TECHNISCHE UNIVERSITÄT ILMENAU

Fakultät für Informatik und Automatisierung

Institut für Theoretische und Technische Informatik

Fachgebiet Prozessinformatik

Analyse von Algorithmen zur automatisierten Entwurfsmustererkennung

Diplomarbeit zur Erlangung des akademischen Grades **Diplom-Informatiker**

vorgelegt der Fakultät für Informatik und Automatisierung der
Technischen Universität Ilmenau von

Antje Seidel
geboren am 10. März 1979 in Arnstadt
Matrikel 97
Matrikelnummer 26136

Betreuende Hochschullehrerin: Prof. Dr.-Ing. habil. Ilka Phillipow
Betreuer: Dipl.-Inf. Detlef Streitferdt

Beginn der Diplomarbeit: 05. Januar 2004
Abgabe der Diplomarbeit: 05. Juli 2004
Inventarisierungsnummer: 2004-07-05 / 065 / IN97 / 2232

Kurzfassung

Untersuchungen zeigen, dass in der Software-Entwicklung mehr als 30 Prozent der Projekte fehlschlagen oder vorzeitig abgebrochen werden und gut die Hälfte wird mit Mängeln bzw. nicht komplett erfüllten Anforderungen ausgeliefert. Lediglich 16 Prozent können erfolgreich abgeschlossen werden. Umfragen zu den Problemen in der Softwareentwicklung ergaben, dass speziell in dem Bereich Anforderungsspezifikation sowie im Bereich Analyse und Entwurf erhebliche Schwierigkeiten auftreten.

Ein Entwurfsmuster (engl. *Design Pattern*) beschreibt ein bestimmtes, häufig auftretendes Entwurfsproblem und bietet dazu einen Lösungsansatz in Form eines generischen Schemas. Durch einen verstärkten Einsatz von Entwurfsmustern in der Entwurfsphase kann die Qualität und die Wartbarkeit von Software erhöht werden, wobei von den Erfahrungen anderer Programmierer profitiert wird.

Da die Wartungskosten von Software rund 50 Prozent der Gesamtkosten ausmachen, ist eine Unterstützung in diesem Bereich sinnvoll. Um Software zu Warten, ist das Verständnis des Systems Voraussetzung. Das Erkennen von verwendeten Entwurfsmustern im Systementwurf kann einen großen Teil zum Verständnis beitragen. Es ist aber meist der Fall, dass die verwendeten Muster nicht gekennzeichnet oder dokumentiert sind und kein Entwickler mehr zur Verfügung steht, welcher diesbezüglich verlässlich Auskunft geben kann. In diesem Fall ist eine automatisierte Entwurfsmustererkennung von enormem Vorteil.

Sebastian Naumann hat in seiner Diplomarbeit [Naumann01] verschiedene Ansätze dazu untersucht und auf dieser Grundlage einen eigenen Ansatz entwickelt. Der daraus resultierende Pseudocode wird implementiert und als Modul in die Entwicklungsumgebung Together ControlCenter integriert. Anhand verschiedener Systeme werden die vorgelegten Algorithmen getestet und deren Leistungsfähigkeit im Einzelnen ausgewertet. In einigen Fällen werden Vorschläge zur Verbesserung der erzielten Ergebnisse gegeben.

Inventarisierungsnummer 2004-07-05 / 065 / IN97 / 2232

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AOL	**Abstract Object Language**
API	**Application Programmers Interface**
AWT	**Abstract Windowing Toolkit**
bzw.	**beziehungsweise**
CASE-Tool	**Computer Aided Software Engineering Tool**
d. h.	**das heißt**
DTD	**Document Type Definition**
RWI	**Read Write Interface**
SCI	**Source Code Interface**
UML	**Unified Modelling Language**
XML	**Extensible Markup Language**
z.B.	**zum Beispiel**

1 Einführung

1.1 Bedeutung von Entwurfsmustern in der Software-Entwicklung

Die Entwicklung von Software wird in fünf Phasen unterteilt. Am Beginn steht die Anforderungsanalyse gefolgt vom Software- und Systementwurf. Nach der Implementierung und erfolgreichem Test wird die Software beim Kunden installiert und gewartet. Speziell in der objektorientierten Softwareentwicklung enthalten die meisten heute bekannten Analyse- und Entwurfsmethoden eigene Vorgehensmodelle. Abgesehen davon, dass in diesen die objektorientierte Terminologie verwendet, der zyklische Charakter der Softwareentwicklung betont und auf die Wichtigkeit der Wiederverwendung hingewiesen wird, stehen sie noch weitgehend in der Tradition der Wasserfall-Modelle. [Gumm+00] Damit verbunden kommen auch die Nachteile dieser Modelle zur Wirkung. Beispielsweise sind grobe Fehler des Entwurfs, wenn sie erst in der abschließenden Phase des Tests festgestellt werden, sehr aufwändig zu beheben, da der Entwurf und die Implementierung geändert und anschließend erneut getestet werden muss. Abbildung 1 veranschaulicht die Ergebnisse einer Umfrage zu Problemen in der Softwareentwicklung in Europa.

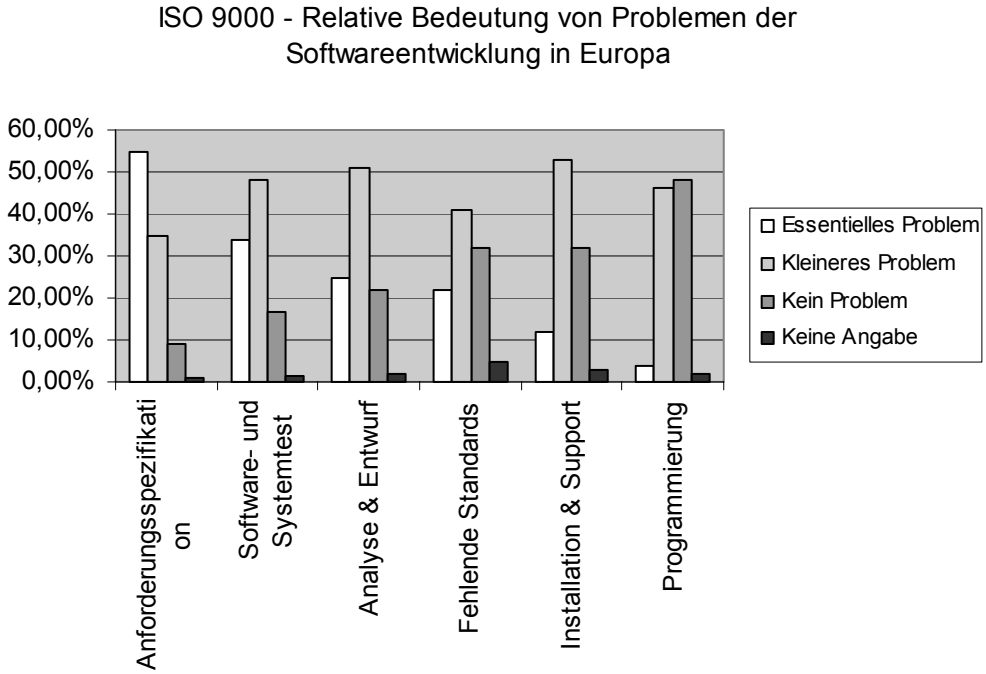

Abbildung 1: Probleme in der Softwareentwicklung in Europa

Aus dem Diagramm wird deutlich, dass die Anforderungsspezifikation essentielle Probleme birgt und auch Analyse und Entwurf sehr problematisch sind. Im Vergleich dazu geht die Programmierung nahezu problemlos von statten.

Eine Untersuchung von Software-Projekten innerhalb der USA ergab, dass mehr als 30 Prozent davon fehlschlagen oder frühzeitig abgebrochen werden und über 50 Prozent werden mit Mängeln ausgeliefert. Nur 16 Prozent aller Projekte können erfolgreich abgeschlossen werden. Abbildung 2 stellt die Ergebnisse graphisch dar.

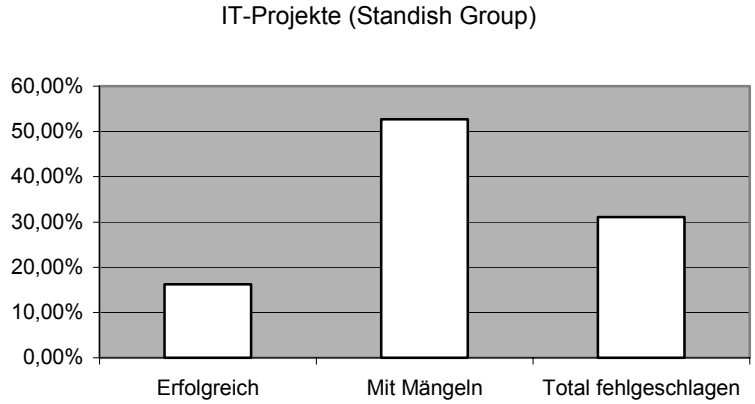

Abbildung 2: Projektergebnisse (Standish Group)

Ein Entwurfsmuster (engl. *Design Pattern*) beschreibt ein bestimmtes, häufig auftretendes Entwurfsproblem und bietet dazu einen Lösungsansatz in Form eines generischen Schemas. Dieses spezifiziert die beteiligten Komponenten und ihre Beziehungen untereinander. Es ist sinnvoll, eine Liste von Vorbedingungen für die Anwendung des Entwurfsmusters aufzustellen und die aus dem Entwurfsmuster resultierenden Vor- und Nachteile zu dokumentieren. Durch die Vergabe eines einheitlichen Namens wird die Identifizierung des Entwurfsmusters gesichert.

Die Kenntnis und die Anwendung von Entwurfsmustern unterstützen den Entwickler speziell in den Phasen Anforderungsanalyse und Softwareentwurf. Entwurfsmuster verleihen der Anwendung Flexibilität und bereits bewährte Konzepte werden wieder verwendet. Dies hat positive Auswirkungen auf die Qualität und die Wartbarkeit der Software. Allerdings setzt das die Kenntnis und den sicheren Umgang mit Entwurfsmustern voraus.

Die Wartungskosten innerhalb des Softwarelebenszyklus betragen rund 50 Prozent der Gesamtkosten. Das Programmverstehen macht wiederum etwa 50 Prozent der Wartung

aus. Da dem Programmverstehen dabei solch eine hohe Bedeutung zukommt, kann durch eine Unterstützung in diesem Bereich eine Kostenreduzierung und Effizienzsteigerung erreicht werden. [Naumann01] Eine zuverlässige, automatisierte Suche nach Entwurfsmustern kann für den Entwickler bezüglich des Programmverstehens eine immense Unterstützung bedeuten.

1.2 Problemstellung

Im vorangegangenen Abschnitt wird die Bedeutung des Einsatzes von Entwurfsmustern innerhalb der Softwareentwicklung deutlich. Zum einen ist die Verwendung von Entwurfsmustern mit einem Gewinn an Flexibilität und Wartbarkeit der Software verbunden und zum anderen ermöglichen sie Außenstehenden ein besseres Verständnis des Softwareentwurfes, sofern diese mit den eingesetzten Mustern vertraut sind. Ohne eine genaue Dokumentation der Verwendung einzelner Muster ist dies allerdings nicht möglich. Da Entwurfsmuster meist nicht speziell gekennzeichnet oder dokumentiert sind, wird für den Bereich der Wartung von Software eine automatisierte Erkennung von Entwurfsmustern angestrebt.

Die für diese Arbeit verwendeten Grundlagen zur automatisierten Entwurfsmustererkennung wurden in der Diplomarbeit von Sebastian Naumann [Naumann01] gelegt. Auf Basis der Entwurfsmuster-Beschreibungen in *Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software* [Gamma+96] wurden eine Reihe von Kriterien zur Erkennung der 23 darin beschriebenen Entwurfsmuster entwickelt. Aufgrund dieser Kriterien entwarf Sebastian Naumann Algorithmen zum Auffinden aller Entwurfsmuster, welche er in einer Pseudocode-Notation hinterlegte.

Ziel dieser Arbeit ist es, diesen Pseudocode zu implementieren und die Algorithmen zu bewerten und gegebenenfalls zu verbessern.

Im folgenden Kapitel werden einige Ansätze zur Bewertung der Algorithmen besprochen und im Anschluss daran werden verschiedene Werkzeuge zur Mustererkennung vorgestellt, wobei die Aufmerksamkeit speziell auf den Test der Werkzeuge gelegt wird.

2 Analyse von Suchalgorithmen - Stand der Technik

2.1 Analyse von Algorithmen

2.1.1 Die Kennwerte Präzision und Recall

Ziel eines Suchalgorithmus ist das Auffinden von allen Implementierungen eines Entwurfsmusters innerhalb eines Softwaresystems. Um die Leistungsfähigkeit eines solchen Algorithmus bewerten zu können, werden alle gefundenen Muster nach [Streitferdt] folgendermaßen klassifiziert:

Positive true:	Das gefundene Muster ist eine Umsetzung des gesuchten Musters.
False positive:	Es wurde ein Muster erkannt, aber durch eine nachträgliche Untersuchung ergibt sich, dass es sich nicht um eine Umsetzung des gesuchten Musters handelt.
False negative:	Ein tatsächlich im System vorhandenes Muster wurde vom Algorithmus nicht als solches erkannt.

Aufgrund dieser Klassifizierung lassen sich weitere Leistungsmerkmale ableiten:

Recall:	Der Recall gibt an, wie viel Prozent der enthaltenen Muster auch erkannt worden sind.
Präzision:	Die Präzision hingegen gibt an, wie viel Prozent der gefundenen Muster auch Implementierungen des gesuchten Musters sind.

Da es in der Praxis meistens der Fall ist, dass nicht im Voraus bekannt ist, welche Muster wie oft in einem Softwareprodukt angewendet worden sind, ist eine verlässliche Aussage über den Wert Recall nicht möglich. Man kann lediglich davon ausgehen, dass die

Algorithmen so gestaltet sind, dass immer alle Implementierungen gefunden werden. Diese Annahme führt folglich stets zu einem Recall-Wert von 100 Prozent. Dadurch verliert dieser Wert allerdings seine Aussagekraft über die Leistungsfähigkeit des Werkzeugs.

Der Wert der Präzision ist in der Praxis besser zu bestimmen. Dazu muss allerdings eine nachträgliche Untersuchung der Ergebnisse erfolgen. Durch diese können mögliche Musterkandidaten gefunden werden, welche schließlich doch keine Implementierung des gesuchten Musters darstellen. Werden keine solchen Muster gefunden, so kann eine Präzision von 100 Prozent angegeben werden.

Wenn davon ausgegangen wird, dass ein Algorithmus zum Auffinden von Entwurfsmustern alle Kriterien beinhaltet, die ein Muster beschreiben, so ergibt sich die Aussage: Alle Muster, welche durch den Algorithmus gefunden wurden, erfüllen alle Anforderungen und sind folglich Implementierungen des gesuchten Musters. Weiterhin lässt sich feststellen, dass es keine weiteren Mustervorkommen außer diesen gibt, da sie sonst durch den Algorithmus gefunden worden wären. Somit folgt, dass die Werte Recall und Präzision stets 100 Prozent betragen, falls der Algorithmus nur die richtigen Kriterien beachtet.

Aufgrund dieser Betrachtungen sollte die Aussagekraft dieser Kennwerte nicht zu sehr überschätzt werden. Sie können allerdings Anstoß zu neuen Überlegungen und zur Verbesserung der Suchkriterien geben.

2.1.2 Ansätze zur automatisierten Mustererkennung

Naumann stellt in seiner Arbeit [Naumann01] eine Vielzahl an Vorgehensweisen zur Mustererkennung vor. Für nahezu alle Ansätze wurde ein Werkzeug entwickelt, welches in fast allen Fällen auch einem Test unterzogen wurde. Im Folgenden werden diese kurz vorgestellt, wobei der Schwerpunkt der Betrachtungen auf den Test gelegt wird.

DP++

Das von Jagdish Bansiya entwickelte und in [Bansiya98] vorgestellte DP++ ist ein Werkzeug zum Auffinden von Entwurfsmustern in C++ Quelltext. Es besteht aus folgenden drei Subsystemen: dem C++ Code Translation System zur Analyse des Quelltextes, dem Pattern Detection System zum Auffinden von Entwurfsmustern und dem Display System zur Visualisierung der gefundenen Muster.

Zur Analyse des Quelltextes werden allerdings ausschließlich die Header-Dateien eingelesen, was zu einem enormen Verlust von essentiellen Informationen führt, da keine Aussagen über die Implementierung von Methoden getroffen werden können. Aufgrund dieser Einschränkungen ist das Erkennen von Erzeugungs- und Verhaltensmustern nahezu unmöglich. Allerdings werden alle Strukturmuster von dem entwickelten Werkzeug abgedeckt.

Um die Leistungsfähigkeit von DP++ zu testen, untersuchte Bansiya ein zuvor von ihm selbst implementiertes Projekt. Durch die überschaubare Anzahl an Klassen konnte die Überprüfung der Korrektheit der Musterkandidaten mittels direkter Kontrolle nachgewiesen werden. Dies ist allerdings keine Vorgehensweise, welche in der Praxis bestand haben kann. Im Falle eines sehr umfangreichen Projektes wäre eine solche Nachuntersuchung viel zu zeitaufwändig.

KT

Das von Kyle Brown in [Brown96] vorgestellte Werkzeug KT ist für die Mustersuche in Smalltalk Quelltext entwickelt worden. Dabei wird der Quelltext nicht direkt untersucht, stattdessen wird dieser als Klassen- und Sequenzdiagramm dargestellt. Auf den Diagrammen werden dann die Untersuchungen hinsichtlich der enthaltenen Entwurfsmuster vorgenommen. Die statischen Zusammenhänge aus dem Klassendiagramm unterstützen die Suche nach Strukturmustern, wobei ohne die dynamischen Aspekte die Erkennung von Verhaltensmustern sehr schwierig wäre.

Zum Test des Systems wurden vier verschieden umfangreiche Systeme ausgewählt. Die erzielten Testergebnisse wurden wie auch bei Bansiya durch nachträgliche direkte Kontrolle auf ihre Korrektheit geprüft.

SPOOL

Das Werkzeug SPOOL wurde von Keller, Schauer, Robitaille und Pagé [Keller+99] zur Mustersuche in C++ Quelltext entwickelt. Im Gegensatz zu DP++ untersucht es allerdings den kompletten Code und nicht nur die Header-Dateien. Der Quelltext wird nach der Verarbeitung durch einen Source Code Parser durch ein UML-Modell (Unified Modeling

Language) dargestellt. Auf dem Modell erfolgt anschließend die Suche nach enthaltenen Entwurfsmustern.

Der Test der Leistungsfähigkeit des Systems wurde an drei verschiedenen Systemen vorgenommen. Bei einigen Mustern vertreten die Autoren den Standpunkt, dass eine nachträgliche Überprüfung der Ergebnisse nicht erforderlich ist, da jedes gefunden Muster den geforderten Merkmalen entspricht und somit auch ein korrektes Muster darstellt. In einigen Fällen, wie beispielsweise bei dem Brückenmuster, ist eine nachträgliche Untersuchung allerdings nicht erlässlich. Angaben über Präzision und Recall werden nicht gemacht, da diese den Autoren zufolge wenig aussagekräftig und zuweilen irreführend seien.

Pat

Das von Krämer und Prechelt [Krämer+96] entwickelte Werkzeug Pat untersucht ebenfalls C++ Quelltext hinsichtlich des Vorkommens von Entwurfsmustern. Auch ihr Ansatz analysiert lediglich die Header-Dateien. Die enthaltenen Informationen werden anschließend in Form einer Prolog-Datenbasis abgelegt. Mittels eigens entwickelter Prolog-Fakten wird diese dann nach Mustern durchsucht.

Zur Kontrolle der Korrektheit der gefundenen Muster wird von Krämer und Prechelt ebenfalls eine nachträgliche direkte Überprüfung vorgenommen. Von ihnen werden allerdings auch keine Angaben zu Präzision und Recall gemacht.

IDEA

Das Werkzeug IDEA basiert ebenfalls auf Prolog-Regeln. Allerdings wird nicht der Quelltext selbst dargestellt, sondern die daraus abgeleiteten Struktur- und Kollaborationsdiagramme. Innerhalb dieser Strukturen wird dann nach vorhandenen Entwurfsmustern gesucht.

Leider machen Bergenti und Poggi in [Bergenti+00] keine Angaben über die Leistungsfähigkeit ihres Werkzeuges.

Mehrstufiger Suchprozess

Antoniol, Fiutem und Cristoforetti [Antoniol+98] nutzen einen mehrstufigen Suchprozess zur Entwurfsmustererkennung. Dabei untersuchen sie die Klassenstruktur des vorliegenden Quelltextes, welche sie entweder direkt aus dem Quelltext oder aus vorhandenen Entwurfsdokumenten extrahieren. Diese wird in Form von AOL (Abstract Object Language) hinterlegt und anschließend mittels eines Pattern Recognizers nach Mustern untersucht. Zur Identifikation der Muster steht eine Datenbank mit verschiedenen möglichen Ausprägungen einzelner Muster zur Verfügung.

Die daraus resultierenden Musterkandidaten werden anschließend durch direkte Kontrolle auf ihre Korrektheit überprüft. Die Autoren geben dazu den Hinweis, dass durch solch eine nachträgliche Kontrolle nicht alle tatsächlich enthaltenen Muster gefunden werden können und somit der Wert Recall immer 100 Prozent beträgt. Die Präzision für ihre Algorithmen geben sie mit durchschnittlich 35 Prozent an.

Flexible Musterdefinition und Fuzzy Logik

Niere, Wadsack und Wendehals [Niere+01] gehen von der Vielgestaltigkeit der Implementierungen von Mustern aus und entwickeln darauf aufbauend eine flexible Definition von Entwurfsmustern. Grundgedanke dabei ist, dass ein Muster auf Submustern basiert und von anderen Mustern erben kann. Zum Zeitpunkt der Veröffentlichung ihrer Ansätze wurde an der Implementierung dieser noch gearbeitet, weshalb keine Angaben über ihre Leistungsfähigkeit gemacht werden konnten.

Pattern Wizard

Für den Pattern Wizard entwickelten Kim und Boldyreff [Kim+00] einige Metriken, um die Charakteristika aller 23 Muster zu bestimmen. Dazu müssen zuerst für den zu untersuchenden Quelltext eben diese Metriken aufgestellt und anschließend mit den musterspezifischen Metriken verglichen werden.

Die so erhaltenen Musterkandidaten wurden von Kim und Boldyreff durch eine direkte Nachkontrolle auf ihre Korrektheit überprüft. Aus diesen Ergebnissen ließ sich eine Präzision des Werkzeuges von 44 Prozent bestimmen. Zum Recall werden an dieser Stelle allerdings keine Angaben gemacht.

BACKDOOR

Shull, Melo und Basili [Shull+96] stellen eine Methode zur manuellen Mustererkennung vor. Es wird ein Vorgehen in sechs Schritten beschrieben, welches zum Teil auch durch Werkzeuge unterstützt werden kann. Getestet wurde ihre Methode an sieben Studentenprojekten der Universität Maryland, wobei insgesamt 22 Mustervorkommen gefunden wurden.

Algorithmen nach Naumann

Sebastian Naumann entwickelte im Rahmen seiner Diplomarbeit [Naumann01] 23 Algorithmen zum Auffinden aller in [Gamma+96] beschriebenen Entwurfsmuster. Diese theoretischen Überlegungen wurden in Form einer Pseudocode-Notation hinterlegt.

Zur Unterstützung seiner Aussagen wurden drei dieser Algorithmen prototypisch in Form von Rational Rose Scripten umgesetzt. Dabei konnten einige Unzulänglichkeiten von Seiten Rational Rose festgestellt werden, welche eine realistische Bewertung der Algorithmen nicht ermöglichte. Aus diesem Grund konnten auch keine Aussagen über Recall und Präzision der Algorithmen getroffen werden.

Fazit

Alle vorgestellten Systeme sind Eigenentwicklungen, welche nicht sehr aussagekräftigen Tests bis hin zu keinerlei Tests unterzogen wurden. Falls es zum Test kam, wurden lediglich die Resultate der Ausführung auf ihre Richtigkeit hin geprüft. Dabei werden nur die Fälle *Positiv true* und *False positiv* entdeckt. Die Menge der Fälle *False negative* kann durch diese Vorgehensweise nicht bestimmt werden. Dadurch ist lediglich eine Angabe des Recall-Wertes möglich, welcher hierbei stets 100 Prozent beträgt, weil davon ausgegangen wird, dass außer den gefundenen Mustern keine weiteren Muster enthalten sind. Der Präzisions-Wert hingegen kann eindeutig bestimmt werden.

Die Probleme beim Aufstellen des Recall-Wertes begründen sich in dem Mangel der Verfügbarkeit eines Referenzsystems, welches eine Dokumentation über das Enthaltensein von Entwurfsmustern besitzt. Ohne ein solches System ist eine vergleichbare Analyse der Algorithmen nicht möglich. Eine Zusammenfassung der Ausführungen wird in Tabelle 1 gegeben.

Tabelle 1: Vorhandene Werkzeuge und Ansätze

Werkzeug	Quelltext	Verarbeitung	Test der Korrektheit der Ergebnisse	Recall	Präzision
DP++	C++	Verarbeitung der Header-Dateien	nachträgliche Überprüfung	keine Angabe	keine Angabe
KT	Smalltalk	Gesamter Quelltext	nachträgliche Überprüfung	keine Angabe	keine Angabe
SPOOL	C++	Verarbeitung des gesamten Quelltextes zu UML-Modell	nachträgliche Überprüfung	keine Angabe	keine Angabe
Pat	C++	Verarbeitung der Header-Dateien durch Prolog	nachträgliche Überprüfung	keine Angabe	keine Angabe
IDEA	UML-Modell	Verarbeitung der aus dem Modell gewonnenen Prologfakten	keine Angabe	keine Angabe	keine Angabe
Werkzeug zum mehrstufigen Suchprozess	C++ Quelltext oder Design Dokumente	Verarbeitung des UML-Modells	nachträgliche Überprüfung	100%	35%
Flexible Musterdefi-nition und Fuzzy Logik	Java	Verarbeitung des gesamten Quelltextes			
Pattern Wizard	C und C++	Verarbeitung des gesamten Quelltextes zu Metriken	nachträgliche Überprüfung	keine Angabe	44%
BACK-DOOR	Quelltext einer objekt-orientierten Sprache	Sichtung des Quelltextes			
Rational Rose Scripte	C++	Verarbeitung des Quelltextes zu UML-Klassendia-grammen und Analyse der Diagramme	nachträgliche Überprüfung	keine Angabe	keine Angabe

2.1.3 Das Referenzsystem

Die Verfügbarkeit eines Test- bzw. Referenzsystems ist eine unablässige Voraussetzung für die Analyse der von Naumann entwickelten Algorithmen. Im Idealfall sollte dieses System für die vorgesehenen Untersuchungen folgenden Anforderungen entsprechen:

- **offener Quelltext**

  Der Quelltext des Systems steht zur freien Verfügung, damit er mittels Reverse-Engineering eingelesen und dessen Struktur weiterverarbeitet werden kann. Das System wurde mit Java implementiert.

- **ausführliche Dokumentation**

  Aufgrund der Dokumentation des Systems können Aussagen über das Vorhandensein von konkreten Entwurfsmustern getroffen werden. Ist ein Muster vorhanden, sind die Anzahl des Auftretens und die beteiligten Klassen bekannt.

- **Vollständigkeit**

  Alle 23 in [Gamma+96] beschriebenen Muster sind mehrfach und in möglichst verschiedenen Ausprägungen im System enthalten.

Trotz intensiver Bemühungen ist es nicht gelungen, ein System zu finden, welches alle drei Anforderungen zur Zufriedenheit erfüllt. Infolge dessen wurde zur Analyse der Algorithmen ein neues Software-Projekt erstellt, welches in seiner Funktion eher rudimentär geblieben ist, allerdings in seiner Struktur die erforderlichen Entwurfsmuster beinhaltet.

Des Weiteren wird die Leistungsfähigkeit der Algorithmen an folgenden drei Systemen evaluiert: einem Zeicheneditor (Drawlet), den Klassen eines Application Servers (Tomcat) und einer Klassenbibliothek (AWT). Diese Softwaresysteme werden in Kapitel 3.2 *Entwicklung eines Referenzsystems* etwas ausführlicher vorgestellt.

2.2 Probleme bei der Mustererkennung

2.2.1 Assoziation, Aggregation und Komposition

Die UML bietet eine Vielzahl von Möglichkeiten, eine Beziehung zwischen zwei oder mehreren Klassen darzustellen. Die einfachste Form ist eine Assoziation. Dies ist die schwächste Bindung zwischen zwei Klassen. Aggregation und Komposition sind speziellere Assoziationen, welche eine festere Bindung symbolisieren.

Ein Beispiel für eine Assoziation ist die Beziehung zwischen den Klassen Professor und Vorlesung. Ein Professor kann Vorlesungen halten oder auch nicht. Dies ist eine schwache Bindung.

Eine Aggregation beschreibt im Gegensatz dazu eine etwas stärkere Bindung zwischen Klassen. Sie drückt eine „ist Teil von“ Beziehung aus. Die Klassen Rad und Auto bilden eine Aggregation. Das Rad ist ein Teil von einem Auto, kann aber durchaus auch separat existieren.

Eine noch stärkere Bindung als die Aggreation ist die Komposition. Sie drückt ebenfalls eine „ist Teil von“ Beziehung aus, wobei in diesem Fall das Teil nicht ohne das Ganze existieren kann. So verhält es sich beispielsweise mit den Klassen Auto und Karosserie.

Abbildung 3 zeigt die UML-Darstellung der hier erwähnten Beispiele.

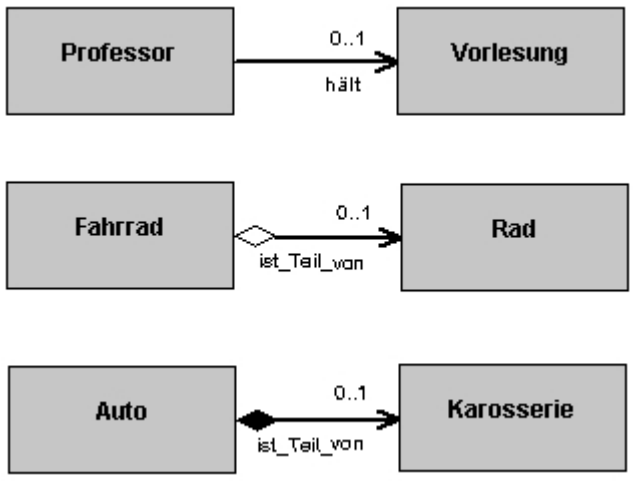

Abbildung 3: Beziehungen zwischen Klassen

Die Umsetzung dieser Beziehungen im Quelltext ist abhängig von den Möglichkeiten der Implementierung der gewählten Programmiersprache. In Java werden diese Abhängigkeiten meist durch Klassenvariablen wiedergespiegelt. Handelt es sich um eine höhere Multiplizität, wird diese durch Listen von Objekten dargestellt. C und C++ bieten darüber hinaus noch die Möglichkeit, diese durch Zeiger zu realisieren.

Im Folgenden wird die Realisierung der Beziehungen als Klassenvariable, als Liste von Objekten und als Zeiger in Quelltextform beispielhaft aufgezeigt:

```
private Vorlesung vorlesung;
private Vorlesung[] vorlesungen;
private Vorlesung *vorlesung;
```

Erschwerend wirkt sich aus, dass in der Praxis beispielsweise von einer vorhandenen Klassenvariablen nicht mehr auf die Art der Beziehung geschlossen werden kann. Together legt diese Metadaten in Form von Kommentaren bezüglich der Variablen ab. Falls man mit anderen Umgebungen entwickelt, kann es dazu führen, dass derartige Informationen aus dem Entwurf bei der Implementierung vollständig verloren gehen und im Nachhinein auch nicht wieder rekonstruiert werden können.

Bei der Mustererkennung ist die Identifizierung solcher Beziehungen zwischen Klassen eine wichtige Voraussetzung. Die verschiedenen Möglichkeiten der Umsetzung von Beziehungen macht dies allerdings schwierig.

2.2.2 Polymorphismus

Polymorphismus ist ein programmiertechnisches Gestaltungsmittel, welches für die Umsetzung einiger Muster von enormer Bedeutung ist. Die objektorientierten Sprachen C++ und Java unterscheiden sich grundsätzlich im Umgang damit. Dies zeigt sich speziell bei der Implementierung von polymorphen Methoden. In Java ist jede Methode standardmäßig polymorph. Wird diese Eigenschaft nicht gewünscht, so muss dies ausdrücklich durch das Schlüsselwort `final` angegeben werden. C++ behandelt dieses Problem genau entgegengesetzt. Soll eine Methode polymorph sein, so muss dies explizit durch das Schlüsselwort `virtual` definiert sein. Solche programmiertechnischen Unterschiede machen es schwierig, ein Werkzeug zu entwickeln, welches polymorphe Methoden in verschiedenen Programmiersprachen erkennt.

Um diese und ähnliche Probleme zu behandeln, wird die Entwicklung einer API angestrebt, welche dem Entwickler von Algorithmen zur Mustererkennung eine Basis von Funktionen bietet, welche unabhängig von der Programmiersprache des zu untersuchenden Quelltextes die benötigten Funktionen ausführen. So sollte es beispielsweise eine Funktion geben, welche angibt, ob eine bestimmte Methode im Quelltext polymorph ist.

2.2.3 Erkennen von Objekterzeugungen

Das Erkennen von Objekterzeugungen innerhalb von Methoden ist eine wichtige Voraussetzung für die Identifikation von Erzeugungsmustern. Die meisten Werkzeuge, welche ein Reverse-Engineering von C oder C++ Quelltext anbieten, analysieren allerdings lediglich die Header-Dateien, wobei Informationen über die Implementierung von Methoden verloren gehen. Ohne diese Informationen ist das Erkennen von vielen Mustern nur sehr begrenzt möglich. Im Fall von Java Quelltext sind die Informationen zwar vorhanden, aber schwierig zu extrahieren.

Dies wäre eine weitere Funktion, welche durch oben beschriebene API abgedeckt werden könnte. Es benötigt eine Funktion zur Überprüfung, ob innerhalb einer Methode oder einer Klasse Objekte erzeugt werden und von welchem Typ diese sind.

Da in dieser Arbeit die Untersuchung auf Java-Quelltext beschränkt ist, wurde die Suche nach Objekterzeugung auf die Suche nach dem Schlüsselwort `new` ausgerichtet. Java lässt allerdings auch andere Arten der Objekterzeugung zu, welche bei der Umsetzung der Algorithmen nicht erkannt werden. Folgende Quelltextzeile zeigt die Erzeugung eines Objektes vom Typ String, welches ohne das Schlüsselwort `new` erfolgt und folglich nicht festgestellt werden kann.

```
String s = "neuer String";
```

2.2.4 Das Mustersuch-API

Wie bereits erwähnt, kann eine eigens für die Mustersuche zugeschnittene API die Entwicklung von Algorithmen zur Mustersuche stark vereinfachen. Falls solch eine Basis zur Verfügung gestellt werden könnte, ist ein Verzicht auf den umständlichen Zugriff auf die OpenAPI von Together möglich. Dadurch werden die Implementierungen übersichtlicher und eventuell auch laufzeiteffizienter.

2.3 Präzisierte Problemstellung

Aufgrund der vorangehenden Betrachtungen kann jetzt eine präzisierte Aufgabenstellung formuliert werden.

Im ersten Kapitel wurde die Bedeutung der Entwurfsmuster beim Softwareentwurf hervorgehoben und die Vorteile einer automatisierten Mustererkennung im Bereich der Wartung von Software erörtert.

Gegenstand dieser Arbeit ist der Test und die Verbesserung der 23 von Naumann in [Naumann01] entwickelten Algorithmen. Da diese Algorithmen nur in Form von Pseudocode vorhanden sind, bedarf es einer Implementierung. Grundlage der von Naumann entwickelten Algorithmen ist das UML-Klassendiagramm. Die UML (Unified Modelling Language) ist allerdings eine allgemeine Beschreibungssprache, welche nicht auf die technischen Besonderheiten von speziellen Programmiersprachen eingeht. Dadurch kommt es zu Problemen bei der Umsetzung des Pseudocodes, welche im Kapitel 2.2 *Probleme bei der Mustererkennung* näher erläutert wurden.

Um beim Test der Algorithmen aussagekräftige Ergebnisse zu erlangen, wird ein Softwaresystem benötigt, welches den in Kapitel 2.1.3 *Das Referenzsystem* entwickelten Anforderungen entspricht. Da kein entsprechendes System gefunden werden konnte, muss ein eigenes entworfen und implementiert werden.

Anhand dieses Referenzsystems können die von Naumann entworfenen Algorithmen getestet und bewertet werden. Auf Basis der Bewertungen werden die Algorithmen entsprechend angepasst, wobei der Schwerpunkt auf der Verbesserung der Leistungsfähigkeit liegt.

3 Eigener Anteil

Im vorhergehenden Kapitel wurde das Testen von Werkzeugen zur Mustererkennung untersucht. Den Kennwerten Präzision und Recall kommt in diesem Zusammenhang eine Schlüsselrolle zu, da nur sie den Vergleich verschiedener Ansätze zulassen. Um die Präzision und den Recall auch für die von Naumann entwickelten Algorithmen bestimmen zu können, müssen diese implementiert und anhand eines Referenzsystems getestet werden, was im folgenden beschrieben wird.

3.1 Implementierung der Algorithmen

Die Firma Togethersoft bietet mit dem Together ControlCenter ein Computer Aided Software Engineering Tool (CASE-Tool), welches im Rahmen dieser Arbeit sowohl für das Reverse-Engineering des Quelltextes als auch für die Verarbeitung der daraus erlangten Informationen in Form von UML-Klassendiagrammen verwendet wird. Dabei werden die programminternen Schnittstellen Read Write Interface (RWI) und Source Code Interface (SCI) der OpenAPI verwendet. Die OpenAPI ist eine Zusammenstellung von verschiedenen Schnittstellen, welche den Zugriff auf interne Daten aus externen Programmen ermöglichen. Im Verlauf der Implementierung soll der Entwurf einer weiteren Application Programmers Interfaces (API) vorgenommen werden, welche speziell auf die Anforderungen der Mustersuche zugeschnitten sein soll. Die gewünschte Struktur der Endanwendung ist in Abbildung 4 dargestellt.

Der von Naumann entwickelte Pseudocode wurde in Java implementiert und als Modul in die Entwicklungsumgebung Together ControlCenter 6.0.1 von Borland integriert. Kapitel 4 geht detaillierter auf die Umsetzung der Algorithmen ein.

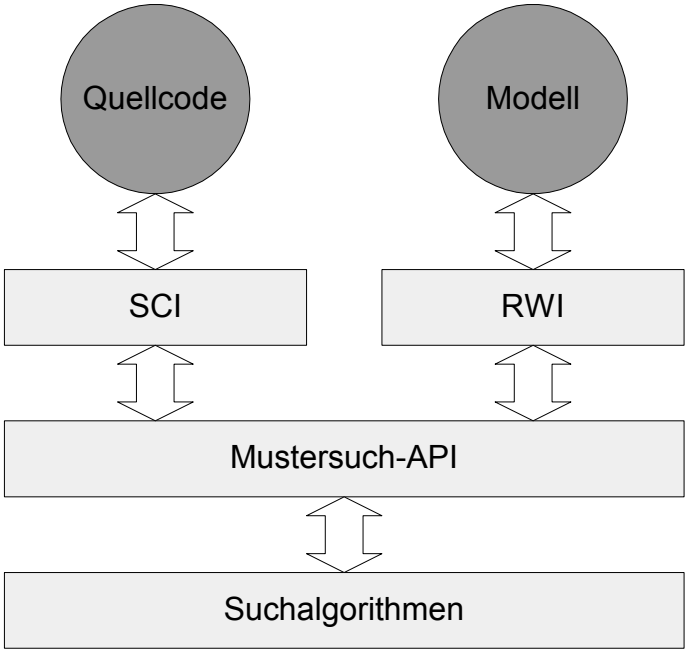

Abbildung 4: Struktur der Anwendung

Da die Mustersuch-API nicht zur Verfügung steht, wird alternativ direkt auf die OpenAPI von Together zugegriffen. Mittels des Read and Write Interfaces können die Daten aus dem Modell gelesen werden. Da Together allerdings beim Reverse-Engineering von C++ Quelltext lediglich die Header-Dateien behandelt, ist der Zugriff auf den konkreten Quelltext im Nachhinein über das Source Code Interface nicht mehr möglich. Daraus ergibt sich die Unmöglichkeit der Umsetzung einiger Algorithmen. Aus diesem Grund wird bei der Untersuchung der Algorithmen ausschließlich Java Quelltext analysiert.

Des Weiteren nutzt Together die Möglichkeit, spezifische Informationen zu Referenzen in Form von Kommentaren abzulegen. Die Kommentare identifizieren die Referenz als Assoziation, Aggregation oder Komposition. Diese Informationen werden beim Reverse-Engineering mit ausgewertet, was allerdings die Entwicklung des Quelltextes mit Together zur Voraussetzung hat. Da davon im Allgemeinen nicht ausgegangen werden kann, ist es in der Praxis wenig von Nutzen.

Die aus den Tests resultierenden Daten werden in Form von Dokumenten der Extensible Markup Language (XML) zur weiteren Verarbeitung hinterlegt. Dazu wird ein eigenes Schema zur Darstellung der zum Teil komplexen Musterstrukturen entwickelt und angewendet, welche ebenfalls in Kapitel 4 näher vorgestellt wird.

3.2 Entwicklung eines Referenzsystems

Wie im zweiten Kapitel bereits erwähnt, ist die Entwicklung eines eigenen Referenzsystems unabdingbar, um die Leistungsfähigkeit der Algorithmen zu analysieren. Im Rahmen dieser Diplomarbeit wird ein rudimentäres System entwickelt, welches alle 23 Muster aus [Gamma+96] strukturell abbildet, aber keine echte Funktionalität enthält. Zum Test der Algorithmen ist dies allerdings durchaus ausreichend.

Das System implementiert alle 23 Muster aus [Gamma+96] jeweils in einem separaten Paket. Die einzelnen Klassen beinhalten nur den für das Erkennen der Muster relevanten Quelltext. Insgesamt wurden 88 Klassen implementiert, weshalb aus Übersichtsgründen auf ein Klassendiagramm verzichtet wird.

Im Verlauf der Nachforschungen haben sich einige weitere Software-Projekte gefunden, welche zwar nicht alle Anforderungen an ein Referenzsystem erfüllen, aber dennoch einige Aussagen über die Leistungsfähigkeit der Algorithmen zulassen. Diese werden im Folgenden kurz vorgestellt und im Verlauf der Tests mit einbezogen.

- **Drawlet**

  Drawlet ist ein Zeicheneditor, der von der Firma Rolemodel Software [RMS] entwickelt wurde und in der Version 2.0 untersucht wird. Die Software hat einen Umfang von 195 Klassen.

- **AWT**

  Das AWT ist das Abstract Windowing Toolkit zur Erstellung von Fenstern mittels Java. Es ist Bestandteil der Java 2 Standard Edition [J2SE] und setzt sich aus 354 Klassen zusammen, welche grundlegende Funktionen und Gestaltungselemente beinhalten.

- **Tomcat**

  Tomcat ist ein Applikationsserver von Jakarta [Jakarta], welcher im Rahmen eines OpenSource Projektes entwickelt wurde und sehr weit verbreitet ist. Das Softwarepaket besteht im Ganzen aus 1035 Klassen.

Die folgende Tabelle veranschaulicht die bekannten implementierten Muster innerhalb der vier verwendeten Systeme. Die Informationen zum AWT und zu Tomcat entstammen den Seiten von PatternStories [Patterns]. Die Angaben zu Drawlet sind der Präsentation [Auer97] entnommen und die zum Pattern-Projekt ergeben sich aus der Tatsache, dass dies eine Eigenentwicklung ist, welche alle Muster implementiert.

Tabelle 2: Implementierte Entwurfsmuster

Entwurfsmuster	Patterns	Drawlet	AWT	Tomcat
Abstrakte Fabrik	X		X	
Erbauer	X			
Fabrikmethode	X	X		
Prototyp	X	X		
Singleton	X		X	
Adapter	X			X
Brücke	X		X	
Dekorierer	X			
Fassade	X			X
Fliegengewicht	X		X	
Kompositum	X	X	X	
Proxy	X			X
Befehl	X			
Beobachter	X	X	X	X
Besucher	X			
Interpreter	X			
Iterator	X	X		
Memento	X	X		
Schablonenmethode	X	X	X	
Strategie	X	X		X
Vermittler	X	X	X	
Zustand	X			
Zuständigkeitskette	X			X

3.3 Analyse der Algorithmen mittels Referenzsystem und weiteren Systemen

3.3.1 Durchführung der Tests

Um die Algorithmen zu testen, muss der Quelltext des zu testenden Systems als Projekt von Together vorliegen. Da das Referenzsystem Patterns mit Together entwickelt wurde, stellt dies kein Problem dar. Der Quelltext der weiteren Systeme wurde mittels Reverse-Engineering eingelesen und als Together-Projekt abgelegt.

Aus der Entwicklungsumgebung heraus lässt sich das Modul zur Mustererkennung starten, welches das aktuell geöffnete Projekt bearbeitet und die Ergebnisse in eine extern erstellte XML-Datei ablegt. Der Zielordner für diese Datei muss zuvor durch einen Eintrag in der eigens dafür erstellten Konfigurationsdatei `Mustererkennung.properties` festgelegt werden.

3.3.2 Ergebnisse der Tests

Um die Ergebnisse der Algorithmen nutzerfreundlich darzustellen und eine Weiterverarbeitung dieser zu ermöglichen, wurde zur Speicherung ein XML-Datenformat gewählt. XML ist eine Metasprache, welche die Definition von individuellen Datenstrukturen ermöglicht. Für jedes der 23 Muster aus [Gamma+96] wurde eine eigene Datenstruktur entwickelt, welche alle Elemente des entsprechenden Musters speichert. Diese Struktur wurde in einer eigenen Document Type Definition Datei (DTD-Datei) abgelegt, welche als Erzeugungsvorschrift für alle entstehenden XML-Dokumente gilt. Der Quelltext der DTD-Datei ist im Anhang A zu finden und der Quelltext der Datei Pattern.xml ist im Anhang B zu finden.

Die folgenden Tabellen dokumentieren die erzielten Ergebnisse der Algorithmen im Test mit dem Referenzsystem und den weiteren Systemen. Die Werte für Präzision und Recall konnten nur für das Patterns-Projekt bestimmt werden, da hier die Anzahl der implementierten Muster bekannt war.

Tabelle 3: Testergebnisse Patterns

ERZEUGUNGSMUSTER	ENTHALTEN	GEFUNDEN	PRÄZISION	RECALL
abstrakte Fabrik	1 Muster	1 Muster	100%	100%
Erbauer	1 Muster	12 Muster	8%	100%
Fabrikmethode	4 Muster	4 Muster	100%	100%
Prototyp	2 Muster	2 Muster	100%	100%
Singleton	1 Muster	1 Muster	100%	100%
STRUKTURMUSTER				
Adapter	1 Muster	3 Muster	33%	100%
Brücke	1 Muster	25 Muster	4%	100%
Dekorierer	1 Muster	1 Muster	100%	100%
Fassade	1 Muster	77 Muster	1%	100%
Fliegengewicht	1 Muster	1 Muster	100%	100%
Kompositum	1 Muster	1 Muster	100%	100%
Proxy	1 Muster	3 Muster	33%	100%
VERHALTENSMUSTER				
Befehl	1 Muster	1 Muster	100%	100%
Beobachter	1 Muster	1 Muster	100%	100%
Besucher	1 Muster	1 Muster	100%	100%
Interpreter	2 Muster	2 Muster	100%	100%
Iterator	1 Muster	1 Muster	100%	100%
Memento	1 Muster	1 Muster	100%	100%
Schablonenmethode	1 Muster	1 Muster	100%	100%
Strategie	1 Muster	17 Muster	6%	100%
Vermittler	1 Muster	25 Muster	4%	100%
Zustand	1 Muster	1 Muster	100%	100%
Zuständigkeitskette	1 Muster	3 Muster	33%	100%

Die Tabellen 4 bis 6 dokumentieren die gefundenen Muster der Systeme Drawlet, AWT und Tomcat. Hier konnten keine Angaben über die Anzahl der implementierten Muster gemacht werden. Es gibt lediglich die Aussage, dass bestimmte Muster auf jeden Fall implementiert sind.

Tabelle 4: Testergebnisse Drawlet 2.0

ERZEUGUNGSMUSTER	ENTHALTEN	GEFUNDEN
abstrakte Fabrik	unbekannt	3 Muster
Erbauer	unbekannt	87 Muster
Fabrikmethode	ja	13 Muster
Prototyp	ja	keine Muster
Singleton	unbekannt	keine Muster
STRUKTURMUSTER		
Adapter	unbekannt	40 Muster
Brücke	unbekannt	61 Muster
Dekorierer	unbekannt	keine Muster
Fassade	unbekannt	194 Muster
Fliegengewicht	unbekannt	keine Muster
Kompositum	ja	keine Muster
Proxy	unbekannt	9 Muster
VERHALTENSMUSTER		
Befehl	unbekannt	13 Muster
Beobachter	ja	keine Muster
Besucher	unbekannt	1 Muster
Interpreter	unbekannt	keine Muster
Iterator	ja	keine Muster
Memento	ja	keine Muster
Schablonenmethode	ja	1 Muster
Strategie	ja	4 Muster
Vermittler	ja	135 Muster
Zustand	unbekannt	keine Muster
Zuständigkeitskette	unbekannt	keine Muster

Tabelle 5: Testergebnisse AWT

ERZEUGUNGSMUSTER	ENTHALTEN	GEFUNDEN
abstrakte Fabrik	ja	10 Muster
Erbauer	unbekannt	127 Muster
Fabrikmethode	unbekannt	27 Muster
Prototyp	unbekannt	11 Muster
Singleton	ja	8 Muster
STRUKTURMUSTER		
Adapter	unbekannt	52 Muster
Brücke	ja	61 Muster
Dekorierer	unbekannt	keine Muster
Fassade	unbekannt	322 Muster
Fliegengewicht	ja	keine Muster
Kompositum	ja	keine Muster
Proxy	unbekannt	14 Muster
VERHALTENSMUSTER		
Befehl	unbekannt	20 Muster
Beobachter	ja	keine Muster
Besucher	unbekannt	6 Muster
Interpreter	unbekannt	keine Muster
Iterator	unbekannt	keine Muster
Memento	unbekannt	keine Muster
Schablonenmethode	ja	22 Muster
Strategie	unbekannt	4 Muster
Vermittler	ja	88 Muster
Zustand	unbekannt	keine Muster
Zuständigkeitskette	unbekannt	keine Muster

Tabelle 6: Testergebnisse Tomcat

ERZEUGUNGSMUSTER	ENTHALTEN	GEFUNDEN
abstrakte Fabrik	unbekannt	4 Muster
Erbauer	unbekannt	469 Muster
Fabrikmethode	unbekannt	22 Muster
Prototyp	unbekannt	2 Muster
Singleton	unbekannt	6 Muster
STRUKTURMUSTER		
Adapter	ja	123 Muster
Brücke	unbekannt	345 Muster
Dekorierer	unbekannt	keine Muster
Fassade	ja	973 Muster
Fliegengewicht	unbekannt	keine Muster
Kompositum	unbekannt	keine Muster
Proxy	ja	73 Muster
VERHALTENSMUSTER		
Befehl	unbekannt	46 Muster
Beobachter	ja	keine Muster
Besucher	unbekannt	4 Muster
Interpreter	unbekannt	keine Muster
Iterator	unbekannt	keine Muster
Memento	unbekannt	3 Muster
Schablonenmethode	unbekannt	17 Muster
Strategie	ja	12 Muster
Vermittler	unbekannt	220 Muster
Zustand	unbekannt	2 Muster
Zuständigkeitskette	ja	26 Muster

Die folgenden drei Kapitel betrachtet jeden Algorithmus im Einzelnen und nehmen aufgrund der Testergebnisse eine Bewertung des Algorithmus vor, wobei dessen Schwächen beziehungsweise mögliche Ursachen für schlechte Testergebnisse analysiert werden und in einigen Fällen Verbesserungsvorschläge gemacht werden.

3.3.3 Erzeugungsmuster

3.3.3.1 Abstrakte Fabrik

Das Abstrakte Fabrik-Muster besteht aus einer abstrakten und einer konkreten Fabrik. Des Weiteren gibt es Fabrikmethoden innerhalb der Fabrik sowie konkrete und abstrakte Produkte. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<abstract-factory>
   <abstract-factory-class>AbstrakteFabrik</abstract-factory-class>
   <concrete-factory-class>KonkreteFabrik</concrete-factory-class>
   <factory-methods>
      <factory-method>fabrikMethode1</factory-method>
      <factory-method>fabrikMethode2</factory-method>
   </factory-methods>
   <concrete-products>
      <concrete-product>KonkretesProdukt1</concrete-product>
      <concrete-product>KonkretesProdukt2</concrete-product>
   </concrete-products>
   <abstract-products>
      <abstract-product>AbstraktesProdukt</abstract-product>
   </abstract-products>
</abstract-factory>
```

Algorithmus 3.1: FindeAbstrakteFabrik nach Naumann [Naumann01]

```
für jede Klasse i (konkrete Fabrik) do
zaehler=0;
   für jede Methode j (Fabrikmethode) der Klasse i do
      erzeugt Methode j etwas?
      ja: ist Rückgabetyp eine Oberklasse von Erzeugtem?
      ja: zaehler+1;
   od
   zaehler größer oder gleich 2? ja: Muster gefunden;
od
```

Der Algorithmus 3.1 zum Auffinden des Musters Abstrakte Fabrik beschränkt die Suche nach dem Muster auf die Suche nach einer konkreten Fabrik, welche mindestens zwei Fabrikmethoden implementiert.

Um die Frage nach dem Erzeugen von Objekten beantworten zu können, ist eine Untersuchung der Implementierung der Methode notwendig. In diesem Fall wird nach dem Schlüsselwort `new` gesucht. Wird es im Quelltext aufgefunden, steht fest, dass die Methode ein neues Objekt erzeugt.

Zur Bestimmung der Klasse des erzeugten Objektes wird der Klassenname aus der entsprechenden Quelltextzeile extrahiert. Durch die Bestimmung der Menge aller

Superklassen der Klasse kann festgestellt werden, ob der Rückgabetyp der Methode in dieser Menge enthalten ist.

Im Test mit dem Referenzsystem Patterns konnte für diesen Algorithmus eine Präzision von 100 Prozent festgestellt werden. Der Recall beträgt ebenfalls 100 Prozent. Die im AWT enthaltenen Muster konnten identifiziert werden.

Der von Naumann entworfene Algorithmus beschränkt sich auf die Suche von konkreten Fabriken mit mindestens zwei Fabrikmethoden. Im Verlauf der Tests hat sich ergeben, dass dies zur Identifikation des Musters ausreichend ist.

3.3.3.2 Erbauer

Das Erbauer-Muster besteht aus einem abstrakten und einem konkreten Erbauer, einem zu erbauenden Produkt und aus einem oder mehreren Direktoren. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<builder>
   <builder-class>AbstrakterErbauer</builder-class>
   <concrete-builder>KonkreterErbauer</concrete-builder>
   <product>Produkt</product>
   <directors>
      <director>Direktor</director>
   </directors>
</builder>
```

Algorithmus 3.2a: FindeErbauer nach Naumann [Naumann01]

```
für jede Klasse i (konkreter Erbauer) do
   für alle Referenzen r (Referenzen auf Produkt) in Klasse i do
      index_liste=leer;
      für alle Methoden j (Zurückliefern des Produkts) in Klasse i do
         gibt Methode j Variable vom Typ r zurück?
         ja: index_liste=index_liste+j;
      od
      für alle Methoden j (Konstruktionsmethode) in Klasse i do
         arbeitet Methode j mit Referenz r?
         ja: ist j nicht in index_liste?
         ja: Muster gefunden
      od
   od
od
```

Der Algorithmus 3.2a *FindeErbauer* konzentriert sich auf die Suche nach konkreten Erbauern. Dabei werden die Methoden der Klasse daraufhin untersucht, ob sie das Produkt als Rückgabetyp haben oder an der Konstruktion des Produktes beteiligt sind.

Der Rückgabetyp von Methoden kann durch eine einfache Abfrage bestimmt und mit dem der referenzierten Klasse verglichen werden. Um zu prüfen, ob die Methode mit einer Referenz auf das Produkt arbeitet, wird sie nach dem Namen der Variablen durchsucht, welche die Referenz speichert.

Die im Referenzsystem Patterns enthaltenen Muster konnten durch diesen Algorithmus alle erkannt werden. Das entspricht einem Wert für den Recall von 100 Prozent. Da aber eine Vielzahl weiterer Muster gefunden wurden, welche nicht der Umsetzung des Entwurfsmusters entsprachen, konnte lediglich eine Präzision von 8 Prozent erreicht werden.

Im Test mit den anderen Systemen hat sich herausgestellt, dass einige Muster gefunden wurden, obwohl keine klare Aussage über das Vorhandensein dieser vorlag.

Der Algorithmus zum Auffinden von Erbauern konzentriert sich auf die Suche nach konkreten Erbauern. Dabei werden alle Methoden der Klasse auf ihre Funktion hin untersucht. Die Trennung von Rückgabemethode und Konstruktionsmethoden wird durch eine Indexliste realisiert. Allerdings sollte beachtet werden, dass diese Liste unter umständen auch leer sein kann. Dieser Fall sollte konkret ausgeschlossen werden. Algorithmus 3.2b enthält die dazu notwendigen Veränderungen.

Algorithmus 3.2b: FindeErbauer

```
für jede Klasse i (konkreter Erbauer) do
   für alle Referenzen r (Referenzen auf Produkt) in Klasse i do
      index_liste=leer;
      für alle Methoden j (Zurückliefern des Produkts) in Klasse i do
         gibt Methode j Variable vom Typ r zurück?
         ja: index_liste=index_liste+j;
      od
      ist index_liste leer?
      ja: Abbruch;
      für alle Methoden j (Konstruktionsmethode) in Klasse i do
         arbeitet Methode j mit Referenz r?
         ja: ist j nicht in index_liste?
         ja: Muster gefunden
      od
   od
od
```

3.3.3.3 Fabrikmethode

Das Fabrikmethode-Muster besteht aus mehreren Fabrikmethoden, einem abstrakten und einem konkreten Erbauer und einem abstrakten und einem konkreten Produkt. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<factory-method>
   <factory-method-names>
      <factory-method-name>fabrikMethode1</factory-method-name>
      <factory-method-name>fabrikMethode2</factory-method-name>
   </factory-method-names>
   <concrete-builder>KonkreteFabrik</concrete-builder>
   <builder>AbstrakteFabrik</builder>
   <concrete-product>KonkretesProdukt</concrete-product>
   <product>AbstraktesProdukt</product>
</factory-method>
```

Algorithmus 3.3: FindeFabrikmethode nach Naumann [Naumann01]

```
für jede Klasse i (Klasse mit Fabrikmethode) do
   für jede Methode j (Fabrikmethode) in Klasse i do
      erzeugt Methode j ein Objekt einer Klasse k (z.B.
      TuerMitZauberspruch), aber ist der Rückgabetyp l (z.B. Tuer)
      einer von k verschiedenen Klasse zugehörig?
      ja: ist Methode j polymorph?
      ja: ist die Klasse des Rückgabetyps l eine Oberklasse der Klasse
      des erzeugten Objekts k?
      ja: Muster gefunden
   od
od
```

Zur Identifikation von Fabrikmethoden werden alle Methoden einer Klasse auf Objekterzeugung hin untersucht. Falls eine Methode ein Objekt erzeugt und der Rückgabetyp der Methode einer Oberklasse vom Typ des erzeugten Objektes entspricht, muss lediglich noch überprüft werden, ob diese Methode polymorph ist.

Um zu klären, ob eine Methode ein Objekt erzeugt, muss in der Implementierung der Methode das Schlüsselwort `new` identifiziert werden. Der aus dem Quelltext extrahierte Typ des neuen Objektes kann anschließend mit dem Rückgabetyp der Methode verglichen werden, wobei auch die Vererbungsbeziehung untersucht wird.

Eine Methode in Java ist immer polymorph, sofern sie nicht durch das Schlüsselwort `final` gekennzeichnet ist. Daher ist die Überprüfung dieser Eigenschaft sehr einfach.

Im Test mittels des Pattern-Projektes hat der Algorithmus für die Werte Recall und Präzision jeweils 100 Prozent erreicht. Die im Drawlet-Projekt vorhandenen Muster konnten ebenfalls identifiziert werden. Die relevanten Merkmale sind einfach zu überprüfen. Daher gestaltet sich die Umsetzung des Algorithmus als trivial. Weiterhin ist festzustellen, dass die Auswertung dieser wenigen Kriterien bereits zu einer Präzision von 100 Prozent führt. Dieser Algorithmus arbeitet bereits optimal und erfordert keine Optimierungen mehr.

3.3.3.4 Prototyp

Das Prototyp-Muster besteht aus einem konkreten und einem abstrakten Prototyp und einem Klienten. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<prototype>
    <abstract-prototype>AbstrakterPrototyp</abstract-prototype>
   <prototype-class>KonkreterPrototyp</prototype-class>
   <client>Klient</client>
</prototype>
```

Algorithmus 3.4a: FindePrototyp nach Naumann [Naumann01]

```
für jede Klasse i (konkreter Prototyp) do
   besitzt Klasse Kopierkonstruktor?
   ja: gibt es Methode j (Klone-Operation), die Kopierkonstruktor
   nutzt, um Objekt der eigenen Klasse zu erzeugen, und ist diese
   Methode polymorph?
   ja: liefert diese Methode j die Klasse i oder Oberklasse der Klasse
   i zurück?
   ja: Muster gefunden
od
```

Dieser Algorithmus 3.4a *FindePrototyp* sucht im Quelltext nach konkreten Prototypen. Dabei werden die Konstruktoren der Klasse untersucht und alle Methoden auf die Nutzung dieser Konstruktoren hin. Stimmt der Rückgabetyp der Methode mit dem der Klasse selbst oder mit dem einer seiner Oberklassen überein, ist eine klone-Operation identifiziert und das Muster gefunden.

Um dem Kopierkonstruktor zu finden wird unter allen Methoden nach einer gesucht, deren Name mit dem der Klasse übereinstimmt. Des Weiteren muss diese Methode einen Parameter vom Typ der Klasse besitzen.

Zur Überprüfung, ob eine Klone-Methode existiert, wird der Quelltext der Methode nach dem Aufruf des Kopierkonstruktors durchsucht und auf Polymorphie geprüft. Für den zweiten Punkt genügt das Nichtvorhandensein einer `final` Deklaration, da alle nicht als `final` deklarierten Methoden in Java automatisch polymorph sind.

Der Rückgabetyp einer Methode kann leicht bestimmt und mit dem einer anderen Klasse und deren Oberklassen verglichen werden.

Durch die Überprüfung der oben genannten Kriterien wurden im Projekt Pattern eine Präzision und ein Recall von jeweils 100 Prozent erreicht. Allerdings wurden die im Projekt Drawlet bekanntermaßen vorhandenen Muster durch diesen Algorithmus nicht erkannt. Dies kann daran liegen, dass sich der Algorithmus sehr streng an die von Gamma

festgelegten Vorgaben hält und somit eine leicht abweichende Implementierung nicht erkennt.

Das Erkennen der Klone-Methode kann Probleme bereiten, falls diese nicht einen Kopierkonstruktor anwendet, sondern stattdessen mittels `return this` eine Referenz von sich selbst zurückliefert. Der Algorithmus muss um diese Überprüfung erweitert werden. Algorithmus 3.4b veranschaulicht diese Erweiterung.

Algorithmus 3.4b: FindePrototyp

```
für jede Klasse i (konkreter Prototyp) do
   besitzt Klasse Kopierkonstruktor?
   ja: gibt es Methode j (Klone-Operation), die den Kopierkonstruktor
   nutzt, um Objekt der eigenen Klasse zu erzeugen, und ist diese
   Methode polymorph?
   ja: liefert diese Methode j die Klasse i oder Oberklasse der Klasse
   i zurück?
   nein:
   gibt es Methode j, die eine Referenz auf sich selbst zurückgibt und
   polymorph ist?
   ja: Muster gefunden
od
```

3.3.3.5 Singleton

Das Singleton-Muster besteht aus nur einer Klasse. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<singleton>
   <singleton-class>Singleton</singleton-class>
</singleton>
```

Algorithmus 3.5: FindeSingleton nach Naumann [Naumann01]

```
für jede Klasse i (Singleton) do
   gibt es ein Attribut (Exemplar-Variable), das als Typ die eigene
   Klasse bzw. eine Oberklasse besitzt und statisch ist?
   ja: gibt es eine Methode (Exemplar-Operation), die als Rückgabetyp
   die eigene Klasse bzw. eine Oberklasse besitzt und die statisch
   ist?
   ja: gibt es keine public Konstruktoren, aber einen der protected
   bzw. private ist?
   ja: Muster gefunden
od
```

Da sich dieses Muster auf eine einzelne Klasse beschränkt, ist der Algorithmus 3.5 *FindeSingleton* sehr einfach gestaltet. Es wird die Existenz eines statischen Attributes vom Typ der Klasse oder einer ihrer Oberklassen nachgewiesen. Des Weiteren muss eine

Methode existieren, welche dieses Attribut zurückgibt und ein Konstruktor, welcher nicht öffentlich ist.

Der Typ aller statischen Attribute kann sehr einfach mit dem der Klasse verglichen werden. Ebenso kann jede Methode geprüft werden, ob sie statisch ist und ob ihr Rückgabetyp dem der Klasse oder einer ihrer Oberklassen entspricht.

Konstruktoren sind Methoden, deren Name dem der Klasse entspricht und sind dadurch leicht aufzufinden. Um auszuschließen, dass es einen öffentlichen Konstruktor gibt, muss lediglich überprüft werden, dass keine dieser Methoden öffentlich ist.

Im Verlauf des Tests konnte der Algorithmus eine Präzision und einen Recall von jeweils 100 Prozent erlangen. Dies bestätigt das Ergebnis der Untersuchungen des AWT, welche belegen, dass sich die bekanntlich vorhandenen Muster auch auffinden lassen. Daraus lässt sich schließen, dass eine Verbesserung des Algorithmus nicht notwendig ist.

3.3.4 Strukturmuster

3.3.4.1 Adapter

Da sich die Untersuchung der Algorithmen lediglich auf Java Quelltext bezieht, wird nur der Algorithmus FindeObjektadapter betrachtet. Es ist mit den sprachlichen Mitteln von Java nicht möglich, einen Klassenadapter zu implementieren, da eine Mehrfachvererbung nicht realisierbar ist.

Das ObjektAdapter-Muster besteht aus Adapter, Ziel und adaptierter Klasse. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<object-adapter>
   <adapter-class>Adapter</adapter-class>
   <adaptee>AdaptierteKlasse</adaptee>
   <target>Ziel</target>
</object-adapter>
```

Algorithmus 3.6: FindeObjektadapter nach Naumann [Naumann01]

```
für alle Klassen i (Adapter) do
   hat Klasse i Oberklasse (Ziel)?
   ja: hat Klasse i Referenz auf andere Klasse k (adaptierte Klasse)?
   ja: ist Klasse k mit Klasse i in einem Vererbungspfad?
   nein: für alle Methoden j der Klasse i do
      ist Methode j eine Überschreibung?
      ja: gibt es im Rumpf von Methode j einen Aufruf einer Methode
      von Klasse k?
      ja: Muster gefunden
   od
od
```

Der Algorithmus 3.6 *FindeObjektadapter* untersucht ausgehend von der Adapter-Klasse die Struktur des Musters. Die direkte Oberklasse dieser wird als Ziel-Klasse erkannt und alle referenzierten Klassen werden getestet, ob es sich um adaptierte Klassen handelt.

Um die Ziel-Klasse zu finden, genügt es, alle ausgehenden Links der Klasse auf Generalisierung bzw. Implementierung zu prüfen. Um dann festzustellen, ob zwei Klassen in einem Vererbungspfad liegen, werden alle Unter- und Oberklassen der ersten Klasse bestimmt. Ist die zweite Klasse enthalten, so liegen sie in einem Vererbungspfad.

Findet man die Deklaration einer Methode in einer der Oberklassen der Klasse, welche die Methode implementiert, so ist diese Methode überschrieben.

Im Test mittels des Pattern-Projektes ergab sich für diesen Algorithmus ein Recall von 100 Prozent. Die Präzision dieses Algorithmus beträgt 33 Prozent. Dieses nicht allzu gute Ergebnis gründet sich nicht auf einer Inkorrektheit des Algorithmus, sondern vielmehr auf der Beschränkung der Untersuchungen auf statische Aspekte. Dabei wird die Semantik der Implementierung vollkommen außer Acht gelassen. So werden beispielsweise im Proxy-Muster dieselben Strukturen angewendet, welche allerdings lediglich ein Ersatzobjekt für ein bestehendes Objekt mit derselben Schnittstelle zur Verfügung stellen.

3.3.4.2 Brücke

Das Brücke-Muster besteht aus Abstraktion und Implementierer. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<bridge>
   <abstraction>Abstraktion</abstraction>
   <implementor>Implementierer</implementor>
</bridge>
```

Algorithmus 3.7a: FindeBrücke nach Naumann [Naumann01]

```
für jede Klasse i (Abstraktion) do
   besitzt Klasse i Referenz auf Klasse j (Implementierer)?
   ja: do
      speichere Klasse i und alle ihre Unterklassen in einer
      Collection x (Abstraktionen);
      speichere alle Methoden der Klassen der Collection x in einer
      Collection y (Implementierer);
      durchsuche die Methodenrümpfe aller Klassen des Baumes der
      Klasse j nach Aufrufen von Methoden der Collection y; keine
      gefunden? weiter
      durchsuche alle Klassen des Baumes der Klasse j nach Referenzen
      auf eine Klasse der Collection x; keine gefunden? Muster erkannt
   od
od
```

Der vorliegende Algorithmus 3.7a *FindeBrücke* geht von einer Referenz zwischen zwei Klassen aus und testet Verbindungen zwischen den Klassen in den Vererbungspfaden beider Klassen. Werden keine solchen Verbindungen festgestellt, handelt es sich bei der Struktur um eine Implementierung des Brücke-Musters.

Der Recall dieses Algorithmus liegt bei 100 Prozent. Die sich aus dem Test ergebene Präzision von 4 Prozent ist kein zufrieden stellendes Ergebnis. Das liegt darin begründet, dass das Muster in seiner minimalen Ausprägung zu sehr beschränkt worden ist.

Alternativ kann der Algorithmus insofern erweitert werden, dass das Vorhandensein wenigstens einer spezialisierten Abstraktion und eines konkreten Implementierers gefordert ist. Algorithmus 3.7b beinhaltet die notwendigen Erweiterungen.

Algorithmus 3.7b: FindeBrücke

```
für jede Klasse i (Abstraktion) do
   besitzt Klasse i Referenz auf Klasse j (Implementierer)?
   ja: do
      speichere Klasse i und alle ihre Unterklassen in einer
      Collection x (Abstraktionen);
      hat Klasse i mindestens eine Unterklasse?
      ja: speichere alle Methoden der Klassen der Collection x in
      einer Collection y (Implementierer);
      hat Klasse j mindestens eine Unterklasse?
      ja: durchsuche die Methodenrümpfe aller Klassen des Baumes der
      Klasse j nach Aufrufen von Methoden der Collection y; keine
      gefunden? weiter
      durchsuche alle Klassen des Baumes der Klasse j nach Referenzen
      auf eine Klasse der Collection x; keine gefunden? Muster erkannt
   od
od
```

3.3.4.3 Dekorierer

Das Dekorierer-Muster besteht aus abstrakten und konkreten Dekorierer und einer Komponente. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<decorator>
   <concrete-decorator>KonkreterDekorierer</concrete-decorator>
   <decorator-class>AbstrakterDekorierer</decorator-class>
   <component>Komponente</component>
</decorator>
```

Algorithmus 3.8: FindeDekorierer nach Naumann [Naumann01]

```
für jede Klasse i (Dekorierer) do
   gibt es 1-zu-1-Aggregation zu einer Klasse j (Komponente)?
   ja: ist Klasse j Oberklasse von Klasse i?
   ja: hat Klasse i Unterklasse k (konkreter Dekorierer)?
   ja: für alle Methoden l von Klasse k do
      gibt es ein Aufruf auf Klasse i gefolgt von lokalem
      Methodenaufruf?
      ja: enthält die in Methode l aufgerufene Methode der Klasse i
      einen Aufruf der gleichnamigen Methode der Klasse j?
      ja: Muster gefunden
   od
od
```

Die Suche nach dem Dekorierer-Muster geht von der Klasse Dekorierer aus und sucht nach der Komponente und nach den konkreten Dekorierern. Wurde dabei die überschriebene Methode zum Dekorieren von Komponenten identifiziert, wurde das Muster gefunden.

Die Kardinalität einer Aggregationsbeziehung lässt sich trivial abfragen.

Zum Test, ob eine Klasse die Oberklasse einer anderen Klasse ist, kann die Menge aller Oberklassen aufgestellt und getestet werden, ob die fragliche Klasse darin enthalten ist.

Ähnlich ist das Vorgehen, um festzustellen, ob eine Klasse Unterklassen besitzt. Dazu wird die Menge aller Unterklassen gebildet. Ist diese leer, steht fest, dass keine Unterklassen existieren.

Der Aufruf einer Methode einer anderen Klasse innerhalb einer Methode kann durch sequentielles Durchlaufen des Quelltextes der Methode gefunden werden. Ist dies der Fall, so kann anschließend ebenso nach einem lokalen Methodenaufruf gesucht werden.

Der Test des Algorithmus 3.8 *FindeDekorierer* mittels des Pattern-Projektes ergab für die Werte Präzision und Recall jeweils 100 Prozent. Leider war das Vorkommen dieses Musters in den anderen zum Test verwendeten Software-Projekten nicht bekannt. Es konnten auch keine weiteren Muster identifiziert werden.

3.3.4.4 Fassade

Das Fassade-Muster besteht im Wesentlichen aus einer Fassade-Klasse. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<facade>
   <facade-class>Fassade</facade-class>
</facade>
```

Algorithmus 3.9a: FindeFassade nach Naumann [Naumann01]

```
für jede Klasse i (Fassade) do
   bilde Menge B (Subsystem) der Klassen, auf die Klasse i Referenz
   hat;
   für jede Klasse j der Menge B do
      hat Klasse j Referenz auf Klasse i? nein: weiter, ja: Abbruch
   od
   bilde Menge A (Klienten) der Klassen, die auf Klasse i Referenz
   haben
   für jede Klasse j der Menge B do
      hat Klasse j Referenz auf eine Klasse aus A? nein: weiter, ja:
      Abbruch
   od
   Muster gefunden
od
```

Der Algorithmus 3.9a *FindeFassade* bildet die Mengen A und B. Die Menge A beinhaltet alle Klassen, die eine Referenz auf die Fassade verwalten und alle Klassen, auf die die Fassade eine Referenz verwaltet, bilden die Menge B. Die Menge B wird bezüglich Referenzen auf die Fassade und bezüglich Referenzen auf Klassen aus der Menge A überprüft. Werden keine dieser Referenzen gefunden, so ist das Muster gefunden.

Der Test dieses Algorithmus mittels des Pattern-Projektes ergab einen Recall von 100 Prozent. Allerdings ist die Präzision von lediglich 1 Prozent nicht zufrieden stellend. In allen weiteren Test-Projekten wurden sehr viele Muster gefunden, aber ob diese auch Implementierungen des Fassade-Musters sind, ist nicht erwiesen. Mittels dieses Algorithmus sind keine zuverlässigen Aussagen über die Umsetzung des Fassade-Musters möglich.

Ein Ansatz zur Verbesserung des Algorithmus sieht das zwingende Vorhandensein von wenigstens einem Klienten und das eines Subsystems von einer Größe von mindestens drei Klassen vor. Dies wurde in Algorithmus 3.9b umgesetzt.

Algorithmus 3.9b: FindeFassade

```
für jede Klasse i (Fassade) do
   bilde Menge B (Subsystem) der Klassen, auf die Klasse i Referenz
   hat;
   hat die Menge B mindestens drei Elemente?
   ja: für jede Klasse j der Menge B do
      hat Klasse j Referenz auf Klasse i? nein: weiter, ja: Abbruch
   od
   bilde Menge A (Klienten) der Klassen, die auf Klasse i Referenz
   haben
   hat die Menge A mindestens ein Element?
   ja: für jede Klasse j der Menge B do
      hat Klasse j Referenz auf eine Klasse aus A? nein: weiter, ja:
      Abbruch
   od
   Muster gefunden
od
```

3.3.4.5 Fliegengewicht

Das Fliegengewicht-Muster besteht aus abstraktem und konkretem Fliegengewicht und einer Fliegengewichtfabrik. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<flyweight>
   <flyweight-factory>FliegengewichtFabrik</flyweight-factory>
   <flyweight-class>AbstraktesFliegengewicht</flyweight-class>
   <concrete-flyweight>KonkretesFliegengewicht</concrete-flyweight>
</flyweight>
```

Algorithmus 3.10a: FindeFliegengewicht nach Naumann [Naumann01]

```
für jede Klasse i (FliegengewichtFabrik) do
   zaehler=0;
   für jede Methode j der Klasse i do
      erzeugt Methode j ein Objekt einer Klasse, welche gleichzeitig
      Rückgabetyp von Methode j ist? ja: zaehler+1
   od
   ist zaehler>0? ja: weiter, nein: Abbruch;
   besitzt Klasse i eine 1-zu-n-Referenz auf eine andere Klasse j
   (Fliegengewicht)?
   ja: gibt es einen Parameter, den alle Methoden der Klasse j
   gemeinsam besitzen?
   ja: für alle Unterklassen k (konkrete Fliegengewichte) der Klasse j
   do
      wird Objekt von k in Klasse i (Fabrik) erzeugt? ja: Muster
      gefunden
   od
od
```

Ausgehend von der Fliegengewichtfabrik wird nach einer 1-zu-n Referenz auf das Fliegengewicht und von dort aus nach konkreten Fliegengewichten gesucht. Lässt sich eine Methode zum Erzeugen von Fliegengewichten finden, so ist das Muster identifiziert.

Eine Methode erzeugt ein Objekt, wenn sich in ihrem Quelltext das Schlüsselwort `new` finden lässt. Daraufhin kann der Typ des erzeugten Objektes bestimmt und mit dem Rückgabetyp der Methode verglichen werden.

Die Kardinalität einer Referenz zwischen zwei Klassen ist hingegen sehr einfach abzufragen.

Um den gemeinsamen Parameter zu finden, werden die Parameter aller Methoden bestimmt und miteinander verglichen.

Falls die Fabrik ein Fliegengewicht-Objekt erzeugt, so ist in dem Quelltext einer ihrer Methoden das Schlüsselwort `new` enthalten und der Typ des erzeugten Objektes entspricht dem der Fliegengewicht-Klasse.

Die im AWT enthaltenen Fliegengewicht-Muster konnten vom vorliegenden Algorithmus 3.10a *FindeFliegengewicht* nicht erkannt werden, obwohl er beim Test mittels des Pattern-Projektes einen Recall sowie eine Präzision von 100 Prozent erreichte. Dies liegt in der Nichterkennung der 1-zu-n Referenz begründet, welche beim Reverse-Engineering schwierig zu erkennen ist. Eine mögliche Strategie zur Verbesserung der Ergebnisse wäre die Vernachlässigung dieser Bedingung, wie Algorithmus 3.10b zeigt.

Algorithmus 3.10b: FindeFliegengewicht

```
für jede Klasse i (FliegengewichtFabrik) do
   zaehler=0;
   für jede Methode j der Klasse i do
      erzeugt Methode j ein Objekt einer Klasse, welche gleichzeitig
      Rückgabetyp von Methode j ist? ja: zaehler+1
   od
   ist zaehler>0? ja: weiter, nein: Abbruch;
   besitzt Klasse i eine Referenz auf eine andere Klasse j
   (Fliegengewicht)?
   ja: gibt es einen Parameter, den alle Methoden der Klasse j
   gemeinsam besitzen?
   ja: für alle Unterklassen k (konkrete Fliegengewichte) der Klasse j
   do
      wird Objekt von k in Klasse i (Fabrik) erzeugt? ja: Muster
      gefunden
   od
od
```

3.3.4.6 Kompositum

Das Kompositum-Muster besteht aus dem Kompositum und der Komponente. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<composite>
   <composite-class>Kompositum</composite-class>
   <component>Komponente</component>
</composite>
```

Algorithmus 3.11: FindeKompositum nach Naumann [Naumann01]

```
für jede Klasse i (Kompositum) do
   besitzt Klasse i eine 1-zu-n-Aggregation zu einer Oberklasse
   (Komponente)? ja: weiter
   besitzt Klasse i Unterklassen? nein: Muster gefunden
   ja: für jede Unterklasse j (spezialisiertes Kompositum) von Klasse
   i do
      für jede Methode k der Klasse j do
         ruft Methode k eine Methode der Klasse j auf, und folgt dem
         ein lokaler Methodenaufruf?
         nein: weiter, ja: Abbruch
      od
   od
   Muster gefunden
od
```

Der vorliegende Algorithmus 3.11 *FindeKompositum* sucht ausgehend vom Kompositum nach der Oberklasse Komponente und gegebenenfalls nach vorhandenen Unterklassen.

Die Kardinalität der Aggregationsbeziehung zwischen Kompositum-Klasse und Komponenten-Klasse lässt sich einfach abfragen. Die Menge der Unterklassen kann gebildet und auf ihre Größe untersucht werden. Ist die Menge leer, sind keine Unterklassen implementiert. Der Methodenaufruf einer anderen Klasse kann mittels einer Untersuchung des Quelltextes der Methoden gefunden werden. Ebenso kann ein folgender Aufruf einer lokalen Methode erkannt werden.

Der Test des Algorithmus mittels des Pattern-Projektes ergab eine Präzision und einen Recall von jeweils 100 Prozent. Diese Ergebnisse konnten allerdings durch den Test der anderen Systeme nicht bestätigt werden. So konnte weder im Drawlet- noch im AWT-Projekt ein Muster gefunden werden, obwohl bekanntlich in beiden Projekten mindestens eine Implementierung des Musters enthalten sein soll. Vermutlich liegt der Grund dafür in der schwierigen Aufgabe des Reverse-Engineering von Kardinalitäten von Aggregationsbeziehungen.

3.3.4.7 Proxy

Das Proxy-Muster besteht aus dem abstrakten und realen Subjekt und der Proxy-Klasse. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<proxy>
   <proxy-class>Proxy</proxy-class>
   <subject>Ziel</subject>
   <real-subject>RealesSubjekt</real-subject>
</proxy>
```

Algorithmus 3.12a: FindeProxy nach Naumann [Naumann01]

```
für jede Klasse i (Proxy) do
   ist Klasse i Unterklasse? ja: weiter, nein: Abbruch;
   gibt es Referenz auf andere Klasse k (echtes Subjekt)?
   ja: für alle öffentlichen Methoden j in Klasse i do
      gibt es die Methode j auch in der Klasse k? ja: weiter
      ruft die Methode j die gleichnamige Methode der Klasse k auf?
      ja: weiter
   od
   Muster gefunden
od
```

Der Algorithmus 3.12a *FindeProxy* testet ausgehend vom Proxy das Vorhandensein eines Subjektes und eines echten Subjektes. Alle öffentlichen Methoden der Proxy-Klasse werden daraufhin überprüft, ob sie den Methodenaufruf an eine gleichnamige Methode der Klasse des echten Subjektes weiterleiten. Ist dies der Fall, so ist das Muster eindeutig identifiziert.

Falls die Proxy-Klasse eine Unterklasse ist, so kann ein ausgehender Link mit der Eigenschaft Implementierung oder Generalisierung gefunden werden. Alle anderen ausgehenden Links können eine Referenz auf das echte Subjekt sein.

Der Test des Algorithmus mittels des Pattern-Projektes ergab einen Recall von 100 Prozent und eine Präzision von 33 Prozent. Die im Tomcat-Projekt enthaltenen Muster wurden erkannt.

Ähnlich dem Algorithmus zum Auffinden des Adapter-Musters ist auch hier zusätzlich sicherzustellen, dass das echte Subjekt und das Subjekt zwei verschiedene Klassen sind. Algorithmus 3.12b ist um diese Abfrage erweitert worden.

Algorithmus 3.12b: FindeProxy nach Naumann [Naumann01]

```
für jede Klasse i (Proxy) do
   ist Klasse i Unterklasse? ja: weiter, nein: Abbruch;
   gibt es Referenz auf andere Klasse k (echtes Subjekt)?
   ja: sind Klasse k und Klasse i nicht in einem Vererbungspfad?
   ja: für alle öffentlichen Methoden j in Klasse i do
      gibt es die Methode j auch in der Klasse k? ja: weiter
      ruft die Methode j die gleichnamige Methode der Klasse k auf?
      ja: weiter
   od
   Muster gefunden
od
```

3.3.5 Verhaltensmuster

3.3.5.1 Befehl

Das Befehl-Muster besteht aus einem abstrakten und mehreren möglichen konkreten Befehlen, Empfängern und einem Aufrufer. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<command>
   <command-class>Befehl</command-class>
   <concrete-commands>
      <concrete-command>KonkreterBefehl1</concrete-command>
      <concrete-command>KonkreterBefehl2</concrete-command>
   </concrete-commands>
   <recievers>
      <reciever>Empfaenger1</reciever>
      <reciever>Empfaenger2</reciever>
   </recievers>
   <caller>Aufrufer</caller>
</command>
```

Algorithmus 3.13: FindeBefehl nach Naumann [Naumann01]

```
für jede Klasse i (Befehl) do
   ist Klasse i abstrakt?
   ja: gibt es eine andere Klasse (Aufrufer), die Referenz auf Klasse
   i besitzt?
   ja: für alle Unterklassen k (konkreter Befehl) von Klasse i do
      besitzt Klasse k Referenz auf andere Klasse l (Empfänger)?
      ja: wird die Klasse l im Konstruktor der Klasse k übergeben?
      ja: für alle Methoden j der Klasse k do
         wird in Methode j Operation von Klasse l aufgerufen? ja:
         weiter
      od
      für alle Klassen n (Klient) do
         erzeugt Klasse n ein Objekt der Klasse k? ja: Muster gefunden
      od
   od
od
```

Ausgehend von der abstrakten Befehl-Klasse wird nach einem Aufrufer und konkreten Befehls-Klassen gesucht. Von den konkreten Befehls-Klassen aus werden die Referenzen auf Empfänger geprüft. Dazu muss der Konstruktor der Befehls-Klasse einen Parameter vom Typ der Empfänger-Klasse besitzen und eine Methode besitzen, welche eine Operation der Empfänger-Klasse aufruft.

Abstrakte Klassen werden in Java als Interfaces realisiert. Dies ist sehr einfach festzustellen. Um den Aufrufer zu identifizieren, werden alle Klassen getestet, ob sie eine Referenz auf die Befehl-Klasse verwalten. Ebenso lässt sich eine Referenz zwischen konkreter Befehls-Klasse und Empfänger feststellen.

Ob der Konstruktor der konkreten Befehls-Klasse einen Parameter vom Typ der Empfänger-Klasse entgegennimmt kann durch eine einfache Typüberprüfung der Parameter geprüft werden.

Die Weiterleitung des Befehls an den Empfänger wird durch Untersuchung des Quelltextes der Methode überprüft. Ist der fragliche Methodenaufruf enthalten, wird der Befehl weitergeleitet.

Um den Klienten zu finden, wird eine Klasse ermittelt, welche eine Referenz auf den konkreten Befehl verwaltet und in einer seiner Methoden ein Objekt dieser Klasse erzeugt.

Der Test des Algorithmus 3.13 *FindeBefehl* mittels des Patterns-Projektes ergab eine Präzision und einen Recall von jeweils 100 Prozent. Leider konnten diese Ergebnisse nicht weiter überprüft werden, da nicht sichergestellt war, dass dieses Muster in einem der anderen Projekte vorhanden ist.

3.3.5.2 Beobachter

Das Beobachter-Muster besteht aus einem abstrakten und mehreren möglichen konkreten Beobachtern und einem Subjekt. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<observer>
   <subject>Subjekt</subject>
   <observer-class>AbstrakterBeobachter</observer-class>
   <concrete-observers>
      <concrete-observer>KonkreterBeobachter1</concrete-observer>
      <concrete-observer>KonkreterBeobachter2</concrete-observer>
   </concrete-observers>
</observer>
```

Algorithmus 3.14: FindeBeobachter nach Naumann [Naumann01]

```
für jede Klasse i (Subjekt) do
   besitzt Klasse i 1-zu-n-Aggregation auf andere Klasse k
   (Beobachter)?
   ja: zaehler=0;
   für jede Methode j der Klasse i do
      besitzt Methode j als Parameter die Klasse k?
      ja: zaehler+1;
   od
   zaehler=2?
   ja: für alle Unterklassen l (konkrete Beobachter) von Klasse k do
      besitzt Klasse l Referenz auf Klasse i bzw. eine Unterklasse
      davon (konkretes Subjekt)?
      ja: Muster gefunden
   od
od
```

Ausgehend vom Subjekt sucht der Algorithmus 3.14 *FindeBeobachter* die abstrakte Beobachter-Klasse und ihre Unterklassen, die konkreten Beobachter. Des Weiteren muss die Subjekt-Klasse noch zwei Methoden deklarieren, welche einen Parameter vom Typ der abstrakten Beobachter-Klasse entgegennehmen.

Die Abfrage der Kardinalität der Referenz zwischen Subjekt- und Beobacher-Klasse gestaltet sich unproblematisch.

Der Typ der Parameter aller Methoden der Subjekt-Klasse kann bestimmt und anschließend mit dem Typ der Beobachter-Klasse verglichen werden.

Mit einem Recall und einer Präzision von jeweils 100 Prozent hat der Algorithmus im Test mittels des Pattern-Projektes ein sehr gutes Ergebnis erzielen können. Leider konnten die in allen anderen Projekten vorhandenen Muster durch diesen Algorithmus nicht identifiziert werden. Dies liegt in den Problemen des Reverse-Engineering von Kardinalitäten von Referenzen begründet.

3.3.5.3 Besucher

Das Besucher-Muster besteht aus einem Besucher und mehreren möglichen Elementen. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<visitor>
   <visitor-class>Besucher</visitor-class>
   <elements>
      <element>Element1</element>
      <element>Element2</element>
   </elements>
</visitor>
```

Algorithmus 3.15: FindeBesucher nach Naumann [Naumann01]

```
für jede Klasse i (Besucher) do
   für jede Methode j der Klasse i do
      besitzt Methode j andere Klasse k (Element) als Parameter?
      ja: für jede Methode m der Klasse k do
         besitzt Methode m als Parameter die Klasse i?
         ja: gibt es in dieser Methode einen Aufruf der Methode j der
         Klasse i?
         ja: übergibt sich bei diesem Aufruf die Klasse k selbst?
         ja: Muster gefunden
      od
   od
od
```

Der Algorithmus 3.15 *FindeBesucher* geht von der Besucher-Klasse aus und untersucht alle vorhandenen Methoden, um die Element-Klassen ausfindig zu machen. Haben diese jeweils eine Methode, welche als Parameter ein Objekt der Besucher-Klasse akzeptiert, wird innerhalb der Methode eine Methode der Besucher-Klasse aufgerufen und übergibt sich die Klasse selbst dabei als Parameter, so ist das Muster gefunden.

Der Typ der Parameter einer Methode kann bestimmt und mit dem einer konkreten Klasse verglichen werden. So können die spezifischen Merkmale für dieses Muster gefunden werden.

Der Test des Algorithmus mittels des Pattern-Projektes ergab eine Präzision von 100 Prozent und einen Recall von ebenfalls 100 Prozent. Leider konnten diese Ergebnisse nicht durch die anderen Projekte bestätigt oder widerlegt werden, da nicht bekannt war, ob Implementierungen des Besucher-Musters vorhanden sind.

3.3.5.4 Interpreter

Das Interpreter-Muster besteht aus einem abstrakten Ausdruck. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<interpreter>
   <interpreter-class>AbstrakterAusdruck</interpreter-class>
</interpreter>
```

Algorithmus 3.16: FindeInterpreter nach Naumann [Naumann01]

```
erstelle eine Menge der enthaltenen Bäume;
für jeden Baum i do
   ist Wurzelklasse abstrakt?
   für alle Methoden j der Wurzelklasse do
      wird Methode j von allen Unterklassen überschrieben?
      nein: Abbruch
      bestimme Anzahl der Referenzen von den Unterklassen zur
      Wurzelklasse
      bestimme Anzahl der Unterklassen
      ist Verhältnis Referenzen / Unterklassen größer gleich 50%?
      ja: für jede Unterklasse k do
         besitzt Klasse k Referenz auf andere Unterklasse?
         ja: Abbruch
      od
      Muster gefunden
   od
od
```

Der Algorithmus 3.16 *FindeInterpreter* geht von der abstrakten Wurzelklasse aus und sucht alle Unterklassen, welche alle abstrakten Methoden implementieren. Wenn mindestens 50 Prozent dieser Unterklassen eine Referenz auf die Wurzelklasse verwalten und keine Referenzen zwischen den Unterklassen vorhanden sind, so ist das Muster gefunden.

Die Menge aller Bäume muss nicht konkret erstellt werden. Es ist ausreichend, die Menge aller Wurzelklassen aufzustellen, da sich der Algorithmus an dieser orientiert. Wurzelklassen sind Klassen, welche keine Oberklassen besitzen.

Um zu testen, ob alle Unterklassen eine bestimmte Methode überschreiben wird in diesen nach derselben Deklaration gesucht.

Der Algorithmus erzielte im Test mittels des Pattern-Projektes einen Recall sowie eine Präzision von jeweils 100 Prozent. Leider konnten diese Ergebnisse nicht weiter bewertet werden, da nicht bekannt ist, ob Implementierungen des Interpreter-Musters in den anderen Projekten vorhanden sind.

3.3.5.5 Iterator

Das Iterator-Muster besteht aus einem abstrakten und konkreten Iteratoren sowie aus einem abstrakten und konkreten Aggregaten. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<iterator>
   <iterator-class>KonkreterIterator</iterator-class>
   <aggregate>KonkretesAggregat</aggregate>
</iterator>
```

Algorithmus 3.17: FindeIterator nach Naumann [Naumann01]

```
für alle Templates i (Iterator) do
   besitzt Template i Aggregation auf anderes Template j (Liste)?
   ja: erzeugt Template j in einer Methode Template i?
   ja: Muster gefunden
od
```

Der Algorithmus 3.17 *FindeIterator* sucht ausgehend von der Iterator-Klasse nach einer zugehörigen Liste-Klasse, welche eine Methode zum erzeugen eines Iteratorobjektes enthalten muss.

Aggregationen zwischen zwei Klassen sind spezielle Referenzen. Sie können trivial festgestellt werden, wobei sie den Typ Aggregation besitzen müssen.

Ob ein Template ein Objekt des anderen erzeugt, kann durch eine Analyse des Quelltextes aller Methoden bestimmt werden. Ist das Schlüsselwort `new` vorhanden, so wird der Typ des erzeugten Objektes bestimmt und mit dem des fraglichen Templates verglichen.

Trotz der Präzision und des Recall von jeweils 100 Prozent, welche im Test mit dem Pattern-Projektes festgestellt werden konnten, wurden die im Drawlet-Projekt vorhandenen Muster durch den Algorithmus nicht identifiziert.

3.3.5.6 Memento

Das Memento-Muster besteht aus einem Urheber, einem Aufbewahrer und einem Memento. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<memento>
   <memento-class>Memento</memento-class>
   <creator>Urheber</creator>
   <depositor>Aufbewahrer</depositor>
</memento>
```

Algorithmus 3.18: FindeMemento nach Naumann [Naumann01]

```
für jede Klasse i (Memento) do
   ist kein öffentlicher Konstruktor vorhanden?
   ja: existiert ein Attribut, das in einer privaten Methode als
   Parameter übergeben wird?
   ja: existiert eine andere private Methode, die dieses Attribut
   zurückliefert?
   ja: ist eine andere Klasse j (Urheber) friend von Klasse i?
   ja: kann Klasse j Objekte der Klasse i erzeugen?
   ja: besitzt Klasse j Referenz auf Klasse i?
   ja: Abbruch, nein: weiter
   für jede Klasse k (Aufbewahrer) do
      besitzt Klasse k Referenz auf Klasse i?
      ja: erzeugt Klasse k Klasse i?
      nein: Muster gefunden
   od
od
```

Ausgehend von der Memento-Klasse sucht der Algorithmus 3.18 *FindeMemento* nach dem Urheber und dem Aufbewahrer. Sind alle drei Klassen mit ihren vorgegebenen Eigenschaften gefunden, so ist das Muster identifiziert.

Der Ausschluss eines öffentlichen Konstruktors ist durch das Nichtvorhandensein einer öffentlichen Methode mit dem Namen der Klasse sichergestellt.

Die Namen aller Attribute können festgestellt und einfach mit den Parametern der Methoden verglichen werden. Schwierig wird es, wenn die Namen nicht einheitlich vergeben wurden. Mit Hilfe des Typs kann nicht die Semantik der Variablen festgestellt werden. Ähnlich verhält es sich mit der Überprüfung, ob eine Methode existiert, welche das Attribut als Rückgabewert hat.

Die Frage der Objekterzeugung kann wiederum durch die Suche nach dem Schlüsselwort `new` entschieden werden, worauf ein Vergleich der Typen folgt.

Eine friend-Beziehung zwischen zwei Klassen besteht genau dann, wenn sich beide Klassen in ein und demselben Paket befinden.

Im Test mittels des Pattern-Projektes erzielte der Algorithmus einen Recall sowie eine Präzision von jeweils 100 Prozent. Allerdings konnten im Drawlet-Projekt vorhandene Muster durch diesen Algorithmus nicht gefunden werden.

3.3.5.7 Schablonenmethode

Das Schablonenmethode-Muster besteht aus nur einer Klasse mit den dazugehörigen Schablonenmethoden. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<template-method>
   <containing-class>AbstrakteKlasse</containing-class>
   <method-names>
      <method-name>schablonenMethode1</method-name>
      <method-name>schablonenMethode2</method-name>
   </method-names>
</template-method>
```

Algorithmus 3.19: FindeSchablonenmethode nach Naumann [Naumann01]

```
für jede Klasse i do
   für jede nicht polymorphe Methode j (Schablonenmethode) do
      für jeden Aufruf einer Methode l (primitive Operation) do
         ist Methode l lokal?
         ja: ist Methode l polymorph?
         ja: Muster gefunden
      od
   od
od
```

Zur Suche nach einer Schablonenmethode wird jeweils eine einzelne Klasse untersucht. Falls diese eine nicht polymorphe Methode implementiert, welche lokale polymorphe Methoden in ihrem Quelltext aufruft, so wurde eine Schablonenmethode gefunden.

Grundsätzlich sind in Java alle Methoden polymorph. Um diese Eigenschaft nicht zu haben, werden diese Methoden als final deklariert. Das wiederum lässt sich einfach testen. Lokale Methoden sind Methoden, deren Implementierung sich in der Klasse befindet, wo diese auch aufgerufen wird.

Der Test des Algorithmus 3.19 *FindeSchablonenmethode* ergab eine Präzision und einen Recall von jeweils 100 Prozent. Diese Ergebnisse konnten bestätigt werden, da die in zwei der drei weiteren Testprojekte vorhandenen Muster gefunden worden sind.

3.3.5.8 Strategie

Das Strategie-Muster besteht aus einer abstrakten und einer oder mehreren konkreten Strategien sowie einem Kontext. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<strategy>
   <abstract-strategy>AbstrakteStrategie</abstract-strategy>
   <context>Kontext</context>
   <concrete-strategies>
      <concrete-strategy>KonkreteStrategie1</concrete-strategy>
      <concrete-strategy>KonkreteStrategie2</concrete-strategy>
   </concrete-strategies>
</strategy>
```

Algorithmus 3.20: FindeStrategie nach Naumann [Naumann01]

```
erstelle eine Menge der enthaltenen Bäume;
für jeden Baum i do
   ist Wurzelklasse (abstrakte Strategie) abstrakt?
   ja: für alle Unterklassen j (konkrete Strategien) do
      besitzt Klasse j dieselbe öffentliche Schnittstelle wie die
      Wurzelklasse?
      ja: besitzt Klasse j eine Referenz auf die Wurzelklasse?
      nein: besitzt Klasse j eine Referenz auf eine andere
      Unterklasse?
      nein: für alle Klassen k (Kontext) do
         besitzt Klasse k eine Referenz auf die Wurzelklasse des
         Baumes i?
         ja: besitzt Klasse k weitere Referenzen auf die Unterklassen
         des Baumes i?
         nein: Muster gefunden
      od
   od
od
```

Die Suche nach dem Strategie-Muster geht von der abstrakten Strategie-Klasse aus und untersucht alle Unterklassen, ob diese dieselbe öffentliche Schnittstelle implementieren wie sie selbst. Ist dies der Fall, so muss noch eine Kontext-Klasse gefunden werden, welche eine Referenz auf die abstrakte Strategie-Klasse und keine zu einer ihrer Unterklassen besitzt.

Abstrakte Klassen werden in Java durch Interfaces realisiert. Ist die Wurzelklasse ein Interface, so ist sie abstrakt.

Die öffentliche Schnittstelle einer Klasse wird durch alle öffentlichen Methoden und Attribute charakterisiert. Zwei Klassen haben also genau dann die gleiche öffentliche Schnittstelle, wenn sie dieselben öffentlichen Methoden und Attribute besitzen.

Der im Test mittels des Pattern-Projektes erreichte Recall von 100 Prozent steht einer Präzision von lediglich 6 Prozent gegenüber. Der vorliegende Algorithmus 3.20 *FindeStrategie* lässt sehr viel Spielraum, was leicht zu Fehlerkennungen führen kann. Im Test der anderen Projekte konnten die vorhandenen Muster identifiziert werden.

3.3.5.9 Vermittler

Das Vermittler-Muster besteht aus einem abstrakten und einem konkreten Vermittler sowie aus abstrakten und konkreten Kollegen. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<mediator>
   <concrete-mediator>KonkreterVermittler</concrete-mediator>
   <concrete-colleagues>
      <concrete-colleague>KonkreterKollege1</concrete-colleague>
      <concrete-colleague>KonkreterKollege2</concrete-colleague>
   </concrete-colleagues>
</mediator>
```

Algorithmus 3.21: FindeVermittler nach Naumann [Naumann01]

```
für jede Klasse i (konkreter Vermittler) do
   bilde aus den Referenzen der Klasse i (konkrete Kollegen) eine
   Menge A;
   für jede Klasse j (konkreter Kollege) der Menge A do
      für jede Referenz k der Klasse j do
         weist Referenz k auf eine Klasse der Menge A?
         ja: Abbruch
      od
   od
   für jede Klasse j der Menge A do
      Sprungmarke:
      besitzt Klasse j Referenz auf Klasse i? ja: Muster gefunden
      nein: besitzt Klasse j Referenz auf Oberklasse (abstrakter
      Vermittler)von Klasse i? ja: Muster gefunden
      nein: dann gehe zu Sprungmarke und führe dieselben Abfragen mit
      der Oberklasse (abstrakter Kollege) der Klasse j durch, solange,
      bis Wurzelklasse erreicht
   od
od
```

Ausgehend vom konkreten Vermittler wird die Menge aller konkreten Kollegen gebildet. Daraufhin werden die Klassen der Menge auf Referenzen untereinander getestet. Zum Abschluss wird geprüft, ob die konkreten Kollegen auch eine Referenz auf den konkreten Vermittler verwalten. Ist dies der Fall, so ist das Muster identifiziert.

Der komplette Algorithmus prüft lediglich Referenzen zwischen Klassen, welche sich sehr einfach durch die Untersuchung aller ausgehenden Links einer Klasse analysieren lassen.

Der im Test mittels des Pattern-Projektes ermittelte Recall für den vorliegenden Algorithmus 3.21 *FindeVermittler* beträgt 100 Prozent. Die Präzision liegt dagegen allerdings nur bei 4 Prozent. Die in den Projekten Drawlet und AWT vorhandenen Muster wurden ebenfalls erkannt.

3.3.5.10 Zustand

Das Zustand-Muster besteht aus einem abstrakten und einem oder mehreren konkreten Zuständen. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<state>
   <abstract-state>AbstrakterZustand</abstract-state>
   <concrete-states>
      <concrete-state>KonkreterZustand1</concrete-state>
      <concrete-state>KonkreterZustand2</concrete-state>
   </concrete-states>
</state>
```

Algorithmus 3.22: FindeZustand nach Naumann [Naumann01]

```
erstelle eine Menge der enthaltenen Bäume;
für jeden Baum i do
   ist Wurzelklasse (Zustand) konkret?
   ja: für alle Unterklassen j (konkrete Zustände) do
      besitzt Klasse j dieselbe öffentliche Schnittstelle wie die
      Wurzelklasse?
      ja: besitzt Klasse j eine Referenz auf die Wurzelklasse?
      nein: besitzt Klasse j eine Referenz auf eine andere
      Unterklasse?
      nein: für alle Klassen k (Kontext) do
         besitzt Klasse k eine Referenz auf die Wurzelklasse des
         Baumes i?
         ja: besitzt Klasse k weitere Referenzen auf die Unterklassen
         des Baumes i?
         nein: Muster gefunden
      od
   od
od
```

Ausgehend von der konkreten Zustand-Klasse wird in der Menge aller Unterklassen nach konkreten Zuständen mit derselben öffentlichen Schnittstelle gesucht, welche keine Referenzen untereinander und zur Zustand-Klasse haben dürfen. Daraufhin wird eine Kontext-Klasse gesucht, welche eine Referenz auf die Zustand-Klasse verwaltet, jedoch keine auf einen konkreten Zustand.

Um festzustellen, ob eine Klasse konkret ist, muss lediglich ausgeschlossen werden, dass es sich dabei um eine Schnittstelle handelt.

Die öffentliche Schnittstelle einer Klasse wird durch alle öffentlichen Methoden und Attribute charakterisiert. Zwei Klassen haben also genau dann die gleiche öffentliche Schnittstelle, wenn sie dieselben öffentlichen Methoden und Attribute besitzen.

Der Test des Algorithmus 3.22 *FindeZustand* mittels des Pattern-Projektes ergab einen Recall und eine Präzision von jeweils 100 Prozent. Da keine Kenntnis über weitere

Implementierungen innerhalb der anderen Referenzprojekte bestand, konnten diese Ergebnisse weder bestätigt noch widerlegt werden.

3.3.5.11 Zuständigkeitskette

Das Zuständigkeitskette-Muster besteht aus einer Wurzelklasse und der durch ihre Unterklassen überschriebenen Methode. Zur Speicherung dieser Daten wurde folgende XML-Struktur entworfen:

```
<chain-of-resp>
   <root-class>Wurzelklasse</root-class>
   <overridden-method>ueberschriebeneMethode</overridden-method>
</chain-of-resp>
```

Algorithmus 3.23: FindeZuständigkeitskette nach Naumann [Naumann01]

```
erstelle eine Menge der enthaltenen Bäume;
für jeden Baum i do
   bestimme, welche Methode j (BearbeiteAnfrage) der Wurzelklasse am
   häufigsten von allen Unterklassen überschrieben wird
   zaehler_weiterleiten=0;
   für alle Klassen im Baum, die Methode j überschreiben, do
      wenn Methode j den Aufruf einer Methode besitzt, die denselben
      Namen wie Methode j trägt, dann zaehler_weiterleiten+1
   od
   ist (zaehler_weiterleiten / Anzahl Klassen, die Methode j
   überschreiben) mindestens 75%?
   ja: Muster gefunden
od
```

Der Algorithmus 3.23 *FindeZuständigkeitskette* untersucht alle Klassen innerhalb eines Vererbungsbaumes darauf, ob sie eine bestimmte Methode implementieren und ob innerhalb der Methode eine Weiterleitung des Aufrufs erfolgt. Die Klassen, welche die Methode implementieren, bilden eine Menge A. Die Klassen der Menge A, welche innerhalb der Methode den Aufruf weiterleiten, bilden eine Untermenge B. Macht die Menge B mindestens 75 Prozent der Menge A aus, so ist das Muster gefunden.

Bei der Umsetzung des Algorithmus wurde auf die Feststellung der Methode, welche am häufigsten aufgerufen wird, verzichtet und stattdessen wurden alle Methoden untersucht.

Im Test mittels des Patterns-Projektes konnte ein Recall von 100 Prozent und eine Präzision von 33 Prozent für diesen Algorithmus festgestellt werden. Die im Tomcat-Projekt vorhandenen Muster konnten ebenfalls erfasst werden.

3.3.6 Untersuchung der Laufzeiten der Algorithmen

Um die Laufzeit eines Algorithmus zu bestimmen, wurde mit Hilfe der Methode `getCurrentTimeMillis()` der Klasse `System` die Rechnerzeit in Millisekunden vor dem Start und nach der Beendigung des Algorithmus gelesen. Die Laufzeit ist die sich aus diesen Messungen ergebene Differenz, wobei die Zeit für die Dokumentation der Ergebnisse nicht mit eingeht. Die Messungen wurden auf einem Rechner mit einem AMD Athlon 1800XP Prozessor mit 384MB RAM unter dem Betriebssystem XP vorgenommen.

Die folgende Tabelle 7 gibt die von Naumann geschätzte Laufzeit der Algorithmen und im Vergleich dazu die gemessene Laufzeit an. Der Parameter n beziffert dabei die Anzahl der Klassen innerhalb des Projektes und gibt einen Eindruck von dessen Umfang.

Tabelle 7: Testergebnisse für die Laufzeit

Algorithmus	Laufzeit	Patterns n = 88	Drawlet n = 195	AWT n = 354	Tomcat n = 1035
Abstrakte Fabrik	O(n)	631 ms	172728 ms	1039965 ms	5098130 ms
Erbauer	O(n)	381 ms	17055 ms	88067 ms	849272 ms
Fabrikmethode	O(n)	541 ms	174861 ms	1001760 ms	4955756 ms
Prototyp	O(n)	10 ms	10 ms	90 ms	540 ms
Singleton	O(n)	10 ms	70 ms	140 ms	471 ms
Adapter	O(n)	520 ms	117229 ms	577801 ms	4769608 ms
Brücke	$O(n^3)$	1563 ms	188912 ms	1277016 ms	10730680 ms
Dekorierer	$O(n^2)$	40 ms	80 ms	300 ms	1052 ms
Fassade	$O(n^3)$	991 ms	16223 ms	102808 ms	843383 ms
Fliegengewicht	$O(n^2)$	50 ms	230 ms	721 ms	12017 ms
Kompositum	O(n)	20 ms	70 ms	311 ms	831 ms
Proxy	O(n)	10 ms	351 ms	2854 ms	5839 ms
Befehl	$O(n^2)$	862 ms	47018 ms	172598 ms	1285859 ms
Beobachter	$O(n^2)$	140 ms	90 ms	300 ms	841 ms
Besucher	O(n)	30 ms	3074 ms	23074 ms	119261 ms
Interpreter	$O(n^2)$	981 ms	61639 ms	769486 ms	4699627 ms
Iterator	O(n)	10 ms	80 ms	291 ms	861 ms
Memento	O(n)	20 ms	60 ms	1922 ms	7351 ms
Schablonenmethode	O(n)	10 ms	40 ms	110 ms	291 ms
Strategie	$O(n^2)$	1172 ms	12768 ms	83160 ms	749838 ms
Vermittler	$O(n^2)$	120 ms	2673 ms	3765 ms	16424 ms
Zustand	$O(n^2)$	481 ms	3666 ms	18617 ms	449476 ms
Zuständigkeitskette	$O(n^2)$	1362 ms	125510 ms	869761 ms	7644573 ms

3.4 Zusammenfassung der Ergebnisse

In diesem Kaptitel wurden die von Naumann entwickelten Algorithmen analysiert und anhand einiger Projekte auf ihre Leistungsfähigkeit untersucht. Die Ergebnisse der Tests wurden in den Tabellen 3 bis 6 dokumentiert und anschließend für jeden Algorithmus im Einzelnen ausgewertet.

Die folgende Tabelle 8 vergleicht Naumanns Vorhersagen bezüglich der Schwierigkeit des Auffindens der Muster mit den von den Algorithmen erzielten Testresultaten.

Tabelle 8: Testergebnisse im Vergleich mit Naumanns Vorhersagen [Naumann01]

Algorithmus	Schwierigkeit	Recall	Präzision	Verbesserung
Abstrakte Fabrik	leicht	100%	100%	nicht notwendig
Erbauer	mittel	100%	8%	vorgenommen
Fabrikmethode	leicht	100%	100%	nicht notwendig
Prototyp	leicht	100%	100%	nicht notwendig
Singleton	leicht	100%	100%	vorgenommen
Objektadapter	leicht	100%	33%	keine Vorschläge
Brücke	schwer	100%	4%	vorgenommen
Dekorierer	leicht	100%	100%	nicht notwendig
Fassade	schwer	100%	1%	vorgenommen
Fliegengewicht	mittel	100%	100%	vorgenommen
Kompositum	leicht	100%	100%	nicht notwendig
Proxy	leicht	100%	33%	vorgenommen
Befehl	mittel	100%	100%	nicht notwendig
Beobachter	leicht	100%	100%	nicht notwendig
Besucher	leicht	100%	100%	nicht notwendig
Interpreter	schwer	100%	100%	nicht notwendig
Iterator	leicht	100%	100%	nicht notwendig
Memento	leicht	100%	100%	nicht notwendig
Schablonenmethode	leicht	100%	100%	nicht notwendig
Strategie	mittel	100%	6%	keine Vorschläge
Vermittler	mittel	100%	4%	keine Vorschläge
Zustand	mittel	100%	100%	nicht notwendig
Zuständigkeitskette	mittel	100%	33%	keine Vorschläge

Wie der Tabelle 8 zu entnehmen ist, sind die meisten der von Naumann getroffenen Voraussagen zutreffend. Ausnahmen sind hier die Algorithmen für die Muster Erbauer, Strategie und Vermittler, welche mit einer Schwierigkeitseinschätzung *mittel* in der Präzision jedoch nur einstellige Prozentsätze aufweisen. Die Algorithmen für die Muster Objektadapter und Proxy wurden sogar als *leicht* eingeschätzt, erreichten allerdings lediglich eine Präzision von 33 Prozent, ebenso wie der als *mittel* eingeschätzte Algorithmus des Musters der Zuständigkeitskette. Von den drei als *schwer* eingeschätzten

Algorithmen konnte der für das Interpreter-Muster jedoch eine Präzision von 100 Prozent nachweisen.

Da die Ergebnisse für einige Algorithmen nicht zufrieden stellend sind, wurden mögliche Ursachen dafür aufgedeckt und einige Verbesserungsmöglichkeiten aufgezeigt. Dies betrifft im Speziellen die Algorithmen für das Erbauer-Muster, das Prototyp-Muster, das Brücke-Muster, das Fassade-Muster, das Fliegengewicht-Muster und das Proxy-Muster. Umsetzung und Test dieser Vorschläge werden künftigen Forschungsarbeiten überlassen.

Im folgenden Kapitel 4 wird etwas näher auf die prototypische Umsetzung des von Naumann in [Naumann01] entwickelten Pseudocodes eingegangen.

4 Prototypische Umsetzung

Dieses Kapitel beschreibt die Implementierung der von Naumann entwickelten Algorithmen auf Grundlage ihrer Pseudocodenotation. Im Rahmen dieser entstand ein neues Modul, welches in die Entwicklungsumgebung Together ControlCenter von Borland integriert wurde. Da Together außerdem das Rerverse-Engineering von Quelltext anbietet, ist kein weiteres Werkzeug notwendig. Abbildung 5 zeigt die notwendigen Schritte zur Analyse des Quelltextes.

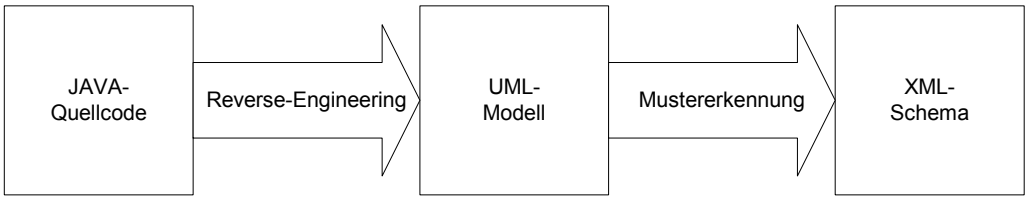

Abbildung 5: Schritte zur Analyse des Quelltextes

Die Beschränkung auf Java-Quelltext wurde vorgenommen, da zur Mustererkennung die Untersuchung des UML-Klassenmodells nicht ausreichend ist und zusätzlich auf den Quelltext zugegriffen werden muss. Im Fall von Java ist dieser auch nach dem Reverse-Engineering noch verfügbar. Wird dagegen C++ Quelltext zum Reverse-Engineering verwendet, so werden lediglich die Header-Dateien eingelesen und verarbeitet. Die konkreten Implementierungen der Klassen stehen danach nicht zur Verfügung. Um genau dieses Problem zu beheben, wird die Entwicklung einer Mustersuch-API angestrebt.

4.1 Das Nutzerinterface

Wie die Ergebnisse in Kapitel 3.2.4 *Untersuchung der Laufzeiten der Algorithmen* zeigen, kann es bei umfangreichen Projekten für einzelne Algorithmen zu Laufzeiten über einer Stunde kommen. Um dem Nutzer unnötige Wartezeiten zu ersparen, wurde eine Schnittstelle zur individuellen Auswahl der auszuführenden Algorithmen entworfen. Diese ist in Abbildung 6 dargestellt. Der Nutzer hat die Möglichkeit, eine ganze Gruppe von Algorithmen zu wählen oder jeden gewünschten Algorithmus im Einzelnen.

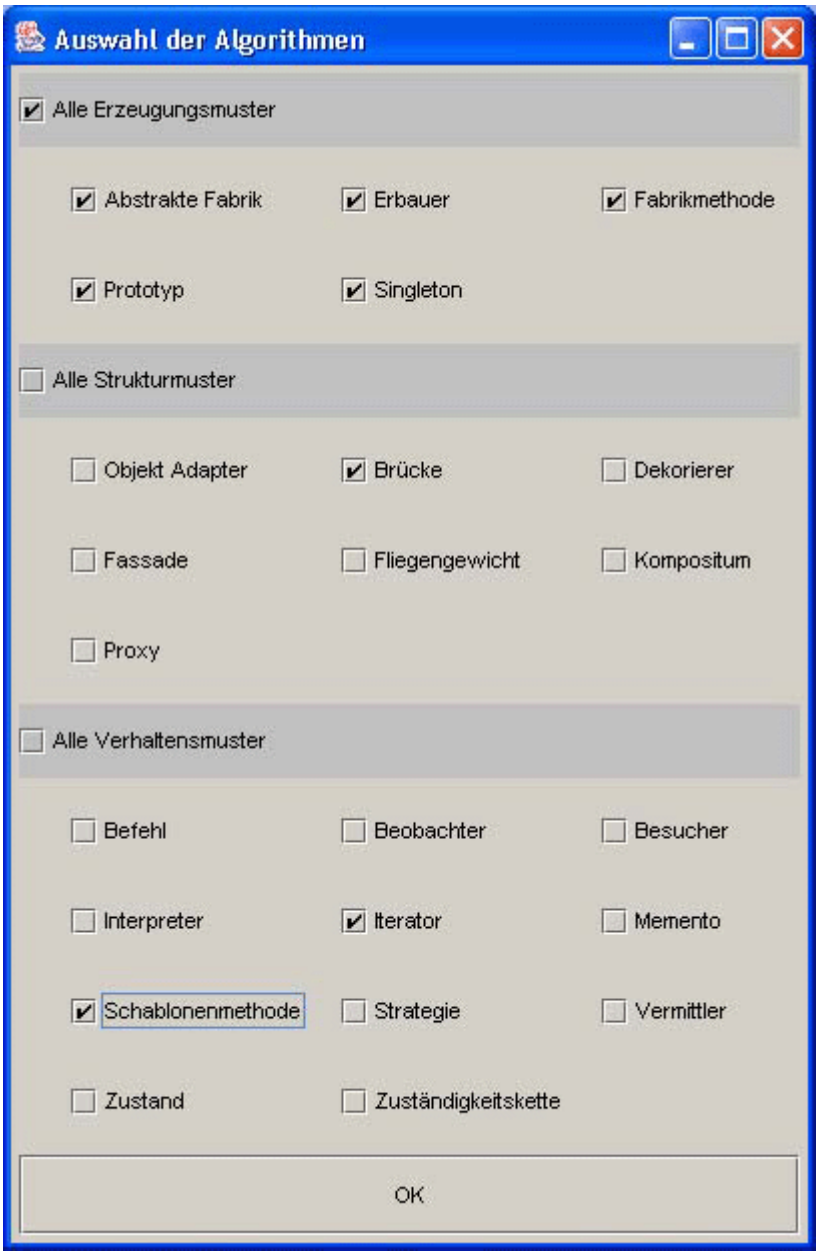

Abbildung 6: Auswahl der Algorithmen

4.2 Die Klassenstruktur des Moduls

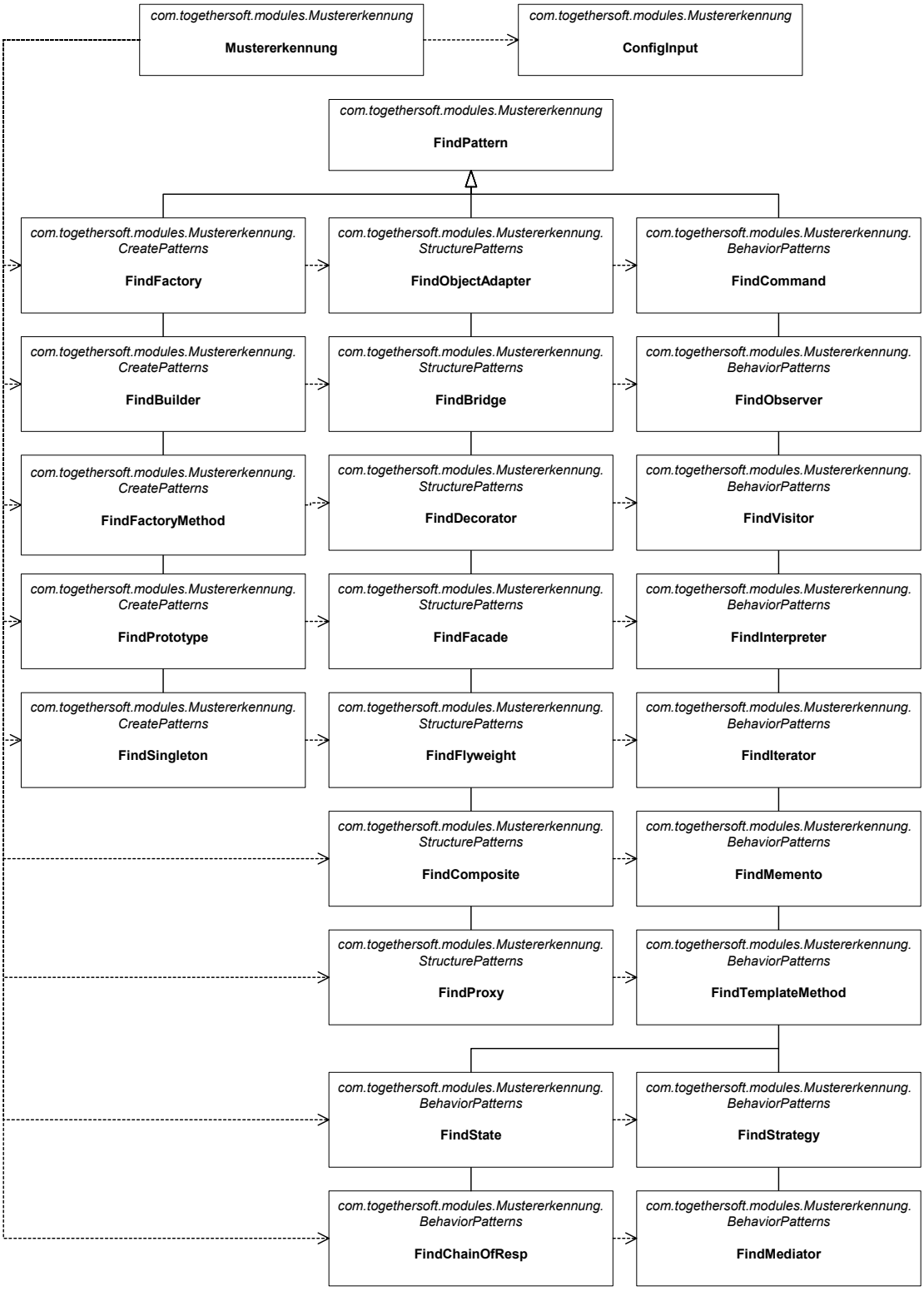

Abbildung 7: Klassendiagramm Mustererkennung

Abbildung 7 zeigt einen Ausschnitt aus der Klassenstruktur des Moduls zur Mustersuche. Die Klasse Mustererkennung ist die zentrale Klasse, die auf alle anderen Klassen Zugriff

hat. Des Weiteren entstand für jeden Algorithmus eine separate Klasse, welche eine find()-Methode für die Suche und eine print()-Methode für die Ausgabe der Ergebnisse implementiert. Alle Klassen sind von der Superklasse FindPattern abgeleitet, welche allgemeine, mehrfach benötigte Funktionen implementiert, wie beispielsweise die Ermittlung aller Superklassen einer Klasse. Die Klasse ConfigInput ist für die Darstellung der bereits erwähnten Nutzerschnittstelle zuständig. Des Weiteren wurden für nahezu jedes Muster eine JavaBean Klasse entwickelt, welche eine Instanz des Musters mit allen ihren Komponenten speichert. Ausnahmen bilden hierbei die Muster Singleton, Fassade und Interpreter, welche nur aus einer Komponente bestehen und daher kein eigenes JavaBean benötigen. JavaBeans sind auf Java basierende wieder verwendbare Software-Komponenten.

4.3 Dokumentation der Ergebnisse

Zur Ausgabe der Ergebnisse der Algorithmen wurde ein XML-Schema entwickelt, welches im vorhergehenden Kapitel bereits erwähnt wurde. Die Ergebnisse werden in einer Datei zusammengefasst und können so besser weiterverarbeitet werden. Diese Datei setzt sich aus den bereits vorgestellten Strukturen der einzelnen Muster und einigen weiteren Elementen zusammen. Im Folgenden ist der Aufbau dieser Datei dargestellt:

```
<project name="patterns.tpr" number-of-classes="88">
   <abstract-factories number-of-patterns="1" duration-in-ms="631">
      <abstract-factory>
      …
      </abstract-factory>
      …
   </abstract-factories>
   …
   <chain-of-resps number-of-patterns="1" duration-in-ms="631">
      <chain-of-resp>
      …
      </chain-of-resp>
      …
   </chain-of-resps>
</project>
```

Im ersten Element project werden der Name des bearbeiteten Projektes und die Anzahl der beinhalteten Klassen angegeben. Die Musterbezogenen Elemente kapseln alle gefundenen Instanzen eines Musters und geben zusätzlich die Laufzeit des Algorithmus sowie die Anzahl der gefundenen Muster an. Die XML-Datei Patterns.xml mit den konkreten Ergebnissen des Tests des Patterns-Projektes ist im Anhang B zu finden.

4.4 Weiterführende Entwicklungen

In Folgenden wird eine Idee zur Visualisierung der Ergebnisse vorgestellt. Die Muster können aus der XML-Datei gelesen werden und in einem in Together integrierten Fenster in einer Baumstruktur angezeigt werden. Es sollte dem Nutzer möglich sein, innerhalb dieser Struktur zu navigieren und durch Selektion eines bestimmten Musters die beteiligten Klassen im Klassendiagramm angezeigt und eingefärbt zu bekommen. Einige Ansätze zum Einfärben von Klassen wurden bereits von Jens Herbig unternommen und in [Herbig03] näher erläutert.

Die entstandene Implementierung zur Mustererkennung ist eine sehr strenge Umsetzung der Algorithmen aus der Diplomarbeit von Sebastian Naumann [Naumann01]. Weiterführende Entwicklungen könnten einige sinnvolle Erweiterungen beinhalten:

Es wäre denkbar, dass der Nutzer eine Möglichkeit bekommt zu entscheiden, wie strikt sich das aufzufindende Muster an die Merkmale aus [Gamma+95] halten muss. Dies könnte durch eine Auswahl der zwingend vorhandenen Komponenten des Musters geschehen oder es werden verschiedene Abstufungen der Übereinstimmungen zur Auswahl angeboten.

Ein weiterer Punkt ist die Einbeziehung verschiedener Ausprägungen der Implementierung eines Musters innerhalb des Algorithmus. Auch hier ist eine vorherige Auswahl durch den Nutzer denkbar.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassende und abschließende Bemerkungen

Thema dieser Diplomarbeit ist die Analyse von Algorithmen zur automatisierten Entwurfsmustererkennung. Im einführenden Kapitel wurde auf die Bedeutung von Entwurfsmustern in der Softwareentwicklung hingewiesen und auf die Vorzüge der automatisierten Erkennung dieser. Allerdings sollte die Leistungsfähigkeit solcher Algorithmen zuvor eingängig geprüft werden.

Im zweiten Kapitel wurden die Kennwerte Präzision und Recall besprochen und einige verschiedene Ansätze zur automatisierten Mustererkennung kurz vorgestellt. Im Verlauf der Betrachtungen wurde der Schwerpunkt speziell auf den Test der teilweise entstandenen Werkzeuge gelegt. Das Resultat dieser Betrachtungen war allerdings nicht zufrieden stellend. Einige Werkzeuge wurden gar nicht getestet oder es wurden keine Angaben zum Test gemacht. Nur einige wenige Autoren konnten Angaben zu Präzision und Recall machen. Die zum Test verwendeten Referenzsysteme waren in erster Linie in C++ implementiert, weshalb sie nicht für die Tests der Algorithmen von Naumann geeignet sind. Daher wurde nach anderen Systemen gesucht und eine Auswahl zusammengestellt, welche durch ein weiteres selbst implementiertes System ergänzt wurde. Dazu zählen der Applikationsserver Tomcat sowie das Abstract Windowing Toolkit (AWT). Im zweiten Teil wurden einige Probleme bei der Mustererkennung erläutert und ein spezielles Mustersuch-API als Lösung dafür vorgeschlagen.

Kapitel drei widmet sich der Analyse der Algorithmen. Nachdem die verwendeten Systeme nochmals detaillierter vorgestellt worden sind, beschrieb der anschließende Teil jeden Algorithmus im Einzelnen mit seinen Stärken und Schwächen, welche darauf mit den Testergebnissen verglichen wurden. In einigen Fällen konnten konkrete Vorschläge zur Verbesserung der Algorithmen gemacht werden. Aus Zeit- und Aufwandsgründen wird der Test dieser zukünftigen Forschungsarbeiten überlassen. Die während der Tests erfassten Laufzeiten der Algorithmen sind anschließend dokumentiert und den von Naumann getroffenen Voraussagen gegenübergestellt worden. Eine Optimierung der Laufzeiten der Algorithmen war allerdings nicht Bestandteil der hier angestrebten Verbesserungen. Abschließend sind die resultierenden Testergebnisse nochmals zusammenfassend dargestellt und mit Naumanns Einschätzungen zur Schwierigkeit der Algorithmen

verglichen. Dabei haben sich einige Abweichungen feststellen lassen, wie beispielsweise bei den Mustern Erbauer, Strategie und Vermittler, welche zu optimistisch bewertet worden sind. Im Gegensatz dazu konnte das als schwer eingeordnete Interpreter-Muster eine Präzision von 100 Prozent erlangen.

Das letzte Kapitel dokumentiert die prototypische Umsetzung der Algorithmen. Dabei wurden die notwendigen Schritte zur Mustererkennung erläutert, das entwickelte Nutzerinterface vorgestellt und eine Übersicht über die implementierten Klassen durch ein UML-Klassendiagramm gegeben. Im Anschluss daran folgte die Beschreibung des resultierenden XML-Schemas, welches zur Speicherung der Ergebnisse der Algorithmen dient.

Abschließend ist zu bemerken, dass die hier entwickelten und getesteten Algorithmen die Ergebnisse aller vorgestellten Ansätze, soweit Angaben dazu gemacht wurden, übertreffen. Der Recall beträgt für alle Muster 100 Prozent und die durchschnittliche Präzision lässt sich auf 68 Prozent bestimmen. Durch die vorgeschlagenen Verbesserungen wird die Präzision noch angehoben. Des Weiteren werden durch diesen Ansatz alle 23 Muster aus [Gamma+96] abgedeckt, was nur bei wenigen anderen der Fall ist.

5.2 Ausblick

Das für die Tests entwickelte Referenzsystem ist leider nur eine rudimentäre Implementierung der Muster. Für zukünftige Untersuchungen wird daher empfohlen, eine Sammlung zu erstellen, welche einen größeren Reichtum an Implementierungsvarianten der verschiedenen Muster beinhaltet. Da dies mit viel Erfahrung und großem Aufwand verbunden ist, konnte es im Rahmen dieser Arbeit nicht umgesetzt werden. Ein Ausbau des bereits vorhandenen Systems wäre alternativ ebenfalls denkbar.

Des Weiteren bleiben noch die Implementierung und der Test der konkreten Vorschläge zur Verbesserung der Algorithmen zu bearbeiten. Auch das mehrfach erwähnte und in seiner groben Struktur vorgestellte Mustersuch-API muss in Zukunft noch detaillierter entworfen und umgesetzt werden. Dies bietet sich als Gegenstand von weiterführenden Diplom- oder Studienarbeiten an.

Literaturverzeichnis

[Auer97] Ken Auer : *Fundamental Elements of an Extendible Java Framework,* 1997/98. RoleModel Software Inc.

[Antoniol+98] G. Antoniol, R. Fiutem, L. Cristoforetti. *Design pattern recovery in object-oriented software.* In 6th International Workshop on Program Comprehension (Ischia, Italy, June 1998), pp. 153-160. http://serg.ing.unisannio.it/~antoniol/papers/iwpc98.ps.gz

[Balzert99] Balzert, Helmut: *Lehrbuch Grundlagen der Informatik*. Spektrum Akademischer Verlag GmbH, Heidelberg, Berlin, 1999.

[BalzertHei99] Balzert, Heide: *Lehrbuch der Objektmodellierung – Analyse und Entwurf*. Spektrum Akademischer Verlag GmbH,Heidelberg, Berlin, 1999.

[Bansiya98] Jagdish Bansiya. *Automatic Design-Pattern Identification.* Dr. Dobb's Journal, 1998. www.ddj.com/articles/1998/9806/9806a/9806a.htm?topic=patterns

[Bergenti+00] Frederico Bergenti, Agostino Poggi. *Improving UML designs using automatic design pattern detection.* In Proc.12th International Conference on Software Engineering and Knowlege Engineering (SEKE 2000), pp. 336-343, Chicago, IL, 2000. ftp://cs.pitt.edu/chang/handbook/66Ub.pdf

[Brown96] Kyle Brown. *Design reverse-engineering and automated design pattern detection in Smalltalk.* Master's thesis, Department of Computer Engineering, North Carolina State University, 1996. http://hillside.net/patterns/papers/

[Gamma+96] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Entwurfsmuster – Elemente wiederverwendbarer Software.* Addison-Wesley, 1996.

[Gumm+00] Gumm, Heinz-Peter; Sommer, Manfred: Einführung in die Informatik, 4. überarbeitete Auflage. Oldenbourg Wissenschaftsverlag GmbH, Oldenbourg, 2000.

[Herbig03] Herbig, Jens: *Reverse Engineering von Entwurfsmustern*, 2003. Technische Universität Ilmenau. Studienjahresarbeit.

[Jakarta] Jakarta
http://jakarta.apache.org/tomcat/

[J2SE] Java 2 Standard Edition
http://java.sun.com/j2se/1.4.0/

[Keller+99] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, Patrick Pagé. *Pattern-based reverse-engineering of design components.* In Proc. of the 21th Int. Conf. on Software Engineering, Los Angeles, USA, pages 226-235. IEEE Computer Society Press, May 1999.
http://www.iro.umontreal.ca/~schauer/Private/Publications/icse1999/icse1999.pdf

[Kim+00] Hyoseob Kim, Cornelia Boldyreff. *A method to recover design patterns using software product metrics.* In Proceedings of the Sixth International Conference on Software Reuse (ICSR6), Vienna, Austria, June 27-29, 2000.
http://www.soi.city.ac.uk/~hkim69/publications/icsr6.pdf

[Krämer+96] Christian Krämer, Lutz Prechelt. *Design recovery by automated search for structural design patterns in object-oriented software.* In Proc. of the Working Conference on Reverse Engeneering (Monterey, CA, November 1996), 208-215.
http://www.ubka.uni-karlsruhe.de/cgi-bin/psgunzip/1996/informatik/35/35.pdf

[Niere+01] Jörg Niere, Jörg P. Wadsack, Lothar Wendehals. *Design pattern recovery based on source code analysis with fuzzy logic.* , Tech. Rep. tr-ri-01-222, University of Paderborn, Paderborn, Germany, March 2001.
http://www.upb.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Papers/2001/tr-ri-01-222.pdf

[Naumann01] Naumann, Sebastian: *Reverse-Engineering von Entwurfsmustern*, 2001. Technische Universität Ilmenau. Diplomarbeit.

[Patterns] PatternStories
http://wiki.cs.uiuc.edu/PatternStories/DesignPatterns

[RMS] Rolemodel Software
http://www.rolemodelsoftware.com/drawlets/index.php

[Schneider+00] Schneider, Uwe; Werner, Dieter: *Taschenbuch der Informatik*. 3., völlig neu bearbeitete Auflage, Fachbuchverlag Leipzig, 2000.

[Streitferdt] Streitferdt, Detlef: *Suchkriterien für Gamma-Muster*, Technische Universität Ilmenau. Whitepaper.

[Shull+96] Forrest Shull, Walcélio L. Melo, Victor R. Basili. *An inductive method for discovering design patterns from object-oriented software systems.* Technical Report UMIACS-TR-96-10, University of Maryland, 1996. http://citeseer.nj.nec.com/cache/papers2/cs/1392/ftp:zSzzSzftp.cs.umd.eduzSzpubzSzpaperszSzpaperszSzncstrl.umcpzSzCS-TR-3597zSzCS-TR-3597.pdf/shull96inductive.pdf

Glossar

Abstrakte Klasse

Von einer abstrakten Klasse können keine Objekte erzeugt werden. Die abstrakte Klasse spielt eine wichtige Rolle in Vererbungsstrukturen, wo sie die Gemeinsamkeiten einer Gruppe von Unterklassen definiert. Damit eine abstrakte Klasse verwendet werden kann, muss von ihr zunächst eine Unterklasse abgeleitet werden. [BalzertHei99]

Aggregation

Eine Aggregation ist ein Sonderfall der Assoziation. Sie liegt dann vor, wenn zwischen den Objekten der beteiligten Klassen eine Beziehung vorliegt, die sich als „ist Teil von“ oder „besteht aus“ beschreiben lässt. [BalzertHei99]

Algorithmus

Der Begriff Algorithmus bezeichnet eine Vorschrift zur Lösung eines Problems, die für eine Realisierung in Form eines Programms auf einem Computer geeignet ist. [Schneider+00]

Assoziation

Modelliert Verbindungen zwischen Objekten einer oder mehrerer Klassen. Eine Assoziation modelliert stets Beziehungen zwischen Objekten, nicht zwischen Klassen. Es ist jedoch üblich, von einer Assoziation zwischen Klassen zu sprechen, obwohl streng genommen die Objekte dieser Klasse gemeint sind. Die Art der Assoziation wird durch ihre Kardinalität angegeben. [Balzert99]

Attribut

Beschreibt, welche Daten die Objekte der Klasse enthalten. [Balzert99]

C

C ist eine prozedurale Programmiersprache zur Systemprogrammierung. [Schneider+00]

C++

C++ ist sowohl eine prozedurale als auch eine objektorientierte Programmiersprache, welche auf C basiert und universell zur Anwendung kommt. [Schneider+00]

CASE

Computer Aided Software Engineering. Der durch Software-Werkzeuge unterstützte Prozess der Software-Entwicklung. [Müller97 zitiert nach Naumann01]

CASE-Tool

Werkzeug zur Unterstützung des Prozesses der Softwareentwicklung.

Design Pattern

Siehe Entwurfsmuster.

Entwurfsmuster

Ein Entwurfsmuster ist eine bestimmte Vorgehensweise bei der Definition von Klassen und bei der Gestaltung ihrer Zusammenarbeit. [Schneider+00]

Erzeugungsmuster

Erzeugungsmuster helfen dabei, ein System unabhängig davon zu machen, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden. [BalzertHei99]

Funktion

Siehe Operation.

Fuzzy-Logik

Die Fuzzy-Logik kann als Erweiterung der klassischen Logik angesehen werden und wurde zur Modellierung von Ungenauigkeit bzw. Vagheit eingeführt. [Schneider+00]

Header-Datei

C- und C++-Programme werden meistens so strukturiert, dass man die Definitionen von verwandten Funktionen in eigene Dateien schreibt. Um diese Funktionsdefinitionen in einer Datei bekannt zu machen, verwendet man die Deklarationen, die in eigenen Header-Dateien zusammengefasst werden. [Schneider+00]

Instanz

Siehe Objekt.

Java

Java ist eine objektorientierte Programmiersprache zum universellen Einsatz, welche Smalltalk und C++ als Vorläufer hat. [Schneider+00]

Kardinalität

Legt die Wertigkeit einer Assoziation fest, d.h. die Anzahl der an einer Assoziation beteiligten Objekte. [Balzert99]

Klasse

Beschreibt in Form einer Schablone eine Kategorie von Objekten, die gleiche oder ähnliche Strukturen und Verhaltensmuster aufweisen. Von einer Klasse können Objekte erzeugt werden. [Balzert99]

Klassendiagramm

Stellt die objektorientierten Konzepte Klasse, Attribute, Operationen und Beziehungen zwischen Klassen in graphischer Form dar. [Balzert99]

Kollaborationsdiagramm

Erweiterung des Objektdiagramms um Botschaften. Durch eine hierarchische Nummerierung, die Angabe des Operationsnamens und der Botschaftsrichtung durch einen Pfeil sind Ablaufsequenzen darstellbar. [Balzert99]

Komposition

Die Komposition ist eine besondere Form der Aggregation. Beim Löschen des Ganzen müssen auch alle Teile gelöscht werden. Jedes Teil kann – zu einem Zeitpunkt – nur zu einem Ganzen gehören. Es kann jedoch einem anderen Ganzen zugeordnet werden. Die dynamische Semantik des Ganzen gilt auch für seine Teile. [BalzertHei99]

Konstruktor

Spezielle Operation zum Erzeugen von Objekten. Der Konstruktorname entspricht dem Klassennamen gefolgt von einer Parameterliste in runden Klammern. Parameter dienen zur Initialisierung von Attributwerten. Sind keine Parameter angegeben, dann wird ein Objekt erzeugt und die Attribute des Objekts werden mit Standardwerten vorbelegt. [Balzert99]

Methode

siehe Operation.

Modul

Ein Modul ist eine eigenständige Programmkonstrukt (z.B. Unterprogramm, Prozedur), der eine Entwurfsentscheidung realisiert. [Schneider+00]

Objekt

Ausprägung physikalischer Komponenten oder abstrakter Sachverhalte, die individuell sind und durch Eigenschaften und Verhalten beschrieben werden. [Balzert99]

Operation

Ausführbare Tätigkeit im Sinne einer Funktion bzw. eines Algorithmus; beschreibt das Verhalten eines Objekts bzw. einer Klasse. [Balzert99]

Polymorphismus

Das mit der Vererbung verbundene Prinzip des Polymorphismus (gleiche Funktionalität für verschiedene Datentypen) erlaubt die Entwicklung allgemein verwendbarer Softwarekomponenten. [Schneider+00]

Präzision

Präzision steht für das Verhältnis der Anzahl richtig erkannter Muster zu der Anzahl insgesamt erkannter Muster. [Streitferdt]

Prolog

Prolog ist eine logische Programmiersprache. Prolog-Programme sind Logikprogramme, die durch ein Resolutionsverfahren mit Backtracking ausgeführt werden. Sie findet vorrangig in der künstlichen Intelligenz Anwendung. [Schneider+00]

Pseudocode

Pseudocode ist eine Notationsform für Algorithmen, die stark an eine Programmiersprache angelehnt ist.

Rational Rose

Rational Rose ist ein CASE-Tool der Firma Rational.

Recall

Recall steht für das Verhältnis der Anzahl tatsächlich enthaltener Muster in einem System zu der Anzahl der erkannten Muster. [Streitferdt]

Referenz

Ein Wert eines Objekts, der ein anderes Objekt identifiziert. [Gamma+96 zitiert nach Naumann01]

Reverse-Engineering

Unter Reverse Engineering versteht man die Extraktion und Repräsentation von Informationen aus einem Softwaresystem (Spezifikation) in einer anderen Form oder auf einem höheren Abstraktionsniveau. [Müller97 zitiert nach Naumann01]

Schnittstelle

Eine Schnittstelle unterscheidet sich dadurch von einer Klasse, dass sie nur abstrakte Methoden und keine Variablen enthält (wohl Klassenkonstanten). [Schneider+00]

Sequenzdiagramm

Graphische, zeitbasierte Darstellung mit vertikaler Zeitachse von Botschaften zwischen Objekten und Klassen. Botschaften werden durch horizontale Linien, Objekte und Klassen durch gestrichelte vertikale Linien repräsentiert. [Balzert99]

Smalltalk

Smalltalk ist eine objektorientierte Programmiersprache, welche zur Programmierung von graphischen Oberflächen angewendet wird. [Schneider+00]

Strukturmuster

Strukturmuster befassen sich damit, wie Klassen und Objekte zu größeren Strukturen zusammengesetzt werden. [BalzertHei99]

Together ControlCenter

Das Together ControlCenter ist ein CASE-Tool der Firma Togethersoft.

Überschreiben

Definition einer von einer Oberklasse geerbten Operation in einer Unterklasse. [Gamma+96 zitiert nach Naumann01]

UML

Unified Modeling Language. Notation zur graphischen Darstellung objektorientierter Konzepte. Zur graphischen Darstellung gehören Klassendiagramme, Objektdiagramme, Kollaborationsdiagramme und Sequenzdiagramme. [Balzert99]

Vererbung

Die Vererbung beschreibt die Beziehung zwischen einer allgemeineren Klasse und einer spezialisierten Klasse. Die spezialisierte Klasse erweitert die Liste der Attribute, Operationen und Assoziationen der Basisklasse. Operationen der Basisklasse dürfen redefiniert (überschrieben) werden. Es entsteht eine Klassenhierarchie oder Vererbungsstruktur. [BalzertHei99]

Verhaltensmuster

Verhaltensmuster befassen sich mit der Interaktion zwischen Objekten und Klassen. Sie beschreiben komplexe Kontrollflüsse, die zur Laufzeit schwer nachvollziehbar sind. Sie lenken die Aufmerksamkeit weg vom Kontrollfluss hin zu der Art und Weise, wie die Objekte interagieren. [BalzertHei99]

Wasserfall-Modell

Das Wasserfall-Modell ist ein Vorgehensmodell der Softwareentwicklung und unterteilt diese in einzelne Phasen, wobei die Ergebnisse einer Phase in die Folgephase „fallen“, d.h. dort als Eingabe benutzt werden. [Schneider+00]

Wartung

Wartung bezeichnet Änderungen an Softwaresystemen, die nach der Inbetriebnahme erfolgen. Dazu gehören das Korrigieren von Fehlern und das Hinzufügen bzw. die Änderung von Funktionalität. [Naumann01]

XML

XML (eXtensible Markup Language) ist eine Dokumentenbeschreibungssprache, welche die Trennung von Struktur und Inhalt, die Beschreibung strukturierter Daten, die Möglichkeit zur strukturellen Validierung von Daten und eine Erweiterbarkeit durch Definition neuer Tags bietet. [Schneider+00]

Anhang A: Quelltext project.dtd

project.dtd

```
<!ELEMENT project (abstract-factories, builders, factory-methods,
prototypes, singletons, object-adapters, bridges, decorators, facades,
flyweights, composites, proxies, commands, observers, visitors,
interpreters, iterators, mementos, template-methods, strategies,
mediators, states, chains-of-resp)>

<!ELEMENT abstract-factories (abstract-factory+)>
<!ELEMENT abstract-factory (abstract-factory-class, concrete-factory-
class, factory-methods, concrete-products, abstract-products)>
<!ELEMENT factory-methods (factory-method+)>
<!ELEMENT concrete-products (concrete-product+)>
<!ELEMENT abstract-products (abstract-product+)>
<!ELEMENT abstract-factory-class (#PCDATA)>
<!ELEMENT concrete-factory-class (#PCDATA)>
<!ELEMENT factory-method (#PCDATA)>
<!ELEMENT concrete-product (#PCDATA)>
<!ELEMENT abstract-product (#PCDATA)>

<!ELEMENT builders (builder+)>
<!ELEMENT builder (builder-class, concrete-builder, product,
directors)>
<!ELEMENT directors (director+)>
<!ELEMENT builder-class (#PCDATA)>
<!ELEMENT concrete-builder (#PCDATA)>
<!ELEMENT product (#PCDATA)>
<!ELEMENT director (#PCDATA)>

<!ELEMENT factory-methods (factory-method+)>
<!ELEMENT factory-method (factory-method-names, concrete-builder,
builder, concrete-product, product)>
<!ELEMENT factory-method-names (factory-method-name+)>
<!ELEMENT factory-method-name (#PCDATA)>
<!ELEMENT concrete-builder (#PCDATA)>
<!ELEMENT builder (#PCDATA)>
<!ELEMENT concrete-product (#PCDATA)>
<!ELEMENT product (#PCDATA)>

<!ELEMENT prototypes (prototype+)>
<!ELEMENT prototype (prototype-class)>
<!ELEMENT prototype-class (#PCDATA)>

<!ELEMENT singletons (singleton+)>
<!ELEMENT singleton (singleton-class)>
<!ELEMENT singleton-class (#PCDATA)>

<!ELEMENT object-adapters (object-adapter+)>
<!ELEMENT object-adapter (adapter-class, adaptee, target)>
<!ELEMENT adapter-class (#PCDATA)>
<!ELEMENT adaptee (#PCDATA)>
<!ELEMENT target (#PCDATA)>

<!ELEMENT bridges (bridge+)>
<!ELEMENT bridge (abstraction, implementor)>
<!ELEMENT abstraction (#PCDATA)>
<!ELEMENT implementor (#PCDATA)>
```

```
<!ELEMENT decorators (decorator+)>
<!ELEMENT decorator (concrete-decorator, decorator-class, component)>
<!ELEMENT concrete-decorator (#PCDATA)>
<!ELEMENT decorator-class (#PCDATA)>
<!ELEMENT component (#PCDATA)>

<!ELEMENT facades (facade+)>
<!ELEMENT facade (facade-class)>
<!ELEMENT facade-class (#PCDATA)>

<!ELEMENT flyweights (flyweight+)>
<!ELEMENT flyweight (flyweight-factory, flyweight-class, concrete-
flyweight)>
<!ELEMENT flyweight-factory (#PCDATA)>
<!ELEMENT flyweight-class (#PCDATA)>
<!ELEMENT concrete-flyweight (#PCDATA)>

<!ELEMENT composites (composite+)>
<!ELEMENT composite (composite-class, component)>
<!ELEMENT composite-class (#PCDATA)>
<!ELEMENT component (#PCDATA)>

<!ELEMENT proxies (proxy+)>
<!ELEMENT proxy (proxy-class, subject, real-subject)>
<!ELEMENT proxy-class (#PCDATA)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT real-subject (#PCDATA)>

<!ELEMENT commands (command+)>
<!ELEMENT command (command-class, concrete-commands, recievers,
caller)>
<!ELEMENT concrete-commands (concrete-command+)>
<!ELEMENT recievers (reciever+)>
<!ELEMENT command-class (#PCDATA)>
<!ELEMENT concrete-command (#PCDATA)>
<!ELEMENT reciever (#PCDATA)>
<!ELEMENT caller (#PCDATA)>

<!ELEMENT observers (observer+)>
<!ELEMENT observer (subject, observer-class, concrete-observers)>
<!ELEMENT concrete-observers (concrete-observer+)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT observer-class (#PCDATA)>
<!ELEMENT concrete-observer (#PCDATA)>

<!ELEMENT visitors (visitor+)>
<!ELEMENT visitor (visitor-class, elements)>
<!ELEMENT elements (element+)>
<!ELEMENT visitor-class (#PCDATA)>
<!ELEMENT element (#PCDATA)>

<!ELEMENT interpreters (interpreter+)>
<!ELEMENT interpreter (interpreter-class)>
<!ELEMENT interpreter-class (#PCDATA)>

<!ELEMENT iterators (iterator+)>
<!ELEMENT iterator (iterator-class, aggregate)>
<!ELEMENT iterator-class (#PCDATA)>
<!ELEMENT aggregate (#PCDATA)>
```

```
<!ELEMENT mementos (memento+)>
<!ELEMENT memento (memento-class, creator, depositor)>
<!ELEMENT memento-class (#PCDATA)>
<!ELEMENT creator (#PCDATA)>
<!ELEMENT depositor (#PCDATA)>

<!ELEMENT template-methods (template-method+)>
<!ELEMENT template-method (containing-class, method-names)>
<!ELEMENT method-names (method-name+)>
<!ELEMENT containing-class (#PCDATA)>
<!ELEMENT method-name (#PCDATA)>

<!ELEMENT strategies (strategy+)>
<!ELEMENT strategy (abstract-strategy, context, concrete-strategies)>
<!ELEMENT concrete-strategies (concrete-strategy+)>
<!ELEMENT abstract-strategy (#PCDATA)>
<!ELEMENT context (#PCDATA)>
<!ELEMENT concrete-strategy (#PCDATA)>

<!ELEMENT mediators (mediator+)>
<!ELEMENT mediator (concrete-mediator, concrete-colleagues)>
<!ELEMENT concrete-colleagues (concrete-colleague+)>
<!ELEMENT concrete-mediator (#PCDATA)>
<!ELEMENT concrete-colleague (#PCDATA)>

<!ELEMENT states (state+)>
<!ELEMENT state (abstract-state, concrete-states)>
<!ELEMENT concrete-states (concrete-state+)>
<!ELEMENT abstract-state (#PCDATA)>
<!ELEMENT concrete-state (#PCDATA)>

<!ELEMENT chains-of-resp (chain-of-resp+)>
<!ELEMENT chain-of-resp (root-class, overridden-method)>
<!ELEMENT root-class (#PCDATA)>
<!ELEMENT overridden-method (#PCDATA)>
```

Anhang B: Quelltext patterns.xml

patterns.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE project SYSTEM "project.dtd">
<project name="patterns.tpr" number-of-classes="88">
<abstract-factories number-of-patterns="1" duration-in-ms="631">
<abstract-factory>
<abstract-factory-class>AbstractFactory</abstract-factory-class>
<concrete-factory-class>ConcreteFactory</concrete-factory-class>
<factory-methods>
<factory-method>createAbstractProduct</factory-method>
<factory-method>createAbstractProduct2</factory-method>
</factory-methods>
<concrete-products>
<concrete-product>ConcreteProduct1</concrete-product>
<concrete-product>ConcreteProduct1</concrete-product>
</concrete-products>
<abstract-products>
<abstract-product>AbstractProduct</abstract-product>
<abstract-product>AbstractProduct</abstract-product>
</abstract-products>
</abstract-factory>
</abstract-factories>
<builders number-of-patterns="12" duration-in-ms="381">
<builder>
<builder-class>Target</builder-class>
<concrete-builder>Adapter</concrete-builder>
<product></product>
<directors>
</directors>
</builder>
<builder>
<builder-class></builder-class>
<concrete-builder>Abstraction</concrete-builder>
<product></product>
<directors>
</directors>
</builder>
<builder>
<builder-class>Builder</builder-class>
<concrete-builder>ConcreteBuilder</concrete-builder>
<product>Product</product>
<directors>
<director>Director</director>
</directors>
</builder>
<builder>
<builder-class></builder-class>
<concrete-builder>Handler</concrete-builder>
<product>Handler</product>
<directors>
<director>Handler</director>
</directors>
</builder>
<builder>
<builder-class>Command</builder-class>
<concrete-builder>ConcreteCommand</concrete-builder>
```

```
<product></product>
<directors>
<director>Caller</director>
</directors>
</builder>
<builder>
<builder-class>Component</builder-class>
<concrete-builder>Composite</concrete-builder>
<product></product>
<directors>
<director>Composite</director>
</directors>
</builder>
<builder>
<builder-class>DecoratorComponent</builder-class>
<concrete-builder>Decorator</concrete-builder>
<product></product>
<directors>
<director>Decorator</director>
</directors>
</builder>
<builder>
<builder-class></builder-class>
<concrete-builder>Creator</concrete-builder>
<product></product>
<directors>
</directors>
</builder>
<builder>
<builder-class></builder-class>
<concrete-builder>Memento</concrete-builder>
<product>String</product>
<directors>
<director>Depositor</director>
</directors>
</builder>
<builder>
<builder-class>ObserverSubject</builder-class>
<concrete-builder>ConcreteSubject</concrete-builder>
<product></product>
<directors>
</directors>
</builder>
<builder>
<builder-class>Subject</builder-class>
<concrete-builder>Proxy</concrete-builder>
<product></product>
<directors>
</directors>
</builder>
<builder>
<builder-class></builder-class>
<concrete-builder>Context</concrete-builder>
<product></product>
<directors>
</directors>
</builder>
</builders>
<factory-methods number-of-patterns="4" duration-in-ms="541">
<factory-method>
<factory-method-names>
```

```
<factory-method-name>createAbstractProduct</factory-method-name>
<factory-method-name>createAbstractProduct2</factory-method-name>
</factory-method-names>
<concrete-builder>ConcreteFactory</concrete-builder>
<builder>AbstractFactory</builder>
<concrete-product>ConcreteProduct1</concrete-product>
<product>AbstractProduct</product>
</factory-method>
<factory-method>
<factory-method-names>
<factory-method-name>factoryMethod</factory-method-name>
</factory-method-names>
<concrete-builder>ConcreteCreator</concrete-builder>
<builder>Creator</builder>
<concrete-product>ConcreteProduct</concrete-product>
<product>Product</product>
</factory-method>
<factory-method>
<factory-method-names>
<factory-method-name>cloneObj</factory-method-name>
</factory-method-names>
<concrete-builder>ConcretePrototype1</concrete-builder>
<builder>Prototype</builder>
<concrete-product>ConcretePrototype1</concrete-product>
<product>Prototype</product>
</factory-method>
<factory-method>
<factory-method-names>
<factory-method-name>cloneObj</factory-method-name>
</factory-method-names>
<concrete-builder>ConcretePrototype2</concrete-builder>
<builder>Prototype</builder>
<concrete-product>ConcretePrototype2</concrete-product>
<product>Prototype</product>
</factory-method>
</factory-methods>
<prototypes number-of-patterns="2" duration-in-ms="10">
<prototype><prototype-class>ConcretePrototype1</prototype-class>
</prototype><prototype><prototype-class>ConcretePrototype2</prototype-
class>
</prototype></prototypes><singletons number-of-patterns="1" duration-
in-ms="10">
<singleton>
<singleton-class>Singleton</singleton-class>
</singleton>
</singletons>
<object-adapters number-of-patterns="3" duration-in-ms="520">
<object-adapter>
<adapter-class>Adapter</adapter-class>
<adaptee>Adaptee</adaptee>
<target>Target</target>
</object-adapter>
<object-adapter>
<adapter-class>ConcreteCommand</adapter-class>
<adaptee>Receiver</adaptee>
<target>Command</target>
</object-adapter>
<object-adapter>
<adapter-class>Proxy</adapter-class>
<adaptee>RealSubject</adaptee>
<target>Subject</target>
```

```
</object-adapter>
</object-adapters>
<bridges number-of-patterns="25" duration-in-ms="1563">
<bridge>
<abstraction>Abstraction</abstraction>
<implementor>Implementor</implementor>
</bridge>
<bridge>
<abstraction>ConcreteBuilder</abstraction>
<implementor>Product</implementor>
</bridge>
<bridge>
<abstraction>Director</abstraction>
<implementor>Builder</implementor>
</bridge>
<bridge>
<abstraction>Caller</abstraction>
<implementor>Command</implementor>
</bridge>
<bridge>
<abstraction>CommandClient</abstraction>
<implementor>Receiver</implementor>
</bridge>
<bridge>
<abstraction>ConcreteCommand</abstraction>
<implementor>String</implementor>
</bridge>
<bridge>
<abstraction>ClassA1</abstraction>
<implementor>Facade</implementor>
</bridge>
<bridge>
<abstraction>ClassA2</abstraction>
<implementor>Facade</implementor>
</bridge>
<bridge>
<abstraction>ClassA3</abstraction>
<implementor>Facade</implementor>
</bridge>
<bridge>
<abstraction>Facade</abstraction>
<implementor>ClassB3</implementor>
</bridge>
<bridge>
<abstraction>ConcreteFlyweight</abstraction>
<implementor>String</implementor>
</bridge>
<bridge>
<abstraction>FlyweightClient</abstraction>
<implementor>SeparateUsedFlyweight</implementor>
</bridge>
<bridge>
<abstraction>FlyweightFactory</abstraction>
<implementor>Flyweight</implementor>
</bridge>
<bridge>
<abstraction>SeparateUsedFlyweight</abstraction>
<implementor>String</implementor>
</bridge>
<bridge>
<abstraction>InterpreterClient</abstraction>
```

```
<implementor>AbstractTerm</implementor>
</bridge>
<bridge>
<abstraction>NonTerminalExpression</abstraction>
<implementor>AbstractTerm</implementor>
</bridge>
<bridge>
<abstraction>ConcreteIterator</abstraction>
<implementor>ConcreteAggregate</implementor>
</bridge>
<bridge>
<abstraction>ConcreteMediator</abstraction>
<implementor>ConcreteColleague2</implementor>
</bridge>
<bridge>
<abstraction>Creator</abstraction>
<implementor>String</implementor>
</bridge>
<bridge>
<abstraction>Depositor</abstraction>
<implementor>Memento</implementor>
</bridge>
<bridge>
<abstraction>Memento</abstraction>
<implementor>String</implementor>
</bridge>
<bridge>
<abstraction>ConcreteSubject</abstraction>
<implementor>Vector</implementor>
</bridge>
<bridge>
<abstraction>PrototypeClient</abstraction>
<implementor>Prototype</implementor>
</bridge>
<bridge>
<abstraction>Proxy</abstraction>
<implementor>RealSubject</implementor>
</bridge>
<bridge>
<abstraction>StrategyContext</abstraction>
<implementor>Strategy</implementor>
</bridge>
</bridges>
<decorators number-of-patterns="1" duration-in-ms="40">
<decorator>
<concrete-decorator>ConcreteDecorator</concrete-decorator>
<decorator-class>Decorator</decorator-class>
<component>DecoratorComponent</component>
</decorator>
</decorators>
<facades number-of-patterns="77" duration-in-ms="991">
<facade>
<facade-class>AbstractFactory</facade-class>
</facade>
<facade>
<facade-class>AbstractProduct</facade-class>
</facade>
<facade>
<facade-class>ConcreteFactory</facade-class>
</facade>
<facade>
```

```
<facade-class>ConcreteProduct1</facade-class>
</facade>
<facade>
<facade-class>Adapter</facade-class>
</facade>
<facade>
<facade-class>Target</facade-class>
</facade>
<facade>
<facade-class>Abstraction</facade-class>
</facade>
<facade>
<facade-class>ConcreteImplementor1</facade-class>
</facade>
<facade>
<facade-class>ConcreteImplementor2</facade-class>
</facade>
<facade>
<facade-class>Implementor</facade-class>
</facade>
<facade>
<facade-class>SpecializedAbstraction</facade-class>
</facade>
<facade>
<facade-class>Builder</facade-class>
</facade>
<facade>
<facade-class>ConcreteBuilder</facade-class>
</facade>
<facade>
<facade-class>Director</facade-class>
</facade>
<facade>
<facade-class>Product</facade-class>
</facade>
<facade>
<facade-class>Caller</facade-class>
</facade>
<facade>
<facade-class>Command</facade-class>
</facade>
<facade>
<facade-class>CommandClient</facade-class>
</facade>
<facade>
<facade-class>ConcreteCommand</facade-class>
</facade>
<facade>
<facade-class>Receiver</facade-class>
</facade>
<facade>
<facade-class>Composite</facade-class>
</facade>
<facade>
<facade-class>Leaf</facade-class>
</facade>
<facade>
<facade-class>ConcreteComponent</facade-class>
</facade>
<facade>
<facade-class>ConcreteDecorator</facade-class>
```

```
</facade>
<facade>
<facade-class>Decorator</facade-class>
</facade>
<facade>
<facade-class>ClassA1</facade-class>
</facade>
<facade>
<facade-class>ClassA2</facade-class>
</facade>
<facade>
<facade-class>ClassA3</facade-class>
</facade>
<facade>
<facade-class>ClassB1</facade-class>
</facade>
<facade>
<facade-class>ClassB2</facade-class>
</facade>
<facade>
<facade-class>ClassB3</facade-class>
</facade>
<facade>
<facade-class>Facade</facade-class>
</facade>
<facade>
<facade-class>ConcreteCreator</facade-class>
</facade>
<facade>
<facade-class>ConcreteProduct</facade-class>
</facade>
<facade>
<facade-class>Creator</facade-class>
</facade>
<facade>
<facade-class>Product</facade-class>
</facade>
<facade>
<facade-class>ConcreteFlyweight</facade-class>
</facade>
<facade>
<facade-class>Flyweight</facade-class>
</facade>
<facade>
<facade-class>FlyweightClient</facade-class>
</facade>
<facade>
<facade-class>FlyweightFactory</facade-class>
</facade>
<facade>
<facade-class>SeparateUsedFlyweight</facade-class>
</facade>
<facade>
<facade-class>AbstractTerm</facade-class>
</facade>
<facade>
<facade-class>InterpreterClient</facade-class>
</facade>
<facade>
<facade-class>InterpreterContext</facade-class>
</facade>
```

```
<facade>
<facade-class>NonTerminalExpression</facade-class>
</facade>
<facade>
<facade-class>TerminalExpression</facade-class>
</facade>
<facade>
<facade-class>Aggregate</facade-class>
</facade>
<facade>
<facade-class>ConcreteAggregate</facade-class>
</facade>
<facade>
<facade-class>ConcreteIterator</facade-class>
</facade>
<facade>
<facade-class>MyIterator</facade-class>
</facade>
<facade>
<facade-class>Colleague</facade-class>
</facade>
<facade>
<facade-class>ConcreteColleague1</facade-class>
</facade>
<facade>
<facade-class>ConcreteColleague2</facade-class>
</facade>
<facade>
<facade-class>ConcreteMediator</facade-class>
</facade>
<facade>
<facade-class>Mediator</facade-class>
</facade>
<facade>
<facade-class>Creator</facade-class>
</facade>
<facade>
<facade-class>Depositor</facade-class>
</facade>
<facade>
<facade-class>Memento</facade-class>
</facade>
<facade>
<facade-class>ConcreteObserver</facade-class>
</facade>
<facade>
<facade-class>ConcreteSubject</facade-class>
</facade>
<facade>
<facade-class>ObserverSubject</facade-class>
</facade>
<facade>
<facade-class>ConcretePrototype1</facade-class>
</facade>
<facade>
<facade-class>ConcretePrototype2</facade-class>
</facade>
<facade>
<facade-class>Prototype</facade-class>
</facade>
<facade>
```

```
<facade-class>PrototypeClient</facade-class>
</facade>
<facade>
<facade-class>ConcreteState</facade-class>
</facade>
<facade>
<facade-class>Context</facade-class>
</facade>
<facade>
<facade-class>ConcreteStrategy1</facade-class>
</facade>
<facade>
<facade-class>ConcreteStrategy2</facade-class>
</facade>
<facade>
<facade-class>ConcreteStrategy3</facade-class>
</facade>
<facade>
<facade-class>Strategy</facade-class>
</facade>
<facade>
<facade-class>StrategyContext</facade-class>
</facade>
<facade>
<facade-class>AbstractClass</facade-class>
</facade>
<facade>
<facade-class>ConcreteClass</facade-class>
</facade>
<facade>
<facade-class>ConcreteElement</facade-class>
</facade>
<facade>
<facade-class>Element</facade-class>
</facade>
<facade>
<facade-class>Visitor</facade-class>
</facade>
</facades>
<flyweights number-of-patterns="1" duration-in-ms="50">
<flyweight>
<flyweight-factory>FlyweightFactory</flyweight-factory>
<flyweight-class>Flyweight</flyweight-class>
<concrete-flyweight>ConcreteFlyweight</concrete-flyweight>
</flyweight>
</flyweights>
<composites number-of-patterns="1" duration-in-ms="20">
<composite>
<composite-class>Composite</composite-class>
<component>Component</component>
</composite>
</composites>
<proxies number-of-patterns="3" duration-in-ms="10">
<proxy>
<proxy-class>Adapter</proxy-class>
<subject>Target</subject>
<real-subject>Adaptee</real-subject>
</proxy>
<proxy>
<proxy-class>Decorator</proxy-class>
<subject>DecoratorComponent</subject>
```

```
<real-subject>DecoratorComponent</real-subject>
</proxy>
<proxy>
<proxy-class>Proxy</proxy-class>
<subject>Subject</subject>
<real-subject>RealSubject</real-subject>
</proxy>
</proxies>
<commands number-of-patterns="1" duration-in-ms="862">
<command>
<command-class>Command</command-class>
<concrete-commands>
<concrete-command>ConcreteCommand</concrete-command>
<concrete-command>ConcreteCommand</concrete-command>
</concrete-commands>
<recievers>
<reciever>Receiver</reciever>
<reciever>Command</reciever>
</recievers>
<caller>Command</caller>
</command>
</commands>
<observers number-of-patterns="1" duration-in-ms="140">
<observer>
<subject>ObserverSubject</subject>
<observer-class>Observer</observer-class>
<concrete-observers>
<concrete-observer>ConcreteObserver</concrete-observer>
</concrete-observers>
</observer>
</observers>
<visitors number-of-patterns="1" duration-in-ms="30">
<visitor>
<visitor-class>Visitor</visitor-class>
<elements>
<element>ConcreteElement</element>
<element>Element</element>
</elements>
</visitor>
</visitors>
<interpreters number-of-patterns="2" duration-in-ms="981">
<interpreter>
<interpreter-class>Component</interpreter-class>
</interpreter>
<interpreter>
<interpreter-class>AbstractTerm</interpreter-class>
</interpreter>
</interpreters>
<iterators number-of-patterns="1" duration-in-ms="10">
<iterator>
<iterator-class>ConcreteIterator</iterator-class>
<aggregate>ConcreteAggregate</aggregate>
</iterator>
</iterators>
<mementos number-of-patterns="1" duration-in-ms="20">
<memento>
<memento-class>Memento</memento-class>
<creator>Creator</creator>
<depositor>Depositor</depositor>
</memento>
</mementos>
```

```
<template-methods number-of-patterns="1" duration-in-ms="10">
<template-method>
<containing-class>AbstractClass</containing-class>
<method-names>
<method-name>templateMethod</method-name>
<method-name>templateMethod</method-name>
</method-names>
</template-method>
</template-methods>
<strategies number-of-patterns="17" duration-in-ms="1172">
<strategy>
<abstract-strategy>AbstractFactory</abstract-strategy>
<context></context>
<concrete-strategies>
<concrete-strategy>ConcreteFactory</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>AbstractProduct</abstract-strategy>
<context></context>
<concrete-strategies>
<concrete-strategy>ConcreteProduct1</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>Implementor</abstract-strategy>
<context>Abstraction</context>
<concrete-strategies>
<concrete-strategy>ConcreteImplementor1</concrete-strategy>
<concrete-strategy>ConcreteImplementor2</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>Command</abstract-strategy>
<context>Caller</context>
<concrete-strategies>
<concrete-strategy>ConcreteCommand</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>Component</abstract-strategy>
<context>Composite</context>
<concrete-strategies>
<concrete-strategy>Leaf</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>DecoratorComponent</abstract-strategy>
<context>Decorator</context>
<concrete-strategies>
<concrete-strategy>ConcreteComponent</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>Flyweight</abstract-strategy>
<context>FlyweightFactory</context>
<concrete-strategies>
<concrete-strategy>ConcreteFlyweight</concrete-strategy>
<concrete-strategy>SeparateUsedFlyweight</concrete-strategy>
</concrete-strategies>
</strategy>
```

```
<strategy>
<abstract-strategy>AbstractTerm</abstract-strategy>
<context>NonTerminalExpression</context>
<concrete-strategies>
<concrete-strategy>TerminalExpression</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>Aggregate</abstract-strategy>
<context></context>
<concrete-strategies>
<concrete-strategy>ConcreteAggregate</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>MyIterator</abstract-strategy>
<context></context>
<concrete-strategies>
<concrete-strategy>ConcreteIterator</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>Colleague</abstract-strategy>
<context></context>
<concrete-strategies>
<concrete-strategy>ConcreteColleague1</concrete-strategy>
<concrete-strategy>ConcreteColleague2</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>Mediator</abstract-strategy>
<context>Colleague</context>
<concrete-strategies>
<concrete-strategy>ConcreteMediator</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>Observer</abstract-strategy>
<context>ObserverSubject</context>
<concrete-strategies>
<concrete-strategy>ConcreteObserver</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>Prototype</abstract-strategy>
<context>PrototypeClient</context>
<concrete-strategies>
<concrete-strategy>ConcretePrototype1</concrete-strategy>
<concrete-strategy>ConcretePrototype2</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>Subject</abstract-strategy>
<context></context>
<concrete-strategies>
<concrete-strategy>Proxy</concrete-strategy>
<concrete-strategy>RealSubject</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>Strategy</abstract-strategy>
```

```
<context>StrategyContext</context>
<concrete-strategies>
<concrete-strategy>ConcreteStrategy1</concrete-strategy>
<concrete-strategy>ConcreteStrategy2</concrete-strategy>
<concrete-strategy>ConcreteStrategy3</concrete-strategy>
</concrete-strategies>
</strategy>
<strategy>
<abstract-strategy>Element</abstract-strategy>
<context></context>
<concrete-strategies>
<concrete-strategy>ConcreteElement</concrete-strategy>
</concrete-strategies>
</strategy>
</strategies>
<mediators number-of-patterns="25" duration-in-ms="120">
<mediator>
<concrete-mediator>AbstractFactory</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>AbstractProduct</concrete-colleague>
<concrete-colleague>ConcreteFactory</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>AbstractProduct</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>AbstractFactory</concrete-colleague>
<concrete-colleague>ConcreteProduct1</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>ConcreteFactory</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>ConcreteProduct1</concrete-colleague>
<concrete-colleague>AbstractFactory</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>Adaptee</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>Adapter</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>Adapter</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>Adaptee</concrete-colleague>
<concrete-colleague>Target</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>Target</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>Adapter</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>Component</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>Leaf</concrete-colleague>
<concrete-colleague>Composite</concrete-colleague>
```

```
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>Leaf</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>Component</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>ConcreteComponent</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>DecoratorComponent</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>ConcreteDecorator</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>Decorator</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>Decorator</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>DecoratorComponent</concrete-colleague>
<concrete-colleague>ConcreteDecorator</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>DecoratorComponent</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>ConcreteComponent</concrete-colleague>
<concrete-colleague>Decorator</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>ConcreteCreator</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>ConcreteProduct</concrete-colleague>
<concrete-colleague>Creator</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>ConcreteProduct</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>ConcreteCreator</concrete-colleague>
<concrete-colleague>Product</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>Creator</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>Product</concrete-colleague>
<concrete-colleague>ConcreteCreator</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>Product</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>Creator</concrete-colleague>
<concrete-colleague>ConcreteProduct</concrete-colleague>
</concrete-colleagues>
```

```
</mediator>
<mediator>
<concrete-mediator>ConcreteMediator</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>ConcreteColleague1</concrete-colleague>
<concrete-colleague>ConcreteColleague2</concrete-colleague>
<concrete-colleague>Mediator</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>ConcreteObserver</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>ConcreteSubject</concrete-colleague>
<concrete-colleague>Observer</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>ConcreteSubject</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>Vector</concrete-colleague>
<concrete-colleague>ObserverSubject</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>Observer</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>ConcreteObserver</concrete-colleague>
<concrete-colleague>ObserverSubject</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>ObserverSubject</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>ConcreteSubject</concrete-colleague>
<concrete-colleague>Observer</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>Context</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>State</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>State</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>Context</concrete-colleague>
<concrete-colleague>ConcreteState</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>Element</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>Visitor</concrete-colleague>
<concrete-colleague>ConcreteElement</concrete-colleague>
</concrete-colleagues>
</mediator>
<mediator>
<concrete-mediator>Visitor</concrete-mediator>
<concrete-colleagues>
<concrete-colleague>Element</concrete-colleague>
```

```
</concrete-colleagues>
</mediator>
</mediators>
<states number-of-patterns="1" duration-in-ms="481">
<state>
<abstract-state>State</abstract-state>
<concrete-states>
<concrete-state>ConcreteState</concrete-state>
</concrete-states>
</state>
</states>
<chains-of-resp number-of-patterns="3" duration-in-ms="1362">
<chain-of-resp>
<root-class>Abstraction</root-class>
<overridden-method>operation</overridden-method>
</chain-of-resp>
<chain-of-resp>
<root-class>Handler</root-class>
<overridden-method>handleRequest</overridden-method>
</chain-of-resp>
<chain-of-resp>
<root-class>Context</root-class>
<overridden-method>sampleOperation</overridden-method>
</chain-of-resp>
</chains-of-resp>
</project>
```

Anhang C: Inhalt der CD

/Algorithmen

Hier ist das Together Projekt abgelegt, welches alle Algorithmen implementiert.

/Diplomarbeit

Hier ist die Diplomarbeit als Microsoft Word Dokument und als PDF abgelegt.

/Diplomarbeit/Quellen

Hier sind einige Quellen für die Diplomarbeit abgelegt.

/Referenzsystem

Hier ist der Quelltext des selbst entwickelten Pattern-Projektes abgelegt.

/Testergebnisse

Hier sind die XML-Dateien mit den Testergebnissen abgelegt.

/Testsysteme

Hier ist der Quelltext der anderen zum Test verwendeten Systeme abgelegt.

Thesen

- Weniger als 20 Prozent aller Software-Projekte werden erfolgreich abgeschlossen.
- Die Analyse- und die Entwurfsphase bereiten den Softwareentwicklern essentielle Probleme.
- Entwurfsmuster unterstützen den Softwareentwickler in den Phasen der Analyse und des Entwurfs, um flexible und wartbare Software zu bauen.
- Durch den Einsatz von Entwurfsmustern werden Konzepte angewendet, welche sich in der Praxis bewährt haben. Durch das Profitieren von Erfahrungen anderer Entwickler werden die Fehler im Entwurf verringert.
- Jedes Softwaresystem muss gewartet werden.
- Es ist notwendig die Funktionsweise eines Softwaresystems zu verstehen, um es warten zu können.
- Das automatisierte Erkennen von Entwurfsmustern in einem Softwaresystem unterstützt das Verstehen der Funktionsweise des Systems.
- Die Leistungsfähigkeit von Algorithmen zur automatisierten Entwurfsmustererkennung muss vor deren praktischen Einsatz geprüft werden.
- Algorithmen zur automatisierten Entwurfsmustererkennung müssen zuverlässige Ergebnisse liefern, um sinnvoll eingesetzt werden zu können.
- Für die automatisierte Erkennung aller von Gamma, Helm, Johnson und Vlissides definierten Muster ist die Verfügbarkeit des gesamten Quelltextes unabdingbar.
- Die von Naumann entworfenen Algorithmen erzielen im Durchschnitt eine Präzision von 68 Prozent, womit sie bessere Ergebnisse erzielen als alle anderen Ansätze.
- Durch die Verbesserung der Algorithmen wird eine Steigerung der Präzision erreicht.

Ilmenau, den 05. Juli 2003 Antje Seidel

Inventarisierungsnummer 2004-07-05 / 065 / IN97 / 2232

Erklärung

Ich erkläre, dass ich die vorliegende Diplomarbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Ilmenau, den 05. Juli 2003 Antje Seidel