Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Institut für Theoretische und Technische Informatik
Fachgebiet Prozessinformatik

Reverse-Engineering von Entwurfsmustern

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

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

Sebastian Naumann
geboren am 4. August 1975 in Zossen
Matrikel 96
Matrikelnummer 25048

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

Beginn der Arbeit: 1. Juni 2001
Abgabe der Arbeit: 3. Dezember 2001

Inventarisierungsnummer: 2001-12-03 / 0047 / IN96 / 2232

Dieses Dokument wurde nach den Regeln der neuen deutschen Rechtschreibung verfasst.

Danksagungen

Danken möchte ich allen, die mich bei dieser Arbeit in irgendeiner Form unterstützt haben. Ganz besonderer Dank geht an meinen Betreuer Detlef Streitferdt für die unermüdlichen Anrufe bei Rational, für die Hinweise bei der Gestaltung und die inhaltlichen Ratschläge. Desweiteren danke ich meiner Freundin Andrea, einfach dafür, dass sie da ist; Alex, für die unzähligen gemeinsamen Mahlzeiten und für das Korrekturlesen; den Herren Fritz, Paschke und Sachse für die stets sofortige Hilfe bei Problemen mit dem Computersystem; Hanna, dass sie mir mein Zimmer zu Hause noch gelassen hat; meinen Eltern für die Unterstützung (besonders auch finanziell) über die Zeit des Studiums; Frau Dittmar für die Kammer hinter dem Bad; und Gott, dass er mit seiner schützenden und segnenden Hand immer bei mir war.

Kurzfassung

Auf Softwaresysteme kommen im Laufe ihres Lebens immer wieder Anpassungen, Änderungen und Weiterentwicklungen zu. Bevor Änderungen an einem System vorgenommen werden können, ist es vorab nötig zu verstehen, wie das System funktioniert. Erst mit diesem Wissen kann eine Änderung sicher durchgeführt werden.

Objektorientierte Entwurfsmuster (engl. *Design Patterns*) dienen dazu, Lösungen für bestimmte grundlegende Entwurfsprobleme in der objektorientierten Software-Entwicklung bereitzustellen. Jedes Entwurfsmuster vermittelt eine ganz bestimmte Idee, das heißt, es ist genau bekannt, welchen allgemeinen Zweck der Einsatz eines jeden Entwurfsmusters verfolgt und auch, welche Vor- und Nachteile gerade dieser Entwurf hat. Beim Verstehen eines Softwaresystems sind Entwurfsmuster daher von herausragender Bedeutung. Kennt ein Entwickler die Entwurfsmuster, die einem System zugrundeliegen, kann er das System auf einer abstrakteren Ebene erfassen und verstehen als es das rein objektorientierte Modell ermöglicht. Bei den in dem Buch "Entwurfsmuster" [Gamma+96] beschriebenen Lösungen zu allgemeinen Entwurfsproblemen handelt es sich um die am häufigsten verwendeten Muster bei der Softwareentwicklung. [Gamma+96] ist daher praktisch ein Quasi-Standard.

Für den Menschen ist es sehr schwierig oder bisweilen sogar unmöglich, Entwurfsmuster manuell in einem Softwaresystem zu identifizieren. Stünde ein Software-Werkzeug zur Verfügung, das diese Aufgabe automatisch ausführen könnte, so wäre das für ihn eine große Erleichterung.

Existierende Ansätze bewältigen diese Aufgabe nur teilweise; lediglich ein einziger ist in der Lage, alle Muster aus [Gamma+96] zu finden, versagt jedoch bei eindeutig definierten Mustern. Eine Gruppe von Arbeiten verwendet für die Suche bestimmte Schlüsselmerkmale. Dabei wird davon ausgegangen, dass jedes Muster Elemente in der Struktur, sogenannte Schlüsselmerkmale, besitzt, die bei einer Anwendung stets vorhanden sind und nach denen gesucht werden kann. Mit dieser Herangehensweise ist es möglich, eindeutig definierte Muster auch eindeutig zu identifizieren. Diese Arbeiten legen jedoch nicht für alle Muster solche Merkmale fest.

Mein Ansatz greift das Prinzip der Schlüsselmerkmale auf. Ich gebe für alle Muster aus [Gamma+96] Merkmale an und verfeinere die Suche durch die Hinzunahme von Merkmalen, die nicht vorhanden sein dürfen. Dadurch werden weniger Muster falsch erkannt. Um Merkmale, die nicht vorhanden sein dürfen, geeignet darzustellen, wird eine Erweiterung der *Object Modeling Technique* (OMT) vorgenommen.

Eine prototypische Umsetzung des Ansatzes erfolgt anhand von drei ausgewählten Mustern unter der Verwendung von *Rational Rose*.

Inventarisierungsnummer 2001-12-03 / 0047 / IN96 / 2232

Inhaltsverzeichnis

1 Einführung 11

1.1 Wartung 12
1.2 Programmverstehen 14
1.3 Entwurfsmuster 15
1.4 Über die Bedeutung von Entwurfsmustern beim Programmverstehen 17
1.5 Schwierigkeiten bei der manuellen Suche nach Entwurfsmustern 18
1.6 Problemstellung 19

2 Die automatische Suche nach Entwurfsmustern – Aktueller Stand 21

2.1 Suche nach minimalen Schlüsselstrukturen 23
2.1.1 DP++ 24
2.1.2 KT 25
2.1.3 SPOOL 26
2.2 Suche nach vollständigen Übereinstimmungen in der Klassenstruktur 29
2.2.1 Pat 30
2.2.2 IDEA 32
2.2.3 Mehrstufiger Suchprozess 33
2.3 Flexible Musterdefinition und Fuzzylogik 36
2.4 Suche anhand von Metriken 37
2.5 Induktive Methode für die manuelle Suche 41
2.6 Zusammenfassung und Fazit 42
2.7 Präzisierte Problemstellung 45

3 Eigener Ansatz 47

3.1 Prinzip 47
3.2 Elemente der Schlüsselstrukturen 48
3.3 Die Muster 52
3.4 Erzeugungsmuster 53
3.4.1 Abstrakte Fabrik 54
3.4.2 Erbauer 56
3.4.3 Fabrikmethode 57
3.4.4 Prototyp 58
3.4.5 Singleton 60
3.5 Strukturmuster 61
3.5.1 Adapter 61
3.5.2 Brücke 64
3.5.3 Dekorierer 65
3.5.4 Fassade 67
3.5.5 Fliegengewicht 69
3.5.6 Kompositum 71
3.5.7 Proxy 73
3.6 Verhaltensmuster 75
3.6.1 Befehl 75

3.6.2 Beobachter 77
3.6.3 Besucher 78
3.6.4 Interpreter 79
3.6.5 Iterator 81
3.6.6 Memento 82
3.6.7 Schablonenmethode 84
3.6.8 Strategie 85
3.6.9 Vermittler 86
3.6.10 Zustand 88
3.6.11 Zuständigkeitskette 89
3.7 Zusammenfassung und Bewertung 92

4 Prototypische Umsetzung 95
4.1 Verfahrensweise 95
4.2 Einschränkungen 97
4.3 Singleton 99
4.4 Interpreter 99
4.5 Kompositum 100
4.6 Darstellung 101
4.7 Fazit 103

5 Zusammenfassung und Ausblick 105

A Beschreibungen der Entwurfsmuster 109
A.1 Erzeugungsmuster 109
A.1.1 Abstrakte Fabrik 110
A.1.2 Erbauer 111
A.1.3 Fabrikmethode 112
A.1.4 Prototyp 112
A.1.5 Singleton 113
A.2 Strukturmuster 114
A.2.1 Adapter 114
A.2.2 Brücke 116
A.2.3 Dekorierer 117
A.2.4 Fassade 118
A.2.5 Fliegengewicht 119
A.2.6 Kompositum 120
A.2.7 Proxy 121
A.3 Verhaltensmuster 122
A.3.1 Befehl 122
A.3.2 Beobachter 123
A.3.3 Besucher 124
A.3.4 Interpreter 126
A.3.5 Iterator 127
A.3.6 Memento 128
A.3.7 Schablonenmethode 128
A.3.8 Strategie 129
A.3.9 Vermittler 130
A.3.10 Zustand 131

A.3.11 Zuständigkeitskette 132

B Notation der Klassendiagramme 133

C Quelltexte der Rational Rose Scripte 135

D Glossar 143

E Literaturverzeichnis 151

F Inhalt der CD 155

1 Einführung

In der Praxis werden Softwaresysteme nur ganz selten von Grund auf neu entwickelt; die Regel ist vielmehr, dass Softwaresysteme bereits im Einsatz sind [Balzert98]. Auf diese im Einsatz befindlichen Systeme kommen im Laufe ihres Lebens immer wieder Anpassungen, Änderungen und Weiterentwicklungen zu [Balzert98]. Bevor jedoch irgendwelche Änderungen an einer Software vorgenommen werden können, ist es unbedingt vonnöten, in einem ersten Schritt zu verstehen, wie das Softwaresystem funktioniert. Erst mit diesem Wissen kann eine Änderung sicher durchgeführt werden. Das Verstehen eines Softwaresystems wird durch folgende Probleme erschwert: eine Dokumentation existiert nicht mehr bzw. ist unzureichend, nur teilweise vorhanden oder veraltet. Dasselbe gilt für Spezifikationen oder Entwurfsmodelle. Zudem stehen die ursprünglichen Entwickler des Systems für Befragungen nicht mehr zur Verfügung. Letztlich bleibt nur der Quelltext als einzig zuverlässiges Dokument erhalten [Müller97]. Aber allein aus dem Quelltext eines Programms wirklich zu verstehen, wie ein System funktioniert und welche Ideen dahinterstecken, ist eine ermüdende und zeitraubende Angelegenheit, die oft nicht zur vollen Zufriedenheit gelöst werden kann – mit dem Resultat, dass nicht genau überprüfbar ist, ob vorgenommene Änderungen nicht doch irgendwelche unerwünschten Seiteneffekte mit sich bringen.

Entwurfsmuster (engl. *Design Patterns*) dienen dazu, Lösungen für bestimmte grundlegende Entwurfsprobleme bereitzustellen; somit profitieren junge unerfahrene Entwickler von den Erfahrungen der älteren, die schon viele Jahre in dem Bereich der Softwareentwicklung tätig sind. Jedes Entwurfsmuster vermittelt eine ganz bestimmte Idee, das heißt, es ist genau bekannt, welchen allgemeinen Zweck der Einsatz eines jeden Entwurfsmusters verfolgt, und auch, welche Vor- und Nachteile gerade dieser Entwurf hat.

Wäre nun stets aus dem Quelltext eines Softwaresystems ersichtlich, welche Entwurfsmuster enthalten sind, welche Klassen zu jedem Muster gehören und welche Klasse welche Rolle darin spielt, würde man zumindest bei den Teilen des Programms, die von Mustern gebildet werden, sehr schnell die dahinterstehende Idee begreifen. Die Realität bietet jedoch leider ein anderes Bild: bei der Anwendung der Musters wird die Information, dass es sich um ein Muster handelt, nicht im Quelltext verankert[1]. Das bedeutet, dass das Wissen um die Muster nicht mehr explizit sondern nur noch implizit vorhanden ist.

Für den Menschen ist es sehr schwer, von Hand in einem Softwaresystem nach Entwurfsmustern zu suchen. Daher ist es erstrebenswert, diese Arbeit von einem Software-Werkzeug ausführen zu lassen. In meiner Arbeit stelle ich dazu einen Ansatz vor, mit dem es möglich ist, die dreiundzwanzig Entwurfsmuster, wie sie in [Gamma+96] beschrieben sind, aus undokumentiertem C++ Quelltext zu extrahieren. Die Schwierigkeit besteht dabei in der nicht eindeutig festgelegten Struktur eines Musters, die ganz im Gegenteil sehr viel-

[1] Im einfachsten Fall kann dies durch Kommentare geschehen.

gestaltig auftreten kann. Mein Ansatz beruht darauf, an jedem Muster Merkmale festzustellen, die bei der Anwendung eines Musters in der Struktur auftreten müssen. Um die Zahl der falsch erkannten Muster zu verringern, definiere ich zudem Merkmale, die nicht vorkommen dürfen.

Die Entwurfsmuster aus [Gamma+96] wurden gewählt, weil es sich bei ihnen um Lösungen zu allgemeinen Problemen handelt. Solche Probleme können bei der Entwicklung jedes Softwaresystems auftreten, ganz egal wie speziell sein Bereich auch ist. Daher handelt es sich bei den Entwurfsmustern aus [Gamma+96] um die am häufigsten verwendeten Muster. Zudem waren die Autoren von [Gamma+96] die ersten, die Entwurfsmuster katalogisierten; an ihrer Darstellung orientieren sich alle anderen Beschreibungen zu Mustern. [Gamma+96] kann somit als Quasi-Standard betrachtet werden.

Diese Arbeit gliedert sich wie folgt: Zunächst gehe ich auf die Bedeutung des Programmverstehens innerhalb der Wartung und auf die Bedeutung und die Eigenschaften von Entwurfsmustern ein. Kapitel 2 gibt einen Überblick über den aktuellen Stand der automatischen Suche nach Entwurfsmustern und benennt die dabei vorhandenen Mängel. In Kapitel 3 stelle ich meinen eigenen Ansatz ausführlich vor, gebe Merkmale für alle Muster an und führe eine Bewertung im Vergleich zu den in Kapitel 2 vorgestellten Ansätzen vor. In Kapitel 4 befasse ich mich mit der prototypischen Umsetzung meines Ansatzes für einige ausgewählte Muster. Kapitel 5 schließt die Arbeit mit einer Zusammenfassung und einem Ausblick ab.

1.1 Wartung

Der Lebenszyklus eines Softwaresystems setzt sich aus der Analyse des Problembereiches, dem Entwurf des Systems, der Implementierung, dem Test und dem Vertrieb sowie der Wartung zusammen. Unter Wartung versteht man dabei alle Änderungen an einem Softwaresystem, die nach der Inbetriebnahme erfolgen [Müller97]. Dazu gehören sowohl Änderungen, die Fehler beseitigen, als auch Änderungen, die das Softwaresystem an neue Anforderungen anpassen.

Laut Studien und statistischen Untersuchungen nimmt die Wartung den größten Anteil der Gesamtkosten eines Softwaresystems ein. In Tabelle 1.1 sind die Ergebnisse einiger Untersuchungen zusammengefasst.

Studie	Wartungsanteil
de Rose und Nyman [deRose+78 zitiert nach Müller97]	60%-70%
Mills [Mills67 zitiert nach Müller97]	75%
Lientz und Swanson [Lientz+80 zitiert nach Müller97]	≥ 50%
Cashman und Holt [Cashman+80 zitiert nach Müller97]	80%
McKee [McKee84 zitiert nach Müller97]	65%-75%
Sentry [Sen80 zitiert nach Müller97]	63%
West [West93 zitiert nach Müller97]	40%

Tabelle 1.1: Ermittelte Wartungsanteile verschiedener Studien [Müller97]

Wie zu sehen ist, sprechen bis auf die letzte alle Studien von einem Wartungsanteil von mehr als 50%, die Studie von Cashman und Holt [CH80 zitiert nach Müller97] sogar von 80%. Aber selbst die optimistische Studie von West [Wes93 zitiert nach Müller97] beziffert den Anteil der Wartungskosten eines Softwaresystems immer noch auf 40%, was etwa den Kosten für Analyse, Design, Implementierung und Test zusammen entspricht [Müller97]!

Seit Bestehen der Informatik wurde stets versucht, durch bessere Sprachen, Konzepte, Methoden und Werkzeuge die Neuentwicklung von Software zu unterstützen [Balzert98]; dies vor allem auch mit dem Ziel, die in Software enthaltenen Fehler auf ein Minimum zu senken bzw. sogar fehlerfreie Software zu erstellen. Wäre das gelungen, hätte der Wartungsanteil stets geringer werden müssen, was aber nicht der Fall war. Gewiss brachten die Neuentwicklungen im Bereich der Softwaretechnik Verbesserungen mit sich, jedoch führten diese durch die steigende Komplexität der Software zu keiner absoluten Senkung der Fehlerrate. Schon seit längerem ist man sich einig, dass es unmöglich ist, Software vollkommen zu erstellen. Immer wieder wird es unvorhersehbare neue Anforderungen geben, die eine Anpassung erforderlich werden lassen – von den zu beseitigenden Fehlern ganz abgesehen. Das bedeutet, dass es Wartung immer geben wird; es geht nur darum, ihren Anteil an den Gesamtkosten auf ein Minimum zu reduzieren. Trotzdem bekannt ist, dass Wartungskosten die Entwicklungskosten übersteigen, wird diese Tatsache in der Forschung kaum berücksichtigt: Keine deutsche Hochschule besitzt einen Lehrstuhl für die Wartung von Software (Stand 1996/97); für die Software-Entwicklung jedoch gibt es viele. Anstatt also zu hoffen, die Software-Krise durch bessere Methoden zur Neuentwicklung von Software überwinden zu können, sollte man vielleicht lieber die Unvollkommenheit von Software anerkennen und der Forschung auf dem Gebiet der Software-Wartung ein größeres Maß an Aufmerksamkeit schenken [Müller97].

1.2 Programmverstehen

Wie im vorhergehenden Abschnitt gezeigt, macht die Wartung den größten Teil der Kosten eines Softwaresystems aus. Jeder Wartungsaktivität wie zum Beispiel der Beseitigung von Fehlern oder der Anpassung an neue Anforderungen geht zwangsläufig immer ein Verstehen des Systems voraus. Müller definiert diesen Verstehensprozess, nämlich das Programmverstehen, wie folgt: "Programmverstehen ist der aktive Vorgang des Erkennens der internen also technischen Arbeitsweise eines Programms." [Müller97] Zum Programmverstehen gehören Fragen wie: Welche Methode ruft welche Methode auf? Wo wird eine Variable X gelesen, wo geschrieben? Wie stehen verschiedene Klassen miteinander in Beziehung? usw. Das Programmverstehen ist nicht nur unabdingbare Voraussetzung für das Vornehmen von Änderungen an einem Softwaresystem sondern nimmt zugleich auch den größten Anteil bei der Wartung ein, was eine Studie von Fjeldstad und Hamlen [Fjeldstad+79 zitiert nach Müller97] belegt. Die Ergebnisse dieser Studie zeigt Tabelle 1.2.

Wartungsaktivität	**Anteil**
Verstehen der Änderungsanforderung	18%
Verstehen der Dokumentation	6%
Verstehen des Codes	23%
Implementierung	19%
Testen	28%
Dokumentation anpassen	6%

Tabelle 1.2: Wartungsaktivitäten nach Fjeldstad und Hamlen [Fjeldstad+79 zitiert nach Müller97]

Nimmt man die ersten drei Zeilen zusammen, ergeben sich 47% für das Programmverstehen, 25% für die Änderung (Code und Dokumentation) und 28% für den Test. Da das Programmverstehen also den größten Anteil innerhalb der Wartung ausmacht, die Wartung selbst aber den größten Teil im Gesamtlebenszyklus einnimmt, ist es offensichtlich, dass dem Programmverstehen eine herausgehobene Bedeutung zufällt. Um den Kostenanteil für die Wartung zu senken bzw. zu minimieren, sollte daher vor allem beim Hauptanteil Programmverstehen angesetzt werden, da eine Effizienzsteigerung hierbei die größten Einsparungen erwarten lässt.

Leider wird das Programmverstehen im Allgemeinen dadurch erschwert, dass eine Dokumentation oder auch Unterlagen über Spezifikation und Entwurf in der Regel nicht mehr vorhanden bzw. veraltet sind und als einzig zuverlässiges Dokument der Quelltext vorliegt. Das zentrale Problem dabei ist, dass das Modell explizit vorhandenes Wissen enthält, dieses Wissen jedoch bei der Implementierung verlorengeht bzw. nur noch implizit vorhanden ist [Neumann+99]. Beispielsweise ist allein aus dem Quelltext nicht ersichtlich, welche Ideen hinter allem stecken. Welche Überlegungen führten zu genau diesem Programm, wie

es vorliegt? Welche Vor- und Nachteile wurden dabei gegeneinander abgewägt? Im besten Fall sind Teile dieser verborgenen Informationen noch in Form von Kommentaren im Quelltext vorhanden [Neumann+99], aber allein aus dem mit Hilfe von Werkzeugen rückgewonnenen Modell sind diese Dinge nicht ersichtlich.

1.3 Entwurfsmuster

Allgemein gesprochen beschreibt ein Muster ein in unserer Umwelt beständig wiederkehrendes Problem sowie eine Lösung dafür, die bei jedem Auftreten des Problems stets wieder angewendet werden kann. Solche Muster, dass heißt, die Beschreibung eines Problems samt einer dazu passenden Lösung, finden sich in praktisch in allen Bereichen wieder. Christopher Alexander war der erste, der in der Architekturdomäne auf Muster hinwies. Seine Arbeiten wurden in viele weitere wissenschaftliche Bereiche übertragen, nicht zuletzt auch in den Entwurf objektorientierter Software, um den es hier gehen soll. 1995 erschien die erste bedeutende Sammlung von Mustern auf diesem Gebiet in dem Buch "Design Patterns – Elements of Reusable Design" von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. Die deutsche Fassung ihres Musterkataloges kam 1996 mit dem Titel "Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software" [Gamma+96] heraus.

Was genau ist ein Entwurfsmuster (engl. *Design Pattern*)? Selbstverständlich trifft allgemeine obige Erklärung für Muster auch auf Entwurfsmuster[2] in der Software-Entwicklung zu, aber Entwurfsmuster lassen sich konkreter beschreiben. Gamma, Helm, Johnson und Vlissides definieren Entwurfsmuster in [Gamma+96] als *"Beschreibung zusammenarbeitender Objekte und Klassen, die maßgeschneidert sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen."* Die Lösung für ein bestimmtes objektorientiertes Entwurfsproblem wird also durch eine Anzahl von Klassen und Objekten erbracht, die auf eine definierte Art und Weise zusammenarbeiten. Die wichtigsten Elemente eines Entwurfsmusters, wie sie auch in [Gamma+96] beschrieben werden, sind der Name des Musters, die Beschreibung des Problems, die Lösung dafür und die aus dieser Lösung resultierenden Konsequenzen, das heißt die Vor- und Nachteile, die diese Lösung besitzt. Der Name benennt das Problem und die Lösung mit ein oder zwei Worten. Er ist besonders wichtig, da sich mit seiner Hilfe Entwickler auf einem höheren Abstraktionsniveau über ihre Entwürfe austauschen können. Der Problemabschnitt erläutert, in welchen möglichen Fällen das Muster angewendet werden kann. Die Lösung beschreibt die Klassen und Objekte sowie deren Beziehungen und Interaktionen, aus denen schließlich der Entwurf für das Problem bestehen sollte. Im Konsequenzabschnitt erfolgt eine Auflistung der Vor- und Nachteile, die der zuvor beschriebene Entwurf mit sich bringt.

[2] Mit Entwurfsmustern sind in dieser Arbeit stets Entwurfsmuster in der objektorientierten Software-Entwicklung gemeint.

In [Gamma+96] werden dreiundzwanzig solcher objektorientierter Entwurfsmuster beschrieben; das Buch stellt zum größten Teil einen Katalog für Entwurfsmuster dar. Die darin enthaltenen Muster beschreiben keine neuartigen Entwürfe, sondern Entwürfe, die in der Vergangenheit von vielen unabhängig voneinander arbeitetenden Entwicklern erdacht und benutzt worden sind. Diese Entwickler standen jeweils vor mehr oder minder denselben Problemen und kamen unabhängig voneinander auf ähnliche Lösungen. Bei den in [Gamma+96] beschriebenen Entwurfsmustern handelt es sich um ein Analyseprodukt diverser Softwaresysteme, das heißt um ein Resultat des *Reverse Engineering* [Niere+01]. Neben den beschriebenen Mustern fanden Gamma, Helm, Johnson und Vlissides noch weitere Muster, die nicht mit in den Katalog aufgenommen wurden, da sie sie für nicht essentiell genug hielten bzw. sie in der Vergangenheit zu selten angewendet wurden. Bei den Mustern jedoch, die sich im Katalog befinden, handelt es sich dagegen um Entwürfe, die in der Vergangenheit vielfach angewendet wurden und bei denen damit zu rechnen ist, dass sie auch in der Zukunft oft benutzt werden, erst recht seit der Veröffentlichung von [Gamma+96]. Das heißt also, dass die beschriebenen Entwurfsmuster schon vor dem Erscheinen von "Design Patterns – Elements of Reusable object-oriented Design" existierten und angewendet wurden, es für sie jedoch noch keine Namen und Beschreibungen gab – sie waren nur unbewusst vorhanden. Durch die Dokumentation der Muster in [Gamma+96] können unerfahrene Entwickler von den Erfahrungen der älteren Entwickler profitieren, denn wenn sie irgendwann vor einem der stets wiederkehrenden Probleme im objektorientierten Entwurf stehen, müssen sie sich nicht erneut eine passende Lösung dafür ausdenken, wie so viele vor ihnen, sondern sie brauchen im Grunde nur die Musterlösung im Katalog nachzuschlagen[3]. Eine weitere Besonderheit der in [Gamma+96] beschriebenen Muster liegt in der Beschäftigung mit Lösungen ganz elementarer Probleme. Solche Muster können sich in jedem Softwaresystem befinden, ganz egal wie speziell der Bereich ist, für den es programmiert wurde[4].

Die in [Gamma+96] vorgestellten Entwurfsmuster sind in erster Linie für den objektorientierten Entwurf gedacht. Die Beschreibungen bestehen demnach aus Elementen wie Klassen, Objekten und Vererbung, die ausschließlich im objektorientierten Umfeld zu finden sind. Dennoch ist es möglich, diese Muster wenigstens teilweise bzw. in abgeänderter Form auch im funktionalen sowie prozeduralen Programmierparadigma zu nutzen.

Die besondere Bedeutsamkeit von Entwurfsmustern liegt in unserem Fall darin, dass jedes Muster eine ganz bestimmte Idee transportiert. Muster können sich in ihrer Struktur gleichen, sie können auch einen ähnlichen Zweck verfolgen, aber in ihrer Idee sind doch alle verschieden. Zur Idee wird in dieser Arbeit das Muster in seiner Gesamtheit gezählt, das

[3] Dabei handelt es sich jedoch um keine triviale Angelegenheit. Bis man ein Muster wirklich in seinem vollen Umfang begreift und es sicher anwenden kann, vergeht mitunter sehr viel Zeit.

[4] Aus diesem Grunde nehmen alle Arbeiten, die sich mit dem Auffinden von Entwurfsmustern aus einem Quellprogramm oder UML-Diagramm beschäftigen (siehe Kapitel 2), ausschließlich auf die Muster aus [Gamma+96] Bezug.

heißt sowohl Zweck und Struktur als auch die Konsequenzen und die sonstigen Beschreibungen.

An dieser Stelle wären an sich kurze Erläuterungen aller Muster aus [Gamma+96] angebracht. Um den Lesefluss jedoch nicht zu unterbrechen, wurden diese in den Anhang A gestellt. Es handelt sich dabei um recht kompakte Darstellungen, die jemandem, der sich zum ersten Mal mit Entwurfsmustern beschäftigt, sehr wahrscheinlich nicht genügen, um zu einem befriedigenden Verständnis der Muster zu gelangen. Für diejenigen sei deshalb auf die Originalliteratur [Gamma+96] verwiesen, die bedeutend mehr Informationen bereit hält als der genannte Anhang A.

In [Gamma+96] wird jedes Muster einer bestimmten Kategorie zugeordnet. Zu diesen Kategorien zählen die *Erzeugungsmuster*, *Strukturmuster* und *Verhaltensmuster*. *Erzeugungsmuster* dienen der Erzeugung von Objekten, wobei der Erzeugungsprozess versteckt wird. *Strukturmuster* befassen sich mit der Komposition von Klassen und Objekten, um größere Strukturen zu bilden. *Verhaltensmuster* schließlich beschäftigen sich mit Algorithmen und der Zuweisung von Zuständigkeiten an Objekte. Ausführlichere Beschreibungen dieser Kategorien finden sich ebenfalls im Anhang A.

1.4 Über die Bedeutung von Entwurfsmustern beim Programmverstehen

Was haben Entwurfsmuster mit Programmverstehen zu tun? Wie im letzten Abschnitt gezeigt, verbirgt sich hinter jedem Entwurfsmuster eine klare Idee, das heißt, es ist genau bekannt, welchen allgemeinen Zweck der Einsatz eines jeden Entwurfsmusters verfolgt, und auch, welche Vor- und Nachteile gerade dieser Entwurf mit sich bringt. Jeder der ein beliebiges Muster kennt und ein Vorkommen dieses Musters verstehen soll, wird sehr schnell erfassen, welche Aufgabe die Klassen, die zu diesem Muster gehören, erfüllen, das heißt, er erkennt die Idee, die hinter diesem Entwurf steckt. Diese Idee allein aus einem rückgewonnenen Klassendiagramm ableiten zu wollen, ist eine sehr schwierige Angelegenheit, da die Idee nicht unbedingt etwas ist, das man allein über die Klassenstruktur ausdrücken kann [Niere+01]. Die Klassenstrukturen der Muster STRATEGIE, BEFEHL und ZUSTAND sind sogar beinahe identisch, jedoch ist die Bedeutung jedes dieser Muster eine völlig andere. Prechelt, Unger und Philippsen belegen in [Prechelt+97] sogar durch ein Experiment die offensichtliche Tatsache, dass das Wissen um Entwurfsmuster in einem Softwaresystem zu einem schnelleren und besseren Verständnis des Systems führt.

Um zu beschreiben und zu dokumentieren, welche Ideen hinter einem bestimmten Entwurf stecken, sind ausführliche Erklärungen nötig – in der Praxis hält sich in der Regel damit niemand auf, zumal es auch eine mühsame Angelegenheit ist, die viel Zeit in Anspruch nimmt. Da die Idee in der Struktur des Musters steckt, reicht das Erkennen des Musters und damit die Referenz auf die Beschreibung des Musters als Dokumentation aus. Dem

Software-Entwickler verbleibt es zu vermerken, welche Klassen zu welchem Muster gehören und welche Rollen sie darin einnehmen, um die oft umfangreiche Bedeutung eines Programmteiles, den ein Muster bildet, zu erklären. Durch die Muster wird also ein höheres Abstraktionsniveau erreicht, das zwar ein größeres Wissen voraussetzt, nämlich das genaue Wissen um die Muster, das aber schließlich zu einer enormen Steigerung der Effizienz führt.

Ideal wäre es, wenn ein Programm aus vielen ineinander verzahnten Entwurfsmustern bestünde[5] [Gamma+96]. Solch ein Programm hätte nur noch sehr wenige Erläuterungen nötig. In der Praxis kommt dieser Fall aber leider so gut wie nie vor.

Nach [Krämer+96] können Muster auch auf bestimmte Tatsachen hinweisen: Das ADAPTER-Muster signalisiert beispielsweise, dass Klassen in verschiedenen Kontexten verwendet werden, in denen sie jeweils unterschiedliche Schnittstellen benötigen. Das Brücke-Muster wiederum trennt eine Schnittstelle von ihrer Implementierung, so dass beide unabhängig voneinander verändert werden können – das deutet darauf hin, dass bei dem Entwurf dieses Bereiches im Programm mit vielen Veränderungen und Wiederverwendung gerechnet wurde. Das PROXY-Muster schließlich lässt die Vermutung zu, dass hier sehr große Objekte gehandhabt werden oder dass es teuer ist, diese Objekte zu erzeugen.

Es ist also leicht einzusehen, dass das Wissen um Entwurfsmuster in einem Programm sehr bedeutend für das Verständnis des Programms ist.

1.5 Schwierigkeiten bei der manuellen Suche nach Entwurfsmustern

Der vorhergehende Abschnitt zeigte, welche Bedeutung Entwurfsmuster beim Programmverstehen besitzen. Für den Menschen ist es jedoch sehr schwierig, in einem großen Softwaresystem nach Mustern zu suchen, da ein Entwurfsmuster in vielerlei Gestalt auftreten kann [Niere+01]. Wieviele *KonkreteFabriken* beispielsweise eine Instanz des ABSTRAKTE-FABRIK-Musters besitzt, wieviele *Produkte* darin erzeugt werden und in welcher Vererbungsbeziehung diese zueinander stehen, wird durch das Muster nicht festgelegt. Es ist nicht an eine feste Struktur gebunden. Die Vielgestaltigkeit der Muster wäre an sich kein Problem, würde sich im Quelltext eine Dokumentation der verwendeten Muster wiederfinden. Jedoch bieten gängige Programmiersprachen wie Java, Smalltalk und C++ keine Unterstützung, um das Vorhandensein von Entwurfsmustern im Quelltext zu verankern. Die einzige Möglichkeit der Dokumentation liegt somit in der Verwendung von Kommentaren. Davon, dass von dieser Möglichkeit viel Gebrauch gemacht wird, kann allerdings nicht ausgegangen werden. Beachtung muss zudem die Zeit vor dem Erscheinen von [Gamma+96] finden, in der die in den Mustern behandelten Konzepte ohne das Wissen um die

[5] Gamma, Helm, Johnson und Vlissides zielen auf die gesteigerte Qualität eines Entwurfs ab, der Muster nutzt, jedoch gilt dieser Ansatz das Programmverstehen ganz genauso.

Muster in [Gamma+96] angewendet wurden. Die genannten Tatsachen machen deutlich, dass bei der Suche nach Entwurfsmustern grundsätzlich auf keine Dokumentation der enthaltenen Muster zurückgegriffen werden kann.

Aufgrund eigener Erfahrung kann ich sagen, dass es bereits bei einem kleinen Programm mit nur ein paar Klassen keine triviale Aufgabe ist, in seinem Quellcode nach Mustern zu suchen. Andere Autoren [Bansiya98, Keller+99, Bergenti+00] untermauern diese Tatsache durch gleichlautende Feststellungen. Keller, Schauer, Robitaille und Pagé vertreten sogar die Ansicht, dass die manuelle Suche nach Entwurfsmustern nicht durchführbar ist, selbst wenn diese Suche durch Werkzeuge[6] unterstützt wird.

1.6 Problemstellung

In den vorangegangenen Abschnitten wurde belegt, dass die Wartungskosten innerhalb des Softwarelebenszyklus rund 50% betragen. Ferner wurde nachgewiesen, dass das Programmverstehen wiederum innerhalb der Wartung ebenfalls etwa 50% ausmacht. Aufgrund dieser hohen Bedeutung, die dem Programmverstehen zukommt, lassen Effizienzsteigerung in diesem Bereich die höchsten Kosteneinsparungen vermuten.

Entwurfsmuster sind einfache und elegante Lösungen für Probleme des objektorientierten Softwareentwurfs. Jedes Muster transportiert eine bestimmte Idee, die hinter dem vorliegenden Musterentwurf steht. Prechelt, Unger und Philippsen wiesen in [Prechelt+97] experimentell nach, dass die Kenntnis enthaltener Entwurfsmuster zu einem schnelleren und besseren Verständnis eines Softwaresystems führt. Voraussetzung dafür ist und bleibt eine genaues Wissen über die Muster, dass heißt, man muss verstanden haben, welchen Zweck ein Muster verfolgt, welche Klassen und Objekte daran beteiligt sind, welche Vor- und Nachteile diese Lösung mit sich bringt. Ein Muster, das der Software-Ingenieur nicht kennt, trägt in keiner Weise zum Programmverstehen bei.

Für den Menschen ist es sehr schwierig, Entwurfsmuster manuell in einem Softwaresystem zu identifizieren. Daher ist es erwünscht, diese Arbeit automatisch von einem Computerprogramm ausführen zu lassen. Das folgende Kapitel 2 stellt derzeitige Ansätze zur automatisierten Suche von Mustern in Quellcode vor.

[6] Hierbei handelt es sich nicht um Werkzeuge, die spezielle Techniken für die Suche nach Entwurfsmustern implementieren.

2 Die automatische Suche nach Entwurfsmustern – Aktueller Stand

Im vergangenen Kapitel wurde erklärt, dass Entwurfsmuster beim Programmverstehen eine herausragende Bedeutung innehaben. Es wurde festgestellt, dass es für den Menschen sehr schwierig ist, in einem Quellprogramm nach Mustern zu suchen und es deshalb nötig ist, diese Arbeit von einem Computerprogramm ausführen zu lassen.

Dieses Kapitel stellt die bislang zu diesem Thema veröffentlichten Arbeiten vor, erläutert den verfolgten Ansatz und – sofern vorhanden – das auf diesem Ansatz basierende entwickelte Software-Werkzeug und nennt abschließend die damit erzielten Ergebnisse. Alle Arbeiten nehmen ausschließlich auf die aus [Gamma+96] stammenden dreiundzwanzig Entwurfsmuster Bezug; andere Muster werden nicht betrachtet.

Um über die Leistungsfähigkeit eines Software-Werkzeuges zum Auffinden von Entwurfsmustern wirklich eine fundierte und repräsentative Aussage treffen zu können, sind einige zusammenhängende Untersuchungen nötig. Zunächst sei gesagt, dass bei der Suche nach einem Muster vier verschiedene Fälle zu unterscheiden sind:

1. Fall: *positive true* – Das Muster wurde von dem Werkzeug erkannt, es ist tatsächlich eine Instanz des gesuchten Musters. Dies ist der anzustrebende Fall.

2. Fall: *positive false* – Das Muster wurde erkannt, aber es stellt sich heraus, dass es sich bei der gefundenen Struktur nicht um eine Instanz des gesuchten Musters handelt.

3. Fall: *negative true* – Das Muster wurde nicht erkannt, jedoch ist es tatsächlich in dem untersuchten System enthalten. Dieser Fall ist unbedingt zu vermeiden.

4. Fall: *negative false* – Das Muster wurde nicht erkannt, es steckt auch nicht im System.

Auf Basis dieser vier Fälle lassen sich weitere Metriken ableiten, mit deren Hilfe schließlich eine Gesamtaussage über die Leistungsfähigkeit des Werkzeuges formuliert werden kann:

Recall: Dieser Wert bezeichnet das Verhältnis zwischen der Anzahl der im untersuchten Softwaresystem tatsächlich enthaltenen Muster zu der Anzahl der erkannten Mustervorkommen. Ein *Recall* von 100% bedeutet also, dass zumindest alle enthaltenen Muster gefunden wurden, eventuell auch mehr; aber übersehen wurde keines (Fall 3: *negative true*).

Präzision: Die Präzision gibt das Verhältnis zwischen der Anzahl der erkannten Muster, die tatsächlich existent sind (Fall 1: *positive true*) zu der Anzahl der insgesamt gefundenen Muster an (Addition von Fall 1: *positiv true* und Fall 2: *negative true*). Eine Präzision von

50% bedeutet demnach, dass die Hälfte aller vom Werkzeug erkannten Muster in Wirklichkeit keine sind.

Zu beachten ist hierbei, dass der Wert für die Präzision nie allein betrachtet werden kann, sondern stets der Fall 3: *negative true* mit einzubeziehen ist. Es dürfte nämlich, übertrieben gesehen, relativ leicht sein, ein Werkzeug zu entwickeln, das für jedes zu untersuchende Softwaresystem eine Präzision von nahezu 100% erbringt[7], jedoch eine bedeutende Anzahl von Mustervorkommen übersieht (ausgedrückt durch den Fall 3: *negative true*).

Günstig wäre zudem der Test des Werkzeuges an einer Art Referenzsystem, das viele Muster enthält, die in einer ausgewählten Weise implementiert wurden. Außerdem müsste es eine Dokumentation dazu geben, in der alle enthaltenen Muster samt ihrem im System befindlichen Ort aufgeführt sind, da nur so eine Beurteilung der Ergebnisse erfolgen kann. Im Grunde müssten alle Werkzeuge an einem solchen Referenzsystem mit den vorgestellten Metriken gemessen werden, um eine repräsentative Aussage zu erhalten.

An dieser Stelle sei noch erwähnt, dass die automatische Suche nach Entwurfsmustern nicht nur für das Programmverstehen von Bedeutung ist, sondern sich auch auf das *Refactoring* von Altsystemen (engl. *legacy systems*) beziehungsweise auf die Neuentwicklung von Softwaresystemen ausweiten lässt. Hierbei ist es möglich, die falsche Anwendung eines Entwurfsmusters zu korrigieren. Desweiteren kann ein Software-Werkzeug zum Auffinden von Entwurfsmustern dazu verwendet werden, die korrekte Anwendung eines Musters zu überprüfen; wird ein verwendetes Muster beispielsweise bei einem Suchlauf nicht erkannt, hat der Software-Entwickler höchstwahrscheinlich etwas falsch gemacht. Zudem führt die Erkennung eines Musters, das der Entwickler nicht anzuwenden geplant hatte, zu einem besseren Verständnis des Entwurfs [Bergenti+00].

Die zur automatischen Mustersuche gemachten und im Folgenden vorgestellten Ansätze lassen sich in verschiedene Kategorien einteilen: Eine Gruppe von Arbeiten verwendet für die Suche bestimmte **minimale Schlüsselstrukturen**, die bei einer Anwendung eines Musters stets vorkommen müssen (*DP++* [Bansiya98], *KT* [Brown96], *SPOOL* [Keller+99]). Eine weitere Gruppe sucht nach **vollständigen Übereinstimmungen in der Klassenstruktur** der Muster (*Pat* [Krämer+96], *IDEA* [Bergenti+00], *Mehrstufiger Suchprozess* [Antoniol+98]). Zu den verbleibenden Kategorien gehört jeweils nur eine Arbeit: In [Niere+01] werden eine **flexible Musterdefinition und Fuzzylogik** eingeführt, um der Vielgestaltigkeit der Muster zu begegnen. Kim und Boldyreff [Kim+00] verwenden für die Realisierung ihres *Pattern Wizard* **Metriken** bei der Suche. *BACKDOOR* [Shull+96] von Shull, Melo und Basili stellt ein **induktive Methode für die manuelle Suche** dar, die jedoch von Software-Werkzeugen unterstützt werden kann.

[7] Dies geschieht durch eine Ausstattung der Suchanfrage mit hohen Restriktionen, das heißt, alle in [Gamma+96] genannten Merkmale eines Musters werden haargenau überprüft. Damit würden jedoch nur eine kleine Menge an Mustern gefunden werden, da, wie oben beschrieben, ein Muster sehr vielgestaltig daherkommen kann.

2.1 Suche nach minimalen Schlüsselstrukturen

Dieser Ansatz beruht darauf, für jedes Muster bestimmte minimale Schlüsselstrukturen festzulegen, die notwendigerweise bei der Anwendung eines Musters immer vorhanden sein müssen. Anhand dieser Merkmale ist es dann möglich, nach einem Muster zu suchen. Zum Beispiel legen Bansiya und Brown als Merkmal für die Suche nach dem KOMPOSITUM-Muster eine 1-zu-n-Aggregationsbeziehung von einer Kindklasse zu einer seiner Elternklassen fest. Ob zusätzlich verschiedene *Blatt*-Klassen vorhanden sind, wird dabei nicht berücksichtigt, da es Vorkommen des KOMPOSITUMS geben kann, die mehrere oder auch gar keine *Blatt*-Klasse besitzen; eine 1-zu-n-Aggregationsbeziehung der *Kompositum*-Klasse zur *Komponenten*-Klasse ist jedoch allen Varianten gemein. Als ein weiteres Beispiel liegt der Suche nach dem DEKORIERER-Muster eine 1-zu-1-Aggregationsbeziehung von einer Kindklasse zu einer seiner Elternklassen zugrunde. Das Klassendiagramm in Abbildung 2.1 veranschaulicht dies. Die grau dargestellten Elemente bilden die Minimalstruktur.

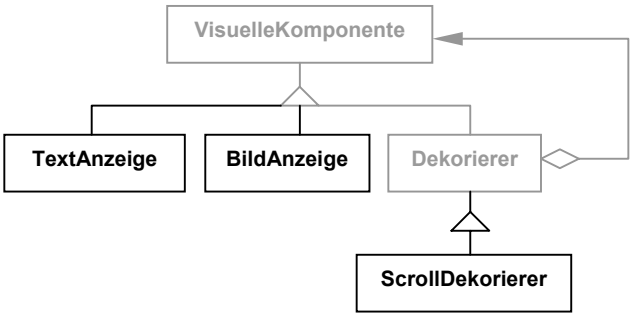


Abbildung 2.1: minimale Schlüsselstruktur des Dekorierer-Musters

Sooft also eine Kindklasse zu einer seiner Elternklassen eine 1-zu-1-Aggregationsbeziehung besitzt, wird ein Vorkommen des DEKORIERER-Musters gemeldet.

Es gibt drei Arbeiten, die den eben erläuterten Ansatz verfolgen: *DP++* von Bansiya, *KT* von Brown und *SPOOL* von Keller, Schauer, Robitaille und Pagé.

2.1.1 DP++

Jagdish Bansiya stellt in [Bansiya98] sein Software-Werkzeug *DP++* zum Auffinden von Entwurfsmustern in C++ Quellcode vor.

Neben dem KOMPOSITUM- und dem DEKORIERER-Muster gibt Bansiya auch zum ADAPTER-Muster eine Suchstrategie an; er führt aus, dass er Merkmale für BRÜCKE, FASSADE, FLIEGENGEWICHT, SCHABLONENMETHODE und ZUSTÄNDIGKEITSKETTE gefunden hat, verzichtet jedoch auf eine Erläuterung dieser Merkmale. Die anderen Muster werden von ihm nicht erwähnt. Die Strukturelemente, aus denen sich die Merkmale eines Musters zusammensetzen können, also die Elemente, die Bansiyas Software-Werkzeug *DP++* unterscheiden kann, sind:

- abstrakte und konkrete Klassen,
- Vererbungsbeziehungen,
- Aggregationsbeziehungen (ausgedrückt durch explizite Referenzen),
- einfache Assoziationsbeziehungen (ausgedrückt etwa durch Methodenparameter),
- Templates,
- Polymorphie und
- Methodenaufrufe.

Bansiyas Software-Werkzeug *DP++* leistet das automatische Auffinden von Entwurfsmustern und die anschließende Visualisierung der gefundenen Muster. *DP++* besteht aus drei Subsystemen, wie in Abbildung 2.2 zu sehen ist.

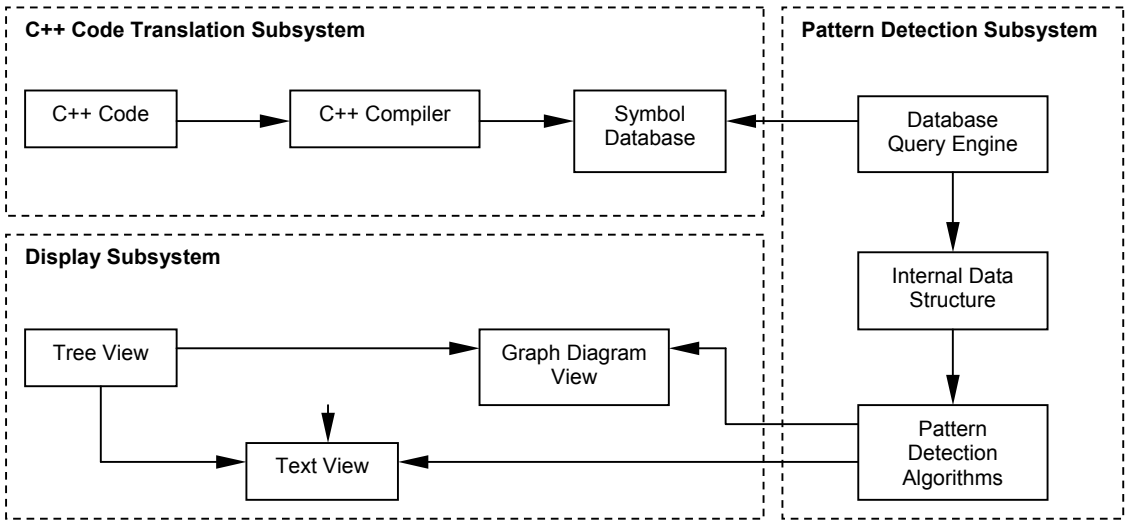


Abbildung 2.2: DP++ [Bansiya98]

Das **C++ Code Translation Subsystem** analysiert den C++ Quelltext, das **Pattern Detection Subsystem** ist für das Auffinden der Entwurfsmuster zuständig, und das **Display Subsystem** sorgt für die graphische Darstellung der Ergebnisse.

Die Leistungsfähigkeit von *DP++* testete Bansiya an mehreren Softwaresystemen, deren Umfang zwischen 30 und 400 Klassen lag. Konkrete Aussagen über *Recall* und Präzision sowie über nicht gefundene Mustervorkommen werden von ihm jedoch nicht gemacht.

2.1.2 KT

Kyle Brown stellt in [Brown96] die Ergebnisse seiner Studie zum Auffinden von Entwurfsmustern in Smalltalk Quellcode vor. Neben den beiden schon genannten Mustern KOMPOSITUM und DEKORIERER benennt Brown Merkmale für ZUSTÄNDIGKEITSKETTE, SCHABLONENEMETHODE, BEFEHL, ZUSTAND und STRATEGIE. Alle anderen Muster werden nicht erwähnt bzw. im Falle des INTERPRETER-Musters als unauffindbar erklärt.

Als praktische Umsetzung seines Ansatzes entwickelte Brown das Software-Werkzeug *KT*, mit dem es möglich ist, Entwurfsmuster in Smalltalk-Programmen zu finden. Für die Realisierung erfolgt nicht wie bei Bansiya eines Analyse des Quellcodes, sondern Brown stützt sich auf die besonderen Meta-Level-Eigenschaften von Smalltalk. Zum Auffinden der Muster wird zunächst aus dem zu untersuchende Smalltalk-Programm eine Darstellung als Klassendiagramm und eine Darstellung als Sequenzdiagramm gewonnen. Auf diese Diagramme erfolgen dann die Anfragen auf mögliche enthaltene Muster. Beide Modellierungskonstrukte sind für die Erkennung unterschiedlicher Muster nützlich. Das statische Klassendiagramm beispielsweise eignet sich für die Erkennung des KOMPOSITUM- und DEKORIERER-Musters; das Sequenzdiagramm, das dynamisches Verhalten repräsentiert, dagegen wird für die Erkennung des ZUSTÄNDIGKEITSKETTE-Musters verwendet.

Für die graphische Darstellung der Ergebnisse, das heißt der erkannten Muster, generiert *KT* ein *Rational Rose Petal File* – ein Dateiformat, das UML-Modelle von Softwaresystemen speichert –, das mit *Rational Rose* gelesen und angezeigt werden kann.

Um die Fähigkeiten von *KT* zu testen, wurden vier Softwaresysteme damit untersucht: System A (62 Klassen), System B (264 Klassen), System C (46 Klassen - *KT* selbst), System D (40 Klassen)[8]. Brown berichtet von gefundenen Vorkommen des KOMPOSITUM-, des DEKORIERER- und des SCHABLONENMETHODE-Musters. Angaben zu *Recall* und Präzision werden jedoch auch hier nicht gemacht.

[8] Die tatsächlichen Namen der getesteten Softwaresysteme dürfen aus rechtlichen Gründen nicht genannt werden. Daher wählte Brown die vorliegenden Synonyme.

2.1.3 SPOOL

Die folgenden Erläuterungen sind [Keller+99] entnommen. Keller, Schauer, Robitaille und Pagé geben Merkmale für das BRÜCKE-, das FABRIKMETHODE- und das SCHABLONENMETHODE-Muster an, da diesen beim Verstehen eines Frameworks eine hervorgehobene Stellung zukommt [Keller+99]. Die übrigen Muster werden von ihnen nicht erwähnt. Für die Umsetzung ihres Ansatzes entwickelten sie das Software-Werkzeug *SPOOL* (Spreading Desirable Properties into the Design of Object-oriented, Large-scale Software Systems).

Die Suche nach Entwurfsmustern anhand von Schlüsselmerkmalen soll im Folgenden noch am Beispiel des SCHABLONENMETHODE-Musters erklärt werden.

Gegeben seien zwei Klassen: eine Klasse *Angestellter* und eine von ihr abgeleitete Klasse *Chef*.

```
class Angestellter
{
public:
    ZeigeGehalt();
    virtual int BerechneGrundgehalt();
    virtual int BerechneZuschlag();
}

class Chef : public Angestellter
{
public:
    virtual int BerechneGrundgehalt();
    virtual int BerechneZuschlag();
}
```

Angestellter implementiert eine Methode *ZeigeGehalt*, die das Gehalt eines Angestellten ausgibt. Sie verwendet zu ihrer Berechnung die Methoden *BerechneGrundgehalt* und *BerechneZuschlag*.

```
Angestellter::ZeigeGehalt()
{
    int Gehalt=0;
    Gehalt=Gehalt+BerechneGrundgehalt();
    Gehalt=Gehalt+BerechneZuschlag();
    printf("Gehalt: %d\n", Gehalt);
}

int Angestellter::BerechneGrundgehalt()
{
    int Grundgehalt=10000;
    return Grundgehalt;
}

int Angestellter::BerechneZuschlag()
{
    int Zuschlag=0;
    return Zuschlag;
}
```

Die Klasse *Chef* überschreibt die Methode *ZeigeGehalt* nicht sondern nur *BerechneGrundgehalt* und *BerechneZuschlag*, da ein Chef ein anderes Grundgehalt und einen anderen Zuschlag bezieht als ein Angestellter.

```
int Chef::BerechneGrundgehalt()
{
    int Grundgehalt=20000;
    return Grundgehalt;
}

int Chef::BerechneZuschlag()
{
    int Zuschlag=5000;
    return Zuschlag;
}
```

Dieses kleine Programm ist ein Beispiel für das SCHABLONENMETHODE-Muster. *SPOOL* arbeitet nun folgendermaßen: Es traversiert alle Klassen, untersucht dabei jede Methode jeder Klasse auf lokale Methodenaufrufe und überprüft anschließend, ob die aufgerufenen lokalen Methoden polymorph sind. Treffen alle Bedingungen zu, so wird der Fund einer SCHABLONENMETHODE ausgegeben. Bezüglich des kleinen Beispiels würde *SPOOL* also zuerst die Klasse *Angestellter* überprüfen und dabei feststellen, das *ZeigeGehalt* Aufrufe der lokalen Methoden *BerechneGrundgehalt* und *BerechneZuschlag* besitzt. Da *BerechneGrundgehalt* und *BerechneZuschlag* polymorph sind (`virtual`-Deklaration), wurde eine Vorkommen des SCHABLONENMETHODE-Musters gefunden. Die Untersuchung der Methoden *BerechneGrundgehalt* und *BerechneZuschlag* bringt sowohl in der Klasse *Angestellter* als auch in der Klasse *Chef* kein positives Ergebnis, da keine lokalen Methodenaufrufe enthalten sind.

SPOOL lässt neben der erläuterten Standardabfrage auch Abweichungen zu; zum Beispiel kann eingestellt werden, dass die lokalen primitiven Methoden abstrakt sein müssen, oder dass es notwendig ist, dass sie von mindestens einer Unterklasse überschrieben werden.

In das *SPOOL*-System integriert sind diverse kleinere Software-Werkzeuge, die jeweils speziell für die Quellcode-Analyse, für die Suche nach Entwurfsmustern und für die graphische Darstellung des zu untersuchenden Softwaresystems und der gefundenen Muster zugeschnitten sind. Abbildung 2.3 zeigt das *SPOOL*-System. Die kursiv geschriebenen Namen am linken und rechten Rand benennen die erwähnten integrierten Werkzeuge.

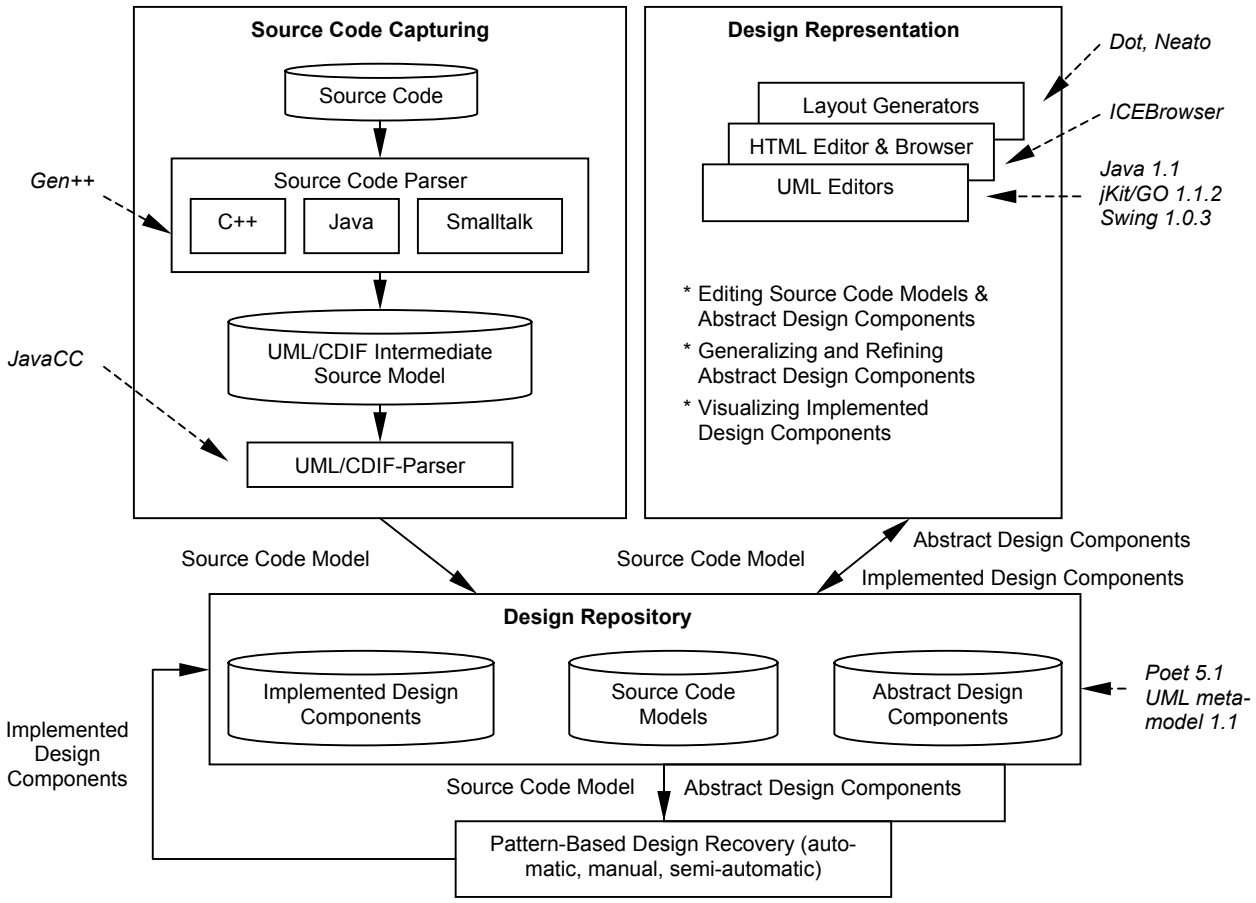


Abbildung 2.3: SPOOL [Keller+99]

Für die Untersuchung eines Softwaresystems wird zunächst der Quellcode analysiert[9] und daraus ein UML-Modell gewonnen, welches im **Design-Repository** abgelegt wird. Hierauf erfolgt dann die Anfrage nach Mustervorkommen. Das **Design-Representation**-Modul stellt anschließend die Ergebnisse graphisch sehr aufwendig dar, um den Software-Ingenieur so gut wie möglich beim Verstehen des Softwaresystems zu unterstützen.

Keller, Schauer, Robitaille und Pagé testeten die Leistungsfähigkeit von *SPOOL* an drei großen Systemen: System A (3103 Klassen), System B (1420 Klassen)[10] und *ET++* (722 Klassen), wobei, wie oben bereits erwähnt, nur SCHABLONENMETHODEN, FABRIKMETHODEN und BRÜCKEN Beachtung fanden. Die Ergebnisse sind in Tabelle 2.1 dargestellt.

[9] Zur Zeit wird nur C++ durch den Parser *Gen++* unterstützt.

[10] Aus rechtlichen Gründen wurden wiederum Synonyme verwendet.

	System A	System B	ET++
Schablonenmethode	3,243	1,857	1,022
Fabrikmethode	247	168	44
Brücke	108	95	46

Tabelle 2.1: mit SPOOL gefundene Muster [Keller+99]

Zu *Recall*- und Präzisionsrate machten Keller, Schauer, Robitaille und Pagé keine Angaben.

2.2 Suche nach vollständigen Übereinstimmungen in der Klassenstruktur

Im Gegensatz zur ersten Gruppe der Ansätze sucht diese zweite Gruppe nicht nach minimalen Schlüsselstrukturen sondern nach vollständigen Übereinstimmungen in der Klassenstruktur. Die für die Suche verwendeten Klassenstrukturen entsprechen weitestgehend den Abbildungen aus [Gamma+96]. Dazu ein Beispiel: Das KOMPOSITUM-Muster besitzt folgende vereinfacht dargestellte Struktur (Abbildung 2.4):

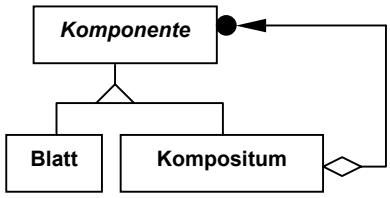


Abbildung 2.4: vereinfachte Struktur des Kompositum-Musters

Nach der vollständigen Übereinstimmung dieser Struktur zu suchen, bedeutet, dass jedesmal, wenn eine Klasse wenigstens zwei Unterklassen besitzt und eine dieser Unterklassen eine 1-zu-n-Aggregationsbeziehung zurück zur Elternklasse führt, ein Vorkommen des KOMPOSITUM-Musters gemeldet wird.

Drei Arbeiten verfolgen den eben dargestellten Ansatz: *Pat* von Krämer und Prechelt, *I-DEA* von Bergenti und Poggi und der *Mehrstufige Suchprozess* von Antoniol, Fiutem und Cristoforetti.

2.2.1 Pat

Krämer und Prechelt [Krämer+96] drücken die Strukturen der Entwurfsmuster, wie sie in [Gamma+96] abgebildet sind, durch *PROLOG*-Regeln aus; der C++ Quelltext des zu analysierenden Softwaresystems[11] wird in eine Darstellung aus *PROLOG*-Fakten übertragen. Auf dieser Grundlage führt dann ein *PROLOG*-System schließlich die Suche nach den Entwurfsmustern aus. Abgedeckt werden damit die Strukturmuster ADAPTER, BRÜCKE, KOMPOSITUM, DEKORIERER und PROXY. Als Beispiel sei hier die Darstellung des OBJEKT-ADAPTER-Musters, ausgedrückt in *PROLOG*-Regeln, aufgeführt.

```
adapter(Target,Adapter,Adaptee):-
   class(_,Target),
   class(concrete,Adapter),
   class(concrete,Adaptee),
   operation(_,_,Target,Request,_,_,_),
   operation(_,_,Adapter,Request,_,_,_),
   operation(_,_,Adaptee,SpecificRequest,_,_,_),
   inheritance(Target,Adapter),
   association(Adapter,Adaptee).
```

Das folgende beispielhafte Code-Fragment eines zu analysierenden Systems

```
class zPane:public zChildWin {
   zDisplay* curDisp;
   /* ... */
public:
   virtual void show(int=SW_SHOWNORMAL);
   /* ... */
};
```

würde in den in den *PROLOG*-Fakten

```
class(conrete, zPane).
inheritance(zChildWin, zPane).
attribute(zPane, curDisp).
operation(virtual, selector, zPane, show, public, "int", "void").
```

resultieren.

Krämer und Prechelt implementierten den beschriebenen Ansatz in ihrem Software-Werkzeug *Pat*. Die Architektur von *Pat* zeigt Abbildung 2.5.

[11] Untersucht werden nur die C++ Header.

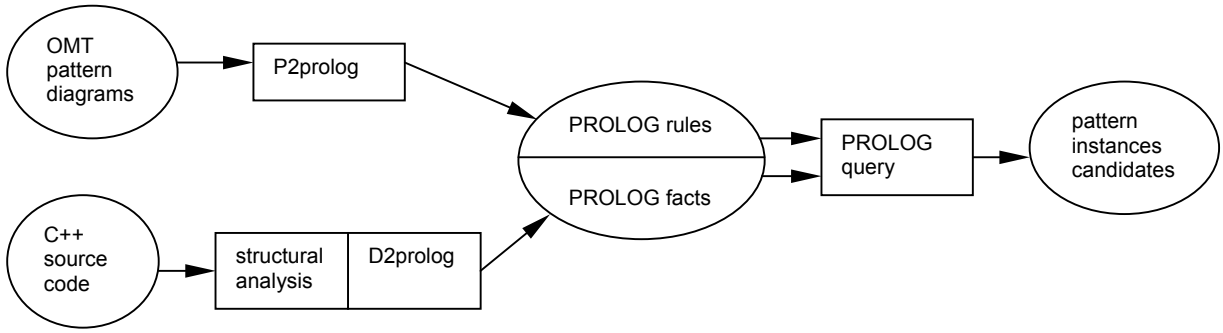


Abbildung 2.5: Architektur des Pat-Systems [Krämer+96]

Die strukturelle Analyse des Quellprogramms wird von dem Software-Werkzeug *Paradigm Plus* vorgenommen, das jedoch einige Mängel aufweist. Es untersucht, wie oben bereits erwähnt, nur die C++ Header und kann überdies nicht zwischen abstrakten und konkreten Klassen unterscheiden, Delegation wird nicht erkannt und auch Sichtbarkeiten von Methoden können nicht erfasst werden.

Pat wurde von Krämer und Prechelt an vier verschiedenen Softwaresystemen getestet: *NME* (Network Management Environment Browser, 9 Klassen), *LEDA* (Library of Efficient Datatypes and Algorithms, 150 Klassen), *zApp* (eine Klassenbibliothek, 240 Klassen) und *ACD* (Automatic call Distribution, 343 Klassen). Die Ergebnisse sind in Tabelle 2.2 zusammengefasst. Die Laufzeiten entstanden auf einem Pentium P133 mit 32 MByte RAM unter Windows 98.

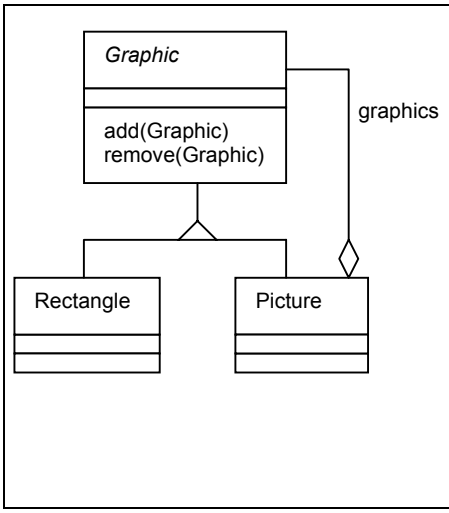
	NME	**LEDA**	**zApp**	**ACD**
Adapter	1	0	12	69
Brücke	0	10	0	0
Kompositum	0	0	0	0
Dekorierer	0	0	0	0
Proxy	0	0	0	0
Recall	100%	100%	100%	100%
Präzision	50%	14%	43%	41%
Laufzeit	1 sec	2 sec	3 sec	36 sec

Tabelle 2.2: Suchergebnisse von Pat [Krämer+96]

2.2.2 IDEA

Die Ergebnisse ihrer Untersuchungen zum automatischen Auffinden von Entwurfsmustern stellen Bergenti und Poggi in [Bergenti+00] vor. Im Unterschied zu Krämer und Prechelt, überprüfen sie nicht nur Übereinstimmungen in den Klassendiagrammen sondern verfeinern ihre Suche mit Hilfe von Kollaborationsdiagrammen. Mit diesem Ansatz konnten die Muster SCHABLONENMETHODE, PROXY, ADAPTER, BRÜCKE, KOMPOSITUM, DEKORIERER, FABRIKMETHODE, ABSTRAKTE FABRIK, ITERATOR, BEOBACHTER und PROTOTYP abgedeckt werden. Vorkommen des FASSADE-, des INTERPRETER- und des SINGLETON-Musters wurden als auf diese Weise unauffindbar beurteilt. Die restlichen Muster werden nicht erwähnt. Im Folgenden seien beispielhaft für das OBJEKTADAPTER-Muster sein Klassendiagramm und sein Kollaborationsdiagramm beschrieben; aufgrund der hohen Ähnlichkeit zu der Struktur und Beschreibung in [Gamma+96] wird jedoch auf eine graphische Darstellung verzichtet. Der OBJEKTADAPTER besteht aus drei Klassen: dem *Ziel*, dem *Adapter* und der *AdaptiertenKlasse*. Der *Adapter* erbt von dem *Ziel* und besitzt eine Referenz auf die *AdaptierteKlasse.* Das Kollaborationsdiagramm schreibt vor, dass *Ziel* eine abstrakte Methode besitzen muss, die der *Adapter* zusammen mit einer Delegation an die *AdaptierteKlasse* implementiert. Für die Umsetzung dieser Vorgehensweise entwickelten Bergenti und Poggi das Software-Werkzeug *IDEA* (Interactive DEsign Assistant).

Im Gegensatz zu *Pat* von Krämer und Prechelt untersucht *IDEA* keine Quellcode-Programme sondern UML-Diagramme eines Software-Entwurfs. Nach dem Erkennen eines Musters erfolgt eine Beurteilung, die die Anwendung des Musters betrifft; bei der Vernachlässigung bestimmter Aspekte werden von *IDEA* Kritiken abgegeben. Die folgende Abbildung 2.6 zeigt dies für das KOMPOSITUM-Muster.



Pattern List		To-do List
Composite		*High*
Graphic	*Component*	Do not put the reference to children in Graphic, put it in Picture only.
Rectangle	*Leaf*	Consider adding getGraphics() in Picture.
Picture	*Composite*	Consider using the Iterator pattern to access Graphic objects in Picture.
		Consider maintaining a reference to the parent from the children.
		Graphic should define as many common operations for Rectangle and Picture as possible.

Abbildung 2.6: Kompositum-Muster mit Kritiken [Bergenti+00]

Auf diese Weise soll der Software-Entwickler in der angepassten Nutzung eines Musters unterstützt werden.

Der gesamte Suchprozess ist in der Abbildung 2.7 skizziert.

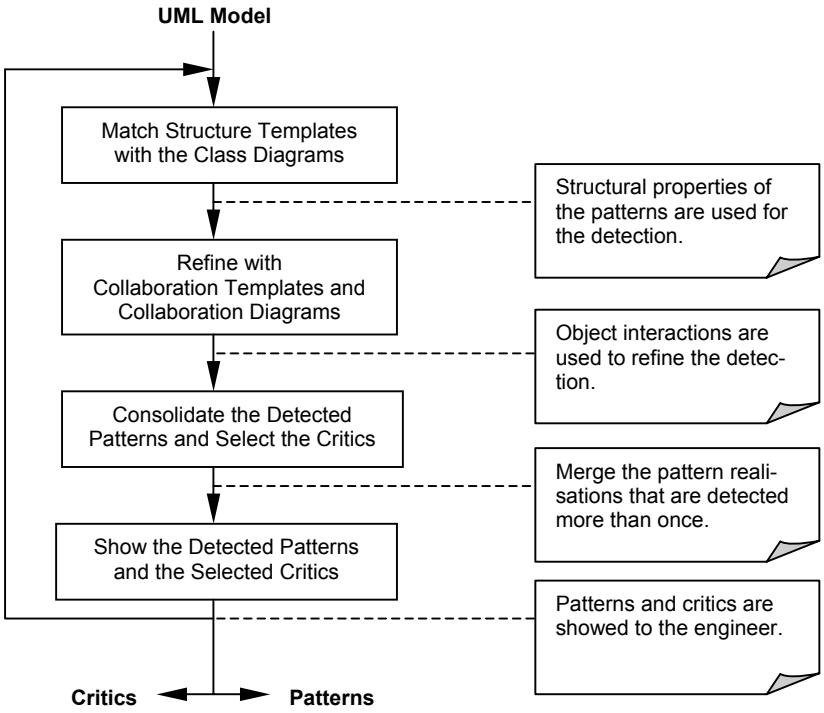


Abbildung 2.7: Suchalgorithmus von IDEA [Bergenti+00]

Intern werden die Strukturdiagramme und die Kollaborationsdiagramme der Referenzmuster wie bei *Pat* durch *PROLOG*-Regeln ausgedrückt. Zu jedem Muster gehören außerdem bestimmte Entwurfsregeln mit korrespondierenden Kritiken sowie ein Satz von Konsolidierungsregeln, die ebenfalls durch *PROLOG*-Regeln repräsentiert werden.

Angaben zur Funktionstüchtigkeit von *IDEA* werden von Bergenti und Poggi nicht gemacht.

2.2.3 Mehrstufiger Suchprozess

Auch Antoniol, Fiutem und Cristoforetti [Antoniol+98] stützen sich bei der Suche nach Entwurfsmustern auf vollständige Übereinstimmungen in der Klassenstruktur. Kollaborationsdiagramme wie bei *IDEA* werden nicht hinzugezogen. Abgedeckt wurden damit die Muster ADAPTER, BRÜCKE, PROXY, KOMPOSITUM und DEKORIERER. Die restlichen Muster werden nicht behandelt. Um den Suchvorgang zu beschleunigen, erfolgt eine stufenweise

Hinzunahme von Restriktionen. Den Ablauf des gesamten Suchprozesses zeigt Abbildung 2.8.

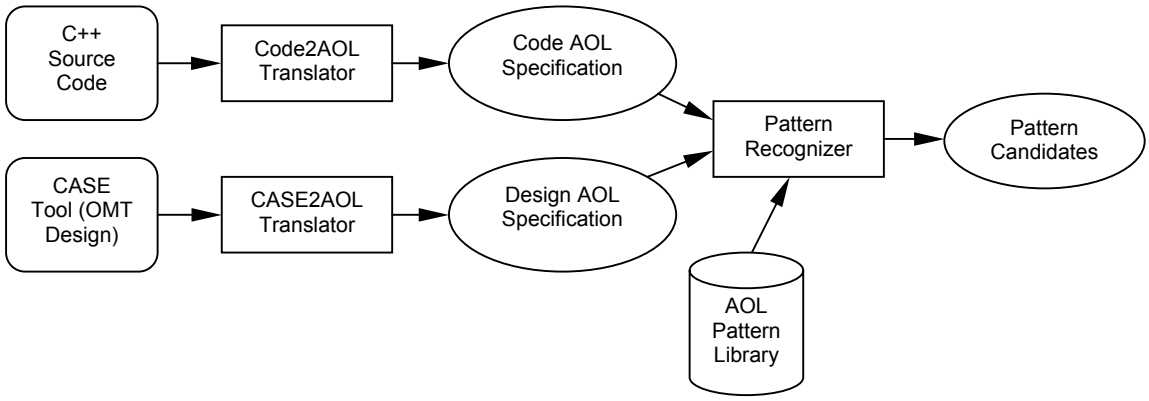


Abbildung 2.8: Ablauf des vollständigen Suchprozesses [Antoniol+98]

Zuerst wird entweder aus C++ Quellcode oder aus einer OMT-Darstellung eine Zwischenrepräsentation (*AOL* - Abstract Object Language), die auf UML basiert, gewonnen. Diese Umwandlung in *AOL* übernehmen die Module **Code2AOL Translator** bzw. **CASE2AOL Translator**. Der **Pattern Recognizer** vergleicht anschließend die gewonnene *AOL*-Darstellung mit den Referenzmustern, die im *AOL*-Format in der **Pattern Library** abgelegt sind. Beispielhaft sei hier das OBJEKTADAPTER-Muster in *AOL* beschrieben:

```
CLASS Target
    OPERATIONS
          PUBLIC Request();
CLASS Adapter
    OPERATIONS
          PUBLIC Request();
CLASS Adaptee
    OPERATIONS
          PUBLIC SpecificRequest()
GENERALIZATION Target
    SUBCLASS Adapter;
RELATION Delegates
    ROLES
          CLASS Adapter MULT One,
          CLASS Adaptee MULT One
```

Der interne Aufbau des **Pattern Recognizers** ist in Abbildung 2.9 dargestellt.

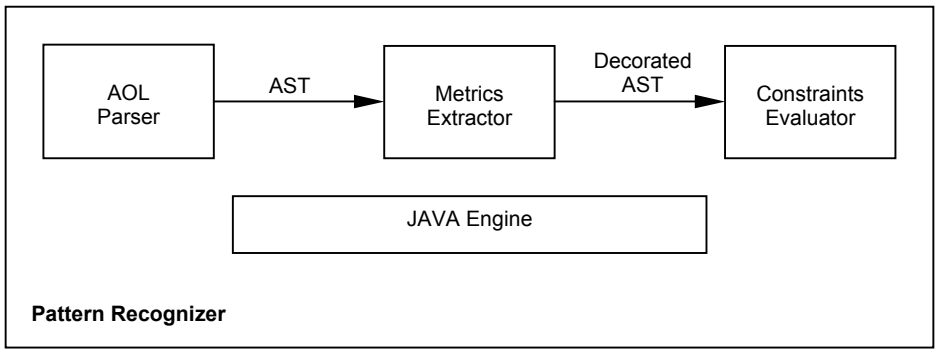


Abbildung 2.9: interner Aufbau des Pattern Recognizers [Antoniol+98]

Der schon erwähnte mehrstufige Suchprozess wurde in den **Constraints Evaluator** integriert. Die folgende Abbildung 2.10 zeigt die interne Struktur dieses Moduls.

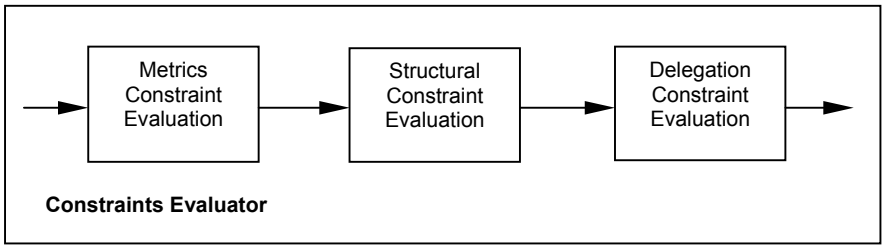


Abbildung 2.10: interner Aufbau des Constraints Evaluators [Antoniol+98]

Der Suchvorgang wurde hier auf drei Stufen aufgeteilt. Die erste Stufe **Metrics Constraint Evaluation** filtert alle Strukturen heraus, deren Metriken mit den Metriken der Entwurfsmuster übereinstimmen. Das OBJEKTADAPTER-Muster beispielsweise besteht aus drei Klassen mit den in Klammern stehenden Metriken *Ziel* (1 Vererbung), *Adapter* (1 Vererbung, 1 Assoziation) und *AdaptierteKlasse* (1 Assoziation). Die nächste Stufe **Structural Constraint Evaluation** wählt aus den übriggebliebenen Strukturen all diejenigen aus, die auch in ihrer Struktur noch mit einem Entwurfsmuster übereinstimmen, wobei jedoch zwischen Delegation und Assoziation noch keine Unterscheidung getroffen wird. Dies geschieht erst in der letzten Stufe **Delegation Constraint Evaluation**.

Die Leistungsfähigkeit ihres Software-Werkzeuges[12] testeten Antoniol, Fiutem und Cristoforetti an folgenden Softwaresystemen: *LEDA*, *libg++* (beides C++ Bibliotheken), *galib* (C++ Genetic Algorithm Library für Optimierungsprobleme), *groff* (GNU Free Software

[12] Es besitzt keinen Namen.

Foundation Version des UNIX troff Utility), *mec* (Trace and Replay Programm) und *socket* (Bibliothek für Interprozess-Kommunikation). Dabei wurden folgende Ergebnisse erzielt (Tabelle 2.3). *ND* bedeutet, dass die Suche ohne die Delegationsbeschränkung durchgeführt wurde, *D*, dass sie mit Delegationsbeschränkung vonstatten ging. *T* gibt die Anzahl der tatsächlich enthaltenen Muster-Instanzen an. Die Systeme *groff* und *socket* enthielten überhaupt keine Muster und sind deshalb in dieser Darstellung nicht enthalten.

	galib			*LEDA*			*libg++*			*mec*		
Muster	**ND**	**D**	**T**	**ND**	**D**	**T**	**ND**	**D**	**T**	**ND**	**D**	**T**
Adapter	52	33	6	110	8	5	15	1	1	30	27	19
Brücke	0	0	0	7	7	0	6	0	0	0	0	0
Proxy	11	0	0	28	0	0	1	0	0	0	0	0
Laufzeit (s)	92	100		207	216		17	25		6	11	
Präzision (%)	9.2	18.2		3.4	33.3		4.5	100		63.3	70.3	

Tabelle 2.3: Ergebnisse der Untersuchung einiger Softwaresysteme [Antoniol+98]

Der *Recall*-Wert betrug überall 100%. Lässt man *groff* und *socket* außen vor, ergibt sich mit der Delegationsbeschränkung aus den verbleibenden Systemen eine durchschnittliche Präzisionsrate von 35%.

2.3 Flexible Musterdefinition und Fuzzylogik

Der Ansatz von Niere, Wadsack und Wendehals [Niere+01] geht davon aus, dass Entwurfsmuster in der Realität nicht genau der Struktur entsprechen, wie sie in [Gamma+96] abgebildet ist, sondern dass diese Struktur durchaus sehr variantenreich sein kann. Zudem sind Unterschiede in der konkreten Implementierung eines Musters zulässig. Um dieser Vielgestaltigkeit zu begegnen, führen sie für die strukturellen Unterschiede eine flexible Definition der Muster ein, die darauf beruht, dass ein Muster aus Submustern bestehen und von anderen Mustern erben kann. In Abbildung 2.11 ist beispielsweise solch eine flexible Definition für das KOMPOSITUM-Muster dargestellt. Ellipsen bezeichnen hierbei andere Muster.

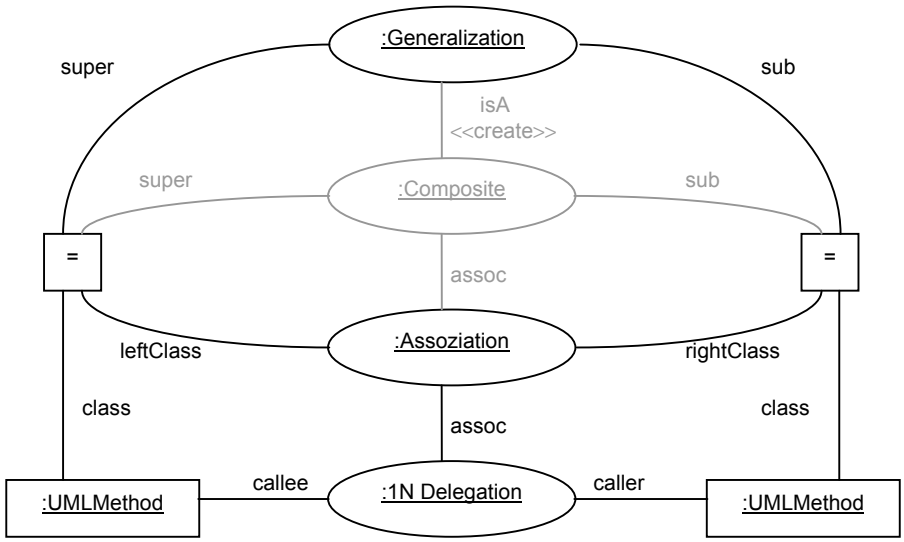


Abbildung 2.11: Definition des Kompositum-Musters (Objektstruktur) [Niere+01]

Das Generalisierungs-Submuster zeigt Abbildung 2.12.

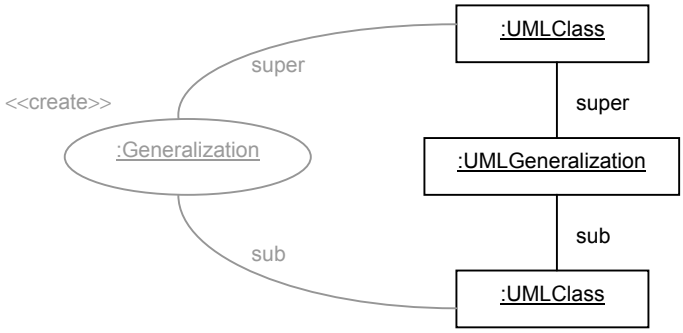


Abbildung 2.12: Generalisierungs-Muster [Niere+01]

Die Unterschiede in der Implementierung werden durch Fuzzy Logik aufgefangen.

An der Implementierung der gezeigten Ansätze wurde zum Zeitpunkt der Veröffentlichung des Artikels noch gearbeitet.

2.4 Suche anhand von Metriken

Der Ansatz von Kim und Boldyreff [Kim+00] geht einen ganz anderen Weg als die bisher vorgestellten. Ihre Idee dabei ist, jedes Muster anhand von Metriken zu charakterisieren, um sie so im Quellcode aufzuspüren. Das zu analysierende Programm muss dazu durch

dieselben Metriken beschrieben werden, um auf dieser Basis die Suche ausführen zu können. Kim und Boldyreff decken mit ihrem Ansatz alle dreiundzwanzig Entwurfsmuster ab; umgesetzt wurde er in ihrem Software-Werkzeug *Pattern Wizard*.

Die verwendeten Metriken zur Charakterisierung sowohl der Entwurfsmuster als auch des zu untersuchenden Programms können in objektorientierte, strukturelle und prozedurale Metriken unterteilt werden. Im folgenden sind jeweils die wichtigsten Metriken aufgelistet und kurz beschrieben.

Objektorientierte Metriken:

- *WMC* (weightet methods per class): gewichtete Methoden pro Klasse;
  WMC1: jede Methode erhält die Wichtung 1, somit wird hier die Anzahl der Methoden gezählt;
  WMCv: private Operationen erhalten die Wichtung 0, öffentliche Methoden erhalten die Wichtung 1
- *DIT* (Depth of inheritance tree): längster Pfad in einem Vererbungsbaum
- *NOC* (Number of Children): Anzahl der direkten Unterklassen
- *CBO* (Coupling between objects): Maß für die Verbindung zu anderen Modulen entweder als Klient oder als Server

Strukturelle Metriken:

- *FI* (Fan-in): Anzahl der Module, die Informationen an das aktuelle Modul senden
- *FO* (Fan-out): Anzahl der Module, an die das aktuelle Modul Informationen sendet
- *IF4* (Information Flow): zusammengesetztes Maß der strukturellen Komplexität, berechnet als das Quadrat des Produkts von *FI* und *FO* eines einzelnen Moduls

Prozedurale Metriken:

- *LOC* (Lines of Code): Anzahl der Codezeilen ohne Leerzeilen und Kommentare
- *MVG* (McCabe's Cyclomatic Complexity): Anzahl linear unabhängiger Pfade durch einen gerichteten azyklischen Graphen, der den Kontrollfluss eines Unterprogramms darstellt
- *COM* (lines of comments) - Anzahl der Kommentarzeilen

Um diese Metriken aus einem Muster bzw. einem ganzen Softwaresystem zu extrahieren, wird ein spezielles Programm eingesetzt: *CCCC* (C and C++ Code Counter). *CCCC* ermittelt absolute Werte, die durch eine statistische Analyse vier Kategorien A bis D zugeordnet werden, wobei A einem hohen Wert und D einem niedrigen Wert entspricht[13]. Dabei ent-

[13] Da die absoluten Werte starken Schwankungen unterliegen können, wurde diese Klassifizierung gewählt.

steht für jedes Muster eine sogenannte Signatur. Einige (unvollständige) Signaturen ausgewählter Entwurfsmuster zeigt Tabelle 2.4.

	Fabrikmethode	**Adapter**	**Kompositum**	**Strategie**
WMC1	B	C	D	C
WMCv	B	C	D	C
DIT	A	C	C	C
NOC	B	C	C	B
CBO	A	C	C	C
LOC	B	A	C	A
MVG	A	B	C	B
COM	A	C	B	A

Tabelle 2.4: Signaturen ausgewählter Muster [Kim+00]

In Abbildung 2.13 ist der Extraktionsprozess der Signaturen für die Muster graphisch dargestellt.

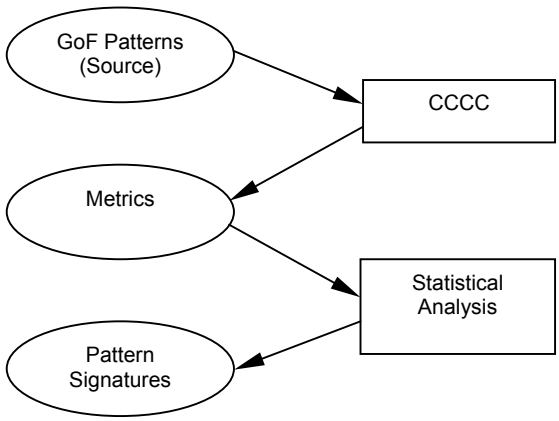


Abbildung 2.13: Extraktionsprozess der Mustersignaturen [Kim+00]

Nachdem die Signaturen für die Entwurfsmuster ermittelt worden sind, muss auch noch das Zielsystem (also das zu untersuchende Programm) auf dieselbe Weise gemessen werden. Dazu wird für jede Klasse des zu untersuchenden Systems eine Signatur erstellt. Dann können die Metriken der Muster mit denen des Programms verglichen werden, um Mustervorkommen herauszufinden. Hierfür kann eine Präzision eingestellt werden, die besagt, wieviele der insgesamt 17 Metriken, die ein Muster charakterisieren, übereinstimmen müssen.

Die Struktur des *Pattern Wizard* in UML-Notation zeigt Abbildung 2.14.

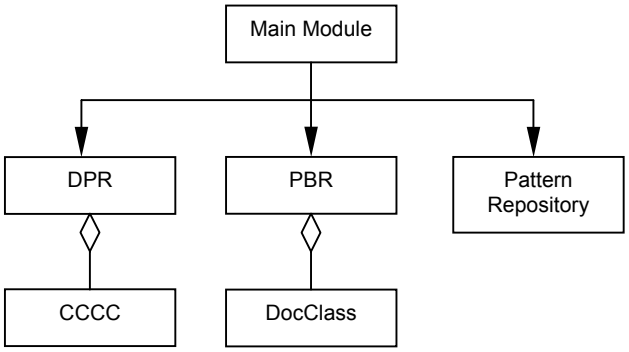


Abbildung 2.14: Struktur des Pattern Wizard [Kim+00]

Der *Pattern Wizard* besteht aus vier Hauptmodulen. **Main Module** koordiniert die Arbeit der drei anderen Module und stellt außerdem die Benutzerschnittstelle dar. **DPR** (Design Pattern Recovery) behandelt das Auffinden der Entwurfsmuster, wie es oben beschrieben wurde. Das **Pattern Repository** enthält die Beschreibungen der Muster ähnlich wie in [Gamma+96]. **PBR** (Pattern-Based Redocumentation) dokumentiert die gefundenen Muster in dem untersuchten System.

Getestet wurde *Pattern Wizard* an drei Systemen. Bei den Systemen 1 und 3 wurde mit 70.59% Präzision (mindestens 12 von 17 möglichen Übereinstimmungen der 17 Metriken) gearbeitet, bei System 2 mit 58.82% (mindestens 10 von 17 möglichen Übereinstimmungen der 17 Metriken). Tabelle 2.5 gibt die vom *Pattern Wizard* identifizierten Muster wieder.

System	Erzeugungsmuster	Strukturmuster	Verhaltensmuster	Muster gesamt
1	4 Singleton (4)	13 Adapter (5) Kompositum (2) Fassade (2) Fliegengewicht (2)	16 Befehl (5) Beobachter (10) Schablonenmethode (1)	33
2	3 Abstrakte Fabrik (1) Erbauer (1) Prototyp (1)	0	0	3
3	0	0	26 Befehl (13) Beobachter (13)	26

Tabelle 2.5: von Pattern Wizard identifizierte Muster [Kim+00]

Die Ergebnisse der anschließenden manuellen Überprüfung der erkannten Muster zeigt Tabelle 2.6.

System	Erzeugungsmuster	Strukturmuster	Verhaltensmuster	Muster gesamt
1	3 Singleton (3)	4 Adapter (1) Kompositum (2) Fliegengewicht (1)	6 Befehl (1) Beobachter (5)	13
2	2 Erbauer (1) Prototyp (1)	0	0	2
3	0	0	12 Befehl (5) Beobachter (7)	12

Tabelle 2.6: von Hand gefilterte Muster [Kim+00]

Der Test ergab eine durchschnittliche Präzisionsrate von 43.55%. Über den *Recall*-Wert wurde keine Aussage gemacht.

2.5 Induktive Methode für die manuelle Suche

Zum Schluss sei die Arbeit von Shull, Melo und Basili [Shull+96] vorgestellt, die zwar keine unmittelbare Umsetzung in ein Software-Werkzeug zuläßt, jedoch durchaus von mehreren solchen unterstützt werden kann. Shull, Melo und Basili schlagen für das Auffinden von Entwurfsmustern eine induktive Methode vor, *BACKDOOR* (Backwards Architecting Concerned with Knowledge Discovery of OO Relationships) genannt, die aus sechs Schritten besteht, die für die Abarbeitung von einem menschlichen Benutzer gedacht sind, wie im Folgenden zu sehen sein wird. Bei *BACKDOOR* sind mehrere Iterationen zulässig und erwünscht. Hier die Schritte im Einzelnen:

Schritt 1: *Problemspezifikation und Entwurfsdokumente ansehen*. Ein Muster ist zu einem großen Teil durch seine Semantik beschrieben. Deshalb sollte versucht werden, aus der Problemspezifikation und den Entwurfsdokumenten soviel wie möglich über den Sinn des Systems und der einzelnen Subsysteme herauszubekommen. Auch wenn diese Dokumente vielleicht nicht dem neuesten Stand entsprechen, sind sie trotzdem immer noch ein gutes Hilfsmittel, um die Beziehungen zwischen den Subsystemen zu verstehen.

Schritt 2: *Ein erstes grobes Modell des Systems aus der Klassendeklaration entwickeln*. Dadurch kann relativ schnell eine Skizze des Systems aufgestellt werden.

Schritt 3: *Das Modell anhand der Klassenimplementation verfeinern.* Hier wird besonders auf die verschiedenen Arten von Beziehungen zwischen den Klassen geachtet (Assoziation, Aggregation, Delegation, lesend/schreibend, Kardinalität).

Schritt 4: Das verfeinerte Modell des Systems benutzen, um potentielle Muster-Kandidaten zu identifizieren. Das geschieht auf der Basis von Vererbung und den Beziehungen zwischen den Klassen.

Schritt 5: *Die Muster-Kandidaten, die in Schritt 4 identifiziert worden sind, analysieren.* Dabei sollte zuerst die Struktur begutachtet werden, anschließend der Zweck und die Implementation.

Schritt 6: *Entwickler und Programmierer konsultieren, um Probleme und Fragen abzuklären.* Dies ist manchmal der einzige Weg zu verstehen, welche Überlegungen zu genau dem vorliegenden Entwurf führten. In der Regel folgt nach solch einer Konsultation eine Rückkehr zu Schritt 4 oder Schritt 5 oder zu einer detaillierteren neuerlichen Iteration aller Schritte.

Zur Überprüfung der Tauglichkeit von *BACKDOOR* wurden sieben Studentenprojekte der Universität Maryland untersucht, die alle dieselbe Aufgabenstellung bearbeitet hatten. Gefunden wurden insgesamt zweiundzwanzig Mustervorkommen, darunter FASSADE, PROXY und VERMITTLER.

2.6 Zusammenfassung und Fazit

In den vorhergehenden Abschnitten wurden alle Arbeiten zum automatischen Auffinden von Entwurfsmustern erläutert. Die folgende Tabelle 2.7 fasst alle Ansätze zusammen.

Name	Werkzeug	Ansatz	abgedeckte Muster	untersuchte Softwaresysteme	Recall	Präzision
DP++ [Bansiya98] (Muster in C++)	ja	nach minimalen Schlüsselstrukturen suchen	Kompositum, Dekorierer, Adapter, Fassade, Brücke, Fliegengewicht, Schablonenmethode, Zuständigkeitskette	Drawing Tool Kit (DTK) (44 Klassen)	keine Angabe	keine Angabe
KT [Brown96] (Muster in Smalltalk)	ja	nach minimalen Schlüsselstrukturen suchen	Kompositum, Dekorierer, Zuständigkeitskette, Schablonenmethode, Strategie, Zustand, Befehl	System A (62 Klassen), System B (264 Klassen), System C (KT) (46 Klassen), System D (40 Klassen)	keine Angabe	keine Angabe
SPOOL [Keller+99] (Muster in C++)	ja	nach minimalen Schlüsselstrukturen suchen	Schablonenmethode, Fabrikmethode, Brücke	System A (3103 Klassen), System B (1420 Klassen), ET++ (722 Klassen)	keine Angabe	keine Angabe
Pat [Krämer+96] (Muster in C++)	ja	nach Übereinstimmungen der Struktur suchen	Adapter, Brücke, Kompositum, Dekorierer, Proxy	NME (9 Klassen), LEDA (150 Klassen), zApp (240 Klassen), ACD (343 Klassen)	100%	37%

Name	Werkzeug	Ansatz	abgedeckte Muster	untersuchte Softwaresysteme	Recall	Präzision
IDEA [Bergenti+00] (Muster in UML)	ja	nach Übereinstimmungen der Struktur suchen, zur Verfeinerung Kollaborationsdiagramme hinzuziehen	Schablonenmethode, Proxy, Adapter, Brücke, Kompositum, Dekorierer, Fabrikmethode, Abstrakte Fabrik, Iterator, Beobachter, Prototyp	keine Angabe	keine Angabe	keine Angabe
Mehrstufiger Suchprozeß [Antoniol+98] (Muster in C++ oder OMT)	ja	nach Übereinstimmungen der Struktur suchen	Adapter, Brücke, Proxy, Kompositum, Dekorierer	LEDA, libg++, galib, groff, mec, socket (keine Angabe zur Anzahl der Klassen)	100%	35%
Flexible Musterdefinition und Fuzzy Logik [Niere+01] (Muster in Java)	nein	der Vielgestaltigkeit der Muster beim Entwurf durch eine flexible Definition (Muster besteht aus Submustern, Vererbung) begegnen und bei der Implementation durch Fuzzy Logik	alle			
Pattern Wizard [Kim+00] (Muster in C++)	ja	Muster durch Metriken charakterisieren, das auf Muster zu untersuchende Programm ebenfalls; auf dieser Grundlage Vergleiche anstellen	alle	System 1, System 2, System 3 (keine Angabe zur Anzahl der Klassen)	keine Angabe	44%
BACKDOOR [Shull+96]	nein	Induktive Methode zum Entdecken von Entwurfsmustern von Hand	alle			

Tabelle 2.7: Ansätze zum Auffinden von Entwurfsmustern

Offensichtlich handelt es sich beim automatischen Auffinden von Entwurfsmustern um keine triviale Angelegenheit handelt, was Tabelle 2.7 auch noch einmal deutlich macht. Es gibt überhaupt nur einen einzigen Ansatz, den *Pattern Wizard* von Kim und Boldyreff, der in der Lage ist, Vorkommen sämtlicher Entwurfsmuster zu finden, wenngleich er auch das SCHABLONENMETHODE-Muster, das einer eindeutigen Definition folgt [Keller+99], nicht eindeutig identifizieren kann – dies lässt den Schluss zu, dass das Verfahren bei anderen, nicht eindeutig definierten Mustern, ebenfalls unsicher arbeitet. Der Ansatz von Niere, Wadsack und Wendehals (*Flexible Musterdefinition und Fuzzy Logik*, Abschnitt 2.3) ist zwar ebenfalls für alle Muster geeignet, befindet sich allerdings noch im Entwicklungsstadium. Der dritte Ansatz, mit dem sich alle Muster finden lassen, ist *BACKDOOR* von Shull, Melo und Basili, jedoch handelt es sich dabei mehr um allgemeine Richtlinien, denn um einen Ansatz, der auf ein Software-Werkzeug übertragen werden könnte. Alle anderen Arbeiten beschränken sich auf eine Teilmenge der Entwurfsmuster aus [Gamma+96], in der Regel auf einige Strukturmuster, und erklären manche Muster sogar für generell unauffindbar oder zumindest aufgrund der Einschränkungen der verwendeten Hilfswerkzeuge für kaum herauszuarbeiten [Brown96, Krämer+96, Bergenti+00]. Sicherlich trifft dies aber

auch auf die Muster zu, die nicht explizit als besonders schwierig zu finden erwähnt wurden. Gerade bei den Ansätzen von Krämer und Prechelt (*Pat*) sowie Antoniol, Fiutem und Cristoforetti (*Mehrstufiger Suchprozess*) wird deutlich, dass es nicht genügt, allein nach Übereinstimmungen in der Struktur zu suchen [Krämer+96] und dabei den Methodenrumpf völlig außer acht zu lassen.

Die Angaben zum *Recall*-Wert sind – sofern sie überhaupt gemacht wurden – durchweg als sehr fragwürdig anzusehen, da es sich stets um einen Wert von 100% handelt, der aussagt, das kein einziges im zu untersuchenden Softwaresystem enthaltenes Muster übersehen wurde. Insbesondere ist diese Aussage zu bezweifeln, da die Arbeiten, die dies behaupten, nämlich *Pat* von Krämer und Prechelt sowie der *Mehrstufige Suchprozess* von Antoniol, Fiutem und Cristoforetti, ihre Suche lediglich auf Übereinstimmungen der Strukturdiagramme, wie sie in [Gamma+96] abgebildet sind, stützen. Da aber, wie schon des öfteren erwähnt, Muster sehr vielgestaltig auftreten können, wäre es mehr als verwunderlich, wenn kein einziges Muster übersehen worden wäre.

Die Werte für die Präzision sind überall sehr niedrig, und das, obwohl sie in keinem Zusammenhang mit der Anzahl der nicht erkannten aber dennoch im System enthaltenen Muster erwähnt werden (Fall 3: *negative true*). Kim und Boldyreff erklären sogar, dass sie diesen Fall 3 nicht berücksichtigt haben [Kim+00][14] . Jedoch lässt diese Tatsache die erreichte Präzision von 44% in einem fragwürdigen Licht erscheinen. Zudem wurden alle Werkzeuge an unterschiedlichen Systemen getestet, was die ermittelten Werte zusätzlich verzerrt.

Als sehr gut einzuschätzen ist dagegen in der Regel die graphische Aufbereitung der gefundenen Muster, insbesondere *SPOOL* von Keller, Schauer, Robitaille und Pagé hebt sich hier hervor. Für denjenigen, der ein Softwaresystem verstehen muss, ist eine gute Darstellung der Ergebnisse auf jeden Fall als Schlüssel zum Erfolg anzusehen [Keller+99].

Abschließend sei noch bemerkt, dass sich alle Autoren einig darüber sind, dass eine Vielzahl von Entwurfsmustern, wenn nicht sogar alle, nicht ohne menschliche Einsicht gefunden werden können. Ihre Vielgestaltigkeit ist zu groß, und sie heben sich zu wenig aus einem System hervor, als dass es ohne eine menschliche Beurteilung möglich wäre.

[14] Das ist durchaus verständlich, denn Systeme mit einer kompletten Dokumentation aller enthaltenen Muster existieren nach meinem Kenntnisstand nicht; somit bliebe der einzige Ausweg, selbst von Hand das zum Test herangezogene Softwaresystem auf Muster zu überprüfen.

2.7 Präzisierte Problemstellung

Nach der Betrachtung der bislang zur automatischen Suche nach Entwurfsmustern gemachten Ansätze, ist es nun möglich, eine präzisierte Aufgabenstellung zu formulieren.

In Kapitel 1 wurde erkannt, dass die Wartungskosten innerhalb des Softwarelebenszyklus rund 50% betragen und dass das Programmverstehen wiederum innerhalb der Wartung ebenfalls etwa 50% ausmacht.

Entwurfsmuster sind einfache und elegante Lösungen für Probleme des objektorientierten Softwareentwurfs. Jedes Muster transportiert eine bestimmte Idee, die hinter dem vorliegenden Musterentwurf steht. Prechelt, Unger und Philippsen wiesen in [Prechelt+97] experimentell nach, dass die Kenntnis enthaltener Entwurfsmuster zu einem schnelleren und besseren Verständnis eines Softwaresystems führt.

Es wurde festgestellt, dass es für den Menschen sehr schwierig ist, Entwurfsmuster manuell in einem Softwaresystem zu identifizieren und daher Unterstützung durch ein Software-Werkzeug nötig ist. Das vorangegangene Kapitel 2 zeigte die bislang zu diesem Thema gemachten Ansätze. Dabei war zu beobachten, dass nur ein einziger Ansatz – die Suche anhand von Metriken – existiert, der ein Auffinden aller Muster aus [Gamma+96] vorsieht, jedoch beim SCHABLONENMETHODE-Muster, das eindeutig definiert ist [Keller+99], und das somit auch eindeutig zu identifizieren wäre, versagt. Zudem beträgt dabei die insgesamt erreichte Präzision unter Missachtung des Falles 3: *negative true* lediglich 44%. Die Arbeiten, die nach minimalen Schlüsselstrukturen suchen, geben nicht für alle Muster solche Schlüsselstrukturen an, zeigen aber, dass es Muster gibt, die eindeutig erkannt werden können [Keller+99]. Dazu gehören das SCHABLONENMETHODE- und das FABRIKMETHODE-Muster [Keller+99]. Sie geben allerdings auch zu, dass sie für bestimmte Muster, zum Beispiel das INTERPRETER-Muster, keine Merkmale gefunden haben [Brown96] bzw. erwähnen viele Muster nicht. Es ist nicht möglich, die Arbeiten von Bansiya, Brown und Keller, Schauer, Robitaille und Pagé zusammenzuführen, um so für jedes Muster eine Suchstrategie zu erhalten, da es Muster gibt, die von keiner der drei Arbeiten behandelt werden.

Aus diesen Nachteilen der existierenden Ansätze ergeben sich drei Forderungen:

1. Es müssen alle Muster aus [Gamma+96] identifiziert werden.
2. Die Methode darf bei eindeutig definierten Mustern wie dem SCHABLONENMETHODE- und dem FABRIKMETHODE-Muster nicht versagen.
3. Anzustreben ist eine Präzision von 100% unter der Bedingung, dass der Fall 3: *negative true* nicht auftritt, also der *Recall*-Wert ebenfalls 100% beträgt.

Um diesen Forderungen nachzukommen, sieht mein Ansatz eine Erweiterung der Suche nach minimalen Schlüsselstrukturen vor. Diese Erweiterung legt dabei für alle dreiundzwanzig Muster aus [Gamma+96] Merkmale fest. Hinzu kommen Merkmale, die nicht

vorhanden sein dürfen, damit es sich um ein Vorkommen eines Musters handelt. Diese Merkmale tragen zwar zu keinem positiven Erkennen eines Musters bei, filtern jedoch die Muster, die dem Fall 2: *positive false* zuzuordnen sind, heraus. Kein weiterer Ansatz verwendet diese Verfahrensweise. Genauere Erläuterungen sind in Kapitel 3 zu finden.

Den Erfolg meines Ansatzes zu validieren, wird im Rahmen dieser Arbeit nur teilweise möglich sein. Ob Merkmale für alle Muster benannt werden und ob diese Merkmale bei dem SCHABLONENMETHODE- und dem FABRIKMETHODE-Muster zu einer eindeutigen Identifizierung führen, kann leicht beurteilt werden. Jedoch ist es hier nicht möglich zu überprüfen, zu welchem Grad Forderung 3 erfüllt werden kann, da, wie bereits in diesem Kapitel kurz erläutert, nach meinem Kenntnisstand keine vollständig dokumentierten Softwaresysteme existieren bzw. hier selbst teilweise dokumentierte Systeme nicht verfügbar waren.

3 Eigener Ansatz

Wie im vorhergehenden Kapitel zu lesen war, sind die bislang gemachten Ansätze zum Auffinden von Entwurfsmustern in einem Quellprogramm oder einem UML-Diagramm kaum zu einem befriedigenden, umfassenden Ergebnis gelangt. In der Regel wurden nur einige der dreiundzwanzig Muster aus [Gamma+96] behandelt; eine Ausnahme bildet nur der Ansatz von Kim und Boldyreff in [Kim+00], der jedoch beim eindeutig definierten SCHABLONENMETHODE-Muster versagt. In diesem Kapitel beschreibe ich meinen eigenen Ansatz, der die Idee von Brown, Bansiya, Keller, Schauer, Robitaille und Pagé, nach bestimmten Schlüsselmerkmalen zu suchen, aufgreift und erweitert, und mit dem ich für jedes der dreiundzwanzig Entwurfsmuster eine Suchstrategie präsentieren werde.

3.1 Prinzip

Fakt ist, dass ein Softwareentwickler, wenn er ein Fragment eines Systems vorgelegt bekommt, sehr wohl sicher entscheiden kann, ob es sich bei diesem Fragment um ein Entwurfsmuster handelt oder nicht. Schwer ist es für ihn nur, in einem großen Softwaresystem nach Mustern zu suchen, von denen er nicht weiß, ob sie im System überhaupt vorhanden sind. Aufgrund dieser Tatsache ist anzunehmen, dass ein Entwurfsmuster gewisse Merkmale, die mitunter sehr feingranular sind, besitzt, die es als ein solches ausweisen. Oft sind es Namen, die auf die Verwendung eines bestimmten Musters hinweisen. Zum Beispiel ist bei der Verwendung einer Klasse, die in ihrem Namen das Suffix *–Fabrik* trägt, stark anzunehmen, dass es sich um ein Vorkommen des ABSTRAKTE-FABRIK-Musters handelt. Jedoch bin ich davon überzeugt, dass ein Muster von einem Softwareentwickler auch ohne Beachtung der Namen identifiziert werden kann, wenn auch mit einem höheren Zeitaufwand. Ist dies der Fall, so ist es auch einem Computerprogramm möglich, bei der Untersuchung eines Codefragments eine solche Entscheidung zu treffen. Eine andere Sache ist es aber, in einem großen Softwaresystem nach Mustern zu suchen. Unter Umständen könnte bei einer Überprüfung aller möglichen Teilmengen[15] des Systems leicht eine exponentielle Zeitkomplexität auftreten. Auch dieser Gefahr gilt es zu begegnen.

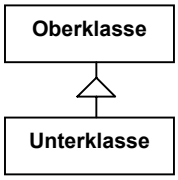
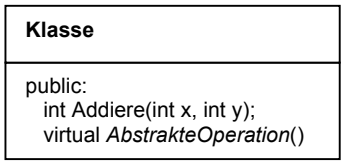
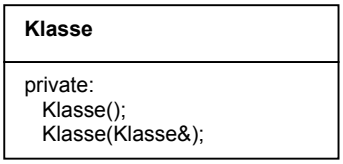
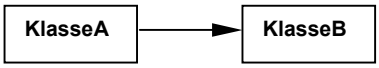
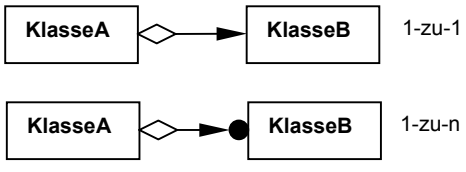
Bei meinem Ansatz, der, wie schon erwähnt, dem von Bansiya, Brown, Keller, Schauer, Robitaille und Pagé ähnelt, halte ich mich eng an die Merkmale eines Musters, nach denen auch ein Mensch bei seiner Beurteilung suchen würde. Im Gegensatz dazu steht der Ansatz von Kim und Boldyreff, der die Muster durch Metriken charakterisiert, was jedoch der menschlichen Denkweise nicht entspricht, hier aber keinesfalls als Mangel gewertet werden soll. Mein Ansatz stellt bei jedem der dreiundzwanzig Muster Schlüsselmerkmale fest,

[15] Hiermit sind Mengen von Klassen gemeint.

die bei einer Anwendung des Musters mit hoher Wahrscheinlichkeit immer vorhanden sind, da es bei fast allen Mustern auch exotischere Varianten gibt, die etwas aus dem Rahmen fallen und daher hier zunächst keine Beachtung finden sollen. Neben Merkmalen die vorhanden sein müssen, lege ich zudem auch solche Merkmale fest, die gerade nicht vorhanden sein sollten, um die Anzahl der den Fall 2: *positive false* betreffenden Muster zu senken. Die Überprüfung nicht vorhandener Merkmale trägt nicht dazu bei, ein Muster positiv zu identifizieren sondern kann höchstens zu einer Minderung der falsch erkannten Mustervorkommen führen. Kein anderer Ansatz bezieht solche Merkmale in die Mustersuche ein. Nach beiden Arten von Merkmalen wird dann in dem zu analysierenden Softwaresystem gesucht. Welche das sind, beschreibt der folgende Abschnitt 3.2. Im Anschluss daran werden die von mir erarbeiteten Merkmale für jedes Muster beschrieben und erläutert.

3.2 Elemente der Schlüsselstrukturen

Welche Merkmale zur Suche nach Entwurfsmustern herangezogen werden, zeigt Tabelle 3.1. Da die Strukturdiagramme in [Gamma+96] in der *Object Modeling Technique* (OMT) von James Rumbaugh erstellt wurden, halte ich mich aus Gründen der Einheitlichkeit ebenfalls daran und verzichte auf eine Darstellung in der *Unified Modeling Language* (UML).

Merkmal	Darstellung
abstrakte Klasse	
konkrete Klasse	
Vererbung	
Attribute (Sichtbarkeit, Typ, Name)	
Operationen (Sichtbarkeit, Polymorphie, Rückgabetyp, Name, Parameter, Abstraktheit)	
Konstruktoren (Sichtbarkeit, Name, Parameter)	
Assoziationsbeziehung	
Aggregationsbeziehung	
Delegation	

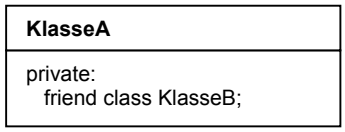
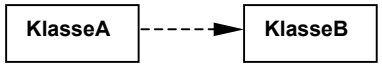
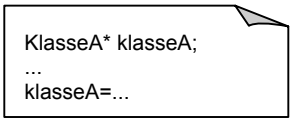
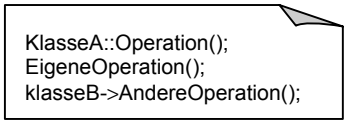
Merkmal	Darstellung
Friend-Beziehung	
Objekterzeugung	
Variablenbenutzung	
Methodenaufrufe	
Templates	

Tabelle 3.1: Schlüsselelemente bei der Mustersuche

Die in Tabelle 3.1 aufgeführten Merkmale tragen zu einer positiven Erkennung von Entwurfsmustern bei. Delegationen und Assoziationsbeziehungen sind nur der Vollständigkeit halber angegeben; bei der weiter unten aufgeführten Beschreibung der Muster werden sie durch die anderen Elemente abgedeckt. Delegation wird durch Methodenaufrufe ersetzt und die Assoziation durch Variablenbenutzung, Methodenaufrufe, Methodenparameter sowie Aggregation. Zwischen Kompositions- und Aggregationsbeziehungen wird nicht unterschieden, da dies für die Mustersuche ohne Belang ist. Alle Kompositionsbeziehungen werden als Aggregationsbeziehungen betrachtet. Eine Aggregationsbeziehung liegt genau dann vor, wenn eine Klasse eine explizite Referenz auf eine andere Klasse verwaltet. Dazu ein Beispiel: Gegeben seien zwei Klassen *KlasseA* und *KlasseB*. In C++ wird der Fall, dass *KlasseA* eine Referenz, das heißt eine Aggregation, auf *KlasseB* besitzt, durch folgende Deklarationen ausgedrückt:

```
class KlasseA
{
   ...
   KlasseB klasseB;
   KlasseB klasseB[10];
   KlasseB* klasseB;
   KlasseB* klasseB[10];
}
```

Bei der Mustersuche kann es Merkmale geben, bei denen unsicher ist, ob sie vorhanden sind. Sie werden allesamt grau mit einer gestrichelten Umrandung dargestellt, wie Tabelle 3.2 zeigt. Die aufgeführten Merkmale stehen beispielhaft für alle Merkmale aus Tabelle 3.1.

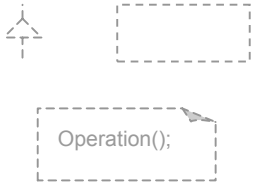
Merkmal	**Darstellung**
Elemente, die vorkommen können, aber nicht vorkommen müssen	

Tabelle 3.2: Elemente, bei denen ein Vorhandensein unsicher ist

Für eine geeignete Kennzeichnung der Merkmale, die nicht vorhanden sein dürfen, wird das Element aus Tabelle 3.3 verwendet.

Merkmal	**Darstellung**
nicht gestattetes Element	

Tabelle 3.3: Kennzeichnung der Elemente, die nicht vorhanden sein dürfen

Merkmale, die durch dieses Symbol gekennzeichnet werden, tragen somit nicht zu einer positiven Identifizierung von Mustern bei; sie senken jedoch die Anzahl der falsch erkannten Muster (Fall 2: *positive false*).

Die Merkmale aus Tabelle 3.2 und Tabelle 3.3 sind Erweiterungen der *Object Modeling Technique* (OMT), die von mir entwickelt wurden, um die für die Mustersuche relevanten Schlüsselelemente anschaulich darzustellen.

Damit wären alle für die Mustersuche wichtigen Elemente beschrieben.

3.3 Die Muster

In den folgenden Abschnitten werde ich für jedes der dreiundzwanzig Entwurfsmuster erklären, anhand welcher Merkmale es sich finden lässt bzw. welche Merkmale nicht vorhanden sein dürfen. Meine Feststellungen beziehen sich hierbei auf die Realisierung der Muster in C++, da auch die meisten Beispiele in [Gamma+96] in dieser Sprache beispielhaft erklärt werden. Die Merkmale sind so gewählt, dass sie sich auf jeden Fall mit Hilfe eines Computerprogramms finden lassen.

Die Erläuterungen der Merkmale der Muster folgen einer bestimmten Struktur, die jedoch nicht durch Teilüberschriften gekennzeichnet ist. Der Aufbau der Struktur ist folgender:

1. Erläuterung des Musters sowie Beschreibung und Begründung der Merkmale.
2. Zusammenfassung der Merkmale.
3. Veranschaulichung der Merkmale anhand einer Abbildung.
4. Skizze eines Algorithmus zur Suche nach dem Muster.
5. Wie die weiteren Klassen des Musters zu finden sind, die nicht zur gesuchten Minimalstruktur dazugehören.
6. Eine kleine Wertung, wie eindeutig das Muster damit gefunden werden kann sowie eine erste grobe Abschätzung der Laufzeit des Algorithmus.

Die Algorithmen verwenden aufgrund der besseren Lesbarkeit einige Konstrukte, bei denen nicht sofort ersichtlich ist, wie diese abgefragt werden können. Dazu ein paar kurze Erläuterungen:

- Der Algorithmus für das ABSTRAKTE-FABRIK-Muster testet für jede Methode, ob sie etwas erzeugt (Suche nach den FABRIKMETHODEN). Dazu muss der Methodenrumpf nach einer Anweisung der Form

  ```
  ...=new TuerMitZauberspruch;
  ```

  durchsucht werden.
- Der Algorithmus für das ERBAUER-Muster testet, ob in eine Methode mit einer Referenz gearbeitet wird. Referenzen, die Klassen auf andere Klassen besitzen, werden durch Attribute ausgedrückt, die ja nichts anderes als Variablen sind. Eine Methode arbeitet mit einer Referenz, wenn sie eine Benutzung der entsprechenden Variable enthält. Das zu überprüfen, macht ebenfalls eine Überprüfung des Methodenrumpfes erforderlich.
- Einige Algorithmen – darunter das ADAPTER-Muster – testen, ob eine bestimmte Klasse mit einer anderen Klasse in einem Vererbungspfad liegt. Das bedeutet, dass eine der beiden Klassen Oberklasse der anderen Klasse ist, wobei jedoch beliebig viele Klassen in der Vererbungshierarchie dazwischen liegen können. Dieses Problem löst ein Algo-

rithmus, der folgendermaßen arbeitet:
Es werden zwei Listen verwaltet: in der einen befinden sich, ausgehend von einer Klasse, alle zugehörigen direkten und indirekten Oberklassen. Diese Liste ist am Anfang leer. Eine zweite Liste verwaltet alle Klassen, deren Oberklassen noch zu bestimmen sind. In ihr befindet sich zu Beginn lediglich die Klasse, die den Ausgangspunkt bildet. Die erste Liste lässt sich auch als *abgearbeit* und die zweite Liste als *noch_zu_bearbeiten* bezeichnen.
Nun werden für jede Klasse der zweiten Liste die Oberklassen bestimmt und in dieser zweiten Liste abgelegt. Die Klasse, deren Oberklassen soeben ermittelt wurden, wird aus der zweiten Liste entfernt und in der ersten Liste abgespeichert. Dies geschieht solange, bis die zweite Liste *noch_zu_bearbeiten* leer ist.
Ob die Ausgangsklasse sich schließlich mit einer anderen Klasse in einem Vererbungspfad befindet, kann durch ein Enthaltensein der potentiellen Oberklasse in der Liste *abgearbeitet* überprüft werden.

- Eine Algorithmen erstellen zunächst eine Menge der enthaltenen Bäume. Damit ist nicht gemeint, alle Teilbäume zu extrahieren, bei denen die Wurzel innerhalb des Systems keine echte Wurzel ist, da sie Oberklassen besitzt. Als Bäume werden stattdessen hier nur solche betrachtet, deren Wurzelklasse keine Oberklassen besitzt. Eine Iteration über alle Klassen und eine Abfrage auf vorhandene Oberklassen ergibt sehr schnell die Menge der im System enthaltenen echten Bäume.

Die kleine Wertung im Schlussabsatz beurteilt, wie eindeutig sich ein Muster anhand der jeweils genannten Merkmale finden lässt. Diese Wertung beruht auf meiner eigenen Programmiererfahrung. Es wird zwischen den Kategorien *schwer*, *mittel* und *leicht* unterschieden. *Leicht* bedeutet, dass die angegebenen Merkmale zu einer eindeutigen Identifizierung führen; *schwer* sagt aus, dass es sich nicht um so eindeutige Merkmale handelt.

Die grobe Abschätzung der Laufzeit der Algorithmen folgt der bekannten *O*-Notation, wobei *n* der Anzahl der Klassen des zu untersuchenden Systems entspricht.

Die Auflistung der Muster erfolgt wieder anhand der drei Kategorien Erzeugungsmuster, Strukturmuster und Verhaltensmuster.

3.4 Erzeugungsmuster

Erzeugungsmuster dienen der Erzeugung von Objekten, wobei sie den Erzeugungsprozess verstecken. Sie helfen dabei, ein System unabhängig davon zu machen, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden. Alles was die Anwendung insgesamt über ihre Objekte weiß, wird durch die von abstrakten Klassen definierten Schnittstellen bestimmt. Erzeugungsmuster ermöglichen somit zu bestimmen, was erzeugt wird, wer es erzeugt, wie es erzeugt wird und wann es erzeugt wird.

Erzeugungsmuster sind vor allem dann von Bedeutung, wenn Systeme beginnen, mehr von *Objektkomposition* als von Vererbung abzuhängen. Objektkomposition und Vererbung sind die beiden bekanntesten Techniken für die Wiederverwendung von Funktionalität in objektorientierten Softwaresystemen. Bei der Objektkomposition (auch Black-Box-Wiederverwendung genannt) wird neue komplexe Funktionalität durch das Zusammenführen von Objekten erreicht; sie erfolgt dynamisch zur Laufzeit, indem Objekte Referenzen auf andere Objekte erhalten. Vererbung (auch White-Box-Wiederverwendung genannt) dagegen wird statisch zur Übersetzungszeit ausgeführt.

3.4.1 Abstrakte Fabrik

Das ABSTRAKTE-FABRIK-Muster besteht aus einer abstrakten Fabrik, konkreten Fabriken, abstrakten Produkten und konkreten Produkten. Wie im Beispielcodeabschnitt des ABSTRAKTE-FABRIK-Musters aus [Gamma+96] zu sehen ist, muss die abstrakte Fabrik keineswegs eine abstrakte Klasse sein; stattdessen kann sie ein Standardverhalten implementieren. Genauso verhält es sich mit den abstrakten Produkten. Daraus ist zu schließen, dass nicht davon ausgegangen werden kann, für die abstrakte Fabrik und die abstrakten Produkte tatsächlich abstrakte Klassen vorzufinden. Da das ABSTRAKTE-FABRIK-Muster üblicherweise mittels FABRIKMETHODEN[16] realisiert wird, gilt es, diese zu finden und daraus ein Vorhandensein des ABSTRAKTE-FABRIK-Musters abzuleiten. Eine FABRIKMETHODE zeichnet sich, dadurch aus, dass sie in ihrem Methodenrumpf ein konkretes Objekt, das eine Spezialisierung eines abstrakten Objekts darstellt, erzeugt und dieses zurückliefert, aber als Rückgabetyp die abstrakte Klasse besitzt (siehe auch Abschnitt A.1.3 Fabrikmethode). Da es unsicher ist, ob es sich bei der abstrakten Fabrik um eine abstrakte Klasse handelt oder nicht, kann sich die Suche nur an einer konkreten Fabrik orientieren, da ausschließlich hier die FABRIKMETHODEN implementiert sind. In meinem Ansatz müssen Fabriken mindestens zwei FABRIKMETHODEN enthalten.

Zusammengefasst sind hier alle Merkmale des ABSTRAKTE-FABRIK-Musters aufgelistet:

- Gesucht wird nach einer konkreten Fabrik.
- Um eine konkrete Fabrik handelt es sich, wenn sie mindestens zwei Methoden besitzt, die der Definition des FABRIKMETHODE-Musters genügen.

Die folgende Abbildung 3.1 verdeutlicht dies.

[16] In [Gamma+96] wird auch von einer Realisierung durch Prototypen berichtet, die jedoch hier keine Beachtung finden soll.

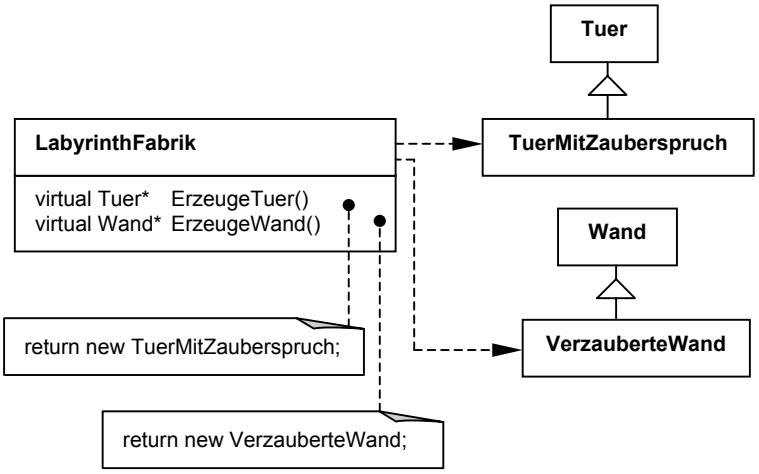


Abbildung 3.1: Merkmale des Abstrakte-Fabrik-Musters

Algorithmus 3.1 skizziert, wie nach dem ABSTRAKTE-FABRIK-Muster gesucht werden kann.

Algorithmus 3.1: FindeAbstrakteFabrik

```
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
```

Wurde eine konkrete Fabrik gefunden, ist es schließlich noch nötig, das komplette Muster zu erfassen. Dazu gehören die abstrakte Fabrik, die abstrakten Produkte, die konkreten Fabriken und die konkreten Produkte. Die Fabriken sind daran zu erkennen, dass sie dieselben beiden FABRIKMETHODEN besitzen wie die gefundene Fabrik; sie befinden sich zudem in ein und demselben Vererbungsbaum. Um die Produkte festzustellen, werden sämtliche Fabriken nach ihren FABRIKMETHODEN durchsucht; zu den Produkten gehören dann schließlich alle Klassen, die entweder als Rückgabetyp oder als erzeugtes Objekt vorkommen.

Schwierigkeit: *leicht*.
Laufzeit des Algorithmus: *O(n)*.

3.4.2 Erbauer

Das ERBAUER-Muster besteht aus einem abstrakten Erbauer, mehreren konkreten Erbauern, dem Direktor sowie dem Produkt. Im Extremfall existiert kein abstrakter Erbauer; der Direktor hält die Referenz direkt auf den konkreten Erbauer. Ein konkreter Erbauer ist dadurch gekennzeichnet, dass er mindestens eine Methode besitzt, die an dem Produkt baut, dass eine weitere Methode das fertige Produkt zurückgibt und dass eine Referenz auf das Produkt existiert. Die Methode, die an dem Produkt baut, benutzt in ihrem Methodenrumpf die Variable, die die Referenz auf das Produkt enthält, denn nur auf diese Weise ist es möglich, den Bau des Produktes fortzuführen.

Die Merkmale auf einen Blick:

- Gesucht wird nach einem konkreten Erbauer.
- Es gibt im konkreten Erbauer eine Methode, die das fertige Produkt zurückliefert.
- Der konkrete Erbauer besitzt eine Aggregationsbeziehung zu dem Produkt.
- Es gibt wenigstens eine Konstruktionsmethode im konkreten Erbauer, die mit der Variable, welche die Referenz auf das Produkt enthält, arbeitet.

Die folgende Abbildung 3.2 verdeutlicht die genannten Merkmale.

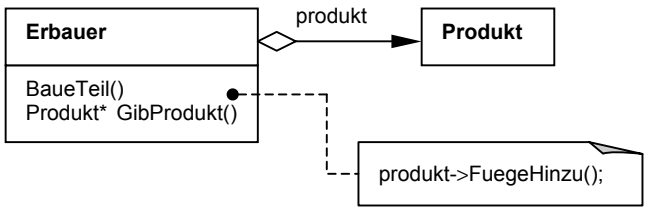


Abbildung 3.2: Merkmale des Erbauer-Musters

Der folgende Algorithmus 3.2 skizziert eine Suchstrategie.

Algorithmus 3.2: 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
      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
```

Um das gesamte Muster zu finden, ist es nötig, alle vorhandenen Erbauer-Klassen sowie den Direktor erfassen. Die Erbauer-Klassen bilden einen Baum. Die Wurzel-Klasse ist der abstrakte Erbauer. Direktoren sind alle Klassen, die eine Referenz auf den abstrakten Erbauer besitzen.

Schwierigkeit: *mittel*.
Laufzeit des Algorithmus: *O(n)*.

3.4.3 Fabrikmethode

Das FABRIKMETHODE-Muster setzt sich aus einem abstrakten Produkt, einem konkreten Produkt, einem Erzeuger und einem konkreten Erzeuger zusammen. Bei dem abstrakten Produkt und dem abstrakten Erzeuger muss es sich nicht um abstrakte Klassen handeln; sie können Standardimplementierungen besitzen. Aufgrund dieser Unsicherheit muss somit nach einem konkreten Erzeuger gesucht werden, der die Fabrikmethode auf jeden Fall implementiert. Gekennzeichnet ist eine Fabrikmethode dadurch, dass sie ein konkretes Produkt erzeugt, als Rückgabetyp jedoch ein abstraktes Produkt besitzt. Fabrikmethoden sind stets polymorph.

Im Folgenden sind die Merkmale des Fabrikmethoden-Musters noch einmal zusammengefasst:

- Gesucht wird nach einem konkreten Erzeuger, der eine Fabrikmethode besitzt.
- Eine Fabrikmethode ist `virtual`.
- Sie erzeugt ein Objekt einer anderen Klasse (das konkrete Produkt), besitzt als Rückgabetyp jedoch eine von dem erzeugten Objekt verschiedene Klasse (das abstrakte Produkt).
- Die Klasse des Rückgabetyps (abstraktes Produkt) ist eine Oberklasse der Klasse, von der ein Objekt erzeugt wird (konkretes Produkt).

Abbildung 3.3 veranschaulicht die angeführten Merkmale.

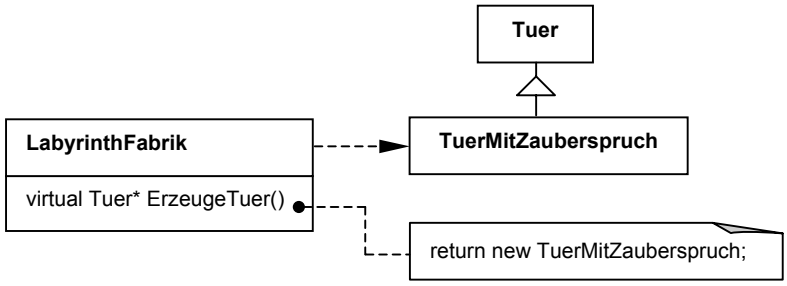


Abbildung 3.3: Merkmale des Fabrikmethode-Musters

Der Algorithmus für die Suche nach dem FABRIKMETHODE-Muster skizziert sich wie folgt:

Algorithmus 3.3: FindeFabrikmethode

```
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. TuerMitZau-
      berspruch), 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
```

Um das Muster zu vervollständigen, muss schließlich noch die Existenz eines abstrakten Erzeugers überprüft werden.

Schwierigkeit: *leicht*.
Laufzeit des Algorithmus: *O(n)*.

3.4.4 Prototyp

Das PROTOTYP-Muster besteht aus einem abstrakten Prototyp, der die Klone-Operation deklariert, konkreten Prototypen, die diese Operation überschreiben und einem Klienten, der auf alle vorhandenen Prototyp-Klassen zugreift. Es kann nicht davon ausgegangen werden, dass der abstrakte Prototyp wirklich eine abstrakte Klasse ist. Ebensogut kann er ein Standardverhalten definieren. Zudem kann der Vererbungsbaum der Prototyp-Klassen eine beliebige Verschachtelungstiefe besitzen. Aufgrund dieser Gegebenheiten erfolgt die Suche anhand eines konkreten Prototyps, der die Eigenschaft besitzt, die Klone-Operation

zu implementieren, die ein Objekt von sich selbst erzeugt und dafür den in der Klasse vorhandenen Kopierkonstruktor verwendet. Die Klone-Operation führt außerdem als Rückgabetyp eine Klasse, die der eigenen bzw. einer Oberklasse davon entspricht. Die Klone-Operation ist polymorph.

Folgend sind die Merkmale des PROTOTYP-Musters noch einmal zusammenfassend aufgeführt:

- Gesucht wird nach der Klone-Operation eines konkreten Prototypen.
- Die Klone-Operation erzeugt ein Objekt der eigenen Klasse und verwendet dazu den Kopierkonstruktor der Klasse.
- Es muss also ein Kopierkonstruktor vorhanden sein.
- Die Klone-Operation besitzt als Rückgabetyp die eigene Klasse oder eine Oberklasse.
- Die Klone-Operation ist `virtual`.

Das Klassendiagramm in Abbildung 3.4 veranschaulicht die genannten Merkmale.

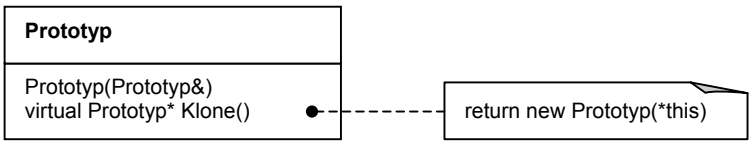


Abbildung 3.4: Merkmale des Prototyp-Musters

Ein entsprechender Algorithmus lautet folgendermaßen:

Algorithmus 3.4: FindePrototyp

```
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 Me-
   thode polymorph?
   ja: liefert diese Methode j die Klasse i oder Oberklasse der Klasse
   i zurück?
   ja: Muster gefunden
od
```

Um das Muster in seiner Gesamtheit zu erfassen, ist zunächst der Baum der Prototyp-Klassen zu untersuchen; hierbei gelten die selben Merkmale wie oben, das heißt, es zählen alle Klassen, die einen Kopierkonstruktor besitzen und die Klone-Operation implementieren. Klienten schließlich sind alle Klassen, die von der Klone-Operation Gebrauch machen.

Schwierigkeit: *leicht*.
Laufzeit des Algorithmus: *O(n)*.

3.4.5 Singleton

Eine SINGLETON-Klasse definiert eine Exemplar-Operation, die es Klienten ermöglicht, auf sein einziges Exemplar zuzugreifen. Da es sich um eine globale Schnittstelle handelt, wird die Exemplar-Operation statisch deklariert. Die Verwaltung des einzigen Exemplars erfolgt durch eine Referenz auf sich selbst bzw. auf eine Oberklasse[17]; diese Referenz ist ebenfalls statisch deklariert. Um anderen Klassen zu verbieten, selbst Objekte von der Singleton-Klasse zu erzeugen, wird der Konstruktor durch `private` oder `protected` verborgen. Andere Möglichkeiten, das SINGLETON-Muster zu implementieren, gibt es nicht.

Die Merkmale des SINGLETON-Musters zusammengefasst lauten:

- Gesucht wird nach einer Singleton-Klasse.
- Diese Klasse besitzt keinen öffentlichen Konstruktor, sondern nur einen `private`- bzw. `protected`-Konstruktor.
- Die Klasse besitzt eine Exemplar-Operation, die `static` ist und als Rückgabetyp die eigene Klasse bzw. eine Oberklasse führt.
- Es existiert eine mit `static` deklarierte Variable vom Typ der eigenen Klasse bzw. einer Oberklasse.

Hinzugezogen werden könnte noch die Tatsache, dass in der Exemplar-Operation ein Objekt der eigenen Klasse erzeugt wird, jedoch genügen die angegebenen Merkmale bereits. Abbildung 3.5 macht die genannten Merkmale deutlich.

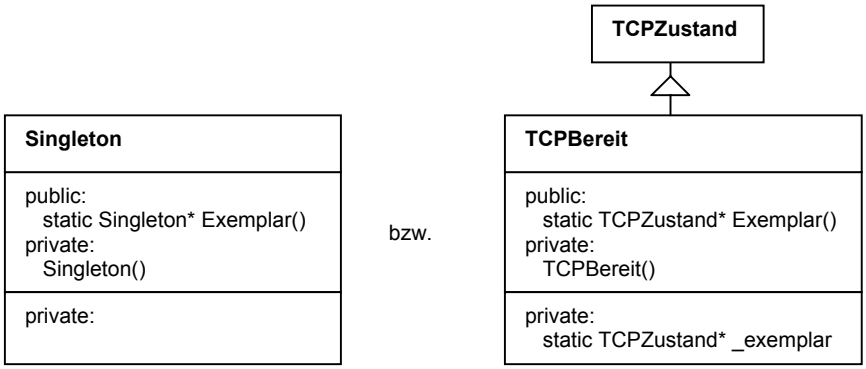


Abbildung 3.5: Merkmale des Singleton-Musters

Der Algorithmus für die Suche entspricht einer einfachen Anfrage:

[17] Im Beispielcodeabschnitt des ZUSTANDS-Musters in [Gamma+96] sind alle konkreten Zustände Ausprägungen des SINGLETON-Musters, die keine Referenz auf sich selbst sondern auf ihre Oberklasse, den abstrakten Zustand, besitzen.

Algorithmus 3.5: FindeSingleton

```
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 das Singletonmuster nur aus einer Klasse besteht, müssen keine weiteren zum Muster gehörenden Klassen gefunden werden.

Schwierigkeit: *leicht*.
Laufzeit des Algorithmus: *O(n)*.

3.5 Strukturmuster

Strukturmuster befassen sich mit der Komposition von Klassen und Objekten, um größere Strukturen zu bilden. Klassenbasierte Strukturmuster benutzen Vererbung, um Schnittstellen oder Implementierungen zusammenzuführen. Objektbasierte Strukturmuster beschreiben Mittel und Wege, Objekte zusammenzuführen. um neue Funktionalität zu gewinnen. Die zusätzliche Flexibilität der Objektkomposition ergibt sich aus der Möglichkeit, das Kompositionsgefüge zur Laufzeit zu ändern, was mit statischer Klassenvererbung nicht möglich ist.

3.5.1 Adapter

Für beide Varianten des ADAPTER-Musters, den objektbasierten und den klassenbasierten ADAPTER, gibt es keine Variationsmöglichkeiten. Der KLASSENADAPTER läßt sich nur über Mehrfachvererbung realisieren, der OBJEKTADAPTER über die bekannte Ableitung von der Ziel-Klasse und der Delegation der Anfragen an die adaptierte Klasse.

Zunächst zum KLASSENADAPTER. Er beruht auf Mehrfachvererbung. Laut [Gamma+96] erbt Adapter die Schnittstelle von Ziel `public`, die Implementierung der adaptierten Klasse jedoch `private`. Mehrfachvererbung gilt als ein Konzept, das Risiken birgt, und mit dem deshalb vorsichtig umgegangen werden sollte. Es wird in der Regel selten auftauchen. Desweiteren überschreibt die Adapter-Klasse wenigstens eine Methode der Ziel-Klasse und ruft darin eine Methode der adaptierten Klasse auf. Dieser Methodenaufruf ist ein unbedingtes Indiz für das Vorhandensein eines ADAPTERS; der Aufwand, der in einen Adapter hineingesteckt wird, kann sehr variabel sein – er reicht von einfacher Delegation bis hin

zu umfangreichen Berechnungen [Gamma+96] –, jedoch ist der Aufruf einer Methode der adaptierten Klasse dennoch stets vorhanden. Dies gilt sowohl für den OBJEKTADAPTER als auch für den KLASSENADAPTER.

Die Merkmale des KLASSENADAPTERS auf einen Blick:

- Gesucht wird Adapter.
- Adapter erbt von zwei Klassen. Von der einen Klasse `public` (Ziel) und von der anderen `private` (adaptierte Klasse).
- Adapter überschreibt wenigstens eine Operation der Ziel-Klasse und ruft darin eine Operation der adaptierten Klasse auf. Diese Operation ist polymorph und muss `virtual` deklariert sein.

Die folgende Abbildung 3.6 veranschaulicht die Merkmale.

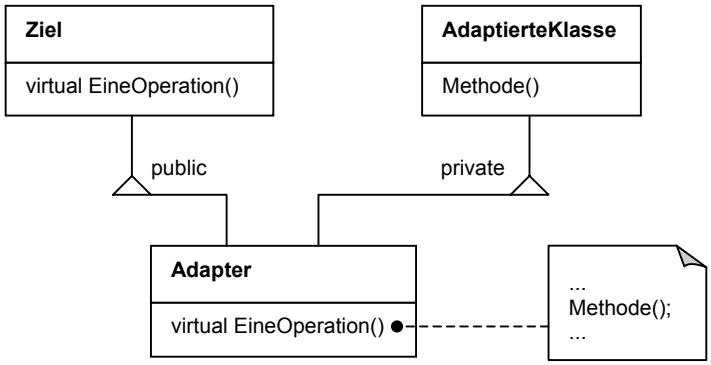


Abbildung 3.6: Merkmale des Klassenadapters

Der Algorithmus für die Suche nach dem KLASSENADAPTER-Muster lautet folgendermaßen:

Algorithmus 3.1: FindeKlassenadapter

```
für alle Klassen i (Adapter) do
   besitzt Klasse i zwei Oberklassen (Ziel und AdaptierteKlasse)?
   ja: erbt sie von der einen Klasse k (Ziel) public und von der ande-
   ren Klasse l (AdaptierteKlasse) private?
   ja: für alle Methoden j der Klasse k do
      überschreibt Methode j eine Methode der Klasse i?
      ja: wird dort eine Methode der Klasse l aufgerufen?
      ja: Muster gefunden
   od
od
```

Damit kann das KLASSENADAPTER-Muster komplett gefunden werden.

Schwierigkeit: *leicht*.
Laufzeit des Algorithmus: *O(n)*.

Nun zum OBJEKTADAPTER. Die Merkmale orientieren sich, wie schon beim KLASSENADAPTER, sehr eng an der vorgegebenen Struktur; Abweichungen davon sind nicht möglich:

- Gesucht wird Adapter-Klasse.
- Adapter ist Unterklasse von einer anderen Klasse (Ziel).
- Adapter besitzt eine Referenz auf die adaptierte Klasse.
- Adapter überschreibt wenigstens eine Methode von Ziel (`virtual` Deklaration) und ruft darin eine Methode der adaptierten Klasse auf.
- Adapter ist keine Ober oder Unterklasse der adaptierten Klasse.

Diese Merkmale sind zum besseren Verständnis in Abbildung 3.7 dargestellt.

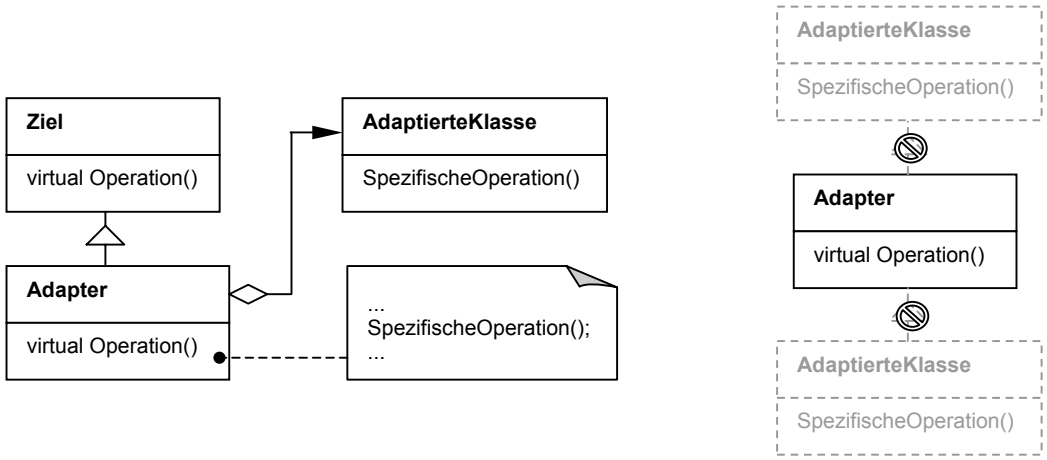


Abbildung 3.7: Merkmale des Objektadapters

Der Algorithmus für die Suche nach dem OBJEKTADAPTER-Muster sieht wie folgt aus:

Algorithmus 3.7: FindeObjektadapter

```
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
```

Das OBJEKTADAPTER-Muster ist damit komplett erfasst.

Schwierigkeit: *leicht*.
Laufzeit des Algorithmus: *O(n)*.

3.5.2 Brücke

Das BRÜCKE-Muster kann sehr variabel verwendet werden. Es ist trivial, dass die Bäume der Abstraktion und der Implementierung jegliche Gestalt bezüglich der Tiefe und Breite ihrer Baumstruktur annehmen können. In [Keller+99] wird zudem von Abstraktionen ohne spezialisierte Abstraktion und konkreten Implementierern ohne gemeinsame Oberklasse berichtet. Als Suchmerkmal greife ich daher auf eine Minimalstruktur zurück, die lediglich aus Abstraktion und Implementierer besteht; jedoch werden eventuell vorhandene Unterklassen zur Urteilsfindung hinzugezogen. Da die Implementierer-Klassen in der Regel primitive Operationen implementieren und die Abstraktions-Klassen ihre eigenen Methoden auf der Basis dieser primitiven Operationen definieren [Gamma+96], kann davon ausgegangen werden, dass es in keiner der Implementierer-Klassen eine Referenz auf eine Abstraktions-Klasse geben wird. Methodenaufrufe von den Implementierer-Klassen auf die Abstraktionsklassen existieren ebenfalls nicht. Abstraktion besitzt aber eine Referenz auf Implementierer.

Zusammengefasst hier noch einmal die Merkmale:

- Gesucht wird nach Abstraktion.
- Abstraktion besitzt Referenz auf Implementierer.
- Von den Implementierer-Klassen gibt es keine Referenz auf eine Abstraktions-Klasse.
- In den Implementierer-Klassen sind keine Methodenaufrufe einer Abstraktions-Klasse enthalten.

Abbildung 3.8 verdeutlicht die genannten Merkmale.

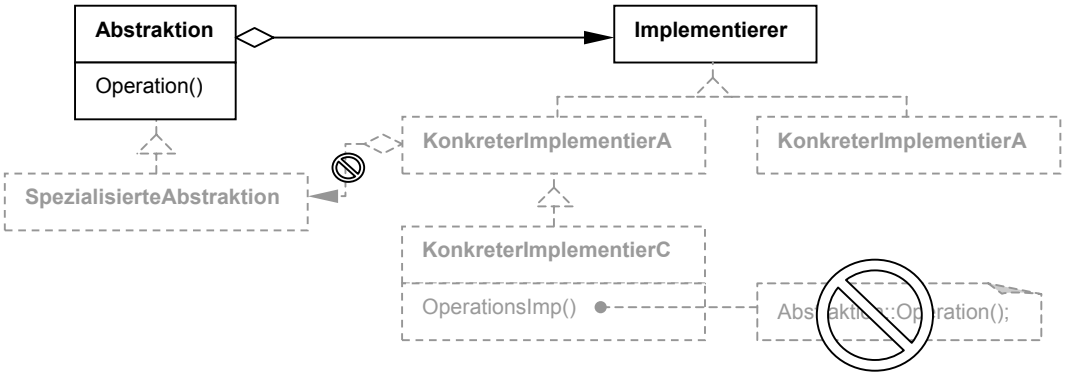


Abbildung 3.8: Merkmale des Brücke-Musters

Der Algorithmus für die Suche nach dem BRÜCKE-Muster lautet folgendermaßen:

Algorithmus 3.8: 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 Collecti-
      on x (Abstraktionen);
      speichere alle Methoden der Klassen der Collection x in einer
      Collection y (Implementierer);
      durchsuche die Methodenrümpfe aller Klassen des Baumes der Klas-
      se j nach Aufrufen von Methoden der Collection y; keine gefun-
      den? weiter
      durchsuche alle Klassen des Baumes der Klasse j nach Referenzen
      auf eine Klasse der Collection x; keine gefunden? Muster erkannt
   od
od
```

Da das Muster aus dem Unterbaum von Abstraktion und dem Unterbaum von Implementierer besteht, ist es praktisch durch den Suchalgorithmus bereits komplett gefunden.

Schwierigkeit: *schwer*.
Laufzeit des Algorithmus: $O(n^3)$.

3.5.3 Dekorierer

Beim DEKORIERER-Muster ist die Zahl der konkreten Dekorierer nicht bestimmt. Außerdem muss die Referenz der Dekorierer-Klasse nicht zwingend auf eine direkte Oberklasse zeigen; es können auch weitere Klassen in dem Vererbungspfad liegen [Bansiya98, Brown96]. Das heißt, die Referenz der Dekorierer-Klasse besteht zu einer Klasse, die Oberklasse der Dekorierer-Klasse ist, jedoch keine direkte Oberklasse der Dekorierer-Klasse

sein muss. Bei dieser Referenz handelt es sich um eine 1-zu-1-Aggregation. Um die Funktionalität einer Komponente zu erweitern, besitzt ein konkreter Dekorierer einen Aufruf einer Methode seiner Oberklasse, das heißt der Dekorierer-Klasse, und im Anschluss daran der Aufruf einer lokalen Methode, die die zusätzliche Funktionalität implementiert. Schließlich enthält diejenige Operation des Dekorierers, die der konkrete Dekorierer aufruft, einen Aufruf auf die Klasse, auf die der Dekorierer die Referenz besitzt.

Im Folgenden sind die Merkmale des DEKORIERER-Musters zusammengefasst:

- Gesucht wird nach Dekorierer-Klasse.
- Dekorierer besitzt eine 1-zu-1-Aggregation auf eine Oberklasse.
- Dekorierer besitzt mindestens eine Unterklasse (konkreter Dekorierer).
- In einer Methode des konkreten Dekorierers erfolgt ein Aufruf der Form Dekorierer::Operation(). In dieser Methode des konkreten Dekorierers wird zudem eine lokale Operation aufgerufen.
- Die Methode Dekorierer::Operation() ruft eine gleichnamige Methode der Komponenten-Klasse auf.

Das folgende Klassendiagramm veranschaulicht diese Merkmale (Abbildung 3.9).

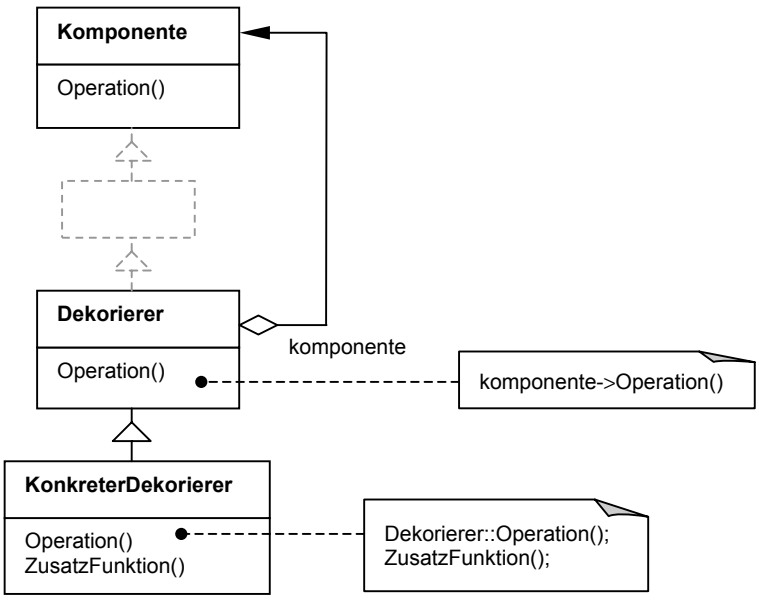


Abbildung 3.9: Merkmale des Dekorierer-Musters

Der Algorithmus sieht wie folgt aus:

Algorithmus 3.9: FindeDekorierer

```
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 Methodenauf-
      ruf?
      ja: enthält die in Methode l aufgerufene Methode der Klasse i
      einen Aufruf der gleichnamigen Methode der Klasse j?
      ja: Muster gefunden
   od
od
```

Zusätzlich zu den bereits gefundenen Klassen Komponente, Dekorierer und konkreter Dekorierer zählen alle Unterklassen von Dekorierer als konkrete Dekorierer. Dekoriert werden können alle Unterklassen von Komponente, die dieselbe Schnittstelle besitzen wie Dekorierer, so dass demzufolge alle Klassen, die diese Bedingungen erfüllen, zum DEKORIERER-Muster gehören.

Schwierigkeit: *leicht*.
Laufzeit des Algorithmus: $O(n^2)$.

3.5.4 Fassade

Das Subsystem beim FASSADE-Muster kann jede denkbare Gestalt annehmen; es ist zudem fraglich, ob es sich dabei um ein abgeschlossenes System handelt. Bezüglich der Fassade-Klasse lassen sich zwei Mengen von Klassen bestimmen. Eine Menge A enthält die Klassen, die auf die Fassade zugreifen, eine Menge B enthält die Klassen, auf die die Fassade zugreift, das heißt, die Klassen des Subsystems. Die Klassen des Subsystems wissen nichts von der Fassade; sie besitzen also keine Referenzen darauf. Anzunehmen ist, dass die Klassen der Menge B, also die Klassen, auf die die Fassade zugreift, auch keine Referenzen auf Klassen der Menge A besitzen bzw. Methoden der Klassen aus A aufrufen. Damit sind die für eine Suchanfrage geeigneten Eigenschaften des FASSADE-Musters bereits erschöpft.

Im Folgenden sind die Merkmale noch einmal zusammenfassend aufgeführt:

- Gesucht wird nach der Fassade.
- Eine Menge A von Klassen besitzt Referenzen auf die Fassade.
- Fassade besitzt Referenzen auf ein Subsystem (Menge B).
- Die Subsystemklassen wissen nichts von der Fassade.
- Die Subsystemklassen wissen auch nichts von den Klassen der Menge A, die auf die Fassade zugreifen.

Die folgende Skizze macht die Merkmale bildhaft deutlich (Abbildung 3.10).

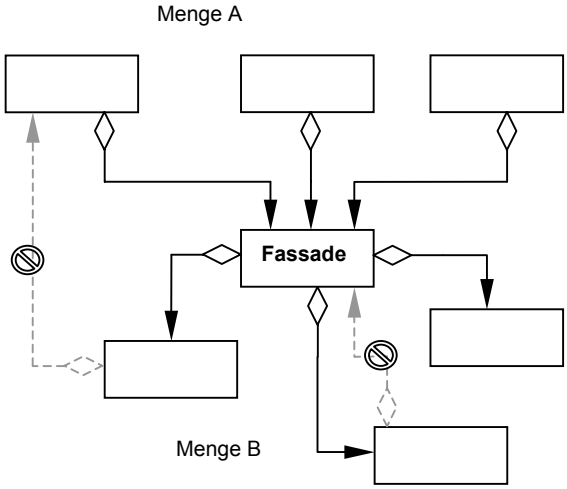


Abbildung 3.10: Merkmale des Fassade-Musters

Ein Algorithmus für die Suche nach dem FASSADE-Muster gestaltet sich folgendermaßen:

Algorithmus 3.10: FindeFassade

```
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 ha-
   ben
   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
```

Im Prinzip ist damit das FASSADE-Muster vollständig erfasst. Eine Schwierigkeit besteht jedoch darin, das Subsystem ausfindig zu machen, da die Fassade nicht auf jede Klasse des Subsystems explizit eine Referenz verwaltet. Zur Erfüllung der Aufgaben, die die Fassade verlangt, wird eben in der Regel nicht das gesamte Subsystem benötigt. Eine Möglichkeit besteht in dem Dazuzählen aller Klassen zu dem Subsystem, die in irgendeiner Form von den Klassen der Menge B Gebrauch machen bzw. von denen die Klassen der Menge B Gebrauch machen. Diese Vorgehensweise setzt allerdings eine Abgeschlossenheit des Subsystems voraus, von der ich hier aber nicht ausgehen möchte. Dieses Problem beeinträchtigt auch die Leistungsfähigkeit der Suchmerkmale.

Schwierigkeit: *schwer*.
Laufzeit des Algorithmus: $O(n^3)$.

3.5.5 Fliegengewicht

Zum FLIEGENGEWICHT-Muster gehört eine Fliegengewichtfabrik, die konkrete Fliegengewichte mittels Fabrikmethoden erzeugt. Das besondere an den verwendeten Fabrikmethoden ist, dass sie nicht exakt dem FABRIKMETHODE-Muster entsprechen, das heißt, sie besitzen als Rückgabetyp nicht eine Oberklasse des zu erzeugenden Objekts sondern genau die Klasse dieses Objekts. Die Fliegengewichtfabrik verwaltet die erzeugten Fliegengewichte durch eine 1-zu-n-Aggregationsbeziehung auf die Fliegengewicht-Klasse. Ein Fliegengewicht zeichnet sich dadurch aus, dass es einen intrinsischen Zustand besitzt, der im Fliegengewicht gespeichert wird, sowie einen extrinsischen Zustand, der vom Kontext abhängt und außerhalb des Fliegengewichtes gespeichert wird. Der extrinsische Zustand wird dem Fliegengewicht per Methodenparameter übergeben. Das gilt für alle Methoden, die das Fliegengewicht implementiert.

Die Merkmale auf einen Blick:

- Gesucht wird nach drei Klassen: der Fliegengewichtfabrik, dem Fliegengewicht und dem konkreten Fliegengewicht.
- Die Fabrik verwendet Fabrikmethoden, die genau das zurückliefern, was sie erzeugen.
- Die Fabrik besitzt eine 1-zu-n-Referenz auf die Fliegengewicht-Klasse.
- Alle Operationen des Fliegengewichts empfangen stets einen bestimmt Parameter (extrinsischer Zustand). Die Methoden dürfen auch zusätzlich zu diesem Parameter weitere Parameter empfangen.
- Ein konkretes Fliegengewicht ist Unterklasse von Fliegengewicht. Ein konkretes Fliegengewicht wird in der Fabrik erzeugt.

Die genannten Merkmale werden durch Abbildung 3.11 veranschaulicht.

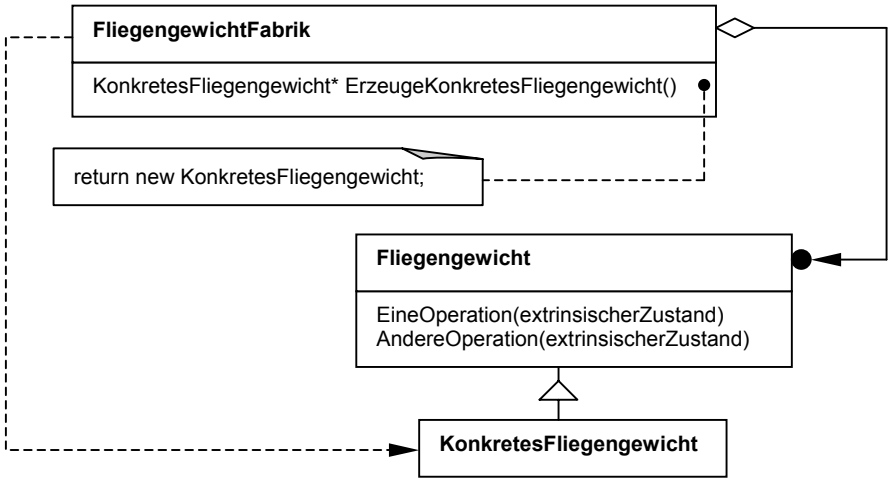


Abbildung 3.11: Merkmale des Fliegengewicht-Musters

Ein Algorithmus dafür skizziert sich folgendermaßen:

Algorithmus 3.11: 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 1-zu-n-Referenz auf eine andere Klasse j
   (Fliegengewicht)?
   ja: gibt es einen Parameter, den alle Methoden der Klasse j gemein-
   sam besitzen?
   ja: für alle Unterklassen k (konkrete Fliegengewichte) der Klasse j
   do
      wird Objekt von k in Klasse i (Fabrik) erzeugt? ja: Muster ge-
      funden
   od
od
```

Um das komplette Muster zu erfassen, werden neben der Fliegengewichtfabrik und der Fliegengewicht-Klasse noch alle konkreten Fliegengewichte benötigt. Ein konkretes Fliegengewicht kann daran erkannt werden, dass es in der Fliegengewichtfabrik erzeugt wird und gleichzeitig die Fliegengewicht-Klasse als Oberklasse besitzt.

Schwierigkeit: *mittel*.
Laufzeit des Algorithmus: $O(n^2)$.

3.5.6 Kompositum

Die Form des KOMPOSITUM-Musters kann sehr vielgestaltig sein. Die in *VisualWorks* verwandte Variante des KOMPOSITUM-Musters [Foote99] hat die in Abbildung 3.12 skizzierte Gestalt.

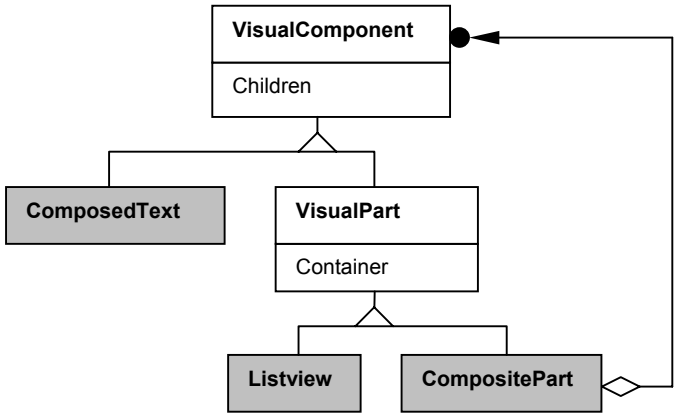


Abbildung 3.12: Kompositum-Muster in VisualWorks

Hier ist zu bemerken, dass die 1-zu-n-Referenz der Kompositum-Klasse sich nicht auf eine direkte Oberklasse beziehen muss. In [Niere+01] wird zudem von einer KOMPOSITUM-Implementierung berichtet, die keine Blatt-Klasse besitzt. Ansonsten ist die Ähnlichkeit zum DEKORIERER-Muster sehr groß, denn auch dort ist eine Referenz von einer Unterklasse zu einer seiner Oberklassen vorhanden. Daher gilt es neben diesem Merkmal vor allem, auf die Unterschiede zu achten und sicherzustellen, dass es sich nicht um ein Vorkommen des DEKORIERER-Musters handelt. Geschehen kann dies zum einen durch eine Unterscheidung der Kardinalität: das KOMPOSITUM-Muster besitzt eine 1-zu-n-Aggregation, das DEKORIERER-Muster dagegen nur eine 1-zu-1-Aggregation. Zum anderen ist zu überprüfen, ob das Hinzufügen von Funktionalität in der beim DEKORIERER-Muster beschriebenen Form (siehe Abschnitt 3.5.3) unterbleibt.

Im Folgenden sind die Merkmale noch einmal zusammengefasst:

- Gesucht wird nach Kompositum-Klasse.
- Kompositum-Klasse besitzt eine 1-zu-n-Aggregationsbeziehung zu einer seiner Oberklassen (Komponente).
- Unterklassen der Kompositum-Klasse, sofern es sie gibt, fügen der Kompositum-Klasse keine Funktionalität hinzu, das heißt, sie rufen keine Methode der Kompositum-Klasse gefolgt von einer eigenen lokalen Methode auf.

Abbildung 3.13 veranschaulicht die angeführten Merkmale.

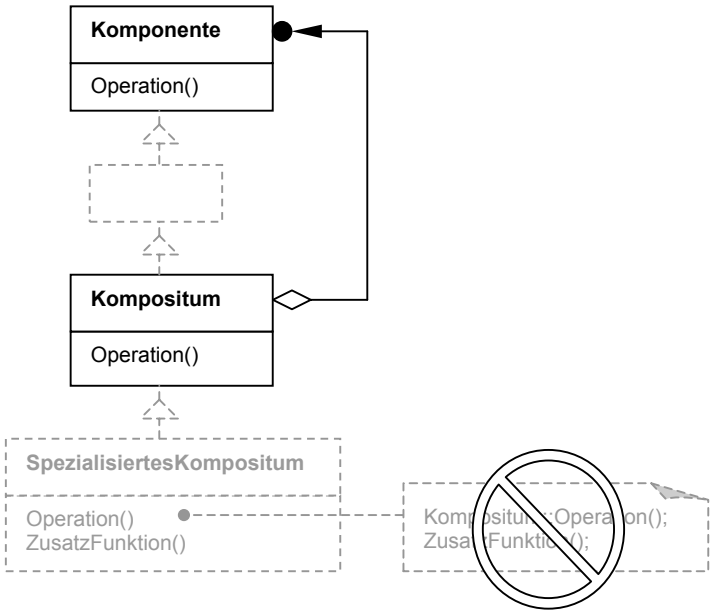


Abbildung 3.13: Merkmale des Kompositum-Musters

Der Algorithmus für das KOMPOSITUM-Muster lautet dann wie folgt:

Algorithmus 3.12: FindeKompositum

```
für jede Klasse i (Kompositum) do
   besitzt Klasse i eine 1-zu-n-Aggregation zu einer Oberklasse (Kom-
   ponente)? 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
```

Zum ganzen Muster gehört schließlich der Baum, den die Komponente als Wurzelklasse bildet.

Schwierigkeit: *leicht*.
Laufzeit des Algorithmus: *O(n)*.

3.5.7 Proxy

Das PROXY-Muster besteht aus den Klassen Subjekt, echtes Subjekt und Proxy. Echtes Subjekt und Proxy erben von Subjekt, und Proxy besitzt eine direkte Referenz auf das echte Subjekt. Dass dies nicht immer so sein muss, zeigt das Beispiel aus [Vlissides99] in Abbildung 3.14, in der eine verzahnte Anwendung des KOMPOSITUM- und des PROXY-Musters zu sehen ist. Die Referenz der Proxy-Klasse (Link) weist hier auf Subjekt (Node), das für die echten Subjekte (File und Directory) stehen kann. Es handelt sich also nur um eine indirekte Referenz der Proxy-Klasse auf das echte Subjekt.

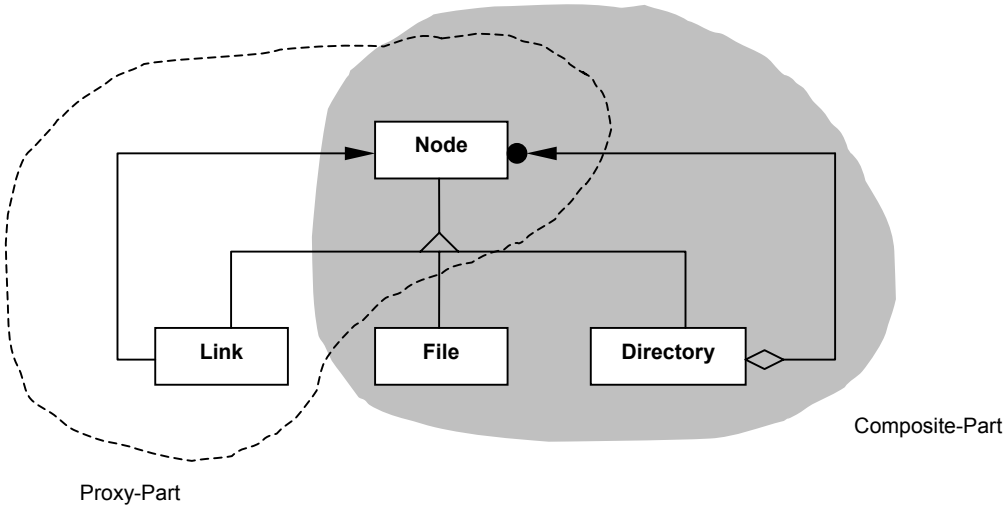


Abbildung 3.14: Beispiel eines Proxy-Musters aus "Entwurfsmuster anwenden" [Vlissides99]

Auch beim PROXY-Muster ist wieder eine Kaskadierung der Vererbung möglich, das heißt, die Klasse, die Subjekt ist, muss nicht direkte Oberklasse von Proxy sein. Deshalb wird diesbezüglich an Proxy lediglich die Bedingung gestellt, dass er Unterklasse einer beliebigen anderen Klasse sein muss. Desweiteren sind alle öffentlichen, in Proxy implementierten Methoden auch im echten Subjekt vorhanden. Nur dadurch kann auch ein Proxy-Objekt für ein echtes Subjekt stehen. In jeder dieser Methoden in Proxy erfolgt ein Aufruf der gleichnamigen Methode des echten Subjekts.

Die Merkmale sind im Folgenden noch einmal zusammengefasst:

- Gesucht wird nach Proxy.
- Proxy ist Unterklasse.
- Proxy besitzt eine Referenz auf eine Klasse, die echtes Subjekt oder nur Subjekt sein kann.

- Alle öffentlichen Methoden von Proxy kommen auch in der Klasse vor, auf die Proxy die Referenz besitzt.
- In jeder dieser Methoden in Proxy erfolgt ein Aufruf der gleichnamigen Methode der Klasse, auf die Proxy die Referenz besitzt.

Damit ist sowohl der Normalfall aus [Gamma+96] als auch der oben beschriebene Fall aus [Vlissides99] abgedeckt. In Abbildung 3.15 sind die eben genannten Merkmale anschaulich dargestellt.

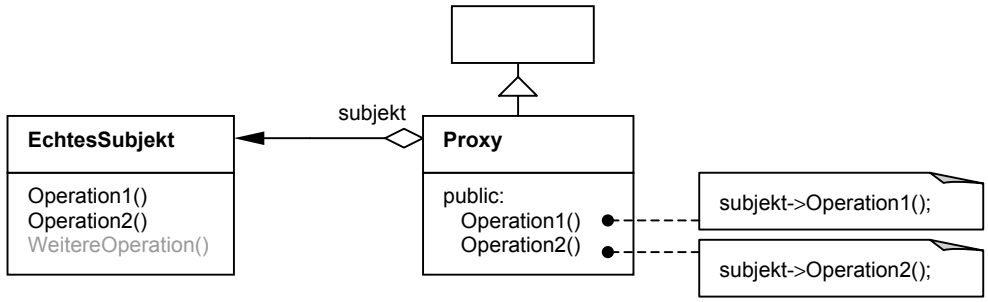


Abbildung 3.15: Merkmale des Proxy-Musters

Ein Algorithmus ließe sich wie folgt skizzieren:

Algorithmus 3.13: FindeProxy

```
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
```

Um das Muster zu vervollständigen, wird neben den bereits gefundenen Klassen Proxy und echtes Subjekt noch die Klasse Subjekt benötigt. Besitzt das echte Subjekt dieselbe Oberklasse wie Proxy, dann ist diese gemeinsame Oberklasse das Subjekt. Ist das echte Subjekt aber Oberklasse von Proxy (siehe obiges Beispiel), so wäre das echte Subjekt in diesem Fall das Subjekt, und als potentiell echte Subjekte müssten alle Unterklassen von Subjekt betrachtet werden. Sogar im Normalfall nach [Gamma+96] wäre es möglich, dass das echte Subjekt, das nicht Oberklasse von Proxy ist, weitere Unterklassen besitzt. Trifft dies zu, so sind diese Unterklassen ebenfalls als potentiell echte Subjekte anzusehen.

Schwierigkeit: *leicht*.
Laufzeit des Algorithmus: *O(n)*.

3.6 Verhaltensmuster

Verhaltensmuster befassen sich mit Algorithmen und der Zuweisung von Zuständigkeiten zu Objekten. Sie beschreiben nicht nur Muster von Objekten oder Klassen, sondern auch die Muster der Interaktion zwischen ihnen; ihr Fokus liegt auf interagierenden Objekten.

Klassenbasierte Verhaltensmuster verwenden Vererbung, um das Verhalten unter den Klassen zu verteilen. Objektbasierte Verhaltensmuster verwenden Objektkomposition anstelle von Vererbung; manche der Muster beschreiben, wie ein Gruppe von Objekten zusammenarbeitet, um eine Aufgabe zu erledigen, die keines der Objekte allein ausführen kann. Besonders wichtig ist hierbei, in welcher Weise zusammenarbeitende Objekte einander kennen: explizite Referenzen aufeinander bedeutet enge Kopplung – im Extremfall kennt jedes Objekt jedes andere. Verhaltensmuster fördern jedoch lose Kopplung. Andere objektbasierte Verhaltensmuster befassen sich mit der Kapselung von Verhalten in einem eigenständigen Objekt und dem Weiterleiten der Operationsaufrufe an dieses Objekt.

3.6.1 Befehl

Dreh- und Angelpunkt des BEFEHLS-Musters ist eine abstrakte Klasse Befehl, die allen konkreten Befehlen gemeinsame Oberklasse ist. Ein Aufrufer besitzt eine Referenz auf die abstrakte Befehls-Klasse. Ein Klient ist dafür zuständig, einen konkreten Befehl zu erzeugen und ihm dabei seinen Empfänger zu übergeben. Dies geschieht in der Regel per Parameterübergabe im Konstruktor des konkreten Befehls. Nach dem Erzeugen des konkreten Befehls weiß dieser um seinen Empfänger, das heißt, er muss eine Referenz auf den Empfänger besitzen. Mit Hilfe dieser Referenz löst der konkrete Befehl eine Aktion beim Empfänger aus.

Im Folgenden sind die Merkmale des BEFEHLS-Musters noch einmal zusammengefasst:

- Gesucht wird nach einer Struktur, die aus mehreren Klassen besteht: dem Aufrufer, dem abstrakten Befehl, einem konkreten Befehl, dem Klienten und dem Empfänger.
- Die Klasse Befehl ist abstrakt.
- Aufrufer besitzt 1-zu-1-Aggregationsbeziehung auf Befehl.
- Konkreter Befehl ist Unterklasse von Befehl.
- Konkreter Befehl besitzt Referenz auf seinen Empfänger. Der Empfänger wird dem konkreten Befehl in seinem Konstruktor übergeben.
- Es gibt einen Klienten, der den konkreten Befehl erzeugt.

Abbildung 3.16 verdeutlicht diese Merkmale.

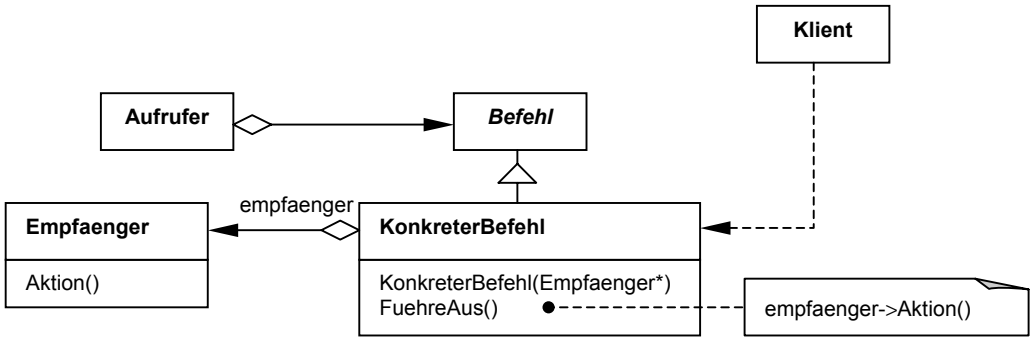


Abbildung 3.16: Merkmale des Befehls-Musters

Ein Algorithmus für die Suche nach dem BEFEHLS-Muster besitzt folgende Form:

Algorithmus 3.14: FindeBefehl

```
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: wei-
         ter
      od
      für alle Klassen n (Klient) do
         erzeugt Klasse n ein Objekt der Klasse k? ja: Muster gefunden
      od
   od
od
```

Um das Muster komplett zu erfassen, sind folgende Schritte auszuführen:

- Zu der Gruppe der konkreten Befehle gehören sämtliche Unterklassen von Befehl.
- Aufrufer sind alle Klassen, die eine Referenz auf Befehl führen.
- Empfänger sind jene Klassen, auf die die konkreten Befehle Referenzen besitzen.
- Klienten sind die Klassen, die einen konkreten Befehl erzeugen.

Schwierigkeit: *mittel*.
Laufzeit des Algorithmus: $O(n^2)$.

3.6.2 Beobachter

Das BEOBACHTER-Muster besteht aus zwei Arten von Klassen: Subjekten und Beobachtern. Dabei ist anzunehmen, dass auf ein Subjekt immer mehrere Beobachter kommen, der umgekehrte Fall jedoch nicht eintritt. Wenn es mehrere Beobachter gibt, ist weiterhin anzunehmen, dass sämtliche Beobachter (die konkreten Beobachter) eine gemeinsame Oberklasse (den abstrakten Beobachter) besitzen. Die Subjekt-Klasse verwaltet eine 1-zu-n-Referenz auf die abstrakte Beobachter-Klasse. Außerdem gibt es in der Subjekt-Klasse zwei Methoden (MeldeAn und MeldeAb), die jeweils einen abstrakten Beobachter als Parameter empfangen. Eine Beobachter-Klasse führt eine Operation zum Aktualisieren seines Zustandes. Diese Operation erfragt den aktuellen Zustand bei seinem Subjekt, auf das der konkrete Beobachter eine Referenz verwalten muss. Da es möglich ist, dass ein konkretes Subjekt sowohl eine Subjekt-Oberklasse besitzt als auch, dass es für sich allein stehen kann, führt die Referenz des konkreten Beobachters entweder zu einer konkreten Subjekt-Klasse, die Unterklasse von Subjekt ist, oder zur Subjekt-Klasse selbst.

Die Merkmale auf einen Blick:

- Gesucht wird nach Subjekt, Beobachter und konkretem Beobachter.
- Subjekt besitzt eine 1-zu-n-Referenz auf Beobachter.
- Subjekt hat zwei Methoden (MeldeAn und MeldeAb), die Beobachter als Parameter empfangen.
- Konkreter Beobachter ist Unterklasse von Beobachter.
- Konkreter Beobachter besitzt eine Referenz auf Subjekt oder eine Unterklasse von Subjekt (konkretes Subjekt).

Die folgende Abbildung 3.17 verdeutlicht die angeführten Merkmale des Beobachter-Musters.

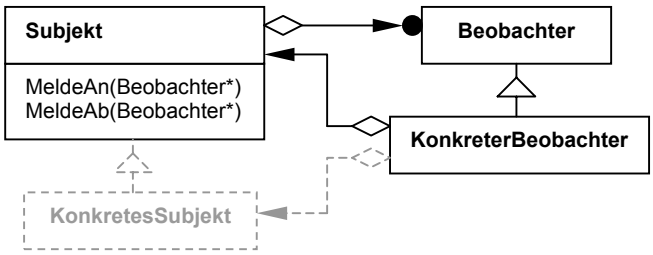


Abbildung 3.17: Merkmale des Beobachter-Musters

Der Algorithmus lautet folgendermaßen:

Algorithmus 3.15: FindeBeobachter

```
für jede Klasse i (Subjekt) do
   besitzt Klasse i 1-zu-n-Aggregation auf andere Klasse k (Beobach-
   ter)?
   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 da-
      von (konkretes Subjekt)?
      ja: Muster gefunden
   od
od
```

Um das Muster vollständig zu erfassen, werden sämtliche Unterklassen von Subjekt und Beobachter ermittelt.

Schwierigkeit: *leicht*.
Laufzeit des Algorithmus: $O(n^2)$.

3.6.3 Besucher

Das BESUCHER-Muster besteht aus mindestens einer Besucher-Klasse sowie einer Klassenstruktur (die Elemente), in der jede Klasse eine Methode besitzt, die ein Besucher-Objekt bzw. eine Referenz darauf als Parameter entgegennimmt. Diese Methode ruft dann mittels der empfangenen Referenz eine entsprechende Operation der Besucher-Klasse auf, wobei sich das zur aufrufenden Methode gehörige Objekt selbst als Parameter übergibt. Die Besucher-Klasse definiert für jede Klasse, die einen Besucher empfangen kann, eine Operation. Das macht es möglich, für eine Klasse häufig eine neue Operation zu definieren ohne dazu die Klasse selbst ändern zu müssen.

Folgend sind die Merkmale für das BESUCHER-Muster zusammengefasst:

- Gesucht wird ausgehend von der Besucher-Klasse.
- Ein Besucher besitzt Operationen, die andere Klassen (die Elemente) als Parameter empfangen.
- Jede dieser Element-Klassen besitzt eine Methode, die die Besucher-Klasse als Parameter empfängt.
- In dieser Methode der Element-Klassen erfolgt ein Aufruf der entsprechenden Methode der Besucher-Klasse. Das Element übergibt sich dabei selbst als Parameter.

Diese Merkmale sind in der folgenden Abbildung 3.18 dargestellt.

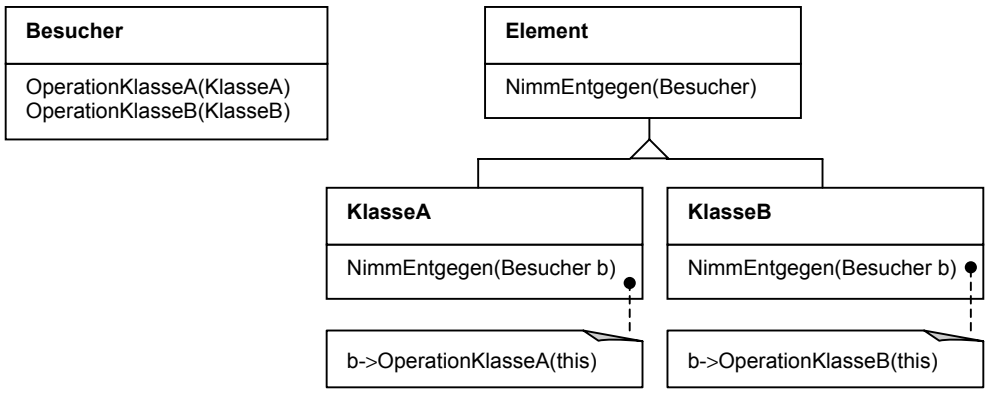


Abbildung 3.18: Merkmale des Besucher-Musters

Der Algorithmus für das BESUCHER-Muster lautet dann wie folgt:

Algorithmus 3.16: FindeBesucher

```
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
```

Um das Muster komplett zu erhalten, ist eine Untersuchung der Methoden der Besucher-Klasse nötig. Treffen die schon oben beschriebenen Eigenschaften zu, so zählt die betreffende Element-Klasse zum Muster dazu. Als konkrete Besucher sind alle Unterklassen von Besucher zu werten.

Schwierigkeit: *leicht*.
Laufzeit des Algorithmus: *O(n)*.

3.6.4 Interpreter

Beim INTERPRETER-Muster handelt es sich um einen Baum, dessen Wurzelklasse abstrakt ist. Alle Unterklassen implementieren eine Operation immer wieder neu. Um beispielsweise zusammengesetzte Ausdrücke bilden zu können, werden Referenzen auf andere Klassen

benötigt. Die Unterklassen verwalten diese Referenzen untereinander jedoch nicht direkt; jede Referenz weist auf die abstrakte Wurzelklasse. Demnach enthält das INTERPRETER-Muster viele Vorkommen des KOMPOSITUM-Musters. Das Verhältnis von einfachen Aggregationsbeziehungen zur Wurzelklasse zur Gesamtzahl der Unterklassen beträgt mindestens 50%.

Die Merkmale nochmals auf einen Blick:

- Gesucht wird nach einem Baum.
- Die Wurzelklasse ist abstrakt.
- Jede Unterklasse implementiert eine Operation immer wieder neu.
- Das Verhältnis einfacher Aggregationsbeziehungen zur Wurzelklasse / Gesamtzahl Unterklassen beträgt mindestens 50%.
- Unterklassen besitzen keine direkten Referenzen aufeinander.

Die folgende Abbildung 3.19 verdeutlicht die Merkmale.

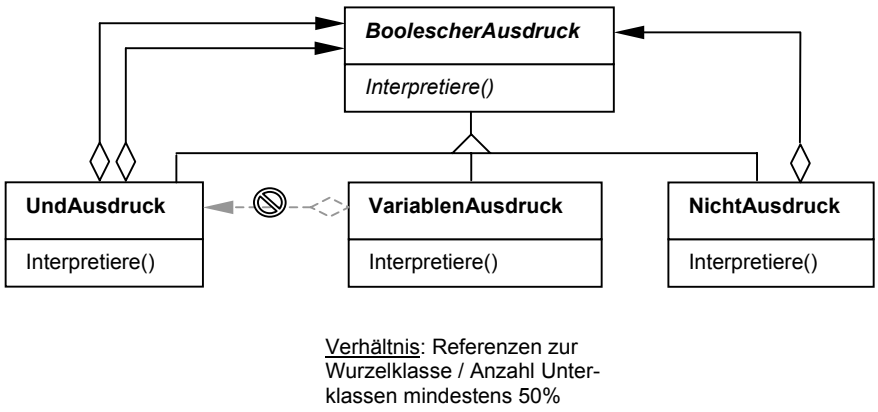


Abbildung 3.19: Merkmale des Interpreter-Musters

Der Algorithmus für die Suche nach dem INTERPRETER-Muster sieht folgendermaßen aus:

Algorithmus 3.17: FindeInterpreter

```
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 Wurzel-
      klasse
      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
```

Das Muster wäre damit bereits komplett erfasst.

Ob die verwendeten Merkmale (Überschreiben der Interpretiere-Operation, Rate der Referenzen zur Wurzelklasse) wirklich stets zutreffen, bedarf einer näheren Untersuchung. Es sind jedoch die einzigen Merkmale des INTERPRETER-Musters, die sich überhaupt für eine Suchanfrage eignen. Sollte sich herausstellen, dass sie nicht in der Mehrheit aller Fälle zutreffen, so würde ich mich Brown [Brown96] anschließen und das INTERPRETER-Muster als unauffindbar bezeichnen.

Schwierigkeit: *schwer*.
Laufzeit des Algorithmus: $O(n^2)$.

3.6.5 Iterator

Die Suche nach dem ITERATOR-Muster stützt sich auf Templates, da dieses Muster in der Regel auf der Verwendung von Templates basiert. Als Minimalstruktur existieren eine Liste sowie ein Iterator, um die Liste zu traversieren. Beide Klassen sind Templates. Das Listen-Template besitzt eine Operation, mit der ein Iterator erzeugt wird. Das Iterator-Template besitzt eine Referenz auf das Listen-Template.

Die Merkmale noch einmal auf einen Blick:

- Gesucht wird nach zwei Templates: Liste und Iterator.
- Iterator hat Referenz auf Liste.
- Liste erzeugt den Iterator in einer Methode.

Veranschaulicht werden die Merkmale durch die Abbildung 3.20.

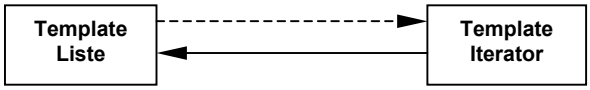


Abbildung 3.20: Merkmale des Iterator-Musters

Der dazugehörige Algorithmus skizziert sich wie folgt:

Algorithmus 3.18: FindeIterator

```
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
```

Eine Untersuchung eventuell vorhandener abstrakter Oberklassen von Iterator und Liste sowie weiterer konkreter Iteratoren bzw. Listen schließt die Suche nach dem ITERATOR-Muster ab.

Schwierigkeit: *leicht*.
Laufzeit des Algorithmus: *O(n)*.

3.6.6 Memento

Das Duden Fremdwörterbuch erklärt das Wort Memento wie folgt: "*Me|men|to [lat.] das; -s, -s: 1. nach dem Anfangswort benanntes Bittgebet für Lebende und Tote in der katholischen Messe. 2. Erinnerung, Mahnung; Denkzettel; Rüge.*" [Duden90] Erinnerung, Mahnung und Denkzettel drücken den Sinn des MEMENTO-Musters recht gut aus.

Das MEMENTO-Muster beinhaltet drei Klassen: den Urheber, das Memento und den Aufbewahrer. Der Aufbewahrer ist für die Aufbewahrung des Mementos zuständig und besitzt demzufolge eine Referenz auf das Memento. Das Memento speichert den internen Zustand des Urhebers, jedoch nur soviel wie nötig. Es besitzt dazu zwei Methoden: eine zum Setzen des Zustandes und eine zum Zurückgeben des gespeicherten Zustandes. Die öffentliche Schnittstelle des Mementos ist nur sehr schmal bzw. gar nicht vorhanden; die private Schnittstelle dagegen beinhaltet wenigstens die eben genannten Methoden sowie einen Konstruktor. Als einzige andere Klasse hat der Urheber das Recht, auf die private Schnittstelle des Mementos (`friend class` Deklaration) zuzugreifen. Der Urheber erzeugt ein Memento, verwaltet aber keine Referenz darauf.

Im Folgenden sind die Merkmale nochmals zusammengefasst:

- Gesucht wird das Memento.
- Es besitzt keinen öffentlichen Konstruktor.
- Memento wird von einem Urheber erzeugt.
- Memento besitzt eine Methode zum Setzen seines Zustandes (Übergabe als Parameter) und eine, die den Zustand zurückliefert (Zustand als Rückgabetyp).
- Urheber darf auf die private Schnittstelle des Mementos zugreifen (`friend class` Deklaration).
- Urheber besitzt keine Referenz auf das Memento.
- Aufbewahrer verwaltet eine Referenz auf das Memento, erzeugt es aber nicht.

Die folgende Abbildung 3.21 veranschaulicht die Merkmale.

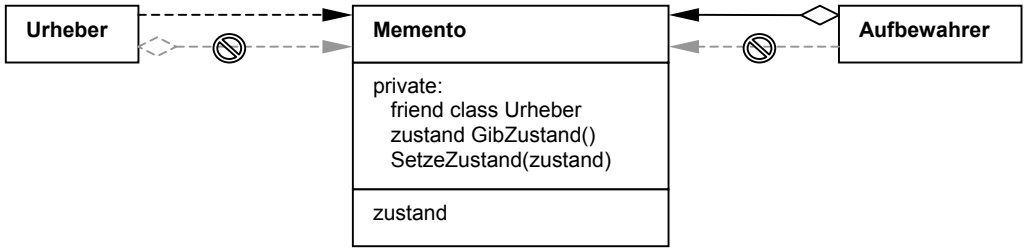


Abbildung 3.21: Merkmale des Memento-Musters

Ein Algorithmus für die Suche nach dem MEMENTO-Muster lautet folgendermaßen:

Algorithmus 3.19: FindeMemento

```
für jede Klasse i (Memento) do
   ist kein öffentlicher Konstruktor vorhanden?
   ja: existiert ein Attribut, das in einer privaten Methode als Para-
   meter übergeben wird?
   ja: existiert eine andere private Methode, die dieses Attribut zu-
   rückliefert?
   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
```

Das Muster wäre somit bereits komplett erfasst.

Schwierigkeit: *leicht*.
Laufzeit des Algorithmus: *O(n)*.

3.6.7 Schablonenmethode

Die Schablonenmethode ist eine nicht polymorphe Methode, die in ihrem Rumpf lokale primitive Operationen aufruft, die in Unterklassen überschrieben werden. Diese primitiven Operationen sind demnach polymorph (*virtual* Deklaration). Sie müssen keineswegs abstrakt sein sondern können stattdessen ein Standardverhalten implementieren. Die Suche richtet sich also nach einer nicht polymorphen Methode (die Schablonenmethode), die in ihrem Methodenrumpf Aufrufe lokaler polymorpher Operationen besitzt.

Die Merkmale auf einen Blick:

- Gesucht wird nach der Schablonenmethode.
- Schablonenmethode ist nicht polymorph.
- Schablonenmethode ruft wenigstens eine lokale polymorphe Operation auf.

Die folgende Abbildung 3.22 zeigt diese Merkmale noch einmal.

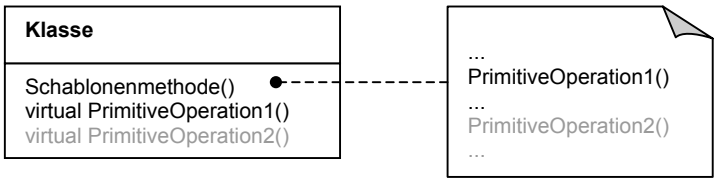


Abbildung 3.22: Merkmale des Schablonenmethode-Musters

Der Algorithmus für das SCHABLONENMETHODE-Muster könnte folgendermaßen aussehen:

Algorithmus 3.20: FindeSchablonenmethode

```
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
```

Um das ganze Muster zu finden, werden alle Unterklassen der gefundenen Klasse darauf untersucht, ob sie wenigstens eine der primitiven Operationen überschreiben.

Schwierigkeit: *leicht*.
Laufzeit des Algorithmus: *O(n)*.

3.6.8 Strategie

Das STRATEGIE-Muster besteht aus einem Kontext, einer abstrakten Strategie und mehreren konkreten Strategien, die allesamt die abstrakte Strategie als gemeinsame Oberklasse besitzen. Um die konkreten Strategien austauschbar zu halten, müssen sie dieselbe öffentliche Schnittstelle besitzen wie die abstrakte Strategie. Kontext hält eine Referenz auf die abstrakte Strategie, aber nicht auf die konkreten Strategien. Da die konkreten Strategien austauschbar sind, werden sie weder untereinander Referenzen besitzen noch Referenzen auf ihre abstrakte Oberklasse verwalten.

Die Merkmale nochmals auf einen Blick:

- Gesucht wird nach einem Baum von Strategie-Klassen.
- Wurzelklasse (abstrakte Strategie) ist abstrakt.
- Alle Unterklassen (konkrete Strategien) besitzen dieselbe öffentliche Schnittstelle wie ihre abstrakte Oberklasse.
- Keine konkrete Strategie besitzt eine Referenz auf die abstrakte Strategie.
- Keine konkrete Strategie besitzt eine Referenz auf eine andere konkrete Strategie.
- Kontext besitzt eine Referenz auf die abstrakte Strategie, aber keine Referenzen auf die konkreten Strategien.

Folgende Abbildung 3.23 verdeutlicht diese Merkmale.

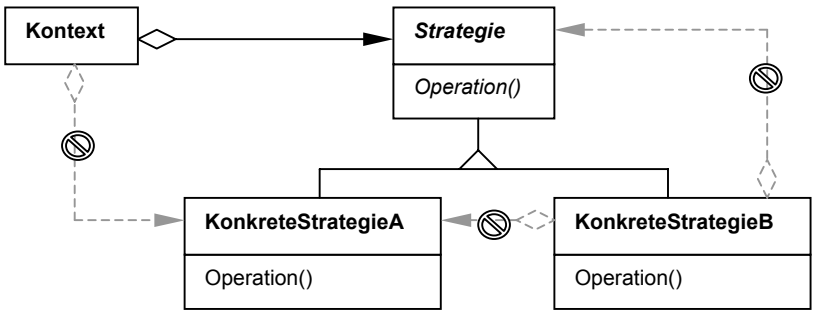


Abbildung 3.23: Merkmale des Strategie-Musters

Ein möglicher Algorithmus lässt sich wie folgt skizzieren:

Algorithmus 3.21: FindeStrategie

```
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 Wur-
      zelklasse?
      ja: besitzt Klasse j eine Referenz auf die Wurzelklasse?
      nein: besitzt Klasse j eine Referenz auf eine andere Unterklas-
      se?
      nein: für alle Klassen k (Kontext) do
         besitzt Klasse k eine Referenz auf die Wurzelklasse des Bau-
         mes i?
         ja: besitzt Klasse k weitere Referenzen auf die Unterklassen
         des Baumes i?
         nein: Muster gefunden
      od
   od
od
```

Damit ist das komplette Muster bereits gefunden.

Schwierigkeit: *mittel*.
Laufzeit des Algorithmus: $O(n^2)$.

3.6.9 Vermittler

Das VERMITTLER-Muster besteht aus zwei Arten von Klassen: Vermittlern und Kollegen. Ein konkreter Vermittler und mehrere konkrete Kollegen sind in jedem Fall vorhanden. Auf einen abstrakten Vermittler und einen abstrakten Kollegen kann auch verzichtet werden. Ein konkreter Vermittler kennt seine Kollegen, das heißt, er verwaltet Referenzen auf sie. Konkrete Kollegen besitzen untereinander keine Referenzen; sie regeln ihre Zusammenarbeit ausschließlich über den konkreten Vermittler. Jeder konkrete Kollege kennt seinen Vermittler. Dies wird jedoch auf unterschiedlichen Wegen realisiert: existiert ein abstrakter Kollege, so besitzt dieser eine Referenz auf den abstrakten Vermittler, sofern es ihn gibt, bzw. auf den konkreten Vermittler. Existiert kein abstrakter Kollege, so verwalten die konkreten Kollegen ihre Referenzen selbst; gibt es einen abstrakten Vermittler, so weisen sie auf diesen, andernfalls direkt auf den konkreten Vermittler. Im Grunde kann hierbei von einem Kreislauf gesprochen werden: von einem konkreten Vermittler existiert eine Referenz auf einen konkreten Kollegen, der wiederum, wenn auch eventuell nicht direkt, wiederum eine Referenz auf den Vermittler besitzt.

Folgend sind die Merkmale des VERMITTLER-Musters noch einmal zusammengefasst:

- Gesucht wird zunächst nach einem konkreten Vermittler.
- Ein konkreter Vermittler besitzt Referenzen auf seine konkreten Kollegen.
- Konkrete Kollegen besitzen untereinander keine Referenzen.

- Existiert ein abstrakter Kollege, so verwaltet dieser eine Referenz auf den abstrakten Vermittler oder direkt auf den konkreten Vermittler, wenn es keinen abstrakten Vermittler gibt.
- Existiert kein abstrakter Kollege, so verwaltet jeder konkrete Kollege eine Referenz auf den abstrakten Vermittler oder direkt auf den konkreten Vermittler, wenn es keinen abstrakten Vermittler gibt.

Das folgende Klassendiagramm veranschaulicht die Merkmale (Abbildung 3.24).

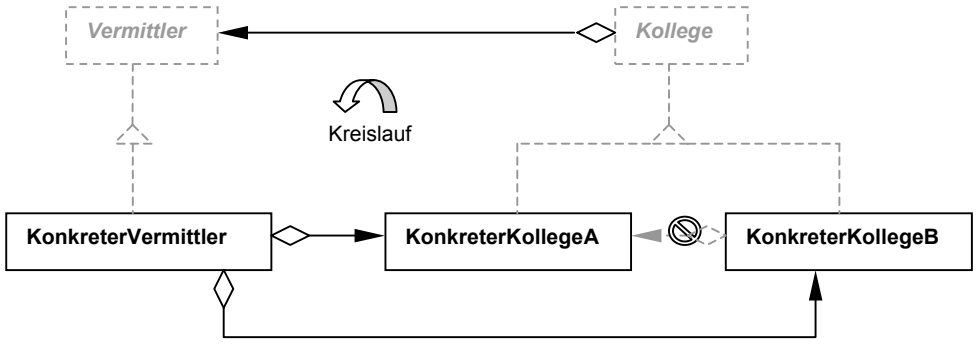


Abbildung 3.24: Merkmale des Vermittler-Musters

Der Algorithmus für das VERMITTLER-Muster skizziert sich wie folgt:

Algorithmus 3.22: FindeVermittler

```
für jede Klasse i (konkreter Vermittler) do
   bilde aus den Referenzen der Klasse i (konkrete Kollegen) eine Men-
   ge 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 Ver-
      mittler)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
```

Das VERMITTLER-Muster vervollständigt sich, indem alle auf den Kreisläufen liegenden Klassen dazugezählt werden.

Schwierigkeit: *mittel*.
Laufzeit des Algorithmus: $O(n^2)$.

3.6.10 Zustand

Das ZUSTANDS-Muster ist dem STRATEGIE-Muster sehr ähnlich. In [Gamma+96] werden zwar Eigenschaften aufgezählt, die das ZUSTANDS-Muster möglicherweise leicht vom STRATEGIE-Muster unterscheidbar machen – beispielsweise sind konkrete Zustände oft SINGLETONS, oder Kontext übergibt sich den konkreten Zuständen häufig selbst als Argument. Jedoch kann nicht davon ausgegangen werden, dass diese Eigenschaften stets vorhanden sind. Das einzige Unterscheidungsmerkmal sehe ich deshalb vorerst nur in der abstrakten Zustands-Klasse, die in der Regel ein Standardverhalten definiert, was die abstrakte Strategie-Klasse nicht tut. Allerdings sind für eine sichere Unterscheidung beider Muster weiterführende Untersuchungen unbedingt nötig.

Die Merkmale entsprechen, wie erwähnt, weitestgehend denen des STRATEGIE-Musters. Der Vollständigkeit halber sind sie hier aber trotzdem noch einmal aufgeführt:

- Gesucht wird nach einem Baum von Zustands-Klassen.
- Wurzelklasse (Zustand) ist nicht abstrakt.
- Alle Unterklassen (konkrete Zustände) besitzen dieselbe öffentliche Schnittstelle wie ihre Oberklasse.
- Kein konkreter Zustand besitzt eine Referenz auf die Wurzelklasse (Zustand) .
- Kein konkreter Zustand besitzt eine Referenz auf einen anderen konkreten Zustand.
- Kontext besitzt eine Referenz auf die Wurzelklasse (Zustand), aber keine Referenzen auf die konkreten Zustände.

Die folgende Abbildung 3.25 verdeutlicht die genannten Merkmale.

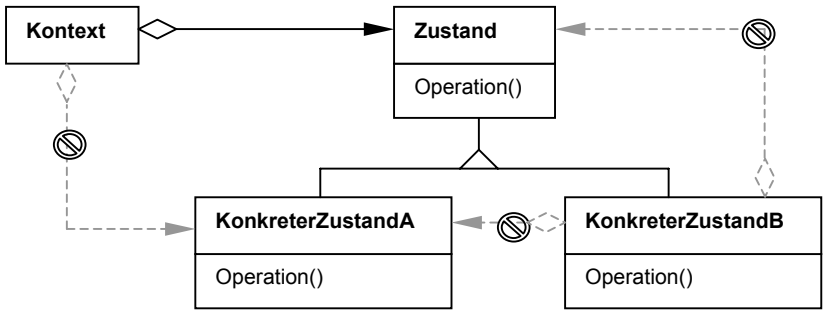


Abbildung 3.25: Merkmale des Zustands-Musters

Der Algorithmus für das ZUSTANDS-Muster funktioniert in derselben Weise wie der beim STRATEGIE-Muster:

Algorithmus 3.23: FindeZustand

```
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 Wur-
      zelklasse?
      ja: besitzt Klasse j eine Referenz auf die Wurzelklasse?
      nein: besitzt Klasse j eine Referenz auf eine andere Unterklas-
      se?
      nein: für alle Klassen k (Kontext) do
         besitzt Klasse k eine Referenz auf die Wurzelklasse des Bau-
         mes i?
         ja: besitzt Klasse k weitere Referenzen auf die Unterklassen
         des Baumes i?
         nein: Muster gefunden
      od
   od
od
```

Damit wäre das komplette Muster erfasst.

Schwierigkeit: *mittel*.
Laufzeit des Algorithmus: $O(n^2)$.

3.6.11 Zuständigkeitskette

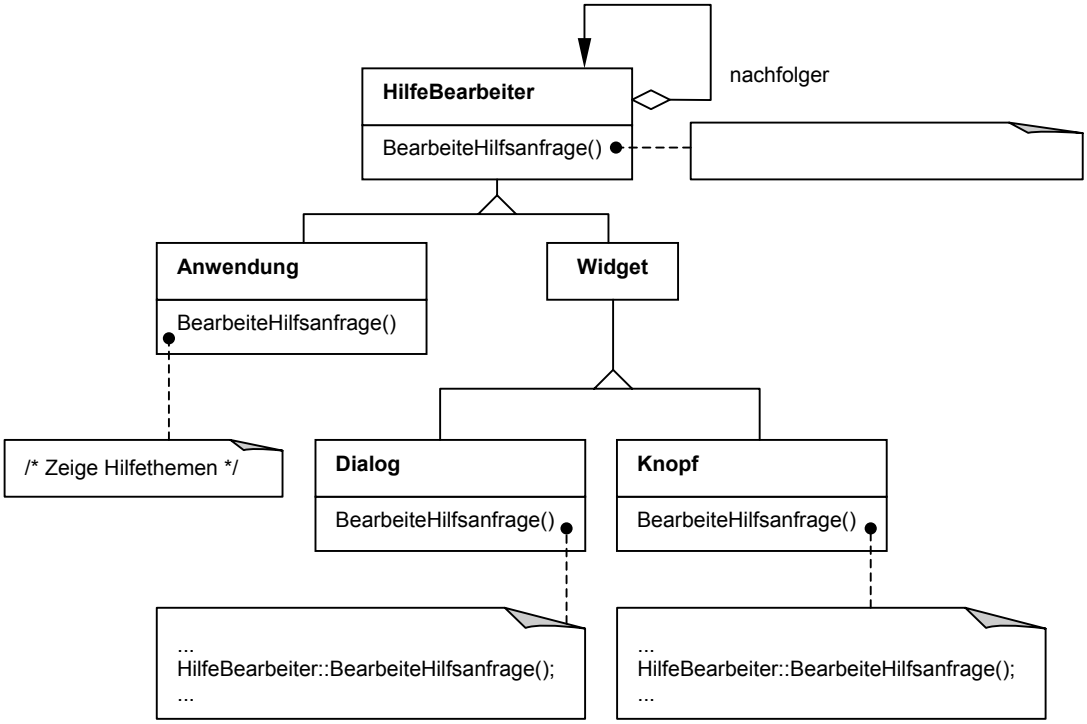
Auch die Teilnehmer des ZUSTÄNDIGKEITSKETTE-Musters können in zwei Arten von Klassen eingeteilt werden: die Klassen der ersten Gruppe sind potentiell in der Lage, eine Hilfeanfrage zu bearbeiten und einen Hilfetext zu liefern, jedoch kann die Anfrage auch an den entsprechenden Nachfolger weitergeleitet werden. Die Klassen der zweiten Gruppe bilden das Ende einer Zuständigkeitskette, das heißt, sie bearbeiten Hilfeanfragen aber leiten keine weiter. Ich vermute, dass etwa 75% aller Klassen, die mit der Hilfebearbeitung zu tun haben, eine eventuelle Weiterleitung der Anfrage vorsehen und etwa 25% aller Klassen das Ende einer Zuständigkeitskette bilden. Kurz: die erste Gruppe steht der zweiten Gruppe mit 75% zu 25% gegenüber. Alle Klassen, die an der Hilfebearbeitung insgesamt beteiligt sind, stecken in einem Baum, dessen Wurzelklasse der Bearbeiter ist. Bearbeiter implementiert die BearbeiteAnfrage-Operation. Nicht alle Klassen dieses Baumes sind an der Bearbeitung einer solchen Anfrage interessiert bzw. benutzen das Verhalten einer Oberklasse. Zu der Berechnung der oben genannten Werte werden nur Klassen herangezogen, die die BearbeiteAnfrage-Operation überschreiben. An der Implementierung dieser Operation ist zu erkennen, ob eine Klasse der ersten oder der zweiten Gruppe angehört. Klassen der ersten Gruppe enthalten einen Aufruf der gleichnamigen Operation einer

anderen Klasse. Klassen der zweiten Gruppe besitzen einen solchen Aufruf nicht. Bei der BearbeiteAnfrage-Operation handelt es sich um die am häufigsten überschriebene Operation in der Baumstruktur.

Im Folgenden sind die Merkmale des ZUSTÄNDIGKEITSKETTE-Musters nochmals zusammengefasst:

- Gesucht wird nach einem Baum.
- Die Wurzelklasse ist der HilfeBearbeiter.
- HilfeBearbeiter implementiert die BearbeiteAnfrage-Operation.
- Die BearbeiteAnfrage-Operation wird nicht in allen Unterklassen überschrieben. Es handelt sich aber um die am häufigsten überschriebene Operation der Wurzelklasse in der Baumstruktur.
- Anhand der BearbeiteAnfrage-Operation können zwei Gruppen von Klassen gebildet werden. Hierbei zählen nur die Klassen, die die BearbeiteAnfrage-Operation überschreiben.
- Zur ersten Gruppe gehören die Klassen, die eine Weiterleitung vorsehen. Das heißt, dass in der BearbeiteAnfrage-Operation ein Aufruf der gleichnamigen Operation einer anderen Klasse erfolgt.
- Zur zweiten Gruppe gehören die Klassen, die keine Weiterleitung vorsehen, und die daher keinen Aufruf einer gleichnamigen Methode in der BearbeiteAnfrage-Operation implementieren.
- Mindestens 75% der betrachteten Klassen gehören der ersten Gruppe an.

Abbildung 3.26 macht die Merkmale anschaulich.



- *BearbeiteHilfsanfrage()* ist die am häufigsten überschriebene Operation der Wurzelklasse im Baum.
- Mindestens 75% der Klassen, die *BearbeiteHilfsanfrage()* überschreiben, sehen eine Weiterleitung vor.

Abbildung 3.26: Merkmale des Zuständigkeitskette-Musters

Ein Algorithmus für die Suche nach dem ZUSTÄNDIGKEITSKETTE-Muster lautet folgendermaßen:

Algorithmus 3.24: FindeZuständigkeitskette

```
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 über-
   schreiben) mindestens 75%?
   ja: Muster gefunden
od
```

Das Muster wäre damit vollständig erfasst.

Schwierigkeit: *mittel*.
Laufzeit des Algorithmus: $O(n^2)$.

3.7 Zusammenfassung und Bewertung

Dieses Kapitel erläuterte meinen Ansatz und gab Merkmale sowie Algorithmen für alle dreiundzwanzig Entwurfsmuster aus [Gamma+96] an. Tabelle 3.4 fasst die Muster mit Angabe ihrer Suchkomplexität, ihrer Schwierigkeit und ihrem Vorteil gegenüber den anderen Ansätzen zusammen.

Muster	**Suchkomplexität**	**Schwierigkeit**	**Vorteil**
Abstrakte Fabrik	O(n)	leicht	kann gegenüber [Kim+00] eindeutig identifiziert werden
Erbauer	O(n)	mittel	von keinem Ansatz außer [Kim+00] abgedeckt
Fabrikmethode	O(n)	leicht	wie [Keller+99]; kann eindeutig identifiziert werden
Prototyp	O(n)	leicht	detailliertere Merkmale
Singleton	O(n)	leicht	von keinem Ansatz außer [Kim+00] abgedeckt; kann gegenüber [Kim+00] eindeutig identifiziert werden
Klassenadapter	O(n)	leicht	
Objektadapter	O(n)	leicht	
Brücke	$O(n^3)$	schwer	bessere Suche durch festgelegte nichtvorhandene Merkmale
Dekorierer	$O(n^2)$	leicht	detailliertere Merkmale
Fassade	$O(n^3)$	schwer	bessere Suche durch festgelegte nichtvorhandene Merkmale
Fliegengewicht	$O(n^2)$	mittel	
Kompositum	O(n)	leicht	bessere Suche durch festgelegte nichtvorhandene Merkmale
Proxy	O(n)	leicht	flexiblere Suche
Befehl	$O(n^2)$	mittel	bessere Suche durch festgelegte nichtvorhandene Merkmale
Beobachter	$O(n^2)$	leicht	flexiblere Suche durch nur eventuell vorhandene Merkmale
Besucher	O(n)	leicht	von keinem Ansatz außer [Kim+00] abgedeckt
Interpreter	$O(n^2)$	schwer	von keinem Ansatz außer [Kim+00] abgedeckt
Iterator	O(n)	leicht	
Memento	O(n)	leicht	von keinem Ansatz außer [Kim+00] abgedeckt

Muster	Suchkomplexität	Schwierigkeit	Vorteil
Schablonenmethode	O(n)	leicht	wie [Keller+99]; kann gegenüber [Kim+00] eindeutig identifiziert werden
Strategie	O(n²)	mittel	bessere Suche durch festgelegte nichtvorhandene Merkmale
Vermittler	O(n²)	mittel	von keinem Ansatz außer [Kim+00] abgedeckt
Zustand	O(n²)	mittel	bessere Suche durch festgelegte nichtvorhandene Merkmale
Zuständigkeitskette	O(n²)	mittel	

Tabelle 3.4: Zusammenfassung der Muster

Mit meinem Ansatz konnten alle Muster abgedeckt werden, was sonst nur der Ansatz von Kim und Boldyreff [Kim+00] zu leisten vermag. Im Unterschied zu letzterem jedoch ist mein Ansatz in der Lage, auch das SCHABLONENMETHODE- und das FABRIKMETHODE-Muster eindeutig zu identifizieren, da dieselbe Suchstrategie wie bei [Keller+99] verwendet wird. Damit wäre die Erfüllung der ersten beiden Forderungen aus Abschnitt *2.7 Präzisierte Problemstellung* gezeigt. Dass die von mir festgelegten Merkmale praktisch umsetzbar sind, wurde durch die jeweilige Angabe eines Algorithmus deutlich – zudem werden im folgenden Kapitel 4 für einige ausgewählte Muster prototypische Implementierungen vorgenommen.

Wie bereits erwähnt, kann die in Abschnitt *2.7 Präzisierte Problemstellung* genannte dritte Forderung, in welchem Maße die bei den bisherigen Arbeiten zu beobachtende geringe Präzisionsrate durch meinen Ansatz verbessert wird, kann in dieser Arbeit nicht getestet werden – dies ist Gegenstand zukünftiger Untersuchungen. Dass die Präzisionsrate jedoch durch die Einführung von Merkmalen, die nicht vorhanden sein dürfen, steigt, ist offensichtlich.

Für die mit *leicht* bewerteten Muster kann ich aufgrund eigener Erfahrung bzw. den Aussagen anderer – wie zum Beispiel [Keller+99] – sagen, dass sich diese eindeutig identifizieren lassen. Meine Erfahrung reicht jedoch nicht aus, um solch ein positives Urteil für alle Muster treffen zu können; diese Muster wurden mit den Schwierigkeiten *mittel* bzw. *schwer* taxiert. Bei diesen Mustern sind tiefergehende Untersuchungen nötig, um die Eindeutigkeit der von mir genannten Merkmale zu überprüfen und gegebenenfalls anzupassen.

Im folgenden Kapitel 4 wird für drei ausgewählte Muster eine prototypische Umsetzung vorgestellt.

4 Prototypische Umsetzung

Dieses Kapitel beschreibt die prototypische Implementierung für die automatische Suche nach Entwurfsmustern, wie sie theoretisch im vorhergehenden Kapitel dargelegt wurde. Als Werkzeug zum Parsen des C++ Quelltextes bot sich *GEN++* an, das schon bei der Realisierung von *SPOOL* [Keller+99] zum Einsatz kam. Jedoch konnte *GEN++* nicht beschafft werden. Als Alternative stand stattdessen das bekannte CASE-Werkzeug *Rational Rose* zur Verfügung, das auch hier am Institut für Theoretische und Technische Informatik verwendet wird.

Zunächst erfolgt eine Erläuterung der grundsätzlichen Verfahrensweise und eine Beschreibung der verwendeten Funktionalität von *Rational Rose*. Im Anschluss daran werden Implementierungen für die Muster SINGLETON, INTERPRETER und KOMPOSITUM vorgenommen und diskutiert.

4.1 Verfahrensweise

Rational Rose ist ein bekanntes CASE-Werkzeug. Mit *Rational Rose* ist es möglich, UML-Diagramme für ein Softwareprojekt zu erstellen und aus diesen Diagrammen eine direkte Umsetzung in ein Codegerüst einer Programmiersprache wie C++ oder Java generieren zu lassen. Zudem kann aber auch der umgekehrte Weg beschritten werden: ein Quelltext-Programm in C++ oder Java liegt vor und wird mit Hilfe eines eingebauten Analyse-Werkzeuges in UML-Klassendiagramme umgewandelt. Diese zweitgenannte Möglichkeit soll hier für die Suche nach Entwurfsmustern in einem Quelltext-Programm Verwendung finden. Der grundsätzliche Ablauf bei der Mustersuche mit *Rational Rose* ist in Abbildung 4.1 schematisch dargestellt.

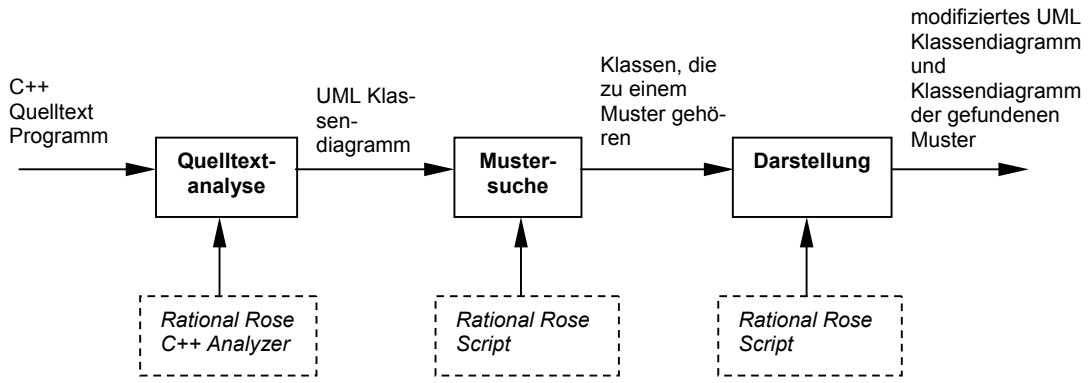


Abbildung 4.1: Verfahrensweise bei der Mustersuche

Eingabe ist ein C++ Quelltext-Programm, das zuerst einer Analyse unterworfen wird. Diese Aufgabe übernimmt der *Rational Rose C++ Analyzer*, der aus dem Quelltext-Programm ein UML-Klassendiagramm generiert. Auf das erstellte UML-Klassendiagramm können nun die Algorithmen für Mustersuche angewendet werden. *Rational Rose* bietet dafür eine Schnittstelle, das *Rose Extensibility Interface*, über das unter anderem der Zugriff auf die geladenen Modelle[18] möglich ist. Dieser Zugriff beinhaltet sowohl das Abfragen von Modellelementen, wie zum Beispiel Klassen und ihre Eigenschaften, als auch das Hinzufügen und Ändern von Modellelementen. Abbildung 4.2 veranschaulicht dies.

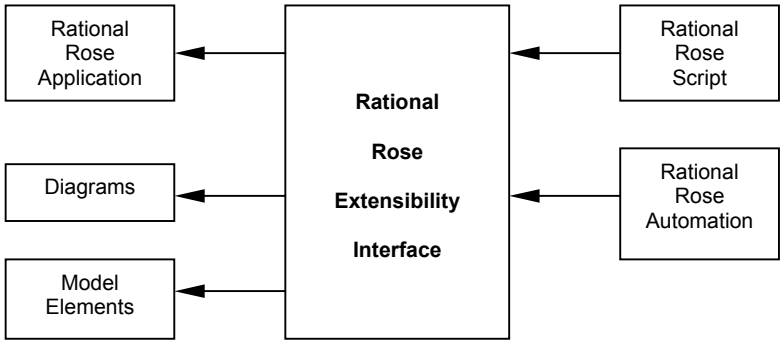


Abbildung 4.2: Rose Application and Extensibility Components

Wie zu sehen ist, gibt es zwei Varianten, mit deren Hilfe über das *Extensibility Interface* ein Zugriff auf die Daten eines Modells vorgenommen werden kann. Zum einen besitzt *Rational Rose* eine eingebaute Skriptsprache, *Rational Rose Script*, die als ein Dialekt der

[18] In diesem Fall besteht das Modell eines Softwaresystems nur aus dem generierten UML-Klassendiagramm.

Basic Programmiersprache anzusehen ist. Zum anderen besteht aber auch die Möglichkeit des Zugriffes von einer anderen Anwendung aus, hier mit *Rational Rose Automation* bezeichnet. *Rational Rose* fungiert in diesem Fall als ein OLE[19]-Server, der für einen OLE-fähigen Klienten Anfragen bearbeitet. Beispielsweise könnte *Microsoft Visual C++* diese Rolle des Klienten einnehmen. Das bedeutet, dass somit Programme, die in *Microsoft Visual C++* geschrieben werden, über das *Rational Rose Extensibility Interface* auf geladene Rational Rose Modelle zugreifen können. Diese Variante sollte hier bei der Mustersuche zunächst zur Anwendung kommen, musste jedoch aufgrund von nicht zu beschaffender Dokumentation schließlich verworfen werden. Daher wird hier die zuerst genannte Variante des Zugriffs auf die Modelle mittels *Rational Rose Script* verwendet.

Aber zurück zur Abbildung 4.1 und der dort dargestellten Verfahrensweise bei der Mustersuche. Nachdem die Quelltext-Analyse ein UML-Klassendiagramm erzeugte, kann nun auf dieses Diagramm mit Hilfe von *Rational Rose Script* zugegriffen werden. Die in Kapitel 3 für die Muster vorgestellten Algorithmen werden dazu in der Skriptsprache formuliert und schließlich auf dem vom *C++ Analyzer* generierten Klassendiagramm ausgeführt. Wurde ein Muster gefunden, so erstellt der Algorithmus eine Menge der Klassen, die an dem Muster beteiligt sind, und übergibt sie dem Modul *Darstellung*, welches dafür zuständig ist, das gefundene Muster für sich allein und innerhalb des gesamten Softwaresystems geeignet darzustellen, um so den Prozess des Programmverstehens möglichst gut zu unterstützen.

Bedauerlicherweise ist die Leistungsfähigkeit des *Rational Rose C++ Analysers* eingeschränkt, das heißt, einige Merkmale, die in Kapitel 3 für die Mustersuche herangezogen werden, können bei der Analyse nicht erkannt werden. Der folgende Abschnitt 4.2 beschreibt diese Einschränkungen ausführlich und erläutert die Auswirkungen.

4.2 Einschränkungen

In Kapitel 3 wurden alle Merkmale beschrieben, die für die Suche nach den Mustern verwendet werden. Leider ist es so, dass einige der Merkmale vom *C++ Analyzer* nicht betrachtet werden bzw. in *Rational Rose* selbst nicht vorgesehen sind. Die folgende Tabelle 4.1 gibt einen Überblick über diese Merkmale.

[19] OLE - *Object Linking and Embedding*. Eine Technologie, die es erlaubt, Elemente verschiedener Anwendungen miteinander zu verknüpfen.

Merkmal	Rational Rose C++ Analyzer
abstrakte Klasse	wird erkannt
konkrete Klasse	wird erkannt
Vererbung	wird erkannt
Attribut (Sichtbarkeit, Typ, Name)	wird erkannt
Operation (Sichtbarkeit, Polymorphie, Rückgabetyp, Name, Parameter, Abstraktheit)	wird erkannt; Ausnahmen: Abstraktheit und Polymorphie werden nicht erkannt; ob eine Operation statisch ist, wird ebenfalls nicht erkannt
Konstruktor (Sichtbarkeit, Name, Parameter)	wird nicht von einer gewöhnlichen Operation unterschieden
Assoziationsbeziehung	wird nicht explizit erkannt; alle Referenzen auf andere Klassen werden als Aggregationsbeziehung gewertet
Aggregationsbeziehung	wird erkannt
Delegation	wird nicht erkannt
Friend-Beziehung	wird nicht erkannt
Objekterzeugung	wird nicht erkannt
Variablenbenutzung	wird nicht erkannt
Methodenaufruf	wird nicht erkannt
Template	wird erkannt

Tabelle 4.1: Fähigkeiten des Rational Rose C++ Analyzers

Wie leicht zu sehen ist, werden einige der Merkmale, die immer wieder bei der Mustersuche auftauchen, nicht erkannt. Als besonders tragisch muss hierbei das Nichterkennen von auftretenden Methodenaufrufen gewertet werden, da es sich um ein Merkmal handelt, das sehr häufig für die Mustersuche eingesetzt wird. Auch die fehlende Objekterzeugung wiegt schwer; die meisten Erzeugungsmuster beruhen darauf.

Aufgrund dieser Mängel ist es nicht möglich, alle in Kapitel 3 vorgestellten Algorithmen mit Hilfe von *Rational Rose Script* zu implementieren. Es bieten sich im Grunde lediglich solche Muster an, die bei der Suche nicht von Methodenaufrufen, Variablenbenutzung und Friend-Beziehungen abhängig sind. Zwei der Muster, auf die dies zutrifft, SINGLETON und INTERPRETER, werden in den beiden folgenden Abschnitten implementiert. Als drittes Muster wurde das KOMPOSITUM gewählt[20], da sich an ihm sehr gut weitere Probleme bei der Mustersuche mit *Rational Rose* aufzeigen lassen.

[20] Obwohl bei ihm teilweise Methodenaufrufe zu prüfen sind.

4.3 Singleton

Zur Erinnerung sind im Folgenden noch einmal die Merkmale für das SINGLETON-Muster aufgeführt:

- Eine Singleton-Klasse besitzt keinen öffentlichen Konstruktor, sondern nur einen `private`- bzw. `protected`-Konstruktor.
- Sie besitzt eine Exemplar-Operation, die `static` ist und als Rückgabetyp die eigene Klasse bzw. eine Oberklasse führt.
- Es existiert eine mit `static` deklarierte Variable vom Typ der eigenen Klasse bzw. einer Oberklasse.

Der Algorithmus war als eine Abfrage der genannten Merkmale für jede Klasse formuliert. Eine vollständige konkrete Umsetzung des Algorithmus' in die *Rational Rose Skriptsprache* kann aus Platzgründen nicht hier aufgeführt werden und wurde stattdessen in den Anhang verlegt (siehe Listing 1 im Anhang C).

Die eigentliche Suche führt die Funktion *FindeSingleton* aus, die als Parameter alle Klassen des Systems entgegennimmt, jede Klasse auf die genannten Eigenschaften überprüft und schließlich das erste gefundene SINGLETON zurückgibt. Die übrigen Funktionen dienen einer einfacheren Realisierung der Suchanfragen in *FindeSingleton*. Der Algorithmus wurde erfolgreich am Beispiel des ZUSTANDS-Musters aus [Gamma+96] erprobt: alle konkreten Zustände konnten als SINGLETONS identifiziert werden.

Zu bemerken ist, dass Attribute, die vom Typ einer anderen Klasse sind, die also Referenzen darstellen, nicht als Attribute gesucht werden können, da sie als solche nicht vorhanden sind, sondern lediglich als Referenzen vorkommen. Desweiteren fällt beim Betrachten des Programmcodes auf, dass selbst die Implementierung dieser einfachen Suchanfrage für das SINGLETON-Muster bereits mehr als zwei Seiten umfasst.

4.4 Interpreter

Die Implementierung für das INTERPRETER-Muster ähnelt der des SINGLETON-Musters, da es sich ebenfalls um eine einfache Überprüfung einiger Merkmale handelt. Der vollständige Quelltext findet sich auch hier im Anhang C (Listing 2).

Kern der Suche ist die Funktion *FindeInterpreter*; die übrigen Funktionen dienen, wie schon beim SINGLETON-Muster, einer übersichtlicheren Implementierung der *FindeInterpreter*-Funktion. Der Algorithmus funktionierte für das Beispiel des INTERPRETER-Musters aus [Gamma+96].

Bis jetzt ging die Suchanfrage davon aus, dass die abstrakte Wurzelklasse des Interpreters wirklich eine Wurzelklasse ist und keine Oberklassen besitzt. Sollte eine nähere Untersuchung des Musters jedoch ergeben, dass dies nicht der Fall sein muss, dass also der Baum des INTERPRETER-Musters ein Teilbaum eines größeren Baumes sein kann, so wäre es nötig, sämtliche Teilbäume zu überprüfen. Das führt unter Umständen zu einer nicht mehr polynomiellen Laufzeit des Algorithmus.

4.5 Kompositum

Die Suchmerkmale für das KOMPOSITUM-Muster sehen zwar eine Überprüfung eventuell vorhandener Methodenaufrufe vor, die hier nicht geleistet werden kann, allerdings eignet es sich hervorragend, um weitere Schwächen von *Rational Rose* bei der Mustersuche darzulegen.

Durch das Weglassen der Untersuchung des möglicherweise vorkommenden Methodenaufrufes einer Unterklasse der KOMPOSITUM-Klasse beschränkt sich die Suche darauf, eine 1-zu-n-Aggregationsbeziehung von einer Klasse zu einer seiner Oberklassen zu finden. Eine vollständige Formulierung dieser Anfrage in *Rational Rose Script* ist im Anhang C, Listing 3, zu sehen.

Die Implementierung der Anfrage obliegt hauptsächlich der Funktion *FindeKompositum*. Alle weiteren Funktionen führen lediglich, wie schon beim SINGLETON- und beim INTERPRETER-Muster, unterstützende Aufgaben aus. Das in [Gamma+96] gezeigte Beispiel für das ABSTRAKTE-FABRIK-Muster brachte die Identifizierung eines KOMPOSITUM-Musters. Die Anwendung der Anfrage auf das Beispiel für das KOMPOSITUM-Muster aus [Gamma+96] dagegen schlug fehl. Abbildung 4.3 macht deutlich, warum.

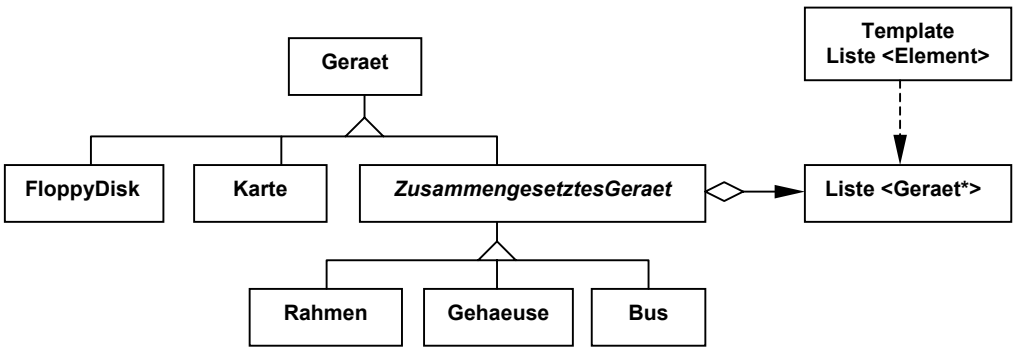


Abbildung 4.3: Beispiel des Kompositum-Musters aus [Gamma+96]

Die 1-zu-n-Aggregationsbeziehung von *ZusammengesetztesGeraet* zu *Geraet* wurde mit Hilfe eines Templates *Liste* realisiert. Diese Liste verwaltet Objekte vom Typ *Geraet*, was

völlig korrekt ist. Der Problem besteht in dem *Rational Rose C++ Analyzer*, der ebenfalls korrekt, eine Aggregationsbeziehung von *ZusammengesetztesGeraet* zu einer Klasse *Liste* konstatiert, die mit dem Typ *Geraet* parametrisiert wurde. Dadurch jedoch kann eine Suche gemäß dem angeführten Algorithmus nicht erfolgreich verlaufen. Die folgende Abbildung 4.4 zeigt ein weiteres Beispiel dieser Problematik. Es handelt sich um eine Realisierung in *Microsoft Visual C++*.

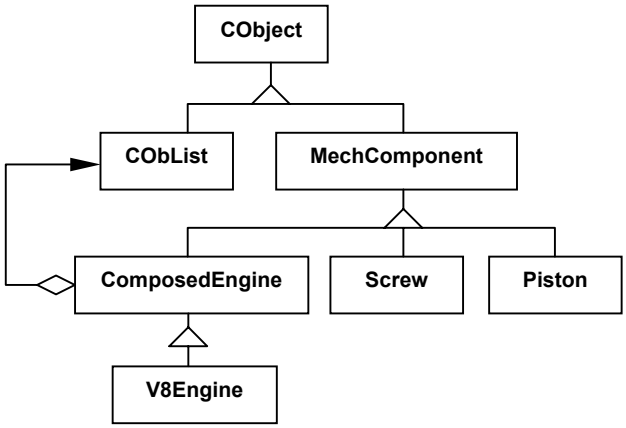


Abbildung 4.4: Ausprägung des Kompositum-Musters in Microsoft Visual C++

Hier erben alle Klassen von *CObject*. *ComposedEngine* ist die Kompositum-Klasse, die eine Liste von Objekten des Typs *CObject* verwaltet. Dies geschieht jedoch durch die Verwendung der Klasse *CObList* (dargestellt durch die Aggregationsbeziehung von *ComposedEngine* auf *CObList*). Es ist offensichtlich, dass der vorgestellte Algorithmus auch hier versagen muss.

Die beiden gezeigten Beispiele machen deutlich, dass *Rational Rose* nicht für die Suche nach Entwurfsmustern ausgelegt ist, denn es wäre gerade dafür wünschenswert, dass die angeführten Realisierungen einer Aggregationsbeziehung erkannt und in der für den Algorithmus passenden Art und Weise dargestellt werden würden. Möglich wäre zwar eine solche Anpassung per Hand, jedoch ist dies sehr mühsam. Die bessere Alternative bei der weiteren Verwendung von *Rational Rose* wäre eventuell ein weiteres Skript, das diese Arbeit automatisch erledigen kann. Eine mögliche Erweiterung von *Rational Rose* wird daher als zukünftige Aktion gesehen.

4.6 Darstellung

Im Rahmen des Programmverstehens kommt der Darstellung der gefundenen Muster eine herausragende Bedeutung zu. Nur eine geeignete Darstellung trägt zu einer Beschleuni-

gung des Programmverstehens bei [Keller+99]. In diesem Abschnitt werden Möglichkeiten gezeigt, wie vorteilhafte Aufbereitungen der Suchergebnisse aussehen können.

Zunächst ist es natürlich wichtig, eine Detailansicht für das gefundene Muster bereitszustellen. Dazu gehören alle Klassen und ihre Assoziations- und Vererbungsbeziehungen sowie eine Erläuterung, welche Rolle jede Klasse in dem vorliegenden Muster spielt. Die folgende Abbildung 4.5 zeigt eine solche Ansicht am Beispiel des OBJEKTADAPTER-Musters.

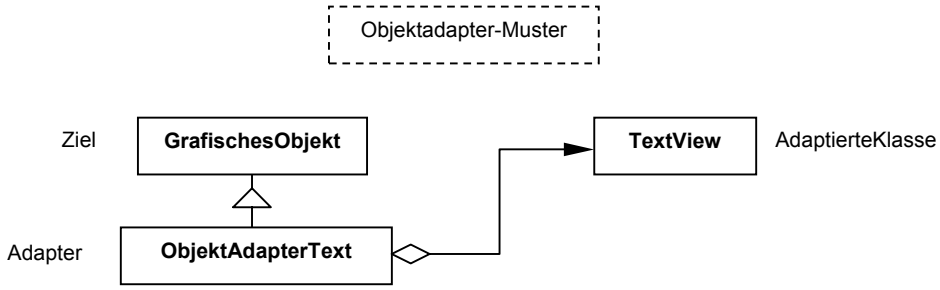


Abbildung 4.5: Detailansicht eines gefundenen Objektadapter-Musters

Neben dem Wissen um die Aufgabe und Rolle jeder einzelnen, zu einem Muster gehörenden Klasse ist es außerdem wichtig, die Rolle des Musters innerhalb des Gesamtsystems verstehen zu können. In [Keller+99] (*SPOOL*) werden dazu zwei Ansichten verwendet. Die erste, das sogenannte *Tree-Layout*, stellt alle Klassen des Gesamtsystems in einem Vererbungsbaum dar; die zum gefundenen Muster gehörenden Klassen sind dunkel eingefärbt. Abbildung 4.6 zeigt ein Beispiel dafür.

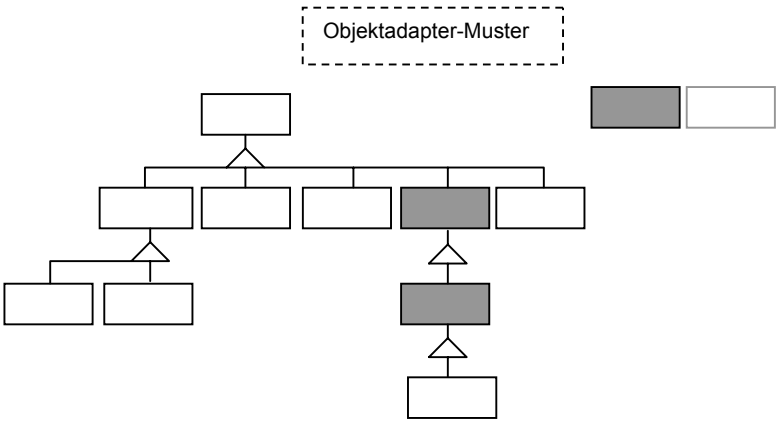


Abbildung 4.6: Tree-Layout

Die zweite verwendete Ansicht, das *Spring-Layout*, zeigt im Prinzip denselben Sachverhalt wie beim *Tree-Layout*, jedoch ist beim *Spring-Layout* die Wurzelklasse als eine Art Quelle anzusehen, von der die Vererbungspfade sich im Stile von kleinen Bächen in alle Richtungen ausbreiten. Abbildung 4.7 verdeutlicht dies.

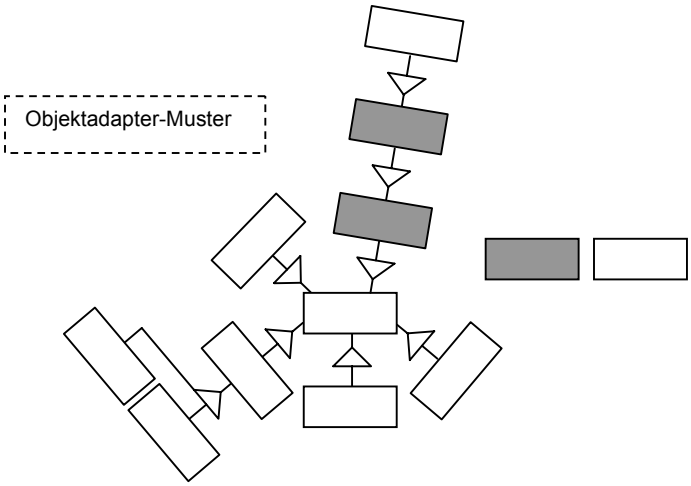


Abbildung 4.7: Spring-Layout

Einige der in Kapitel 2 vorgestellten Software-Werkzeuge verfügen außerdem über ein *Muster-Repository*, das im Prinzip die Erklärungen zu den Mustern, wie sie auch in [Gamma+96] nachzulesen sind, enthält. Dadurch kann sich der Software-Ingenieur beim Programmverstehen stets noch einmal über das jeweilige Muster informieren, ohne immer das Buch zur Hand haben zu müssen.

Eine Umsetzung der genannten Ansichtsformen in *Rational Rose* wurde bislang nicht vorgenommen. Für das *Spring-Layout* ist dies auch gar nicht möglich. Die beiden anderen jedoch, die Detailansicht und das *Tree-Layout*, ließen sich recht gut in *Rational Rose* implementieren.

4.7 Fazit

Dieses Kapitel behandelte eine teilweise prototypische Umsetzung des in Kapitel 3 gemachten Ansatzes für die Suche nach Entwurfsmustern. Als Werkzeug dafür diente *Rational Rose*. Es zeigte sich, dass Rational Rose in der Analyse des Quelltext-Programms Schwächen aufweist, deretwegen eine ernsthafte Suche nach vielen Mustern nicht möglich ist. Für die Muster SINGLETON, INTERPRETER und KOMPOSITUM wurde trotzdem eine Implementierung der jeweiligen Algorithmen in *Rational Rose Script* vorgenommen. Eine Besprechung geeigneter Darstellungsformen für gefundene Muster schloss sich daran an.

Festzustellen bleibt, dass sich *Rational Rose* aufgrund der genannten Defizite, die hauptsächlich der Analysephase zuzuordnen sind, für die automatische Mustersuche disqualifiziert. Hier müssen also bessere Werkzeuge gefunden bzw. entwickelt werden. Recht vielversprechend scheint das Analysewerkzeug *GEN++* zu sein, welches Keller, Schauer, Robitaille und Pagé für die Implementierung ihres *SPOOL*-Systems verwenden, das jedoch für diese Arbeit nicht verfügbar war.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde das automatische Auffinden von Entwurfsmustern in einem in C++ implementierten Softwaresystem behandelt. Das einführende Kapitel erläuterte, welch hohen Stellenwert das Programmverstehen im Bereich der Wartung einnimmt, die wiederum selbst im Software-Lebenszyklus gesehen etwa die Hälfte aller anfallenden Kosten trägt.

Entwurfsmuster sind elegante Lösungen zu immer wiederkehrenden Problemen, die durch den Transport ihrer Idee für das Programmverstehen eine zentrale Rolle einnehmen. Sie bieten eine hervorragende Möglichkeit, einem Softwareentwickler, der die Funktionsweise eines Programms erfassen soll, zu vermitteln, welche Aufgabe bestimmte Klassen, die zu einem Muster gehören, erfüllen und welche Vor- und Nachteile diese Lösung mit sich bringt.

Für den Menschen ist es sehr schwierig bis unmöglich, alle Mustervorkommen aus einem großen Softwaresystem herauszufiltern. Dadurch ist die Unterstützung durch ein Software-Werkzeug unbedingt nötig.

Kapitel 2 stellte Ansätze zur automatischen Mustersuche vor. Es existieren verschiedene Arten von Ansätzen: Eine Gruppe verwendet minimale Schlüsselmerkmale (*DP++, KT, SPOOL*), eine zweite Gruppe sucht nach vollständigen Übereinstimmungen in der Klassenstruktur (*Pat, IDEA, Mehrstufiger Suchprozeß*), ein dritter aber noch nicht implementierter Ansatz begegnet der Vielgestaltigkeit der Muster mit Hilfe einer flexiblen Musterdefinition und Fuzzylogik, ein vierter Ansatz charakterisiert alle Muster anhand von Metriken und führt auf dieser Basis Vergleiche aus (*Pattern Wizard*), eine letzte Arbeit stellt eine induktive Methode für die Mustersuche von Hand vor (*BACKDOOR*). Die gemachten Ansätze weisen verschiedene Defizite auf: *Pattern Wizard* allein ist in der Lage, sämtliche Muster aus [Gamma+96] zu finden, versagt jedoch beim eindeutig definierten SCHABLONENMETHODE-Muster. Die Arbeiten der ersten und zweiten Gruppe decken jeweils nur einen Teil der Muster ab. Die erzielte Präzisionsrate der entwickelten Ansätze ist durchweg sehr niedrig. Aus den Schwächen dieser Ansätze konnte eine präzisierte Problemstellung formuliert werden: Es ist erstens eine Abdeckung aller Muster gefordert, zweitens eine sichere Identifizierung von eindeutig definierten Mustern wie dem SCHABLONENMETHODE- und dem FABRIKMETHODE-Muster und drittens eine Steigerung der Präzisionsrate.

Diese Forderungen angehend, stellte Kapitel 3 meinen eigenen Ansatz vor, der darauf beruht, für jedes Muster Merkmale festzulegen, die stets bei einer Anwendung des jeweiligen Musters vorhanden sein sollten. Hinzu kommt die Überprüfung von Merkmalen, die notwendigerweise nicht vorliegen dürfen, damit es sich um das entsprechende Muster handelt. Letztere Merkmale führen zu keiner positiven Identifizierung eines Musters, senken aber die Rate der falsch erkannten Muster (Fall 2: *positive false*). Das Problem einer geeigneten Darstellung solcher Merkmale führte zu einer Erweiterung der *Object Modeling Technique* (OMT). Für jedes Muster wurden seine Merkmale erklärt und anhand eines OMT-

Diagramms anschaulich dargestellt. Ein Algorithmus skizzierte jeweils, in welcher Weise eine Suche nach den genannten Merkmalen vollzogen werden kann. Ein Vergleich meines Ansatzes mit den Arbeiten aus Kapitel 2 ergab eine teilweise Erfüllung der in der präzisierten Problemstellung entwickelten Forderungen: dass mein Ansatz alle Vorkommen aller Muster finden kann und beim eindeutig definierten SCHABLONENMETHODE-Muster nicht versagt, konnte gezeigt werden. Eine Untersuchung der Höhe der Präzisionsrate meines Verfahrens musste offen bleiben.

Kapitel 4 schließlich nahm eine prototypische Umsetzung der in Kapitel 3 genannten Merkmale mit Hilfe des CASE-Werkzeuges *Rational Rose* vor. Dabei musste festgestellt werden, dass *Rational Rose* gewisse Defizite aufweist, die zum größten Teil in der Analyse des zu untersuchenden C++ Quelltextes liegen. Diese Mängel besagen, dass einige wichtige Merkmale, die unbedingt für eine ernsthafte Mustersuche vonnöten sind, beispielsweise das Erkennen von Methodenaufrufen, nicht gefunden werden können. Für zwei Muster, die nicht davon betroffen sind, das SINGLETON- und das INTERPRETER-Muster, wurde eine Umsetzung der in Kapitel 3 vorgestellten Algorithmen in die *Rational Rose* Skriptsprache vorgenommen. Für das KOMPOSITUM-Muster, welches ebenfalls konkret implementiert wurde, mussten bei der Überprüfung der Merkmale Abstriche gemacht werden. Kapitel 4 zeigte überdies Möglichkeiten zu einer geeigneten Darstellung gefundener Muster auf.

Aus den vorgenommenen Untersuchungen ergeben sich unmittelbar einige Themen, die weiterführend zu bearbeiten sind:

- Es muss ein besseres Werkzeug für die konkrete Implementierung der Algorithmen gefunden bzw. entwickelt werden; das Analyse-Werkzeug *GEN++*, das bei *SPOOL* [Keller+99] zum Einsatz kommt, scheint dazu geeignet.
- Sobald ein besseres Analyse-Werkzeug zur Verfügung steht, können alle weiteren Algorithmen implementiert werden.
- Danach ist es möglich, das vollständig implementierte Software-Werkzeug einem umfangreichen Test zu unterziehen, wie er zu Beginn des Kapitels 2 beschrieben wurde.
- In Kapitel 3 zeigte sich, dass meine Erfahrung nicht ausreicht, um alle Muster mit der Schwierigkeit *leicht* einschätzen zu können. Für die mit *mittel* bzw. *schwer* eingestuften Muster ist es daher nötig, weitergehende Untersuchungen durchzuführen, um die genannten Merkmale zu validieren oder gegebenenfalls zu revidieren. Bei dieser tiefergehenden Untersuchung handelt es sich allerdings um eine sehr zeitaufwendige Tätigkeit. Einfachster und wohl auch einziger Weg dafür ist eine Analyse vieler verschiedener Implementierungen eines Musters, am besten aus ganz unterschiedlichen Quellen. In die Abschnitte zum PROXY- und zum KOMPOSITUM-Muster flossen bereits solche Fremdimplementierungen ein. Es ist nicht einfach, an viele verschiedene Implementierungen eines bestimmten Musters zu gelangen. Das Internet bietet zwar recht viele Seiten, auf denen Entwurfsmuster erklärt werden, jedoch geschieht dies stets nur anhand der Erläuterungen, die auch in [Gamma+96] nachzulesen sind. Eine weitere Möglichkeit wäre, sich selbst eine vielfältige Menge an Implementierungsvarianten eines Mus-

ters zu überlegen; solche Überlegungen wären aber nicht praxisorientiert, dass heißt, die verschiedenen Implementierungen würden nicht zur Lösung eines echten Problems erdacht werden und hätten darum keinen besonders hohen Wert. Desweiteren besitzt jeder Mensch einen bestimmten Denkstil, der ihn davon abhält, sich wirklich sehr verschiedene Umsetzungen eines Musters zu überlegen. Das Ergebnis wären sicherlich lauter Muster, die sich einander in einer bestimmten Weise ähneln würden. Daraus resultiert, dass diese Arbeit also nur von einem Team von Software-Entwicklern gut gelöst werden kann.

Weiterführend sind folgende Themen näher zu untersuchen:

- Es kann sinnvoll sein, verwendete Klassen- oder Methodennamen für die automatische Mustersuche heranzuziehen. In [Gamma+96] werden teilweise sogar Namenskonventionen vorgeschlagen; beispielsweise sollte eine Fabrik des ABSTRAKTE-FABRIK-Musters stets das Suffix *-Fabrik* tragen.
- Inwieweit treffen die genannten Merkmale auch für andere Programmiersprachen wie beispielsweise Java oder Delphi zu? Müssen die Merkmale für diese Sprachen abgeändert werden?

Alle Autoren der in Kapitel 2 gemachten Ansätze sind sich einig, dass die automatische Suche nach Entwurfsmustern nicht mit völliger Sicherheit geschehen kann, sondern dass stets im Nachhinein eine Überprüfung durch einen Softwareentwickler erfolgen muss. Es ist möglich, dass auch tiefergehende Untersuchungen der von mir vorgeschlagenen Merkmale zu diesem Ergebnis gelangen. Daher ist zu einer Kommentierung verwendeter Entwurfsmuster im Quelltext auf jeden Fall geraten. Dies sollte jedoch möglichst nicht nur durch Kommentare geschehen; stattdessen wäre eine feste Verankerung im Quelltext wünschenswert. Solche Sprachkonstrukte sind bisher allerdings in den gängigen Programmiersprachen nicht vorhanden. Tatsubori und Chiba stellen als Alternative dazu in [Tatsubori+98] ihren *OpenJava* genannten Ansatz vor. *OpenJava* ist eine Erweiterung der Java-Programmiersprache. Es verwendet ein *Präcompiling*, das OpenJava-Quellcode mit Hilfe einer Metaklasse in gewöhnlichen Java-Quellcode übersetzt. Aus diesem gewöhnlichen Code wird anschließend von einem gängigen Java-Compiler der Bytecode erzeugt. Das folgende Beispiel veranschaulicht diese Vorgehensweise am Beispiel des OBJEKTADAPTER-Musters:

Gegeben sind ein Interface *Stack* und eine Klasse *Vector*. *Vector* beinhaltet die gewünschte Funktionalität und soll an *Stack* angepasst werden. *Vector* entspricht folglich der adaptierten Klasse und *Stack* der Ziel-Klasse. Um *Vector* an *Stack* anzupassen, wird eine Klasse *VectorStack* gebildet, die den Adapter darstellt (Listing 5.1).

Listing 5.1: VectorStack.oj

```
public class VectorStack
   instantiates AdapterPattern
   adapts Vector to Stack
{
   Object peek() forwards lastElement;
   void push(Object o) forwards addElement;
   Object pop() { ... }
}
```

Das Listing zeigt eine Implementierung in *OpenJava*; die feste Verankerung des Musters im Quelltext ist deutlich zu sehen. Um diese Implementierung der Klasse *VectorStack* in Bytecode übersetzen zu können, wird eine Metaklasse zu Hilfe genommen, durch die eine Übertragung der erweiterten musterspezifischen Konstrukte in gewöhnlichen Java-Code vorgenommen werden kann.

Solange sich diese Vorgehensweise jedoch nicht allgemein etabliert hat bzw. solange Konstrukte, die der Implementierung von Mustern dienlich sind, nicht in die gängigen Programmiersprachen integriert werden, ist eine Dokumentation durch Kommentare ratsamer.

Um beim Programmverstehen Nutzen aus Entwurfsmustern ziehen zu können, ist eine umfassende Kenntnis dieser Muster unbedingte Voraussetzung für den Software-Ingenieur. Es ist nicht ausreichend für ihn, nur schemenhaftes Wissen über Entwurfsmuster zu besitzen, sondern er muss in ihrer Anwendung sicher sein. Zu einem solchen Bildungsstand zu gelangen, dauert jedoch sehr lange. Die Lehre an den Universitäten ist hier gefordert, ihren Anteil für die Erreichung dieses Zieles zu leisten. Bislang wird dem zumindest an unserer Universität nur ungenügend nachgekommen, was dadurch belegt wird, dass ein Student der Informatik sein Studium abschließen kann, ohne zu wissen, dass es Entwurfsmuster überhaupt gibt. Da es sich bei diesen Mustern aber um Konzepte handelt, die sehr, sehr viele Softwareentwickler täglich verwenden und die über eine längere Zeit von verschiedenen Experten weiterentwickelt und wieder neu durchdacht wurden, ist es auf jeden Fall lohnenswert, Entwurfsmuster stärker in die Lehre zu integrieren.

A Beschreibungen der Entwurfsmuster

Dieser Anhang enthält die Beschreibungen aller dreiundzwanzig Entwurfsmuster aus [Gamma+96] in kompakter Form. Jedes Muster wird anhand seines Zweckes, seiner Struktur und einer kurzen Erklärung beschrieben. Der Zweck ist jeweils direkt aus [Gamma+96] übernommen. Die Struktur folgt der Notation der *Object Modeling Technique* (OMT) von James Rumbaugh; eine Beschreibung dieser Notation ist in Anhang B zu finden. Direkt unter der Überschrift eines jeden Musters ist der jeweilige englische Name aufgeführt; unter der dahinterstehenden Seitennummer kann eine ausführlichere Beschreibung des Musters in [Gamma+96] nachgelesen werden. Die Unterteilung der Muster erfolgt gemäß [Gamma+96] anhand ihrer Zugehörigkeit zu den Gruppen der Erzeugungs-, der Struktur- und der Verhaltensmuster.

A.1 Erzeugungsmuster

Erzeugungsmuster dienen der Erzeugung von Objekten, wobei sie den Erzeugungsprozess verstecken. Sie helfen dabei, ein System unabhängig davon zu machen, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden. Alles was die Anwendung insgesamt über ihre Objekte weiß, wird durch die von abstrakten Klassen definierten Schnittstellen bestimmt. Erzeugungsmuster ermöglichen somit zu bestimmen, was erzeugt wird, wer es erzeugt, wie es erzeugt wird und wann es erzeugt wird.

Erzeugungsmuster sind vor allem dann von Bedeutung, wenn Systeme beginnen, mehr von *Objektkomposition* als von Vererbung abzuhängen. Objektkomposition und Vererbung sind die beiden bekanntesten Techniken für die Wiederverwendung von Funktionalität in objektorientierten Softwaresystemen. Bei der Objektkomposition (auch Black-Box-Wiederverwendung genannt) wird neue komplexe Funktionalität durch das Zusammenführen von Objekten erreicht; sie erfolgt dynamisch zur Laufzeit, indem Objekte Referenzen auf andere Objekte erhalten. Vererbung (auch White-Box-Wiederverwendung genannt) dagegen wird statisch zur Übersetzungszeit ausgeführt. "*Bei der Objektkomposition bewegt sich die Konzentration von der Programmierung festgelegten Verhaltens weg, hin zur Definition einer kleineren Menge grundlegender Verhaltenseinheiten, die zu beliebig komplexem Verhalten zusammengesetzt werden können; daher verlangt das Erzeugen von Objekten mit bestimmtem Verhalten mehr als nur das Erzeugen eines Objekts einer einzelnen Klasse.*" [Gamma+96]

Es gibt zwei Arten von Erzeugungsmustern: klassenbasierte und objektbasierte. Klassenbasierte Erzeugungsmuster verlagern Teile der Objekterzeugung in Unterklassen. Dazu gehört nur das FABRIKMETHODE-Muster. Objektbasierte Erzeugungsmuster dagegen delegie-

ren die Objekterzeugung an ein anderes Objekt. Zu dieser Gruppe gehören die Muster ABSTRAKTE FABRIK, ERBAUER, PROTOTYP und SINGLETON.

A.1.1 Abstrakte Fabrik

(ABSTRACT FACTORY) - Seite 107

Zweck: *Biete eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen.*

Die Struktur des ABSTRAKTE-FABRIK-Musters zeigt Abbildung A.1.

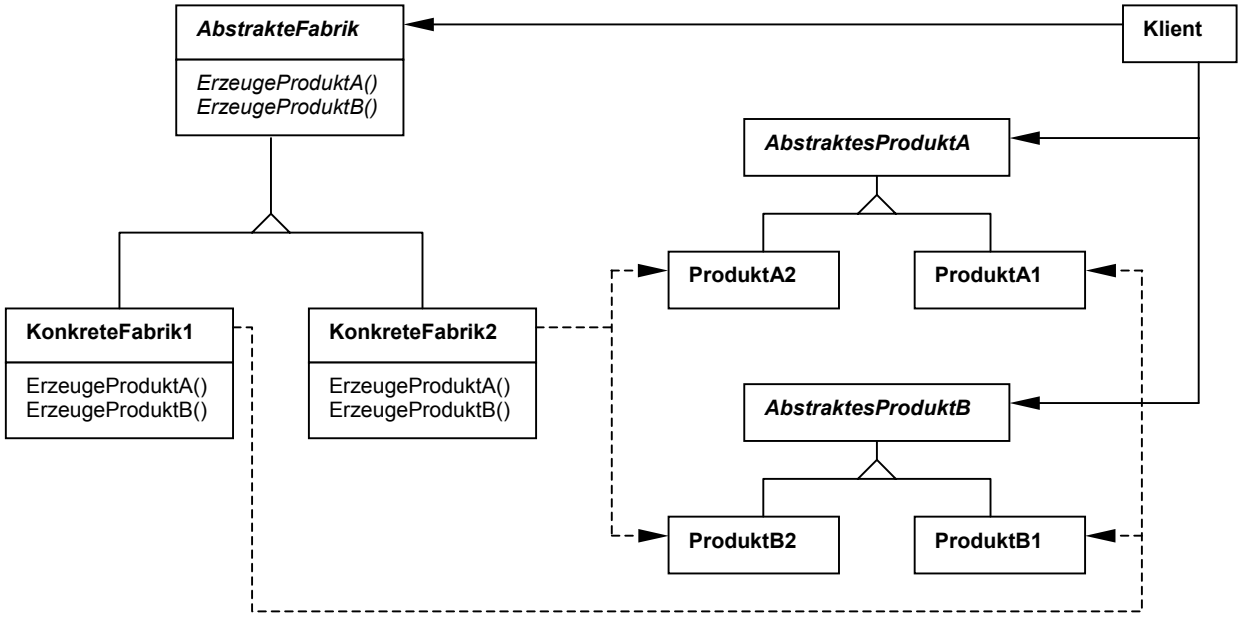


Abbildung A.1: Struktur des Abstrakte-Fabrik-Musters

AbstrakteFabrik deklariert eine Schnittstelle für Operationen, die konkrete Produktobjekte erzeugen (*ErzeugeProduktA* und *ErzeugeProduktB*). Eine *KonkreteFabrik* implementiert diese Operationen zur Erzeugung der konkreten Produktobjekte *ProduktA1*, *ProduktA2*, *ProduktB1* und *ProduktB2*. Die konkreten Produktklassen *ProduktA1*, *ProduktA2*, *ProduktB1* und *ProduktB2* implementieren die Schnittstellen der abstrakten Produktklassen *AbstraktesProduktA* und *AbstraktesProduktB*. Ein *Klient* verwendet nur die Schnittstellen, die von den *AbstrakteFabrik*- und *AbstraktesProdukt*-Klassen deklariert werden; auf *KonkreteFabriken* oder *KonkreteProdukte* greift er nicht zu.

Das ABSTRAKTE-FABRIK-Muster bietet unter anderem den Vorteil, dass eine Familie verwandter Produktobjekte (beispielsweise Fenster, Scrollbars, Buttons für *Microsoft Win-*

dows) auch wirklich ausschließlich zusammen eingesetzt werden und es zu keiner Vermischung mit einer anderen Produktfamilie kommt.

A.1.2 Erbauer

(BUILDER) - Seite 119

Zweck: *Trenne die Konstruktion eines komplexen Objekts von seiner Repräsentation, so dass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann.*

Abbildung A.2 zeigt die Struktur des ERBAUER-Musters.

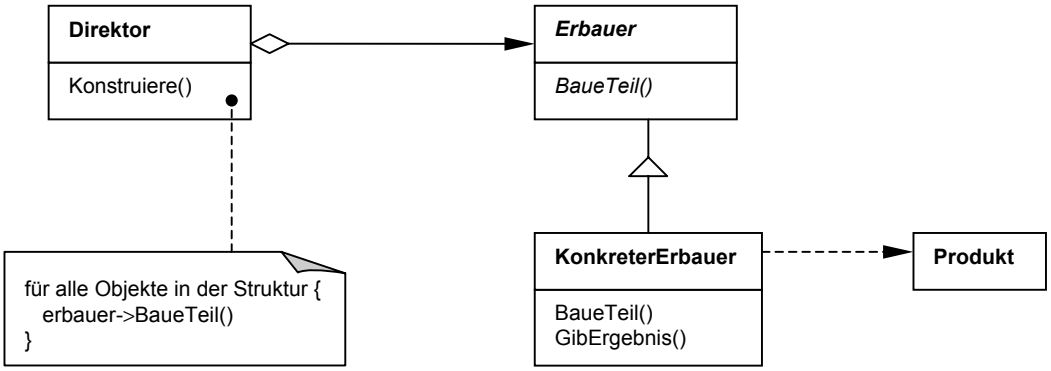


Abbildung A.2: Struktur des Erbauer-Musters

Erbauer spezifiziert eine abstrakte Schnittstelle zum Erzeugen von Teilen eines *Produkt*-Objekts (*BaueTeil*). Der *KonkreteErbauer* implementiert die Schnittstelle des *Erbauers* und bietet außerdem eine Operation zum Zurückgeben des *Produktes* (*GibErgebnis*). Der *Direktor* konstruiert ein Objekt (das *Produkt*) unter Verwendung der Schnittstelle des *Erbauers*.

Ein Vorteil des ERBAUER-Musters ist der, dass ein Algorithmus zum Erzeugen eines komplexen Objekts unabhängig davon bleibt, aus welchen Teilen das Objekt besteht und wie sie zusammengesetzt werden.

A.1.3 Fabrikmethode

(FACTORY METHOD) - Seite 131

Zweck: *Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.*

Die Struktur des FABRIKMETHODE-Musters ist in Abbildung A.3 zu sehen.

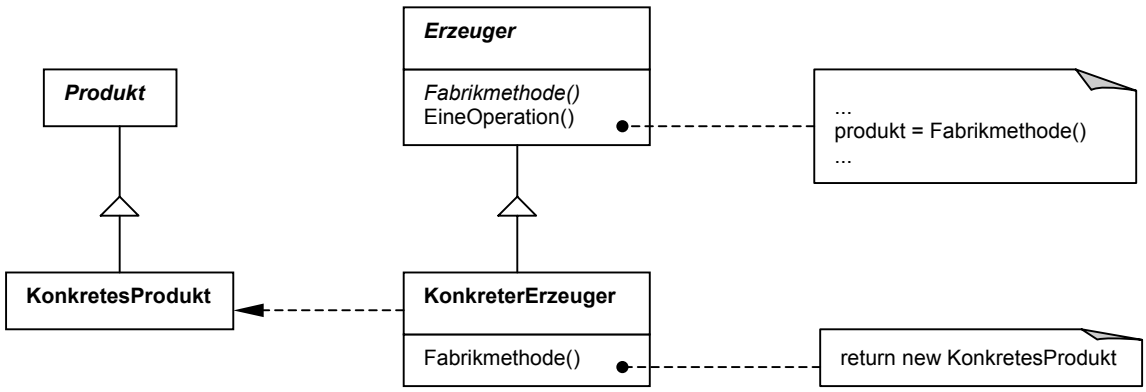


Abbildung A.3: Struktur des Fabrikmethode-Musters

Erzeuger deklariert die abstrakte *Fabrikmethode*, die ein Objekt des Typs *Produkt* zurückliefert. Der *KonkreteErzeuger* überschreibt die Fabrikmethode, so dass in ihr ein *KonkretesProdukt* erzeugt und zurückgegeben wird.

Hilfreich ist das FABRIKMETHODE-Muster zum Beispiel dann, wenn eine Klasse Objekte von anderen Klassen erzeugen soll, sie diese Klassen aber im Voraus nicht kennt – ein Fall, der bei einem Framework auftreten kann.

A.1.4 Prototyp

(PROTOTYPE) - Seite 144

Zweck: *Bestimme die Arten zu erzeugender Objekte durch die Verwendung eines prototypischen Exemplars und erzeuge neue Objekte durch Kopieren dieses Prototypen.*

Die Struktur des PROTOTYP-Musters zeigt Abbildung A.4.

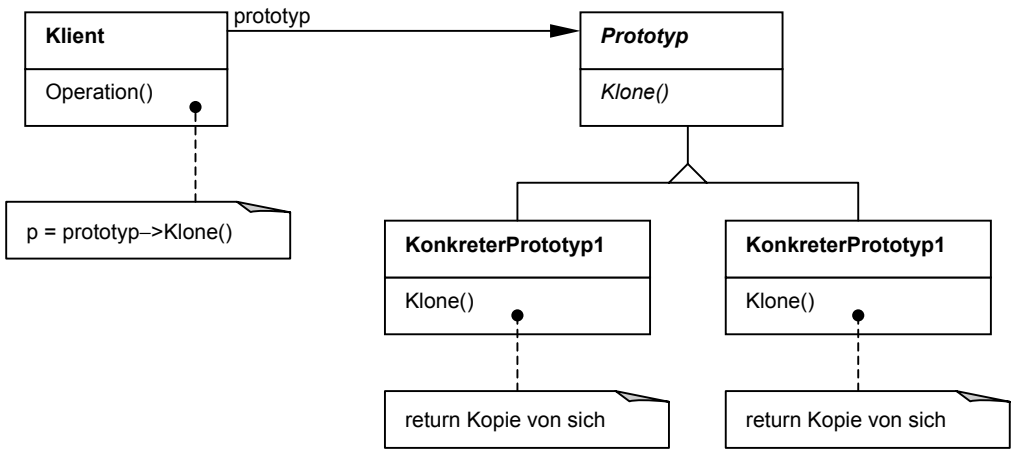


Abbildung A.4: Struktur des Prototyp-Musters

Prototyp ist eine abstrakte Klasse, die eine Schnittstelle deklariert, die es ermöglicht, sich selbst zu klonen (*Klone*-Operation). Die *KonkretenPrototypen* implementieren diese Operation. Der *Klient* erzeugt ein neues Objekt, indem er einem *Prototypen* befiehlt, sich selbst zu klonen.

Mit dem PROTOTYP-Muster ist es möglich, Objekte von Klassen zu erzeugen, die erst zur Laufzeit spezifiziert werden, beispielsweise durch dynamisches Laden.

A.1.5 Singleton

(SINGLETON) - Seite 157

Zweck: *Sichere ab, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit.*

Abbildung A.5 zeigt die Struktur des SINGLETON-Musters.

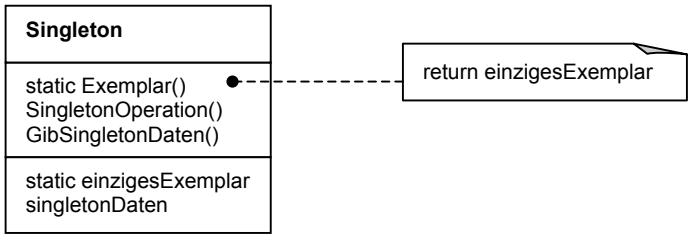


Abbildung A.5: Struktur des Singleton-Musters

Ein *Singleton* definiert eine *Exemplar*-Operation, die es Klienten ermöglicht, auf sein einziges Exemplar zuzugreifen. *Exemplar* ist in C++ eine statische Member-Funktion. Klienten greifen auf ein *Singleton*-Exemplar ausschließlich durch die *Exemplar*-Operation der *Singleton*-Klasse zu.

Das SINGLETON-Muster sichert ab, dass von einer Klasse nur ein einziges Objekt erzeugt wird.

A.2 Strukturmuster

Strukturmuster befassen sich mit der Komposition von Klassen und Objekten, um größere Strukturen zu bilden. Klassenbasierte Strukturmuster benutzen Vererbung, um Schnittstellen oder Implementierungen zusammenzuführen. Objektbasierte Strukturmuster beschreiben Mittel und Wege, Objekte zusammenzuführen, um neue Funktionalität zu gewinnen. Die zusätzliche Flexibilität der Objektkomposition ergibt sich aus der Möglichkeit, das Kompositionsgefüge zur Laufzeit zu ändern, was mit statischer Klassenvererbung nicht möglich ist. Zu den klassenbasierten Strukturmustern gehört nur der KLASSENADAPTER. OBJEKTADAPTER, BRÜCKE, DEKORIERER, FASSADE, FLIEGENGEWICHT, KOMPOSITUM und PROXY sind objektbasierte Strukturmuster.

A.2.1 Adapter

(ADAPTER) - Seite 171

Zweck: *Passe die Schnittstelle einer Klasse an eine andere von ihren Klienten erwartete Schnittstelle an. Das Adaptermuster lässt Klassen zusammenarbeiten, die wegen inkompatibler Schnittstellen ansonsten dazu nicht in der Lage wären.*

Für das ADAPTER-Muster gibt es zwei Varianten: die eine, der KLASSENADAPTER, basiert auf Mehrfachvererbung, und der andere, der OBJEKTADAPTER, basiert auf einfacher Verer-

bung und Delegation. Die folgenden beiden Abbildungen A.6 und A.7 zeigen die Strukturen der beiden ADAPTER-Varianten.

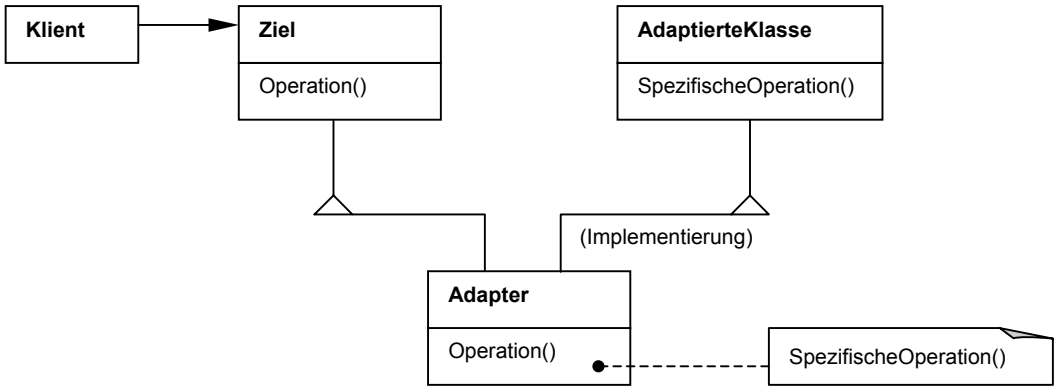


Abbildung A.6: Struktur des Klassenadapters

Der *Klient* arbeitet nur mit Objekten zusammen, die der *Ziel*-Schnittstelle entsprechen. *Ziel* definiert die vom *Klienten* verwendete Schnittstelle. *AdaptierteKlasse* ist eine Klasse, die zwar die vom *Klienten* gewünschte Funktionalität liefert, die der *Klient* aber wegen ihrer unpassenden Schnittstelle nicht verwenden kann. Der *Adapter* passt die Schnittstelle der *AdaptiertenKlasse* an die *Ziel*-Schnittstelle an.

Beim KLASSENADAPTER erbt *Adapter* die Schnittstelle von *Ziel* und die Implementierung von der *AdaptiertenKlasse*.

Beim OBJEKTADAPTER erbt *Adapter* ebenfalls die Schnittstelle von *Ziel*. *Adapter* überschreibt dann diese Operationen und realisiert sie, indem sie die Anfragen an die *AdaptierteKlasse* delegiert.

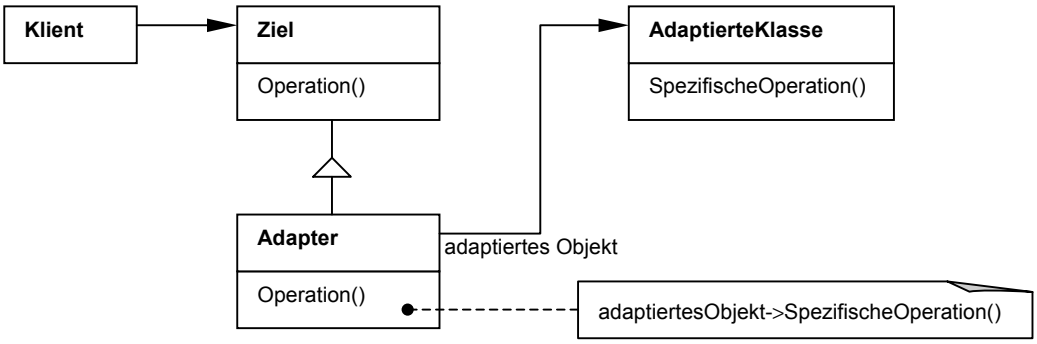


Abbildung A.7: Struktur des Objektadapters

A.2.2 Brücke

(BRIDGE) - Seite 186

Zweck: *Entkopple eine Abstraktion von ihrer Implementierung, so dass beide unabhängig voneinander variiert werden können.*

Abbildung A.8 zeigt die Struktur des BRÜCKE-Musters.

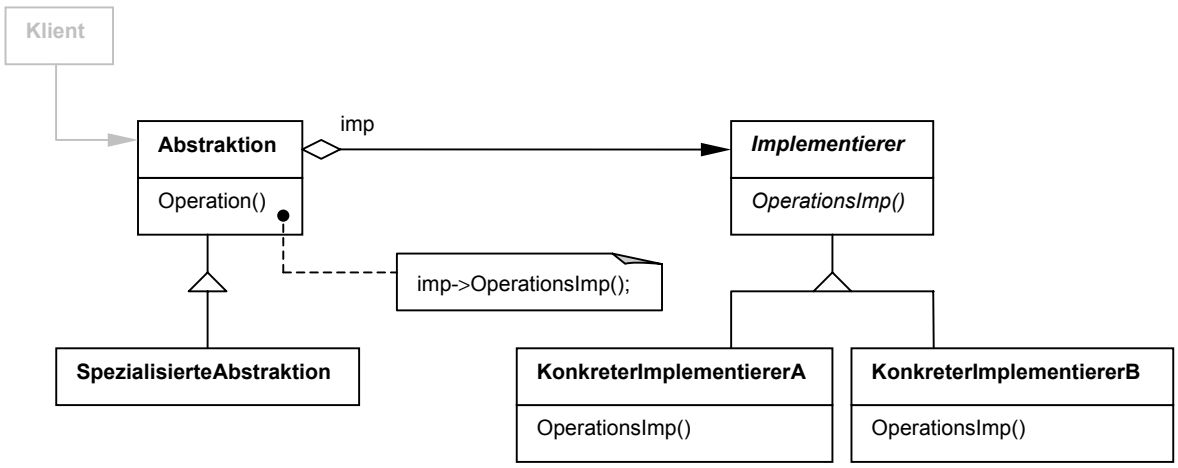


Abbildung A.8: Struktur des Brücke-Musters

Abstraktion definiert die Schnittstelle der Abstraktion und verwaltet eine Referenz auf ein Objekt vom Typ *Implementierer*. Die *SpezialisierteAbstraktion* erweitert die durch *Abs-*

traktion definierte Schnittstelle. *Implementierer* definiert die Schnittstelle für die Implementierungsklassen (*OperationsImp*-Operationen). Diese Schnittstelle bietet üblicherweise nur primitive Operationen, wohingegen *Abstraktion* kompliziertere Operationen definiert, die auf den primitiven Operationen von *Implementierer* basieren. Die *KomkretenImplementierer* implementieren schließlich die *Implementierer*-Schnittstelle und definieren ihre konkreten Implementierungen. Anfragen von *Klienten* werden von der *Abstraktion* an ihr Implementierungsobjekt weitergeleitet.

Das BRÜCKE-Muster kann auch helfen, wenn ein starkes Wachstum einer Klassenhierarchie zu verzeichnen ist. Durch die Aufspaltung der Objekte dieser Hierarchie in ihre Abstraktion und ihre Implementierung lässt sich die Anzahl der Klassen vermindern.

A.2.3 Dekorierer

(DECORATOR) - Seite 199

Zweck: *Erweitere ein Objekt dynamisch um Zuständigkeiten. Dekorierer bieten eine flexible Alternative zur Unterklassenbildung, um die Funktionalität einer Klasse zu erweitern.*

Abbildung A.9 zeigt die Struktur des DEKORIERER-Musters.

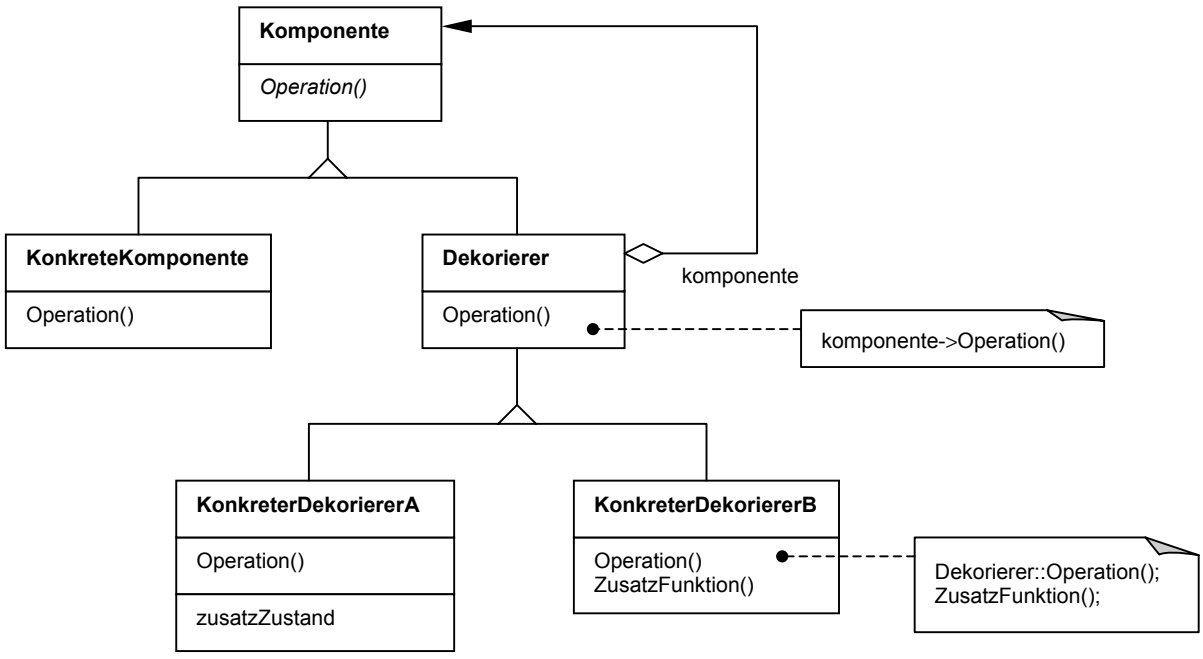


Abbildung A.9: Struktur des Dekorierer-Musters

KonkreteKomponente definiert ein Objekt, das um zusätzliche Funktionalität erweitert werden soll. *Dekorierer* verwaltet eine Referenz auf ein *Komponenten*-Objekt und definiert eine Schnittstelle, die der Schnittstelle von *Komponente* entspricht. *KonkreteDekorierer* sind Unterklassen von *Dekorierer*; sie fügen der *Komponente* Funktionalität hinzu, indem sie die Implementierung von *Dekorierer* verwenden (*Dekorierer::Operation*) und zusätzliche Funktionalität verwenden (*Zusatzfunktion*).

Mit dem DEKORIERER-Muster kann einem Objekt Funktionalität hinzugefügt werden. Es ist möglich, diese Funktionalität auch wieder zu entfernen.

A.2.4 Fassade

(FACADE) - Seite 212

Zweck: *Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Verwendung des Subsystems vereinfacht.*

Die Struktur des FASSADE-Musters ist in Abbildung A.10 zu sehen.

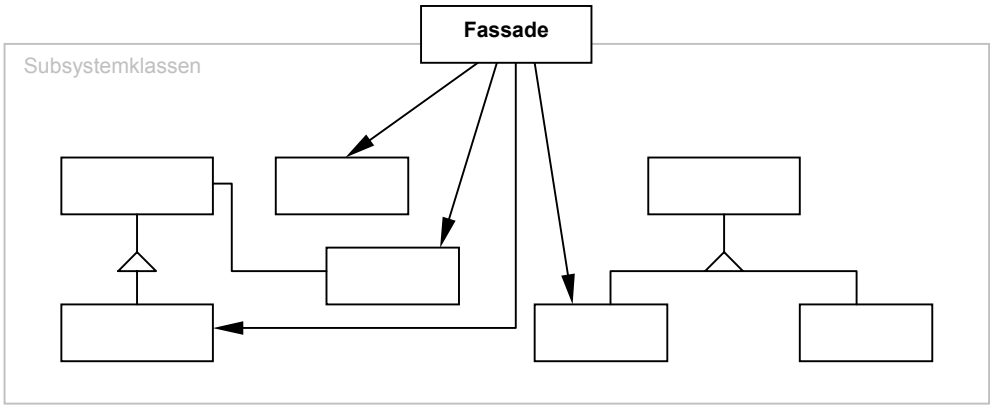


Abbildung A.10: Struktur des Fassade-Musters

Fassade delegiert Anfragen von Klienten an das zuständige Subsystemobjekt. Die *Subsystemklassen* implementieren die Subsystemfunktionalität und führen die von der *Fassade* zugewiesenen Aufgaben aus. Sie wissen nichts von der *Fassade*, das heißt, sie besitzen keine Referenzen auf sie.

Soll eine einfache Schnittstelle zu einem komplexen Subsystem angeboten werden, so ist das FASSADE-Muster dazu gut geeignet. Den meisten Klienten genügt die voreingestellte Sicht, die die Fassade auf das Subsystem bietet. Klienten, die eine größere Funktionalität von dem Subsystem benötigen, greifen direkt auf die Subsystemklassen zu.

A.2.5 Fliegengewicht

(FLYWEIGHT) - Seite 223

Zweck: *Nutze Objekte kleinster Granularität gemeinsam, um große Mengen von ihnen effizient verwenden zu können.*

Abbildung A.11 zeigt die Struktur des FLIEGENGEWICHT-Musters.

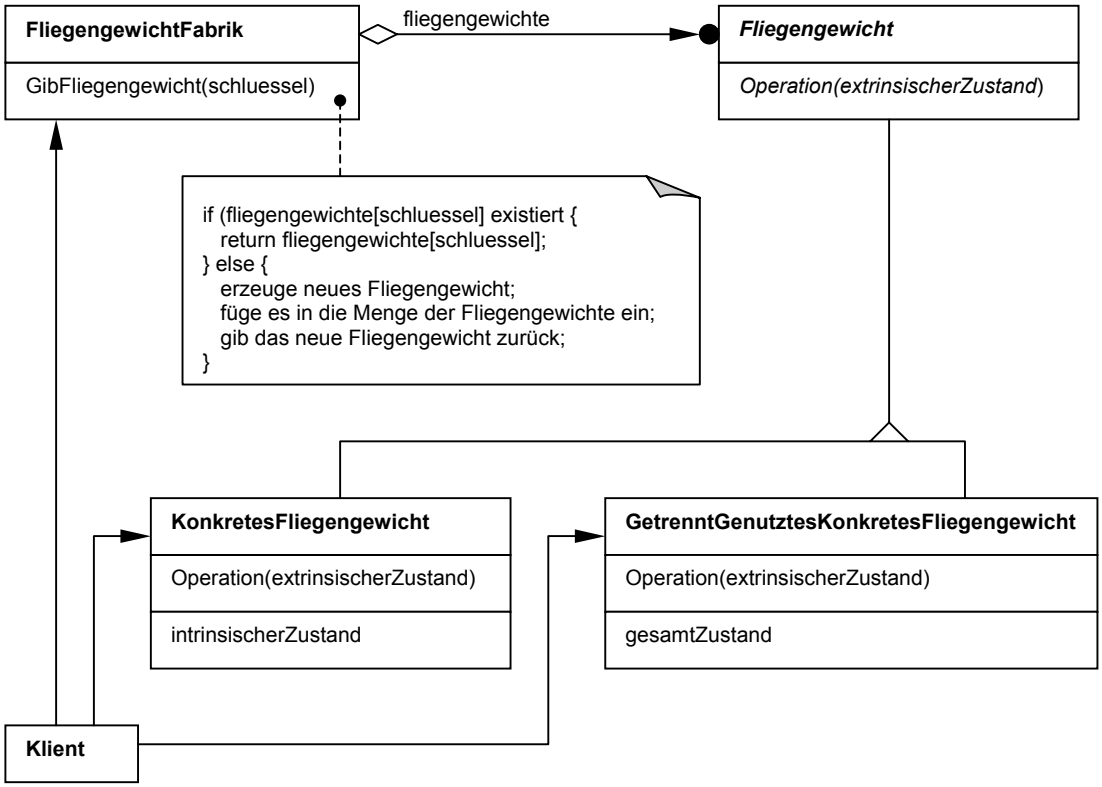


Abbildung A.11: Struktur des Fliegengewicht-Musters

Ein *Fliegengewicht* ist ein Objekt, das gleichzeitig in verschiedenen Kontexten verwendet werden kann. Um dies zu realisieren, besitzen *Fliegengewichte* einen *extrinsischen* und einen *intrinsischen* Zustand. Der *intrinsische* Zustand wird im Fliegengewicht selbst gespeichert; er besteht ausschließlich aus für das Fliegengewicht kontextunabhängigen Informationen. Dadurch ist es möglich, ein *Fliegengewicht* gemeinsam zu nutzen. Der *extrinsische* Zustand eines *Fliegengewichts* hängt vom Kontext ab. *Klienten* sind dafür zuständig, das *Fliegengewicht* mit dem benötigten *extrinsischen* Zustand zu versorgen.

Fliegengewicht deklariert eine Schnittstelle, durch die die *Fliegengewichte* einen *extrinsischen* Zustand erhalten und verarbeiten können. *KonkretesFliegengewicht* implementiert die *Fliegengewicht*-Schnittstelle und erweitert das Objekt um einen *intrinsischen* Zustand, sofern dieser vorhanden ist. *Klient* verwaltet eine Referenz auf *Fliegengewichte* und berechnet oder speichert den *extrinsischen* Zustand der *Fliegengewichte*. *GetrenntGenutztesKonkretesFliegengewicht* ist ein Beispiel dafür, dass nicht alle *Fliegengewicht*-Unterklassen gemeinsam genutzt werden müssen. *FliegengewichtFabrik* erzeugt und verwaltet die *Fliegengewicht*-Objekte.

Möchte eine Anwendung eine große Menge von Objekten verwenden, und sind die Speicherkosten allein aufgrund der Anzahl der Objekte sehr groß, und kann ein Großteil des Objektzustandes in den Kontext verlagert, also extrinsisch modelliert werden, so empfiehlt es sich, das FLIEGENGEWICHT-Muster anzuwenden, um dadurch die Speicherkosten erheblich zu senken.

A.2.6 Kompositum

(COMPOSITE) - Seite 239

Zweck: *Füge Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren. Das Kompositionsmuster ermöglicht es Klienten, einzelne Objekte sowie Kompositionen von Objekten einheitlich zu behandeln.*

Die folgende Abbildung A.12 zeigt die Struktur des KOMPOSITUM-Musters.

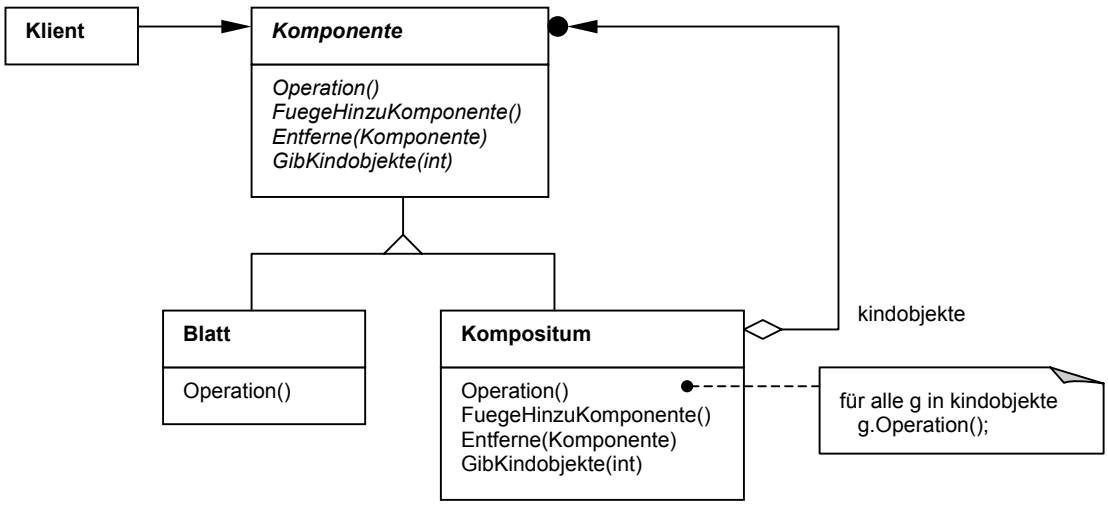


Abbildung A.12: Struktur des Kompositum-Musters

Komponente deklariert die Schnittstelle für Objekte in der zusammengefügten Struktur und implementiert optional ein Standardverhalten für die allen Klassen gemeinsame Schnittstelle. *Komponente* deklariert ferner eine Schnittstelle zum Zugriff auf und zur Verwaltung von Kindobjektkomponenten. *Blatt* repräsentiert Objekte in der Komposition, die keine Kindobjekte besitzen. *Kompositum* definiert ein Verhalten für Komponenten, die Kindobjekte haben können. Es speichert zudem die Kindobjektkomponenten und implementiert die kindobjekt-bezogenen Operationen der Schnittstelle von *Komponente*. Der *Klient* manipuliert die Objekte in der Komposition durch die Schnittstelle von *Komponente*.

A.2.7 Proxy

(PROXY) - Seite 254

Zweck: *Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts.*

In Abbildung 1.13 ist die Klassenstruktur des PROXY-Musters zu sehen.

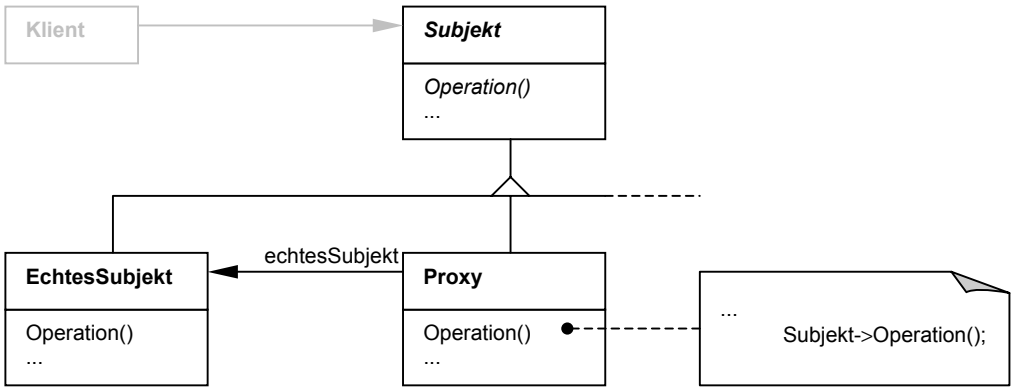


Abbildung A.13: Struktur des Proxy-Musters

Subjekt definiert die gemeinsame Schnittstelle von *EchtesSubjekt* und *Proxy*, so dass ein *Proxy* überall dort benutzt werden kann, wo ein *EchtesSubjekt* erwartet wird. *EchtesSubjekt* definiert das eigentliche Objekt, das durch den *Proxy* repräsentiert wird. *Proxy* bietet eine Schnittstelle, die mit der von *Subjekt* identisch ist, so dass ein *Proxy* für ein *EchtesSubjekt* eingesetzt werden kann. *Proxy* verwaltet außerdem eine Referenz auf *EchtesSubjekt*, die es ermöglicht, auf *EchtesSubjekt* zuzugreifen.

Mit Hilfe des PROXY-Musters brauchen Objekte, für deren Erzeugung und Initialisierung hohe Kosten aufgebracht werden müssen, nicht sofort sondern erst auf Verlangen erzeugt werden.

A.3 Verhaltensmuster

Verhaltensmuster befassen sich mit Algorithmen und der Zuweisung von Zuständigkeiten zu Objekten. Sie beschreiben nicht nur Muster von Objekten oder Klassen, sondern auch die Muster der Interaktion zwischen ihnen[21]; ihr Fokus liegt auf interagierenden Objekten.

Klassenbasierte Verhaltensmuster verwenden Vererbung, um das Verhalten unter den Klassen zu verteilen. Objektbasierte Verhaltensmuster verwenden Objektkomposition anstelle von Vererbung; manche der Muster beschreiben, wie ein Gruppe von Objekten zusammenarbeitet, um eine Aufgabe zu erledigen, die keines der Objekte allein ausführen kann. Besonders wichtig ist hierbei, in welcher Weise zusammenarbeitende Objekte einander kennen: explizite Referenzen aufeinander bedeutet enge Kopplung – im Extremfall kennt jedes Objekt jedes andere. Verhaltensmuster fördern jedoch lose Kopplung. Andere objektbasierte Verhaltensmuster befassen sich mit der Kapselung von Verhalten in einem eigenständigen Objekt und dem Weiterleiten der Operationsaufrufe an dieses Objekt.

Zu den klassenbasierten Verhaltensmustern gehören die Muster INTERPRETER und SCHABLONENMETHODE. BEFEHL, BEOBACHTER, BESUCHER, ITERATOR, MEMENTO, STRATEGIE, VERMITTLER, ZUSTAND und ZUSTÄNDIGKEITSKETTE sind objektbasierte Verhaltensmuster.

A.3.1 Befehl

(COMMAND) - Seite 273

Zweck: *Kapsele einen Befehl als ein Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine Schlange zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen.*

Die folgende Abbildung A.14 zeigt die Struktur des BEFEHLS-Musters.

[21] In [Gamma+96] wird dies bei einigen Mustern durch einfache Sequenzdiagramme veranschaulicht. Aus Platzgründen wird hier jedoch darauf verzichtet.

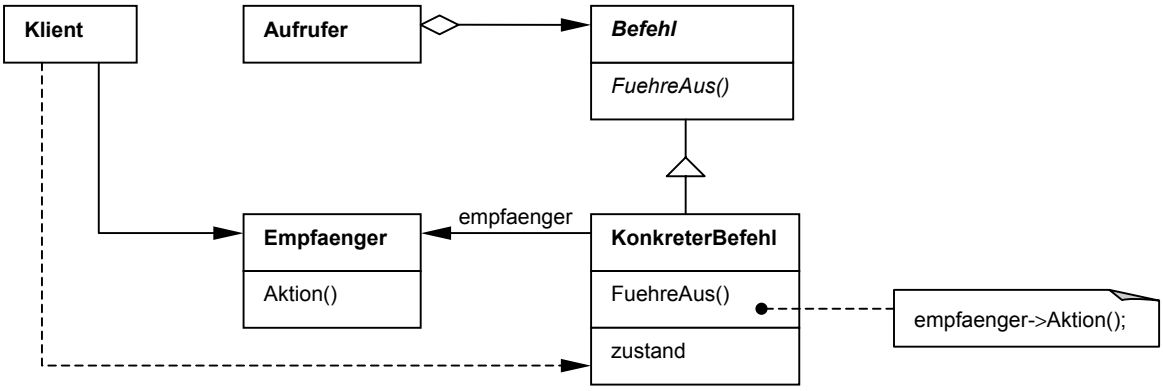


Abbildung A.14: Struktur des Befehls-Musters

Befehl deklariert eine Schnittstelle zum Ausführen einer Operation. *KonkreterBefehl* definiert die Anbindung eines *Empfängers* an eine Aktion und implementiert die *FuehreAus*-Operation durch Aufrufen der entsprechenden Operation(en) beim *Empfänger*. *Klient* erzeugt ein *KonkreterBefehl*-Objekt und übergibt ihm den *Empfänger*. *Aufrufer* befiehlt dem *Befehl*-Objekt, die Anfrage auszuführen. *Empfänger* weiß, wie die an die Ausführung einer Anfrage gebundenen Operationen auszuführen sind. Jede Klasse kann ein *Empfänger* sein.

Das BEFEHLS-Muster ermöglicht es beispielsweise Steuerungselementen einer Klassenbibliothek, Anfragen an unbekannte Anwendungsobjekte zu richten, indem es die Anfrage selbst zu einem Objekt macht.

A.3.2 Beobachter

(OBSERVER) - Seite 287

Zweck: *Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.*

Abbildung A.15 zeigt die Struktur des BEOBACHTER-Musters.

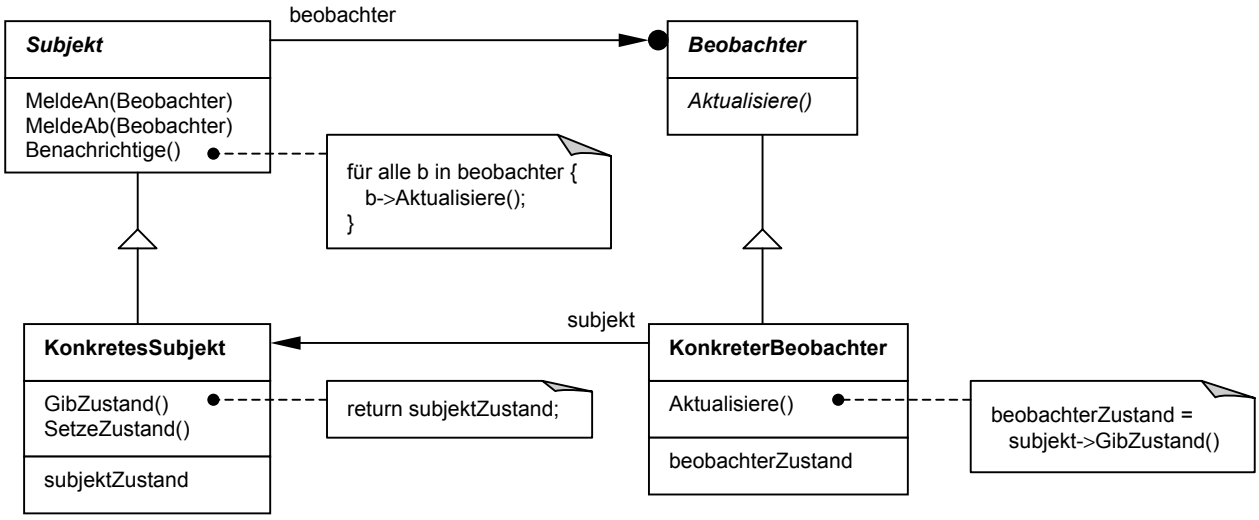


Abbildung A.15: Struktur des Beobachter-Musters

Subjekt kennt seine *Beobachter*. Ein *Subjekt* kann von einer beliebigen Anzahl von *Beobachtern* beobachtet werden. *Subjekt* bietet zudem eine Schnittstelle zum An- und Abmelden von *Beobachtern*. *Beobachter* definiert eine Aktualisierungsschnittstelle für Objekte, die über Änderungen eines *Subjekts* benachrichtigt werden sollen. *KonkretesSubjekt* speichert den für *KonkreteBeobachter* relevanten Zustand, und es benachrichtigt seine *Beobachter*, wenn sich dieser Zustand ändert. *KonkreterBeobachter* verwaltet eine Referenz auf ein *KonkretesSubjekt* und speichert den Zustand, der mit dem des *Subjekts* in Einklang stehen soll.

A.3.3 Besucher

(VISITOR) - Seite 301

Zweck: *Kapsele eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt. Das Besuchermuster ermöglicht es, eine neue Operation zu definieren, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.*

Die folgende Abbildung A.16 stellt die Struktur des BESUCHER-Musters dar.

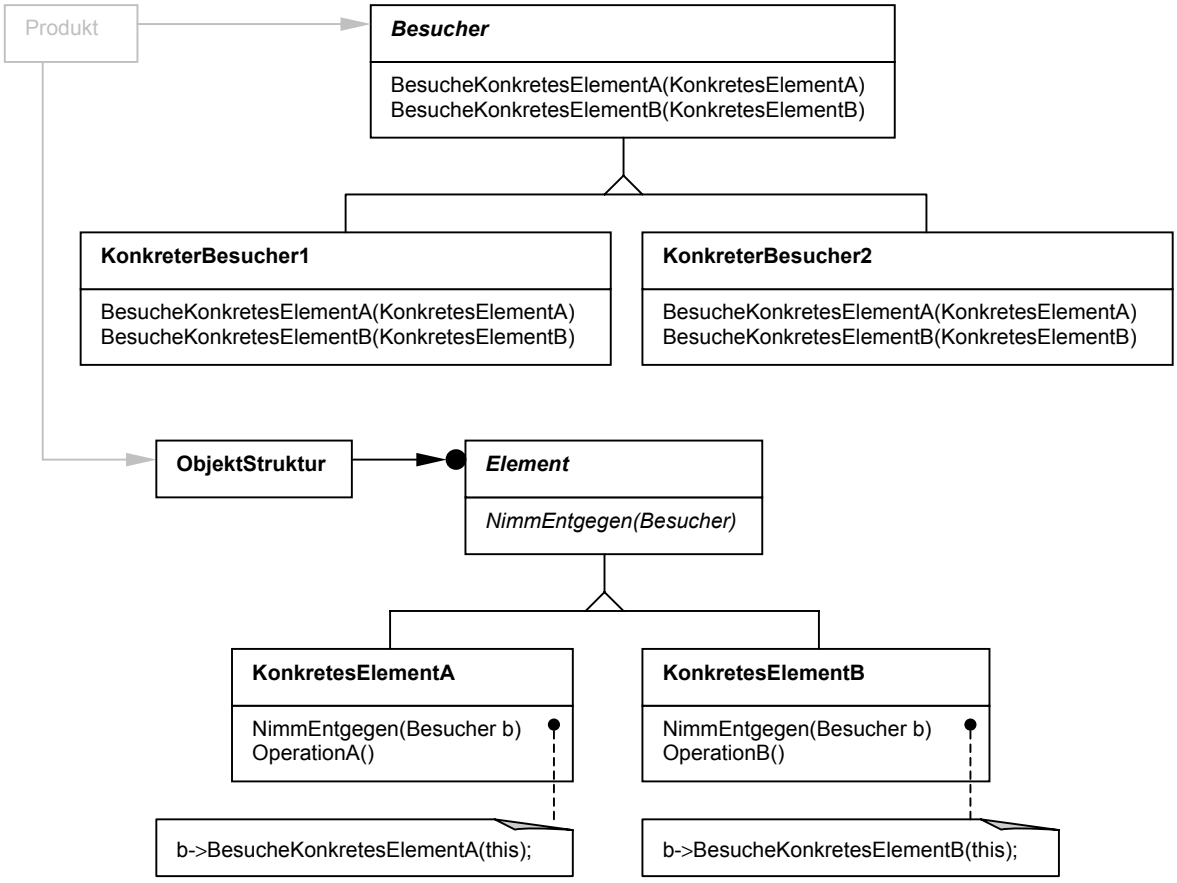


Abbildung A.16: Struktur des Besucher-Musters

Besucher deklariert eine *Besuche*-Operation für jede *KonkretesElement*-Klasse in der Objektstruktur. Der Name der Operation und ihre Signatur benennen die Klasse, welche die *Besuche*-Operation des *Besuchers* aufruft. Dies ermöglicht es dem *Besucher*, die konkrete Klasse des besuchten *Elements* zu bestimmen. Der *Besucher* kann dann unter Verwendung der konkreten Schnittstelle auf das *Element* zugreifen. *KonkreterBesucher* implementiert jede von der *Besucherklasse* deklarierte Operation. *Element* definiert eine *NimmEntgegen*-Operation, die einen *Besucher* als Argument empfangen kann. *KonkreteElemente* implementieren die *NimmEntgegen*-Operation.

Liegen Klassen, die eine Objektstruktur definieren, vor, und ändern sich diese Klassen praktisch nie, und möchte man aber häufig neue Operationen für die Struktur definieren, so ist es mit Hilfe des BESUCHER-Musters möglich, diese neuen Operationen hinzuzufügen, ohne Änderungen an den betroffenen Klassen vornehmen zu müssen.

A.3.4 Interpreter

(INTERPRETER) - Seite 319

Zweck: *Definiere für eine gegebene Sprache eine Repräsentation der Grammatik sowie einen Interpreter, der die Repräsentation nutzt, um Sätze in der Sprache zu interpretieren.*

Die folgende Abbildung A.17 zeigt die Struktur des INTERPRETER-Musters.

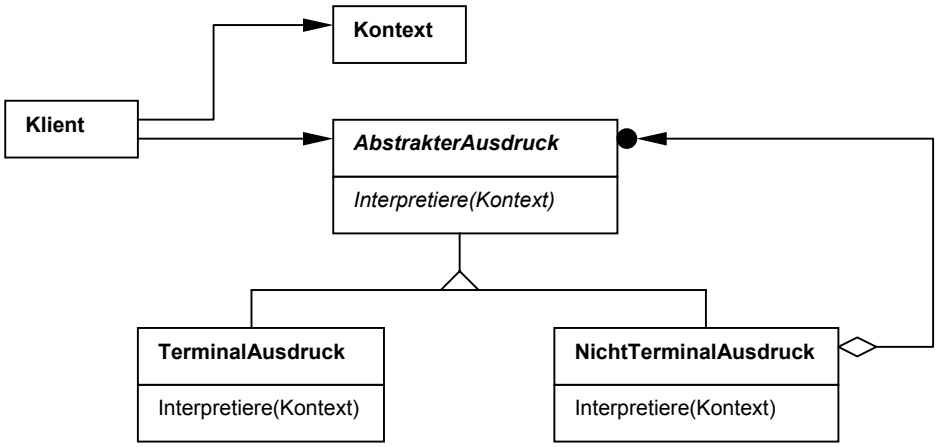


Abbildung A.17: Struktur des Interpreter-Musters

AbstrakterAusdruck deklariert eine abstrakte *Interpretiere*-Operation, die allen Knoten im abstrakten Syntaxbaum gemein ist. *TerminalAusdruck* implementiert die *Interpretiere*-Operation, die mit den Terminalsymbolen in der Grammatik verbunden ist. Für jedes Terminalsymbol in einem Satz wird ein Exemplar dieser Klasse benötigt. *NichtTerminalAusdruck* implementiert die *Interpretiere*-Operation für Nichtterminalsymbole der Grammatik. *Kontext* enthält die für den Interpreter globalen Informationen. *Klient* konstruiert einen abstrakten Syntaxbaum, der einen bestimmten Satz in der von der Grammatik definierten Sprache repräsentiert. Der abstrakte Syntaxbaum wird aus den Exemplaren der *NichtTerminalAusdruck*- und *TerminalAusdruck*-Klassen zusammengesetzt. *Klient* ruft außerdem die *Interpretiere*-Operation auf.

Das INTERPRETER-Muster kann man sehr gut anwenden, wenn eine Sprache mit einer einfachen Grammatik interpretiert werden soll und die Ausdrücke der Sprache als abstrakte Syntaxbäume darstellbar sind.

A.3.5 Iterator

(ITERATOR) – Seite 335

Zweck: *Ermögliche den sequentiellen Zugriff auf die Elemente eines zusammengesetzten Objekts, ohne seine zugrundeliegende Repräsentation offenzulegen.*

Die Struktur des ITERATOR-Musters zeigt Abbildung A.18.

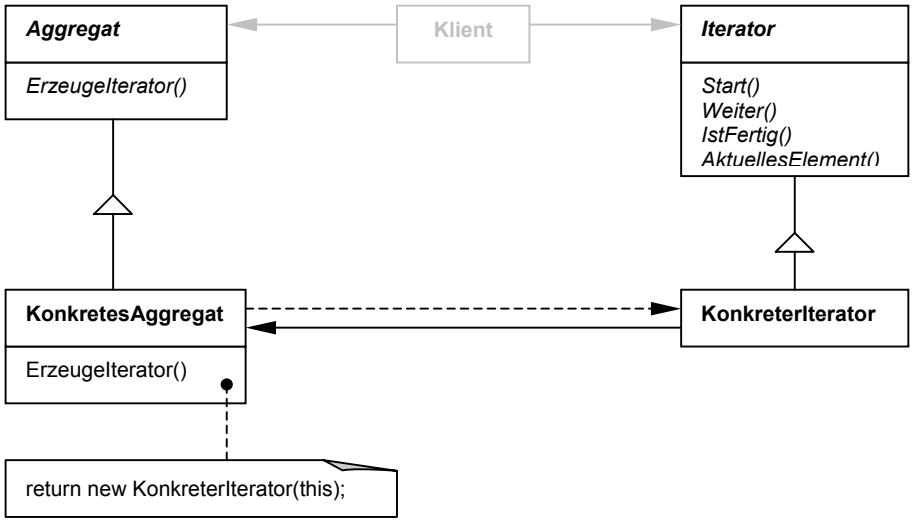


Abbildung A.18: Struktur des Iterator-Musters

Iterator definiert eine Schnittstelle zur Traversierung von Elementen. *KonkreterIterator* implementiert die Schnittstelle von *Iterator* und verwaltet die aktuelle Position während der Traversierung des *KonkretenAggregats*. *Aggregat* definiert eine Schnittstelle zum Erzeugen eines Objekts der Klasse *Iterator*. *KonkretesAggregat* implementiert die Operation zum Erzeugen eines *KonkretenIterators*, indem es ein Objekt der passenden *KonkreterIterator*-Klasse zurückgibt.

Das ITERATOR-Muster ermöglicht es auch, mehrfache gleichzeitige Traversierungen auf zusammengesetzten Objekten vorzunehmen.

A.3.6 Memento

(MEMENTO) - Seite 354

Zweck: *Erfasse und externalisiere den internen Zustand eines Objekts, ohne seine Kapselung zu verletzen, so dass das Objekt später in diesen Zustand zurückversetzt werden kann.*

Die Struktur des MEMENTO-Musters ist in Abbildung A.19 dargestellt.

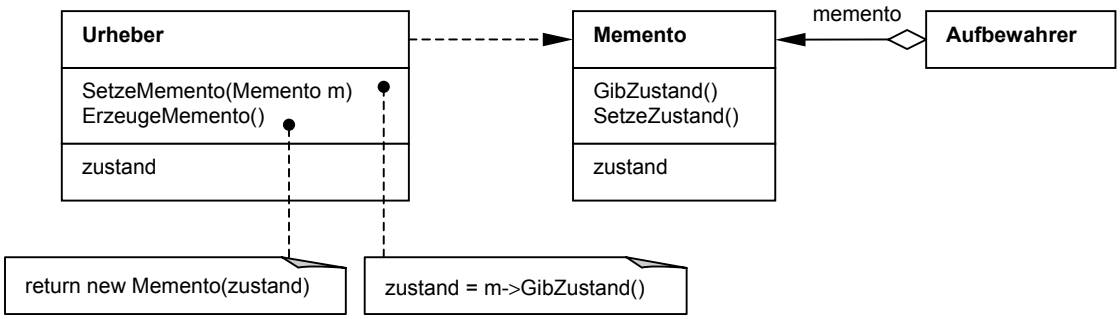


Abbildung A.19: Struktur des Memento-Musters

Memento speichert den internen Zustand des *Urheber*-Objekts. Das *Memento* kann soviel oder so wenig vom internen Zustand des *Urhebers* zwischenspeichern, wie nötig ist. *Urheber* erzeugt ein *Memento*, das eine Momentaufnahme seines aktuellen internen Zustands enthält. *Aufbewahrer* ist für die Aufbewahrung des *Mementos* zuständig; er arbeitet niemals mit dem Inhalt des *Mementos* oder untersucht es, aber er kann veranlassen, dass ein durch das Memento gespeicherter früherer Zustand des *Urhebers* wieder hergestellt wird.

Mit Hilfe des MEMENTO-Musters ist es möglich, einen Undo-Mechanismus zu realisieren.

A.3.7 Schablonenmethode

(TEMPLATE METHOD) - Seite 366

Zweck: *Definiere das Skelett eines Algorithmus in einer Operation und delegiere einzelne Schritte an Unterklassen. Die Verwendung einer Schablonenmethode ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne seine Struktur zu verändern.*

Die Struktur des SCHABLONENMETHODE-MUSTERS stellt Abbildung A.20 dar.

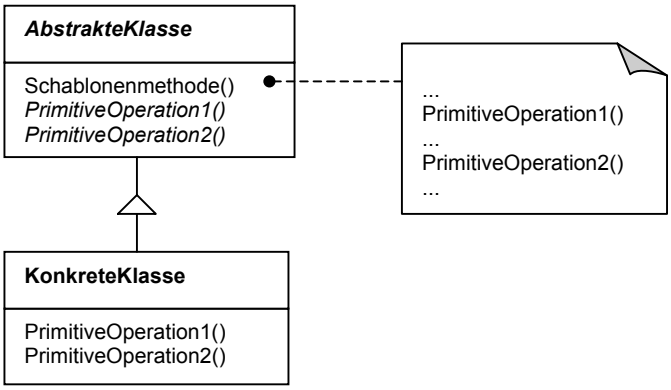


Abbildung A.20: Struktur des Schablonenmethode-Musters

AbstrakteKlasse deklariert abstrakte primitive Operationen *PrimitiveOperation1* und *PrimitiveOperation2*, die von konkreten Unterklassen definiert werden, um die Schritte eines Algorithmus zu implementieren. *AbstrakteKlasse* implementiert außerdem eine *Schablonenmethode*, die das Skelett eines Algorithmus definiert. Die *Schablonenmethode* ruft sowohl *PrimitiveOperationen* der *AbstraktenKlasse* also auch Operationen anderer Klassen auf. Die *KonkreteKlasse* implementiert die *PrimitivenOperationen*, die die unterklassenspezifischen Schritte des Algorithmus ausführen.

Das SCHABLONENMETHODE-Muster gestattet es, die sich verändernden Teile eines Algorithmus einmal festzulegen und es dann Unterklassen zu überlassen, das variierende Verhalten zu implementieren.

A.3.8 Strategie

(STRATEGY) - Seite 373

Zweck: *Definiere eine Familie von Algorithmen, kapsele jeden einzelnen und mache sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu variieren.*

Die Struktur des STRATEGIE-Musters ist in Abbildung A.21 dargestellt.

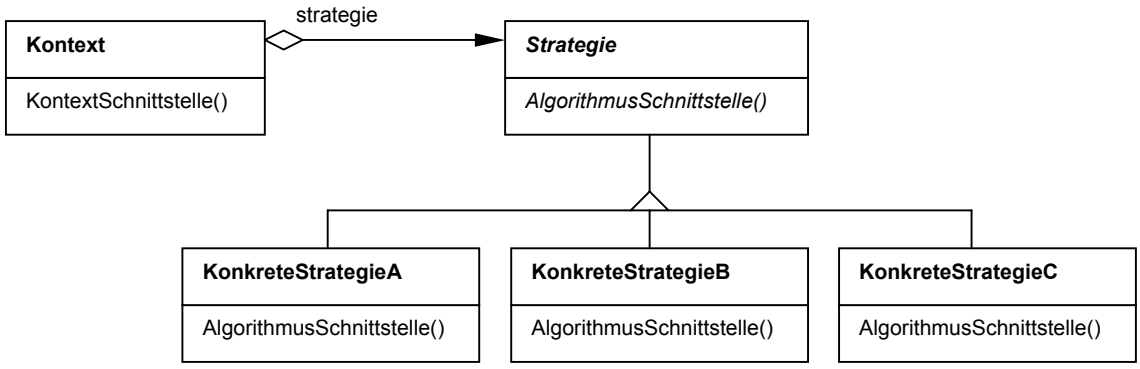


Abbildung A.21: Struktur des Strategie-Musters

Strategie deklariert eine Schnittstelle, die von allen unterstützten Algorithmen angeboten wird. Das *Kontext*-Objekt verwendet diese Schnittstelle, um den durch eine *KonkreteStrategie* definierten Algorithmus aufzurufen. Jede *KonkreteStrategie*-Klasse implementiert einen Algorithmus unter Verwendung der *Strategie*-Schnittstelle; so ist es möglich, sie gegeneinander austauschbar zu halten.

A.3.9 Vermittler

(MEDIATOR) - Seite 385

Zweck: *Definiere ein Objekt, welches das Zusammenspiel einer Menge von Objekten in sich kapselt. Vermittler fördern lose Kopplung, indem sie Objekte davon abhalten, aufeinander explizit Bezug zu nehmen. Sie ermöglichen es, das Zusammenspiel der Objekte von ihnen unabhängig zu variieren.*

Abbildung A.22 zeigt die Struktur des VERMITTLER-Musters.

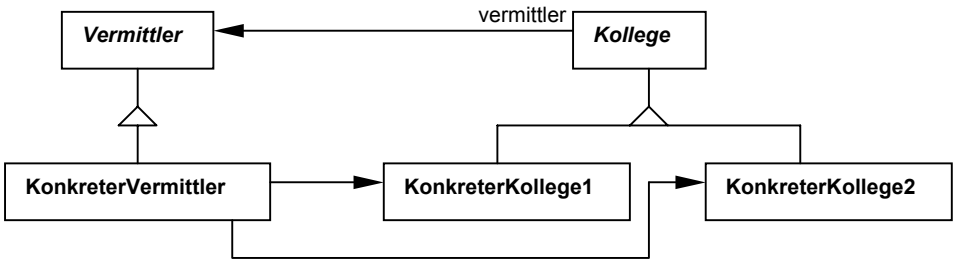


Abbildung A.22: Struktur des Vermittler-Musters

Vermittler definiert eine Schnittstelle für die Interaktion mit *Kollegen*-Objekten. *KonkreterVermittler* implementiert das Gesamtverhalten durch Koordination der *Kollegen*-Objekte. Er verwaltet Referenzen auf seine *Kollegen*-Objekte. Jede *Kollegen*-Klasse kennt ihre *Vermittler*-Klasse. Jedes *Kollegen*-Objekt arbeitet mit seinem *Vermittler* zusammen, statt dies mit seinen *Kollegen*-Objekten zu tun.

A.3.10 Zustand

(STATE) - Seite 398

Zweck: *Ermögliche es einem Objekt, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Es wird so aussehen, als ob das Objekt seine Klasse gewechselt hat.*

Die folgende Abbildung A.23 stellt die Struktur des ZUSTANDS-Musters dar.

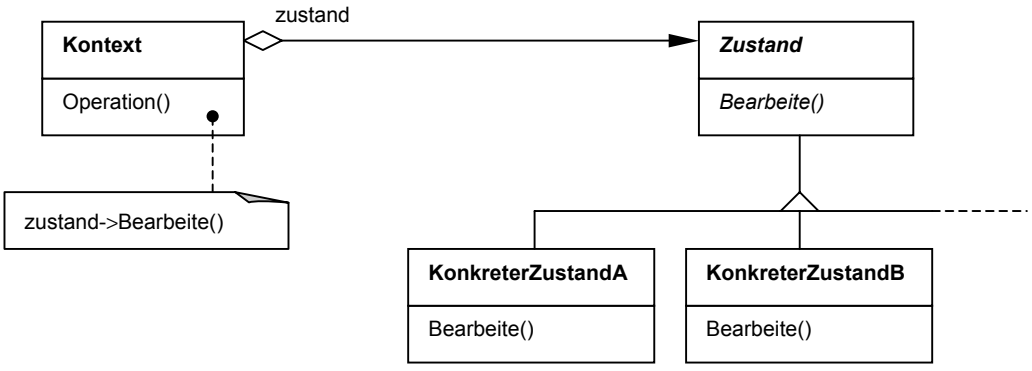


Abbildung A.23: Struktur des Zustands-Musters

Kontext definiert die Klienten interessierende Schnittstelle. *Kontext* verwaltet außerdem ein Exemplar einer *KonkretenZustands*-Klasse, die den aktuellen Zustand definiert. *Zustand* deklariert eine Schnittstelle, die allen *KonkretenZustands*-Klassen gemein ist. Jede *KonkreteZustands*-Klasse implementiert ein Verhalten, das mit einem Zustand des *Kontext*-Objekts verbunden ist.

Hängt das Verhalten eines Objekts von seinem Zustand ab, und muss es sein Verhalten in Abhängigkeit von diesem Zustand und zur Laufzeit ändern können, so empfiehlt sich die Verwendung des ZUSTANDS-Musters.

A.3.11 Zuständigkeitskette

(CHAIN OF RESPONSIBILITY) - Seite 410

Zweck: *Vermeide die Kopplung des Auslösers einer Anfrage an seinen Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Anfrage zu erledigen. Verkette die empfangenden Objekte, und leite die Anfrage an der Kette entlang, bis ein Objekt sie erledigt.*

Die Struktur des ZUSTÄNDIGKEITSKETTE-Musters zeigt die folgende Abbildung A.24.

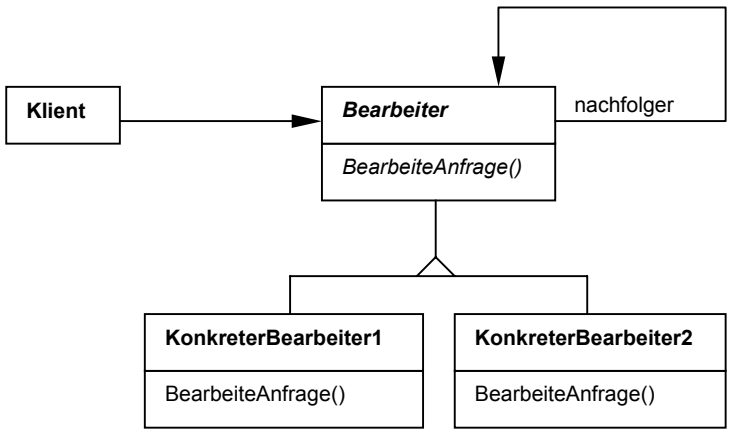
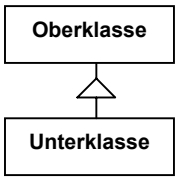
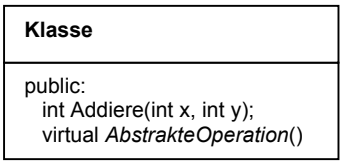
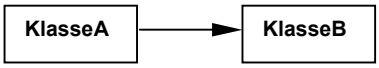
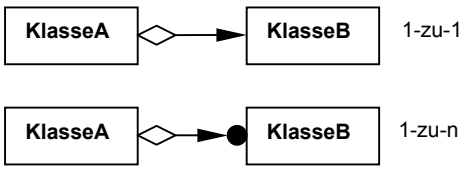


Abbildung A.24: Struktur des Zuständigkeitskette-Musters

Bearbeiter definiert eine Schnittstelle zur Bearbeitung von Anfragen und implementiert optional eine Verbindung zu seinem Nachfolgerobjekt. *KonkreterBearbeiter* arbeitet genau die Anfrage ab, für die er zuständig ist. Er kann auf sein Nachfolgerobjekt zugreifen. Wenn der *KonkreteBearbeiter* eine Anfrage erhält und diese bearbeiten kann, so tut er dies auch; andernfalls leitet er sie an sein Nachfolgerobjekt weiter. Der *Klient* löst die Anfrage bei einem *KonkreterBearbeiter*-Objekt in der Kette aus.

B Notation der Klassendiagramme

Dieser Anhang (Tabelle B.1) gibt einen Überblick über die für die Klassendiagramme verwendete Notation. Diese Notation entspricht der *Object Modeling Technique* (OMT) von James Rumbaugh.

Merkmal	**Darstellung**
abstrakte Klasse	
konkrete Klasse	
Vererbung	
Attribute (Sichtbarkeit, Typ, Name)	
Operationen (Sichtbarkeit, Polymorphie, Rückgabetyp, Name, Parameter, Abstraktheit)	
Assoziationsbeziehung	
Aggregationsbeziehung	
Objekterzeugung	

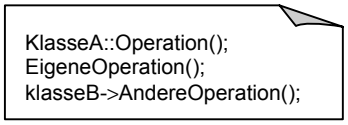
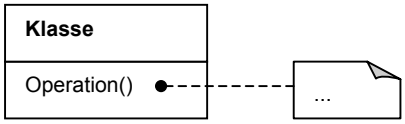
Merkmal	Darstellung
Methodenaufrufe	
Methodenimplementierung	

Tabelle B.1: Elemente der Object Modeling Technique (OMT)

C Quelltexte der Rational Rose Scripte

In diesem Anhang sind die vollständigen *Rational Rose Scripte* für die Suche nach den Mustern SINGLETON, INTERPRETER und KOMPOSITUM aufgelistet.

Listing C.1: FindeSingleton.ebs

```
' Liefert für eine Klasse alle Oberklassen (nicht nur die direkten) und die Klasse selbst
Function GetWithAllSuperclasses (theClass As Class) As ClassCollection
	Dim marked As New ClassCollection
	Dim notmarked As New ClassCollection
	Dim inArbeit As Class

	notmarked.Add theClass
	While notmarked.Count>0
		Set inArbeit=notmarked.GetAt(1)
		notmarked.Remove inArbeit
		marked.Add inArbeit
		notmarked.AddCollection inArbeit.GetSuperclasses()
	Wend
	Set GetWithAllSuperclasses=marked
End Function

' Prüft, ob eine Klasse in einer Collection vorhanden ist, aber anhand des Namens
Function ExistsClassByName(theClasses As ClassCollection, theClass As String) As Boolean
	Dim KlasseInArbeit As Class
	Dim existiert As Boolean

	existiert=false

	For i=1 To theClasses.Count
		Set KlasseInArbeit=theClasses.GetAt(i)
		If (theClass=KlasseInArbeit.Name) Or (theClass=KlasseInArbeit.Name+"*") Then
			existiert=true
		End If
	Next i
	ExistsClassByName=existiert
End Function

' Liefert alle Konstruktoren einer Klasse
Function GetAllConstructors (theClass As Class) As OperationCollection
	Dim alleMethoden As OperationCollection
	Dim alleKonstruktoren As New OperationCollection
	Dim MethodeInArbeit As Operation

	Set alleMethoden=theClass.Operations
	For i=1 To alleMethoden.Count
		Set MethodeInArbeit=alleMethoden.GetAt(i)
		If MethodeInArbeit.Name=theClass.Name Then
			alleKonstruktoren.Add MethodeInArbeit
		End If
	Next i
	Set GetAllConstructors=alleKonstruktoren
End Function

' Liefert alle normalen Methoden einer Klasse (keine Konstruktoren)
Function GetAllNormalOperations (theClass As Class) As OperationCollection
	Dim alleMethoden As OperationCollection
	Dim alleNormalenMethoden As New OperationCollection
	Dim MethodeInArbeit As Operation

	Set alleMethoden=theClass.Operations
	For i=1 To alleMethoden.Count
		Set MethodeInArbeit=alleMethoden.GetAt(i)
		If MethodeInArbeit.Name<>theClass.Name Then
			alleNormalenMethoden.Add MethodeInArbeit
		End If
```

```
        Next i
        Set GetAllNormalOperations=alleNormalenMethoden
End Function

' Sucht nach dem Singleton-Muster, das erste gefundene wird zurückgegeben
Function FindeSingleton(alleKlassen As ClassCollection) As Class
        Dim Singleton As Class
        Dim KlasseInArbeit As Class
        Dim alleReferenzen As AssociationCollection
        Dim ReferenzInArbeit As Association
        Dim andereRolle As Role
        Dim alleMethoden As OperationCollection
        Dim MethodeInArbeit As Operation
        Dim returnType As String
        Dim flag As Boolean
        Dim ok As Boolean

        For i=1 To alleKlassen.Count
                ok=true

                ' Prüfe auf statische Referenz auf eigene Klasse oder Oberklasse
                flag=false
                Set KlasseInArbeit=alleKlassen.GetAt(i)
                Set alleReferenzen=KlasseInArbeit.GetAssociations()
                For j=1 To alleReferenzen.Count
                        Set ReferenzInArbeit=alleReferenzen.GetAt(j)
                        Set andereRolle=ReferenzInArbeit.GetOtherRole(KlasseInArbeit)
                        If andereRolle.Static Then
                                If GetWithAllSuperclasses(KlasseInArbeit)._
                                Exists(andereRolle.Class) Then
                                        flag=true
                                End If
                        End If
                Next j
                If flag=false Then ok=false

                ' Prüfe auf Methode, die eigene Klasse oder Oberklasse zurückgibt
                flag=false
                Set alleMethoden=GetAllNormalOperations(KlasseInArbeit)
                For j=1 To alleMethoden.Count
                        Set MethodeInArbeit=alleMethoden.GetAt(j)
                        If ExistsClassByName(GetWithAllSuperclasses(KlasseInArbeit),_
                           MethodeInArbeit.ReturnType) Then
                                        flag=true
                        End If
                Next j
                If flag=false Then ok=false

                ' Prüfe, ob kein Public-Konstruktor vorhanden
                flag=true
                Set alleMethoden=GetAllConstructors(KlasseInArbeit)
                For j=1 To alleMethoden.Count
                        Set MethodeInArbeit=alleMethoden.GetAt(j)
                        If MethodeInArbeit.ExportControl.Name="PublicAccess" Then
                                flag=false
                        End If
                Next j
                If flag=false Then ok=false

                ' Prüfe, ob Private- oder Protected-Konstruktor vorhanden
                flag=false
                Set alleMethoden=GetAllConstructors(KlasseInArbeit)
                For j=1 To alleMethoden.Count
                        Set MethodeInArbeit=alleMethoden.GetAt(j)
                        If (MethodeInArbeit.ExportControl.Name="ProtectedAccess") Or _
                           (MethodeInArbeit.ExportControl.Name="PrivateAccess") Then
                                flag=true
                        End If
                Next j
                If flag=false Then ok=false

                ' Alle Bedingungen erfüllt? dann gib Singleton zurück und verlasse Funktion
                If ok=true Then
                        Set FindeSingleton=KlasseInArbeit
                        Exit Function
                End If
```

```
        Next i

End Function


Sub Main

        Dim theClasses As ClassCollection
        Dim theClass As Class
        Dim Singleton As Class

        viewport.open

        ' Alle Klassen des Modells erhalten
        Set theClasses=RoseApp.CurrentModel.GetAllClasses()

        ' Suche nach Singleton veranlassen
        Set Singleton=FindeSingleton(theClasses)
        Print "Das Singleton: "+Singleton.Name

End Sub
```

Listing C.2: FindeInterpreter.ebs

```
' Liefert für eine Klasse alle Unterklassen (nicht nur die direkten)
Function GetAllSubclasses (theClass As Class) As ClassCollection
	Dim marked As New ClassCollection
	Dim notmarked As New ClassCollection
	Dim inArbeit As Class

	Set notmarked=theClass.GetSubclasses()
	While notmarked.Count>0
		Set inArbeit=notmarked.GetAt(1)
		notmarked.Remove inArbeit
		marked.Add inArbeit
		notmarked.AddCollection inArbeit.GetSubclasses()
	Wend
	Set GetAllSubclasses=marked
End Function

' Liefert alle normalen Methoden einer Klasse (keine Konstruktoren)
Function GetAllNormalOperations (theClass As Class) As OperationCollection
	Dim alleMethoden As OperationCollection
	Dim alleNormalenMethoden As New OperationCollection
	Dim MethodeInArbeit As Operation

	Set alleMethoden=theClass.Operations
	For i=1 To alleMethoden.Count
		Set MethodeInArbeit=alleMethoden.GetAt(i)
		If MethodeInArbeit.Name<>theClass.Name Then
			alleNormalenMethoden.Add MethodeInArbeit
		End If
	Next i
	Set GetAllNormalOperations=alleNormalenMethoden
End Function


' Liefert alle Wurzelklassen
Function GetAllRoots(theClasses As ClassCollection) As ClassCollection
	Dim KlasseInArbeit As Class
	Dim alleWurzelKlassen As New ClassCollection

	For i=1 To theClasses.Count
		Set KlasseInArbeit=theClasses.GetAt(i)
		If KlasseInArbeit.GetSuperclasses().Count=0 Then
			alleWurzelKlassen.Add KlasseInArbeit
		End If
	Next i
	Set GetAllRoots=alleWurzelKlassen
End Function

' Untersucht zwei Methoden auf Gleichheit
Function CompareOperations(operation1 As Operation, operation2 As Operation) As Boolean
	Dim parameters1 As ParameterCollection
	Dim parameters2 As ParameterCollection
	Dim parameter1 As Parameter
	Dim parameter2 As Parameter
	Dim ok As Boolean

	ok=true
	If operation1.Name<>operation2.Name Then ok=false
	Set parameters1=operation1.Parameters
	Set parameters2=operation2.Parameters
	If parameters1.Count=parameters2.Count Then
		For i=1 To parameters1.Count
			Set parameter1=parameters1.GetAt(i)
			Set parameter2=parameters2.GetAt(i)
			If parameter1.Type<>parameter2.Type Then
				ok=false
			End If
		Next i
	Else ok=false
	End If

	CompareOperations=ok

End Function
```

```
' Sucht nach dem Interpreter-Muster, das erste gefundene wird zurückgegeben
Function FindeInterpreter(alleKlassen As ClassCollection) As ClassCollection
        Dim alleWurzelKlassen As ClassCollection
        Dim Interpreter As New ClassCollection
        Dim KlasseInArbeit As Class
        Dim alleUnterklassen As ClassCollection
        Dim alleMethoden As OperationCollection
        Dim MethodeInArbeit As Operation
        Dim andereKlasseInArbeit As Class
        Dim alleAnderenMethoden As OperationCollection
        Dim andereMethodeInArbeit As Operation

        Dim alleReferenzen As AssociationCollection
        Dim ReferenzInArbeit As Association
        Dim andereRolle As Role

        Dim flag As Boolean
        Dim teilflag1 As Boolean
        Dim teilflag2 As Boolean
        Dim ok As Boolean
        Dim AnzahlReferenzen As Integer

        Set alleWurzelKlassen=GetAllRoots(alleKlassen)
        For i=1 To alleWurzelKlassen.Count
                ok=true

                Set KlasseInArbeit=alleWurzelKlassen.GetAt(i)
                Set alleUnterklassen=GetAllSubclasses(KlasseInArbeit)

                ' Prüfe auf Abstraktheit
                If Not KlasseInArbeit.Abstract Then ok=false

                ' Prüfe, ob eine Methode immer wieder überschrieben wird
                Set alleMethoden=GetAllNormalOperations(KlasseInArbeit)
                flag=false
                For j=1 To alleMethoden.Count
                        Set MethodeInArbeit=alleMethoden.GetAt(j)
                        teilflag1=true
                        For k=1 To alleUnterklassen.Count
                                Set andereKlasseInArbeit=alleUnterklassen.GetAt(k)
                                Set alleAnderenMethoden=GetAllNormalOperations_
                                (andereKlasseInArbeit)
                                teilflag2=false
                                For l=1 To alleAnderenMethoden.Count
                                        Set andereMethodeInArbeit=alleAnderenMethoden.GetAt(l)
                                        If CompareOperations(MethodeInArbeit, _
                                        andereMethodeInArbeit) Then
                                                teilflag2=true
                                                Exit For
                                        End If
                                Next l
                                If teilflag2=false Then
                                        teilflag1=false
                                        Exit For
                                End If
                        Next k
                        If teilflag1=true Then
                                flag=true
                                Exit For
                        End If
                Next j
                If flag=false Then ok=false

                ' Bestimme Anzahl der Referenzen auf Wurzelklasse
                AnzahlReferenzen=0
                For j=1 To alleUnterklassen.Count
                        Set andereKlasseInArbeit=alleUnterklassen.GetAt(j)
                        Set alleReferenzen=andereKlasseInArbeit.GetAssociations()
                        For k=1 To alleReferenzen.Count
                                Set ReferenzInArbeit=alleReferenzen.GetAt(k)
                                Set andereRolle=ReferenzInArbeit._
                                GetOtherRole(andereKlasseInArbeit)
                                If andereRolle.Class.Name=KlasseInArbeit.Name Then
                                        If andereRolle.Cardinality="0..1" Then
                                                AnzahlReferenzen=AnzahlReferenzen+1
                                        Else
```

```
                                        AnzahlReferenzen=_
                                        AnzahlReferenzen+val(andereRolle.Cardinality)
                                End If
                        End If
                Next k
        Next j

        If alleUnterklassen.Count>0 Then
                If AnzahlReferenzen/alleUnterklassen.Count<0.5 Then ok=false
        Else ok=false
        End If

        ' Prüfe auf Referenzen untereinander
        flag=false
        For j=1 To alleUnterklassen.Count
                Set andereKlasseInArbeit=alleUnterklassen.GetAt(j)
                Set alleReferenzen=andereKlasseInArbeit.GetAssociations()
                For k=1 To alleReferenzen.Count
                        Set ReferenzInArbeit=alleReferenzen.GetAt(k)
                        Set andereRolle=ReferenzInArbeit._
                        GetOtherRole(andereKlasseInArbeit)
                        If alleUnterklassen.Exists(andereRolle.Class) Then
                                flag=true
                                Exit For
                        End If
                Next k
        Next j
        If flag=true Then ok=false

        ' Alle Bedingungen erfüllt? dann gib Interpreter zurück und verlasse Funktion
        If ok=true Then
                Set Interpreter=alleUnterklassen
                Interpreter.Add KlasseInArbeit
                Set FindeInterpreter=Interpreter
                Exit Function
        End If

    Next i

End Function


Sub Main

    Dim theClasses As ClassCollection
    Dim KlasseInArbeit As Class
    Dim Interpreter As ClassCollection

    viewport.open

    ' Alle Klassen des Modells erhalten
    Set theClasses=RoseApp.CurrentModel.GetAllClasses()

    ' Suche nach Interpreter veranlassen
    Set Interpreter=FindeInterpreter(theClasses)
    Print "Der Interpreter: "
    For i=1 To Interpreter.Count
        Print "- "+Interpreter.GetAt(i).Name
    Next i

End Sub
```

Listing C.3: FindeKompositum.ebs

```
' Liefert für eine Klasse alle Oberklassen (nicht nur die direkten)
Function GetAllSuperclasses (theClass As Class) As ClassCollection
        Dim marked As New ClassCollection
        Dim notmarked As New ClassCollection
        Dim inArbeit As Class

        Set notmarked=theClass.GetSuperclasses()
        While notmarked.Count>0
                Set inArbeit=notmarked.GetAt(1)
                notmarked.Remove inArbeit
                marked.Add inArbeit
                notmarked.AddCollection inArbeit.GetSuperclasses()
        Wend
        Set GetAllSuperclasses=marked
End Function


' Liefert für eine Klasse alle Unterklassen (nicht nur die direkten)
Function GetAllSubclasses (theClass As Class) As ClassCollection
        Dim marked As New ClassCollection
        Dim notmarked As New ClassCollection
        Dim inArbeit As Class

        Set notmarked=theClass.GetSubclasses()
        While notmarked.Count>0
                Set inArbeit=notmarked.GetAt(1)
                notmarked.Remove inArbeit
                marked.Add inArbeit
                notmarked.AddCollection inArbeit.GetSubclasses()
        Wend
        Set GetAllSubclasses=marked
End Function


' Sucht nach dem Kompositum-Muster, das erste gefundene wird zurückgegeben
Function FindeKompositum(alleKlassen As ClassCollection) As ClassCollection
        Dim KlasseInArbeit As Class
        Dim alleKompositumKlassen As New ClassCollection
        Dim alleReferenzen As AssociationCollection
        Dim ReferenzInArbeit As Association
        Dim andereRolle As Role
        Dim flag As Boolean

        For i=1 To alleKlassen.Count

                flag=false
                Set KlasseInArbeit=alleKlassen.GetAt(i)
                Set alleReferenzen=KlasseInArbeit.GetAssociations()
                For j=1 To alleReferenzen.Count
                        Set ReferenzInArbeit=alleReferenzen.GetAt(j)
                        Set andereRolle=ReferenzInArbeit.GetOtherRole(KlasseInArbeit)
                        If val(andereRolle.Cardinality)>1 Then
                                If GetAllSuperclasses(KlasseInArbeit)._
                                Exists(andereRolle.Class) Then
                                        flag=true
                                        Exit For
                                End If
                        End If
                Next j

                If flag=true Then
                         Set alleKompositumKlassen=GetAllSubclasses(andereRolle.Class)
                         alleKompositumKlassen.Add andereRolle.Class
                         Set FindeKompositum=alleKompositumKlassen
                         Exit Function
                End If

        Next i

End Function


Sub Main
```

```
        Dim theClasses As ClassCollection
        Dim Kompositum As ClassCollection
        Dim KompositumKlasseInArbeit As Class


        viewport.open

        ' Alle Klassen des Modells erhalten
        Set theClasses=RoseApp.CurrentModel.GetAllClasses()

        ' Suche nach Kompositum veranlassen
        Set Kompositum=FindeKompositum(theClasses)
        Print "Das Kompositum : "
        For i=1 To Kompositum.Count
                Set KompositumKlasseInArbeit=Kompositum.GetAt(i)
                Print "- "+KompositumKlasseInArbeit.Name
        Next i

End Sub
```

D Glossar

Abstrakte Klasse

Eine Klasse, deren Hauptaufgabe darin besteht, eine Schnittstelle zu definieren. Eine abstrakte Klasse delegiert Teile oder auch die gesamte Implementierung an ihre Unterklassen. Von einer abstrakten Klasse können keine Objekte erzeugt werden. [Gamma+96]

Abstrakte Operation

Eine Operation, die eine Signatur deklariert, sie aber nicht implementiert. In C++ entspricht eine abstrakte Operation einer rein virtuellen (*pure virtual*) Member-Funktion. [Gamma+96]

Aggregationsbeziehung

Die Beziehung zwischen einem aggregierten Objekt und seinen Teilen. Eine Klasse definiert diese Beziehung für ihre Exemplare (zum Beispiel aggregierte Objekte).

Aggregiertes Objekt

Ein Objekt, das aus weiteren untergeordneten Objekten zusammengesetzt ist. Diese untergeordneten Objekte heißen Teile des Aggregats. Das Aggregat ist für sie zuständig. [Gamma+96]

Anfrage

Ein Objekt führt eine Operation aus, wenn es eine entsprechende Anfrage von einem anderen Objekt erhält. Ein übliches Synonym für Anfrage ist Nachricht. [Gamma+96]

Anwendungsverstehen

(engl. *Application Understanding*) Anwendungsverstehen ist der aktive Vorgang des Erkennens von Systemstrukturen und -eigenschaften von Softwaresystemen (Solche Strukturen und Eigenschaften werden zum Beispiel durch die folgenden Fragen charakterisiert: welche Programme, Datenbanken, Dateien etc. gehören zu einer Anwendung? welche Daten werden erzeugt? welche Abhängigkeiten existieren zwischen den Programmen der Anwendung?). [Müller97]

Assoziationsbeziehung

Eine Assoziationsbeziehung beschreibt eine Relation zwischen Klassen, das heißt die gemeinsame Semantik und Struktur einer Menge von Objektbeziehungen. [Oestereich98]

Bekanntschaftsbeziehung

Eine Klasse, die eine andere Klasse referenziert, hat eine Bekanntschaftsbeziehung mit dieser Klasse. [Gamma+96]

Black-Box-Wiederverwendung

Ein Wiederverwendungsstil, der auf Objektkomposition basiert. Zusammengesetzte Objekte zeigen einander keine internen Details und gleichen somit "Black Boxes". [Gamma+96]

CARE

Computer Aided Reengineering bzw. Reverse-Engineering. In Anlehnung an CASE die computerunterstützte Wartung von Software. [Müller97]

CASE

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

Delegation

Eine Implementierungstechnik, bei der Objekte eine Anfrage an ein anderes Objekt weiterleiten (delegieren). Das Objekt, an das die Anfrage delegiert wird, arbeitet die Anfrage anstelle des Originalobjekts ab. [Gamma+96]

Design Pattern

Siehe Entwurfsmuster.

Design-Recovery

Design Recovery ist der Versuch, aus vorliegendem Quellcode, existierenden Entwurfsdokumenten (wenn verfügbar), persönlichen Erfahrungen und allgemeinem Wissen über die Anwendungsdomäne eine Systemarchitektur wiederzugewinnen. Die Rekonstruktion des Software-Entwurfs ist heute eine aufwendige, teure und kreative Tätigkeit. Durch ausschließlich automatische Analyse des Quellcodes lässt sich die Software-Architektur nicht ermitteln. [Balzert98]

Elternklasse

Die Klasse, von der eine andere Klasse erbt. Synonyme sind Oberklasse (*Smalltalk*), Basisklasse (*C++*) und Vorfahrenklasse (engl. *Ancestor Class*). [Gamma+96]

Empfänger

Das Zielobjekt einer Anfrage. [Gamma+96]

Entwurfsmuster

Ein Entwurfsmuster benennt, motiviert und erläutert systematisch einen allgemeinen Entwurfs, der ein in objektorientierten Systemen immer wiederkehrendes Entwurfsproblem löst. Es beschreibt das Muster, die Lösung, wann die Lösung anwendbar ist sowie die Konsequenzen der Anwendung. Es gibt weiterhin Implementierungstips und Beispiele. Die Lösung ist eine allgemeine Anordnung von Objekten und Klassen, die das Problem lösen. Die Lösung wird maßgeschneidert und implementiert, um das Problem in einem konkreten Kontext zu lösen. [Gamma+96]

Exemplar

Ein Exemplar ist ein Objekt einer Klasse. [Gamma+96]

Forward-Engineering

Forward Engineering ist der Transformationsprozess einer Spezifikation von einem höheren in ein niedrigeres Abstraktionsniveau. (Das höhere Abstraktionsniveau kann zum Beispiel ein logisches, implementierungsunabhängiges Design in einer formalen oder nicht formalen Notation sein. Das Ziel der Transformation ist in der Regel Programmiersprachennotation.) [Müller97]

Framework

Eine Menge kooperierender Klassen, welche die Elemente eines wiederverwendbaren Entwurfs für eine bestimmte Art von Software darstellen. Ein Framework bietet eine Architekturhilfe beim Aufteilen des Entwurfs in abstrakte Klassen und beim Definieren ihrer Zuständigkeiten und Interaktionen. Ein Entwickler passt das Framework für eine bestimmte Anwendung an, indem er Unterklassen der Frameworkklassen bildet und ihre Objekte zusammensetzt. [Gamma+96]

Kapselung

Das Ergebnis des Versteckens von Repräsentation und Implementierung innerhalb eines Objekts. Die Repräsentation ist nicht sichtbar. Man kann nicht direkt von außerhalb des Objekts auf sie zugreifen. Operationen sind die einzige Möglichkeit, auf die Repräsentation eines Objekts zuzugreifen und sie zu modifizieren. [Gamma+96]

Kindklasse

Siehe Unterklasse.

Klasse

Eine Klasse definiert die Objektschnittstelle und ihre Implementierung. Sie definiert die interne Repräsentation und die Operationen, die das Objekt ausführen kann. [Gamma+96]

Klassendiagramm

Ein Diagramm, das Klassen, ihre interne Struktur und Operationen sowie die statischen Beziehungen zwischen ihnen darstellt. [Gamma+96]

Klassenoperation

Eine Operation, die auf eine Klasse und nicht auf ein einzelnes Objekt angewendet wird. In *C++* kann man Klassenoperationen als statische Member-Funktionen implementieren. [Gamma+96]

Konkrete Klasse

Eine Klasse ohne abstrakte Operationen. Es können von ihr Objekte erzeugt werden. [Gamma+96]

Konstruktor

In *C++* eine Operation, die automatisch aufgerufen wird, um ein neues Exemplar zu initialisieren. [Gamma+96]

Kopplung

Das Ausmaß, in dem Softwarekomponenten voneinander abhängen. [Gamma+96]

Legacy Systems

Siehe Software-Altsysteme.

Metrik

Eine Metrik quantifiziert eine bestimmte Eigenschaft eines Objekts, zum Beispiel das Volumen (in m³) eines Körpers. Bekannte Metriken für Softwaresysteme sind etwa McCabe's zyklomatische Komplexität oder Halstead's Volumen. [Müller97]

Objekt

Ein zur Laufzeit existierendes "Ding", das Daten und Operationen, die auf diesen Daten arbeiten, zusammenfasst. [Gamma+96]

Objektkomposition

Das Zusammensammeln oder Komponieren von Objekten, um komplexeres Verhalten zu erhalten. [Gamma+96]

OLE

Object Linking and Embedding. Eine Microsoft Technologie, die es erlaubt, Elemente verschiedener Anwendungen miteinander zu verknüpfen. Zum Beispiel ist es möglich, ein Excel-Diagramm in eine PowerPoint Präsentation einzugliedern; wenn das Excel-Diagramm eine Änderung erfährt, so wird diese auch in der PowerPoint-Präsentation sichtbar. [myTutorials00]

OMT

Object Modeling Technique. Eine Notation für Software-Modelle von James Rumbaugh.

Operation

Die Daten eines Objekts können nur durch seine Operationen manipuliert werden. Ein Objekt führt eine Operation aus, wenn es eine Anfrage erhält. In C++ heißen Operationen Member-Funktionen, in Smalltalk heißen sie Methoden. [Gamma+96]

Polymorphie

Die Möglichkeit, Objekte passender Schnittstellen zur Laufzeit füreinander einzusetzen. [Gamma+96]

Präzision

Bezeichnet das Verhältnis von echten zu falsch erkannten Mustern einer Suchanfrage.

Programmverstehen

(engl. *Program Understanding*): Programmverstehen ist der aktive Vorgang des Erkennens der internen also technischen Arbeitsweise eines Programms. (Charakterisierende Fragen sind etwa: welche Prozedur ruft welche Prozedur auf? welcher Pfad wird eingeschlagen, wenn Variable X den Wert Y hat? wo wird Variable X gelesen, wo geschrieben? was sind die Auswirkungen einer bestimmten Änderung?). In neueren Veröffentlichungen wird häufig nicht mehr zwischen Anwendungsverstehen und Programmverstehen unterschieden und nur noch der Begriff Programmverstehen verwendet. [Müller97]

Recall

Bezeichnet das Verhältnis von gefundenen Dokumenten zu in der angefragten Datenbank tatsächlich enthaltenen Dokumenten. In diesem Fall sind mit Dokumenten Muster gemeint, und Datenbank steht für ein Softwaresystem.

Redokumentation

(engl. *Redocumentation*) Unter Redokumentation versteht man die Erzeugung einer semantisch äquivalenten Repräsentation des betrachteten Objekts (Programm oder andere Spezifikation). Die Repräsentation geschieht auf demselben Abstraktionsniveau und ist somit eine alternative Darstellungsform. [Müller97]

Reengineering

Unter Reengineering werden alle Aktivitäten nach Inbetriebnahme eines Programmsystems zusammengefasst, die das Verständnis von Software erhöhen oder die Wartbarkeit, Wiederverwendbarkeit oder Weiterentwickelbarkeit von Software verbessern oder erst ermöglichen. [Arnold93 zitiert nach Müller97]

Referenz

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

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]

Sanierung

Herstellung der wirtschaftlichen, fachlichen und/oder softwaretechnischen Leistungsfähigkeit eines Software-Altsystems durch eine geeignete Kombination von Reverse Engineering-, Reengineering- und Forward Engineering-Maßnahmen. [Balzert98]

Schnittstelle

Die Menge aller durch die Operationen eines Objekts definierten Signaturen. Die Schnittstelle beschreibt die Menge von Anfragen, auf welche ein Objekt antworten kann. [Gamma+96]

Signatur

Die Signatur einer Operation definiert ihren Namen, Parameter und Rückgabewert. [Gamma+96]

Software-Altsysteme

(engl. *Legacy Systems*) Softwaresysteme, die aus dem Blickwinkel der Gegenwart, mit veralteten Software-Methoden und -Konzepten entwickelt wurden, und ohne eine Sanierung für den vorgesehenen Einsatzzweck und die vorgesehene Einsatzumgebung

nicht oder nicht mehr wirtschaftlich und/oder fachlich verwendet werden können. [Balzert98]

Subsystem

Eine unabhängige Gruppe von Klassen, die zusammenarbeiten, um eine bestimmte Menge von Aufgaben zu erfüllen. [Gamma+96]

Überschreiben

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

UML

Unified Modeling Language. Ist eine Sprache und Notation zur Modellierung von Softwaresystemen.

Unterklasse

Eine Klasse, die von einer anderen Klasse erbt. In C++ heißt eine Unterklasse abgeleitete Klasse. [Gamma+96]

Vererbung

Eine Beziehung, die ein "Ding" auf Basis eines anderen "Dings" definiert. Klassenvererbung definiert eine neue Klasse auf Basis einer oder mehrerer Elternklassen. Die neue Klasse erbt seine Schnittstelle und Implementierung von seinen Elternklassen. Die neue Klasse heißt Unterklasse oder (in C++) abgeleitete Klasse. Klassenvererbung kombiniert Schnittstellenvererbung mit Implementierungsvererbung. Schnittstellenvererbung definiert eine neue Schnittstelle auf Basis einer oder mehrerer Schnittstellen. Implementierungsvererbung definiert eine neue Implementierung auf Basis einer oder mehrerer existierender Implementierungen. [Gamma+96]

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.

White-Box-Wiederverwendung

Ein auf Klassenvererbung basierender Stil der Wiederverwendung. Eine Unterklasse verwendet die Schnittstelle und Implementierung seiner Elternklasse wieder, kann aber möglicherweise Zugriff zu ansonsten privaten Aspekten seiner Elternklassen haben. [Gamma+96]

E Literaturverzeichnis

[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

[Arnold93] Robert S. Arnold, editor. *Software Reengineering.* IEEE Computer Society Press, 1993.

[Balzert98] Helmut Balzert: *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung.* Spektrum Akademischer Verlag, 1998.

[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/

[Cashman+80] P. M. Cashman and A. W. Holt. *A Communication-Oriented Approach to Structuring the Software Maintenance Environment.* Software Engineering Notes, 5(1), January 1980.

[Chikofsky90] Elliot J. Chikofsky and James H. Cross II. *Reverse Engineering and Design Recovery: A Taxonomy.* IEEE Software, 7(1), January 1990.

[deRose+78] B. de Rose and T. Nyman. *The Software Life Cycle - A Management and Technology Challenge in the Department of Defense.* IEEE Transaction on Software Engineering, SE-4(4), July 1978.

[Duden88] *DUDEN Informatik – Ein Sachlexikon für Studium und Praxis.* Dudenverlag, 1988.

[Duden90] *DUDEN Fremdwörterbuch*. Dudenverlag, 1990.

[Fjeldstad+79] R. K. Fjeldstad and W. T. Hamlen. *Application Program Maintenance Study - Report to our Respondents.* In Proc. GUIDE 48, Philadelphia, PA, 1979.

[Foote99] Brian Foote. *A Smalltalk Patterns Safari.* Slide-Show, 1999. http://www.laputan.org/talks/safari/sld001.htm

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

[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

[Lientz+80] Bennet P. Lientz and E. Burton Swanson. *Software Maintenance Management.* Addison-Wesley, 1980.

[McClure92] Carma McClure. *The Three Rs of Software Automation – Re-engineering Repository Reusability.* Prentice Hall, 1992

[McKee84] J. R. McKee. *Maintenance as a Function of Design.* In Proc. AFIPS National Computer Conference, 1984.

[Mills76] H. D. Mills. *Software Development.* IEEE Transaction on Software Engineering, SE-2(4), December 1976.

[Müller97] Bernd Müller. *Reengineering. Eine Einführung.* B.G. Teubner Stuttgart 1997.

[myTutorials00] Tutorials und Definitionen. 2000. http://www.mytutorials.com/define/

[Neumann+99] Rainer Neumann, Benedikt Schulz, Wolf Zimmermann. *Vererbung bei der Sanierung objektorientierter Systeme.* In Proceedings of the 1st German Workshop on Software-Reengineering, S. 121-124, 1999. http://www.uni-koblenz.de/~ist/RWS99/beitraege/NeumannSchulzZimmermann.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

[Oestereich98] Bernd Oestereich. *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified modeling language.* Oldenbourg, 1998.

[Prechelt+97] Lutz Prechelt, Barbara Unger, Michael Philippsen. *Documenting Design Patterns in Code Eases Program Maintenance.* In Proc. of the ICSE Workshop on Process Modeling and Empirical Studies of Software Evolution, pp. 72-76, Boston, MA, May 1997. http://citeseer.nj.nec.com/cache/papers2/cs/1428/http:zSzzSzwwwipd.ira.uka.dezSz~precheltzSzBibliozSzjakk_pmesse97.pdf/prechelt97documenting.pdf

[Rational00] *Using the Rose Extensibility Interface. Rational Rose 2001.* Rational Software Corporation, 2000.

[Schneider86] Hans-Jochen Schneider, editor. *Lexikon der Informatik und Datenverarbeitung.* Oldenbourg Verlag, 2. edition, 1986.

[Sen80] Sentry Market Research. CASE 1988-1989. Westborough, MA, 1980.

[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

[Tatsubori+98] Michiaki Tatsubori, Shigeru Chiba. *Programming Support of Design Patterns with Compile-time Reflection.* OOPSLA'98 Workshop Reflective Programming in C++ and Java, Vancouver, Canada, 1998.

[Vlissides99] John Vlissides. *Entwurfsmuster anwenden.* Addison-Wesley 1999.

[West93] Richard West. *Reverse Engineering - An Overview.* HMSO, London, 1993.

F Inhalt der CD

\Diplomarbeit\Diplomarbeit von Sebastian Naumann.doc
\Diplomarbeit\Diplomarbeit von Sebastian Naumann.pdf
\Diplomarbeit\Strukturdiagramme der Entwurfsmuster.doc
\Diplomarbeit\Sonstige Abbildungen.doc
\Diplomarbeit\Thesen.doc
\Diplomarbeit\Thesen.pdf

\Implementierte Muster\Abstrakte Fabrik
\Implementierte Muster\Adapter
\Implementierte Muster\Besucher
\Implementierte Muster\Bridge
\Implementierte Muster\Command
\Implementierte Muster\Decorator
\Implementierte Muster\Erbauer
\Implementierte Muster\Fabrikmethode
\Implementierte Muster\Fassade
\Implementierte Muster\Fliegengewicht
\Implementierte Muster\Interpreter
\Implementierte Muster\IteratorMuster
\Implementierte Muster\Kompositum
\Implementierte Muster\Memento
\Implementierte Muster\Observer
\Implementierte Muster\Prototyp
\Implementierte Muster\Proxy
\Implementierte Muster\SchablonenMethode
\Implementierte Muster\Singleton
\Implementierte Muster\Strategie
\Implementierte Muster\Vermittler
\Implementierte Muster\Zustaendigkeitskette
\Implementierte Muster\Zustand

\Paper\Antoniol+98 - vollständige Strukturen - mehrstufiger Suchprozess.pdf
\Paper\Bansiya98 - Schlüsselstrukturen - DP++.pdf
\Paper\Bergenti+00 - vollständige Strukturen - IDEA.pdf
\Paper\Brown96 - Schlüsselstrukturen - KT.pdf
\Paper\Keller+99 - Schlüsselstrukturen - SPOOL.pdf
\Paper\Kim+00 - Metriken - Pattern Wizard.pdf
\Paper\Krämer+96 - vollständige Strukturen - Pat.pdf
\Paper\Neumann+99 - Vererbung bei der Sanierung objektorientierter Systeme.pdf
\Paper\Niere+01 - flexible Musterdefinition und Fuzzy Logik.pdf
\Paper\Prechelt+97 - Documenting Design Patterns.pdf
\Paper\Shull+96 - induktive Methode - BACKDOOR.pdf

\Rational Rose Modelle\abstraktefabrik.mdl
\Rational Rose Modelle\abstraktefabrik2.mdl
\Rational Rose Modelle\composite2.mdl
\Rational Rose Modelle\composite.mdl
\Rational Rose Modelle\interpreter.mdl
\Rational Rose Modelle\kompositum.mdl
\Rational Rose Modelle\Liste.mdl
\Rational Rose Modelle\memento.mdl
\Rational Rose Modelle\zustand.mdl

\Rational Rose Skripte\FindeSingleton.ebs
\Rational Rose Skripte\FindeInterpreter.ebs
\Rational Rose Skripte\FindeKompositum.ebs

Thesen

- Jedes Softwaresystem muss gewartet werden.
- Um ein Softwaresystem warten zu können, muss man zuvor seine Funktionsweise verstanden haben.
- Objektorientierte Entwurfsmuster sind elegante Lösungen zu ständig wiederkehrenden Problemen.
- Entwurfsmuster sind für das Programmverstehen essentiell wichtig, da sie durch ihre Struktur auf ihre dahinterstehende Idee verweisen. Dadurch wird der Sinn von Klassen, die gemeinsam ein Entwurfsmuster bilden, sofort deutlich. Entwurfsmuster besitzen ein höheres Abstraktionsniveau als ein Softwaremodell.
- Die genaue Kenntnis von Entwurfsmustern ist Voraussetzung, um aus ihnen beim Programmverstehen Nutzen ziehen zu können. Sich das genaue Wissen über Entwurfsmuster anzueignen, braucht viel Zeit und praktische Erfahrung.
- Für den Menschen ist es sehr schwierig, in sehr großen Softwaresystemen gar unmöglich, manuell nach Entwurfsmustern zu suchen. Dies macht die Unterstützung durch ein Werkzeug erforderlich, das das Auffinden von Entwurfsmustern automatisch ausführt.
- Die derzeit vorhandenen Ansätze für die automatische Suche nach Entwurfsmustern sind unzureichend; entweder werden nicht alle Muster aus [Gamma+96] abgedeckt oder eindeutig definierte Muster können nicht eindeutig identifiziert werden.
- Ansätze, die ihre Suche auf minimal vorhandene Schlüsselstrukturen stützen, sind in der Lage, eindeutig definierte Muster auch eindeutig zu identifizieren. Das Defizit dieser Ansätze liegt in der nicht vollständigen Abdeckung aller Entwurfsmuster aus [Gamma+96]. Es können jedoch auf diese Weise für alle Muster solch minimale Schlüsselstrukturen festgelegt werden.
- Die Hinzunahme von Merkmalen, die nicht vorhanden sein dürfen, führt zu keiner positiven Identifizierung von Entwurfsmustern, senkt allerdings die Rate der falsch erkannten.
- Das CASE-Werkzeug *Rational Rose* ist für die automatische Suche nach Entwurfsmustern nicht geeignet, da verschiedene Elemente wie Methodenaufrufe und Variablenbenutzung nicht vorgesehen sind.
- Die graphische Darstellung erkannter Muster ist für das Programmverstehen von herausragender Bedeutung.
- Für einige Entwurfsmuster ist eine Validierung der von mir festgelegten Merkmale nötig. Diese Aufgabe kann durch die Untersuchung vieler verschiedener Implementie-

rungen eines Musters erfolgen. Die Entwicklung dieser vielfältigen Implementierungen erfordert ein Team erfahrener Softwareentwickler.

- Der Wartung wird in der Forschung und in der Lehre nicht genügend Beachtung geschenkt. Keine deutsche Hochschule besitzt einen Lehrstuhl für die Wartung von Software (Stand 1996/97).

Ilmenau, den 26. November 2001 Sebastian Naumann

Erklärung

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

Ilmenau, den 29. November 2001 Sebastian Naumann