setState` bekommt als Argument ein Objekt `a` vom Typ `AbstractState`. Beim Anlegen einer Zuweisung, kann man diese Objekte referenzieren. Die Abbildung zeigt eine Zuweisung `a=s` an der Lifeline des Objektes `a:abstractState`. Eine Zuweisung lässt sich mithilfe des entsprechenden Werkzeugs auf der Palette („Assignment“) erstellen. Die linke Seite der Zuweisung wird durch das Objekt bestimmt, zu dem die Lifeline gehört, auf der man die Zuweisung erstellt. Die rechte Seite kann man anhand eines Auswahlfeldes, das alle zuvor gebundenen Variablen des passenden Typs enthält (abgesehen von dem Objekt, dem die Lifeline gehört), bestimmen. Wählt man für eine Zuweisung auf der Lifeline des

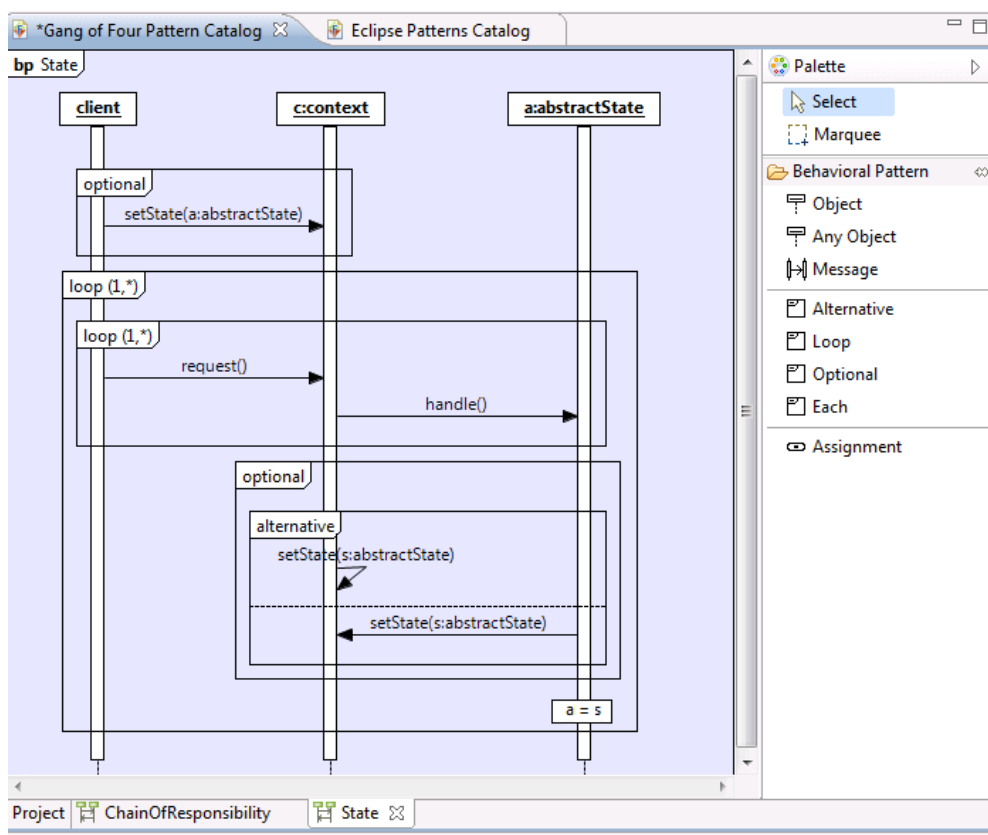


Abbildung 4.8: Screenshot des Verhaltensmuster-Editors mit Argumenten und einer Zuweisung

Objektes `a:abstractState` zum Beispiel das in dem vorherigen Methodenaufruf übergebene Objekt `s:abstractState` aus, lautet die Zuweisung `a=s` und wird damit entsprechend beschriftet.

# 5 Praktische Anwendung

In diesem Kapitel wird eine praktische Anwendung beschrieben, die mit RECLIPSE durchgeführt wurde. Dabei wird zuerst auf Ergebnisse eingegangen, die eine frühere Durchführung produzierte. Danach wird die praktische Anwendung beschrieben, die mit den Erweiterungen durchgeführt wurde, die im Rahmen dieser Arbeit umgesetzt wurden.

## 5.1 Bisherige Ergebnisse

Die Entwurfsmustererkennung von RECLIPSE wurde in [Wen07] unter anderem am Beispiel des Quellcodes von Eclipse (Version 2.1, ca. 66 Plug-Ins) durchgeführt. Eclipse stellt ein geeignetes Untersuchungsbeispiel dar, da die Software zu groß ist, um manuell analysiert zu werden. Außerdem wurden bei der Entwicklung mehrere Entwurfsmustern verwendet und dokumentiert [GB03]. Dies ist hilfreich, um die Qualität der automatischen Entwurfsmustererkennung zu beurteilen.

Die Untersuchung konzentrierte sich auf sechs ausgewählte Entwurfsmusterimplementierungen, darunter drei Observer- und drei Strategy-Muster. Diese Entwurfsmusterimplementierungen wurden in der Strukturanalyse korrekt als Kandidaten der jeweiligen Entwurfsmuster identifiziert. Auf Grund der identischen Strukturmuster des Strategy- und des State-Entwurfsmusters wurden jedoch die drei Strategy-Implementierungen mit derselben Bewertung auch als State-Kandidaten erkannt. Die State-Kandidaten sind False-Positives.

Die Verhaltensanalyse bestätigte zwei der drei Strategy-Implementierungen und verwarf korrekterweise die zugehörigen State-Kandidaten. Das Ergebnis der anderen Strategy-Implementierung ist nicht eindeutig. Spätere Analysen haben ergeben, dass der Grund dafür sein könnte, dass das Verhalten dieser Implementierung zur Laufzeit eher dem Chain-of-Responsibility-Muster ähnelt [Wen07].

Die Observer-Kandidaten konnten nicht durch die Verhaltensanalyse bestätigt werden. Die Gründe dafür wurden bereits in Kapitel 3.1 geschildert.

## 5.2 Praktische Anwendung des Observer-Entwurfsmusters

Im Folgenden wird die Durchführung einer praktische Anwendung beschrieben, die im Rahmen dieser Arbeit umgesetzt wurde. Das untersuchte Beispielprogramm enthält eine Implementierung des Observer-Entwurfsmusters und wurde mit der

struktur- und verhaltensbasierten Entwurfsmustererkennung von RECLIPSE, einschließlich der in dieser Arbeit vorgenommenen Erweiterungen, untersucht. Anschließend werden die Ergebnisse präsentiert.

### 5.2.1 Beispielprogramm

In einer neuen praktischen Anwendung wurde das schon im Verlauf der Arbeit vorgestellte Observer-Beispiel untersucht.

Der Kandidat wurde bereits in Kapitel 2.2.1 beschrieben. Abbildung 5.1 zeigt das Ergebnis der Strukturanalyse in RECLIPSE. Die Variablenbindungen der Struk-

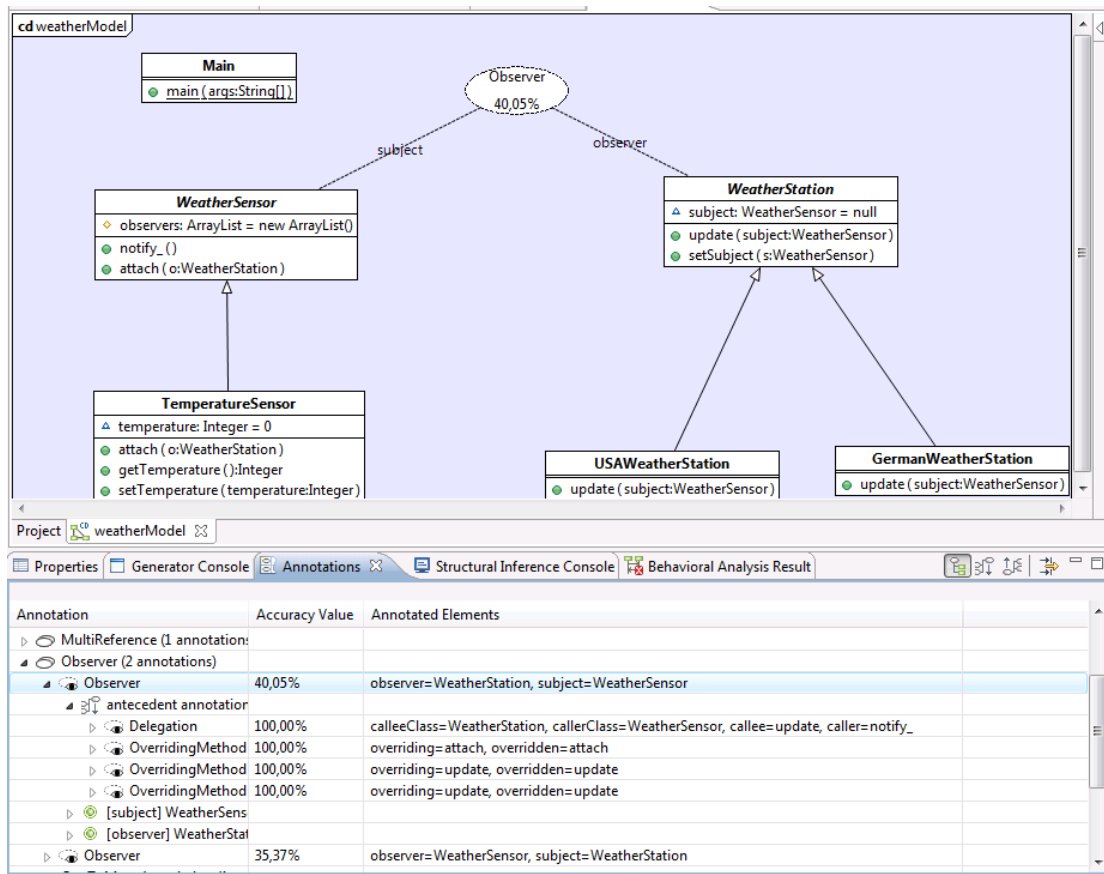


Abbildung 5.1: Ergebnisse der statischen Analyse

turanalyse für den Kandidaten entsprechen denen, die bereits in Kapitel 2.2.2, Tabelle 2.7 gezeigt wurden. Diese sind zum Teil in Abbildung 5.1 sichtbar.

Der Großteil des für die Verhaltensanalyse anhand von RECLIPSE erzeugten Traces gleicht dem, der in Kapitel 2.2.2, Abbildung 3.19 veranschaulicht wurde. Zuerst ruft eine Instanz von USAWeatherStation die Methode attach auf dem Subjekt vom Typ TemperatureSensor auf. Ebenso verhält sich eine Instanz von GermanWeatherStation. Nachdem der TemperatureSensor notify auf sich selbst

aufruft, führt er jeweils die update-Methoden der beiden Wetter-Stationen aus. Der für die praktische Anwendung verwendete Trace führt den notify-Aufruf und die darauf folgenden update-Aufrufe drei mal aus.

Das verwendete Verhaltensmuster wurde in Abschnitt 3.3.1 präsentiert. Die Instanz des Subjects trägt den Namen `s`, die Menge von Observern wurde `oSet` genannt.

## 5.2.2 Analyseergebnisse

Die Verhaltensanalyse für das Beispielprogramm mit dem neuen Verhaltensmuster und dem erweiterten Analysevorgang in RECLIPSE war erfolgreich. Abbildung 5.2 zeigt die zugehörige Ergebnis-Ansicht von RECLIPSE. Dort sieht man das Ergebnis

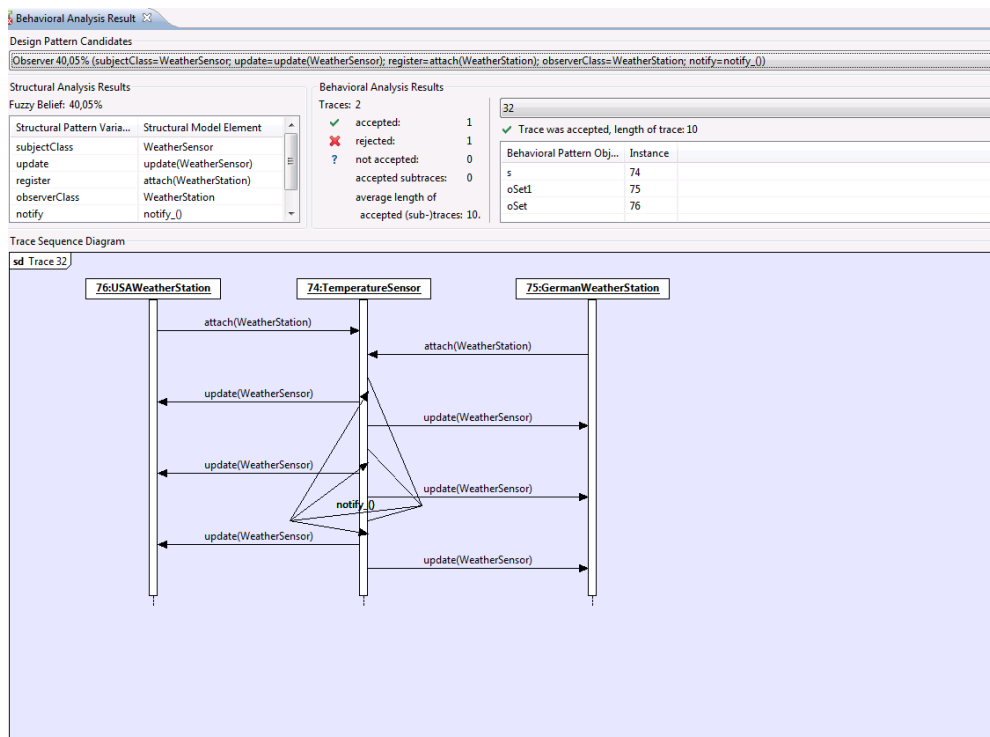


Abbildung 5.2: Ergebnisse der Untersuchung mit zwei Observern und dem neuen Verhaltensmuster

für den Kandidaten mit den oben genannten Variablenbindungen. Es wurden zwei Traces untersucht, von denen einer akzeptiert und einer verworfen wurde. Das im unteren Teil des Fensters dargestellte Sequenzdiagramm zeigt den akzeptierten Trace, welcher eine Länge von zehn Methodenaufrufen hat. In der Liste, die sich rechts oberhalb des Traces befindet, sieht man die Variablenbindungen der Verhaltensanalyse. Das Objekt `s` des Verhaltensmusters wurde an Instanz 74 gebunden. Im Sequenzdiagramm sieht man, dass dies der `TemperatureSensor` ist. Die Instanzen 75 und 76 wurden dem Mengenobjekt `oSet` zugewiesen, sie entsprechen

der `GermanWeatherStation` und der `USAWeatherStation`. Der verworfene Trace entsteht durch den zweiten Trigger. Trigger des Observer-Verhaltensmusters sind Aufrufe, die den `register`-Aufruf repräsentieren. In diesem Fall ist also einmal der Aufruf `attach(WeatherStation)` von `USAWeatherStation` ein Trigger, aber ebenfalls der im eigentlichen Trace als zweites folgende Aufruf `attach(WeatherStation)` von `GermanWeatherStation`. Im zweiten Fall wird also nur ein Observer registriert, jedoch folgen immer `update`-Aufrufe auf zwei Objekte, womit der Trace nicht konform zum Verhaltensmuster ist. Auf diese Problematik wird in Kapitel 7.2 genauer eingegangen.

### 5.2.3 Erweitertes Beispielprogramm

Als nächstes wurde das oben beschriebene Beispiel so erweitert, dass sich zur Laufzeit ein weiterer Observer, also insgesamt drei Observer registrieren. Der Anfang des verwendeten Traces wird in Abbildung 5.3 dargestellt. Der dritte Observer heißt `FrenchWeatherStation`.

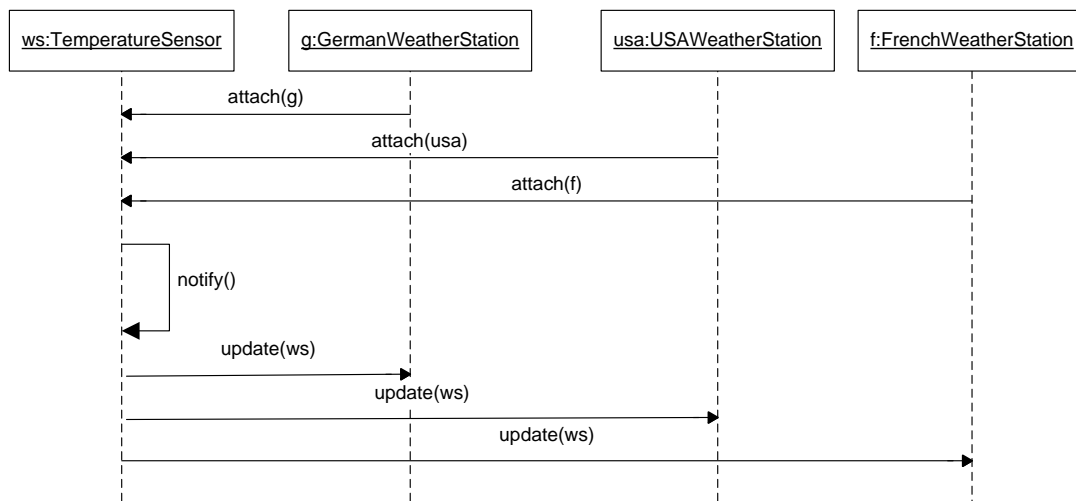


Abbildung 5.3: Trace für das Observer-Beispiel mit drei Observern

Bei der Analyse anhand des vorherigen Observer-Verhaltensmusters aus [Wen07] wird der Trace verworfen, da eine Objektmenge mit mehr als zwei Instanzen zu einer Abfolge von Methodenaufrufen führt, die nicht konform zu dem Verhaltensmuster ist. Dies ist ein Fehler, da es sich um eine korrekte Observer-Implementierung handelt. Die alte Analyse erzeugt hier ein *False-Negative*.

Mit dem neuen Verhaltensmuster und der erweiterten Verhaltensanalyse wird auch das Beispiel mit den drei Observern korrekt erkannt. Der akzeptierte Trace hat eine Länge von 14 Methodenaufrufen und enthält drei Instanzen, die an das Mengenobjekt `oSet` gebunden wurden.

Außerdem wurde die Untersuchung auch mit einem Beispiel mit vier Observern und einem mit nur einem Observer gemacht. Auch für diese beiden Fälle war die Verhaltensanalyse erfolgreich.

## 5.3 Praktische Anwendung des State-Entwurfsmusters

Neben der oben beschriebenen Untersuchung wurde eine weitere praktische Anwendung hinsichtlich einer Implementierung des State-Entwurfsmusters gemacht. Auch dieses Programm wurde mit Hilfe der Entwurfsmustererkennung von RECLIPSE, einschließlich der vorgenommenen Erweiterungen in der Verhaltensanalyse, betrachtet. Im weiteren Verlauf des Kapitels werden die Ergebnisse präsentiert.

### 5.3.1 Beispielprogramm

Das Beispielprogramm stellt die Tätigkeit einer Wetterstation dar. Abbildung 5.4

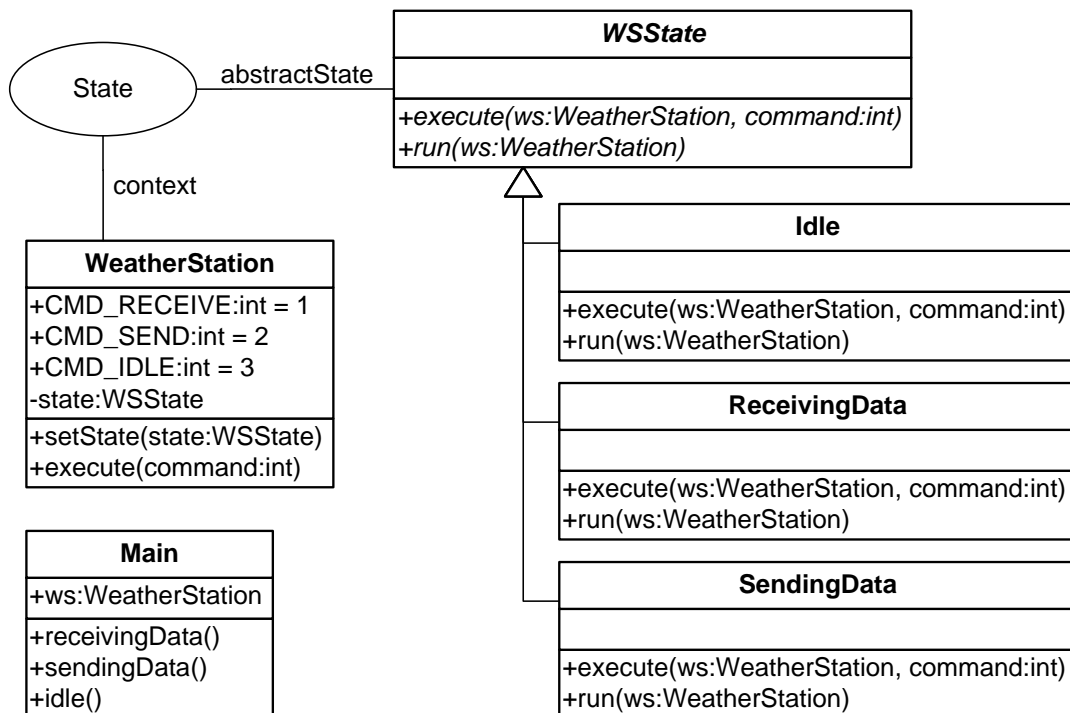


Abbildung 5.4: State-Kandidat der Beispielimplementierung

zeigt den Kandidaten. Es gibt eine abstrakte Klasse `WSSState` und drei konkrete Klassen, die von dieser erben: `Idle`, `ReceivingData` und `SendingData`. Diese Klassen verfügen über die Methoden `execute(WeatherStation, int)` und `run(WeatherStation)`, des Weiteren gibt es eine Klasse `WeatherStation` mit den

Methoden `setState(WSSState)` und `execute(int)`. `WeatherStation` entspricht im State-Muster dem `Context`, während `WSSState` die Klasse `AbstractState` repräsentiert. Wie man auch in Tabelle 5.3.1 sieht, sind die Methoden `request`, `handle` und `setState` an `WeatherStation::execute`, `WSSState::execute` und `setState` gebunden.

Strukturmuster	Kandidat
Context	WeatherStation
AbstractState	WSSState
request	WeatherStation::execute
handle	WSSState::execute
setState	setState

Tabelle 5.1: Variablenbindungen für den State-Kandidaten

Das verwendete State-Verhaltensmuster wurde bereits in Abschnitt 3.3.3 dargestellt und beschrieben.

Der in der Verhaltensanalyse untersuchte Trace wird in Abbildung 5.5 veranschaulicht. Hier wird zuerst der Zustand durch den Client (hier repräsentiert durch

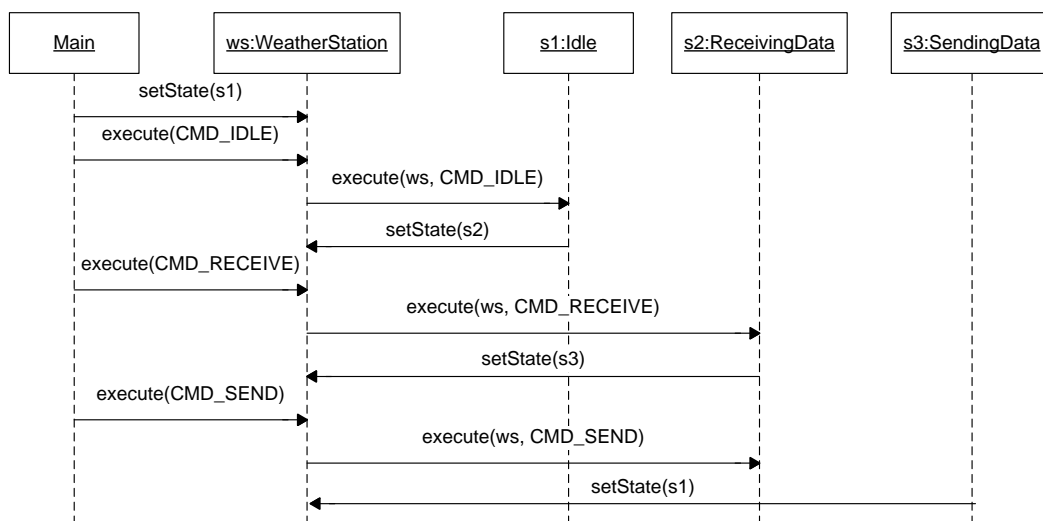


Abbildung 5.5: Untersuchter Trace für das State-Beispiel

die Klasse `Main`) gesetzt. Dann wird auf dem Objekt `ws:WeatherStation` die Methode `execute(int)` aufgerufen, welche diesen Aufruf an den aktuellen Zustand (`s1:Idle`) delegiert. Danach erfolgt ein Zustandswechsel von `s1` zu `s2:ReceivingData`, wiederum durch eine Delegation der `execute`-Methode durch die `WeatherStation`. Der neue Zustand leitet einen Zustandswechsel zu `s3:SendingData` ein. Am Ende setzt `s3` wieder den Zustand `Idle`.

### 5.3.2 Analyseergebnisse

Abbildung 5.6 zeigt das Ergebnisfenster von RECLIPSE nach der Durchführung der Verhaltensanalyse mit dem oben beschriebenen Beispieltrace und dem neuen State-Verhaltensmuster als Eingabe. Wie man sieht, wurde der Trace inklusive

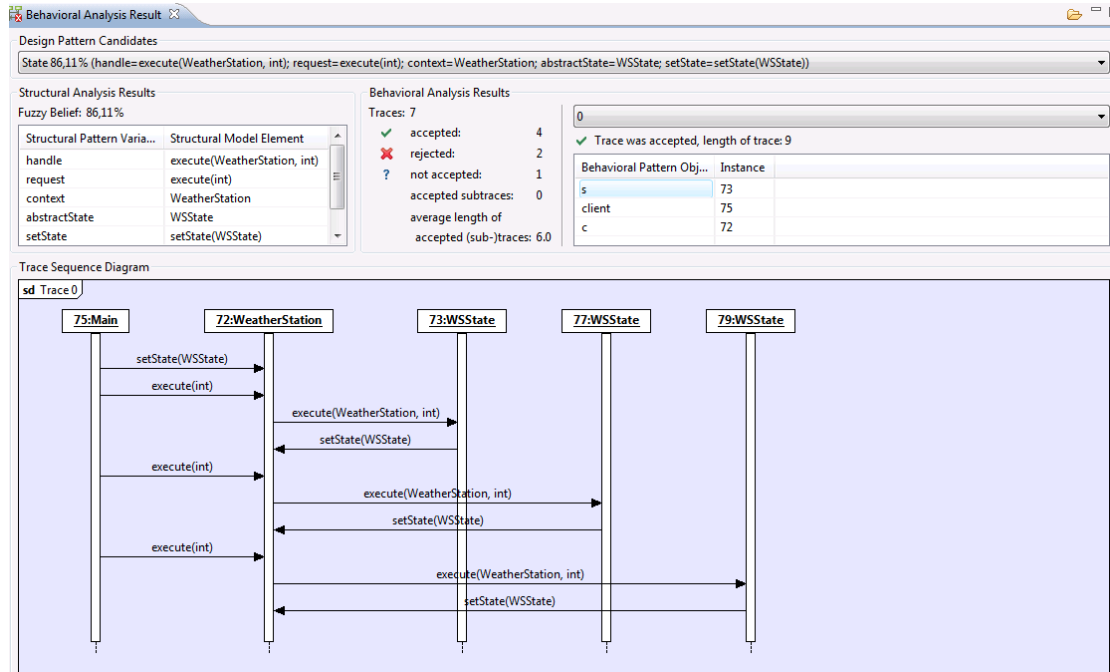


Abbildung 5.6: Analyseergebnisse für das State-Beispiel

der drei Zustandswechsel akzeptiert, obwohl im Verhaltensmuster nur ein Zustand, dem per Zuweisung verschiedene konkrete Zustände zugeordnet werden können, spezifiziert wurde. Durch die Zuweisung wurden dem Zustand `abstractState` nacheinander die Instanzen `73:WSSState` (entspricht Zustand `Idle`), `77:WSSState` (entspricht `ReceivingData`) und `79:WSSState` (entspricht `SendingData`) zugewiesen.

Außerdem wurde ein weiterer Trace akzeptiert, der einen Subtrace des oben dargestellten bildet, da er erst bei dem zweiten Methodenaufruf anfängt, welcher ebenfalls ein Trigger ist. Dies ist möglich, da die erste Nachricht im Verhaltensmuster optional ist. Zwei weitere Teil-Traces, die durch die Trigger des in der obigen Darstellung fünften und achten Methodenaufrufs, entstehen, werden ebenfalls akzeptiert. Die restlichen Teil-Traces werden verworfen oder nicht akzeptiert.

Dieselbe Analyse wurde auch mit einem Beispiel mit vier States durchgeführt. Auch für dieses Beispiel wurde der Trace akzeptiert.



## 6 Verwandte Arbeiten

In diesem Kapitel wird ein Überblick über andere Arbeiten gegeben, die thematisch dieser Arbeit ähneln. Im ersten Teil wird auf die Berücksichtigung von Objektmengen eingegangen, der zweite Teil beschäftigt sich mit verschiedenen Verfahren zur Entwurfsmustererkennung.

### 6.1 Verwendung von Mengenobjekten

Mengenobjekte findet man nicht nur in den Verhaltensmustern von RECLIPSE, sondern auch in den Strukturmustern. Jörg Niere beschreibt in seiner Arbeit die Darstellung eines Mengenobjektes als Knoten, der durch einen „dreidimensionalen“ Rahmen gekennzeichnet wird [Nie04]. Diese Darstellung gleicht also der Darstellung der in dieser Arbeit eingeführten Mengenobjekte in Verhaltensmustern. Auch in Strukturmustern bedeutet ein Mengenobjekt, dass ihm eine beliebig große Anzahl Instanzen zugeordnet sein können.

Die Strukturmuster gehen auf die von Albert Zündorf beschriebenen Story-Diagramme zurück [Zün01]. Eine Menge von Objekten wird hier von einem so genannten Multi-Objekt repräsentiert. Auch in Zündorfs Arbeit wird es zur Beschreibung von 1:n-Assoziationen verwendet und mit einem doppelten Rahmen gekennzeichnet.

### 6.2 Kombinierte statische und dynamische Entwurfsmusteranalysen

Auf dem Gebiet der Entwurfsmustererkennung wird seit einiger Zeit geforscht. Es gibt jedoch nur einige wenige Ansätze, die sich mit dem Verhalten von Entwurfsmustern zur Laufzeit beschäftigen. Die dynamische Analyse wird meist benutzt, um die Ergebnisse der statischen Analyse zu präzisieren.

Heuzeroth et al. kombinierten bereits 2003 eine statische Entwurfsmustererkennung mit einer darauf folgenden dynamischen Analyse des Verhaltens zur Laufzeit [HHHL03], [HML03]. Bei der dynamischen Analyse wird das zu untersuchende Programm ausgeführt und dabei die Ausführung der in der statischen Analyse gefundenen Kandidaten beobachtet. Der Suchraum wird damit durch die statische Analyse eingeschränkt. Dies gleicht dem Konzept, das RECLIPSE zugrunde liegt. Die dynamische Analyse verwendet Algorithmen, die überprüfen, ob das

Protokoll der Methodenaufrufe der ausgeführten Kandidaten konform zu dem erwarteten Verhalten des Entwurfsmusters ist. Die Autoren implementierten vorerst für verschiedene Entwurfsmuster je eine eigene Analyse, dabei wurden die Muster Observer, Mediator, Visitor, Composite und Chain of Responsibility realisiert. Daraufhin wurde ein Generator entwickelt, der die Analysealgorithmen aus den vorher spezifizierten Mustern automatisch erstellt. Als Spezifikationsprache wird SanD bzw. SanD-Prolog verwendet. Es handelt sich also um eine textuelle Beschreibung der Entwurfsmuster, im Gegensatz zu der bei RECLIPSE verwendete grafischen Modellierung. Die Autoren bezeichnen besonders SanD-Prolog als kompliziert und unübersichtlich. Dies macht die Handhabung für den Benutzer schwierig, da Musterspezifikationen nur schwer erweitert oder angepasst werden können. Der Ansatz von Heuzeroth et al. wurde ausgiebig evaluiert und konnte zumindest für das Observer-Muster gute Ergebnisse erzielen.

Hayashi et al. veröffentlichten 2008 eine Methode die ebenfalls sowohl eine statische als auch eine dynamische Entwurfsmustererkennung integriert [HKS<sup>+</sup>08]. Die Autoren konzentrieren sich auf die Gamma-Muster. Dabei wird mit Meta-Mustern nach Pree([PS97]) gearbeitet, um die gemeinsamen Eigenschaften von Entwurfsmustern hierarchisch zu strukturieren und damit den Aufwand des Erkennungsprozesses zu reduzieren. Zum Beispiel das Observer-Muster enthält ein "1:NConnection"-Meta-Muster. Dieser Teil ist vergleichbar mit der Strukturanalyse von RECLIPSE, bei der die Muster ebenfalls inkrementell spezifiziert werden. Hayashi et al. erwähnen ebenfalls dynamische Methodenbindung als Begründung für die Erfordernis einer dynamischen Analyse. Bei ihrer dynamischen Analyse werden Methodenaufrufe, inklusive Konstruktor-Aufrufe und die zugehörigen Objektinstanzen betrachtet. Die in der statischen und dynamischen Analyse gewonnenen Daten werden in einer Prolog-Datenbank gespeichert. Die Erkennungsregeln für die Entwurfsmuster wurden in einem Prolog-Programm verfasst, das mit den Daten der Analysen arbeitet. Die Musterspezifikationen sind also hart kodiert und damit eher schwer austauschbar. Mithilfe der Prolog-Regeln erhält man als Ausgabe Kandidaten für Entwurfsmusterimplementierungen. Mit der Entwurfsmustererkennung durch Prolog entstehen keine Probleme mit Objektmengen, wie sie in den Verhaltensmustern von RECLIPSE auftauchen. In einer experimentalen Anwendung wurden neben einigen False-Positives alle Entwurfsmusterinstanzen der untersuchten Software gefunden.

Einen weiteren Ansatz zur Entwurfsmustererkennung, der eine dynamische Analyse verwendet, beschrieben Sartipi et al. 2008 [SH08]. Die Autoren arbeiten an einer verhaltensgesteuerten Entwurfsmustererkennung mit zwei Teilschritten. Anders, als bei vielen anderen Arbeiten, wird hier zuerst die dynamische Analyse und danach eine statische Analyse durchgeführt. Mithilfe der dynamischen Analyse soll die Analyse auf die Teile des Quelltextes begrenzt werden, die spezielle Features implementieren. Dabei kann der Benutzer feature-spezifische Szenarios auswählen. Diese Szenarios werden ausgeführt und für jedes ein Trace erzeugt. So erhält man die Teile der Software, also Klassen und Methoden, die zu dem ausgewählten Feature gehören, um speziell diese analysieren zu können. Danach

werden Ausführungsmuster (maximal lange Sequenz an Methodenaufrufen, die in mehreren Traces vorkommt) extrahiert und analysiert, so dass man ein Mapping zwischen Softwarefeature und Implementierung erhält und anschließend den Suchraum für die folgende statische Analyse generieren kann. Die statische Analyse identifiziert daraufhin Entwurfsmusterinstanzen in dem von der dynamischen Analyse geschaffenen Suchraum anhand einer Strukturanalyse. Die Instanzen von Entwurfsmustern werden dabei anhand einer „Pattern Description Language (PDL)“ beschrieben. Dabei handelt es sich um eine textuelle Beschreibung der Struktur eines Entwurfsmusters. In diesem Ansatz wird also nicht wirklich eine dynamische Entwurfsmusteranalyse verwendet. Den ersten Schritt dient eher einer Vorbereitung für eine darauf folgende rein statische Entwurfsmustererkennung.

Auch ein 2005 von Pettersson beschriebener Ansatz besteht aus einer Kombination aus statischer und anschließender dynamischer Analyse [Pet05]. Die statische Analyse arbeitet mit abstrakten Syntaxgraphen. Zur Analyse werden prädikatenlogische Formeln als Constraints benutzt. Die dynamische Analyse besteht aus zwei Phasen. Zuerst werden Informationen über das Laufzeitverhalten der zu untersuchenden Applikation gesammelt. Gespeichert werden dabei Methodensignaturen, Integer-Kodierungen der Argumente und Integer-Kodierungen des This-Pointers. Danach werden diese Daten ausgelesen und bearbeitet. Für das Observer-Muster wird dabei beispielsweise untersucht, ob jeder Observer, der vom Subjekt wieder entfernt wird, auch vorher dem Subjekt zugefügt worden ist. Wenn dies nicht so ist, gilt das Methodenprotokoll als verworfen. Im Gegensatz zu der Analyse mit dem in dieser Arbeit vorgestellten Observer-Verhaltensmuster, wird das typische Verhalten mit Hinblick auf die Notify- und die Update-Methode, nicht untersucht. Als Grund nennt der Autor hier die zusätzlich benötigte Menge an Speicher. Er nimmt diesen Punkt jedoch in seiner Liste für zukünftige Erweiterungen seines Ansatzes auf. Aufgrund der eingeschränkten Verhaltensanalyse scheint auch in diesem Ansatz das Vorkommen von Objektmengen keine Problematik darzustellen. Die praktische Anwendung ergab, dass eine anschließende dynamische Analyse genauere Ergebnisse lieferte als die statische Analyse alleine, jedoch trotzdem einige False-Positives aufgetreten sind.



# 7 Zusammenfassung und Ausblick

Im letzten Kapitel wird ein Überblick über die Inhalte der Arbeit gegeben und die Ergebnisse werden zusammengefasst. Der zweite Teil des Kapitels gibt einen Ausblick auf mögliche Erweiterungen und diskutiert Probleme des Ansatzes.

## 7.1 Zusammenfassung

In der vorliegenden Arbeit wurde das Konzept der dynamische Entwurfsmustererkennung erweitert und innerhalb des Prototyps RECLIPSE umgesetzt. Das Problem bei der bisherigen dynamischen Analyse in Reclipse war, dass mit der verwendeten Spezifikationsprache keine variablen Anzahlen von Objektmengen, wie sie z.B. bei den Entwurfsmustern Observer, Chain of Responsibility und State auftauchen, dargestellt und untersucht werden konnten. Ziel der Erweiterung war die Verbesserung der Entwurfsmustererkennung durch das Verringern von False-Negatives in den Ergebnissen der dynamischen Analyse. Dazu wurde die Verhaltensspezifikation so angepasst, dass ein angemessener Umgang mit Objektmengen möglich ist. Außerdem musste die Verhaltensanalyse der Entwurfsmustererkennung an verschiedenen Stellen entsprechend modifiziert werden.

Zu diesem Zweck wurde die Spezifikationsprache der Verhaltensanalyse um Mengenobjekte und Each-Fragmente erweitert. Mengenobjekten kann bei der Verhaltensanalyse eine beliebig große Anzahl konkreter Instanzen von Verhaltensmusterobjekten zugeordnet werden. Mit einem Each-Fragment kann man Methodenaufrufe, die auf allen Instanzen innerhalb eines Mengenobjektes erfolgen sollen, modellieren. Dieser Fall tritt beispielsweise beim Observer-Entwurfsmuster auf.

Bei Selbstaufrufen auf Mengenobjekten lässt sich nun zwischen Aufrufen von einem Objekt der Menge auf einem anderen Objekt derselben Menge und tatsächlichen Selbstaufrufen eines Objekts der Menge unterscheiden. Um beide Fälle modellieren zu können, wurde die Nachrichteneigenschaft „self call“ eingeführt. Ein Entwurfsmuster, bei dem diese Eigenschaft benötigt wird, ist das Chain-Of-Responsibility-Muster.

Außerdem wurde die Darstellung der Nachrichten im Verhaltensmuster um Argumente erweitert. Um die Identität eines Verhaltensmusterobjektes zu ändern, wurden Zuweisungen eingeführt. Diese referenzieren Variablen, die in einem vorherigen Methodenaufruf als Argument auftraten und gebunden wurden. Benötigt werden diese Notationselemente beispielsweise beim State-Entwurfsmuster.

Der in Reclipse verwendete Editor zur Spezifikation von Verhaltensmustern wurde entsprechend erweitert.

Aufgrund der neuen Sprachelemente mussten auch die zur Analyse verwendeten Automaten angepasst werden. Mit Hilfe der erweiterten Automaten lassen sich die oben genannten neuen Konzepte in der Verhaltensanalyse umsetzen. Der Analysevorgang in Reclipse wurde entsprechend angepasst.

Die Praktische Anwendung ergab, dass mithilfe der erweiterten Verhaltensanalyse nun auch Instanzen des Observer-Entwurfsmusters mit unterschiedlichen Anzahlen von Observern zur Laufzeit erkannt werden. Dies wurde für ein bis vier Observer getestet. Außerdem waren auch Analysen für Implementierungen des State-Entwurfsmusters mit einer variablen Anzahl Zustände erfolgreich. Dies wurde für drei und vier Zustände validiert.

Als Fazit lässt sich sagen, dass das Ziel der Berücksichtigung von Objektmengen bei der dynamischen Entwurfsmustererkennung erreicht wurde und sich die Erweiterung der verhaltensbasierten Entwurfsmustererkennung als hilfreich herausstellte.

Jedoch gibt es weiterhin Möglichkeiten zur Verbesserung, welche im folgenden Abschnitt diskutiert werden.

### 7.2 Ausblick

Trotz der umgesetzten Erweiterungen der Spezifikationsprache gibt es immer noch Situationen, die nicht durch Verhaltensmuster modelliert werden können. Beispielsweise erlaubt die Spezifikationsprache es derzeit nicht, mehrere Mengen des gleichen Typs zu verwenden. Außerdem bleibt die Semantik für eine Nachricht von einem Mengenobjekt zu einem Mengenobjekt in Kombination mit einem Each-Fragment undefiniert. Im Editor von Reclipse wird die Spezifikation solcher Nachrichten deswegen verboten. Des weiteren besteht nicht die Möglichkeit, einzelne Objekte in den Mengen gezielt zu referenzieren. Diese Punkte könnten in einer zukünftigen Erweiterung von Reclipse bearbeitet werden, sind aber für die bisher von uns untersuchten Entwurfsmuster nicht notwendig.

Eine weitere mögliche Erweiterung des Ansatzes könnte in einer Ausweitung der Mengenoperationen bestehen. Bisher ist es nur möglich Objekte einer Menge hinzuzufügen, jedoch lassen sich Objekte aus einer Menge bisher nicht wieder entfernen. Da jedoch beispielsweise bei einer Instanz des Observer-Musters, das Abmelden eines Observers beim Subjekt, ein korrektes Verhalten darstellt, sollte man auch dies berücksichtigen. Dazu müsste das Strukturmuster des Observer-Musters um eine entsprechende Methode (z.B. „deregister“) erweitert werden und das entsprechende Verhalten im Verhaltensmuster neu spezifiziert werden. Eine solche Erweiterung würde ein noch genaueres Erkennen des Verhaltens einer Observer-Entwurfsmusterinstanz ermöglichen. Dies erfordert eine Anpassung der für die dynamische Analyse verwendeten Automaten.

Des weiteren ist eine umfassende Evaluation der struktur- und verhaltensbasierten Entwurfsmustererkennung von Reclipse notwendig. Dazu sollte ein praxisnahes Softwaresystem betrachtet werden, das von einer realistischen Größe und Komple-

xität ist. Dabei können auch Anforderungen an eine Entwurfsmustererkennung, wie Skalierbarkeit, Präzision und Anpassbarkeit, wie sie in [Wen07] genannt wurden, mit Bezug auf die Erweiterung der Analyse, ausgewertet werden.

Ein Problem, das bei der Interpretation der Ergebnisse der Verhaltensanalyse auftaucht, ist die Anzahl der untersuchten Traces. Wie in der praktischen Anwendung dargelegt wurde, kommt es vor, dass durch Methodenaufrufe, die einem Trigger entsprechen, Subtraces eines Traces untersucht werden. Dabei kann es vorkommen, dass ein Subtrace von einem akzeptierten Trace nicht akzeptiert wird, weil wichtige Methodenaufrufe fehlen<sup>1</sup>. Daher überwiegt die Anzahl der verworfenen Traces meist der Anzahl der akzeptierten Traces, obwohl es sich um ein korrektes Laufzeitverhalten des Entwurfsmusters handelt. Dies kann zu einer fehlerhaften Bewertung der Ergebnisse durch den Benutzer führen. Besonders lange Traces sind eigentlich ein besonders positives Indiz für eine wirkliche Entwurfsmusterimplementierung. Jedoch steigt die Anzahl der Trigger und damit die der analysierten Subtraces mit der Länge der Traces an, was mehr verworfene Subtraces zur Folge haben kann. Diese Problematik sollte nicht unterschätzt werden. Eine Lösung für dieses Problem wäre die Filterung der verworfenen Traces durch Trigger. Wenn man verhindert, dass in einem Trace mehrere Trigger enthalten sind, wird die Anzahl der verworfenen Traces reduziert. Dies stellt eine weitere Schwierigkeit dar, da Verhaltensmuster in der Regel nicht das globale Verhalten eines Entwurfsmusters beschreiben, sondern Ausschnitte typischen lokalen Verhaltens. Die Beobachtung mehrerer Traces ist also notwendig. Vielleicht ist auch die Vergabe von Prioritäten für die durch Methodenaufrufe, die einen Trigger darstellen, entstandene Traces eine Möglichkeit zur Behebung des Problems. Traces, die Subtraces von bereits akzeptierten Traces darstellen, erhalten dabei eine niedrigere Priorität, was der Benutzer in seine Bewertung des Ergebnisses einfließen lassen kann. Eine weitere Möglichkeit zur Lösung des Problems wäre der Einsatz von Methodenaufrufsequenzen als Trigger, anstatt von einzelnen Methodenaufrufen. Für das Observer-Muster könnte man beispielsweise eine Sequenz aus einer beliebig großen Anzahl aufeinanderfolgenden register-Aufrufe als Trigger verwenden.

Zusammenfassend kann man sagen, dass die dynamische Entwurfsmustererkennung von RECLIPSE das Potential für weitere vielversprechende Erweiterungen bietet und sich weitere Arbeit auf dem Gebiet lohnt.

---

<sup>1</sup>vgl. Ergebnisse der Untersuchung der Observer-Implementierung in Abschnitt 5.2.2



# Anhang A

## Anhang

### A.1 Verhaltensmuster

#### A.1.1 Observer

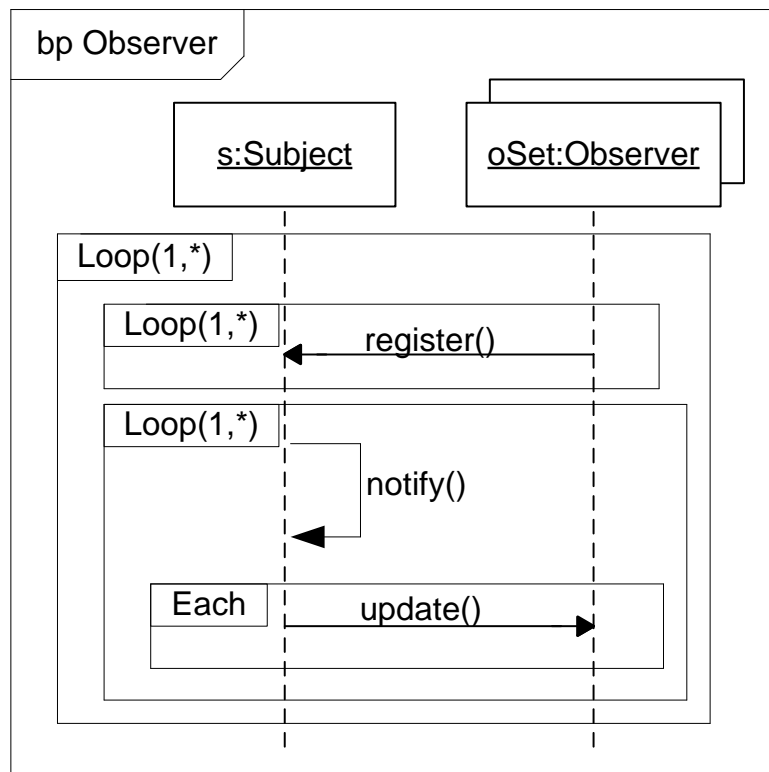


Abbildung A.1: Observer-Verhaltensmuster

## A.1.2 Chain Of Responsibility

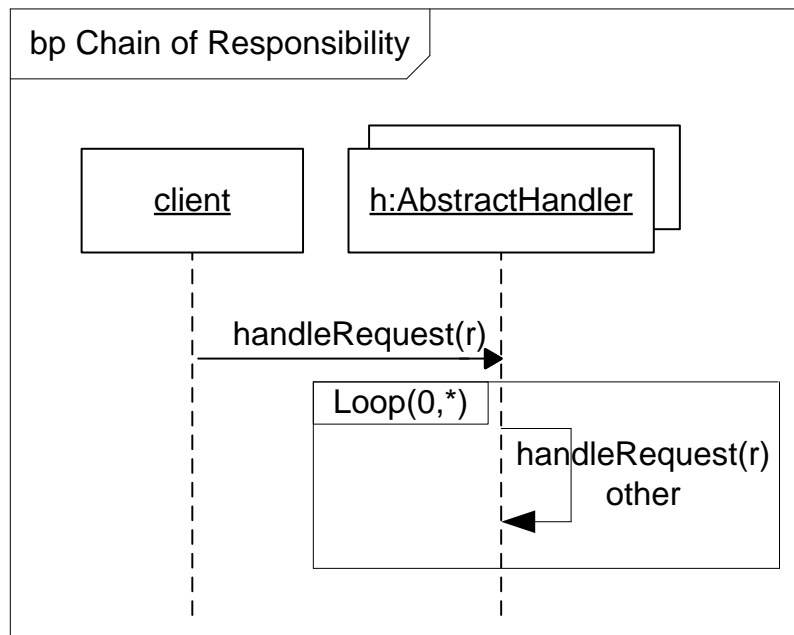


Abbildung A.2: Chain-of-Responsibility-Verhaltensmuster

## A.1.3 State

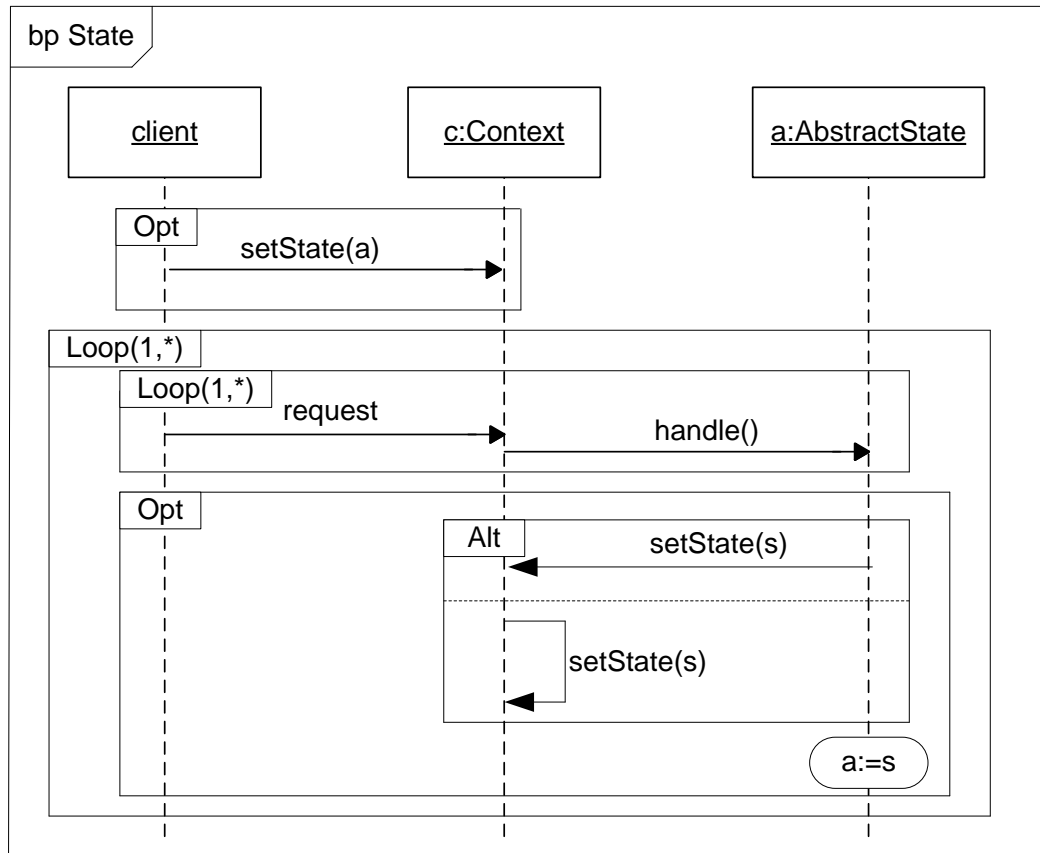


Abbildung A.3: State-Verhaltensmuster

## A.2 Neue Datenformate

### A.2.1 Verhaltensmusterkatalog

```
<!ELEMENT BehavioralPatternsCatalog (BehavioralPatternEntry*)>

<!ELEMENT BehavioralPatternEntry (DFA, Trigger+)>
<!ATTLIST BehavioralPatternEntry name CDATA #REQUIRED
                                negative CDATA #IMPLIED>

<!ELEMENT DFA ((PermittedMethodCallSymbol|
                ProhibitedMethodCallSymbol|
                ProhibitedCallerSymbol)+, State+, Assignment*, Transition*,
                StartState, RejectingState)>

<!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
                typeName CDATA #REQUIRED
                set CDATA #IMPLIED
                forEach CDATA #IMPLIED>

<!ELEMENT Callee EMPTY>
<!ATTLIST Callee name CDATA #REQUIRED
                typeName CDATA #REQUIRED
                set CDATA #IMPLIED
                forEach CDATA #IMPLIED>

<!ELEMENT PermittedCaller EMPTY>
<!ATTLIST PermittedCaller name CDATA #REQUIRED
                typeName CDATA #REQUIRED
                set CDATA #IMPLIED
                forEach CDATA #IMPLIED>
```

```
<!ELEMENT State EMPTY>
<!ATTLIST State id CDATA #REQUIRED
               name CDATA #IMPLIED
               type CDATA #REQUIRED>

<!ELEMENT Assignment EMPTY>
<!ATTLIST Assignment id CDATA #REQUIRED
                    leftSide CDATA #REQUIRED
                    rightSide CDATA #REQUIRED>

<!ELEMENT Transition EMPTY>
<!ATTLIST Transition previousStateId CDATA #REQUIRED
                    nextStateId CDATA #REQUIRED
                    symbolId CDATA #REQUIRED
                    assingmentId CDATA #IMPLIED>

<!ELEMENT StartState EMPTY>
<!ATTLIST StartState id CDATA #REQUIRED>

<!ELEMENT RejectingState EMPTY>
<!ATTLIST RejectingState id CDATA #REQUIRED>

<!ELEMENT Trigger EMPTY>
<!ATTLIST Trigger callerName CDATA #IMPLIED
                 callerTypeName CDATA #IMPLIED
                 calleeName CDATA #IMPLIED
                 calleeTypeName CDATA #REQUIRED
                 methodName CDATA #REQUIRED>
```

## A.3 Modelle der Beispielprogramme

### A.3.1 Observer-Beispiel

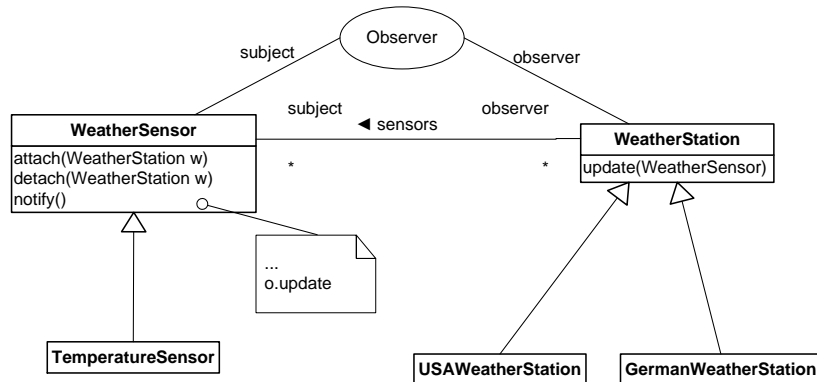


Abbildung A.4: Observer-Kandidat aus dem Beispielprogramm

### A.3.2 State-Beispiel

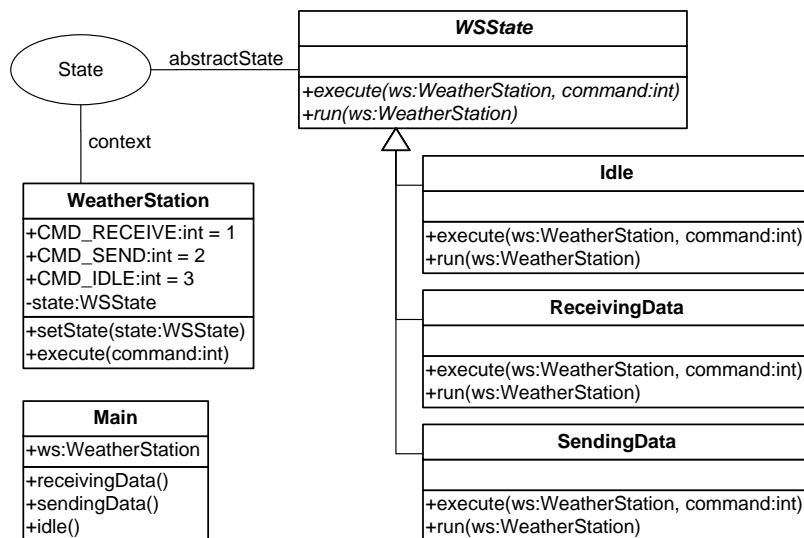


Abbildung A.5: State-Kandidat aus dem Beispielprogramm

# Literatur

- [Ant98] ANTONIOL, G.: Design Pattern Recovery in Object-Oriented Software, 1998
- [Boo08] BOOCH, G.: Architectural Organizational Patterns. In: *IEEE Software* (2008), S. 18–19
- [BP00] BERGENTI, F.; POGGI, A.: Improving UML Designs Using Automatic Design Pattern Detection, 2000
- [CJ90] CHIKOFFSKY, EJ; JH, II: Reverse engineering and design recovery: A taxonomy. In: *IEEE software* (1990), S. 13–17
- [GB03] GAMMA, E.; BECK, K.: *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 2003
- [GHJV95] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J.: *Design Patterns: elements of reusable object-oriented software*. Bd. 395. 1995
- [GMW06] GIESE, H.; MEYER, M.; WAGNER, R.: A Prototype for Guideline Checking and Model Transformation in Matlab/Simulink. In: *Proc. of the 4rd International Fujaba Days 2006* (2006)
- [HHHL03] HEUZEROTH, D.; HOLL, T.; HOGSTROM, G.; LOWE, W.: Automatic design pattern detection, 2003
- [HKS<sup>+</sup>08] HAYASHI, S.; KATADA, J.; SAKAMOTO, R.; KOBAYASHI, T.; SAEKI, M.: Design Pattern Detection by Using Meta Patterns. In: *IEICE TRANSACTIONS on Information and Systems* (2008)
- [HML03] HEUZEROTH, D.; MANDEL, S.; LOWE, W.: Generating design pattern detectors from pattern specifications. In: *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings, 2003*
- [HU93] HOPCROFT, J.E.; ULLMAN, J.D.: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. 1993
- [KGH06] KACZOROL, O.; GUÉHÉNEUC, Y.; HAMEL, S.: Efficient Identification of Design Patterns with Bit-vector Algorithm, 2006

- [KP96] KRÄMER, C.; PRECHELT, L.: Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software, 1996
- [KSR99] KELLER, R. K.; SCHAUER, R.; ROBITAILLE, P.: Pattern-Based Reverse-Engineering of Design Components, 1999
- [Mey06] MEYER, M.: Pattern-based Reengineering of Software Systems, 2006
- [Nie04] NIERE, J.: Inkrementelle Entwurfsmustererkennung. (2004)
- [Pet05] PETERSSON, N.: Measuring precision for static and dynamic design pattern recognition as a function of coverage. In: *ACM SIGSOFT Software Engineering Notes* (2005)
- [PS97] PREE, W.; SIKORA, H.: Design patterns for object-oriented software development (tutorial). In: *Proceedings of the 19th international conference on Software engineering* ACM New York, NY, USA, 1997
- [SG98] SEEMANN, J.; VON GUDENBERG, J. W.: Pattern-Based Design Recovery of Java Software, 1998
- [SH08] SARTIPI, K.; HU, L.: Behavior-Driven Design Pattern Recovery, 2008
- [SO06] SHI, N.; OLSSON, R. A.: Reverse Engineering of Design Patterns from Java Source Code, 2006
- [TCHS05] TSANTALIS, N.; CHATZIGEORGIOU, A.; HALKIDIS, S. T.; STEPHANIDES, G.: A Novel Approach to Automated Design Pattern Detection, 2005
- [Wen07] WENDEHALS, L.: Struktur- und verhaltensbasierte Entwurfsmustererkennung. (2007)
- [WQY98] WOODS, S. G.; QUILICI, A. E.; YANG, Q.: Constraint-Based Design Recovery for Software Reengineering - Theory and Experiments. (1998)
- [Zün01] ZÜNDORF, A.: Rigorous object oriented software development. In: *University of Paderborn* (2001)

# Abbildungsverzeichnis

1.1	Observer-Verhaltensmuster . . . . .	3
1.2	Observer-Verhaltensmuster mit erweiterter Syntax nach [Wen07] .	4
1.3	State-Verhaltensmuster mit erweiterter Syntax [Wen07] . . . . .	5
2.1	Strukturmuster des Observer-Entwurfsmusters . . . . .	9
2.2	Metamodell der Strukturmuster nach [Wen07] . . . . .	11
2.3	Observer-Entwurfsmuster-Kandidat . . . . .	12
2.4	Der kombinierte Erkennungsprozess . . . . .	12
2.5	Observer-Verhaltensmuster . . . . .	13
2.6	Metamodell der Verhaltensmuster nach [Wen07] . . . . .	14
2.7	Ausschnitt aus Observer-Kandidat und -Strukturmuster . . . . .	16
2.8	Ausschnitt aus Observer-Kandidat und -Verhaltensmuster . . . .	17
2.9	Transformation einer Nachricht des Verhaltensmusters in ein Auto- matenfragment . . . . .	17
2.10	Automat aus dem Observer-Verhaltensmuster . . . . .	18
2.11	Ergebnisse einer Strukturanalyse . . . . .	20
3.1	Observer-Verhaltensmuster in bisheriger Syntax . . . . .	21
3.2	State-Verhaltensmuster in bisheriger Syntax . . . . .	22
3.3	Mengenwertiges Verhaltensmusterobjekt, sog. Mengenobjekt . . .	23
3.4	Nachrichten von und zu Mengenobjekten . . . . .	24
3.5	Each-Fragment . . . . .	24
3.6	Nachrichteneigenschaft self call . . . . .	25
3.7	Methodenaufruf mit Argument . . . . .	26
3.8	Zuweisungsobjekt . . . . .	26
3.9	Neues Observer-Verhaltensmuster . . . . .	27
3.10	Chain-of-Responsibility-Verhaltensmuster . . . . .	28
3.11	Neues State-Verhaltensmuster . . . . .	29
3.12	Transformationsregel für Mengenobjekte . . . . .	30
3.13	Transformationsregel für Each-Fragmente . . . . .	31
3.14	Transformationsregel für <code>self call = true</code> . . . . .	31
3.15	Transformationsregel für <code>self call = false</code> . . . . .	31
3.16	Transformationsregel für Zuweisungen . . . . .	32
3.17	Automat aus dem neuen Observer-Verhaltensmuster . . . . .	32
3.18	Übergangsautomat für einen Kandidaten des Observer-Musters . .	34
3.19	Beispiel-Trace . . . . .	34
3.20	Aufzeichnung der Zustände beim Erkennungsvorgang . . . . .	35

4.1	Erweitertes Metamodell für Verhaltensmuster . . . . .	38
4.2	Erweitertes Modell des deterministischen Automaten . . . . .	42
4.3	Veranschaulichung eines Schrittes des Analyse-Algorithmus . . . . .	44
4.4	Veranschaulichung der Methode <code>accept(MethodCall, Token)</code> der Klasse <code>Transition</code> . . . . .	45
4.5	Erweiterung von <code>Transition::accept(MethodCall, Token)</code> für Zuweisungen . . . . .	47
4.6	Screenshot des erweiterten Verhaltensmuster-Editors mit Mengenobjekt und Each-Fragment . . . . .	48
4.7	Screenshot des Editors zu Nachrichteneigenschaft <code>self call</code> . . . . .	49
4.8	Screenshot des Verhaltensmuster-Editors mit Argumenten und einer Zuweisung . . . . .	50
5.1	Ergebnisse der statischen Analyse . . . . .	52
5.2	Ergebnisse der Untersuchung mit zwei Observern und dem neuen Verhaltensmuster . . . . .	53
5.3	Trace für das Observer-Beispiel mit drei Observern . . . . .	54
5.4	State-Kandidat der Beispielimplementierung . . . . .	55
5.5	Untersuchter Trace für das State-Beispiel . . . . .	56
5.6	Analyseergebnisse für das State-Beispiel . . . . .	57
A.1	Observer-Verhaltensmuster . . . . .	67
A.2	Chain-of-Responsibility-Verhaltensmuster . . . . .	68
A.3	State-Verhaltensmuster . . . . .	69
A.4	Observer-Kandidat aus dem Beispielprogramm . . . . .	72
A.5	State-Kandidat aus dem Beispielprogramm . . . . .	72

# Tabellenverzeichnis

2.1	Variablenbindungen für den Beispiel-Kandidaten . . . . .	16
2.2	Ergebnisse einer Verhaltensanalyse . . . . .	19
5.1	Variablenbindungen für den State-Kandidaten . . . . .	56