Strukturuntersuchung großer Softwaresysteme

Zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
von der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)
genehmigte

Dissertation

von

Markus Bauer

aus Kirchheim unter Teck

Tag der mündlichen Prüfung:	20. Dezember 2005
Erster Gutachter:	Prof. Dr. Gerhard Goos
Zweiter Gutachter:	Prof. Dr. Welf Löwe

Danksagung

Mein ganz besonderer Dank gilt meinem Doktorvater Prof. Dr. Gerhard Goos. Seine Fragen und Anregungen haben meiner Arbeit in allen Phasen ihres Entstehens immer wieder wichtige Denkanstöße gegeben und sie in die richtige Richtung gelenkt. Meinem Koreferenten Prof. Dr. Welf Löwe danke ich für die gute Betreuung, seine Ermutigungen und wertvollen Hinweise zu Zeiten, in denen meine Arbeit neue Impulse benötigte.

Eine gute wissenschaftliche Arbeit kann immer nur im Team entstehen. In disesem Sinne bedanke ich mich bei meinen Kollegen der Gruppe Programmstrukturen am Forschungszentrum Informatik in Karlsruhe. Helmut Melcher, Oliver Ciupke, Benedikt Schulz, Thomas Genßler, Holger Bär, Christoph Andriessens, Olaf Seng, Adrian Trifu, Mircea Trifu, Volker Kuttruff und Peter Szulman danke ich nicht nur für zahlreiche inspirierende Diskussionen, sondern insbesondere den letztgenannten auch für ihre Unterstützung, welche mich bei meiner Arbeit als Abteilungsleiter in der Endphase meiner Promotion sehr entlastete.

Die Idee zu dieser Arbeit entstand im Forschungsprojekt FAMOOS. Ich danke meinen Mitstreitern von der Universität Bern, Nokia, DaimlerChrysler und der Sema Group für die gute Zusammenarbeit und vor allem Sander Tichelaar, Radu Marinescu, Oliver Ciupke und Benedikt Schulz für die vielen Diskussionen zum Thema Softwaresanierung. Olaf Seng danke ich für die vielen langen und kurzen Tage während mehrerer Projekte in Finnland, in denen wir unser Verständnis über Softwarestrukturen und deren Qualität ausbauen konnten, welches die inhaltliche Grundlage dieser Arbeit darstellt.

Auch meinen Kollegen am Lehrstuhl von Herrn Prof. Dr. Goos gebührt mein Dank. Insbesondere danke ich Uwe Assmann, Rainer Neumann, und Andreas Ludwig für zahlreiche gute Anregungen und für das System *Recoder*, welches als Basis für die Implementierung meiner Arbeit diente. Rainer Neumann und Welf Löwe verdanke ich viele Einsichten und methodische Hinweise, die mir das Anfertigen dieser Dissertation deutlich erleichtert haben.

Danken möchte ich auch meinen Diplomanden Adrian Trifu, Mircea Trifu und Matthias Biehl, die zahlreiche Ideen und große Teile der Implementierung meines Verfahrens in Form des Werzeuges *ACT* beisteuerten.

Für die gründliche Durchsicht meiner Arbeit danke ich Rainer Neumann, Holger Bär, Olaf Seng, Volker Kuttruff und meinem Vater.

Ganz besonders bedanke ich mich bei meiner Familie und einigen engen Freunden, ohne die diese Arbeit nie fertig geworden wäre. Meine Eltern haben in mir die Neugier und

den Wissensdurst geweckt, deren jüngstes Ergebnis diese Arbeit ist. Meine Schwester Anne und meine Freunde – allen voran Berit Pölkow und Ralf Kowalsky – haben mich insbesondere in den letzten Monaten, beim Zusammenschreiben der Ergebnisse meiner Arbeit, in besonderem Maße ermuntert und bestärkt.

Karlsruhe, im Mai 2006,

Markus Bauer

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Problemstellung und Ziel

Die Herausforderung bei der Entwicklung großer und moderner Softwaresysteme liegt heute nicht mehr nur darin, geeignete Datenstrukturen und Algorithmen zur Lösung von Anwendungsproblemen zu finden, sondern vielmehr darin, geeignete Systemstrukturen zu finden, mit denen sich die Systeme kostengünstig und mit hoher Qualität entwickeln lassen. Nur mit geeigneten Systemstrukturen und entsprechend übersichtlichen Systementwürfen können Softwareingenieure die Komplexität moderner Systeme überhaupt bewältigen und langlebige, wartbare und weiterentwicklungsfähige Systeme konstruieren. Auch für organisatorische Aspekte der Softwareentwicklung (beispielsweise arbeitsteilige Softwareentwicklung in Teams, Wiederverwendung) spielt die Struktur eines Softwaresystems eine zentrale Rolle.

In der Praxis finden wir leider zahlreiche Softwaresysteme, deren Struktur sich den mit der Wartung und Weiterentwicklung beauftragten Softwareingenieuren nicht mehr ausreichend erschließt. Dies liegt daran, dass Softwaresysteme Alterungsprozessen unterliegen, in deren Verlauf Strukturen verwässern: Wartungsarbeiten und Weiterentwicklungen, die von Softwareentwicklern vorgenommen werden, die mit den Konzepten der Systemstruktur nicht ausreichend vertraut sind, führen zu immer zahlreicheren Strukturverletzungen – bis irgendwann eine ursprünglich sauber durchdachte Struktur weitestgehend verloren geht. Dieses Phänomen wird noch verschärft: nur allzu häufig existiert keinerlei taugliche Dokumentation von Systemstrukturen, entweder, weil solche Dokumente überhaupt nie erstellt wurden oder weil sie während der Weiterentwicklung und Wartung der Systeme nicht entsprechend mitgepflegt wurden. Aufwand und Kosten für die Pflege und Weiterentwicklung solcher Softwaresysteme sind hoch – in manchen Fällen sind Pflege und Weiterentwicklung sogar so aufwändig, dass notwendige Arbeiten gar nicht mehr mit vertretbarem Aufwand durchgeführt werden können und somit unterbleiben.

Einen Ausweg aus dieser Problematik bieten Verfahren, mit deren Hilfe die Struktur eines existierenden Systems untersucht werden kann. Solche Verfahren müssen in der Lage sein, aus der Implementierung eines Systems (also aus dessen Quelltext) eine geeignete Systemstruktur zu extrahieren. Diese kann dann zunächst als Grundlage zur Do-

kumentation der bestehenden Struktur des Systems genutzt werden. Zudem kann sie als Ausgangspunkt für eine systematische Säuberung und Überarbeitung der Systemstruktur verwendet werden, um das System für zukünftige Weiterentwicklungen in Form zu bringen. Hiermit beschäftigt sich die vorliegende Arbeit:

Ziel der Arbeit ist ein Verfahren, das die Gewinnung von Systemstrukturen aus der Implementierung realer, objektorientierter Softwaresysteme praktikabel macht.

Die Struktur eines Softwaresystems definiert sich über seine Bauteile und die Beziehungen zwischen diesen Bauteilen. Solche Bauteile können von unterschiedlichster Granularität sein, beginnend bei Implementierungsstrukturen (beispielsweise Methoden und Klassen) bis hin zu Teilsystemen. Teilsysteme sind naturgemäß von besonders großer Bedeutung für den Entwurf großer Systeme. Allerdings können in der Regel nur Informationen über die Implementierungsstrukturen direkt aus dem Quelltext eines Systems gewonnen werden. Verfahren zur Ableitung einer günstigen Zerlegung des Systems in Teilsysteme bilden daher den Schwerpunkt dieser Arbeit.

Damit wir ein solches Verfahren für nutzbringend halten, muss es bestimmten Kriterien genügen. Wir erwarten, dass wir ein Verfahren zur Extraktion von Systemstrukturen für große Systeme einsetzen können *(Skalierbarkeit)*. Unter großen Systemen verstehen wir solche, deren Quelltext mehrere hunderttausend oder gar mehrere Millionen Codezeilen umfasst. Die Fülle von Bauteilen, aus denen solche Systeme bestehen, erfordert es, dass ein solches Verfahren *automatisiert* durchgeführt werden kann. Es müssen hierzu Softwarewerkzeuge erstellt werden können, um die Anwendung des Verfahrens zu erleichtern. Zudem soll das Verfahren in der Lage sein, beliebige Systeme zu bearbeiten. Es muss also weitgehend unabhängig von Anwendungsdomäne, Implementierungstechnik, Programmiersprache und Infrastrukturtechnik sein bzw. sich leicht an unterschiedliche dieser Rahmenbedingungen anpassen lassen *(Allgemeinheit)*.[1] Zudem spielt natürlich die *Qualität* der Ergebnisse des Verfahrens eine wesentliche Rolle. Das Verfahren muss eine Zerlegung des Softwaresystems liefern, die dem Softwareingenieur bei dem Verständnis des Systems bzw. bei der Wartung und Weiterentwicklung des Systems von Nutzen ist. Dies bedeutet, dass Bauteile, die für den Softwareingenieur konzeptionell zusammengehören, auch gemeinsam in ein Teilsystem aufgenommen werden. Umgekehrt dürfen Teilsysteme, wie sie durch das Verfahren berechnet werden, dann auch nur zusammengehörende Bauteile des Systems enthalten. Zusätzlich müssen sämtliche Bauteile des Systems geeigneten Teilsystemen zugeordnet werden *(Vollständigkeit)*.

1.2 Verwandte Arbeiten

Die meisten existierenden Ansätze zur Gewinnung von Systemstrukturen basieren auf der Idee, aus Implementierungskonzepten, die sich leicht aus dem Quelltext eines Systems gewinnen lassen, Systemstrukturen abzuleiten. Diese Ansätze lassen sich in zwei

[1] Wir schränken uns in der vorliegenden Arbeit bereits etwas ein, indem wir objektorientierte Systeme betrachten.

Gruppen einteilen. Es gibt Ansätze, die auf dem Prinzip der *Mustersuche* basieren. Hierbei werden werkzeuggestützt zuvor spezifizierte Strukturmuster in der Implementierungsstruktur der Systeme identifiziert. Die Probleme dieser Ansätze beruhen darauf, dass ihre Anwendbarkeit von der Definition geeigneter, auffindbarer Strukturmuster abhängt. Die Praxis zeigt, dass sich kaum geeignete Muster finden lassen, mit deren Hilfe beliebige Systeme vollständig in Teilsysteme zerlegt werden können.

Andere Ansätze beruhen auf Ähnlichkeitsmaßen, die sich zwischen den Bauteilen des Systems definieren lassen, und *Ballungsanalysealgorithmen*, die zusammengehörige Bauteile auf Basis dieser Maße zu Teilsystemen zusammenstellen können. Die Praxistauglichkeit dieser Ansätze ist bisher nur eingeschränkt gegeben, da sie in vielen Fällen Systemzerlegungen produzieren, die für Softwareingenieure nur schwer nachvollziehbar und verständlich sind. Die Schwierigkeit liegt hier darin, geeignete Ähnlichkeitsmaße zu entwickeln, mit denen sich bessere Systemzerlegungen gewinnen lassen.

Daneben gibt es auch noch weitere Ansätze, die sich nicht primär auf den Quelltext des zu untersuchenden Systems abstützen, sondern extern *vorhandenes Entwicklerwissen* zur Bildung von Teilsystemstrukturen heranziehen. Dazu verwenden sie Entwicklerbefragungen oder die existierende Systemdokumentation. Solche Ansätze versagen häufig, da eben dieses Entwicklerwissen kaum mehr ausreichend vorhanden ist – die ursprünglichen Entwickler stehen meist nicht mehr zur Verfügung, eine geeignete Dokumentation fehlt oder ist nicht mehr aktuell.

1.3 Lösungsansatz und Ergebnisse

Der in dieser Arbeit vorgestellte *hybride Ansatz zur Extraktion von Systemstrukturen* beruht auf einer Kombination von Mustersuche und Ballungsanalyse. Er gliedert sich in die drei Schritte *Modellextraktion*, Modellanreicherung durch *Mustersuche* und die eigentliche Zerlegung des Systems in Teilsystemstrukturen durch die *Ballungsanalyse*.

Die Implementierungsstruktur eines konkreten Systems wird in dieser Arbeit über ein Strukturmodell dargestellt. Dieses enthält die Bauteile des Systems (in Form von Klassen und Methoden) und die Beziehungen zwischen diesen Bauteilen. Die zum Aufbau des Strukturmodells erforderlichen Informationen werden im Rahmen der Modellextraktion durch gängige Übersetzerbautechnik aus der Implementierung des Systems in Form des Quelltextes gewonnen.

Die Bauteile des Systems werden dann mit Hilfe eines Algorithmus zur Ballungsanalyse Teilsystemen zugeordnet. Die Ballungsanalyse zieht Ähnlichkeitsmaße heran, um zu bewerten, ob jeweils zwei Bauteile zusammen in ein Teilsystem platziert werden sollen. Wir definieren diese Maße so, dass die Entscheidung, ob zwei Bauteile zusammen gruppiert werden, Grundsätzen eines guten Systementwurfs genügt. Beispielsweise müssen die durch die Ballungsanalyse berechneten Teilsysteme über eine gewisse innere Kohäsion verfügen, wohingegen die externe Kopplung zwischen den Teilsystemen minimiert

werden muss. Dahinter verbirgt sich das NP-harte Problem, eine gemäß einer Zielfunktion optimale Zerlegung einer Menge von Elementen zu berechnen. Wir verwenden einen genetischen Algorithmus zur Berechnung einer guten Näherungslösung.

Unsere Ähnlichkeitsmaße werten eine Vielzahl von Beziehungen zwischen den Bauteilen des Systems aus. Als entscheidend erweist sich die Gewichtung dieser Beziehungen – so hat beispielsweise der Aufruf einer Hilfsfunktion eine völlig andere Bedeutung für die Zerlegung eines Systems als Aufrufe zwischen Funktionen, die Einzelschritte einer Anwendungsfunktionalität implementieren. Wir stellen daher der Ballungsanalyse eine Mustersuche voran, deren Ziel es ist, den Elementen und Beziehungen im Strukturmodell Rollen in der Architektur des Systems zuzuordnen. Diese Rollen gehen dann in Form von Gewichten in die Berechnung der Ähnlichkeitsmaße ein. Der dem Suchprozess zugrunde liegende Musterkatalog ergibt sich ebenso wie die Gewichtung der Beziehungen zwischen den Bauteilen des Systems aus der Betrachtung bewährter Architekturkonzepte, welche in Form von Architektur- bzw. Entwurfsmustern vorliegen: es zeigt sich, dass bestimmte Muster verstärkt innerhalb von Teilsystemen auftreten, wohingegen andere Muster auf Teilsystemgrenzen hinweisen.

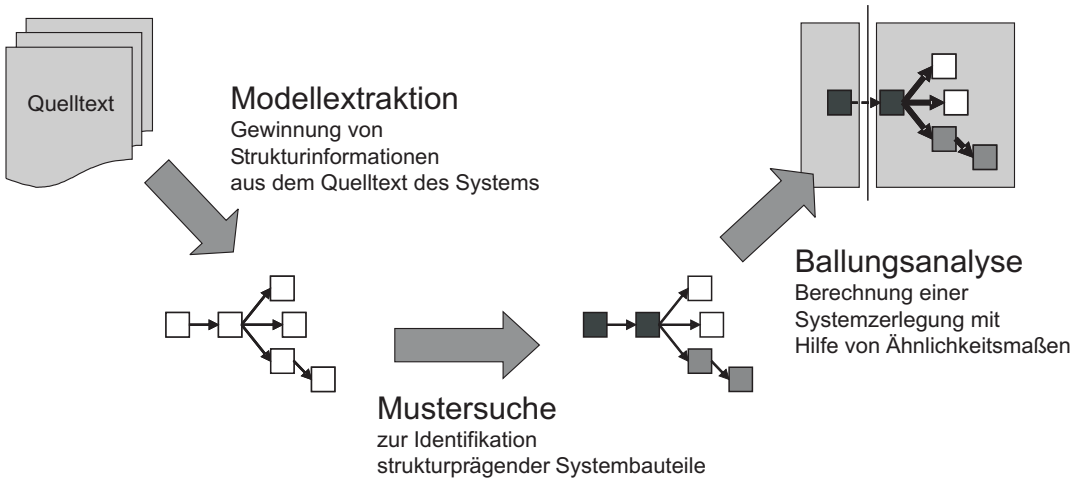


Abbildung 1.1: Systemzerlegung durch Mustersuche und Ballunsganalyse

Das Werkzeug *ACT* implementiert unser hybrides Verfahren zur Extraktion von Systemstrukturen für Systeme, die in der Programmiersprache *Java* implementiert sind. Mit *ACT* kann die Tauglichkeit des Verfahrens anhand von Fallstudien belegt werden.

Das hybride Verfahren macht die werkzeuggestützte Extraktion von Teilsystemstrukturen praktikabel. Es kann auch große Softwaresysteme mit weit mehr als hunderttausend Zeilen Quellcode vollständig in Teilsysteme zerlegen. Die Qualität der durch das Verfahren berechneten Teilsysteme übertrifft die gängiger Verfahren deutlich. Unser Verfahren ist erstmalig in der Lage, weit verbreitete Standardstrukturen in Softwarearchitekturen

– wie beispielsweise geschichtete Architekturen, Client-Server-Architekturen, Rahmenwerke und Bibliotheken – zu berücksichtigen und sinnvollen Teilsystemen zuzuordnen.

1.4 Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich wie folgt: Kapitel 2 definiert, was wir unter der Struktur von Softwaresystemen verstehen wollen. Es untersucht die Bedeutung der Systemstruktur für die Entwicklung, Pflege und Weiterentwicklung von Softwaresystemen und analysiert, warum Strukturen im Laufe der Lebensdauer eines Softwaresystems degenerieren. Kapitel 3 stellt den für unsere Arbeit relevanten Stand der Technik zusammen und zeigt, dass das Ziel unserer Arbeit, Teilsystemstrukturen von Softwaresystemen aus ihrer Implementierung zu berechnen, bisher noch nicht in befriedigender Weise erreicht ist. Kapitel 4 beschreibt den Lösungsansatz dieser Arbeit im Überblick. Kapitel 5 zeigt, wie wir die Implementierungsstrukturen von Softwaresystem so modellieren können, dass wir daraus mit Hilfe der Verfahren zur Mustersuche aus Kapitel 6 und zur Ballungsanalyse aus Kapitel 7 Teilsystemstrukturen berechnen können. Kapitel 6 und 7 stellen den eigentlichen Kern unserer Arbeit dar. Hierin beschreiben wir, wie wir – ausgehend von einer Klassifikation der Bauteile des Systems – mit Hilfe zweier unterschiedlicher Algorithmen und unter Verwendung von Ähnlichkeitsmaßen Teilsysteme gewinnen können, die als Basis für das Verständnis, die Pflege, die Sanierung und die Weiterentwicklung des Systems herangezogen werden können. Die Tauglichkeit unseres Verfahrens weisen wir in Kapitel 8 anhand mehrerer Fallstudien nach. Kapitel 9 fasst die Ergebnisse unserer Arbeit zusammen und gibt einen Ausblick auf Erweiterungsmöglichkeiten und weiterführende Fragestellungen.

Kapitel 2

Softwaresysteme und ihre Struktur

In diesem Kapitel legen wir die begrifflichen Grundlagen, auf denen die folgenden Kapitel aufbauen. Wir legen dazu zunächst fest, was wir unter der Struktur von Softwaresystemen verstehen wollen. Wir untersuchen dann, wie sich die Struktur von Softwaresystemen im Laufe ihrer Lebensdauer verändert und welchen Einfluss die Struktur auf die Wartung und Weiterentwicklung von Softwaresystemen hat. Wir leiten daraus den Bedarf nach Verfahren zur Gewinnung von verständlichen Teilsystemstrukturen ab und geben Kriterien an, die solche Verfahren erfüllen müssen, um praktikabel zu sein.

2.1 Systeme und Teilsysteme

Wir definieren zunächst die für unsere Arbeit zentralen Begriffe des Softwaresystems und seiner Struktur: Ein *System* ist eine Menge von *Gegenständen*, die über *Beziehungen* in einem inneren Zusammenhang stehen. Die Gegenstände bezeichnen wir auch als *Bausteine* des Systems oder *Systemkomponenten* (GOOS, 1997).

Ein System, dessen Bausteine aus Software bestehen, nennen wir *Softwaresystem* (BALZERT, 1996). Die *statische Struktur*[1] eines Softwaresystems ist gegeben durch die Menge aller statisch beobachtbaren Bausteine und der Beziehungen zwischen diesen (CIUPKE, 2001).

2.1.1 Objektorientierte Softwaresysteme

Wir interessieren uns insbesondere für Strukturen von *objektorientierten* Softwaresystemen. Wir folgen der Begriffsbildung von GOOS (1996) und verstehen darunter Softwaresysteme, die aus einer Menge miteinander *kooperierender Objekte* bestehen. Jedes Objekt repräsentiert dabei einen beliebigen Gegenstand aus einem für das Softwaresystem relevanten Ausschnitt der (realen oder gedachten) Welt, der Anwendungsdomäne. Ein solches Objekt lässt sich durch *Merkmale* charakterisieren. Dies können durch Werte

[1] Softwaresysteme haben auch eine dynamische Struktur, diese ergibt sich aus den während der Ausführung des Systems beobachtbaren Bestandteilen und ihrer Beziehungen. Wir beschränken uns hier auf diejenigen Bestandteile eines Systems, die während seiner Konstruktion beobachtbar sind.

darstellbare Eigenschaften sein, sogenannte *Attribute*, oder Tätigkeiten, die das Objekt ausführen kann. Diese Tätigkeiten werden in Form von Operationen, den *Methoden*, spezifiziert, die dem entsprechenden Objekt zugeordnet werden. Objekte mit gleichen Eigenschaften und gleichem Verhalten (also gleichen Merkmalen) gehören zu einer *Klasse*.

Eine wesentliche Aufgabenstellung während des Entwurfs eines objektorientierten Systems besteht darin, aus Anwendungsfällen, die das gewünschte Verhalten des Systems beschreiben, geeignete Objekte und Klassen abzuleiten.

Dazu sind folgende Fragestellungen von Interesse (BUDD, 2002):

- Welche Objekte bzw. Klassen gibt es? Welche Merkmale (in Form von Attributen und Methoden) besitzen diese?
- Wie stehen die Objekte untereinander in Beziehung? Lassen sich Objekte bzw. deren Klassen voneinander ableiten, beispielsweise weil zwischen ihnen eine *ist-ein*-Beziehung vorliegt?
- Welche Tätigkeiten führen die Objekte aus? Wie wirken die Objekte zusammen?

Aus der Beantwortung dieser Fragen ergibt sich die Struktur objektorientierter Systeme auf Klassenebene. Von besonderer Relevanz ist hierbei das *Objektmodell* eines Systems. Es beschreibt die *statische Struktur* der Objekte bzw. Klassen eines Systems und die Beziehungen zwischen diesen.[2] Natürlich müssen wir während des Entwurfs eines Systems auch das Verhalten des Systems angeben. Dies ist Aufgabe des *Verhaltensmodells*. Dieses ist allerdings für diese Arbeit von weit geringerer Bedeutung.

Wir bezeichnen die Entwurfstätigkeit, die die Struktur eines objektorientierten Systems hervorbringt, auch als *Programmieren im Großen*. Das *Programmieren im Kleinen* dagegen beschäftigt sich mit dem Ausformulieren der einzelnen Methoden des Systems.[3]

2.1.2 Teilsysteme als Grundlage für die Konstruktion großer Systeme

Zur Konstruktion großer objektorientierter Systeme, die aus tausenden von Klassen und mehreren hunderttausend Zeilen Quelltext bestehen, benötigen wir weitergehende Konzepte zur Strukturierung. Wir setzen die Systeme dazu aus *Teilsystemen* zusammen. Teilsysteme sind solche Bausteine eines Systems, die ihrerseits wieder Systeme sind.

[2] In den letzten Jahren haben sich zur Darstellung von Objektmodellen die *Klassendiagramme* der UML durchgesetzt (ÖSTERREICH, 1998). Wir werden in dieser Arbeit von diesen Diagrammen reichlich Gebrauch machen.

[3] Die Begriffe *Programming-in-the-Large* bzw. *Programming-in-the-Small* gehen auf DEREMER und KRON (1976) zurück, die diese Begriffe zunächst für das *strukturierte Programmieren* geprägt haben. GOOS (1996) und andere übertragen diesen Begriff jedoch auf naheliegende Weise auf die objektorientierte Programmierung.

Der Teilsystembegriff ist somit *rekursiv*, dass heißt, Teilsysteme können ihrerseits wieder Teilsysteme (mit geringerem Umfang) enthalten. Teilsysteme verfügen über eine *Schnittstelle*, die sich aus den Beziehungen zu anderen Teilsystemen des Systems ergibt. Wir präzisieren den Begriff des Teilsystems und seiner Schnittstelle in Abschnitt 2.1.3.

Die Tätigkeit, ein System in Teilsysteme zu zerlegen, nennt man *Modularisierung* oder *Dekomposition*. Modularisierung ist ein grundlegendes Konstruktionsprinzip, welches für alle Ingenieursdisziplinen von Bedeutung ist.

In der Konstruktion von Softwaresystemen verbinden wir mit einer geeigneten Modularisierung eine Reihe von Zielen (PARNAS, 1972; GOOS, 1996; BASS et al., 1998):

- *Übersichtlichkeit des Systementwurfs:* Die Zerlegung eines Systems in Teilsysteme erleichtert es uns, die Komplexität eines Systems zu beherrschen. Um ein komplexes System (oder ein Teilsystem davon) zu verstehen, genügt uns zunächst einmal ein globales Verständnis über das Zusammenwirken der Teilsysteme auf "oberster" Ebene. Innerhalb der Teilsysteme ist dann zusätzlich lediglich ein lokales Verständnis über das Zusammenwirken der darin enthaltenen Bauteile erforderlich.

- *Arbeitsteilung bei der Entwicklung:* Verschiedene Entwicklungsteams können unabhängig von einander einzelne Teilsysteme erstellen, wenn vorgegeben ist, wie die einzelnen Teilsysteme zusammenarbeiten sollen. Dies erfordert, dass die Schnittstellen der Teilsysteme festgelegt sind. Solche Teilsysteme fördern zudem die Testbarkeit des Systems und Lokalisierung von Fehlern.

- *Kapseln von Entwurfsentscheidungen:* Wenn Entwurfsentscheidungen lokal, also aus der Innensicht eines Teilsystems heraus, gefällt und umgesetzt werden können, so lassen sich diese zu späteren Zeitpunkten revidieren, ohne dass andere Teile des Systems modifiziert werden müssen. Systeme, die Entwurfsentscheidungen gezielt in Teilsystemen kapseln, lassen sich leichter weiterentwickeln und flexibel an neue Anforderungen anpassen.

- *Wiederverwendung:* Wir können Teilsysteme nutzen, indem wir wiederkehrende Aufgaben so abgrenzen, dass sie in verschiedenen Einsatzkontexten verwendet werden können. Diese Vorgehensweise vermeidet aufwendige Neuentwürfe und Neuimplementierungen.

- *Verteilter Betrieb, Flexibilität bei Auslieferung und Installation:* Zusätzlich zu den bisher aufgeführten Zielen, deren Perspektive in der Konstruktion von Systemen besteht, gibt es weitergehende Ziele. Eine geeignete Zerlegung von Systemen erleichtert auch die Installation und den Betrieb von Systemen, da Abhängigkeiten zwischen Systemteilen nur im Rahmen definierter Schnittstellen bestehen. Auf diese Weise lassen sich Systeme leichter verteilt betreiben oder Teilsysteme unabhängig voneinander ausliefern und installieren.

2.1.3 Eigenschaften von Teilsystemen

Wenn wir die Ziele, die wir mit der Modularisierung von Softwaresystemen verbinden, erreichen wollen, müssen die Teilsysteme bestimmte Eigenschaften aufweisen. Wir wollen diese im Folgenden etwas genauer beleuchten.

Schnittstellen und Geheimnisprinzip. Zunächst fordern wir, dass es zwischen den Teilsystemen klare, definierte Grenzen gibt, so dass Teilsysteme möglichst unabhängig voneinander betrachtet werden können. Aus diesem Grund kommt der bereits erwähnten *Schnittstelle* eines Teilsystems eine große Bedeutung zu.

Die Schnittstelle stellt die Außenansicht eines Teilsystems dar. Sie verbirgt das Innenleben des Teilsystems, die *Implementierung*, vor der Außenwelt *(Geheimnisprinzip)*.[4] Die Schnittstelle ist somit die einzige Information, die der Außenwelt über das Teilsystem bekannt ist. Die Einzelheiten der Implementierung des Teilsystems müssen durch die Schnittstelle so abgeschirmt sein, dass die Außenwelt weder beabsichtigt noch unbeabsichtigt die korrekte Arbeitsweise des Teilsystems stören kann. Umgekehrt ist die Schnittstelle die einzige Information, die dem Entwickler eines Teilsystems über dessen zukünftigen Einsatz bekannt ist.

Die Schnittstelle eines Teilsystems definiert eine Menge von *Operationen*. Dies sind die Dienstleistungen, die das Teilsystem seiner Außenwelt anbietet und die von anderen Teilsystemen, den Dienstnehmern, in Anspruch genommen werden können. Die Operationen einer Schnittstelle haben die Form $op(P_1, ..., P_n) : R$ und werden durch ihren Namen op, durch die Typen ihrer Parameter $P_1, ..., P_n$ und ihres Rückgabewertes R eindeutig identifiziert. Diese Informationen sind jedoch für eine sinnvolle Verwendung des Teilsystems noch nicht ausreichend. Deshalb gehen die Operationen mit Dienstnehmern *Verträge* ein: sofern der Dienstnehmer eine bestimmte Vorbedingung erfüllt, garantiert die Implementierung der Operationen im Innern des Teilsystems eine bestimmte Nachbedingung (MEYER, 1997). Diese Verträge regeln auch das Zusammenspiel der verschiedenen Operationen eines Teilsystems; sie geben dadurch auch die erlaubten Ausführungsfolgen der Operationen an der Schnittstelle eines Teilsystems, das *Schnittstellenprotokoll*, vor.

In objektorientierten Systemen ist die Schnittstelle eines Teilsystems häufig in Form einer ausgezeichneten Klasse, der *Fassade*, gegeben (GAMMA et al., 1995). Die Methoden dieser Klasse implementieren die über die Schnittstelle angebotenen Dienstleistungen, indem sie sich auf Klassen aus dem Inneren des Teilsystems abstützen.

Kohäsion und Kopplung. Damit Teilsysteme ihren Beitrag zur Erhöhung der Übersichtlichkeit eines Systems leisten können, müssen die Dienstleistungen in Form der Operationen, die ein einzelnes Teilsystem an seiner Schnittstelle anbietet, ein logisch

[4] Wir beachten in diesem Zusammenhang, dass die Außenwelt eines Teilsystems alle anderen Teilsysteme, die im Rahmen ihrer Implementierung eine Beziehung zu dem Teilsystem eingehen, einschließt.

zusammenhängendes und vollständiges Ganzes bilden. Um diese Forderung zu präzisieren, benötigen wir die Begriffe *Kohäsion* und *Kopplung*.

Unter *Kohäsion* eines Teilsystems (oder allgemeiner: eines Bausteins) verstehen wir den Grad, mit dem seine einzelnen Bestandteile dazu beitragen, dass es eine einzige, wohldefinierte Verantwortlichkeit erfüllt (COAD und YOURDON, 1991).

Eine logisch zusammenhängende Schnittstelle ist untrennbar mit einer hohen Kohäsion des betreffenden Teilsystems verknüpft: Gehören die an der Schnittstelle angebotenen Operationen logisch zusammen, so erwarten wir dies auch für die Bausteine im Inneren des Teilsystems, mit deren Hilfe die an der Schnittstelle angebotene Funktionalität implementiert wird. Auf Bausteine, die sich im Inneren des Teilsystems befinden und anderweitige Funktionalität bereitstellen, können wir aufgrund des Geheimnisprinzips ohnehin nicht sinnvoll zurückgreifen.

Unter der *Kopplung* eines Teilsystems (allgemeiner: eines Bausteins) verstehen wir den Grad seiner Vernetzung mit anderen Teilen des Systems (COAD und YOURDON, 1991).

Wenn die Gliederung eines Systems in Teilsysteme den Zielen aus Abschnitt 2.1.2 Rechnung tragen soll, so müssen wir fordern, dass die Kopplung zwischen den Teilsystemen möglichst gering ausfällt. Den Schnittstellen der Teilsysteme kommt hier besondere Bedeutung zu. Die durch die Schnittstelle eines Teilsystems angebotenen Operationen müssen in dem Sinne vollständig sein, dass die Funktionalität, die das Teilsystem der Außenwelt zur Verfügung stellt, weitestgehend ohne die Verwendung anderer Teilsysteme genutzt werden kann. Häufig müssen wir allerdings zur Implementierung eines Teilsystems auf die Dienste anderer Teilsysteme zurückgreifen. Dadurch erhöht sich die Kopplung des Teilsystems zu den übrigen Bestandteilen des Systems. Sein Einsatz setzt nun voraus, dass andere Teilsysteme zur Verfügung stehen, die an ihrer Schnittstelle eben diese Dienste in Form von Operationen anbieten.[5]

Teilsysteme, die eine starke Kopplung zu anderen Teilsystemen eines Systems aufweisen, lassen sich daher nur schwer isoliert betrachten und verstehen. Eine zu starke Kopplung erschwert zudem die unabhängige Entwicklung und die Wiederverwendung eines Teilsystems, sie verletzt im Extremfalls sogar das Geheimnisprinzip.

Komplexität. Wenn die Zerlegung eines Systems in Teilsysteme das Verständnis des Systems fördern soll, dann müssen die einzelnen Teilsysteme möglichst leicht zu verstehen sein. Neben der Kopplung zu anderen Systemteilen spielt dabei auch die innere *Komplexität* der einzelnen Teilsysteme eine Rolle. Ein Teilsystem ist dann komplex, wenn es aus sehr vielen verschiedenartigen Bausteinen besteht, und zwischen diesen Bausteinen wiederum viele verschiedenartige Beziehungen bestehen.

[5] Die Menge der externen Operationen aus anderen Teilsystemen, die das Teilsystem in seiner Implementierung voraussetzt, bildet die *Importschnittstelle* des Teilsystems (engl. *required interface*). Hinzu kommen noch die Typen der Parameter und Rückgabewerte der Schnittstellenoperationen, die ggf. ebenfalls der Importschnittstelle hinzugerechnet werden müssen.

Dieser Komplexitätsbegriff begründet sich aus der Erkenntnis, dass das menschliche Gehirn mit Strukturen, die zahlreiche, verschiedenartige miteinander verknüpfte Elemente enthalten, relativ schlecht zurechtkommt[6].

Dieser Erkenntnis müssen wir bei der Zerlegung eines Systems in Teilsysteme Rechnung tragen: *"Reducing complexity (...) must be the goal of every step in system specification, design and implementation"* (WIRTH, 1995). Wird die Komplexität eines Teilsystems zu hoch, ist eine weitere Zerlegung des Teilsystems in kleinere Teilsysteme erforderlich. Dadurch entsteht dann eine Hierarchie von Teilsystemen, wobei sich Teilsysteme auf höherer Ebene auf Teilsysteme niedrigerer Ebene abstützen (PARNAS, 1974, 1978).

2.2 Architekturen und Muster

Die wesentliche Aussage der voranstehenden Abschnitte besteht darin, dass wir die *Konstruktion* (im Sinne der Tätigkeit des Entwickelns) eines Systems durch eine geschickt gewählte Zerlegung in Teilsysteme erheblich vereinfachen können.

Neben der Vereinfachung der Konstruktion gibt es allerdings noch weitere Aspekte, die für die Struktur von Systemen von Belang sind. In vielen Systemen spielt beispielsweise die Zerlegung des Systems in unterschiedliche nebenläufige Ausführungseinheiten (*Ausführungsfäden*[7] oder *Prozesse*) und die damit verbundene Festlegung von Kommunikationsmechanismen eine Rolle.

Für alle Belange, die für ein Softwaresystem von Bedeutung sind, geeignete Strukturen (also Bestandteile mit Eigenschaften und Beziehungen zwischen diesen) zu definieren, ist Aufgabe der *Softwarearchitektur* (BASS et al., 1998; SHAW und GARLAN, 1996). Die Zerlegung eines Systems in Teilsysteme aus der Sicht des Entwurfs bzw. der Implementierung, wie wir sie in den vorangehenden Abschnitten beschrieben haben, ist somit Teil der Architektur eines Systems.[8]

Mit der Festlegung der Architektur eines Systems werden frühzeitig eine Reihe von wichtigen Entwurfsentscheidungen getroffen, die sich nur sehr schwer wieder revidieren lassen, gleichzeitig jedoch weitreichende Auswirkungen haben. Einige Beispiele mögen dies illustrieren:

- Die Architektur zwingt der Implementierung eines Softwaresystems gewisse Randbedingungen auf. Beispielsweise müssen sich alle Bauteile des Systems an eine durch die Architektur vorgegebene asynchrone Kommunikationsform oder an

[6] Die Kapazität des Kurzzeitgedächtnisses des Menschen reicht in der Regel für etwa sieben Elemente.

[7] engl. *threads*

[8] KRUCHTEN (1995) definiert beispielsweise ein Modell mit 4+1 Sichten auf die Architektur von Softwaresystemen. Er unterscheidet dabei die Sichten *Logical View*, *Development View*, *Process View* und *Physical View*, die sich allesamt aus Anwendungsfällen (*Scenario View*) entwickeln lassen. Unsere Strukturierung von Systemen in Klassen und Teilsysteme aus Abschnitt 2.1 deckt dabei die ersten beiden Sichten ab.

eine bestimmte *Middleware* halten. GARLAN et al. (1995) beschreiben zahlreiche weitere solcher Randbedingungen.

- Die Architektur bestimmt nicht nur die Struktur des zu konstruierenden Softwaresystems, sondern auch die Struktur der Entwicklungsmannschaft. Sie bestimmt, wie die Entwicklungsarbeit auf mehrere Teams verteilt werden kann oder in welcher Reihenfolge bestimmte Teile des Systems entwickelt werden müssen, wenn man frühzeitig vorführbare Prototypen des Systems haben möchte.
- Die Architektur beeinflusst nichtfunktionale Anforderungen. So hängen zahlreiche Qualitätseigenschaften, wie beispielsweise Performance, Erweiterbarkeit und dergleichen stark von der gewählten Architektur des Systems ab (BASS et al., 1998).

Um das Risiko von Fehlentscheidungen zur verringern, definiert man heute Architekturen für Softwaresysteme, indem man in möglichst großem Umfang auf bewährte Strukturen zurückgreift, deren Eigenschaften, Vor- und Nachteile gut erforscht und dokumentiert sind. Solche bewährten Strukturen sind häufig in Form so genannter *Muster* dokumentiert. Solche Muster sind ein wichtiges Instrument, die systematische Wiederverwendung von Entwurfswissen sicherzustellen. Aus diesem Grund stellt die Verwendung von Mustern heute eine Standardtechnik bei der Konstruktion von Systemen dar.

Muster prägen die Architektur (und damit die Modularisierung) von Systemen stark. Insbesondere lassen sich Architekturen und ggf. sogar komplette Systementwürfe systematisch über Muster herleiten (BECK und JOHNSON, 1994; GAMMA et al., 1995; RÜPING, 1997) beschreiben anhand mehrerer Fallbeispiele, wie man hierzu vorzugehen hat. Dabei wird deutlich, dass es Muster unterschiedlicher Granularität gibt, die sich in verschiedenen Phasen des Systementwurfs einsetzen lassen. Eine besondere prägende Rolle für die Architektur eines Systems – und damit auch für die Zerlegung in seine Teilsysteme – spielen Architekturmuster[9] (BUSCHMANN et al., 1996). Diese kommen daher schon in sehr frühen Phasen des Systementwurfs zum Einsatz.

Beispiel 1: ***Architekturmuster Schichtenarchitektur***

Eines der bekanntesten und am häufigsten eingesetzten Muster ist die *Schichtenarchitektur*[10] . In der Schichtenarchitektur werden die Bausteine eines Systems in mehreren Schichten angeordnet. Üblicherweise repräsentieren die Schichten ein gewisses Abstraktionsniveau ihrer Bauteile. In der Regel dürfen Bausteine in einer Schicht beliebig voneinander abhängen. Zwischen den Schichten gelten dann allerdings strenge Abhängigkeitsregeln. Es gibt unterschiedliche Ausprägungen des Musters mit jeweils eigenen Regelsätzen. Die beiden wichtigsten Ausprägungen sind die folgenden:

1. Bausteine in Schichten mit höherem Abstraktionsniveau dürfen Dienste in Anspruch nehmen, die über die Schnittstellen von Schichten mit niedrigerem Abstraktionsniveau angeboten werden. Bausteine aus niedrigeren Schichten dürfen dagegen nicht von Bauteilen in höheren Schichten abhängen *(strikte Schichtung)* .

[9] auch: *Architekturstile*
[10] engl. layered architecture

2. Bausteine einer Schicht dürfen nur auf die Dienste, die durch die nächstniedrigere Schicht angeboten werden, zurückgreifen – alle anderen Abhängigkeiten sind verboten. *(lineare Schichtung)*.

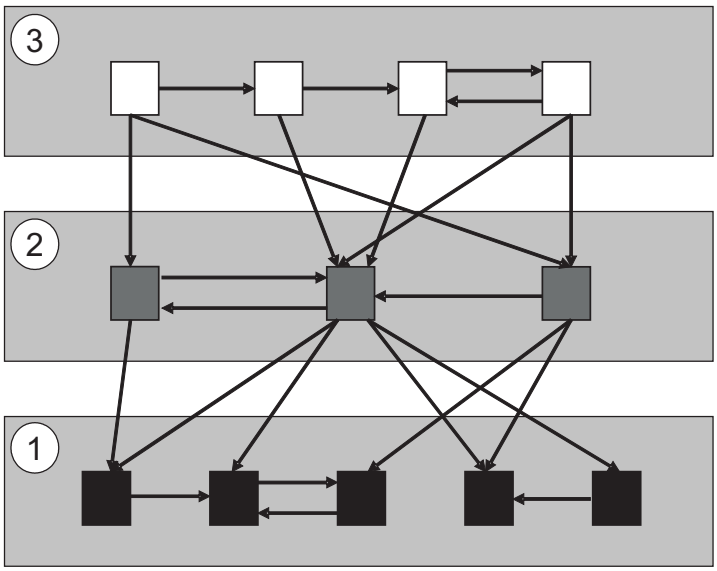


Abbildung 2.1: Schichtenarchitektur mit linearer Schichtung

Die Gliederung eines Systems in Schichten ist dann sinnvoll, wenn die Funktionalität eines Systems auf natürliche Weise unterschiedlichen Abstraktionsebenen zugeordnet werden kann. In diesem Fall definiert jede Schicht Betriebsmittel für die höheren Schichten. Falls diese Voraussetzung gegeben ist, bietet die Schichtenarchitektur folgende Vorteile:

- Die Gliederung des Systems in Schichten erhöht die Übersichtlichkeit des Systementwurfs. Das Prinzip der Gliederung kommt der Denkweise des Menschen entgegen und ist daher leicht verständlich und nachvollziehbar.
- Die Schichtenarchitektur erzwingt keine zu starke Einschränkung des Softwareingenieurs, da er neben der strengen Gliederung in Schichten noch weitgehende Strukturierungsmöglichkeiten innerhalb der einzelnen Schichten besitzt.
- Schichtenarchitekturen erhöhen neben der Übersichtlichkeit auch die Änderbarkeit, die Wartbarkeit, die Portabilität und die Testbarkeit von Systemen.

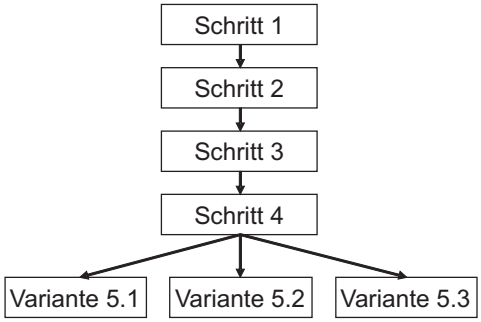
Den Vorteilen stehen aber auch Nachteile gegenüber:

- Schichtenarchitekturen können zu Performanceeinbußen führen, da Daten- und Steuerfluss über häufig verschiedene Schichten geführt werden müssen. Dies führt zu zahlreichen effizienzmindernden Indirektionen, insbesondere bei linearer Schichtung.
- Die einzelnen Schichten fallen in vielen Systemen recht umfangreich aus. Dies führt zu unübersichtlich strukturierten Systemen, sofern nicht weitere Prinzipien zur Strukturierung der Interna der einzelnen Schichten definiert werden.

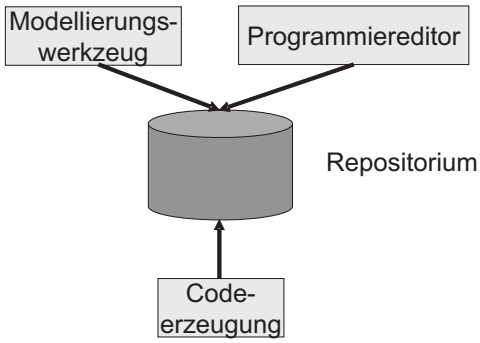
Schichtenarchitekturen werden sehr häufig für die Konstruktion betriebliche Informationssysteme (Unternehmenssoftware) verwendet. Solche Systeme bestehen in der Regel aus drei Schichten: Datenbank, Geschäfts- bzw. Anwendungslogik und Benutzerschnittstelle.

Eine ausführliche Darstellung des Musters findet sich in BALZERT (1996) oder BUSCHMANN et al. (1996).

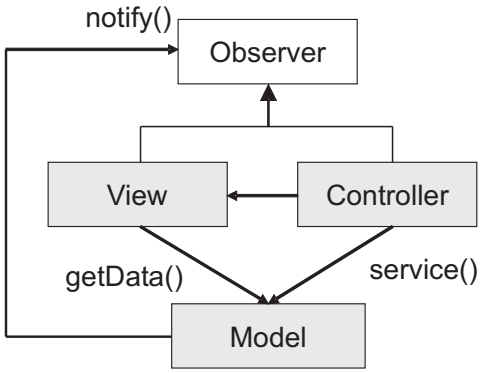
In Kapitel 4 gehen wir auf die strukturprägende Bedeutung von Architekturmustern noch genauer ein. Wir beschließen daher diesen Abschnitt mit einer Zusammenstellung weiterer häufig verwendeter Architekturmuster (vgl. auch Abbildung 2.2):



(a) Fließbandarchitektur



(b) Repositorium



(c) Model–View–Controller

Abbildung 2.2: Einige häufig verwendete Architekturmuster

- *Fließbandarchitektur:* Die *Fließbandarchitektur* [11] kommt in Systemen zum Einsatz, die einen Datenstrom verarbeiten. Die einzelnen Verarbeitungsschritte werden in jeweils separaten Teilsystemen, den Filtern, implementiert. Durch Austau-

[11] engl. *pipes and filters*

schen einzelner Filter können so leicht unterschiedliche Varianten eines Systems erzeugt werden. Übersetzer weisen in der Regel eine Fließbandarchitektur auf.

- *Repositorium:* Das Architekturmuster *Repositorium* prägt einen datenzentrierten Aufbau von Softwaresystemen. Im Zentrum steht das Repositorium (beispielsweise in Form einer Datenbank) mit dem die einzelnen Teilsysteme interagieren. Die Teilsysteme sind nicht direkt miteinander verknüpft. Einige *CASE*[12]-Werkzeuge verwenden diese Architektur.

- *Client-Server:* Das Architekturmuster *Client-Server* dient zur Realisierung verteilter Systeme. Die Funktionalität der Anwendung wird auf einen oder mehrere Dienstanbieter, so genannte Server, verteilt, die ihre Dienste über eine Kommunikationsinfrastruktur einem oder mehreren Dienstnehmern (Clients) zur Verfügung stellen.

- *Makler:* Das Architekturmuster *Makler*[13] kann verwendet werden, wenn ein System in Form einer Menge verteilter, kooperierender Teilsysteme realisiert werden soll. Eine ausgezeichnete Komponente, der Makler, vermittelt zwischen Dienstanbietern (Servern) und Dienstnehmern (Clients). Das Muster *Makler* stellt eine Weiterentwicklung des Musters *Client-Server* dar, in der die Dienstnehmer keine Kenntnis mehr darüber besitzen müssen, welcher Dienstleister welche Funktionen anbietet. Verteilte Systeme, die auf *CORBA* oder *DCOM* beruhen, nutzen dieses Muster.

- *Model-View-Controller:* Das Architekturmuster *Model-View-Controller* zerlegt Anwendungen mit grafischer Benutzerschnittstelle in drei Teile: das Modell enthält die Kernfunktionalität und Datenstrukturen der Anwendung, Controller und View implementieren zusammen die Benutzerschnittstelle des Systems. Eine Variante davon ist die *Document-View-Architektur*, in der die Funktion des Controllers der View-Komponente zugeschlagen wird.

2.3 Logische Struktur und Implementierungsstruktur

In den vorangehenden Abschnitten haben wir verschiedenartige Bauteile eines Systems aufgeführt. Diese lassen sich zu einer einheitlichen Vorstellung über die statische Struktur eines objektorientierten Softwaresystems verdichten. Abbildung 2.3 gibt diese Sicht wieder.

Ein objektorientiertes Softwaresystem gliedert sich demzufolge in unterschiedliche Teilsysteme (in der Regel sogar in eine Hierarchie von Teilsystemen). Diese Teilsysteme

[12] engl. *Computer-Aided Software-Engineering*
[13] engl. *broker*

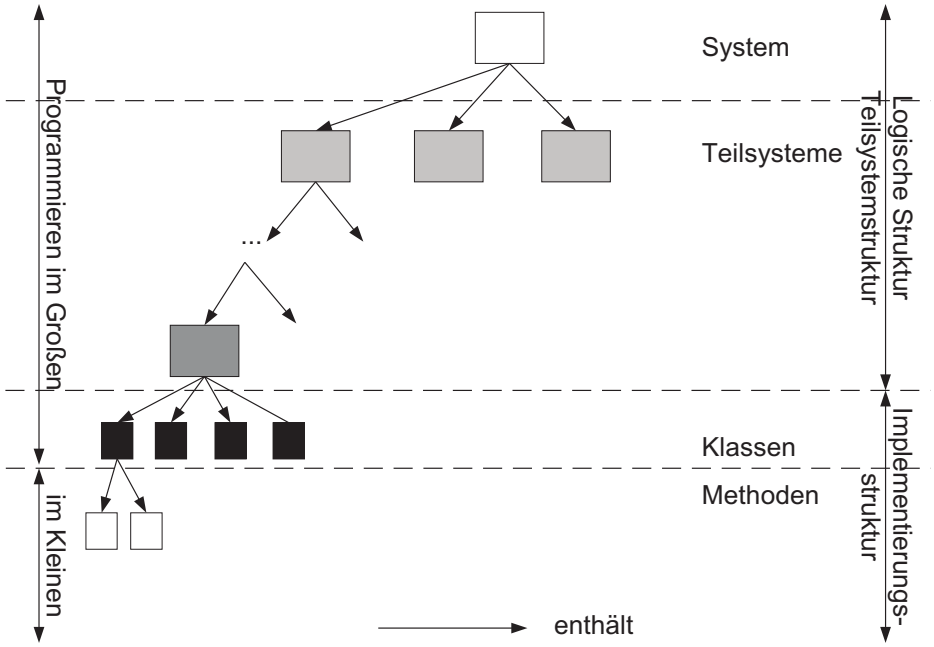


Abbildung 2.3: Struktur objektorientierter Systeme

enthalten Klassen, die die Konzepte der Anwendungswelt des Systems implementieren, ergänzt durch Klassen, die technische Konzepte des Systems implementieren. Die Zerlegung des Systems in Teilsysteme und Klassen ist Aufgabe einer Entwurfstätigkeit, die wir als *Programmieren im Großen* bezeichnet haben. Das Implementieren der Klassen mitsamt ihrer Merkmale (Attribute und Methoden) ist Aufgabe des *Programmierens im Kleinen*.

Abbildung 2.3 zeigt, dass wir die Struktur eines Systems auf unterschiedlichen *Abstraktionsebenen* betrachten können.

Objektorientierte Programmiersprachen unterstützen die Strukturierung von Systemen auf unterer Ebene, indem sie Sprachkonstrukte wie Klassen und Methoden anbieten. Die Struktur eines Systems auf der Ebene von Klassen und Methoden spiegelt sich in diesem Fall unmittelbar in seinem Quelltext wider. Diese Ebene bezeichnen wir daher als *Implementierungsstruktur*. Wir werden in Kapitel 5 noch präzise festlegen, welche Bausteine und Beziehungen Teil der Implementierungsstruktur sind.

Teilsysteme, wie wir sie in Abschnitt 2.1 eingeführt haben, führen zu einer Strukturierung des Systems auf höherer Abstraktionsebene. Die Struktur eines Softwaresystems auf der Ebene von Teilsystemen nennen wir *logische Struktur* oder *Teilsystemstruktur*.

Die Bildung von Teilsystemen im Sinne von Abschnitt 2.1 wird in der Regel nicht direkt von Programmiersprachen unterstützt. Die Teilsystemstruktur eines Systems lässt sich daher nicht direkt dem Quelltext von Systemen entnehmen.

Natürlich gibt es verschiedene Möglichkeiten, die Teilsystemstruktur in die Implementierungsstruktur zu übertragen. In vielen Systemen, die in der Programmiersprache *C++* implementiert sind, werden die Quelltexte der Implementierungen von Teilsystemen in separaten Verzeichnissen organisiert. In Java-Systemen kann entsprechend das Sprachkonstrukt `package` verwendet werden. Schnittstellen können – wie bereits dargestellt – mit Hilfe von Fassaden implementiert werden. Alle diese Ansätze basieren allerdings auf (projekt- oder firmenspezifischen) Konventionen, die vorschreiben, wie eine Teilsystemstruktur als Ergebnis des Entwurfs eines Systems in die Implementierung übertragen wird, es handelt sich dabei nicht um standardisierte Vorgehensweisen.

2.4 Die Rolle der Struktur für die Evolution von Softwaresystemen

Softwaresystemen stellen heute das Rückrat vieler Unternehmen und Organisationen dar. Sie verkörpern wertvolles Wissen über das Anwendungsfeld, in dem sie eingesetzt werden. Aus diesem Grund sind Softwaresysteme in aller Regel über lange Jahre hinweg im Einsatz.

Die lange Lebensdauer der Softwaresysteme führt dazu, dass diese Softwaresysteme kontinuierlich angepasst und erweitert werden müssen, um den wachsenden Bedürfnissen ihrer Anwender und veränderten Rahmenbedingungen, die sich aus dem Umfeld ergeben, in dem die Systeme arbeiten, gerecht zu werden. Die Arbeiten, die hierfür notwendig sind, fassen wir unter dem Begriff *Softwareevolution* zusammen. Diese umfasst neben Tätigkeiten wie Fehlerbereinigungen, die der Softwarewartung zuzuschreiben sind, auch umfangreichere Entwicklungsaufgaben. In der Regel müssen während der Lebensdauer eines Softwaresystems umfangreichere Ergänzungen und Weiterentwicklungen durchgeführt werden, um das System an neue Anforderungen anzupassen. Häufig verschwimmt die Grenze zwischen der Entwicklung eines Systems und seiner Betriebs- bzw. Wartungsphase, weil der Aufwand für die Entwicklungsarbeiten, die nach der ersten Auslieferung eines Systems stattfinden, den Aufwand für die initiale Entwicklung bei weitem übertrifft.

Die Struktur eines Softwaresystems beeinflusst die Kosten, die bei seiner Pflege und Weiterentwicklung anfallen, in erheblichem Ausmaß (CIUPKE, 2001). Aus diesem Grund besteht die Herausforderung bei der Entwicklung großer Softwaresysteme darin, geeignete Systemstrukturen zu finden, mit denen nicht nur die Erstentwicklung, sondern auch die Pflege und die Weiterentwicklung eines Systems erleichtert werden kann. Aus unseren Überlegungen aus Abschnitt 2.1.2 wird klar, dass eine Schlüsselrolle dabei der Architektur und der damit verbundenen Teilsystemstruktur eines Softwaresystems zukommt.

Leider finden wir in der Praxis zahlreiche Softwaresysteme, deren Struktur sich den mit der Evolution des Systems beauftragten Softwareingenieuren nicht mehr ausreichend

erschließt.

Die Gründe hierfür sind vielfältig:

- Bei der Konstruktion großer und komplexer Systeme entstehen fast nie auf Anhieb perfekte Entwürfe. Ingenieure machen Fehler und wählen weniger geeignete Lösungen für Konstruktionsprobleme, sei es durch mangelnde Kenntnisse und fehlende Erfahrung oder durch Versäumnisse. Fast alle technischen Konstruktionen können erst nach und nach perfektioniert werden. CIUPKE (2001) führt hier die Konstruktion von Autos als Beispiel an. Heutige Fahrzeuge haben sich seit der ersten Konstruktion von Carl Benz im Jahre 1886 erheblich weiterentwickelt. Dabei wurde die Konstruktion von Modellreihe zu Modellreihe verbessert, Konstruktionsfehler und Unzulänglichkeiten wurden beseitigt oder ab und zu auch ein paar neue eingeführt.

  Auch in Softwaresystemen können wir dies beobachten: In Softwaresystemen, die in den ersten Jahren nach der Einführung des objektorientierten Entwurfs entwickelt wurden, standen die Möglichkeiten zur Strukturierung der Systeme mit Hilfe von Klassen und der Einsatz von Vererbung im zentralen Blickpunkt der Softwareingenieure. Eine saubere Gliederung in Teilsysteme unterblieb häufig.

- Während der Weiterentwicklung werden selbst anfänglich einfache und gut verständliche Strukturen verwässert, weil sich die Ergänzungen nur schwer in die existierenden Strukturen einpassen lassen (PARNAS, 1994). Termin- und Kostendruck sowie mangelnde Kenntnisse über bereits existierenden Strukturen verschärfen dieses Phänomen noch, so dass eine eventuell erforderliche systematische Überarbeitung der Systemstruktur unterbleibt. Es ist zudem weder wirtschaftlich noch organisatorisch machbar, bei jeder Änderung die Struktur eines Systems im Ganzen neu zu überdenken. Dies ist erst dann sinnvoll, wenn feststeht, dass die Weiterentwicklung eines Systems eine andere Richtung nimmt, als dies bei der Definition der ursprünglichen Struktur geplant war.

- Häufig werden Architektur und Teilsystems eines Systems nur unzureichend dokumentiert. Selbst wenn die initiale Struktur hinreichend gut dokumentiert wird, so unterbleibt häufig die Anpassung der Dokumentation nach Modifikationen an der Struktur. Da Entwicklungsmannschaften der Fluktuation unterliegen, schwindet so nach und nach das Wissen über die Struktur eines Softwaresystems, welches über die unmittelbar vorhandenen Implementierungsstrukturen hinausgeht.

Damit die Weiterentwicklung der Systeme solcher Systeme praktikabel wird, sind Sanierungsarbeiten[14] erforderlich. Aufgabe der Sanierungsarbeiten ist es, die Struktur des Systems zu extrahieren, zu dokumentieren und danach so zu überarbeiten, dass diese eine solide Basis für die Weiterentwicklung des Systems darstellt.

[14] Wir verwenden den Begriff Softwaresanierung für das dem Englischen entstammende *software reengineering*

Einen wesentlichen Beitrag zu solchen Sanierungsarbeiten leisten Verfahren, die es erlauben, die logische Struktur eines Softwaresystems in Form der Teilsystemstrukturen aus den Implementierungsstrukturen zurück zu gewinnen. Ein solches Verfahren bildet den Gegenstand dieser Arbeit.

2.5 Anforderungen an Verfahren zur Strukturuntersuchung

Damit ein Verfahren zur Extraktion von Teilsystemstrukturen sinnvoll einsetzbar ist, muss es folgende Eigenschaften aufweisen:

- *Allgemeinheit:* Das Verfahren soll für beliebige objektorientierte Softwaresysteme einsetzbar sein. Dazu muss es unabhängig von Anwendungsdomäne, Implementierungs- und Infrastrukturtechniken sein, oder es muss sich zumindest einfach an solche Rahmenbedingungen anpassen lassen.

- *Skalierbarkeit:* Das Verfahren muss auch auf große Softwaresysteme mit mehr als einer Million Zeilen Quellcode und Tausenden von Klassen anwendbar sein, da gerade für solche Systeme Teilsystemstrukturen als zusätzliches Abstraktionsmittel von großem Nutzen sind.

- *Automatisierbarkeit:* Unmittelbar aus der Forderung nach Skalierbarkeit ergibt sich, dass das Verfahren automatisierbar sein muss. Für Systeme mit mehr als einer Million Zeilen Quellcode ist das manuelle Untersuchen des Quellcodes ohne Werkzeugunterstützung nicht mehr praktikabel.

- *Qualität der Analyseergebnisse*: Die Zerlegung des Systems, die ein solches Verfahren berechnet, soll den Softwareingenieuren, die mit der (Weiter-) Entwicklung des Systems betraut sind, wertvolle Hinweise auf den Aufbau des Systems geben. Dazu ist es erforderlich, dass die Teilsysteme in einer durch solche Verfahren berechneten Zerlegung des Systems den grundlegenden Zielen einer Modularisierung (siehe Abschnitt 2.1.2) Rechnung tragen.

  Die Qualität des Analyseergebnisses kann nur schwer formal bewertet werden. Einige Ersatzkriterien können bei einer Beurteilung von Analyseergebnissen hilfreich sein:

  – *Aussagekraft der Analyseergebnisse:* Möglich ist hierzu der Vergleich mit einer durch Experten erarbeiteten Referenzzerlegung des Systems. Dabei muss dann gelten, dass Bauteile, die für den Experten semantisch zusammen gehören, auch gemeinsam in ein Teilsystem aufgenommen werden. Umgekehrt dürfen Teilsysteme dann auch nur zusammengehörende Bauteile des Systems enthalten.

– *Vollständigkeit:* Alle Elemente, die in der Implementierung eines Systems auftauchen, sollen Strukturen (möglichst: Teilsystemen) zugeordnet werden.

Wir verwenden diese Kriterien im folgenden sowohl zur Bewertung existierender Ansätze des Standes der Technik als auch zur Bewertung unseres eigenen Verfahrens.

Kapitel 3

Stand der Technik

In diesem Kapitel stellen wir bereits existierende Verfahren vor, mit Hilfe derer die Struktur von Softwaresystemen untersucht werden können. Unser Augenmerk legen wir dabei auf Verfahren, die es uns erlauben, aus den *Implementierungsstrukturen* von Softwaresystemen Rückschlüsse auf ihre *logische Struktur* in Form von Teilsystemen zu schließen. Wir bewerten diese Verfahren hinsichtlich der Erfolgskriterien aus Abschnitt 2.5.

Gemeinsam ist allen Verfahren, dass sie versuchen, die Schwemme an unzureichend strukturierten Informationen, welche die Quelltexte von Softwaresystemen in sich tragen, zu filtern und zu sinnvollen Abstraktionen anzuordnen. Prinzipiell lassen sich die bereits existierenden Ansätze zur Strukturuntersuchung von Softwaresystemen vier Gruppen zuordnen:

1. *Strukturuntersuchung mit Softwaremaßen:* Mit Hilfe von Softwaremaßen werden aus den Implementierungsstrukturen eines Softwaresystems quantitative Werte abgeleitet, mit Hilfe derer sowohl eine Klassifikation der Bauteile des Systems vorgenommen als auch die Qualität von Strukturen beurteilt werden.

2. *Verfahren zur Mustersuche:* Verfahren zur Mustersuche identifizieren in den Implementierungsstrukturen eines Softwaresystems bestimmte zuvor spezifizierte Muster. Die per Mustersuche identifizierten logischen Strukturen können zum besseren Verständnis eines Softwaresystems und seiner Struktur beitragen.

3. *Ballungsanalyseverfahren:* Ansätze, die sich auf Ballungsanalyseverfahren abstützen, versuchen mit Hilfe von Algorithmen aus dem *Data Mining* die Bauteile eines Softwaresystems in Gruppen anzuordnen, aus denen dann logische Strukturen in Form von Teilsystemen abgeleitet werden können.

4. *Sonstige Ansätze:* Hierunter fassen wir Ansätze, wie beispielsweise das Erstellen von Reflexionsmodellen, die externes Entwicklerwissen heranziehen, um zusammengehörende Bauteile eines Systems zu identifizieren. Diese Ansätze sind nur teilweise automatisierbar.

3.1 Softwaremaße

Softwaremaße[1] bilden definierte Fragmente einer Softwarestruktur auf einen (zumeist) numerischen Wert ab. Es existieren unzählige Arbeiten zur Definition von Softwaremaßen (McCabe, 1976; Fenton und Pfleeger, 1997; Lorenz und Kidd, 1994; Chidamber und Kemerer, 1994). Zumeist messen diese Maße unterschiedliche Ausprägungen von Komplexität, Kopplung oder Kohäsion für verschiedene Arten von Bauteilen eines Systems. Die Messergebnisse dienen in der Regel dazu, Aussagen über die Qualität der Implementierungsstrukturen eines Softwaresystems abzuleiten.

Im Vordergrund bei der Vermessung eines Softwaresystems mit Hilfe von Softwaremaßen steht daher eine Klassifikation seiner Bauteile auf der Ebene der Implementierungsstrukturen – beispielsweise hinsichtlich ihrer Komplexität oder davon abgeleitet hinsichtlich ihrer Qualität.

Marinescu (2001, 2002) beschreibt darüberhinausgehend, wie Softwaremaße systematisch verwendet werden können, um strukturelle Mängel in der Implementierungsstruktur von Softwaresystemen aufzudecken.

Nur wenige Ansätze beschäftigen sich mit dem Gewinnen logischer Strukturen einer höheren Abstraktionsebene.

Bauer (1999a) demonstriert, wie Softwaremaße auch zur Verbesserung des Systemverständnisses herangezogen werden können, indem in der Implementierung eines Softwaresystems werkzeuggestützt Bauteile lokalisiert werden können, die Schlüsselkonzepte der Anwendungsdomäne darstellen. Dazu wird eine Kombination von Maßen verwendet: Klassen, die eine starke Kopplung zu anderen Teilen des Systems aufweisen und gleichzeitig hinreichend komplex sind, bilden Kandidaten für Klassen, die wesentliche Konzepte aus der Anwendungswelt eines Softwaresystems implementieren. Der Nachweis der Tragfähigkeit dieses Ansatzes erfolgt anhand mehrerer Fallstudien. Demeyer et al. (1999) liefern mit dem Werkzeug *CodeCrawler* eine geeignete Werkzeugunterstützung zur Anwendung des Verfahrens: Die Klassen eines Softwaresystems können graphisch in Form von Rechtecken als Knoten eines Abhängigkeits- oder Vererbungsgraphen dargestellt werden. Die Kantenlängen der Rechtecke sind dann proportional zum gewählten Kopplungs- und Komplexitätsmaß, so dass interessante Kandidaten unter den Klassen des Systems schnell identifiziert werden können.

Die auf diese Weise identifizierten Schlüsselkonzepte eines Systems können insbesondere für neue Mitglieder von Entwicklungsmannschaften eine wertvolle Hilfe für das Verständnis eines Softwaresystems darstellen. Die Identifikation von Schlüsselkonzepten führt allerdings nicht zu einer übersichtlichen und *vollständigen* Zerlegung eines

[1] Korrekt ist eigentlich der Begriff Softwareproduktmaß, der deutlich macht, dass es um die Vermessung von *Produkteigenschaften* geht. Wir verwenden hier den etwas kürzeren Begriff *Softwaremaß*. In der Literatur ist häufig auch der aus Sicht der Mathematik weniger treffende Begriff *Softwaremetrik* gebräuchlich.

Systems in Teilsysteme, wie sie für die Pflege und Weiterentwicklung eines Softwaresystems erforderlich ist.

SIMON (2001) entwickelt sogenannte Distanzmaße für die Bauteile eines Softwaresystems und nützt diese für eine Visualisierung der Implementierungsstrukturen von Softwaresystemen. Die Distanz zwischen je zwei Bauteilen eines Softwaresystems ergibt sich aus der Art und Anzahl der statischen Abhängigkeiten zwischen den Bauteilen. Mit Hilfe geeigneter Layoutalgorithmen lassen sich die Bauteile eines Systems in zwei- oder dreidimensionalen Visualisierungen so anordnen, dass Bauteile mit geringer Distanz und vielen Abhängigkeiten untereinander nahe beieinander platziert werden und Bauteile mit großer Distanz und wenigen Abhängigkeiten untereinander entsprechend entfernt liegen. Diese Vorgehensweise entspricht einer Art visueller Ballungsanalyse. Aus diesem Grund weist dieser Ansatz dieselben Probleme auf, wie die Verfahren, die wir in Abschnitt 3.3 diskutieren.

Die Verwendung von Softwaremaßen zur Untersuchung von Strukturen eines Softwaresystems wird durch eine Reihe von Problemen erschwert:

- *Schwierigkeiten bei der Interpretation der Messdaten:* In der Regel liefert die Vermessung eines Softwaresystems mit Hilfe von Produktmaßen eine Schwemme von Messdaten. Um diese zu interpretieren, ist es erforderlich, Schwellwerte zu definieren, mit Hilfe derer die Bauteile eines Systems klassifiziert werden können. So muss beispielsweise für ein Maß, welches die Komplexität einer Methode misst, ein Schwellwert festgelegt werden, ab der die Methode als zu komplex betrachtet werden soll. Die Kalibrierung der Schwellwerte erfordert erhebliche Erfahrung in der Anwendung von Softwaremaßen. Leider können nur eng verwandte Projekte mit Hilfe einheitlicher Schwellwerte untersucht werden, da die meisten Maße empfindlich von der verwendeten Programmiersprache, Implementierungstechnik, der Anwendungsdomäne oder gar der Systemgröße abhängen. Aus diesem Grund ist auch die Vergleichbarkeit von Messergebnissen unterschiedlicher Projekte nur schwer möglich.

  Diese Schwierigkeiten sind die Ursache dafür, dass Verfahren zur Untersuchung von Systemstrukturen, die sich ausschließlich auf Softwaremaße abstützen, nur dann für beliebige Systeme einsetzbar sind, wenn sie vorher aufwendig angepasst werden. Die Messung kann in der Regel werkzeuggestützt durchgeführt werden. Die Fülle der dabei entstehenden Messdaten führt allerdings ohne eine präzise Definition geeigneter Schwellwerte dazu, dass Vermessungen großer Softwaresysteme nur mit beträchtlichem Aufwand durchführbar sind. BAUER (1999a) und SIMON (2001) entschärfen diese Probleme etwas durch den Einsatz von Visualisierungstechniken. Die Interpretation der Visualisierungen ist allerdings vergleichsweise aufwendig und kaum automatisierbar.

- *Messergebnisse sind nicht konstruktiv:* Softwaremaße leisten – eine präzise Definition der Maße und der zugehörigen Schwellwerte vorausgesetzt – gute Dienste

bei der Untersuchung der Qualität der Implementierungsstruktur von Softwaresystemen. Sie bieten hingegen kaum Anhaltspunkte für die Rückgewinnung von Teilsystemstrukturen und Architekturen. Hierfür eignen sie sich erst, wenn sie mit Verfahren zur Ballungsanalyse kombiniert werden. Sie eignen sich allerdings – eine sinnvolle Definition der Maße vorausgesetzt – zur Begutachtung von Teilsystemstrukturen.

3.2 Strukturuntersuchung durch Mustersuche

Die Grundidee bei der Untersuchung von Softwarestrukturen durch Mustersuche[2] besteht darin, Informationen über die Struktur eines Softwaresystems in einer Datenbank abzulegen. Dieser Datenbestand kann dann mit verschiedenen Anfragemechanismen auf bestimmte, benutzerdefinierte Muster durchsucht werden. Aus dem Vorhandensein der Muster können dann Erkenntnisse über die logische Struktur von Softwaresystemen abgeleitet werden.

PRECHELT und KRÄMER (1996) beschreiben das Werkzeug *Pat*, mit dem Entwurfsmodelle von Softwaresystemen auf die Verwendung von Entwurfsmustern aus GAMMA et al. (1995) hin untersucht werden können. *Pat* übersetzt dazu zunächst Beschreibungen der Entwurfsmuster, die in Form von *OMT*-Klassengraphen spezifiziert werden können, in *Prolog*-Fakten. Danach übersetzt *Pat* Strukturinformationen aus dem Repositorium eines CASE-Werkzeug *Paradigm Plus* ebenfalls nach *Prolog*. Mit Hilfe einfacher *Prolog*-Programme können dann alle Vorkommen der zuvor spezifizierten Entwurfsmuster im Klassengraph des Systems identifiziert werden. *Pat* wurde im Rahmen von vier industriellen *C++*-Fallstudien erprobt und konnte die Entwurfsmuster *Adapter, Brücke, Kompositum, Dekorierer und Proxy* erkennen. Einen vergleichbaren Ansatz verfolgen SEEMANN und VON GUDENBERG (1998).

CIUPKE (1999, 2001) geht einen ähnlichen Weg, um Strukturprobleme in objektorientierten Systemen aufzudecken. Mit Hilfe von Techniken aus dem Übersetzerbau werden dabei im Rahmen einer Faktenextraktion Strukturinformationen über das zu untersuchende Softwaresystem aus dem *Java*- bzw. *C++*-Quelltext gewonnen und in einer Datenbank gespeichert. Diese Datenbank kann dann mit Mitteln der relationalen Algebra (oder mittels *Prolog*-Anfragen) auf Muster, die typische Entwurfsfehler repräsentieren, untersucht werden.

Obwohl die soeben beschriebenen Ansätze wertvolle Erkenntnisse über die Implementierungsstruktur eines Softwaresystems liefern können, eignen sie sich kaum für unsere eigentliche Problemstellung, nämlich logische Strukturen in Form von Teilsystemstrukturen zu identifizieren.

[2] engl.: *pattern matching*

Wir können zwar Muster auf der Ebene von Implementierungsstrukturen, wie sie beispielsweise in Form der Entwurfsmuster aus GAMMA et al. (1995) vorliegen, spezifizieren und werkzeuggestützt lokalisieren. Bei grobgranularen Mustern – beispielsweise bei Architekturmustern – versagt die Mustersuche jedoch. Die Ursachen hierfür sind die folgenden:

- *Variantenvielfalt bei der Umsetzung von Architekturen:* Es gibt zahllose Varianten bei der Umsetzung der Architekturstile bzw. Architekturmuster, so dass sich keine eindeutige, umkehrbare Abbildung der Musterbeschreibung auf Elemente der Implementierungsstruktur erstellen lässt. Eine solche Abbildung ist jedoch eine Voraussetzung für die bislang existierenden Verfahren zur Mustersuche.
- *Zerwartung von Strukturen:* Die ursprünglichen Strukturen, die bei der Umsetzung von Architekturstilen entstanden sind, "verwischen" durch unsachgemäße Wartung und Weiterentwicklung des Systems im Laufe der Lebensdauer eines Systems zunehmend. Auf diese Weise schleichen sich beispielsweise zusätzliche Bauteile und Abhängigkeiten ein, so dass eine Mustersuche die entsprechenden Strukturen nicht mehr zuverlässig identifizieren kann.

Ein wesentliches Problem ergibt sich zudem durch die prinzipielle Funktionsweise der Mustersuche. Sie identifiziert lediglich jene Bauteile, die unmittelbare Bestandteile vordefinierter Muster sind. Die meisten Systeme, die einem bestimmten Architekturstil folgen, enthalten zahllose weitere Bauteile, die sich nicht unmittelbar den einzelnen Elementen der Architektur zuordnen lassen. Zudem werden bei weitem nicht alle Systeme mit Hilfe von Architekturmustern konstruiert. Für existierende Systeme, deren logische Struktur keinem vordefinierten Architekturstil folgt, ist die nachträgliche Strukturierung mit Hilfe von Architekturmustern kaum möglich.

Deshalb sind musterbasierte Ansätze nicht in der Lage, *alle* Elemente eines Systems *sinnvollen* Strukturen zuzuordnen. Diese Verfahren liefern also in der Regel keine Ergebnisse, die unseren Anforderungen hinsichtlich der Qualität der Analyseergebnisse genügen.

3.3 Verfahren zur Ballungsanalyse

Eine immer wiederkehrende Aufgabe vieler Fachdisziplinen besteht in der Auswertung von Daten. Eine große Rolle spielen hierbei Verfahren zur *Ballungsanalyse*, die eine Menge von Daten so in Gruppen, die so genannten *Ballungen*, einteilen, dass die Elemente einer Gruppe gemäß geeigneter Kriterien als zusammengehörend betrachtet werden können.[3] Solche Verfahren sind weithin etabliert und es gibt hierzu unzählige Ansätze (siehe beispielsweise HARTIGAN (1975); KAUFMAN und ROUSSEEUW (1990)).

[3] Gebräuchlich sind auch die Begriffe *Clusteranalyse* respektive *Cluster*.

Zur Entscheidungsfindung, ob Elemente in ein- und dieselbe Gruppe platziert werden sollen, verwenden diese Verfahren sogenannte *Ähnlichkeitsmaße*.

Der Ansatzpunkt für die Verwendung von Ballungsanalyseverfahren zur Zerlegung von Softwaresystemen liegt in den Prinzipien zur Teilsystemkonstruktion begründet, die wir in Abschnitt 2.1.3 bereits beleuchtet haben: Teilsysteme enthalten zusammengehörende Bauteile, die in der Regel stark von einander abhängen. Abhängigkeiten zwischen Teilsystemen soll es dagegen möglichst wenige geben. Gelingt es, Ähnlichkeitsmaße so zu definieren, dass sie die Stärke der Abhängigkeiten zwischen den Bauteilen erfassen, so lassen sich Systemzerlegungen prinzipiell mit Hilfe gewöhnlicher Ballungsanalyseverfahren berechnen.

WIGGERTS (1997) untersucht als erster umfassend die Möglichkeit, Strukturen von Altsystemen mit Hilfe von Ballungsanalysen zu extrahieren. Dabei nimmt er eine Klassifikation existierender Algorithmen vor:

- *Hierarchische Algorithmen* berechnen eine Hierarchie von Zerlegungen, in der die jeweils nächsthöhere Hierarchiestufe aus der vorausgehenden hervorgeht, indem die zwei ähnlichsten Ballungen zu einer neuen, größeren Ballung verschmolzen werden. So entsteht ein binärer Baum aus Ballungen *(Dendrogramm)*, dessen Ebenen jeweils unterschiedlich feine Zerlegungen darstellen. Schneidet man das Dendrogramm horizontal in einer bestimmten Höhe, so erhält man eine Zerlegung des Systems, deren Granularität von der "Schnitthöhe" abhängt.

- *Graphtheoretische Algorithmen* arbeiten auf Graphen und berechnen Ballungen als Teilgraphen anhand graphtheoretischer Konzepte, wie beispielsweise Zusammenhangskomponenten oder minimale Spannbäume. Geeignete Graphen erhält man, indem man die Bauteile des Systems als Knoten verwendet und die Abhängigkeiten mit Hilfe gewichteter Kanten des Graphen modelliert. In Abschnitt 7.3 stellen wir einen solchen Algorithmus vor.

- *Optimierende Algorithmen* verwenden eine (gegebene oder zufällig erzeugte) initiale Zerlegung des Systems und versuchen, diese durch sukzessive Anpassungen zu verbessern, beispielsweise durch Verschieben von Einheiten bzw. Bauteilen zwischen Ballungen, so dass eine Zielfunktion, die mit Hilfe des Ähnlichkeitsmaßes gebildet werden kann, optimiert wird. Einen solchen Algorithmus stellen wir in Abschnitt 7.4 vor.

WIGGERTS (1997) weist darauf hin, dass die Wahl der Ähnlichkeitsmaße, auf Basis derer Algorithmen zur Ballungsanalyse zusammengehörende Elemente gruppieren, einen entscheidenden Einfluss auf das Ergebnis hat.

Die Arbeit von WIGGERTS (1997) kann als Katalysator für eine Reihe weiterer Arbeiten gelten, die unterschiedliche Ballungsanalyseverfahren zur Strukturierung von Softwaresystemen heranziehen.

Algorithmus 1 Hierarchische Ballungsanalyse *HAC*

Eingabe:
Menge $M = \{m_1, \ldots m_n\}$ mit m Elementen
Ähnlichkeitsmaß $\omega : M \times M \rightarrow \mathbf{R}$
Ausgabe: Dendrogramm aus Zerlegungen der Menge M
- - Jedes Element kommt in eine eigene Ballung
$C := \phi$
for $i := 1$ to n **do**
  $C_i := \{m_i\}$
  $C := C \cup \{C_i\}$
end for
- - Berechne Ähnlichkeit ω zwischen Ballungen
for $i := 1$ to n **do**
  for $j := 1$ to n **do**
    $\omega(C_i, C_j) := \omega(m_i, m_j)$
  end for
end for
repeat
  Füge C ins Dendrogramm ein
  Wähle die Ballungen $C_i, C_j \in C$, die einander am ähnlichsten sind
  - - Verschmelze C_i und C_j
  $R := C_i \cup C_j$
  - - Aktualisiere Ähnlichkeitsmaß ω
  for $k := 1$ to n **do**
    $\omega(R, C_k) := \max(\omega(C_i, C_k), \omega(C_j, C_k))$
  end for
  $C := (C \cup \{R\}) - \{C_i, C_j\}$
until $|C| = 1$

Wir stellen im folgenden eine Auswahl der wichtigsten dieser Arbeiten zusammen:

TZERPOS und HOLT (1998) demonstrieren das Potenzial von Ballungsanalysealgorithmen, indem sie unterschiedliche Algorithmen einsetzen, um die Quelltextdateien von in der Programmiersprache *C* geschriebenen Softwaresystemen anhand der *include*-Beziehungen zu Teilsystemen zu gruppieren.

KOSCHKE (2000) untersucht verschiedene Verfahren zur Extraktion von Strukturinformationen in Altsystemen, die mit Hilfe prozeduraler Programmiersprachen wie *C*, *COBOL* oder *Pascal* implementiert wurden. Davon ausgehend entwickelt er mehrere Verfahren zur Identifikation von abstrakten Datentypen in solchen Altsystemen. Diese Verfahren nützen graphbasierte und hierarchische Algorithmen zur Ballungsanalyse und fassen mit deren Hilfe beispielsweise Funktionen bzw. Prozeduren zu Modulen zusammen, wenn diese dieselben globalen Variablen nützen.

MANCORIDIS et al. (1998, 1999); MITCHELL (2002) entwickeln das Werkzeug *Bunch*, welches verschiedene Algorithmen zur Ballungsanalyse einsetzt, um Softwaresysteme in Teilsysteme zu zerlegen. Die Autoren setzen erstmalig einen genetischen Algorithmus (DOVAL et al., 1999) zur Zerlegung von Softwaresystemen ein. Die Fähigkeiten von *Bunch* werden mit Hilfe mehrerer Fallstudien (in *C* und *C++*) erprobt. Die Grundlage der Ballungsanalyse bilden – wie bei TZERPOS und HOLT (1998) – Quelltextdateien und ihre *include*-Beziehungen. Die Autoren berichten auch von der erfolgreichen Zerlegung objektorientierter Systeme auf Basis von statisch ermittelbaren Methodenaufrufen zwischen Klassen. Sie definieren dazu ein sehr einfaches, binäres Ähnlichkeitsmaß zwischen den Klassen des Systems. Dieses ist 1, falls zwischen den Methoden zweier Klassen eine Aufrufbeziehung besteht, andernfalls ist es 0.

RAYSIDE et al. (2000) und eine durch den Autor dieser Arbeit betreute Diplomarbeit von TRIFU (2001) spezifizieren erstmalig Ähnlichkeitsmaße, die weitergehende strukturelle Abhängigkeiten zwischen Klassen erfassen. Sie berücksichtigen bei der Bildung der Ähnlichkeitsmaße Vererbungsbeziehungen, Methodenaufrufe, Attributzugriffe und Typabhängigkeiten, wie sie durch die Deklaration von Variablen entstehen. Diese Abhängigkeiten gehen mit unterschiedlichem Gewicht in die Ähnlichkeitsmaße zwischen Klassen ein. RAYSIDE et al. (2000) verwendet für die Ballungsanalyse ein hierarchisches Verfahren, TRIFU (2001) experimentiert mit hierarchischen Verfahren und graphbasierten Verfahren. Eine Weiterentwicklung eines der graphbasierten Verfahren von TRIFU (2001) ist Gegenstand von Abschnitt 7.3.

E ABREU et al. (2000) erprobt diese Konzepte an einer Reihe von objektorientierten Fallstudien. Er zeigt, dass Ballungsanalysen in vielen Fällen Systemzerlegungen berechnen, die für den menschlichen Softwareingenieur nur schwer nachvollziehbar sind. Dies wird durch die bereits erwähnte Diplomarbeit von TRIFU (2001) bestätigt.

Obgleich es offensichtlich eine Vielzahl von Arbeiten gibt, die Ballungsanalyseverfahren zur Zerlegung von Softwaresystemen in Teilsystemstrukturen einsetzen, liefern diese Verfahren für unsere Problemstellung noch keine befriedigende Lösung:

- Nur eine überschaubare Menge von Arbeiten (RAYSIDE et al., 2000; TRIFU, 2001; MITCHELL, 2002) geht darauf ein, wie objektorientierte Systeme werkzeuggestützt in Teilsysteme zerlegt werden können. Die meisten anderen Arbeiten beschränken sich auf die Extraktion abstrakter Datentypen (durch die Analyse von Variablenzugriffen und Funktionsaufrufen) oder die Remodularisierung von *C*-Systemen (durch die Analyse von *include*-Abhängigkeiten).
- Alle Ballungsanalyseverfahren für objektorientierte Systeme versagen regelmäßig bei bestimmten Teilen der untersuchten Systeme. Besonders schlechte Ergebnisse liefern die Verfahren, wenn der zu untersuchende Quelltext Rahmenwerke, geschichtete Architekturen, Bibliotheken oder dergleichen enthält. Dies liegt daran, dass alle diese Arbeiten lediglich syntaktische Informationen über Bauteile und die Beziehungen zwischen diesen verwerten, wie sie direkt aus dem Quelltext der Systeme gewonnen werden können. Die Rollen, welche die Bauteile für die Architektur eines Systems spielen, bleiben unberücksichtigt. Wir vertiefen diese Überlegungen in Abschnitt 4.1.

Fazit: Die existierenden Ballungsanalyseverfahren erfüllen die meisten unserer Anforderungen, die wir in Abschnitt 2.5 an Verfahren zur Zerlegung von objektorientierten Systemen in Teilsysteme gestellt haben. Sie sind für beliebige Systeme verwendbar, skalieren für große Systeme und sind automatisierbar. Sie liefern eine vollständige Zerlegung der Systeme. Leider sind diese Zerlegungen häufig nicht geeignet, Softwareingenieure beim Verstehen der Struktur eines Systems und somit bei der Pflege und der Weiterentwicklung zu unterstützen.

3.4 Reflexionsmodelle

Neben den bislang betrachteten Verfahren gibt es noch weitere, die sich nicht ausschließlich auf den Quelltext bzw. die Analyse der Implementierungsstrukturen abstützen, sondern zusätzlich extern *vorhandenes Entwicklerwissen* zur Bildung von Teilsystemstrukturen heranziehen.

Wir wollen im folgenden ein solches Verfahren untersuchen, welches im Gegensatz zu vielen anderen Verfahren dieser Gruppe zumindest teilweise durch Werkzeuge unterstützt werden kann.

MURPHY et al. (2001) schlagen ein Verfahren vor, das sie Reflexionsmodelle[4] nennen. Mit Hilfe von Reflexionsmodellen sollen Softwareingenieure ihr Verständnis über den Aufbau eines Softwaresystems mit Hilfe von Strukturinformationen, die sie aus dem Quelltext des Systems erhalten, systematisch verbessern können. Das Verfahren funktioniert wie folgt (siehe dazu auch Abbildung 3.1):

[4] Die Erfinder des Verfahrens verwenden absichtlich diese Schreibweise, um ihr Verfahren von der Reflektion im Sinne der Introspektion in der objektorientierten Programmierung zu unterscheiden.

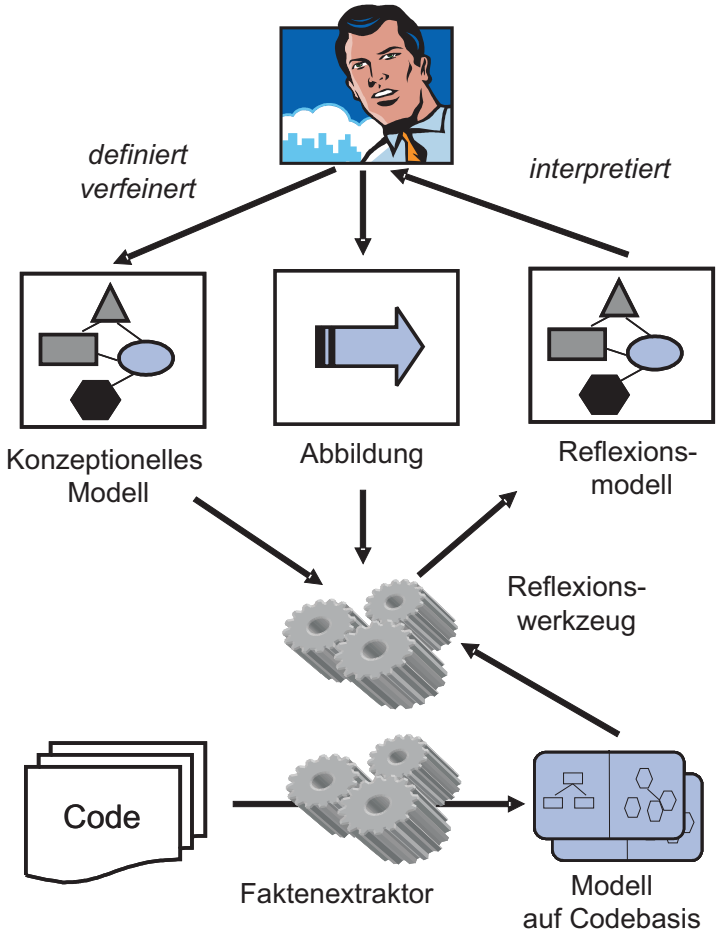


Abbildung 3.1: Arbeiten mit Reflexionsmodellen

Mit Hilfe eines Werkzeugs zur Faktenextraktion wird zunächst ein quelltextnahes Modell der Implementierungsstruktur des zu untersuchenden Systems gewonnen. Dabei kommen recht einfache Werkzeuge zur Querverweisextraktion (*Cross Referencing*) zum Einsatz, die beispielsweise die Funktionen des Systems und die Aufrufbeziehungen zwischen diesen extrahieren.

Unabhängig davon konstruiert ein Softwareingenieur auf Basis von Expertenwissen, Systemdokumentation und anderen Informationsquellen ein abstraktes, konzeptionelles Modell des zu analysierenden Systems, in dem er dessen Teilsysteme und die Abhängigkeiten zwischen diesen spezifiziert. Zudem definiert er Abbildungsregeln, mit deren Hilfe Elemente des quellcodenahen Modells Elementen des konzeptionellen Modells zugeordnet werden können. Das so genannte Reflexionswerkzeug erzeugt aus diesen Elementen ein Reflexionsmodell (Spiegelmodell). Dieses stellt die Diskrepanzen zwischen dem quelltextnahen Modell und dem konzeptionellen Modell dar. Dazu werden im konzeptionellen Modell Abhängigkeiten, die sich aus dem quelltextnahen Modell und den

Abbildungsregeln ergeben, ergänzt und markiert. Ebenso werden Abhängigkeiten markiert, die es nur im konzeptionellen Modell gibt.

Der Softwareingenieur kann nun das Reflexionsmodell interpretieren. Dabei analysiert er die hervorgehobenen Abhängigkeiten mit Hilfe des Quelltextes des Systems. Genügt das Reflexionsmodell seinen Ansprüchen noch nicht, verfeinert er das abstrakte Modell, die Abbildungsregeln und das konzeptionelle Modell entsprechend und erzeugt ein überarbeitetes Reflexionsmodell. Diese Schritte wiederholt er, bis das Reflexionsmodell detailliert genug ist und nur noch wenige Diskrepanzen zum quellcodenahen Modell aufweist. Endergebnis ist ein verfeinertes, mit Hilfe des Quelltextes überprüftes konzeptionelles Modell, welches zugleich eine Zerlegung des Systems in Teilsysteme darstellt.

Die Erfinder des Verfahrens berichten, dass sich ein Entwickler von Microsoft mit Hilfe von Reflexionsmodellen nach ca. vier Wochen Arbeit ein detailliertes Verständnis der inneren Struktur von *Excel* erarbeiten konnte.

Für unsere Problemstellung, ein Softwaresystem in Teilsysteme zu zergliedern, ergeben sich allerdings folgende Probleme:

- Die Ergebnisse des Verfahrens hängen stark vom vorhandenen Entwicklerwissen und der Qualität des konzeptionellen Modells ab. Unerfahrene Softwareingenieure können – falls keine ausführliche Dokumentation über das System zur Verfügung steht – nur mit hohem Zeitaufwand und unter großen Schwierigkeiten ein geeignetes konzeptionelles Modell des Systems erstellen.

- Das Erstellen der Abbildungsregeln zwischen Quelltext und konzeptionellem Modell ist sehr aufwendig und erfordert ein umfangreiches Studium des Quelltextes des Systems (bzw. der Elemente des quelltextnahen Modells). Aufgrund der Vielzahl von Abbildungsregeln, die für größere Systeme definiert werden müssen, skaliert das Verfahren schlecht.

- Die Ergebnisse des Verfahrens werden zunächst immer eine – in der Regel veraltete – Soll-Architektur des Systems widerspiegeln. Erst in späteren Phasen des Verfahrens kristalliert sich die tatsächlich vorhandene Ist-Architektur heraus. Je größer die Diskrepanz zwischen der veralteten Soll-Architektur und der tatsächlich vorhandenen Ist-Architektur des Systems ist, desto schwieriger und langwieriger gestaltet sich die Anwendung des Verfahrens.

- Die Grundprinzipien guter Architektur und der Aufbau von Systemen nach bewährten Mustern werden bei der Erstellung des konzeptionellen Modells bzw. der Zerlegung des Systems nicht explizit berücksichtigt.

3.5 Zusammenfassung

Wir haben in diesem Kapitel Verfahren betrachtet, die sich mit der Berechnung logischer Strukturen – in Form von Teilsystemstrukturen – von Softwaresystemen beschäftigen. Wir haben dabei festgestellt, dass keiner der beschriebenen Ansätze die Anforderungen, die wir in Kapitel 2.5 an solche Verfahren gestellt haben, befriedigen kann.

Keines der Verfahren kann für *beliebige*, *große* objektorientierte Softwaresysteme mit *vertretbarem Aufwand* eine Zerlegung in Teilsysteme berechnen, die Softwareingenieuren als *gute* Basis für das Verständnis, die Pflege und die Weiterentwicklung der Systeme dienen kann.

Softwaremaße können zwar dazu herangezogen werden, die Qualität bereits existierender Teilsysteme zu beurteilen. Sie können allerdings nur in Verbindung mit Visualisierungen bzw. Ballungsanalyseverfahren konstruktiv, also zur Berechnung bzw. Gewinnung von Teilsystemstrukturen, eingesetzt werden.

Verfahren zur *Mustersuche* eignen sich lediglich zur Identifikation von feingranularen Strukturen auf der Ebene der Implementierungsstruktur eines Softwaresystems. Zur Berechnung von logischen Strukturen, die einer vollständige Zerlegung von Softwaresystemen in Teilsysteme entsprechen, eignen sie sich nicht.

Verfahren zur *Ballungsanalyse* erfüllen weitestgehend unsere Anforderungen. In vielen Systemen sind die berechneten Teilsystemstrukturen allerdings für Softwareentwickler nur teilweise nachvollziehbar.

Ansätze, die sich wie die betrachteten *Reflexionsmodelle* neben dem Quellcode der Systeme noch auf weitere Informationsquellen abstützen, sind nur teilweise werkzeuggestützt durchführbar. Die Qualität der so bestimmten Teilsysteme hängt stark von den zur Verfügung stehenden Ressourcen in Form von Zeitaufwand und Entwicklerwissen ab.

Tabelle 3.1 fasst die Bewertung existierender Verfahren anhand unserer Anforderungen aus 2.5 nochmals zusammen.

Die Tabelle zeigt, dass einige der existierenden Ballungsanalyseverfahren die meisten Anforderungen recht gut erfüllen. Wir zeigen im Rahmen dieser Arbeit, wie sich diese Verfahren so ausbauen lassen, dass sie deutlich aussagekräftigere Zerlegungen von Systemen in Teilsysteme hervorbringen.

	Software-maße	**Mustersuche**	**Ballungs-analyse**	**Reflexions-modelle**
Skalierbarkeit	○	+	+	○
Automatisierbarkeit	○	+	+	−
Allgemeinheit	○	+	+	+
Qualität der Strukturen				
Vollständigkeit	−	−	+	○
Aussagekraft	○	○	○	○/+

Tabelle 3.1: Zusammenfassende Bewertung des Standes der Technik

Kapitel 4

Ein hybrider Ansatz zur Strukturextraktion

Wir stellen in diesem Kapitel den dieser Arbeit zugrundeliegenden Ansatz vor. Unser Ansatz kombiniert erstmalig die Stärken von musterbasierten Verfahren zur Architekturextraktion mit Techniken aus der Ballungsanalyse. Die grundlegende Idee besteht darin, der Ballungsanalyse eine Mustersuche voran zu stellen, die die Elemente des Systems und die Beziehungen zwischen diesen gemäß ihrer Bedeutung für die Architektur des Systems klassifiziert. Die Ballungsanalyse verwertet diese Klassifikation, indem sie die Abhängigkeiten zwischen den Elementen des Systems mit entsprechenden Gewichten versieht. Dadurch liefert die Ballungsanalyse Ergebnisse, die aus der Perspektive von Softwareingenieuren wesentlich bessere Zerlegungen des Systems in Teilsysteme darstellen.

4.1 Problemanalyse

Die im vorangehenden Kapitel 3 vorgestellten Ansätze, mit Hilfe von Ballungsanalyseverfahren Zerlegungen von Softwaresystemen in Teilsysteme vorzunehmen, liefern in vielen Fällen Ergebnisse, die für Softwareingenieure nur schwer nachvollziehbar sind (E ABREU et al., 2000).

TRIFU (2001) zeigt in einer durch den Autor dieser Arbeit betreuten Diplomarbeit, dass die Qualität der Systemzerlegungen, die mit Hilfe von Verfahren zur Ballungsanalyse berechnet werden können, stark davon abhängt, wie die Ähnlichkeitsmaße, anhand derer die einzelnen Bausteile des Systems zu Teilsystemkandidaten zusammengestellt werden, definiert werden. Wie in Abschnitt 3.3 dargestellt, berechnen sich diese Ähnlichkeitsmaße in der Regel über Softwaremaße, die die statischen Abhängigkeiten zwischen den Klassen des Systems messen. So verwendet MITCHELL (2002) hierfür beispielsweise ein binäres Maß zwischen je zwei Klassen, welches 1 ist, falls die eine Klasse eine Methode enthält, in deren Implementierung eine Methode der anderen Klasse aufgerufen wird. RAYSIDE et al. (2000) bzw. TRIFU (2001) dagegen verwenden Verhältnismaße, welche Aufrufe zwischen den Methoden der Klasse, Vererbungsbeziehungen, und Variablennutzungen quantitativ berücksichtigen.

Wir untersuchen anhand einer Struktur, wie sie in Systemen häufig vorkommt, warum Ballungsanalyseverfahren, die auf solchen Ähnlichkeitsmaßen basieren, häufig zu unbefriedigenden Systemzerlegungen führen.

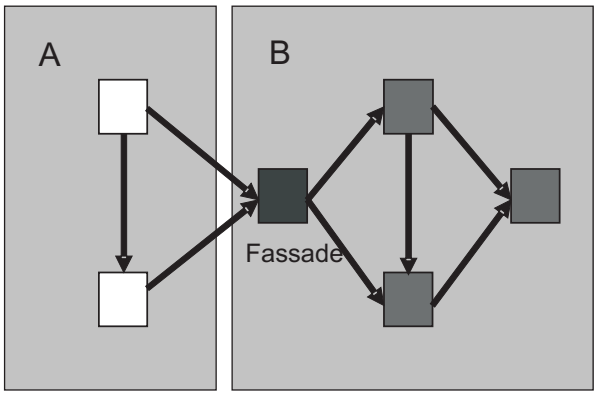


Abbildung 4.1: Abhängigkeitsgraph für ein Teilsystem mit Fassade

Abbildung 4.1 zeigt zwei Teilsysteme, *A* und *B*. Die Klassen des Teilsystems *A* verwenden Dienste, die die Klassen in Teilsystem *B* über eine Fassadenklasse der Außenwelt anbieten. Gemäß des Geheimnisprinzips (siehe Abschnitt 2.1.3) können Dienste, die das Teilsystem *B* anbietet, *ausschließlich* über die Schnittstelle von *B*, also über die Fassadenklasse, in Anspruch genommen werden. Durch diese Dienstnutzung ergeben sich eine Vielzahl von Abhängigkeiten zwischen Dienstnutzern und der Fassade. Bei der Berechnung einer Systemzerlegung durch eine Ballungsanalyse wird aufgrund dieser Abhängigkeiten eine solche Fassadenklasse häufig den nutzenden Teilsystemen (hier dem Teilsystem *A*) zugeordnet werden, anstatt dem Teilsystem, für das sie die Schnittstelle bildet.

Fallstudien von E ABREU et al. (2000) und TRIFU (2001) sowie eigene Beobachtungen (siehe Kapitel 8) zeigen, dass Effekte dieser Art die Qualität der durch Ballungsanalyseverfahren berechneten Systemzerlegungen stark beeinträchtigen. Insbesondere betrifft dies Systeme, die geschichtete Architekturen (siehe 2.2), Bibliotheken (bzw. Teilsysteme, die technische Funktionen implementieren) oder Rahmenwerke enthalten. In allen diesen Fällen führt die regelkonforme Verwendung dieser Strukturen zu einer Vielzahl von Abhängigkeiten, die den Einsatz von Ballungsanalysen zur Zerlegung von Systemen gemäß der in Kapitel 2 dargestellten Prinzipien zur Systemkonstruktion erschweren.

In streng linear geschichteten Architekturen (siehe Seite 14) entstehen beispielsweise zahlreiche Abhängigkeiten zwischen je zwei Schichten, weil die Dienste der jeweils höheren Schicht auf Basis der an der Schnittstelle der nächstniedrigen Schicht angebotenen Dienste implementiert werden. Im Falle von Bibliotheken führt die durchaus erwünschte konsequente Verwendung von Bibliotheksfunktionen zu zahlreichen Abhängigkeiten, die eine klare Abtrennung der Bibliothek verhindern. Bei Rahmenwerken (PREE, 1997;

RÜPING, 1997) tritt ein vergleichbarer Effekt auf. Es entstehen eine Vielzahl von Abhängigkeiten zwischen anwendungsspezifischen Systemteilen und dem Rahmenwerk durch die Nutzung der durch das Rahmenwerk angebotenen Funktionalität bzw. durch die Anpassung des Rahmenwerks durch anwendungsspezifischen Code mittels der Mechanismen Vererbung (samt Polymorphie) und Delegation (GENSSLER, 2004).

4.2 Kombination von Mustersuche und Ballungsanalyse

Betrachten wir nochmals die in Abbildung 4.1 dargestellte Situation: Wäre vor der Ballungsanalyse bekannt, dass die dunkel eingefärbte Klasse die Rolle einer Fassade innehat, so können wir diese Information ausnützen und die Ballungsanalyse geeignet modifizieren. Eine naheliegende Möglichkeit, dies zu tun, ist in Abbildung 4.2 dargestellt.

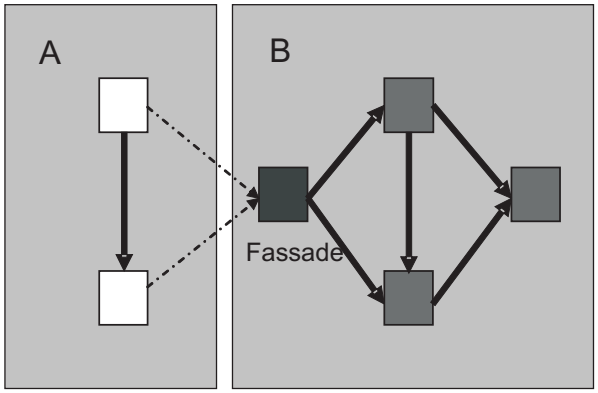


Abbildung 4.2: Gewichtete Abhängigkeiten nach Identifikation einer Fassade

Dazu werden die Abhängigkeiten zwischen den Bauteilen des Systems mit Gewichten versehen. In unserem Fall erhalten die Abhängigkeiten zwischen Dienstnutzern und der Fassadenklasse geringere Gewichte (hier durch gestrichelte Kanten dargestellt), so dass die Fassade während der Ballungsanalyse dem Teilsystem *B* zugeschlagen werden kann.[1]

Wir halten fest, dass die Defizite bereits vorhandener Ansätze, aussagekräftige Systemzerlegungen mit Hilfe von Verfahren zur Ballungsanalyse zu berechnen, darin begründet sind, dass die Rollen, die die einzelnen Bauteile für die Architektur eines Systems spielen, nicht berücksichtigt werden.

Wir entwickeln daher im folgenden eine Strategie, mit der die Bauteile eines Systems zunächst gemäß ihrer Bedeutung und Rollen für die Architektur klassifiziert werden. Diese

[1] Zusätzlich können die Gewichte für die Abhängigkeiten zwischen der Fassadenklasse und den Klassen zur Implementierung der Funktionalität des Teilsystems erhöht werden.

Rollen werden dann durch eine passende Gewichtung der Abhängigkeiten berücksichtigt, die für die nachfolgende Zerlegung des Systems durch Ballungsanalyseverfahren herangezogen werden sollen.

Zur Klassifikation der Bauteile des Systems wollen wir Verfahren zur Mustersuche heranziehen. Wir suchen dabei nach Mustern, die auf bestimmte Strukturen hinweisen, die für die Architektur und die damit verbundenen Gliederung des Systems in Teilsysteme eine Bedeutung haben. Wir wollen Strukturen, auf die solche Muster passen, im folgenden *Fingerabdrücke* nennen, da diese – vergleichbar mit entsprechenden Indizien in der Kriminalistik bei der Aufklärung von Sachverhalten – Hinweise auf die Architektur eines Systems liefern.

Solche Fingerabdrücke sind Strukturen auf der Ebene der Implementierungsstrukturen (vgl. Abbildung 2.3) eines Softwaresystems. Unsere Begutachtung des Standes der Technik in Abschnitt 3.2 zeigt, dass sich Strukturen auf der Ebene der Implementierungsstrukturen werkzeuggestützt mit gutem Erfolg über eine Mustersuche identifizieren lassen.

Fingerabdrücke entstehen, weil bei der Umsetzung von Architekturen in Implementierungen immer wieder bestimmte Entwurfsmuster eingesetzt werden (siehe auch Abschnitt 2.2). Im voranstehenden Abschnitt haben wir bereits ein solches Muster betrachtet: Fassadenklassen dienen zur expliziten Herausbildung von Schnittstellen für Teilsysteme und stellen ein Mittel zur Einhaltung des Geheimnisprinzips dar. Die Wahl des Architekturstils *Schichtenarchitektur* zieht die Verwendung des Entwurfsmusters Fassade zur Bildung von Schnittstellen für die einzelnen Schichten nach sich – zumindest wenn beim Übergang zur Implementierung des Systems gewisse *Best Practices* eingehalten werden, wie sie im Rahmen entsprechender Musterbeschreibungen (siehe z.B. BUSCHMANN et al. (1996)) angegeben sind.

Es gibt eine ganze Reihe verschiedener Architekturstile, die durch Architekturmuster dokumentiert sind. In den Musterbeschreibungen finden sich Implementierungsanweisungen, die in der Regel auf feinergranularere Muster auf der Ebene der Implementierungsstrukturen verweisen, die bei der Umsetzung der Architektur herangezogen werden. Häufig sind dies Entwurfsmuster, wie sie beispielsweise in GAMMA et al. (1995) beschrieben sind.

Abbildung 4.3 illustriert diesen Zusammenhang zwischen Architekturstilen bzw. Architekturmustern und den Fingerabdrücken, indem sie aufzeigt, welche feingranularen Muster bei der Implementierung der Architekturstile aus Abschnitt 2.2 eingesetzt werden können.

Beispiel 2: ***Umsetzung von Architekturstilen mit Hilfe feingranularer Muster***

Wir greifen zwei Beispiele aus Abbildung 4.3 heraus, um diesen Zusammenhang zu vertiefen:

- Eine geschichtete Architektur stützt sich – wie bereits erwähnt – auf das Enwturfsmuster *Fassade* ab, um die Bauteile der einzelnen Schichten hinter definierten Schnittstellen zu verbergen. Insbesondere die untersten Schichten definieren häufig Hilfsroutinen allgemeiner

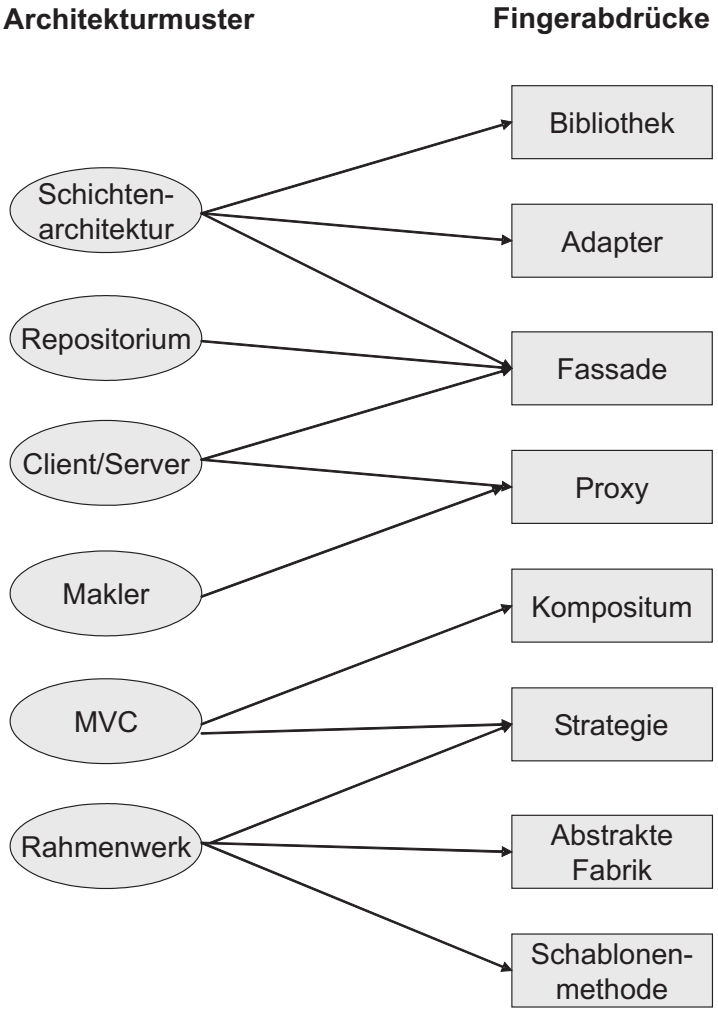


Abbildung 4.3: Architekturmuster und Fingerabdrücke

Bauart, die in den oberen Schichten verwendet werden können und sollen. Solche Hilfsroutinen haben aufgrund der Tatsache, dass sie in der Regel nicht speziell auf die Anwendungsdomäne des Systems zugeschnitten sind, den Charakter wiederverwendbarer *Bibliotheken*. Häufig entstammen solche bibliotheksartigen Teilsysteme sogar anderen, bereits implementierten Systemen. In diesem Fall kann es notwendig sein, die Schnittstellen (und ggf. auch in engen Grenzen das Verhalten) der Bibliothek an einen neuen Einsatzkontext anzupassen. Hierfür stellt das Entwurfsmuster *Adapter* (siehe Abschnitt 6.4) ein geeignetes Instrument dar.

- Die Definition und Verwendung eines Rahmenwerkes ist im klassischen Sinn kein Architekturstil. Ein Rahmenwerk wirkt aber genau wie Architekturstile strukturprägend und nützt feingranulare Muster in seiner Implementierung. Rahmenwerke definieren einen generischen Funktionsrahmen für eine Familie von Softwaresystemen. Sie legen insbesondere die Architektur der Systeme sowie einen einheitlichen (konfigurierbaren) Kontrollfluss fest. Zur Anpassung des Rahmenwerks an seinen Einsatzkontext bzw. zur Konstruktion eines konkreten Systems sind vordefinierte Adaptionspunkte, sogenannte *hot spots* vorgesehen. Solche

Adaptionspunkte können auf unterschiedliche Art realisiert werden. So können beispielsweise anwendungsspezifische Varianten von Algorithmen definiert und im Rahmenwerk verwendet werden, indem die Entwurfsmuster *Strategie* oder *Schablonenmethode* verwendet werden. In manchen Rahmenwerken kann auch eine Menge vordefinierter Klassen durch zusätzliche, an den Einsatzkontext angepasste Klassen ersetzt werden. Ist das Rahmenwerk für die Erzeugung von Objekten dieser Klassen verantwortlich, so muss hierfür häufig eine Fabrik definiert werden, die einer abstrakten Schnittstelle zur Objekterzeugung, der *abstrakten Fabrik*, entspricht.

Die Suche nach Fingerabdrücken in Form feingranularer Muster verspricht mehr Erfolg als nach Architekturmustern selbst zu suchen:

- Fingerabdrücke sind robuster gegen Zerwartung. Einzelne Muster können zwar im Laufe der Zeit unkenntlich werden, aber einige Muster, die zur Umsetzung eines Architekturmusters herangezogen wurden, bleiben in der Regel erhalten.
- Fingerabdrücke lösen auch das Problem der Variantenvielfalt bei der Umsetzung der Architekturmuster. Selbst unbekannte Varianten von Architekturmustern können so berücksichtigt werden, weil sie sich auf eine relativ kleine Menge bereits bekannter Entwurfsmuster abstützen.

Durch das Erkennen von Fingerabdrücken können wir zwar nicht eindeutig auf das Vorhandensein bestimmter Architekturmuster schließen, aber wir können trotzdem wertvolle Erkenntnisse über die Rolle der Elemente und Beziehungen des Strukturmodells gewinnen. Diese Erkenntnisse können wir dann bei der späteren Zerlegung des Systems in Teilsysteme ausbeuten.

4.3 Der hybride Ansatz in der Übersicht

Aus diesen Überlegungen kristalliert sich folgendes Verfahren zur Zerlegung eines Systems in Teilsysteme heraus:

- *Faktenextraktion zur Gewinnung eines Strukturmodells:* Der erste Schritt besteht in der Gewinnung eines Modells der Implementierungsstrukturen. Dieses Strukturmodell muss die für die nachfolgende Mustersuche benötigten Informationen über die Bauteile des Systems, ihre Eigenschaften und Beziehungen enthalten. Wie ein solches Modell beschaffen sein muss, welche Informationen darin enthalten sein müssen und wie wir diese gewinnen können, ist Inhalt von Kapitel 5.
- *Mustersuche:* Die Mustersuche dient dazu, bestimmten Elementen des Strukturmodells bzw. Beziehungen zwischen den Elementen Rollen zuzuordnen, die diese

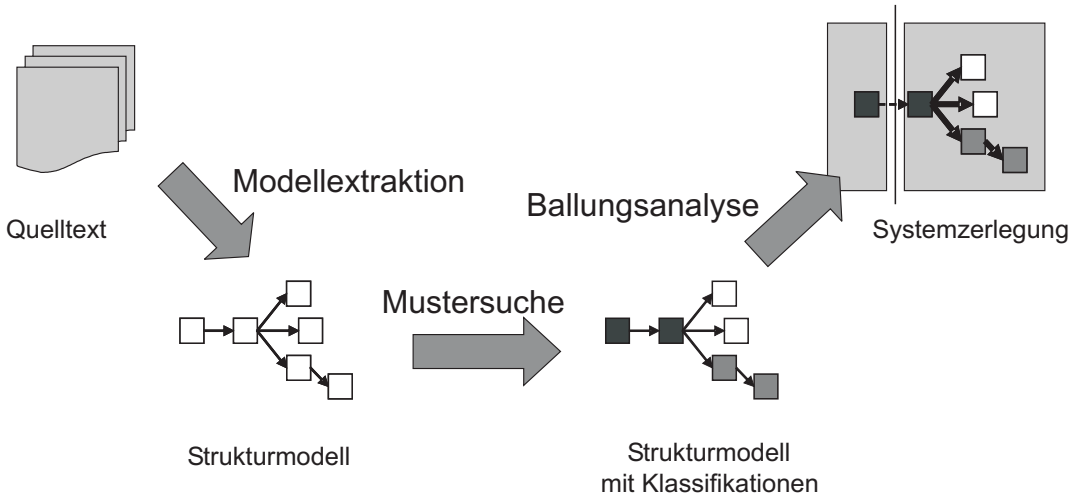


Abbildung 4.4: Hybrides Verfahren zur Strukturextraktion

in der Architektur des Systems einnehmen. Basis ist hierbei die Suche nach Mustern auf der Ebene der Implementierungsstrukturen, den Fingerabdrücken, die auf bestimmte architekturprägende Elemente des Systems hinweisen. Wir beschreiben in Kapitel 6, welche Fingerabdrücke für den Zweck der Teilsystembildung wesentlich sind, und wie wir diese mit Hilfe werkzeuggestützt anwendbarer Heuristiken in der Implementierungsstruktur eines Systems identifizieren können.

- *Ballungsanalyse:* Mit Hilfe von Verfahren zur Ballungsanalyse zerlegen wir das System in Teilsysteme. Dabei werden Ähnlichkeitsmaße herangezogen, um zu entscheiden, ob einzelne Elemente zusammen in ein Teilsystem gruppiert werden sollen. Ausgangspunkt hierfür ist ein gewichteter Abhängigkeitsgraph des Systems, den wir aus den Elementen des Strukturmodells ableiten können. Beim Aufbau des Abhängigkeitsgraphen müssen die Beziehungen zwischen den Bauteilen des Strukturmodells mit Hilfe der Ähnlichkeitsmaße in Kantengewichte des Abhängigkeitsgraphen überführt werden. In die Berechnung der Ähnlichkeitsmaße fließen die Ergebnisse der Mustersuche ein. Je nach der Rolle, die die zuvor identifizierten Fingerabdrücke in Architekturen typischerweise einnehmen, werden die Gewichte für Beziehungen zwischen bestimmten Bauteilen des Systems für die nachfolgende Ballungsanalyse angepasst.

  Kapitel 7 zeigt, wie der Abhängigkeitsgraph eines Systems aus dem Strukturmodell des Systems abgeleitet werden kann und wie die Abhängigkeiten zwischen den Bauteilen des Systems in Ähnlichkeitsmaße überführt werden. Wir geben dann zwei Verfahren zur Ballungsanalyse an, mit deren Hilfe eine Zerlegung des Abhängigkeitsgraphen berechnet werden kann. Diese Zerlegung spiegelt dann die Zerlegung des Softwaresystems in Teilsystemstrukturen wider.

Kapitel 5

Modellierung und Gewinnung von Strukturinformationen

Die Aufgabe dieses Kapitels besteht in der Modellierung der Implementierungsstruktur eines Systems, so wie sie mit üblichen Techniken aus dem Übersetzerbau aus der Implementierung des Systems gewonnen werden kann. Mit dieser Modellierung präzisieren wir zugleich auch den Begriff Implementierungsstruktur.

5.1 Strukturmodelle

Für die Analyseverfahren, die wir in den folgenden Kapiteln vorstellen werden, benötigen wir eine geeignete Modellierung der Implementierungsstruktur des Systems. Dazu legen wir zunächst in Form eines *Metamodells* fest, welche Informationen wir über die Implementierungsstruktur eines Systems benötigen und wie diese zusammenhängen.

Für ein konkretes zu untersuchendes Softwaresystem kann mittels Verfahren aus dem Übersetzerbau aus dem Quelltext eine *Ausprägung*[1] dieses Metamodells erzeugt werden. Eine solche konkrete Ausprägung wollen wir im folgenden *Strukturmodell* nennen. Demnach beschreibt das Metamodell die Menge aller erlaubten Strukturmodelle.

Bevor wir ein Metamodell festlegen, das uns als Basis für die weitere Arbeit dienen soll, stellen wir zunächst einige wichtige Anforderungen zusammen, denen dieses Metamodell (und damit auch alle Strukturmodelle) genügen soll.

- *Ausdruckskraft:* Unser Metamodell muss so beschaffen sein, dass es alle Informationen enthält, die wir für unseren Einsatzzweck, die Extraktion von Teilsystemstrukturen, benötigen.
- *Eindeutigkeit:* Ein Strukturmodell soll die Implementierungsstruktur eines Systems möglichst eindeutig beschreiben. Dies bedeutet, dass gleiche Systeme und gleiche Systemteile im Verlaufe der Modellgewinnung stets zu gleichen Modellen bzw. Modellfragmenten abgebildet werden. Umgekehrt ergibt sich daraus, dass

[1] auch: Instanz

wir aus einem konkreten Modell eindeutig auf das modellierte System schließen können.

Entsprechen wir der Forderung nach Eindeutigkeit nicht, gestalten sich Entwurf und Implementierung von Analyseverfahren kompliziert, weil wir aus den Modellinformationen keine eindeutigen Schlüsse mehr auf die Implementierungsstruktur des Systems ziehen können.

- *Einfachheit und Redundanzfreiheit:* Ein Strukturmodell sollte möglichst einfach und redundanzfrei sein. Enthält ein Strukturmodell größere Mengen an überflüssigen Informationen, erhöhen sich Komplexität und Menge der in Analyseverfahren zu verarbeitenden Daten. Dies erschwert zunächst die Entwicklung und Implementierung von Analyseverfahren, da das Metamodell dadurch komplizierter und schwerer verständlich wird. Zudem können sich auch die Laufzeiteigenschaften von Implementierungen der Analyseverfahren verschlechtern. Komplexe Daten erfordern in der Regel aufwendigere Verarbeitungsoperationen. Große Datenmengen führen zu längeren Laufzeiten und erhöhtem Speicherplatzbedarf.

  Redundanzen in einem Strukturmodell liegen dann vor, wenn das zugrundeliegende Metamodell Informationen definiert, die aus anderen bereits im Modell enthaltenen Informationen abgeleitet bzw. errechnet werden können. Redundanz erschwert den Umgang mit Strukturmodellen, da explizit Vorkehrungen getroffen werden müssen, um sicherzustellen, dass die im Modell gespeicherten Informationen widerspruchsfrei sind. Ein Metamodell, welches beispielsweise indirekte Vererbungsbeziehungen zwischen Klassen enthält, obwohl diese transitiv aus den direkten Vererbungsbeziehungen zwischen Klassen ableitbar sind, verletzt offensichtlich die Forderung nach Redundanzfreiheit.

- *Sprachunabhängigkeit:* Wir haben uns zu Beginn dieser Arbeit vorgenommen, ein Verfahren zu entwickeln, welches in der Lage ist, ein objektorientiert konstruiertes Softwaresystem weitestgehend unabhängig von der Programmiersprache, in der es implementiert wurde, in Teilsysteme zu zerlegen. Aus diesem Grund wollen wir fordern, dass die Strukturmodelle, die ja den Ausgangspunkt unseres Verfahrens darstellen, weitgehend programmiersprachenunabhängig gestaltet sind. Dazu muss das Metamodell in der Lage sein, alle Strukturierungselemente üblicher Programmiersprachen aufzunehmen.

  Ein Metamodell, welches der Forderung nach Sprachunabhängigkeit genügt, bildet somit eine programmiersprachenunabhängige Schnittstelle zum Umgang mit der Implementierungsstruktur eines Softwaresystems. Diese Schnittstelle erlaubt es uns, die weiteren Analyseverfahren, die wir im weiteren Verlauf dieser Arbeit entwickeln werden, programmiersprachenunabhängig zu formulieren.

- *Erweiterbarkeit:* CIUPKE (2001) führt als weitere Anforderung noch die Erweiterbarkeit von Metamodellen an. Obwohl diese Eigenschaft nicht von zentraler Be-

deutung für unsere Aufgabenstellung ist, wollen wir ihr dennoch Rechnung tragen. Die Erweiterbarkeit des Metamodells ist von Bedeutung, wenn wir die in dieser Arbeit vorgestellten Verfahren ausbauen und ergänzen wollen. Einige Vorschläge hierzu werden wir abschließend in Kapitel 9 machen.

Wir wollen anhand zweier etablierter Repräsentationen für Softwaresysteme diese Anforderungen erläutern:

Beispiel 3:

Klassendiagramme der *UML* (BOOCH und RUMBAUGH, 1998; ÖSTERREICH, 1998) stellen keine eindeutigen Strukturmodelle dar. Wir stellen nämlich zunächst fest, dass die Beziehung, die zwischen zwei Klassen eines objektorientierten Systems durch die Deklaration eines Attributs entsteht, in UML-Klassendiagrammen auf verschiedene Weise dargestellt werden kann, beispielsweise als Aggregation oder als Assoziation. Für ein und dasselbe Quelltextfragment eines Systems sind somit unterschiedliche Modelle möglich. Umgekehrt können wir aus dem Vorhandensein einer Assoziation in einem UML-Klassendiagramm nicht ohne weiteres Rückschlüsse auf die Struktur des Systems ziehen: in der Implementierung des Systems könnte sich diese Assoziation unter anderem in Form einer Deklaration eines Attributs oder als Deklaration eines Parameters, einer Variablen oder des Rückgabetyps einer Methode wiederfinden.[2]

Attributierte, abstrakte Syntaxbäume würden im Sinne der Forderungen nach Ausdruckskraft und Eindeutigkeit sicherlich geeignete Metamodelle zur Konzeption eines Analyseverfahrens für unsere Problemstellung darstellen, allerdings würden wir viele der darin enthaltenen Informationen nicht benötigen (und andere, dringend benötigte Informationen müssten wir erst aufwendig berechnen), so dass abstrakte Syntaxbäume das Kriterium der Einfachheit für unsere Problemstellung nicht erfüllen.

Klassendiagramme der UML entsprechen der Forderung nach Sprachunabhängigkeit, abstrakte Syntaxbäume dagegen nicht.

Wir stellen fest, dass sich die Beurteilung eines Metamodells bezüglich dieser Anforderungen immer nur bei gleichzeitiger Betrachtung der Analyseverfahren, die auf Basis dieses Modells konzipiert werden sollen, durchführen lässt. Unterschiedliche Analyseverfahren benötigen unterschiedliche Informationen über die Struktur eines Systems, so dass insbesondere die Anforderungen Ausdruckskraft, Eindeutigkeit und Einfachheit individuell passend in Entwurfsentscheidungen umgesetzt werden müssen.

5.2 Festlegung des Metamodells

Damit wir mit Hilfe unseres Metamodells die Implementierungsstruktur eines Softwaresystems beschreiben können, müssen wir festlegen, welche Elemente – auch *Entitäten* genannt – darin aufgenommen werden sollen und wie diese zu einander in *Beziehung*

[2]Eine ausführliche Diskussion der UML als Metamodell für die Implementierungsstruktur von Softwaresystemen findet sich in (TICHELAAR, 2001, Seite 46).

stehen. Unser Metamodell definiert daher, welche *Typen* von Entitäten und Beziehungen wir betrachten wollen. Zusätzlich wollen wir zur genaueren Beschreibung der Elemente und Beziehungen noch weitere Informationen vorhalten, dies tun wir mit Hilfe von *Attributen*, die wir den Elementen und Beziehungen zuordnen.

5.2.1 Strukturierungsmittel – Klassen, Namensräume und Methoden

Das zentrale Konzept in der statischen Struktur objektorientierter Systeme ist die *Klasse*. Klassen stellen den Konstruktionsplan für die Objekte eines Systems dar – sie legen Struktur und Verhalten von Objekten fest und definieren, wie Objekte erzeugt werden können. Die zentrale Bedeutung der Klassen ergibt sich somit direkt aus dem Konstruktionsprinzip objektorientierter Systeme, welches wir in Abschnitt 2.1 bereits betrachtet haben: ein System besteht (zur Laufzeit) aus einer veränderlichen Menge miteinander kooperierender Objekte. Entsprechend stützen sich die meisten Beschreibungen und Spezifikationen, die während der Konstruktion eines Systems angefertigt werden, auf Klassen. Auch in unserem Metamodell spielt die Entität Klasse eine zentrale Rolle. Damit wir Klassen eindeutig identifizieren können, statten wir sie mit dem Attribut name aus.

Dem Quelltext einer Klasse können wir ihre *Schnittstelle* entnehmen. Die Schnittstelle einer Klasse legt fest, welche *Methoden* ein Objekt anbietet. Zusätzlich werden die Eigenschaften eines Objektes zudem noch durch *Attribute* bestimmt. Attribute sind im allgemeinen (Verweise auf) andere Objekte, die sich während der Lebenszeit des Objektes, das heißt zwischen Erzeugung und Zerstörung, ändern können.

Wenn Klassen einen Teil ihrer Schnittstelle gemeinsam haben, können diese Gemeinsamkeiten mit Hilfe einer Oberklasse zusammengefasst werden. Die Oberklasse enthält dann den gemeinsamen Teil der Schnittstellen jener Klassen, den Unterklassen, und zwischen der Oberklasse und den Unterklassen entsteht dann eine *Vererbungsbeziehung*. Diese Beziehung, erbtVon, nehmen wir in unser Metamodell auf.

Durch die Bildung von Oberklassen können Klassen entstehen, die nicht mehr alle Informationen enthalten, um zur Laufzeit Objekte erzeugen zu können. Solche Klassen heißen *abstrakte* Klassen. Solche Klassen kennzeichnen wir in unserem Metamodell, mit Hilfe des booleschen Attribut istAbstrakt. Manche Programmiersprachen kennen Klassen, die nur aus einer Schnittstellendefinition bestehen. Um solche Klassen identifizieren zu können, verwenden wir das boolesche Attribut istSchnittstelle[3].

[3] Wir behandeln somit Schnittstellen wie Klassen. In der Tat können wir Schnittstellen als abstrakte Klassen sehen, in denen *alle* Methoden abstrakt sind. Der Grund für die Existenz von Schnittstellen in modernen Programmiersprachen ist die Vermeidung von Mehrfachvererbung und der damit verbundenen Probleme, wie beispielsweise Mehrdeutigkeiten bei der Auswahl gültiger Methodenimplementierungen in diamandförmigen Vererbungsgraphen

In einigen Programmiersprachen können Klassen und weitere Sprachelemente Namensräumen zugeordnet werden. Wir betrachten Namensräume zunächst nicht als strukturprägende Elemente eines Systems. Namensräume definieren für uns lediglich ein hierarchisches Namensschema für Klassen. Namensräume werden häufig um Mechanismen zur Einschränkung der Sichtbarkeit von Sprachelementen erweitert. Auf diese Weise können Sprachelemente beispielsweise als öffentlich oder lokal sichtbar (bezogen auf den eigenen Namensraum) deklariert werden. Namensräume nehmen wir in unser Metamodell in Form der Entität Namensraum auf. Öffentlich sichtbare Klassen können über das boolesche Attribut istOeffentlich gekennzeichnet werden.

Beispiel 4: *Modellelemente in* Java

Die Programmiersprache *Java* (FLANEGAN, 2005; GOSLING et al., 2000) kennt neben Klassen und abstrakten Klassen (deklariert durch `abstract class`) auch reine Schnittstellen. Diese werden durch das Schlüsselwort `interface` deklariert. Von einer Klasse können Unterklassen abgeleitet werden (Schlüsselwort `extends`); Schnittstellen müssen dagegen implementiert werden (Schlüsselwort `implements`). Für unsere Zwecke sind diese beiden Fälle gleichbedeutend; daher bilden wir beide Beziehungen im Metamodell durch erbtVon ab.

Klassen und Interfaces werden in Namensräumen angeordnet. Diese heißen in *Java* Pakete (Schlüsselwort `package`). *Java* unterstützt Mechanismen zur Einschränkung der Sichtbarkeit von Klassen: alle Klassen sind lokal im jeweiligen Paket sichtbar, sie können jedoch mit Hilfe des Schlüsselwortes `public` als öffentlich sichtbar deklariert werden. Nur in diesem Fall lassen sie sich auch in anderen Paketen verwenden.

Obwohl wir Pakete zunächst nur als Namensräume behandeln, verwenden in der Praxis viele *Java*-Entwickler Pakete auch zur Strukturierung von Systemen, indem zusammengehörende Klassen in ein gemeinsames Paket platziert werden. Durch eine wohldurchdachte Paketstruktur und die explizite Bildung von Schnittstellen für die Pakete lassen sich damit Teilsysteme definieren. Die Schnittstelle des Teilsystems ist dann in sauberen Entwürfen durch *Fassaden* gegeben (siehe dazu auch Abschnitt 2.1.3). Diese Fassaden sind Instanzen öffentlicher Klassen eines Pakets und kapseln alle Interna eines Paketes – insbesondere bilden sie die einzigen Ein- und Ausgangspunkte für den Steuer- und Datenfluss (BÄR, 2004).

Die Methode ist für uns ebenfalls eine wichtige eigenständige Entität unseres Metamodells. Methoden definieren das Verhalten von Objekten. Enthält die Definition einer Klasse eine Methodendefinition, so wollen wir diesen Sachverhalt in unserem Metamodell durch die Beziehung hatMethode wiedergeben.

Eine Methode wird durch ihren Namen und ihre Signatur, die sich aus der Definition der Methode aus dem Quelltext ergibt, eindeutig bestimmt. Die Signatur einer Methode legt fest, welche *Parameter* sie erwartet und welche Art von Ergebnis (*Rückgabetyp*) sie liefert. Für das Modellelement Methode definieren wir dementsprechend das Attribut name, sowie die Beziehungen hatRückgabetyp und hatParameter.

In vielen objektorientierten Sprachen kann die Sichtbarkeit von Methoden festgelegt werden. So kann beispielsweise festgelegt werden, dass eine Methode nur von Methoden derselben Klasse, aus Unterklassen oder von beliebigen anderen Klassen aus aufgerufen

werden kann. Wir führen für die unterschiedliche Sichtbarkeiten entsprechende Attribute ein. Für unsere Zwecke ist hier nur wesentlich, ob eine Methode global sichtbar ist. In diesem Fall kennzeichnen wir diese mit Hilfe des Attributs istOeffentlich.

Viele Sprachen kennen das Konzept von Klassenmethoden. Solche Klassenmethoden beeinflussen nicht den inneren Zustand von Objekten. Sie können also auch aufgerufen werden, wenn noch keine Objekte erzeugt wurden. Häufig werden diese Methoden auch zur Objekterzeugung oder damit verknüpften Aufgabenstellungen verwendet. Wir markieren solche Methoden in unserem Metamodell durch das Attribut istKlassenmethode.

Die Erzeugung und Initialisierung von Objekten obliegt in den meisten Sprachen speziell ausgezeichneten Methoden, sogenannten Konstruktoren. Wir markieren solche Methoden durch das Attribut istKonstruktor.

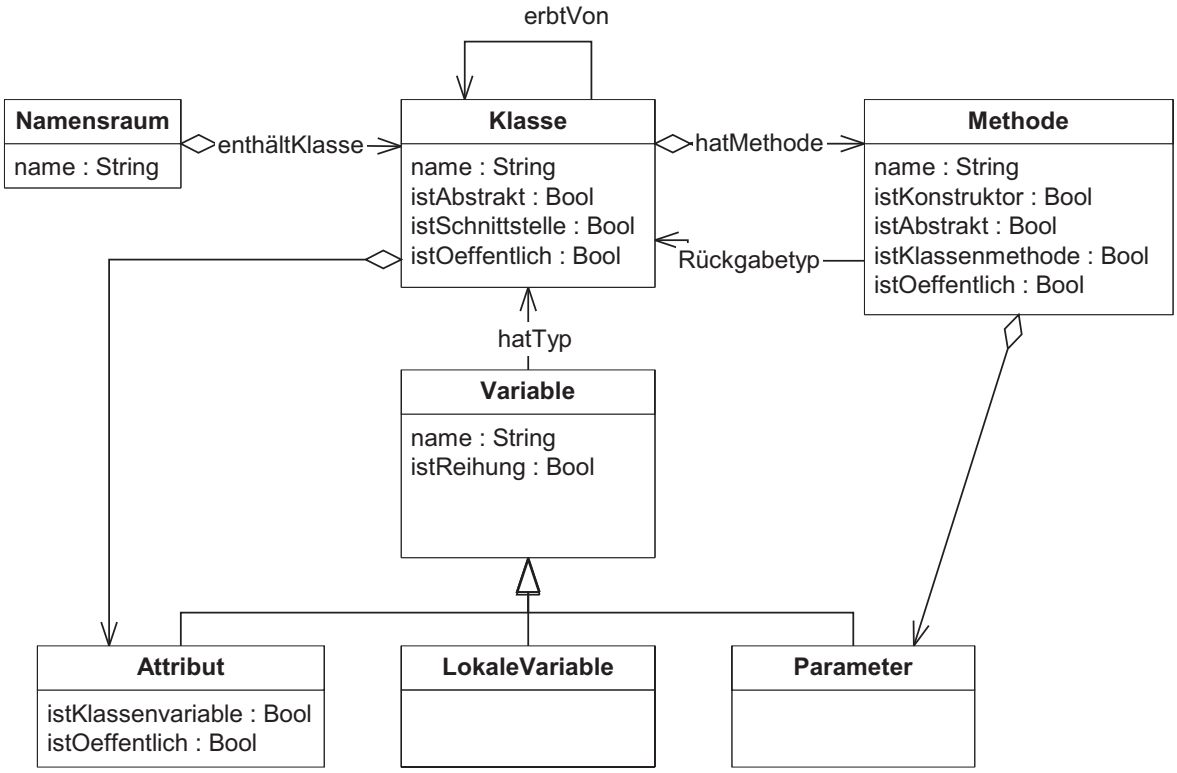


Abbildung 5.1: Metamodell: Namensräume, Klassen, Methoden und Variablen

5.2.2 Kooperation von Objekten – Variablen, Zugriffe und Methodenaufrufe

Die Implementierung von Methoden – und damit die Definition des Verhaltens von Objekten – erfolgt in *Methodenrümpfen*. Da Methoden nicht zwangsläufig über einen Rumpf verfügen müssen – im Falle abstrakter Klassen sind abstrakte Methoden möglich, die lediglich die Signatur einer Methode definieren – sehen wir in unserem Metamodell eine eigenständige Entität Rumpf vor. Ein Methodenrumpf enthält Anweisungen der verwendeten Programmiersprache.

Viele dieser Anweisungen greifen lesend bzw. schreibend auf *Variable* zu, die auf Objekte des Systems verweisen. Die Verwendung von Variablen hat einen entscheidenden Anteil an der Kooperation zwischen Objekten, daher nehmen wir in unser Metamodell eine eigenständige Entität Variable auf. Eine Variable hat einen Namen (Attribut name und einen *Typ*. In vielen objektorientierten Programmiersprachen können wir Klassen mit Typen gleichsetzen.[4] Eine Klasse bestimmt den Typ eines Objektes. Unterklassen werden dabei zu Untertypen, so dass Variable vom Typ T Objekte der Klasse T und aller Unterklassen von T aufnehmen können. In typisierten Sprachen werden Variablen explizit mit einem Typ deklariert – diesen Typ nennen wir *statischen* Typ. Der *dynamische* Typ einer Variable zu einem Zeitpunkt während der Programmausführung ist der speziellste Typ des Objekts, auf das die Variable gerade verweist. Wir merken uns den statischen Typ über die Beziehung hatTyp.

Eine Variable kann an verschiedenen Stellen vereinbart sein: Häufig kommen Variable als Attribute vor. Diese können – ähnlich wie Methoden – unterschiedliche Sichtbarkeiten haben. Auch hier interessiert uns in erster Linie, ob Attribute global sichtbar sind (siehe Attribut istOeffentlich). Zudem kann es sich um Klassenvariablen oder Instanzvariablen handeln (istKlassenvariable). Aufgrund dieser besonderen Eigenschaften von Attributen führen wir eine von Variable abgeleitete Entität Attribut ein. Die Zugehörigkeit eines Attributes zu einer Klasse modellieren wir mit der Beziehung hatAttribut. Weitere Verwendungsmöglichkeiten für Variable sind Parameter für Methoden oder lokale Variablen in Methodenrümpfen. Parameter nehmen wir als Entität Parameter in das Modell auf und verknüpfen sie über die Beziehung hatParameter mit der zugehörigen Methode. Lokale Variablen fügen wir ebenfalls dem Modell hinzu (LokaleVariable) und verknüpfen sie mit Hilfe der Beziehung hatVariable mit dem entsprechenden Methodenrumpf Rumpf.

Beispiel 4 (fortgesetzt):

In der Programmiersprache *Java* werden Klassenmethoden und Klassenvariablen durch das Schlüsselwort (`static`) ausgezeichnet. Methoden und Attribute können verschiedene Sichtbarkeiten ha-

[4] Einige objektorientierte Programmiersprachen, wie beispielsweise *Java* und *C++*, kennen zusätzlich zu Klassen primitive Typen wie Zahlen, Zeichen und ggf. weitere Typen wie Reihungen. Wir behandeln diese Typen wie Klassen.

ben. Global sichtbare Methoden und Attribute erhalten das Schlüsselwort (`public`). Zudem können Methoden und Attribute so vereinbart werden, dass sie nur in Unterklassen (`protected`), im selben Namensraum (ohne Schlüsselwort) oder nur in der Klasse selbst sichtbar sind (`private`). In *Java* erhalten alle Variablen einen statischen Typ durch entsprechende Deklarationen.

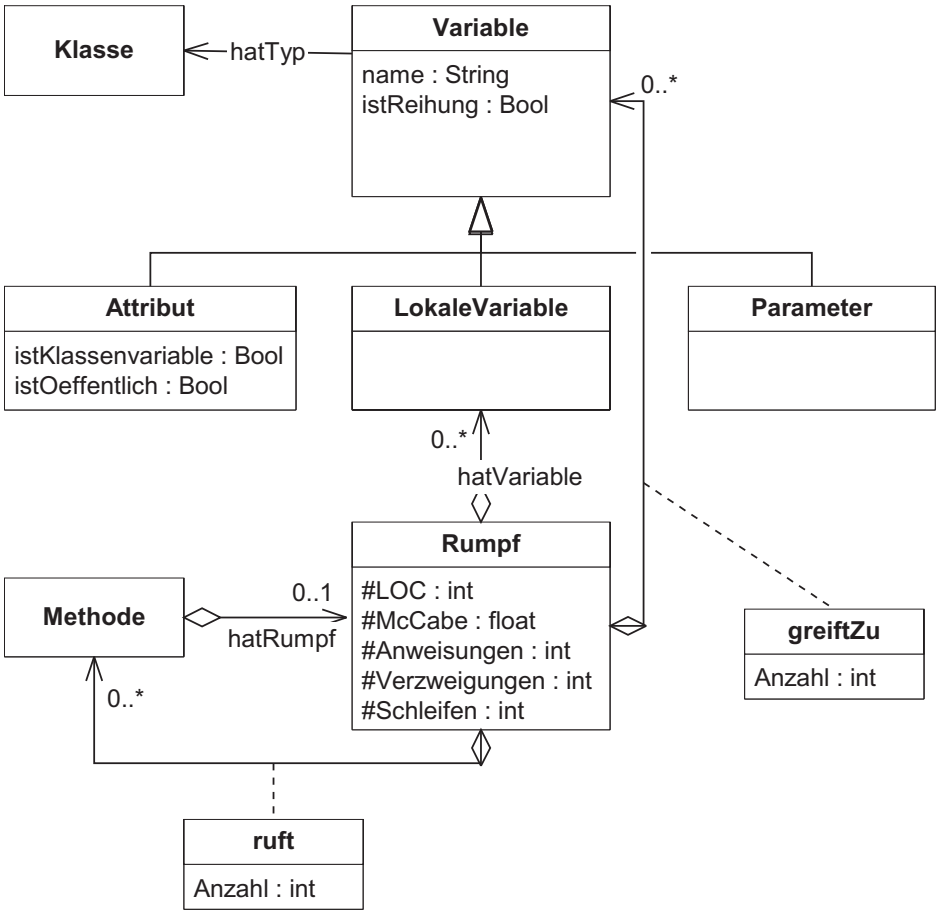


Abbildung 5.2: Metamodell: Methodenaufrufe und Variablenzugriffe

In Methodenrümpfen können Variable gelesen und neu belegt werden. Wir nehmen diese Variablenzugriffe in akkumulierter Form in das Modell auf, indem wir uns in der attributierten Beziehung greiftZu für jede Variable merken, an wievielen Stellen im Methodenrumpf auf sie zugegriffen wurde (Attribut #Zugriffe).

In Methodenrümpfen können auch Methodenaufrufe erfolgen. Methodenaufrufe erfolgen – sofern sie nicht an Klassenmethoden gehen – über Variable, die auf das Objekt verweisen, auf dem die zu rufende Methode ausgeführt werden soll. Dadurch entsteht in unserer statischen Sicht eine Beziehung ruft zwischen dem Methodenrumpf, der den Aufruf enthält und der speziellsten Methodendefinition mit passender Signatur, die in der

Klasse T des statischen Typs der Variablen oder einer der Oberklassen definiert wurde. Ein Beispiel soll dies verdeutlichen:

Beispiel 5: ***Beziehung durch einen Methodenaufruf***

Eine Klasse A enthalte eine Methode $m_1()$, in deren Rumpf ein Aufruf $b.m_2()$ erfolgt, dabei sei b eine Variable mit dem statischen Typ T_b.[5] Dann entsteht eine Beziehung ruft von $m_1()$ zu einer Methode $m_2()$ der Klasse B, wobei $B = T_b$, falls T eine Definition von $m_2()$ enthält. Andernfalls ist B die speziellste Oberklasse von T, die eine Definition für $m_2()$ enthält. Dieser Fall ist in Abbildung 5.3 dargestellt.

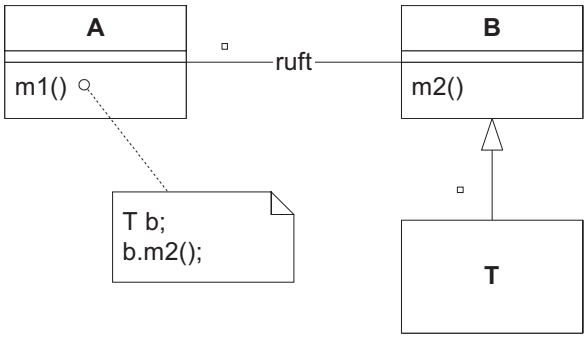


Abbildung 5.3: Beziehung durch einen Methodenaufruf

Bei Aufrufen an Klassenmethoden verfahren wir entsprechend, allerdings entnehmen wir die Typinformation zur Bestimmung der passenden Methode nicht einer Variablen, sondern direkt dem Aufruf selbst, der einen entsprechenden Klassenbezeichner enthält.

Wie im Falle der Variablenzugriffe, interessiert uns eine akkumulierte Sicht auf die Methodenaufrufe: wir merken uns mit Hilfe des Attributs #Aufrufe für jede Aufrufbeziehung, an wievielen verschiedenen Stellen entsprechende Aufrufe vorkommen.

Zusätzlich erfassen wir zu den Methodenrümpfen noch einige elementare Maße, beispielsweise die Anzahl der im Code enthaltenen Anweisungen, Verzweigungen und Schleifen, sowie einige Komplexitätsmaße und stellen diese über geeignete Attribute (LOC, #Anweisungen, #Verzweigungen und weitere) dar.

Beispiel 6: ***Ein kleines*** **Java-***Programm und sein Strukturmodell*

Listing 5.1 zeigt ein kleines Programm in *Java*. Dieses besteht aus drei Klassen: Eine implementiert das Entwurfsmuster *Singleton*, eine weitere eine Datenverbindung `Verbindung`, die von `Singleton` abgeleitet ist. Die dritte Klasse, `Kunde`, nimmt die Dienste der Datenverbindung

[5] b kann dabei auch das speziell ausgezeichnete Attribut *this* bzw. *self* sein, welches das eigene Objekt selbst bezeichnet, also ein Objekt des Typs A.

```
1  // Entwurfsmuster Singleton
2  public class Singleton {
3      private static Singleton ex = null;
4      public Singleton exemplar() {
5          if (ex == null) {
6              ex = new Singleton();
7          }
8          return ex;
9      }
10 }
11
12 // Eine Verbindung soll es nur einmal geben
13 public class Verbindung extends Singleton {
14     protected Verbindung() {
15         // ... Verbindung oeffnen
16     }
17     public sende(String paket) {
18         // ... Paket senden
19     }
20 }
21
22 // Ein Kunde verwendet die Verbindung
23 public class Kunde {
24     public kommuniziere() {
25         String nachricht;
26         //...
27         Verbindung v = Verbindung.exemplar();
28         v.sende(nachricht);
29         //...
30     }
31 }
```

Listing 5.1: Beispielprogramm

in Anspruch. 5.4 zeigt das zugehörige Strukturmodell für dieses Programm. Wir beobachten, dass eine großer Teil der Beziehungen in diesem Strukturmodell durch Variablennutzungen und Methodenaufrufe in den Methoden `exemplar()` und `kommuniziere()` entsteht.

5.3 Diskussion

Im letzten Abschnitt haben wir ein Metamodell für die Implementierungsstrukturen von Softwaresystemen definiert. Dieses erfüllt die Anforderungen, die wir in Abschnitt 5.1 formuliert hatten.

Die folgenden Kapitel 6 und 7 belegen, dass unser Modell alle Informationen enthält, die wir zur Lösung unserer Aufgabe, der Gewinnung von Teilsystemstrukturen, benötigen. Für diese Aufgabe ist die *Ausdruckskraft* des Modells demnach ausreichend.

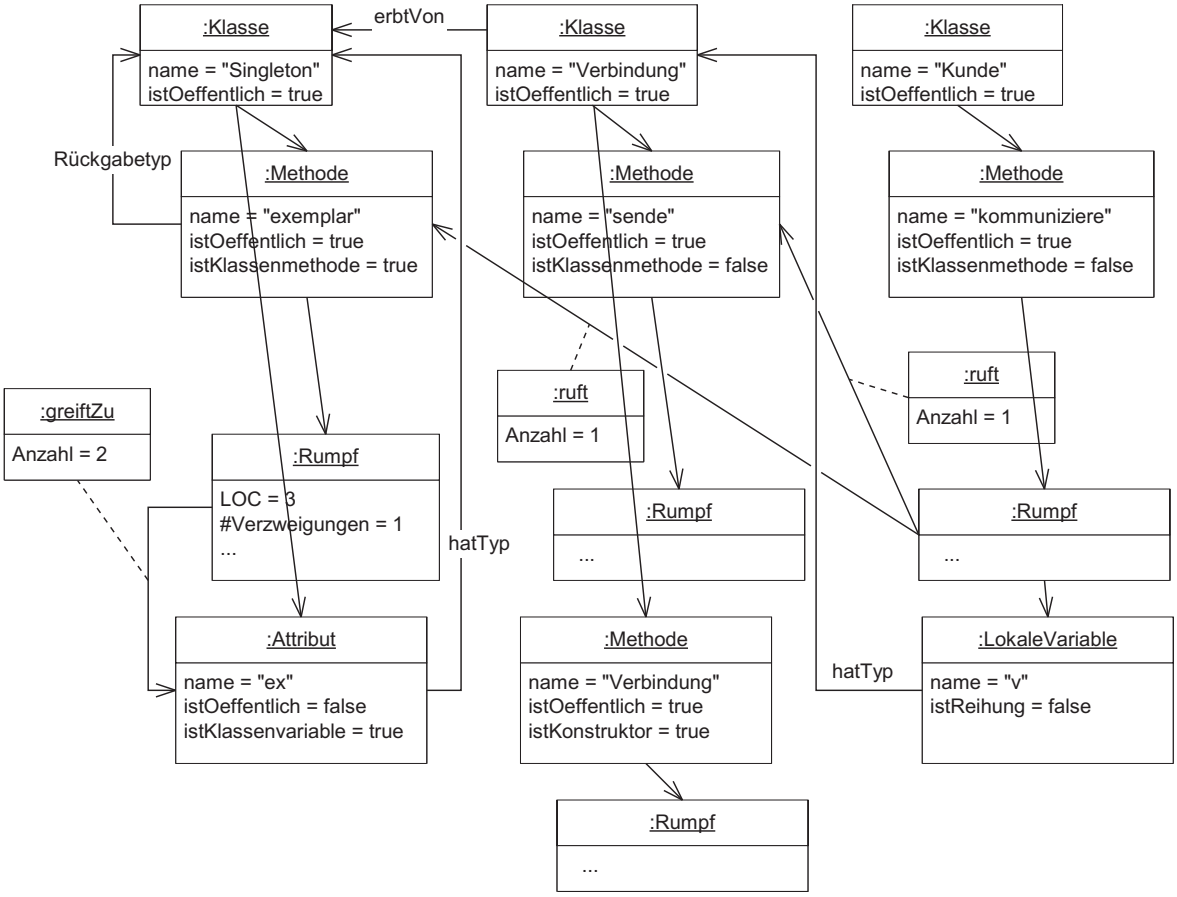


Abbildung 5.4: Strukturmodell für das Beispiel aus Listing 5.1

Unser Modell ist – innerhalb der Familie der objektorientierten Sprachen – weitgehend *sprachunabhängig*. Wir opfern dieser Sprachunabhängigkeit allerdings Feinheiten einzelner Programmiersprachen. Als Beispiel mag die Sichtbarkeit von Methoden gelten: Unser Modell unterscheidet lediglich zwischen *öffentlichen* Methoden und *privaten* Methoden. Einige Programmiersprachen wie *Java* oder *C++* bieten feinere Sichtbarkeitsstufen, die Programmiersprache *Smalltalk* dagegen tut dies nicht. Für unsere Zwecke reicht die von uns gewählte Modellierung aus.

Unser Modell ist weitgehend *eindeutig*. Ausnahmen ergeben sich lediglich durch die Feinheiten einzelner Programmiersprachen. Es ist ferner *redundanzfrei*, da es keine Informationen enthält, die wir direkt aus anderen Informationen des Modells ableiten können.

Die nächsten Kapitel werden zeigen, dass wir unser Modell sehr bequem verwenden können, um unsere Aufgabenstellung zu verfolgen. Insbesondere werden wir sehen, dass

wir auf einfache und übersichtliche Weise tiefergreifende Erkenntnisse über die Bauteile des Systems ableiten können. Es wird sich erweisen, dass unser Modell für unsere Zwecke sehr gut geeignet ist.

Alle Arbeiten, die sich mit Verfahren zur werkzeuggestützten Analyse von Softwaresystemen beschäftigen, definieren in der Regel Metamodelle, die für die jeweils vorgeschlagenen Analyseverfahren gemäß der oben angeführten Kriterien optimal abgestimmt werden. Darüberhinaus gibt es einige wenige Arbeiten, die ein weitgehend universell einsetzbares Metamodell für die Implementierungsstrruktur von Softwaresystemen definieren. Stellvertretend seien hier die Arbeiten von CIUPKE (2001) und TICHELAAR (2001) erwähnt. Wir stellen die wesentlichen Unterschiede zwischen unserem Modell und diesen beiden Modellen kurz zusammen.

- Unser Metamodell enthält im Unterschied zu dem von CIUPKE (2001) vorgeschlagenen Modell ausführlichere Informationen zu den Abhängigkeiten, die durch die Verwendung von Variablen und durch Methodenaufrufe entstehen. Ciupke interessiert nur, ob zwischen zwei Klassen bzw. deren Methoden eine Abhängigkeit entsteht, uns interessieren weitergehende Informationen, wie z.B. die Anzahl der Verwendungen einer Variable.

- Unser Metamodell stellt eine erhebliche Vereinfachung des Metamodells von TICHELAAR (2001) dar, da für unsere Arbeit der Aspekt der Codetransformationen und die dafür benötigten Detailinformationen aus dem abstrakten Syntaxbaum nicht relevant sind. Beide Modelle enthalten allerdings denselben Kern, da sie beide auf dem *FAMIX*-Modell (BÄR et al., 1999) basieren.

- Unser Metamodell enthält zusätzlich einige elementare Softwaremaße, die Eigenschaften der Methodenrümpfe wiedergeben. Wir benötigen diese Informationen zur Klassifikation von Methoden, die wir im Rahmen der Mustersuche nach Architekturfragmenten vornehmen (siehe Kapitel 6).

Unser Metamodell beschreibt – wie alle vergleichbaren Modelle – lediglich die statische Struktur von Softwaresystemen. Für unsere Zwecke ist diese ausreichend. In Einzelfällen wären zwar Kenntnisse über Laufzeitinformationen, beispielsweise Interaktionsprotokolle zwischen Bauteilen, von Interesse. Solche Informationen lassen sich für große Systeme allerdings nicht in ausreichender Form beschaffen. Der aktuelle Stand der Technik kennt hierfür lediglich zwei Wege:

- Wir können Systeme während ihrer Ausführung beobachten und entsprechende Informationen – beispielsweise durch Instrumentierung des Codes der Systeme – aufzeichnen (HEUZEROTH et al., 2002). Problematisch ist insbesondere bei großen Systemen, dass wir sicherstellen müssen, dass unsere Ausführung auch alle (relevanten) Ausführungspfade durch das System abdeckt.

- Alternativ können wir diese Informationen durch abstrakte Interpretation statisch berechnen. Die abstrakte Interpretation objektorientierter Systeme – beispielsweise zur Ermittlung von Interaktionsprotokollen – erfordert eine interprozedurale Analyse des Daten- und Steuerflusses. Diese ist sehr rechenzeit- und speicheraufwendig. Trotz einiger Fortschritte auf diesem Gebiet (siehe beispielsweise BÄR (2004)) können große Systeme mit diesen Mitteln noch nicht erfolgreich bearbeitet werden.

Wir verzichten daher in dieser Arbeit auf die Berücksichtigung solcher Laufzeitinformationen.

5.4 Modellextraktion

Ein Strukturmodell, das unserem Metamodell aus Abschnitt 5.2 entspricht, enthält ausschließlich Informationen, die sich durch eine Analyse des Quelltextes des zu untersuchenden Systems gewinnen lassen. Wir nennen den Prozess zur Gewinnung dieser Informationen aus dem Quelltext eines Systems *Fakten-* bzw. *Modellextraktion*.

Zur Faktenextraktion können wir gängige Technik aus dem Übersetzerbau einsetzen. Im Prinzip stellt unser Modell eine Abstraktion des attributierten Strukturbaums dar. Dieser bildet die Grundlage moderner Übersetzer (GOOS und WAITE, 1984; KASTENS, 1990). Wir benötigen daher einen Zerteiler (Parser) samt Symbolentschlüsselung für die zu analysierende Programmiersprache, sowie Teile der semantischen Analyse. Insbesondere sind hier die Namens- und die Typanalyse von Bedeutung. Letztere benötigen wir vor allem, um in objektorientierten Systemen den statischen, deklarierten Typ von Variablen (bzw. Ausdrücken) zu bestimmen, da wir diesen – wie in Abschnitt 5.2 beschrieben – für die Ermittlung von Variablenzugriffen und Methodenaufrufen benötigen.

Die Faktenextraktion ist demzufolge – im Gegensatz zu unserem Metamodell – immer speziell auf die zu analysierende Programmiersprache zugeschnitten.

5.5 Zusammenfassung

In diesem Kapitel haben wir beschrieben, wie Implementierungsstrukturen objektorientierter Softwaresysteme beschaffen sind, und wie wir diese so modellieren können, dass wir diese als Ausgangsbasis für unsere Verfahren zur Zerlegung von Softwaresystemen in Teilsysteme nutzen können.

Unser Modell für die Implementierungsstrukturen haben wir dabei sprachunabhängig gewählt, so dass wir unsere Verfahren so allgemein formulieren können, dass sie auf beliebige objektorientierte Softwaresysteme angewendet werden können. Lediglich die Extraktion der Informationen aus dem Quellcode muss programmiersprachenabhängig mit Hilfe etablierter Übersetzerbautechniken erfolgen.

Kapitel 6

Klassifikation von Strukturinformationen

Nach der Faktenextraktion und dem damit einhergehenden Aufbau eines Strukturmodells, wie im vorangehenden Kapitel beschrieben, liegen uns nun grundlegende Informationen über die Implementierungsstruktur des zu untersuchenden Softwaresystems vor. Diese Informationen sind allerdings bislang rein syntaktischer Natur. Wir wissen, welche Elemente unser System enthält und wie diese voneinander abhängen. Wir haben jedoch noch keine genaueren Kenntnisse darüber, welche Rolle diese Elemente hinsichtlich der Architektur des Systems spielen.

Inhalt dieses Kapitel ist daher, wie wir uns Erkenntnisse über die Elemente des Softwaresystems mit Hilfe von Verfahren zur Mustersuche verschaffen. Wir gehen dabei schrittweise vor. Wir beginnen bei einer Klassifikation der Methoden unseres Systems. Danach identifizieren wir Bestandteile von Bibliotheken. Zuletzt ordnen wir die Klassen unseres Systems bestimmten Musterstrukturen zu, die uns Fingerzeige auf ihre Rolle in der Architektur des Systems liefern.

6.1 Formalismus zur Beschreibung von Mustern

Wir benötigen zunächst einen Formalismus, mit dessen Hilfe wir Schlüsse aus den Informationen, die im Strukturmodell eines Softwaresystems enthalten sind, ziehen können. Unser Ziel ist es, dass wir mit Hilfe von Formeln aus der *Prädikatenlogik erster Stufe* strukturelle Eigenschaften beliebiger Elemente genauer beschreiben und herleiten können. Das folgende Beispiel soll dies illustrieren:

$$\begin{aligned} hatKlassenmethode(x) \quad &\rightarrow \quad Klasse(x) \wedge hatMethode(x,y) \\ &\quad\ \wedge istKlassenmethode(y) \end{aligned} \tag{6.1}$$

Diese einfache Formel definiert, wie sich für beliebige Klassen x bestimmen lässt, ob sie eine Klassenmethode enthalten. Damit wir solche Formeln verwenden können, müssen wir zunächst definieren, wie wir Strukturmodelle gemäß unseres Metamodells in prädikatenlogische Strukturen abbilden können.

Den Konventionen aus SCHÖNING (1992) folgend, definieren wir uns zunächst eine geeignete Struktur $\mathscr{A} = (\mathscr{U}_{\mathscr{A}}, \mathscr{I}_{\mathscr{A}})$, um die Bedeutung der Formeln festlegen zu können. $\mathscr{U}_{\mathscr{A}}$ sei dabei das Universum, welches den Formeln zugrundeliegen soll, und $\mathscr{I}_{\mathscr{A}}$ die Interpretation der Formeln.

Wir definieren zunächst eine Basismenge $\mathscr{B}_0$ von Prädikat- und Funktionssymbolen zur Struktur $\mathscr{A}$. Diese ergeben sich direkt aus der Definition unseres Metamodells für Strukturmodelle von Softwaresystemen:

- Eine Entität E des zugrundeliegenden Metamodells wird zu einem einstelligen Prädikatsymbol E.
- Eine Beziehung R des Metamodells wird zu einem zweistelligen Prädikatsymbol R.
- Ein Attribut A_E einer Entität E wird zu einem einstelligen Prädikat A_E, falls A_E ein boolesches Attribut ist. Stellt A_E keinen booleschen Wert dar, bilden wir A_E in ein einstelliges Funktionssymbol A_E ab.

  Ein Attribut A_R einer Beziehung wird zu einem zweistelligen Prädikatsymbol A_R, falls A_R ein boolesches Attribut ist. Stellt A_R keinen booleschen Wert dar, so bilden wir A_R in ein zweistelliges Funktionssymbol A_R ab.

Wir beachten dabei, dass die Syntax der Funktionen und Prädikate (in Form der Symbole und der Stelligkeiten) bereits durch die Definition unseres Metamodells aus Abschnitt 5.2 festliegt und somit nicht von den konkreten Strukturen des zu untersuchenden Softwaresystems abhängt. Eine Liste aller Prädikate und Funktionen, wie sie sich gemäß dieser Konstruktion ergeben, findet sich in Anhang A.

Das Universum unserer Struktur $\mathscr{A}$ und die Belegung der Prädikate und Funktionen hängt hingegen vom zu untersuchenden System (bzw. von seinem Strukturmodell) ab.

Das Universum $\mathscr{U}_{\mathscr{A}}$ umfasst primär alle Elemente des Strukturmodells des zu untersuchenden Systems. Daneben enthält $\mathscr{U}_{\mathscr{A}}$ noch die Menge der reellen Zahlen $\mathbf{R}$, so dass wir Softwaremaße auf Funktionen in prädikatenlogischen Formeln abbilden können.

Die Belegung der Funktionen und Prädikate gewinnen wir systematisch aus dem Strukturmodell unseres Softwaresystems, indem wir folgende Interpretation $\mathscr{I}_{\mathscr{A}}$ zu unserer Struktur festlegen.

- Für die Interpretation des einstelligen Prädikatsymbols E zu einer Entität E des Metamodells gelte:

$$\mathscr{I}_{\mathscr{A}}(E) \quad = \quad \{x \in \mathscr{U}_{\mathscr{A}} \mid x \text{ ist eine Ausprägung der Entität } E\}$$

- Für die Interpretation des zweistelligen Prädikates R zu einer Beziehung R des Metamodells gelte:

$$\mathscr{I}_{\mathscr{A}}(R) \quad = \quad \{(x,y) \in \mathscr{U}_{\mathscr{A}} \times \mathscr{U}_{\mathscr{A}} \mid \text{zwischen den Elementen } x \text{ und } y \text{ besteht die Beziehung } R\ \}$$

- Für die Interpretation des einstelligen Prädikates A_E zur Darstellung des booleschen Attributs A_E einer Entität gelte:

$$\mathscr{I}_{\mathscr{A}}(A_E) \quad = \quad \{x \in \mathscr{U}_{\mathscr{A}} \mid \text{das Attribut } A_E \text{ des Elementes } x \text{ hat den Wert } \mathit{wahr}\}$$

  Stellt A_E keinen booleschen Wert dar, so bilden wir A_E durch die einstellige Funktion A_E mit folgender Interpretation ab: $\mathscr{I}_{\mathscr{A}}(A_E)$ ist die Funktion A_E, die einer Entität x des Strukturmodells den Wert des Attributes $A_E(x)$ zuordnet.

  Entsprechend verfahren wir mit Attributen A_R, die eine Beziehung A_R des Metamodells beschreiben. Für den booleschen Fall erhält das zweistelliges Prädikat A_R folgende Interpretation:

$$\mathscr{I}_{\mathscr{A}}(A_R) \quad = \quad \{(x,y) \in \mathscr{U}_{\mathscr{A}} \times \mathscr{U}_{\mathscr{A}} \mid \text{das Attribut } A_R \text{ der Ausprägung } (x,y) \text{ der Beziehung } R \text{ hat den Wert } \mathit{wahr}\}$$

  Stellt A_R keinen booleschen Wert dar, so bilden wir A_R durch die zweistellige Funktion A_E mit folgender Interpretation ab: $\mathscr{I}_{\mathscr{A}}(A_R)$ ist die Funktion A_R, die einer Beziehung $(x,y) \in R$ des Strukturmodells den Wert des Attributes $A_R(x,y)$ zuordnet.

Aus der Formelmenge $\mathscr{B}_0$ ergibt sich durch die Belegung der Prädikate und Funktionen eine Faktenmenge $\mathscr{B}_{\mathscr{A}}$, die eine vollständige Beschreibung der Strukturen des Softwaresystems im Sinne des Metamodells aus Abschnitt 5.2 darstellt.

Beispiel 7: ***Darstellung eines Strukturmodells durch Formeln der Prädikatenlogik***

Für das Strukturmodell des kleinen Systems in *Java* aus Beispiel 6 besteht das Universum $\mathscr{U}_{\mathscr{A}}$ unter anderem aus folgenden Elementen: `Singleton`, `exemplar`, `ex` und r_1. r_1 steht dabei für den namenlosen Rumpf der Methode `exemplar`. Zudem enthält $\mathscr{B}_{\mathscr{A}}$ beispielsweise folgende gültige Prädikate:

$$
\begin{aligned}
\mathscr{A} &\models \mathit{Klasse}(\texttt{Singleton}) \\
\mathscr{A} &\models \mathit{hatMethode}(\texttt{Singleton},\texttt{exemplar}) \\
\mathscr{A} &\models \mathit{istOeffentlich}(\texttt{exemplar}) \\
\mathscr{A} &\models \mathit{Attribut}(\texttt{ex}) \\
\mathscr{A} &\models \mathit{hatAttribut}(\texttt{Singleton},\texttt{ex}) \\
\mathscr{A} &\models \mathit{Rumpf}(r_1) \\
\mathscr{A} &\models \mathit{hatRumpf}(\texttt{exemplar}, r_1) \\
\mathscr{A} &\models \#\mathit{Zugriffe}(r_1, \texttt{ex}) = 2
\end{aligned}
$$

Betrachten wir nun nochmals die Formel 6.1. Mit Hilfe einer nach dem soeben eingeführten Konstruktionsprinzip erzeugten Struktur $\mathscr{A}$ können wir nun ausgehend von der Faktenmenge $\mathscr{B}_{\mathscr{A}}$ für jede Klasse x eines Softwaresystems einfach ermitteln, ob x eine Klassenmethode enthält. Wir können die Formel 6.1 aber auch dazu verwenden, um alle Klassen x zu bestimmen, die (mindestens) eine Klassenmethode enthalten.

Wir verwenden dazu die Prinzipien des *logischen Programmierens* in *Prolog* (CLOCKSIN und MELLISH, 1994). *Prolog* liegt ein maschinell ausführbares Resolutionskalkül zugrunde, das sich hervorragend dazu eignet, zu überprüfen, ob abgeleitete Prädikate (wie $\mathit{hatKlassenmethode}(x)$ in unserem Beispiel) gelten, oder wie freie Variable (x) in unserem Beispiel belegt werden müssen, damit *Zielprädikate* (wie $\mathit{hatKlassenmethode}(x)$) gelten.

In diesem Sinne können wir *Prolog* als Spezifikationssprache zur Mustersuche verwenden. Wir werden davon in den nächsten Abschnitten ausführlich Gebrauch machen.[1]

Wir können *Prolog* auch als Ausführungsumgebung für die Mustersuche verwenden. Sowohl PRECHELT und KRÄMER (1996) als auch CIUPKE (1999) weisen nach, dass *Prolog* hierzu auch in der Praxis geeignet ist und es die umfangreichen Faktenmengen, die aus Strukturmodellen entstehen können, ausreichend schnell verarbeiten kann. TRIFU (2003) gibt in einer durch den Autor dieser Arbeit betreuten Diplomarbeit *Prolog*-Implementierungen für die Suchmuster in den nächsten Abschnitten an. Für den Werkzeugprototypen *ACT*, der eine Implementierung der Konzepte dieser Arbeit darstellt, wurden die Mustersuchen allerdings in *Java* implementiert. Wir werden auf diese Entwurfsentscheidung in Kapitel 8 noch genauer eingehen.

Im weiteren Verlauf dieses Kapitels werden wir zu unserer prädikatenlogischen Struktur $\mathscr{A}$ weitere Prädikate und Funktionen hinzufügen, und wir werden auch das Universum

[1] Die Syntax von *Prolog* unterscheidet sich aus Gründen der leichteren Handhabung bei der Eingabe von Programmen an einigen Stellen von der mathematischen Schreibweise der Formeln, die wir in dieser Arbeit verwenden. So würde man Formel 6.1 in *Prolog* auf folgende Weise angeben:

```
hatKlassenmethode(X) :-
            Klasse(X), hatMethode(X, Y), istKlassenmethode(Y).
```

$\mathcal{U}_{\mathcal{A}}$ noch erweitern. Wir verwenden die Prädikate $=$, $\neq$, $\leq$ und $<$ in Infixschreibweise gemäß ihrer mathematischen Bedeutung. An einigen Stellen konstruieren wir Mengen von Strukturelementen, die ein bestimmtes Prädikat erfüllen. In diesem Zusammenhang verwenden wir das Prädikat $\in$ und die Funktion $|\cdot|$ (Kardinalität) entsprechend ihrer mathematischen Bedeutung. Diese Erweiterungen beeinträchtigen jedoch weder die Korrektheit unserer Musterspezifikationen noch deren Ausführbarkeit im Rahmen des logischen Programmierens[2].

6.2 Klassifikation von Methoden

Mit Hilfe des Formalismus, den wir im vorangehenden Abschnitt eingeführt haben, wollen wir nun eine Klassifikation der Methoden eines Softwaresystems vornehmen. Für diese Klassifikation wollen wir jede Methode hinsichtlich folgender Kriterien einordnen: *Art*, *Rolle im Vererbungsgraphen* und *Verwendungszweck*.

Die *Art* einer Methode gibt an, was eine Methode tut. Natürlich können wir mit automatisierbaren Verfahren keine präzisen Kenntnisse über das Verhalten einer Methode erlangen. Wir können aber bestimmte Verhaltensmuster definieren und automatisch erkennen, die uns später bei der Suche nach komplexeren Mustern von Nutzen sein werden. Wir erkennen die folgenden Muster:

- *Abstrakte Methode:* Eine abstrakte Methode verfügt über keine Implementierung. Ob eine Methode abstrakt ist, können wir über das Attribut istAbstrakt unmittelbar dem Strukturmodell des Systems entnehmen. Infolgedessen haben wir hierfür im Rahmen unserer Konstruktion in Abschnitt 6.1 schon ein entsprechendes Prädikat $istAbstrakt(m)$ definiert.

- *Konstruktor:* Ein Konstruktor legt fest, wie Objekte einer Klasse erzeugt und initialisiert werden sollen. Ob eine Methode ein Konstruktor ist, können wir ebenfalls direkt dem Strukturmodell entnehmen – wir verwenden hierfür das Prädikat $istKonstruktor(m)$.

- *Leere Methode:* Der Rumpf einer leeren Methode enthält keinerlei Anweisungen. Leere Methoden werden in der Regel als *Hakenmethoden*[3] verwendet. Mit Hilfe von Hakenmethoden können in Unterklassen bestimmte Aspekte des Verhaltens einer Klasse angepasst werden, indem die Hakenmethoden passend überschrieben werden. Häufig tauschen leere Methoden im Zusammenhang mit dem Entwurfsmuster *Schablonenmethode* auf.

[2] So lässt sich beispielsweise eine Mengenkonstruktion der Form $M = \{x \mid P(x)\}$, wie wir sie später häufiger benötigen werden, in *Prolog* mit Hilfe des Prädikats `findall` durch das Codestück `findall(X, p(X), M)` ausdrücken (WIELEMAKER, 1997).

[3] engl. *hook methods*

Die Erkennung von leeren Methoden ist trivial:

$$istLeer(m) \rightarrow hatRumpf(m,r) \wedge (\#Anweisungen(r) = 0) \tag{6.2}$$

- *Schablonenmethode:* Eine Schablonenmethode liegt vor, wenn die Methode Teil des Entwurfsmusters *Schablonenmethode* (GAMMA et al., 1995) ist. Eine Schablonenmethode definiert das Skelett eines Algorithmus. Einzelschritte des Algorithmus können dabei in abstrakte oder leere Methoden (sogenannte *primitive Operationen*) ausgelagert werden. Verschiedene Unterklassen stellen dann unterschiedliche konkrete Implementierungen dieser abstrakten Methoden zur Verfügung. Durch dieses Muster können unterschiedliche Varianten von Konzepten und Algorithmen implementiert werden.

  Wir können eine Schablonenmethode einfach daran erkennen, dass ihr Rumpf einen oder mehrere Aufrufe an abstrakte oder leere Methoden derselben Klasse enthält:

$$\begin{aligned} &istSchablonenMethode(m) \rightarrow \\ &\quad hatRumpf(m,r) \wedge ruft(r,m') \wedge (istAbstrakt(m') \vee istLeer(m')) \\ &\quad \wedge hatMethode(c,m) \wedge hatMethode(c,m') \end{aligned} \tag{6.3}$$

- *Fabrikmethode:* Eine Fabrikmethode ist ebenfalls ein Muster aus dem Musterkatalog von GAMMA et al. (1995). Ihre Aufgabe besteht darin, die Erzeugung von Objekten zu kapseln, so dass diese in Unterklassen überschrieben und damit einfach die Menge der Klassen, aus denen Objekte erzeugt werden können, erweitert werden kann. Fabrikmethoden werden zur Umsetzung des Musters *Abstrakte Fabrik* benötigt. Hierauf gehen wir in Abschnitt 6.4 ein.

  Fabrikmethoden enthalten einen einzelnen Aufruf eines Konstruktors und geben das dadurch erzeugte Objekt zurück. Darüberhinaus enthalten sie nur noch in geringem Umfang Code.

$$\begin{aligned} &istFabrikmethode(m) \rightarrow \\ &\quad hatRumpf(m,r) \wedge \#Aufrufe(r,k) = 1 \wedge istKonstruktor(k) \\ &\quad \wedge hatRueckgabeTyp(m,t) \wedge hatMethode(t,k) \\ &\quad \wedge \#Anweisungen(r) \leq \xi_{OCode} \\ &\quad \wedge \#Verzweigungen(r) \leq 1 \wedge \#Aufrufe(r) = 1 \end{aligned} \tag{6.4}$$

An dieser Stelle wird besonders deutlich, dass es sich bei unseren Ausdrücken zur Mustersuche um *Heuristiken* handelt. Wir modellieren die Tatsache, dass Fabrikmethoden außer dem Code zur Objekterzeugung noch geringfügige Mengen anderen Codes – so genannten *Organisationscode* – enthalten können, durch die Forderung, dass die Fabrikmethode höchstens $\xi_{OCode} = 5$ weitere Anweisungen enthalten darf, darunter eine Verzweigung. Wir verwenden also gewisse "Faustregeln" zur Klassifikation der Elemente eines Systems, die in vielen Fällen gelten. Es kann durchaus Strukturen in Systemen geben, bei denen unsere Mustersuche versagt. Wir werden allerdings in Kapitel 8 sehen, dass diese Faustregeln in der Praxis sehr gut funktionieren.

Die meisten dieser Faustregeln basieren auf bestimmten Parametern, die Schwellwerte beschreiben. Dies sind Mindest- bzw. Höchstwerte, die erfüllt sein müssen, damit ein Bauteil entsprechend klassifiziert werden kann. Wir führen für diese Parameter symbolische Konstanten der Form ξ_{Name} ein. Wir stellen alle diese Konstanten in Anhang B zusammen und geben Werte für sie an, mit denen sich in der Praxis gute Ergebnisse erzielen lassen.

- *Delegierende Methode:* Die Aufgabe einer delegierenden Methode besteht darin, ihren Aufruf an eine Methode einer anderen Klasse weiter zu leiten. Diese Art von Delegation ist Grundbestandteil vieler Entwurfsmuster. Wir kommen auf Methoden dieser Bauart in Abschnitt 6.4 noch zurück.

  Eine delegierende Methode m kann ihren Aufruf entweder an eine Klassenmethode m'' weiterleiten, oder an eine Methode m' eines aggregierten Objektes vom Typ c'. m' darf dabei kein Konstruktor sein. m darf neben der Delegation nur noch in kleinerem Umfang zusätzlichen Code organisatorischer Natur enthalten, beispielsweise für Protokollierungsfunktionen oder Sicherheitsabfragen. Dieser Forderung tragen wir mit einer Heuristik Rechnung, indem wir fordern, dass m über nur wenige Anweisungen, eine Verzweigung und eine Schleife verfügen darf.

$$
\begin{aligned}
&istDelegierend(m) \rightarrow \\
&\quad hatRumpf(m,r) \wedge \#Anweisungen(r) \leq \xi_{OCode} \\
&\qquad \wedge \#Verzweigungen(r) \leq 1 \wedge \#Schleifen(r) \leq 1 \wedge \#Aufrufe(r) \leq 1 \\
&\quad \wedge (ruft(r,m') \wedge hatAttribut(r,x) \wedge hatTyp(x,c') \wedge c \neq c' \\
&\qquad\quad \wedge hatMethode(c,m) \wedge hatMethode(c',m') \\
&\qquad\quad \wedge \neg istKonstruktor(m'') \\
&\quad \vee ruft(r,m'') \wedge istKlassenmethode(m''))
\end{aligned}
\tag{6.5}
$$

- *Aliasmethode:* Eine Aliasmethode ist ein Spezialfall einer delegierenden Methode. Dabei erfolgt die Delegation jedoch nicht an eine Methode einer anderen Klasse,

sondern an eine Methode derselben Klasse. Entsprechend unterscheidet sich das Suchmuster auch nicht wesentlich von dem der delegierenden Methode. Der einzige Unterschied besteht darin, dass die Methode, an die delegiert wird, in der selben Klasse enthalten sein muss.

$$\begin{aligned} &istAlias(m) \rightarrow \\ &\quad hatRumpf(m,r) \wedge \#Anweisungen(r) \leq \xi_{OCode} \\ &\quad \wedge \#Verzweigungen(r) \leq 1 \wedge \#Schleifen(r) \leq 1 \wedge \#Aufrufe(r) \leq 1 \\ &\quad \wedge ruft(r,m') \wedge hatMethode(c,m) \wedge hatMethode(c,m') \\ &\quad \wedge \neg istKonstruktor(m') \end{aligned} \tag{6.6}$$

- *Zugriffsmethode:* Eine Zugriffsmethode[4] regelt den Zugriff auf ein Attribut einer Klasse. Dies können wir durch das folgende Suchmuster ausdrücken:

$$\begin{aligned} &istZugriffsmethode(m) \rightarrow \\ &\quad hatRumpf(m,r) \\ &\quad \wedge |verwendeteAttribute(r)| = 1 \wedge v \in verwendeteAttribute(r) \\ &\quad \wedge \#Zugriffe(r,v) = 1 \\ &\quad \wedge \#Anweisungen(r) \leq \xi_{OCode} \wedge \#Verzweigungen(r) = 0 \\ &\quad \wedge \#Schleifen(r) = 0 \wedge \#Aufrufe(r) = 0 \end{aligned} \tag{6.7}$$

  wobei

$$\begin{aligned} &verwendeteAttribute(r) = \{v \mid greiftZu(r,v) \\ &\qquad \wedge hatRumpf(m,r) \wedge hatMethode(c,m) \wedge hatAttribut(c,v)\} \end{aligned} \tag{6.8}$$

  Neben dem Zugriff auf einfache Attribute könnten wir hier auch das Hinzufügen und das Entfernen von Objekten in Reihungen oder Behälterklassen zulassen.

- *Normale Methode:* Diese Bezeichnung trifft auf alle Methoden zu, die in keines der obenstehenden Muster fallen.

Wir können Methoden auch nach ihrer *Rolle im Vererbungsgraphen* klassifizieren. Dabei geht es im wesentlichen darum, zu bestimmen, ob eine Methode die Funktionalität von Methoden in Oberklassen aufgreift, ob sie sie erweitert oder ob sie gar völlig neue Funktionalität implementiert. Wir unterscheiden dabei die folgenden Fälle (siehe auch LANZA und DUCASSE (2001) für eine vergleichbare Klassifikation):

[4] Zugriffsmethoden werden im *Java*-Umfeld häufig auch als *Getter*- und *Setter*-Methoden bezeichnet.

- *Implementierende Methode:* Ein solche Methode stellt eine Implementierung für eine abstrakte Methode in einer Oberklasse bereit.

$$\begin{aligned} &implementiert(m) \rightarrow \\ &\quad istUntertyp(c,c') \wedge habenKonformeSignatur(m,m') \\ &\quad hatMethode(c,m) \wedge hatMethode(c',m') \\ &\quad \wedge hatRumpf(m,r) \wedge istAbstrakt(m') \end{aligned} \tag{6.9}$$

wobei

$$\begin{aligned} &istUntertyp(c,c') \rightarrow \\ &\quad erbtVon(c,c') \vee erbtVon(c,c'') \wedge istUntertyp(c'',c')) \end{aligned} \tag{6.10}$$

$$\begin{aligned} &habenKonformeSignatur(m,m') \rightarrow \\ &\quad name(m) = name(m') \\ &\quad \wedge hatParameterTypen(m,P) \wedge hatParameterTypen(m',P') \wedge P = P' \\ &\quad \wedge hatRueckgabeTyp(m,r) \wedge hatRueckgabeTyp(m',r) \end{aligned} \tag{6.11}$$

Die Definition des Prädikats *istUntertyp* ist dabei rekursiv aufgebaut, um der Transitivität der Vererbungs- bzw. Untertypbeziehung Rechnung zu tragen. Das Prädikat *habenKonformeSignatur* verdient noch eine besondere Beachtung: Es muss für einige Programmiersprachen leicht abgeändert werden, so dass deren Konformitätsregeln für Methodensignaturen korrekt modelliert werden. Wir geben hier eine für die Programmiersprache *Java* passende Definition an.

Mit Hilfe dieser beiden Prädikate können wir ein Prädikat definieren, welches angibt, ob eine Methode m eine Methode m' einer Oberklasse überschreibt. Dieses Prädikat wird uns in der Folge noch von Nutzen sein:

$$\begin{aligned} &ueberschreibt(m,m') \rightarrow \\ &\quad habenKonformeSignatur(m,m') \\ &\quad \wedge hatMethode(c,m) \wedge hatMethode(c',m') \wedge istUntertyp(c,c') \end{aligned} \tag{6.12}$$

- *Erweiternde Methode:* Eine erweiternde Methode überschreibt eine Methodenimplementierung in einer Oberklasse und ruft diese im Rahmen ihrer eigenen Implementierung auf. Dies ist in objektorientierten Systemen recht häufig der Fall, wenn eine prinzipiell passende Implementierung in einer Oberklasse leicht an die spezialisierte Unterklasse angepasst werden soll (beispielsweise, um die etwas schärfere Klasseninvariante der Unterklasse zu bewahren).

$$\begin{aligned} &erweitert(m) \rightarrow \\ &\quad ueberschreibt(m,m') \wedge hatRumpf(m,r) \wedge ruft(r,m') \end{aligned} \tag{6.13}$$

- *Ersetzende Methode:* Eine ersetzende Methode überschreibt eine Methodenimplementierung in einer Oberklasse *ohne* diese im Rahmen ihrer eigenen Implementierung aufzurufen. Die neue Implementierung in der Unterklasse dient demnach als vollwertiger Ersatz der Implementierung in der Oberklasse.

$$\begin{aligned} &ersetzt(m) \rightarrow \\ &\quad ueberschreibt(m,m') \wedge hatRumpf(m,r) \wedge \neg ruft(r,m') \end{aligned} \tag{6.14}$$

- *Hinzugefügte Methode:* Eine hinzugefügte Methode ruft eine oder mehrere Methoden einer Oberklasse auf, ohne dabei eine bereits existierende Methode einer Oberklasse zu überschreiben.

$$\begin{aligned} &hinzugefuegt(m) \rightarrow \\ &\quad istUntertyp(c,c') \\ &\quad \wedge\neg(habenKonformeSignatur(m,m') \\ &\qquad \wedge hatMethode(c,m) \wedge hatMethode(c',m')) \\ &\quad hatMethode(c',m'') \wedge hatRumpf(m,r) \wedge \neg ruft(r,m'') \end{aligned} \tag{6.15}$$

- *Neue Methode:* Eine neue Methode überschreibt weder eine Methodenimplementierung in einer Oberklasse noch ruft sie Methoden einer Oberklasse auf. Dies ist der Fall, wenn keines der anderen Prädikate zutrifft.

Wir werden in Abschnitt 6.4 sehen, dass wir die Rolle einer Methode im Vererbungsgraphen benötigen, um Bestandteile von Rahmenwerken und Entwurfsmustern, die Vererbung verwenden (insbesondere *Kompositum, Abstrakte Fabrik* und *Strategie*), zu identifizieren.

Zusätzlich zu der soeben eingeführten Klassifikation nach Art und Rolle im Vererbungsgraphen können wir Methoden auch noch hinsichtlich ihres *Verwendungszwecks* einordnen.

Wir unterscheiden dabei:

- *Initialisierer:* Dies sind Konstruktoren oder Methoden, die nichts anderes tun, als die Attribute eines neu erzeugten Objektes geeignet zu belegen.

$$istInitialisierer(m) \rightarrow hatMethode(c,m) \wedge \frac{|modifizierteAttribute(c)|}{|attribute(c)|} \geq \xi_{ModAttr} \tag{6.16}$$

  wobei

$$\begin{aligned} attribute(c) &= \{x \mid hatAttribut(c,x)\} \\ modifizierteAttribute(m,c) &= \{x \mid hatAttribut(c,x) \wedge greiftZu(m,x)\} \end{aligned} \tag{6.17}$$

  Das Prädikat stuft eine Methode dann als Initialisierer ein, wenn ein bestimmter Anteil $\xi_{ModAttr}$ der Attribute modifiziert wird. Die Heuristik funktioniert dann recht gut, wenn wir fordern, dass 90 Prozent der Attribute modifiziert bzw. initialisiert werden: $\xi_{ModAttr} = 0,9$.

- *Schnittstellenmethoden:* Dies sind Methoden, welche die nach außen sichtbare Schnittstelle einer Klasse bilden.

  Für *Java* können wir dazu das folgende Heuristik verwenden:

$$bildetSchnittstelle(m) \rightarrow istOeffentlich(m) \tag{6.18}$$

- *Implementierungsmethoden:* Alle Methoden, die weder Initialisierer noch Schnittstellenmethoden sind.

6.3 Identifikation von Bibliotheksklassen

Die Implementierung von Anwendungsfunktionalität in Softwaresystemen stützt sich häufig auf Funktionen allgemeinerer Bauart ab. Beispiele für solche Funktionen gibt es unzählige. Nahezu jedes Softwaresystem verwendet beispielsweise Funktionen zur Ausgabe von Text und zur Manipulation von Zeichenketten. In objektorientierten Systemen

werden diese in der Regel in Hilfsklassen bzw. Bibliotheksklassen zusammengefasst. Solche Klassen können in Form externer, binärer Bibliotheken organisiert und bei Bedarf zur Übersetzungszeit durch einen Binder[5] zum ausführbaren Code hinzugebunden werden.[6] In vielen Systemen ist ein bestimmter Anteil solcher Hilfsroutinen unmittelbarer Bestandteil des Quellcodes. Da von solchen Hilfsroutinen an verschiedenen Stellen einer Anwendung Gebrauch gemacht wird, gibt es zahlreiche statische Abhängigkeiten zu den entsprechenden Klassen. Bei der Zerlegung von Systemen mit Hilfe von Ballungsanalysen werden diese Hilfsroutinen dann häufig zum Anwendungscode gruppiert.

Für das Verständnis des Systems sind die dadurch entstehenden Systemzerlegungen allerdings weniger geeignet, da die Bibliotheksklassen nichts zu den Belangen der eigentlichen Anwendungsfunktionalität beitragen. Aus diesem Grund wollen wir Bibliotheksklassen *vor* der Ballungsanalyse identifizieren, so dass diese in der Ballungsanalyse isoliert werden können.

Zur Identifikation von Bibliotheksklassen verwenden wir folgende einfache Heuristik: Eine Klasse ist dann eine Bibliotheksklasse, wenn sie von sehr vielen verschiedenen anderen Klassen, den *Kunden* verwendet wird.

$$istBibliotheksklasse(c) \rightarrow |aurufendeKlassen(c)| \geq \xi_{Kunden} \tag{6.19}$$

wobei

$$\begin{aligned} &aufrufendeKlassen(c) = \\ &\quad \{c' \mid hatMethode(c,m) \wedge istSchnittstellenmethode(m) \\ &\qquad \wedge hatMethode(c',m') \wedge hatRumpf(m',r') \wedge ruft(r',m)\} \end{aligned} \tag{6.20}$$

Die Konstante ξ_{Kunden} gibt an, wieviele Klassen als Nutzer der Klasse c auftreten müssen, damit c als Bibliotheksklasse gelten kann. Unsere Erfahrung zeigt, dass wir eine Klasse als Bibliotheksklasse einstufen können, wenn sie mehr als acht Nutzer hat.

6.4 Identifikation von Entwurfsmustern

Neben der Klassifizierung von Methoden und der Identifikation von Bibliotheksklassen können wir weitere strukturprägende Muster identifizieren. Im folgenden diskutieren wir welche Rolle die Muster *Fassade, Abstrakte Fabrik, Strategie, Adapter, Proxy und Kompositum* für die Architektur eines Softwaresystems spielen und zeigen, wie wir entsprechende Strukturen über Heuristiken identifizieren können.

[5] engl. *Linker*

[6] Im Falle des dynamischen Bindens findet dieser Vorgang sogar erst zur Laufzeit unmittelbar vor der Ausführung des entsprechenden Codes statt.

6.4.1 Fassade

Wir haben in Kapitel 4 bereits ausführlich dargestellt, welche Rolle eine *Fassade* für die Architektur eines Systems spielt: Eine Fassade bietet eine einheitliche Schnittstelle für die Elemente eines Teilsystems. Dazu verwendet sie eine *Fassadenklasse*, die eine abstrakte Schnittstelle definiert. Diese abstrakte Schnittstelle kapselt die Interna des Teilsystems und sie erleichtert die Verwendung des Teilsystems.

Eine Fassadenklasse delegiert den größten Teil der Funktionalität an eine Anzahl von Klassen im Inneren des Teilsystems, dessen Schnittstelle sie definiert. Wir können diesen Sachverhalt zur Identifikation einer Fassade nutzen. Wir benötigen zunächst folgende Mengen:

- $delegierteMethoden(c)$ ist die Menge aller Methoden, an die eine Klasse c Aufrufe delegiert:

$$\begin{aligned} delegierteKlassen(c) = \{m \mid & ruft(r,m) \\ & \wedge hatMethode(c,n) \wedge istDelegierend(n) \wedge hatRumpf(n,r)\} \end{aligned} \tag{6.21}$$

- $delegierte_K(c)$ ist die Menge aller Klassen, an die eine Klasse c Aufrufe delegiert.

$$\begin{aligned} & delegierteKlassen(c) = \\ & \quad \{c' \mid hatMethode(c',m) \wedge m' \in delegierteMethoden(c)\} \end{aligned} \tag{6.22}$$

- $gerufeneMethoden(c)$ ist die Gesamtmenge aller Methoden, die eine Klasse c aufruft:

$$\begin{aligned} & gerufeneMethoden(c) = \\ & \quad \{m \mid hatMethode(c,n) \wedge hatRumpf(n,r) \wedge ruft(r,m)\} \end{aligned} \tag{6.23}$$

Das Suchmuster für Fassadenklassen ist dann die folgende, recht einfache Heuristik:

$$\begin{aligned} & fassade(c,L) \rightarrow \\ & \quad L = delegierteKlassen(c) \wedge |L| \geq \xi_{VerborgeneKlassen} \\ & \quad \wedge \frac{|delegierteMethoden(c)|}{|gerufeneMethoden(c)|} \geq \xi_{Delegation} \end{aligned} \tag{6.24}$$

Gilt das Prädikat $fassade(c, L)$, so enthält c die Fassadenklasse und L die Menge der Klassen, für die c die Fassade bildet.

$\xi_{Delegation}$ gibt den Mindestanteil an Delegation, den die Klasse aufweisen muss, um als Fassade zu gelten. $\xi_{VerborgeneKlassen}$ gibt an, wieviele Klassen durch die Klasse "abgeschirmt" werden müssen, damit sie als Fassade gelten kann. Ein sinnvoller Wert für $\xi_{Delegation}$ liegt bei $0,5$ und für $\xi_{VerborgeneKlassen}$ bei 3.

Eine Fassade gibt einen entscheidenenden Hinweis für die Zerlegung eines Softwaresystems in Teilsysteme – sie markiert die Grenze des Teilsystems. Wir haben in Kapitel 4 bereits dargestellt, wie wir die Abhängigkeiten zwischen Kunden und Fassadenklasse einerseits und Fassadenklasse und den Klassen, die die abzuschirmende Interna implementieren zu gewichten haben, dass die Ballungsanalyse diese Grenze sinnvoll berücksichtigen kann – siehe dazu insbesondere auch Abbildung 4.2.

6.4.2 Proxy

Das Entwurfsmuster *Proxy* wirkt ebenfalls strukturprägend. Die Aufgabe eines Proxies ist es, ein Objekt mit einem Stellvertreterobjekt auszustatten. Dieses Stellvertreterobjekt ersetzt aus Sicht des nutzenden Codes das Ursprungsobjekt. Dazu delegiert das Stellvertreterobjekt die meisten Aufrufe an das Ursprungsobjekt. Die Aufgabe des Stellvertreterobjektes ist es, zusätzliche Belange zu implementieren, ohne dass der Kunde diese zusätzlichen Belange gesondert berücksichtigen muss. Beispielsweise lassen sich mit Hilfe eines Proxies Sicherheitsüberprüfungen, Caching- oder Verteilbarkeitsmechanismen implementieren. Die grundlegende Struktur des Musters *Proxy* ist in Abbildung 6.1 dargestellt.

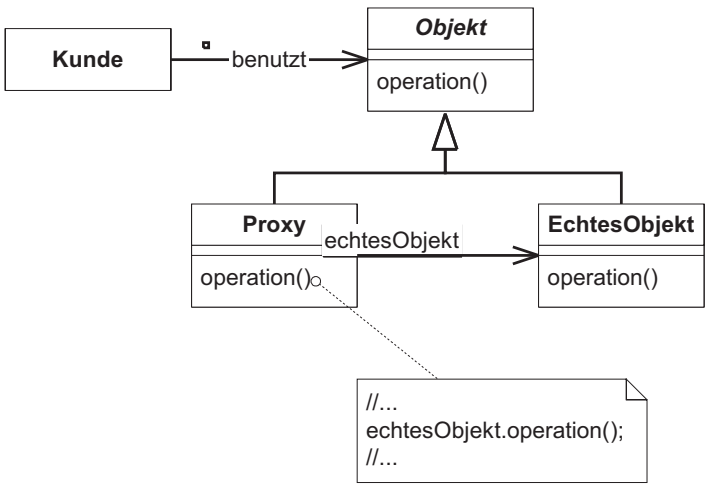


Abbildung 6.1: Proxy

Zur Suche nach Vorkommen des *Proxy*-Musters können wir ähnlich vorgehen wie bei der Erkennung von Fassaden. Das Haupterkennungsmerkmal bildet ein hohes Maß an Delegation. Die Delegationsziele beschränken sich – im Unterschied zur Fassade – auf eine einzige Klasse, nämlich die des Ursprungsobjektes. Da das Stellvertreterobjekt als vollständiger Ersatz für das Ursprungsobjekt verwendet werden soll, muss es dieselbe Schnittstelle besitzen. GAMMA et al. (1995) schlagen vor, dass dies über eine gemeinsame abstrakte Oberklasse erzwungen wird.

Insgesamt ergibt sich daraus folgende Heuristik zur Erkennung des Entwurfsmuster *Proxy*:

$$\begin{aligned} &proxy(p,c,s) \rightarrow \\ &\quad |delegierteKlassen(p)| = 1 \\ &\quad \wedge c \in delegierteKlassen(p) \wedge erbtVon(p,x) \wedge erbtVon(c,x) \\ &\quad \wedge \frac{|delegierteMethoden(p)|}{|gerufeneMethoden(p)|} \geq \xi_{Delegation} \end{aligned} \tag{6.25}$$

Gilt das Prädikat $proxy(p,c)$, so enthält p die Stellvertreterklasse, c die Ursprungsklasse, für welche die Stellvertreterklasse definiert wurde und s die gemeinsame Schnittstelle von Stellvertreterklasse und Ursprungsklasse.

Wir können davon ausgehen, dass es für das Verständnis eines Systems sinnvoll ist, alle Bauteile eines *Proxy*-Musters in ein Teilsystem zu packen. Kunden, also Klassen, die das Stellvertreterobjekt bzw. die abstrakte Schnittstelle von Stellvertreterobjekt und Ursprungsobjekt verwenden, liegen meistens in einem anderen Teilsystem. Abschnitt 7.2 bzw. Abbildung 7.2(b) ist zu entnehmen, wie wir die Abhängigkeiten zwischen den beteiligten Bauteilen gewichten müssen, dass die Ballungsanalyse eine passende Zuordnung in Teilsysteme vornimmt.

Die Heuristik zur Erkennung des Musters *Proxy* findet unter Umständen auch Vorkommen des Musters *Dekorierer* (GAMMA et al., 1995). Dies ist insbesondere bei kleineren Werten für den Parameter $\xi_{Delegation}$ der Fall. *Proxy* und *Dekorierer* beschreiben strukturell sehr ähnliche Muster. Da auch die beim *Dekorierer* beteiligten Bausteine häufig innerhalb eines Teilsystems zu finden sind, entstehen dadurch für die Zerlegung eines Softwaresystems keine nachteiligen Effekte.

6.4.3 Adapter

Das Entwurfsmuster *Adapter* ist ebenso wie die Entwurfsmuster *Fassade* und *Proxy* ein Delegationsmuster. Das Entwurfsmuster *Adapter* führt eine *Adapterklasse* ein, mit dessen Hilfe die Schnittstelle einer Klasse an einen spezifischen Verwendungskontext angepasst werden kann.

Die Suchheuristik für einen *Adapter* gleicht der für das Muster *Fassade*. Der einzige Unterschied ergibt sich daraus, dass eine Fassade die Schnittstellen *mehrerer* Klassen zu einer neuen Schnittstelle bündelt, ein *Adapter* stellt dagegen eine neue Schnittstelle für *eine einzige* Klasse bereit.

$$\begin{aligned} &adapter(a,c) \rightarrow \\ &\quad |delegierteKlassen(a)| = 1 \wedge c \in delegierteKlassen \\ &\quad \wedge \frac{|delegierteMethoden(a)|}{|gerufeneMethoden(a)|} \geq \xi_{Delegation} \end{aligned} \tag{6.26}$$

Gilt das Prädikat $adapter(a,c)$, so enthält a die Adapterklasse und c die zu adaptierende Klasse.

In der Regel ist es sinnvoll, die Adapterklasse der Verwendungsstelle, also dem Kunden, zuzuordnen. Abschnitt 7.2 bzw. Abbildung 7.2(c) ist zu entnehmen, wie wir die Abhängigkeiten zwischen den beteiligten Bauteilen gewichten müssen, dass die Ballungsanalyse eine passende Zuordnung in Teilsysteme vornimmt.

6.4.4 Kompositum

Die Grundidee des Entwurfsmusters *Kompositum* ist in einer abstrakten Klasse Komponente zu sehen, die sowohl primitive Objekte *(Blätter)* als auch Zusammensetzungen (Behälter, *Komposita*) primitiver Objekte repräsentiert. Die abstrakte Klasse Komponente definiert eine Schnittstelle, mit deren Hilfe sowohl Blätter als auch Komposita auf einheitliche Weise manipuliert werden können (vgl. Abbildung 6.2).

Das Entwurfsmuster *Kompositum* tritt in Softwaresystemen sehr häufig auf, da es sich sehr gut zur Modellierung baumartiger Strukturen eignet. Es gibt unzählige Beispiele für Modellierungsaufgaben, bei denen baumartige Strukturen eingestzt werden können (GAMMA et al., 1995). So finden wir baumartige Strukturen beispielsweise in graphischen Benutzerschnittstellen: Dialogfenster enthalten Dialogelemente. Dialogelemente können dabei entweder einzelne Elemente (Schaltflächen, Textfelder und dergleichen) oder Gruppen von Dialogelementen sein.

Die Klasse Kompositum, die das Kompositum primitiver Objekte modelliert, können wir anhand folgender Charakteristika identifizieren: Das Kompositum definiert Operationen, die es an die in ihm enthaltenen Objekte (Kinder) delegiert. Wir suchen daher also nach delegierenden Methoden, die über mindestens eine Schleife verfügen. Ziel der Delegation sind Methoden der abstrakten Klasse Komponente. Diese ist eine Oberklasse von

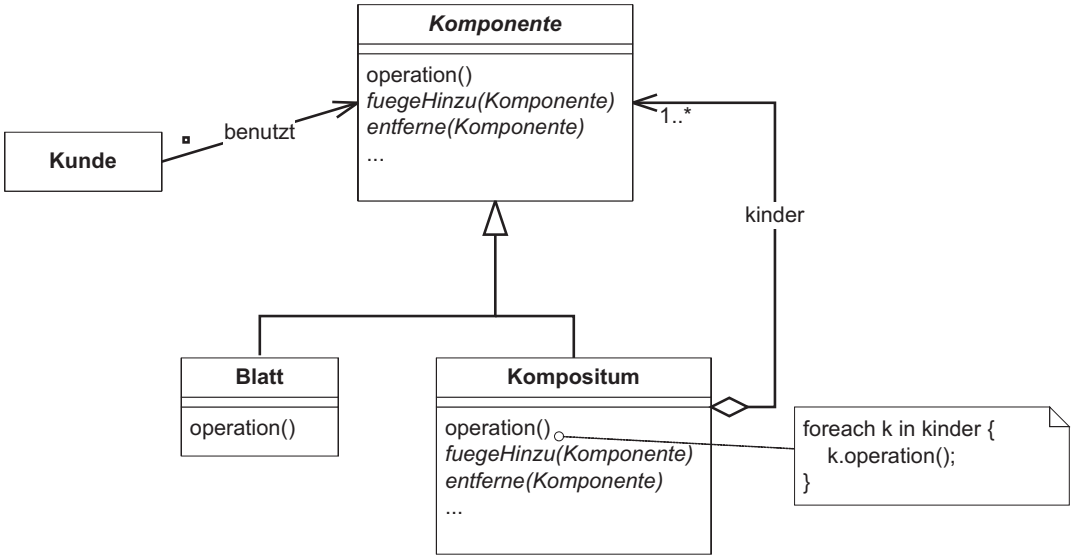


Abbildung 6.2: Kompositum

Kompositum und verfügt über Methoden, die signaturkonform zu den delegierenden Methoden in Kompositum sind. [7]

$$
\begin{aligned}
&kompositum(k,c) \rightarrow \\
&\quad hatMethode(k,o) \wedge istDelegierend(o) \wedge hatRumpf(o,r) \wedge \#Schleifen(r) >= 1 \\
&\quad \wedge ruft(r,o') \wedge habenKonformeSignatur(o,o') \\
&\quad \wedge hatMethode(c,o') \wedge istUntertyp(k,c)
\end{aligned} \tag{6.27}
$$

Gilt das Prädikat $kompositum(k,c)$, so enthält k das Kompositum und c die einheitliche, abstrakte Schnittstelle für alle Elemente der baumartigen Struktur.

Für das Verständnis eines Softwaresystems ist es sinnvoll, *alle* am Muster *Kompositum* beteiligten Elemente in dasselbe Teilsystem zu platzieren. Abschnitt 7.2 bzw. Abbildung 7.2(d) ist zu entnehmen, wie wir die Abhängigkeiten zwischen den beteiligten Bauteilen gewichten müssen, damit die Ballungsanalyse eine passende Zuordnung in Teilsysteme vornimmt.

[7] Im Prinzip könnten wir uns auch an der Aggregationsbeziehung zwischen Kompositum und Komponente orientieren. Diese Aggregationsbeziehung können wir aus dem Quelltext von *Java*-Systemen allerdings mit den Mitteln der Strukturextraktion (siehe Kapitel 5) nicht zuverlässig extrahieren, da solche 1:n-Beziehungen in *Java* im allgemeinen mit Hilfe von Behälterklassen implementiert werden. Die Elementtypen dieser Behälter, die den Zielen der Aggregationsbeziehung entsprechen, können für *Java* nur mit Hilfe einer aufwändigen Typinferenz (siehe beispielsweise BAUER (1999b)) berechnet werden. Wir müssen daher an deren Stelle die Methodendelegationen für unsere Suchheuristik heranziehen.

6.4.5 Strategie

Ziel des Entwurfsmusters *Strategie* ist es, eine Familie von Algorithmen so zu organisieren, dass Kunden je nach Bedarf unterschiedliche Ausprägungen der Algorithmen verwenden können. Dazu sieht das Entwurfsmuster *Strategie* die Definition einer abstrakten Schnittstelle für die Algorithmen vor. In Abbildung 6.3 ist diese Schnittstelle durch die abstrakte Klasse Strategie bzw. deren Methode algorithmus() gegeben. Unterschiedliche Ausprägungen des Algorithmus können dann in Form davon abgeleiteter, konkreter Strategien implementiert werden.

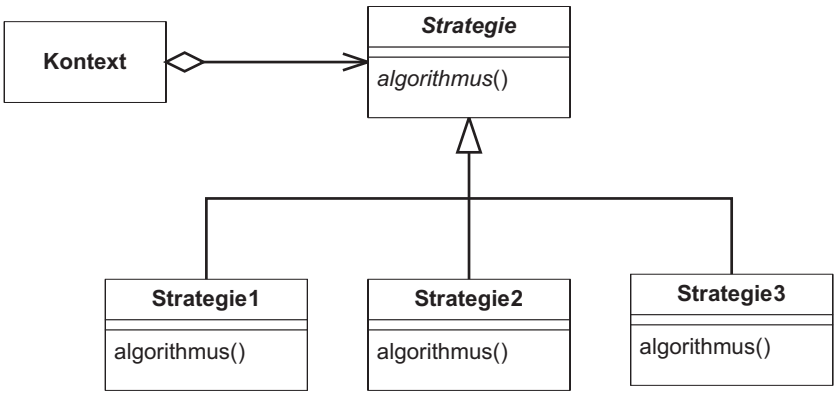


Abbildung 6.3: Strategie

Zur Identifikation von Vorkommen des Entwurfsmusters *Strategie* in den Implementierungsstrukturen eines Softwaresystems verwenden wir folgende Suchheuristik: Konkrete Strategien verfügen über eine öffentliche Methode, die eine abstrakte Methode einer Oberklasse implementiert – nämlich die der abstrakten Schnittstelle für den Algorithmus aus der Klasse Strategie. Zudem gehen wir davon aus, dass die Implementierung des Algorithmus der konkreten Strategie hinreichend komplex ist. Dies können wir mit Hilfe eines Maßes zur Codekomplexität überprüfen. Hierzu eignet sich beispielsweise die *zyklomatische Komplexität* nach McCabe (1976), die als Maß für die Komplexität eines Quelltextfragmentes die Anzahl der darin enthaltenen Verzweigungen (inklusive der Verzweigungen, die sich aus Schleifen ergeben) heranzieht.

$$
\begin{aligned}
&strategie(c,s) \rightarrow \\
&\quad M = \{m \mid hatMethode(c,m) \wedge istOeffentlich(m) \\
&\qquad \wedge hatRumpf(m,r) \wedge \#Verzweigungen(r) \geq \xi_{minKomplexitaet}\} \\
&\quad \wedge |M| = 1 \wedge m \in M \\
&\quad \wedge istUntertyp(c,s) \wedge hatMethode(s,m') \wedge istAbstrakt(m') \\
&\quad \wedge habenKonformeSignatur(m,m')
\end{aligned} \tag{6.28}
$$

Gilt das Prädikat $strategie(c, s)$, so enthält c eine konkrete Strategie zur abstrakten Strategie s. Die Konstante $\xi_{minKomplexitaet}$ gibt die minimale Komplexität an, die wir von einer Methode fordern, die den Algorithmus einer konkreten Strategie implementieren soll. In der Praxis hat sich hierfür eine zyklomatische Komplexität von drei Verzweigungen bewährt.

Für das Verständnis eines Systems ist sinnvoll, die Definition der Schnittstelle für die Strategie und alle Implementierungen davon in Form der Strategien in ein gemeinsames Teilsystem zu platzieren. Eine entsprechende Gewichtung der Abhängigkeiten zwischen den Bauteilen ist in Abbildung 7.2(e) dargestellt.

Alternativ kann es sinnvoll sein, die Schnittstelle von den konkreten Implementierungen zu trennen. In diesem Fall müssen die Gewichte der Abhängigkeiten zwischen den konkreten Strategien und der abstrakten Schnittstelle kleiner gewählt werden als die für die Abhängigkeiten zwischen Kunde und abstrakter Schnittstelle. Auf diese Weise kann die Ballungsanalyse Kunde und abstrakte Strategie zusammenhalten und die konkreten Strategien abtrennen.

6.4.6 Abstrakte Fabrik

Aufgabe des Entwurfsmusters *Abstrakte Fabrik* ist es, eine Schnittstelle zur Erzeugung einer Familie verwandter oder voneinander abhängiger Objekte zu definieren.

Das Muster enthält folgende Bestandteile: Die *Abstrakte Fabrik* definiert eine Schnittstelle zur Erzeugung verschiedener Produkte. Für jede Familie von Produkten gibt es davon abgeleitet eine konkrete Fabrik, die die verschiedenen Produkte der Familie erzeugen kann. Die unterschiedlichen Produkte werden mit Hilfe abstrakter Produkte definiert. Eine Familie zusammengehörender Produkte kann dann durch Spezialisierung der abstrakten Produkte erfolgen. Die Struktur des Musters *Abstrakte Fabrik* ist in Abbildung 6.4 dargestellt.

Das Muster *Abstrakte Fabrik* kann immer dann angewandt werden, wenn unterschiedliche Familien aus jeweils gleichartigen Elementen implementiert werden sollen und der Kunde ohne Änderungen je nach Bedarf unterschiedliche Familien verwenden soll. GAMMA et al. (1995) führen als Szenario hierfür die Elemente einer graphischen Benutzerschnittstelle an. Die einzelnen Familien implementieren dabei Oberflächenelemente für unterschiedliche Ausführungsplattformen. Dadurch kann ein plattformunabhängiger Teil der Implementierung der Benutzeroberfläche abgetrennt werden, der lediglich die Schnittstellen der abstrakten Produkte und der abstrakten Fabrik verwendet und daher keine Kenntnisse über die verschiedenen Ausführungsplattformen und deren Eigenheiten haben muss.

Zur Erkennung des Entwurfsmusters *Abstrakte Fabrik* suchen wir zunächst nach den konkreten Fabriken. Diese können wir einfach identifizieren. Sie enthalten zahlreiche *Fabrikmethoden* (siehe Abschnitt 6.2). Die zur konkreten Fabrik gehörende abstrakte

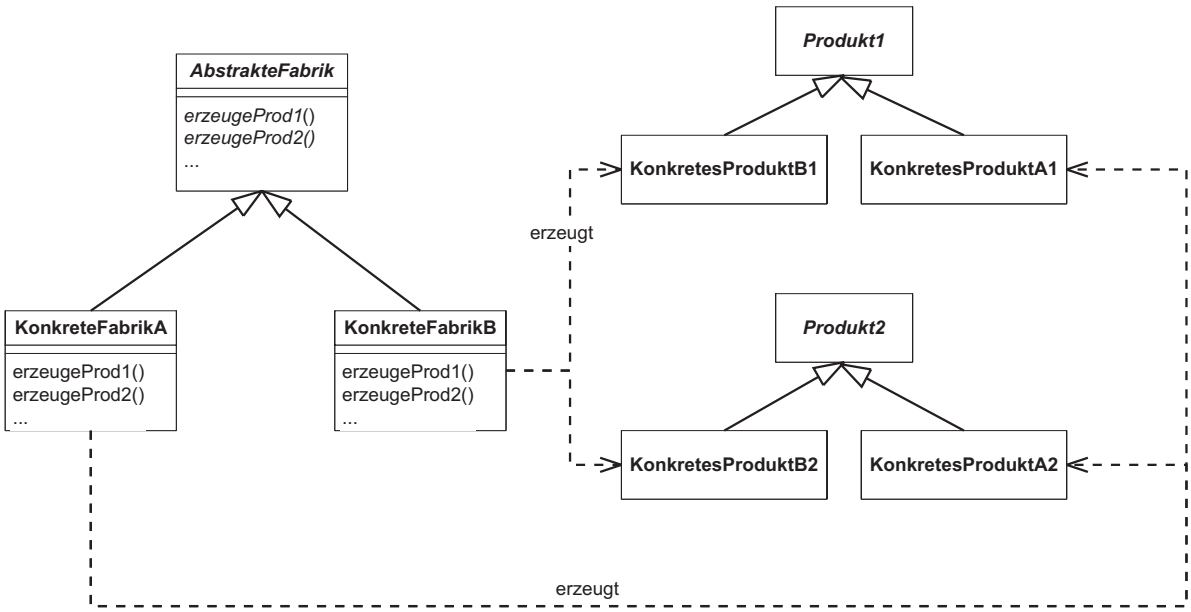


Abbildung 6.4: Abstrakte Fabrik

Fabrik erhalten wir, wenn wir nach einer Oberklasse suchen, die die abstrakten Definitionen der Fabrikmethoden enthält.

$$
\begin{aligned}
&konkreteFabrik(c,a) \rightarrow \\
&\quad M = \{m \mid hatMethode(c,m) \wedge istFabrikMethode(m) \wedge implementiert(m) \\
&\qquad\qquad \wedge istUntertyp(c,a) \wedge hatMethode(a,m') \\
&\qquad\qquad \wedge habenKonformeSignatur(m,m')\} \\
&\quad \wedge |M| \geq \xi_{minFabrikMeth}
\end{aligned} \tag{6.29}
$$

Gilt das Prädikat $konkreteFabrik(c,a)$, so enthält c eine konkrete Fabrik und a die zugehörige abstrakte Fabrik. Der Parameter $\xi_{minFabrikMeth}$ gibt die minimale Anzahl von Fabrikmethoden vor, über die eine Klasse verfügen muss, um sich als konkrete Fabrik zu qualifizieren. $\xi_{minFabrikMeth} = 4$ hat sich in der Praxis als tauglich erwiesen.

$$
abstrakteFabrik(a,L) \rightarrow L = \{c \mid konkreteFabrik(c,a)\} \tag{6.30}
$$

Gilt das Prädikat $abstrakteFabrik(a,L)$, so enthält a eine abstrakte Fabrik und L die Menge der dazu zugehörenden konkreten Fabriken.

Die Produkte, die jeweils zu den abstrakten und den konkreten Fabriken passen, lassen sich einfach über die Rückgabetypen der Fabrikmethoden ermitteln. Gilt das folgende Prädikat, so enthält L alle Produkte, die die konkrete Fabrik c erzeugen kann.

$$konkreteProdukte(c,L) \rightarrow$$
$$L = \{p \mid konkreteFabrik(c,a) hatMethode(c,m)$$
$$\wedge\wedge istFabrikMethode(m) \wedge hatRueckgabetyp(m,p)\} \qquad (6.31)$$

Zum Verständnis eines Softwaresystems trägt es bei, wenn wir jeweils zusammengehörende Produkte und Fabriken in ein Teilsystem gruppieren. Eine solche Zerlegung der Struktur des Musters *Abstrakte Fabrik* ist im Abschnitt 7.2 in Abbildung 7.2(f) dargestellt. Dieser Abbildung ist auch zu entnehmen, wie wir die Abhängigkeiten zwischen den Bauteilen des Musters bewerten müssen, damit die Ballungsanalyse später eine geeignete Aufteilung des Systems berechnen kann.

6.5 Diskussion und Zusammenfassung

Wir haben in diesem Kapitel gezeigt, wie sich die Bauteile des Systems mit Hilfe einer formalisierbaren Auswertung unseres Modells der Implementierungsstrukturen klassifizieren lassen. Mit Hilfe dieser Klassifikation können wir Aussagen darüber machen, welche Rolle die einzelnen Bauteile für die Architektur eines Systems spielen. Insbesondere können wir umfangreiches Wissen darüber gewinnen, an welchen architekturprägenden Strukturen die einzelnen Bauteile beteiligt sind.

Wir haben gesehen, wie – ausgehend von einer Klassifikation der Methoden des Systems – einfache Suchheuristiken dazu verwendet werden können, um Bibliotheksklassen und verschiedene Entwurfsmuster in den Implementierungsstrukturen eines Systems aufzuspüren.

Unsere Suchheuristiken basieren auf der Methodik des deduktiven Schließens. Die einzelnen Schlussregeln haben wir in diesem Kapitel in Form prädikatenlogischer Formeln spezifiziert, die sich beispielsweise gemäß dem Prinzip des logischen Programmierens direkt in ausführbare *Prolog*-Programme zur Mustersuche übertragen lassen. Alternativ können die Suchheuristiken auch mit anderen Mitteln implementiert werden, beispielsweise mit Hilfe von Anfragesprachen, die auf relationaler Algebra basieren. CIUPKE (2001) führt vor, wie hierzu vorzugehen ist.

Wir haben für einige Strukturmuster den Zusammenhang zwischen ihrer Musterstruktur und sinnvollen Zerlegungen des Systems in Teilsysteme diskutiert. Darauf aufbauend zeigen wir im nächsten Kapitel, dass wir unsere Klassifikation der Bauteile des Systems und die identifizierten Strukturmuster dazu nützen können, mit Hilfe von Verfahren zur Ballungsanalyse die Bauteile des Systems zu Teilsystemen zusammenzustellen, die eine gute Ausgangsbasis für das Verständnis und damit auch für die Weiterentwicklung eines Softwaresystems bilden.

Natürlich lassen sich neben den in diesem Kapitel vorgestellten noch weitere Klassifikationen der Bauteile des Systems vornehmen, indem zusätzliche Suchheuristiken definiert werden. Sowohl unser Modell für die Implementierungsstrukturen eines Softwaresystems als auch die in diesem Kapitel vorgestellte Spezifikationstechnik bieten hierzu vielfältige Möglichkeiten.

Allerdings können wir nicht alle strukturprägenden Muster allein auf Basis der statischen Struktur eines Softwaresystems identifizieren. Ein Beispiel hierfür stellt das *Beobachter*-Muster dar. Zur zuverlässigen Identifikation dieses Musters benötigen wir Kenntnisse über die Aufrufreihenfolgen von Methoden in Form von Protokollen. Wie wir bereits in Abschnitt 5.3 diskutiert haben, lassen sich solche Informationen für große Softwaresysteme nach dem heutigen Stand der Technik leider kaum zuverlässig beschaffen, so dass wir auf die Suche nach solchen Mustern verzichten müssen.

Kapitel 7

Ballungsanalysen zur Zerlegung von Softwaresystemen

Wir erläutern in diesem Kapitel, wie wir Systeme, die wir gemäß der in Kapitel 5 erarbeiteten Strukturmodelle darstellen können, mit Hilfe von Algorithmen zur Ballungsanalyse in Teilsysteme zerlegen können. Neben diesen Algorithmen werden Ähnlichkeitsmaße eine wesentliche Rolle spielen, mit deren Hilfe wir bewerten können, ob jeweils zwei Bauteile zusammen in ein Teilsystem platziert werden können.

7.1 Grundkonzepte

Zunächst benötigen wir eine gegenüber den Strukturmodellen aus Kapitel 5 nochmals deutlich vereinfachte Sicht auf Softwaresysteme. Wir definieren dazu den Abhängigkeitsgraphen eines Systems:

Definition 1 *Der* Abhängigkeitsgraph *eines Systems ist gegeben durch den gerichteten, gewichteten Graphen* $G(V,E,\omega)$ *. Die Knotenmenge* $V = v_1,...,v_n$ *entspricht dabei den* n *Klassen des Systems und die Kantenmenge* $E \subseteq V \times V$ *modelliert die Abhängigkeiten zwischen den Klassen des Systems.*

Die Gewichtungsfunktion $\omega : V \times V \rightarrow \mathbf{R}$ *bildet ein Maß für die Stärke der Abhängigkeiten zwischen jeweils zwei Klassen. Für* $e \in (V \times V) \setminus E$ *gelte* $\omega(e) = 0$.

Mit Hilfe des Abhängigkeitsgraphen G können wir nun unsere Aufgabe, ein Softwaresystem in Teilsysteme zu zerlegen, wie folgt formulieren: Gesucht ist eine Zerlegung oder *Partition* der Knotenmenge V in k Teilmengen $C = C_1,...,C_k$ mit folgenden Eigenschaften:

1. *Konsistenz:* Für alle i, j mit $i \neq j$ gelte:

$$C_i \cap C_j = \emptyset \tag{7.1}$$

2. *Vollständigkeit:* Es gelte:

$$\bigcup_{i=1}^{k} C_i = V \tag{7.2}$$

Natürlich reicht es für unsere Zwecke nicht aus, von einer Zerlegung lediglich diese beiden Eigenschaften zu fordern – sonst wäre die Zerlegung $C = \{V\}$, die das ganze System als unzerlegbar betrachtet, bereits eine gültige Lösung unseres Problems. Wir fordern zusätzlich noch, dass eine bestimmte *Qualitätsfunktion* $f(G,C)$ maximiert wird. Damit können wir die *Optimalität* einer Zerlegung fordern. Die Zerlegung C ist dann optimal bzgl. der Qualitätsfunktion f, wenn für jede beliebige Zerlegung B gilt:

$$f(G,B) \leq f(G,C) \tag{7.3}$$

Es gibt verschiedene Möglichkeiten, eine für unsere Aufgabenstellung passende Qualitätsfunktion f zu definieren (vgl. dazu auch Abschnitt 7.4). Gemeinsam ist allen Definitionen aber, dass sie die Kantengewichte des Graphen in Form der Gewichtungsfunktion ω zur Gewinnung einer Qualitätsaussage über die Zerlegung C heranziehen.

Gleichung 7.4 enthält ein Beispiel für eine solche Qualitätsfunktion (CLARK et al., 2003):

$$f(G,C) = \sum_{i=1}^{k} Q_i \tag{7.4}$$

Q_i ist dabei der Qualitätsfaktor[1] für die Teilmenge C_i:

$$Q_i = \begin{cases} 0 & , \quad \mu_i = 0 \\ \frac{\mu_i}{\mu_i + \frac{1}{2}\sum_{i=1}^{k}(\varepsilon_{i,j}+\varepsilon_{j,i})} & , \quad \text{sonst} \end{cases} \tag{7.5}$$

Der Qualitätsfaktor Q_i setzt dabei die *interne Kohäsion* der Teilmenge C_i,

$$\mu_i = \sum_{v_1,v_2 \in C_i} \omega(v_1,v_2) \tag{7.6}$$

ins Verhältnis zur Summe aus eben dieser internen Kohäsion und der *externen Kopplung*

$$\varepsilon_{i,j} = \begin{cases} 0 & , \quad i = j \\ \sum_{v_1 \in C_i, v_2 \in C_j} \omega(v_1,v_2) & , \quad i \neq j \end{cases} \tag{7.7}$$

[1] CLARK et al. (2003) bezeichnen Q_i als Qualitätsfaktor bzw. *Clusterfaktor* obwohl es sich eigentlich einen Summanden handelt.

der Teilmenge C_i zu den anderen Teilmengen der Zerlegung C.

Diese Qualitätsfunktion trägt der Forderung Rechnung, dass unsere Teilmengen C_i in Bezug auf die Kantengewichte ω über einen möglichst starken inneren Zusammenhang und eine lose Kopplung zu anderen Teilmengen aus C verfügen sollten.

Mit Hilfe des naiven Algorithmus 2 lässt sich prinzipiell einfach eine Zerlegung unseres Graphen berechnen, die sowohl unseren Forderungen nach Konsistenz und Vollständigkeit als auch nach Optimalität der Zerlegung bezüglich f gerecht wird. Der Algorithmus berechnet einfach einen *Suchraum* aus allen möglichen Zerlegungen des Graphen und wählt daraus diejenige Zerlegung C_0 aus, für die f den besten Wert annimmt.

Algorithmus 2 Naive Berechnung einer optimalen Zerlegung C_0 von G

$C_0 := \{\{v_1\},\{v_2\},...,\{v_n\}\}$
$C := C_0$
$\mathcal{Q} := \{C\}$ - - *Arbeitsliste*
while $|\mathcal{Q}| > 0$ **do**
  - - *Bearbeite nächstes Element der Arbeitsliste*
  Wähle $C \in \mathcal{Q}$
  $Q := Q \backslash \{C\}$
  - - *Erzeuge neue Zerlegungen durch paarweises Verschmelzen*
  for all $C_1, C_2 \in C$ **do**
    $C' := \{\{C_1 \cup C_2\}, C \backslash \{C_1, C_2\}\}$
    if $f(G,C) > f(G,C_0)$ **then**
      $C_0 := C$
    end if
    - - *Füge neue Zerlegung zur Arbeitsliste hinzu*
    $\mathcal{Q} := \mathcal{Q} \cup \{C'\}$
  end for
end while
- - C_0 *ist beste Zerlegung von* G

Für die Praxis eignet sich dieser Algorithmus jedoch nicht, da er schon für kleine Graphen nicht mehr in akzeptabler Zeit zu einem Ergebnis kommt. Dies liegt daran, dass der Suchraum, der ja alle unterschiedlichen Möglichkeiten umfasst, die Menge V in Teilmengen zu zerlegen, sehr schnell wächst. Die Größe des Suchraums lässt sich über die Anzahl der Möglichkeiten berechnen, eine Menge von n Elementen in m Teilmengen zu zerlegen. Ist m gegeben, so gibt es $S(n,m)$ Möglichkeiten dies zu tun, wobei $S(n,m)$ die zweite *Sterlingsche Zahl* genannt wird und sich durch folgende Rekurrenz berechnen lässt (GRAHAM et al., 1992):

$$\begin{aligned} S(n,1) &= 1 \\ S(n,n) &= 1 \\ S(n,m) &= S(n-1,m-1)+mS(n-1,m), \qquad m \leq n \end{aligned} \tag{7.8}$$

In unserem Fall ist jedoch die Anzahl m der Teilmengen zunächst nicht bekannt, so dass die Größe des Suchraums

$$M(n) = \sum_{m=1}^{n} S(n,m) \tag{7.9}$$

ist.

Um das rasante Wachstum von $M(n)$ zu verdeutlichen, sind in Abbildung 7.1 die Funktionsgraphen von $M(n)$ und der Exponentialfunktion e^n im Vergleich dargestellt.

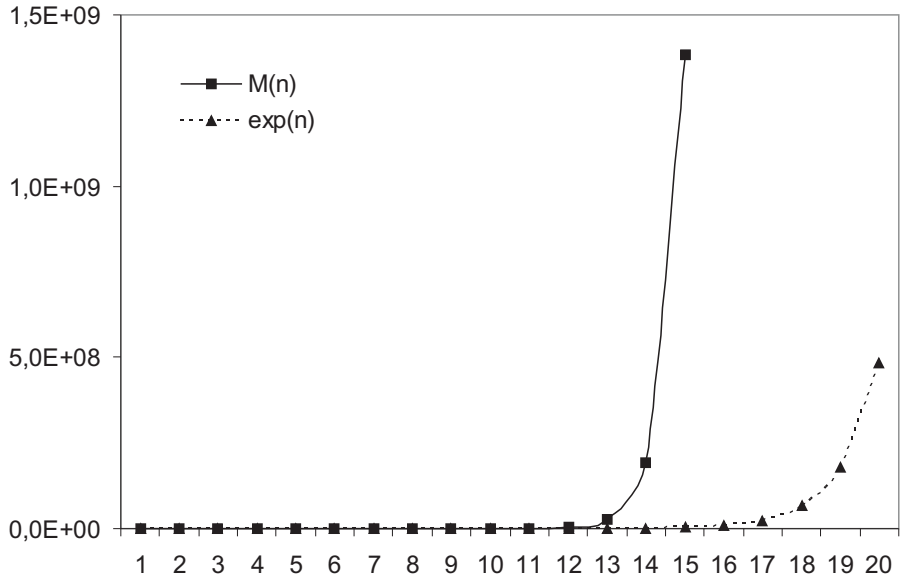


Abbildung 7.1: Größe des Suchraums für optimale Zerlegungen im Vergleich zur Exponentialfunktion

Leider ist das Problem, eine optimale Partition eines Graphen G zu finden, NP-hart (BRUCKER, 1977). Demzufolge ist uns heute kein Algorithmus bekannt, der eine optimale Zerlegung von G in polynomiellem Aufwand berechnen kann.

Für unseren Fall müssen wir daher die Forderung nach Optimalität lockern und mit einer "relativ guten" Näherungslösung leben. Zur Bewertung dieser Näherungslösungen leistet uns eine geeignet definierte Qualitätsfunktion f gute Dienste.

Wir benötigen nun also geeignete Verfahren, die uns solche Näherungslösungen berechnen. Hierzu können wir Verfahren zur *Ballungsanalyse* einsetzen, wie wir sie in Abschnitt 3.3 bereits kurz vorgestellt hatten. Ausgangspunkt der Ballungsanalyse ist der Abhängigkeitsgraph des Systems. Die Klassen des Systems in Form der Knoten des Abhängigkeitsgraphen sind dabei die zu gruppierenden Elemente. Die Abhängigkeiten zwischen Klassen, wie sie durch die Kanten des Graphen dargestellt sind, bilden die Ähnlichkeitsmaße, mit deren Hilfe die Ballungsanalyse gesteuert wird.

In Abschnitt 7.2 stellen wir dar, wie wir ein Strukturmodell gemäß Kapitel 5 Schritt für Schritt in einen Abhängigkeitsgraphen überführen können. Hierbei definieren wir auch geeignete Ähnlichkeitsmaße. Dann geben wir in den Abschnitten 7.3 und 7.4 zwei unterschiedliche Verfahren an, die eine Zerlegung des Graphen so vornehmen, dass wir gute Werte für die Qualitätsfunktion f erhalten.

7.2 Ähnlichkeitsmaße

Bevor wir Verfahren zur Ballungsanalyse zur Zerlegung eines Softwaresystems einsetzen können, müssen wir zunächst aus dem Modell der Implementierungsstrukturen des Systems einen Abhängigkeitsgraphen gewinnen, dessen Knoten – gemäß Definition 1 des vorangehenden Abschnitts – die Klassen des Systems repräsentieren. Bei der Definition der Kanten des Abhängigkeitsgraphen spielen *Ähnlichkeitsmaße* eine Schlüsselrolle, da sich aus ihnen die Gewichtung der Kanten ergibt.

Die Aufgabe der Ähnlichkeitsmaße besteht darin, messbar zu machen, wie *ähnlich* je zwei Klassen sind, so dass die Verfahren zur Ballungsanalyse durch Auswertung dieser Maße bestimmen können, ob die Klassen in ein gemeinsames Teilsystem platziert werden sollen – *ähnliche* Klassen werden nach Möglichkeit gemeinsam in ein Teilsystem platziert.[2]

Da wir Zerlegungen von Softwaresystemen berechnen wollen, die für das Verständnis der Systeme förderlich sind, müssen wir die Ähnlichkeitsmaße in geeigneter Form definieren. Grundsätzlich lassen wir uns von folgenden Prinzipien leiten:

- *Direkte Abhängigkeiten:* Wir betrachten Klassen als ähnlich, wenn zwischen ihnen viele Abhängigkeiten bestehen. Diese *direkten Abhängigkeiten* zwischen

[2] Für unsere Aufgabe, die darin besteht, aus der Sicht eines Softwareingenieurs zusammengehörende Klassen in Teilsysteme zu gruppieren, ist der Begriff *Ähnlichkeitsmaß* nicht sehr illustrativ. Der Begriff entstammt wie viele Arbeiten zur Ballungsanalyse dem *Data Mining*. Wir ziehen den dort etablierten Begriff des *Ähnlichkeitsmaßes* dem für unseren Anwendungsfall aussagekräftigeren Begriff *Abhängigkeitsmaß* vor.

Klassen können wir unmittelbar aus dem Modell der Implementierungsstrukturen gewinnen. Betrachten wir die Beziehungen unseres Metamodells für die Implementierungsstrukturen (siehe Abschnitt 5.2 bzw. Anhang A), so stellen wir fest, dass es unterschiedliche Typen direkter Abhängigkeiten zwischen Klassen gibt: Vererbung, Aggregation und Assoziation. Letztere Abhängigkeiten entstehen durch Methodenaufrufe und Attributzugriffe sowie durch Typdeklarationen in Methoden (Methodenparameter, Rückgabetypen und lokale Variablen).

- *Indirekte Abhängigkeiten:* Werden Klassen stets zusammen verwendet, so besteht zwischen ihnen offensichtlich ein Zusammenhang. Es bietet sich daher an, diese Klassen in ein gemeinsames Teilsystem zu platzieren. Aus diesem Grund betrachten wir auch solche Klassen als ähnlich. Wir sprechen in diesem Zusammenhang auch von *indirekten Abhängigkeiten*.

Die Definition geeigneter Ähnlichkeitsmaße, welche die direkten Abhängigkeiten zwischen Klassen berücksichtigen, ist Gegenstand der Abschnitte 7.2.1, 7.2.2 und 7.2.3. Die Definition von Ähnlichkeitsmaßen für die indirekten Abhängigkeiten erfolgt in Abschnitt 7.2.4.

Auf diese Weise entstehen zwischen je zwei Klassen eine Vielzahl von Einzelmaßen, die wir in Abschnitt 7.2.5 in ein einheitliches Maß zusammenführen, aus dem sich dann die Kanten und Gewichte des Abhängigkeitsgraphen $G = (V, E, \omega)$ ergeben.[3]

Bei der Definition der direkten Ähnlichkeitsmaße ist stets auch der *Kontext* der Abhängigkeiten von Interesse. Dieser ergibt sich durch die Auswertung der Klassifikation der Bauteile des Systems durch die Mustersuche, wie wir sie in Kapitel 6 vorgestellt haben. Abbildung 7.2 zeigt einige solcher Kontexte. Abbildung 7.2(a) ist beispielsweise zu entnehmen, dass Aufrufe an eine zuvor identifizierte Fassadenklasse niedriger gewichtet werden, als die Aufrufe von der Fassadenklasse zu den Delagationszielen der Fassadenklasse (siehe dazu auch Abschnitt 6.4).

7.2.1 Vererbung

Ist eine Klasse c_1 von einer Klasse c_2 abgeleitet, so besteht zwischen c_1 und c_2 eine Abhängigkeit, die wir durch das Ähnlichkeitsmaß $\omega_{Ver}(c_1, c_2)$ erfassen. Dieses Maß berücksichtigt unterschiedliche Kontexte für die Verwendung einer Vererbungsbeziehung.

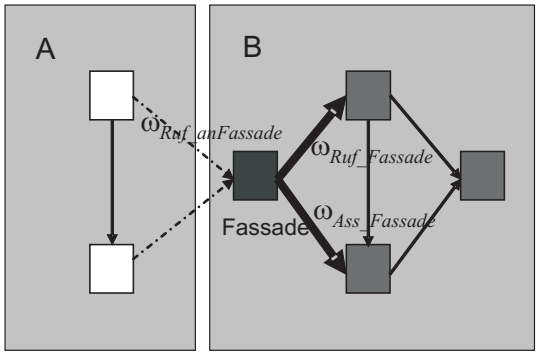
Folgende Kontexte sind von besonderer Bedeutung:

- Die Vererbungsbeziehung ist Teil eines Entwurfsmusters. Gehört die Vererbungsbeziehung zur Struktur der Entwurfsmuster *Proxy* oder *Kompositum*, dann können

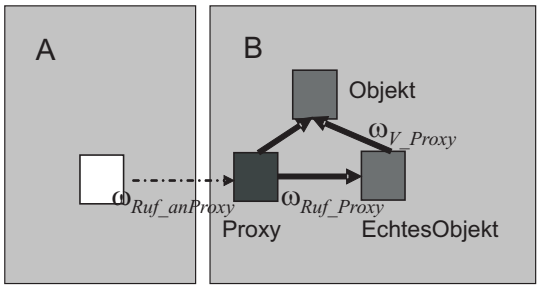
[3] Wenn wir die Einzelabhängigkeiten direkt in den Abhängigkeitsgraphen aufnehmen würden, so würde dieser die Form eines *Multigraphen* annehmen. Ein Multigraph erlaubt das Vorhandensein mehrerer Kanten zwischen je zwei Knoten. Abbildung 7.2 zeigt Ausschnitte eines solchen Multigraphen.

Ähnlichkeitsmaß	**Beschreibung**
$\omega_{Ver}(c_1, c_2)$	Abhängigkeit durch eine *Vererbungsbeziehung* zwischen den Klassen c_1 und c_2
$\omega_{Agg}(c_1, c_2)$	Maß für die Abhängigkeit, die durch eine *Aggregation* zwischen den Klassen c_1 und c_2 entsteht
$\omega_{Typ}(c_1, c_2)$	Maß für die Abhängigkeiten, die durch *Typdeklarationen* des Typs c_2 (Parameter, lokale Variablen, Rückgabetypen) in Methoden der Klasse c_1 entstehen
$\omega_{Ruf}(c_1, c_2)$	Maß für die Abhängigkeiten, die durch *Aufrufe* von Methoden der Klasse c_2 in den Methodenrümpfen der Klasse c_1 entstehen
$\omega_{Att}(c_1, c_2)$	Maß für die Abhängigkeiten, die durch *Zugriffe auf Attribute* der Klasse c_2 in Methoden der Klasse c_1 entstehen
$\omega_{Ind}(c_1, c_2)$	Maß für die *indirekte* Abhängigkeit, die dadurch entsteht, dass c_1 und c_2 von Kunden gemeinsam verwendet wird
$\omega(c_1, c_2)$	Gesamtmaß für alle Abhängigkeiten zwischen den Klassen c_1 und c_2, welches alle obenstehenden Einzelmaße zusammenfasst

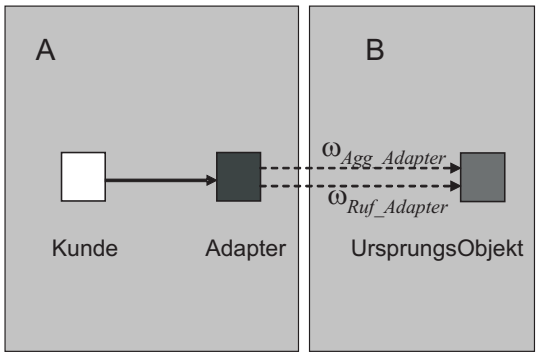
Tabelle 7.1: Übersicht über die Ähnlichkeitsmaße



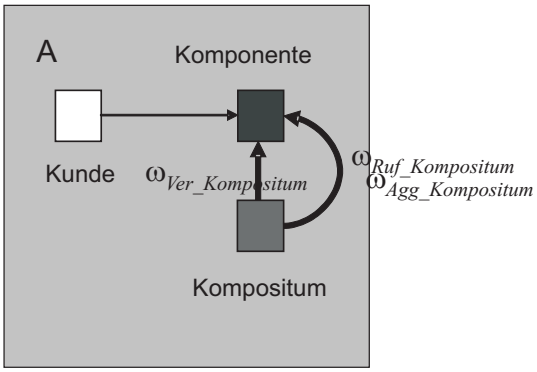
(a) Fassade



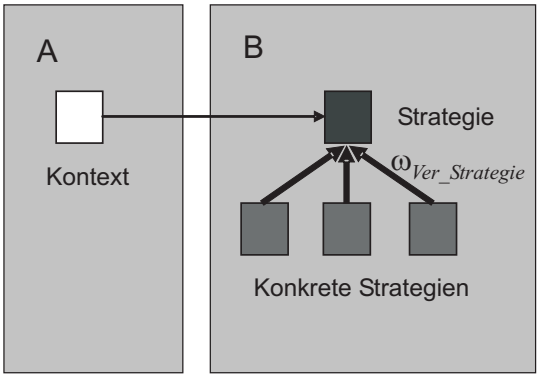
(b) Proxy



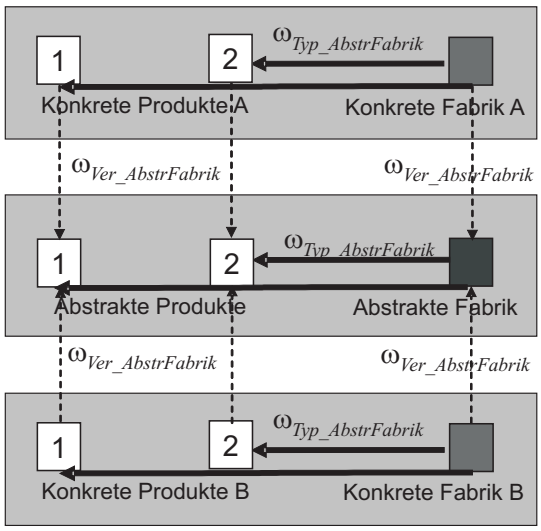
(c) Adapter



(d) Kompositum



(e) Strategie



(f) Abstrakte Fabrik

Abbildung 7.2: Gewichtung der verschiedenen Abhängigkeiten bei Entwurfsmustern

wir dies – wie bereits in Abschnitt 6.4 diskutiert – als Fingerzeig dafür verwenden, dass die beiden Klassen c_1 und c_2 gemeinsam in ein Teilsystem platziert werden sollten. Um dies zu erreichen, benötigen wir für die entsprechende Kante zwischen c_1 und c_2 im Abhängigkeitsgraphen hohe Gewichte. Wie in Abbildung 7.2 dargestellt, weisen wir $\omega_{Ver}(c_1, c_2)$ daher große Konstanten ω_{Ver_Proxy} bzw. $\omega_{Ver\_Kompositum}$ zu.

Gehört die Vererbungsbeziehung zum Entwurfsmuster *Strategie*, so können wir für $\omega_{Ver}(c_1, c_2)$ einen großen Wert wählen, so dass die abstrakte Strategie und alle davon abgeleiteten konkreten Strategien gemeinsam in ein Teilsystem platziert werden. Alternativ können wir auch sehr kleine Werte wählen, so dass die konkreten Strategien in ein eigenes Teilsystem abgetrennt werden.

Einen niedrigen Wert $\omega_{Ver\_AbstrFabrik}$ wählen wir für $\omega_{Ver}(c_1, c_2)$, wenn die Vererbungsbeziehung zwischen c_1 und c_2 innerhalb des Musters *Abstrakte Fabrik* auftritt, so dass die einzelnen Fabriken (zusammen mit den Produkten, für deren Erzeugung sie verantwortlich sind) in separate Teilsysteme platziert werden können.

- Die Vererbungsbeziehung dient zur *Implementierung* einer Schnittstelle. In diesem Fall enthält die Klasse c_1 einen hohen Anteil von Methoden, die abstrakte Methoden der Oberklasse c_2 implementieren. In diesem Fall gehen wir davon aus, dass c_1 und c_2 zusammen in ein Teilsystem platziert werden können, so dass wir dem Ähnlichkeitsmaß $\omega_{Ver}(c_1, c_2)$ einen hohen Wert $\omega_{Ver\_Implementierung}$ zuweisen können.

- Im Regelfall – also wenn keine der obenstehenden Kontexte zutreffen – drückt eine Vererbungsbeziehung eine *Spezialisierungsbeziehung* zwischen c_1 und c_2 aus. Auch in diesem Fall gehen wir davon aus, dass wir c_1 und c_2 in ein gemeinsames Teilsystem platzieren können und verwenden ebenfalls einen hohen Wert $\omega_{Ver\_Spezialisierung}$ für $\omega_{Ver}(c_1, c_2)$.

  Eine Ausnahme hiervon machen wir, wenn c_2 Teil eines *Rahmenwerks* ist. In diesem Fall wollen wir Rahmenwerk und Anwendungscode (also den Code zur Instanzierung bzw. Anpassung des Rahmenwerks) voneinander trennen. Daher weisen wir $\omega_{Ver}(c_1, c_2)$ einen niedrigen Wert $\omega_{Ver\_Rahmenwerk}$ zu. Wir erkennen diesen Kontext beispielsweise daran, dass c_2 *Schablonenmethoden* (siehe Abschnitt 6.2) enthält.

Wir können diese kontextabhängige Bewertung von Abhängigkeiten mit Hilfe der Klassifikation der Bauteile des Systems aus Kapitel 6 leicht in Form eines Regelwerks angeben.

Es gilt: $\omega_{Ver}(c_1, c_2) = 0$, falls c_1 nicht direkt von c_2 abgeleitet ist. Erbt c_1 von c_2, so gilt:

Kontext	Konstante	Gewichtung[5]
in *Proxy*	ω_{Ver_Proxy}	●●●
in *Kompositum*	$\omega_{Ver\_Kompositum}$	●●●
in *Strategie*	$\omega_{Ver\_Strategie}$	●●●
in *Abstrakter Fabrik*	$\omega_{Ver\_AbstrFabrik}$	●
Implementierung einer Schnittstelle	$\omega_{Ver\_Implementierung}$	●●
Spezialisierung	$\omega_{Ver\_Spezialisierung}$	●●
Anpassung einer Klasse eines Rahmenwerk	$\omega_{Ver\_Rahmenwerk}$	●

Tabelle 7.2: Kontexte bei Vererbungsbeziehungen

$$\omega_{Ver}(c_1,c_2) = \begin{cases} \omega_{Ver\_Proxy}, & proxy(c_1,x,c_2) \\ \omega_{Ver\_Kompositum}, & kompositum(c_1,c_2) \\ \omega_{Ver\_Strategie}, & strategie(c_1,c_2) \\ \omega_{Ver\_AbstrFabrik}, & c_1 \in L \wedge abstrakteFabrik(c_2,L) \\ \omega_{Ver\_Implementierung}, & \frac{|\{m|hatMethode(c_2,m) \wedge istAbstrakt(m)\}|}{NOM(c_2)} \geq \xi_{minAbstr} \\ \omega_{Ver\_Rahmenwerk}, & hatMethode(c_2,m) \\ & \quad \wedge istSchablonenmethode(m) \\ \omega_{Ver\_Spezialisierung}, & \text{sonst} \end{cases} \tag{7.10}$$

$\xi_{minAbstr}$ steht dabei für den Mindestanteil an abstrakten Methoden, den eine Klasse haben muss, damit wir davon ausgehen können, dass sie eine Schnittstelle vorgibt. Das Softwaremaß $NOM(c)$ [4] gibt die Anzahl der Methoden an, die c enthält.

Tabelle 7.2 stellt die Kontexte und die zugehörige Gewichtung der Vererbungsbeziehung schematisch zusammen. Eine konkrete, sinnvolle Belegung der entsprechenden Konstanten geben wir in Anhang B.2 an. Die Belegung dort stimmt mit jener überein, die wir für den Nachweis der Tragfähigkeit unseres Ansatzes in Kapitel 8 verwendet haben.

7.2.2 Aggregation

Eine Aggregationsbeziehung zwischen zwei Klassen c_1 und c_2 besteht dann, wenn c_1 ein Attribut vom Typ c_2 besitzt. Die Aggregationsbeziehung stellt eine Abhängigkeit dar, die wir mit dem Ähnlichkeitsmaß $\omega_{Aggr}(c_1,c_2)$ erfassen wollen.

Das Ähnlichkeitsmaß $\omega_{Aggr}(c_1,c_2)$ ist – wie das Ähnlichkeitsmaß für die Vererbungsbeziehung – kontextabhängig. Folgende Kontexte sind von besonderem Interesse:

[4] engl. *number of methods*

[5] Dabei steht ●●●für hohe Werte, ●●für mittelgroße Werte und ●für eher kleinere Werte.

- Die Aggregationsbeziehung tritt innerhalb einer Entwurfsmusterstruktur auf. Gemäß unseren Überlegungen aus Abschnitt 6.4 gehen wir dann folgendermaßen vor: Besteht die Aggregationsbeziehung zwischen einer *Fassadenklasse* und den Klassen, welche die Fassade verbirgt, so erhält die Beziehung ein hohes Gewicht. Ebenso verfahren wir, wenn die Aggregationsbeziehung im Entwurfsmuster *Proxy* zwischen Stellvertreterobjekt und Ursprungsobjekt oder innerhalb des Entwurfsmusters *Kompositum* auftritt. Aggregationen von Fassadenklassen und Proxy-Klassen erhalten zudem ein geringes Gewicht, um die gewünschten Effekte bei der Teilsystembildung noch zu verstärken (siehe dazu Abbildung 7.2). Im Falle des Musters *Adapter* verwenden wir für die Aggreagation zwischen Adapterklasse und anzupassender Ausgangsklasse ein geringes Gewicht.

- Die Aggregationsbeziehung drückt eine *Komposition* aus. In diesem Fall kann das aggregierte Objekt vom Typ c_2 nicht ohne das umfassende Objekt vom Typ c_1 existieren – es ist ein untrennbarer Bestandteil des umfassenden Objektes. Aus diesem Grund erzwingen wir durch hohe Werte für das Ähnlichkeitsmaß, dass die beiden Klassen zusammen in ein Teilsystem platziert werden. Da in den meisten objektorientierten Programmiersprachen Kompositionsbeziehungen nicht durch Sprachmittel von gewöhnlichen Aggregationen zu unterscheiden sind, ist die Kompositionsbeziehung auch nicht unmittelbar Teil unseres Strukturmodells. Wir benötigen demnach eine Heuristik zur Erkennung von Kompositionsbeziehungen. Wir gehen davon aus, dass eine solche vorliegt, wenn die Implementierung der Klasse c_1 für die Erzeugung des aggregierten Objektes (also für die Initialisierung des Attrubutes vom Typ c_2) verantwortlich ist.

- In allen anderen Fällen nehmen wir eine neutrale Gewichtung der Abhängigkeit vor.

Daraus ergibt sich folgendes Regelwerk: Es gilt $\omega_{Agg}(c_1,c_2) = 0$, wenn zwischen den Klassen c_1 und c_2 keine Aggregation besteht. Ansonsten ist:

$$\omega_{Agg}(c_1,c_2) = \begin{cases} \omega_{Agg\_Fassade}, & fassade(c_1,L) \wedge c_2 \in L \\ \omega_{Agg\_Proxy}, & proxy(c_1,c_2,x) \\ \omega_{Agg\_Kompositum}, & kompositum(c_1,c_2) \\ \omega_{Agg\_zuFassade}, & fassade(c_2,x) \\ \omega_{Agg\_zuProxy}, & proxy(c_2,x,y) \\ \omega_{Agg\_Adapter}, & adapter(c_1,c_2) \\ \omega_{Agg\_Komposition}, & hatMethode(c_1,m) \wedge hatRumpf(m,r) \wedge ruft(r,k) \\ & \qquad \wedge hatMethode(c_2,k) \wedge istKonstruktor(k) \\ \omega_{Agg\_Standard}, & \text{sonst} \end{cases} \tag{7.11}$$

Kontext	Konstante	Gewichtung
in *Fassade*	$\omega_{Agg\_Fassade}$	●●●
in *Proxy*	ω_{Agg_Proxy}	●●●
in *Kompositum*	$\omega_{Agg\_Kompositum}$	●●●
zu *Fassade*	$\omega_{Agg\_zuFassade}$	●
zu *Proxy*	$\omega_{Agg\_zuProxy}$	●
in *Adapter*	$\omega_{Agg\_Adapter}$	●
Komposition	$\omega_{Agg\_Komposition}$	●●●
gewöhnliche Aggregation	$\omega_{Agg\_Standard}$	●●

Tabelle 7.3: Kontexte bei Aggregationsbeziehungen

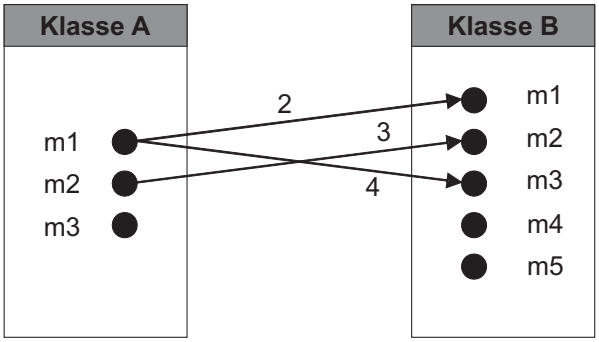
Tabelle 7.3 stellt die Kontexte und die zugehörige Gewichtung der Aggregationsbeziehung schematisch zusammen, die genau Belegung der Konstanten für die Gewichte findet sich in Anhang B.2.

7.2.3 Assoziationen: Aufrufe, Attributzugriffe und Typdeklarationen

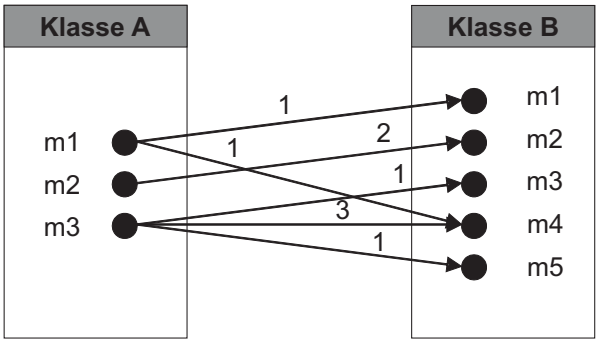
Assoziationsbeziehungen entstehen durch Methodenaufrufe und Variablenzugriffe sowie durch Typdeklarationen in Methoden (bei der Deklaration von Methodenparametern, Rückgabetypen und lokale Variablen). Methodenaufrufe sind für die Kooperation von Objekten verantwortlich, deshalb prägen sie die Abhängigkeiten zwischen den Klassen des Systems in besonderem Maße.

Methodenaufrufe. Gemäß unseres Strukturmodells (siehe Kapitel 5) besteht zwischen zwei Methoden m und n eine Aufrufbeziehung, wenn der Methodenrumpf von m einen Aufruf der Form $x.n()$ enthält. Ist m eine Methode der Klasse c_1 und c_2 der statische (deklarierte) Typ von x, so ergibt sich durch diesen Methodenaufruf eine statische Abhängigkeit zwischen den Klassen c_1 und c_2. Wir beachten, dass wir dabei Abhängigkeiten zwischen zwei Methoden auf Klassen übertragen haben. Zwischen den Methoden der beiden Klassen können zahlreiche weitere Aufrufbeziehungen bestehen. Wir benötigen daher ein Ähnlichkeitsmaß ω_{Ruf}, welches diesem Umstand in geeigneter Weise Rechnung trägt.

Wir beobachten zunächst, dass der Grad der Kopplung zwischen zwei Klassen durch Methodenaufrufe stark von der Lokalität der Aufrufe abhängt. In Abbildung 7.3 sind zwei unterschiedliche Fälle dargestellt, in denen jeweils zwei Klassen durch eine identische Anzahl von Methodenaufrufen miteinander verknüpft sind. Im Fall b) erscheint die Kopplung zwischen den beiden Klassen jedoch stärker, weil die Aufrufabhängigkeiten breiter unter den Methoden der Klassen c_1 und c_2 gestreut sind. Um dieser Beobachtung



(a) Enge Streuung



(b) Breite Streuung

Abbildung 7.3: Lokalität von Aufrufabhängigkeiten

Rechnung zu tragen, definieren wir die durch Methodenaufrufe entstehende Abhängigkeit wie folgt:

$$\omega_{Ruf}(c_1,c_2) = \frac{\omega'_{Ruf}(c_1,c_2)}{NOM(c_1)} \cdot \left(\frac{\#rufendeMeth(c_1,c_2)}{NOM(c_1)} + \frac{\#gerufeneMeth(c_1,c_2)}{NOM(c_2)} \right)$$

mit

$$\begin{aligned} \omega'_{Ruf}(c_1,c_2) &= \sum_{(r,n)\in R} \omega_{Ruf}(r,n) \\ R &= \{(r,n) \mid hatMethode(c_1,m) \wedge hatRumpf(m,r) \\ &\qquad \wedge hatMethode(c_2,n) \wedge ruft(r,n)\} \end{aligned} \tag{7.12}$$

$\omega_{Ruf}(r,n)$ ist dabei ein kontextabhängiges Maß für alle Aufrufe zwischen dem Methodenrumpf r und der Methode n, auf das wir noch genauer eingehen. $\#rufendeMethoden(c_1,c_2)$ steht für die Anzahl von Methoden in c_1, welche Aufrufe zu Methoden aus c_2 enthalten und $\#gerufeneMethoden(c_1,c_2)$ steht für die Anzahl von Methoden aus c_2, welche von c_1 aus aufgerufen werden. $NOM(c_1)$ bzw. $NOM(c_2)$ bezeichnen die Anzahl der Methoden in c_1 bzw. c_2. Die Menge R enthält die Paare aus Methoden von c_1 und c_2, zwischen denen eine Aufrufbeziehung besteht.

Abbildung 7.4 gibt eine intuitive Rechtfertigung für diese Formel: Die Kopplung zwischen c_1 und c_2 durch Methodenaufrufe ist proportional zum Flächeninhalt des hervorgehobenen Trapezes, welches bei entsprechender Anordnung der beteiligten Methoden ein gutes Maß für die Lokalität der Aufrufe zwischen c_1 und c_2 darstellt.

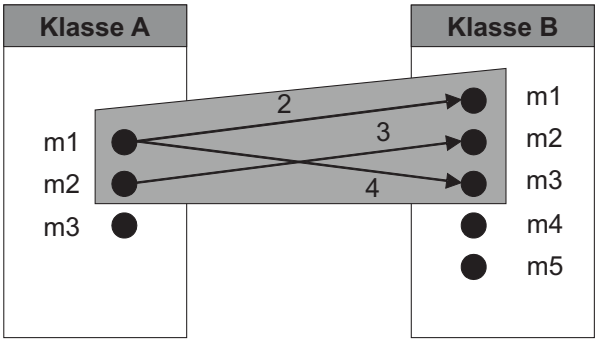


Abbildung 7.4: Intuitive Rechtfertigung der Metrik für Aufrufabhängigkeiten

Bei der Berechnung des Maßes $\omega_{Ruf}(r,n)$, welches die Stärke der Abhängigkeiten durch Aufrufe von Methode m (bzw. deren Rumpf r) zu n angibt, berücksichtigen wir folgende Kontexte:

- Die Aufrufe kommen innerhalb einer Entwurfsmusterstruktur vor. Dann gehen wir im Einklang mit unseren Überlegungen aus Abschnitt 6.4 wie folgt vor: Gehört m zu einer Fassadenklasse und n zu einer der dahinter verborgenen Implementierungsklassen oder sind die Aufrufe vom m zu n Teil des Kommunikationsprotokolls zwischen Stellvertreter und Ursprungsobjekt im Muster *Proxy* bzw. zwischen Kompositum und Komponente im Muster *Kompositum*, dann messen wir dem Maß $\omega_{Ruf}(m,n)$ ein hohes Gewicht zu. Ist m eine Methode eines *Adapters* und n eine Methode einer zu adaptierenden Klasse, dann messen wir dem Aufruf ein geringes Gewicht bei, weil wir davon ausgehen, dass die Adapterklasse der Verwendungsstelle zugeordnet werden soll.
- Die Aufrufe gehen *an* eine *Bibliotheksklasse* (siehe Abschnitt 6.3), einen Proxy oder eine Fassadenklasse. In diesem Fall gehen wir davon aus, dass der Aufruf zwischen m und n über eine Teilsystemgrenze verläuft. Dementsprechend verwenden wir kleine Gewichte $\omega_{Ruf\_Bibliothek}$, $\omega_{Ruf\_anProxy}$ und $\omega_{Ruf\_anFassade}$.
- Die Aufrufe gehen an eine *Klassenmethode*. Klassenmethoden sind häufig Hilfsroutinen, die wir ähnlich behandeln wollen wie Methoden aus Bibliotheksklassen. Wir verwenden daher für diese Aufrufe eher geringe Gewichte.
- Die Aufrufe stammen von einer *Delegationsmethode*. In diesem Fall können wir davon ausgehen, dass die beiden Klassen, zu denen m und n gehören, relativ eng miteinander kooperieren. Wir verwenden daher für diese Aufrufe hohe Gewichte.
- Trifft keiner der anderen Kontexte zu, dann verwenden wir ein neutrales Gewicht für die entsprechenden Aufrufe.

Daraus ergeben sich folgende Regeln:

$$\omega_{Ruf}(r,n) = \#Aufrufe(r,n) \cdot \begin{cases} \omega_{Ruf\_Fassade}, & fassade(c_1,L) \wedge c_2 \in L \\ \omega_{Ruf\_Proxy}, & proxy(c_1,c_2,x) \\ \omega_{Ruf\_Kompositum}, & kompositum(c_1,c_2) \\ \omega_{Ruf\_Adapter}, & adapter(c_1,c_2) \\ \omega_{Ruf\_anFassade}, & fassade(c_2,x) \\ \omega_{Ruf\_Bibliothek}, & istBibliotheksklasse(c_2) \\ \omega_{Ruf\_anProxy}, & proxy(c_2,x,y) \\ \omega_{Ruf\_Klassenmethode}, & istKlassenmethode(c_2) \\ \omega_{Ruf\_Delegation}, & delegierend(c_1) \\ \omega_{Ruf\_Standard}, & \text{sonst} \end{cases} \tag{7.13}$$

c_1 ist dabei die Klasse, die m enthält, und c_2 die Klasse, die n enthält.

Tabelle 7.4 stellt die Kontexte und die zugehörige Gewichtung der Aufrufbeziehungen schematisch zusammen. Die genaue Belegung der Konstanten für die Gewichte findet sich in Anhang B.2.

Kontext	Konstante	Gewichtung
innerhalb *Fassade*	$\omega_{Ruf\_Fassade}$	●●●
innerhalb *Proxy*	ω_{Ruf_Proxy}	●●●
innerhalb *Kompositum*	$\omega_{Ruf\_Kompositum}$	●●●
innerhalb *Adapter*	$\omega_{Ruf\_Adapter}$	●
an *Fassade*	$\omega_{Ruf\_anFassade}$	●
an Bibliotheksklasse	$\omega_{Ruf\_Bibliothek}$	●
an *Proxy*	$\omega_{Ruf\_anProxy}$	●
an Klassenmethode	$\omega_{Ruf\_Klassenmethode}$	●
von Delegationsmethode	$\omega_{Ruf\_Delegation}$	●●●
gewöhnlicher Aufruf	$\omega_{Ruf\_Standard}$	●●

Tabelle 7.4: Kontexte bei Aufrufbeziehungen

Attributnutzung. Neben Methodenaufrufen können auch Zugriffe auf Attribute Abhängigkeiten zwischen Klassen verursachen. In Systemen mit durchdachter Implementierung spielen diese eine weitaus geringere Rolle als Methodenaufrufe, da Attributzugriffe die Kapselung der Interna einer Klasse aufweichen.

Unser Strukturmodell (siehe Kapitel 5) modelliert Variablenzugriffe als Beziehung zwischen Methoden (bzw. deren Rumpf) und Variablen. Daraus können wir eine Beziehung auf Klassenebene ableiten: Sei m eine Methode der Klasse c_1, in der auf ein Attribut x aus der Klasse c_2 zugegriffen wird, dann besteht eine Attributzugriffsbeziehung auf Klassenebene zwischen der Klasse c_1 und der Klasse c_2.

Wir definieren für diese Abhängigkeiten das Ähnlichkeitsmaß ω_{Att}. Dazu gehen wir vor wie bei der Definition des Ähnlichkeitsmaßes für Methodenaufrufe.

$$\omega_{Att}(c_1,c_2) = \frac{\omega'_{Att}(c_1,c_2)}{NOM(c_1)} \cdot \left(\frac{\#zugreifendeMeth(c_1,c_2)}{NOM(c_1)} + \frac{\#verwendeteAttr(c_1,c_2)}{NOA(c_2)} \right)$$

mit

$$\begin{aligned} \omega'_{Att}(c_1,c_2) &= \sum_{(r,x)\in R} \omega_{Att}(r,x) \\ R &= \{(r,x) \mid hatMethode(c_1,m) \wedge hatRumpf(m,r) \\ &\qquad \wedge hatMethode(c_2,x) \wedge greiftZu(r,x)\} \end{aligned} \tag{7.14}$$

$\omega_{Att}(r,x)$ ist dabei ein kontextabhängiges Maß für alle Nutzungen des Attributes x, die im Rumpf r einer Methode m vorkommen. Auf seine Berechnung gehen wir im Anschluss noch genauer ein. $\#zugreifendeMethoden(c_1,c_2)$ steht für die Anzahl von Methoden in

c_1, welche Zugriffe auf Attribute von c_2 enthalten und $\#verwendeteAttr(c_1,c_2)$ steht für die Anzahl von Attributen aus c_2, welche von c_1 aus verwendet werden. $NOM(c_1)$ bezeichnet die Anzahl der Methoden in c_1, $NOA(c_2)$[6] bezeichnet die Anzahl der Attribute in c_2.

Bei der Ermittlung des kontextabhängigen Maßes $\omega_{Att}(r,x)$ berücksichtigen wir nur wenige unterschiedliche Kontexte:

- Attribute aus Bibliotheken und Klassenattribute. Falls x ein Klassenattribut ist, so gehen wir davon aus, dass die Abhängigkeit weniger bedeutend ist, da solche Zugriffe nicht wie bei der Nutzung gewöhnlicher Attribute den *internen* Zustand eines gekapselten Objektes betreffen. Häufig findet sich diese Art der Attributnutzung auch in Bibliotheksklassen zur Vereinbarung von Konstanten. Ist die Klasse c_2, die das Attribut x enthält, eine Bibliotheksklasse, so veranschlagen wir für die entsprechende Abhängigkeit nur ein sehr geringes Gewicht.
- Gewöhnliche Attributnutzungen. Trifft keiner der anderen Kontexte zu, so messen wir der Attributnutzung ein etwas stärkeres Gewicht bei.

Das entsprechende Regelwerk lautet:

$$\omega_{Att}(r,x) = \#Zugriffe(r,n) \cdot \begin{cases} \omega_{Ruf\_Fassade}, & fassade(c_1,L) \wedge c_2 \in L \\ \omega_{Ruf\_Proxy}, & proxy(c_1,c_2,x) \\ \omega_{Ruf\_Kompositum}, & kompositum(c_1,c_2) \\ \omega_{Ruf\_Adapter}, & adapter(c_1,c_2) \\ \omega_{Ruf\_anFassade}, & fassade(c_2,x) \\ \omega_{Ruf\_Bibliothek}, & istBibliotheksklasse(c_2) \\ \omega_{Ruf\_anProxy}, & proxy(c_2,x,y) \\ \omega_{Ruf\_Klassenmethode}, & istKlassenmethode(c_2) \\ \omega_{Ruf\_Delegation}, & delegierend(c_1) \\ \omega_{Ruf\_Standard}, & \text{sonst} \end{cases} \tag{7.15}$$

c_1 steht dabei für die Klasse, zu der die Methode m bzw. deren Rumpf r gehört, c_2 für die Klasse, in der x definiert ist.

Verfeinern ließe sich diese Betrachtung noch durch eine Unterscheidung von *Lesezugriffen* und *Schreibzugriffen*. Schreibzugriffen könnten wir dann höhere Gewichte zuordnen, da diese den inneren Zustand des Objekts, auf dessen Attribut zugegriffen wird, manipulieren können.

Tabelle 7.5 stellt die Kontexte und die zugehörige Gewichtung der Attributnutzungen schematisch zusammen. Die genaue Belegung der Konstanten für die Gewichte findet sich in Anhang B.2.

[6] engl. *number of attributes*

Kontext	Konstante	Gewichtung
bei Klassenattributen	$\omega_{Att\_Klassenattr}$	•
bei Attributen aus Bibliotheksklassen	$\omega_{Att\_Bibliothek}$	•
gewöhnlicher Attributnutzung	$\omega_{Att\_Standard}$	••

Tabelle 7.5: Kontexte bei Attributzugriffen

Typdeklarationen. Weitere Abhängigkeiten zwischen zwei Klassen c_1 und c_2 entstehen durch Typdeklarationen. Solche Typdeklarationen können in Form von Methodenparametern, Rückgabetypen von Methoden oder bei der Vereinbarung lokaler Variablen auftreten. Eine Abhängigkeit durch Typdeklarationen entsteht, wenn eine Methode m aus c_1 eine Deklaration des Typs c_2 enthält, c_2 also als Typ eines formalen Parameters von m, einer lokalen Variable oder des Rückgabewertes von m auftritt.

Auch hier berücksichtigen wir wieder die Verteilung der entsprechenden Abhängigkeiten über die Methoden aus c_1.

$$\omega_{Typ}(c_1,c_2) = \frac{\omega'_{Typ}(c_1,c_2)}{NOM(c_1)} \cdot \frac{\#deklarierendeMeth(c_1,c_2)}{NOM(c_1)}$$

mit

$$\begin{aligned}
\omega'_{Typ}(c_1,c_2) &= \sum_{x \in V} \omega_{Att}(x,c_2) \\
V &= \{x \mid hatMethode(c_1,m) \wedge hatRumpf(m,r) \\
&\qquad \wedge (hatParameter(m,x) \vee hatVariable(m,x)) \\
&\qquad \wedge hatTyp(x,c_2)\} \\
&\cup \{m \mid hatMethode(c_1,m) \wedge hatRueckgabetyp(m,c_2)\} \qquad (7.16)
\end{aligned}$$

$\omega_{Typ}(x,c_2)$ ist dabei ein kontextabhängiges Maß für die Typabhängigkeit zu c_2 die durch die Variable x bzw. den Rückgabewert der Methode x zustande kommt. $\#deklarierendeMeth(c_1,c_2)$ steht für die Anzahl von Methoden in c_1, welche Deklarationen des Typs c_2 enthalten. $NOM(c_1)$ bezeichnet die Gesamtzahl der Methoden in c_1.

Bei der Berechnung von $\omega_{Typ}(x,c_2)$ berücksichtigen wir folgende Kontexte:

- Die Abhängigkeit kommt durch den Rückgabewert der Methode x aus der Klasse c_1 zustande. Ist x eine Fabrikmethode und c_1 eine abstrakte bzw. konkrekte Fabrik, so ist c_2 ein Produkt, welches zur Fabrik c_1 gehört. Gemäß unserer Überlegungen aus Abschnitt 6.4 wollen wir Fabriken und Produkte zusammen in ein Teilsystem

Kontext	Konstante	Gewichtung
Rückgabetypen in Fabrik	$\omega_{Typ\_AbstrFabrik}$	●●●
Rückgabetypen	$\omega_{Typ\_Rueckgabetyp}$	●●
Formale Parameter	$\omega_{Typ\_Parameter}$	●●
Lokale Variablen	$\omega_{Typ\_Variable}$	●

Tabelle 7.6: Kontexte bei Typdeklarationen

platzieren (vgl. Abbildung 7.2(f)). Aus diesem Grund verwenden wir in diesem Fall ein größeres Gewicht als gewöhnliche Rückgabetypen.

- Die Abhängigkeit entsteht durch formale Parameter oder lokale Variablen. Ist x ein formaler Parameter einer Methode in c_1, so verwenden wir hierfür etwas höhere Gewichte als für den Fall, dass x eine lokale Variable einer Methode ist.

$$\omega_{Typ}(x,c_2) = \begin{cases} \omega_{Typ\_AbstrFabrik}, & abstrakte(a,L) \wedge (c_1 \in L \vee c_1 = a) \\ & \quad \wedge hatMethode(c_1) \wedge istRueckgabetyp(x,c_2) \\ & \quad \wedge istFabrikmethode(x) \\ \omega_{Typ\_Rueckgabetyp}, & hatMethode(c_1,x) \wedge istRueckgabetyp(x,c_2) \\ \omega_{Typ\_Parameter}, & hatMethode(c_1,m) \\ & \quad \wedge hatParameter(m,x) \wedge hatTyp(x,c_2) \\ \omega_{Typ\_Variable}, & hatMethode(c_1,m) \\ & \quad \wedge hatVariable(m,x) \wedge hatTyp(x,c_2) \\ 0, & \text{sonst} \end{cases} \tag{7.17}$$

Tabelle 7.6 stellt die Kontexte und die zugehörige Gewichtung der Abhängigkeiten durch Typdeklarationen schematisch zusammen. Die genaue Belegung der Konstanten für die Gewichte findet sich in Anhang B.2.

7.2.4 Indirekte Abhängigkeiten

Indirekte Abhängigkeiten ergeben sich aus der gemeinsamen Verwendung von Elementen des Systems in anderen Teilen des Systems. Dieses Maß ist insbesondere zur korrekten Zerlegung von Bibliotheken nützlich, da für diese das *Common Reuse Principle* (MARTIN, 1996) gilt. Es besagt, dass Klassen, die gemeinsam verwendet werden können oder gar müssen, in ein gemeinsames Teilsystem platziert werden sollten, damit Nutzern der Bibliothek die Verwendung erleichtert wird.

Die *Java*-Standardbibliothek beispielsweise verfolgt dieses Prinzip konsequent. So sind dort Behälterklassen und die zugehörigen Iteratoren in einem gemeinsamen Paket untergebracht.

Zur Erfassung der indirekten Abhängigkeiten zwischen Klassen berücksichtigen wir lediglich Methodenaufrufe. Wenn aus einem Methodenrumpf einer Klasse heraus Aufrufe zu Methoden anderer Klassen erfolgen, so entstehen zwischen je zwei Klassen aus der Menge der aufgerufenen Klassen indirekte Abhängigkeiten.

Abbildung 7.5 zeigt beispielsweise wie eine indirekte Abhängigkeit zwischen zwei Klassen *C* und *D* ensteht, wenn die Methoden aus zwei anderen Klassen *A* und *B* Methoden aus *C* und *D* aufrufen.

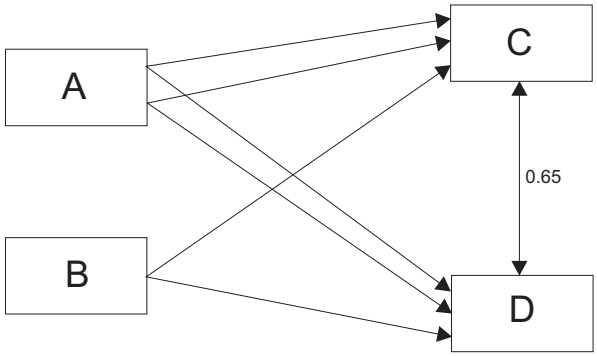


Abbildung 7.5: Indirekte Abhängigkeiten zwischen zwei Klassen *C* und *D*

Wir messen diese Abhängigkeiten mit Hilfe des folgenden Maßes:

$$\omega_{Ind}(c_1,c_2) = \sum_{x\in\{x|Klasse(x)\}} \frac{|M|}{NOM(x)}, \tag{7.18}$$

wobei

$$\begin{aligned} M \quad = \quad & \{m|hatMethode(x,m) \wedge ruft(m,m') \wedge hatMethode(c_1,m') \\ & \wedge ruft(m,m'') \wedge hatMethode(c_1,m'')\} \end{aligned} \tag{7.19}$$

7.2.5 Berechnung eines einheitlichen Ähnlichkeitsmaßes

Wir haben in den vorangehenden Abschnitten eine Reihe von Ähnlichkeitsmaßen zwischen je zwei Klassen des Systems definiert (siehe Tabelle 7.1). Wir führen diese nun über eine gewichtete Summe in ein einheitliches Ähnlichkeitsmaß zusammen. Dieses Maß verwenden wir dann als Gewichtungsfunktion ω für die Abhängigkeitsgraphen von Softwaresystemen.

$$\begin{aligned}\omega(c_1,c_2) &= c_{Ver}\cdot\omega_{Ver}(c_1,c_2)+c_{Agg}\cdot\omega_{Agg}(c_1,c_2)+c_{Ruf}\cdot\omega_{Ruf}(c_1,c_2)\\ &+ c_{Att}\cdot\omega_{Att}(c_1,c_2)+c_{Typ}\cdot\omega_{Typ}(c_1,c_2)\end{aligned} \tag{7.20}$$

Mit Hilfe der Konstanten $c_{Ver},\ldots,c_{Typ}$ können wir die Gewichtung der Einzelmaße anpassen. In Anhang B.2 geben wir geeignete Werte für diese Konstanten an.

Algorithmen zur Ballungsanalyse setzen häufig einen *ungerichteten* Graphen voraus. Wir können unseren gerichteten Abhängigkeitsgraphen $G=(V,E,\omega)$ leicht in einen ungerichteten Graphen überführen. Die Kantengewichte des ungerichteten Graphen berechnen wir dann nach folgender Formel:

$$\bar{\omega}(c_1,c_2)=\max(\omega(c_1,c_2),\omega(c_2,c_1)) \tag{7.21}$$

7.3 *MMST* – ein graphbasierter Algorithmus zur Ballungsanalyse

Wir haben in den voranstehenden Abschnitten gesehen, wie wir ein Softwaresystem mit Hilfe eines Abhängigkeitsgraphen $G=(V,E,\omega)$ darstellen können. Wir können somit auf die Problemstellung zurückkommen, die wir zu Beginn dieses Kapitels dargestellt haben: Wir wollen eine Zerlegung C des Graphen G im Sinne der Definition aus Abschnitt 7.1 berechnen, so dass eine Qualitätsfunktion f – wie beispielsweise die aus Gleichung 7.4 – möglichst gute Werte aufweist.

Wir wollen dazu zunächst einen deterministischen Algorithmus vorstellen, der den maximalen Spannbaum des Graphen als Basis für die Ballungsanalyse heranzieht.

Definition 2 *Sei $G=(V,E,\omega)$ ein* zusammenhängender*, ungerichteter Graph*[7]*. Der* Teilgraph $B=(V,E')$, $E'\subseteq E$ *heißt* Spannbaum*, wenn B zyklenfrei und ebenfalls zusammenhängend ist.*

B ist ein maximaler Spannbaum*, wenn er ein maximales* Gewicht

$$\omega(B)=\sum_{e\in E'}\omega(e) \tag{7.22}$$

besitzt. Das heißt: Für jeden weiteren Spannbaum B' von G gilt: $\omega(B)\geq\omega(B')$.

[7] Der ungerichtete Graph G ist genau dann zusammenhängend, wenn in G je zwei beliebige verschiedene Knoten v_1 und v_2 (evtl. über andere Knoten) miteinander verbunden sind (d.h. es existiert eine Kette zwischen i und j).

ZAHN (1971) und HARTIGAN (1975) zeigen, wie wir mit Hilfe des maximalen Spannbaumes eines Graphen eine günstige Zerlegung des Graphen berechnen können: Wir berechnen zunächst den maximalen Spannbaum und entfernen aus diesem dann alle Kanten, deren Gewicht wir als zu gering betrachten. Aus den dadurch verbleibenden Zusammenhangskomponenten ergibt sich dann eine Zerlegung des Graphen.[8] Wir stellen zunächst fest, dass diese Zerlegung die Forderungen aus Abschnitt 7.1 nach Konsistenz und Vollständigkeit erfüllt.

Wenn wir eine vernünftige Strategie angeben können, nach der wir die Kanten aus dem Spannbaum entfernen wollen, hoffen wir, dass wir auch gute Werte für unsere Qualitätsfunktion f erhalten. Da Kanten mit hohen Gewichten bevorzugt Teil des Spannbaumes sind, können wir durch gezieltes Löschen von "leichten" Kanten dafür sorgen, dass sich die Kanten mit hohen Gewichten innerhalb von Zusammenhangskomponenten befinden.

Als Strategie zum Entfernen von Kanten aus dem Spannbaum schlägt ZAHN (1971) folgende Strategie vor: eine Kante $e = (v_1, v_2)$ wird dann entfernt, wenn ihr Gewicht $\omega(e)$ kleiner ist, als das mit einem *Toleranzfaktor* α multiplizierte Durchschnittsgewicht der inzidenten Kanten zu $v1$ und $v2$.

Im allgemeinen berechnet dieses Verfahren recht brauchbare Zerlegungen des Graphen (TRIFU, 2001). Es favorisiert allerdings Zerlegungen, die einige wenige sehr große Ballungen aufweisen.

Wir schlagen daher einen modifizierten Algorithmus vor, den wir *MMST* nennen wollen (Algorithmus 3, siehe auch BAUER und TRIFU (2004)). Der Algorithmus läuft in zwei Phasen ab.

Die erste Phase des Algorithmus ergibt sich aus dem Algorithmus von KRUSKAL (1956) zur Bestimmung des minimalen bzw. maximalen Spannbaums. Der Algorithmus verwendet eine *Union-Find-Datenstruktur* zur Repräsentation von Teilmengen des Graphen (OTTMANN und WIDMEYER, 1993). Eine *Union-Find*-Datenstruktur unterstützt folgende Operationen:

- $MakeSet(v)$ erzeugt eine neue Teilmenge, die mit dem Element v belegt wird.
- $Union(v, v')$ vereinigt die zwei Teilmengen, in denen sich die Elemente v und v' befinden.
- $Find(v)$ erlaubt es, zu einem beliebigen Element die Teilmenge zu bestimmen, zu der es gehört.

Union-Find-Datenstrukturen verwenden Bäume zur Speicherung der Elemente der Teilmengen, so dass die Operationen $Union$ und $Find$ besonders effizient ausgeführt werden

[8] ZAHN (1971) und HARTIGAN (1975) verwenden einen minimalen Spannbaum und entfernen "schwere Kanten" zur Bestimmung der Ballungen, da die Kantengewichte Distanzmaße darstellen, bei denen große Werte bedeuten, dass Elemente in verschiedene Teilmengen platziert werden sollen.

Algorithmus 3 MMST

-- 1. Phase: Berechne den maximalen Spannbaum, ignoriere dabei allerdings zu leichte Kanten
Sortiere E nach absteigendem Kantengewicht
for all $v \in V$ **do**
  $MakeSet(v)$
end for
for all $(v_1, v_2) \in E$ **do** *-- absteigend nach Kantengewicht*
  if $Find(v_1) \neq Find(v_2)$ **then**
    -- Kante (v_1, v_2) würde zum maximalen Spannbaum gehören
    if $\omega(v_1, v_2) < \frac{1}{2}\alpha(\phi(v_1) + \phi(v_2))$ **then**
      Ignoriere zu leichte Kante
    else
      $Union(v_1, v_2)$ *-- Verschmelze Ballungen*
    end if
  end if
end for
-- 2. Phase: "Adoptiere" einsame Knoten
for all $v \in V$ **do**
  if $Size(v) = 1$ **then**
    Bestimme die inzidente Kante $e = (v, v')$ zu v mit dem größtem Gewicht $\omega(v, v')$.
    if $\omega(v, v') > 0$ **then**
      $Union(v, v')$ *-- Schlage v der passendsten Ballung zu*
    end if
  end if
end for
-- Die Union-Find-*Struktur enthält das Ergebnis der Ballungsanalyse.*

können. Für den Aufwand für *Union* können wir $O(|V|)$ Operationen veranschlagen, für *Find* liegt der Aufwand bei $O(\log|V|)$ Operationen. Zusätzlich benötigen wir noch die Operationen $Size(v)$ und $\phi(v)$. $Size(v)$ gibt die Anzahl der Elemente der Ballung zu v und $\phi(v)$ das arithmetische Mittel der Gewichte aller internen Kanten der durch v repräsentierten Ballung an.

Anstatt den maximalen Spannbaum zu konstruieren, berechnen wir direkt die Zusammenhangskomponenten, die später die gesuchten Ballungen ergeben. Ob wir dabei zwei bereits bestehende Zusammenhangskomponenten verschmelzen, machen wir davon abhängig, in wie weit sich durch das Hinzunehmen der Verbindungskante die innere Kohäsion der dadurch entstehenden Zusammenhangskomponente verschlechtert. Wir verschmelzen Zusammenhangskomponenten nur dann, wenn die Verbindungskante das durchschnittliche Kantengewicht der inneren Kanten der beiden Zusammenhangskomponenten multipliziert mit dem *Toleranzfaktor* α nicht unterschreitet.

Mit Hilfe des Parameters α lässt sich (in gewissen Grenzen) die Granularität der entstehenden Ballungen steuern. Kleinere Werte für α führen zu kleineren Zusammenhangskomponenten bzw. Ballungen. Taugliche Werte für α liegen zwischen 0,5 und 0,9. Für die Messungen in Kapitel 8 haben wir einen Wert von $\alpha = 0,7$ verwendet.

Nach dem Abschluss der ersten Phase enthält die *Union-Find*-Struktur die gesuchten Ballungen, welche die Teilsystemkandidaten für das zu untersuchende Softwaresystem darstellen.

Allerdings können dabei *einelementige* Ballungen auftreten. Teilsysteme, die genau eine Klasse enthalten, tragen allerdings nicht zu einer verständlichen Teilsystemstruktur bei. Unser Algorithmus enthält daher eine zweite Phase, in der wir solche einelementigen Ballungen entfernen, indem wir die "einsamen" Knoten geeigneten anderen Ballungen zuschlagen.

Unser auf der Berechnung des maximalen Spannbaums basierender Algorithmus zur Ballungsanalyse hat – eine entsprechende Implementierung der *Union-Find*-Struktur vorausgesetzt – einen Aufwand von $O(|E|\log|V|)$.

7.4 *HGGA* – ein genetischer Algorithmus zur Ballungsanalyse

Der im vorangehenden Abschnitt beschriebene Algorithmus *MMST* arbeitet *gierig*. Er erzeugt schrittweise durch Verschmelzen von jeweils zwei Ballungen eine neue Ballung, sofern die Kohäsion der entstehenden Ballung nicht allzu schwach wird. Ballungen, die durch Kanten mit besonders hohen Gewichten verbunden sind, werden dabei zuerst verschmolzen. Der Algorithmus *MMST* berechnet bereits eine brauchbare Zerlegung C unseres Abhängigkeitsgraphen $G(V,E,\omega)$ im Sinne von Abschnitt 7.1, für die die Qualitätsfunktion f gute Werte aufweist. Wir stellen im folgenden einen *genetischen Algorithmus* vor, der in der Praxis deutlich bessere Zerlegungen berechnet.

7.4.1 Grundlagen genetischer Algorithmen

Genetische Algorithmen (HOLLAND, 1975; GOLDBERG, 1989) verwenden die Grundprinzipien der biologischen Evolution zur Lösungssuche für meist NP-harte Probleme.

Die gängige Modellvorstellung der biologischen Evolution[9] ist die eines Anpassungsprozesses einzelner Arten an Umweltbedingungen über Generationen hinweg. Dabei verändert sich die Erbsubstanz, der *genetische Code*, der Individuen einer Population von einer Generation zur nächsten. Der genetische Code (*Genotyp*) trägt alle wesentlichen Informationen zu Aufbau, Organisation, Funktionalität und Erscheinungsbild eines Individuums (*Phänotyp*). Er ist in Form von *Chromosomen* codiert, die Bestandteile der Zellen jedes Individuums sind. Die Chromosomen setzen sich aus einzelnen *Genen* zusammen. Jedes Gen trägt dabei einen gewissen Anteil zum genetischen Code bei. Welche Merkmale eines Individuums ein bestimmtes Gen beeinflusst, ergibt sich aus seiner Position im Chromosom, dem *Locus*. Wie die entsprechenden Merkmale des Individuums ausfallen, ergibt sich aus dem Wert, dem *Allel*[10], des Gens.

Die wichtigsten treibenden Faktoren in der biologischen Evolution sind die *Mutation*, durch welche bei der Teilung einer Zelle kleinere Fehler bei der Replikation des genetischen Codes vorkommen, und die *Selektion*, bei der sich besser an die Umweltbedingungen angepasste Individuen der Population – also solche mit "besserer" Erbsubstanz – gegenüber anderen Individuen durchsetzen. Zusätzlich kennt die Evolutionslehre noch die *Rekombination*, bei der über die sexuelle Fortpflanzung der Individuen neue genetische Codes entstehen, die sich aus der Kombination der genetischen Codes der Eltern ergeben. Durch die Mutation werden Neuerungen in das vorhandene genetische Material einer Population eingeführt. Die Selektion garantiert, dass sich sinnvolle Neuerungen langfristig gegen unnütze Neuerungen durchsetzen können. Die Rekombination erzeugt kein grundsätzlich neues genetisches Material, allerdings sorgt sie dafür, dass sich die beste Kombination der in der Population vorhandenen Erbsubstanz durchsetzt. Im durch die Fortpflanzung entstehenden biologischen Zyklus einer Population spielt die Selektion an zwei Stellen eine besondere Rolle: sie beeinflusst sowohl die Überlebenschance eines Individuums in der Population als auch die Fähigkeit des Individuums, einen Fortpflanzungspartner zu finden und somit Teile seines genetischen Codes in nachfolgende Generationen zu übertragen.

Wir können den Anpassungsprozess der biologischen Evolution auf die Suche nach Lösungen eines Problems übertragen, in dem wir mit Hilfe der folgenden Vorgehensweise einen genetischen Algorithmus konstruieren (WEICKER, 2002):

[9] Die Grundzüge der Evolutionslehre gehen auf die Arbeiten von C. DARWIN (1809 – 1882) und G. J. MENDEL (1822 – 1884) zurück. Darwin beschrieb 1859 in seinem Werk *The Origin of Species* die Prinzipien der Selektion und Mutation. Mendel fand kurze Zeit danach, im Jahr 1865, das Prinzip der Rekombination.

[10] In der Natur sind Chromosome Polymer-Stränge aus Desoxyribonukleinsäure (DNA). Das Allel eines Gens ergibt sich aus den Aminosäuren, die im Polymer am Locus des Gens auftreten.

1. *Repräsentation der Problemlösungen.* Der Suchraum, der alle möglichen Lösungen für das Problem umfasst, wird auf eine Menge von Chromosomen abgebildet. Ein Chromosom codiert ein einzelnes Element des Lösungsraumes und wird durch eine Zeichenkette über einem endlichen Alphabet $\mathscr{Z}$ dargestellt. Die Chromosomen bilden die Repräsentationen der möglichen Lösungen für das Problem, auf denen der Algorithmus arbeitet.

2. *Bestimmen einer Startpopulation.* Zunächst wird eine Startpopulation aus μ möglichen Lösungen für das Problem berechnet, die als erste Generation dient. Häufig wird diese zufällig bestimmt.

3. *Selektion.* Nach einer bestimmten Selektionstrategie werden mit Hilfe einer Bewertungsfunktion f (häufig auch *Fitnessfunktion* genannt) eine bestimmte Anzahl der "besseren" Lösungen ausgewählt.

4. *Fortpflanzung und Rekombination.* Diese ausgewählten Lösungen bilden die Eltern der nächsten Generation. Mit Hilfe eines Rekombinationsoperators werden aus je zwei Lösungen, dem Elternpaar, neue Lösungen, die Nachkommen (bzw. Kinder), erzeugt. Der Rekombinationsoperator ersetzt in der Regel einen Ausschnitt aus dem Chromosom des einen Elternteils durch einen entsprechenden Aussschnitt aus dem Chromosom des anderen Elternteils und formt so das Chromosom eines Nachkommens. Ingesamt werden dabei λ Nachkommen erzeugt.

5. *Zusammenstellen einer neuen Generation.* Nach einer festgelegten Strategie wird die nächste Generation der möglichen Lösungen zusammengestellt. Bei der $(\mu + \lambda)$-Strategie entsteht die nächste Generation aus einer Auswahl von μ Individuen, die sich aus den Eltern und den Kindern der Rekombination rekrutieren. In der (μ, λ)-Strategie werden bei der Bildung der neuen Generation nur Kinder berücksichtigt (SCHWEFEL, 1995).[11] Wird die $(\mu + \lambda)$-Strategie verwendet, so kann ein Individuum beliebig über mehrere Generationen hinweg existieren. Bei der (μ, λ)-Strategie sterben alle Individuen nach einer Generation. Dadurch kann je nach Wahl des Rekombinationsoperators eine Folgegeneration mit insgesamt "schlechterem" Erbgut entstehen.

6. *Mutation.* Ein kleiner Anteil der μ Lösungen der neuen Generation unterliegen nun noch der Mutation. Dabei werden Lösungen modifiziert, um neue Bereiche des Suchraums zu erschließen. Die Mutation zielt auch darauf ab, zu verhindern, dass die Lösungen in der Population gegen ein lokales Optimum konvergieren.

7. *Iteration.* Nach Bildung einer neuen Generation aus μ Problemlösungen kann diese erneut den Evolutionsschritten ab Schritt 3 unterworfen werden.

[11] In diesem Fall müssen bei der Rekombination $\lambda \geq \mu$ Nachkommen erzeugt werden. Man beachte, dass dazu aus jedem Elternpaar zwei oder mehr Nachkommen hervorgehen müssen.

In den einzelnen Schritten spielt der *Zufall* eine gewisse Rolle: Selektion, Rekombination und Mutation erfolgen zwar nach definierten Strategien, sie enthalten allerdings stets randomisisierte Elemente.

Algorithmus 4 Grundform genetischer Algorithmus

```
 - - Erstelle eine Startpopulation der Größe μ
population := erzeugePopulation()
 - - Population über n Generationen hinweg entwickeln
for i := 1 to g_max do
   eltern := selektiere(population)
   kinder := erzeugeKinder(eltern)
   population := formierePopulation(eltern, kinder)
   mutiere(population)
end for
```

Aus unserer Vorgehensweise kristalliert sich ein Grundschema für genetische Algorithmen (siehe Algorithmus 4), heraus. Dieses Grundschema stellt einen allgemeinen Bauplan für genetische Algorithmen dar. Es enthält zahlreiche Freiheitsgrade, die passend zum Problem, zu dessen Lösung ein genetischer Algorithmus eingesetzt werden soll, gebunden werden müssen.

In den folgenden Abschnitten entwickeln wir unseren genetischen Algorithmus *HGGA (Hybrid Genetic Grouping Algorithm)* (BAUER et al., 2005) zur Zerlegung von Abhängigkeitsgraphen von Softwaresystemen, indem wir diese Freiheitsgrade der Reihe nach binden. Wir stellen dazu für jeden Freiheitsgrad einige der wichtigsten Alternativen vor und wählen daraus die für unseren Zweck geeignetste aus.

Abbildung 7.6 gibt in eine Übersicht über die Freiheitsgrade und zeigt Alternativen zur Bindung der Freiheitsgrade. Die Bestandteile unseres *HGGA* sind in Abbildung 7.6 schwarz hervorgehoben, die des in Abschnitt 3.3 bereits vorgestellten Werkzeugs *Bunch* (MANCORIDIS et al., 1999) grau.

Bei der Ausgestaltung unseres genetischen Algorithmus orientieren wir uns am *Genetic Grouping Algorithm (GGA)*, der von FALKENAUER (1994) vorgeschlagen wurde, um das *Behälterproblem*[12] zu lösen (FALKENAUER und DELCHAMBRE, 1992), bei dem es darum geht, eine Anzahl unterschiedlich schwerer Gegenständen auf eine Anzahl von Behältern einer gewissen Kapazität so zu verteilen, dass möglichst wenige Behälter benötigt werden. Der GGA unterscheidet sich von den Urformen der genetischen Algorithmen (HOLLAND, 1975; GOLDBERG, 1989), wie sie auch in *Bunch* zum Einsatz kommen, hinsichtlich der Codierung der Chromosome und der Operationen zur Rekombination. Unser *HGGA* greift die Grundprinzipien (FALKENAUER, 1998) des GGA auf

[12] engl. *bin packing problem*

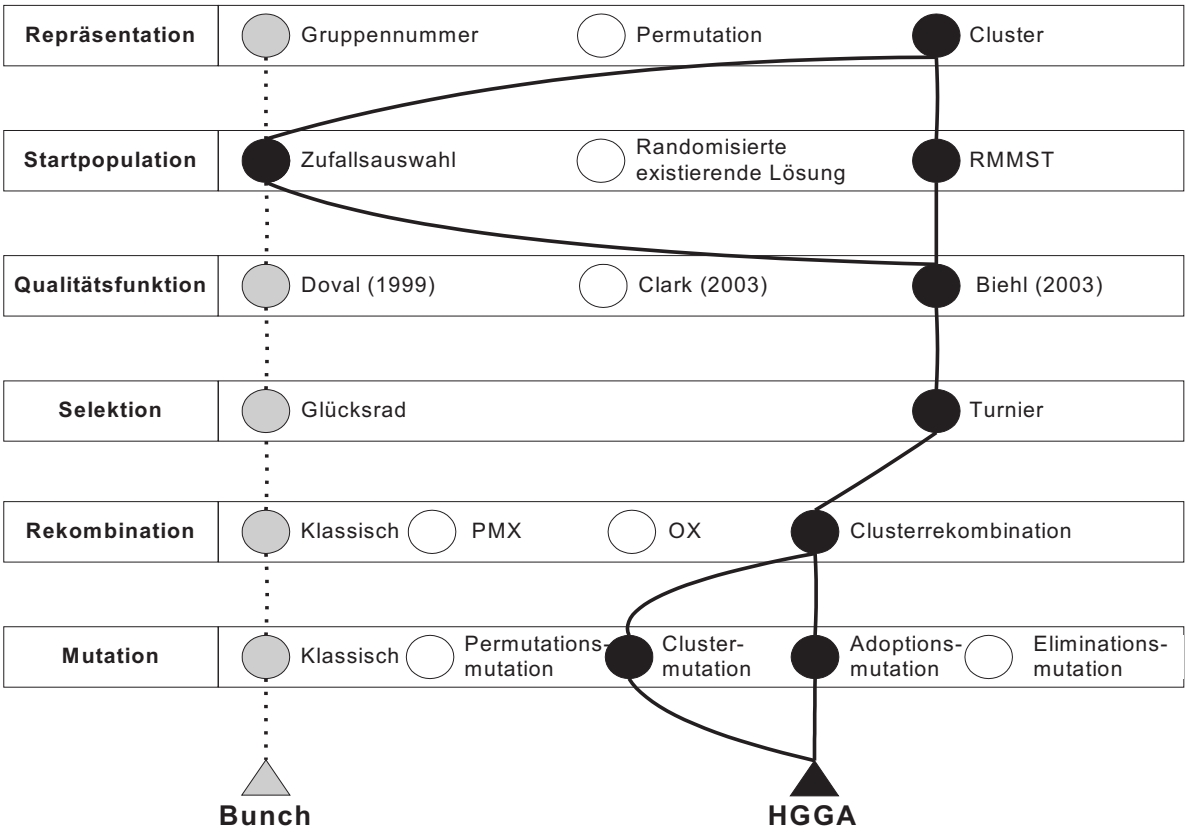


Abbildung 7.6: Variantenbildung bei genetischen Algorithmen zur Ballungsanalyse

und überträgt sie auf unser Problem, gute Zerlegungen für die Abhängigkeitsgraphen von Softwaresystmen zu finden.

Unser Algorithmus ist *hybrid*, weil er an verschiedenen Stellen problemspezifische Heuristiken einsetzt. Dies trifft insbesondere auf die Ausgestaltung der Mutationsoperationen und auf die Berechnung der Startpopulation zu.

Leider bietet der aktuelle Stand der Forschung nur wenige theoretisch fundierte Erkenntnisse darüber, wie genetische Algorithmen ausgestaltet werden müssen, damit sie eine bestimmte Art von Problem rasch und mit guter Qualität lösen. Es sind daher Experimente mit unterschiedlichen Alternativen für die zu bindenden Freiheitsgrade erforderlich. Für unsere Aufgabe wurden solche Experimente in einer durch den Autor dieser Arbeit betreuten Studienarbeit durchgeführt (BIEHL, 2004). Die wichtigsten Ergebnisse dieser Experimente stellen wir in Kapitel 8 zusammen.

7.4.2 Repräsentation

Eine gute Repräsentation der Elemente des Lösungsraumes und die damit verbundene Codierung der Elemente in Chromosomen ist eine Grundvoraussetzung für einen wirksamen genetischen Algorithmus. Die Codierung stellt eine bijektive Abbildung zwischen den Individuen des Lösungsraumes (Phänotypen) und den Chromosomen (Genotypen) dar. Diese Codierung muss so beschaffen sein, dass sie jedem Element des Lösungsraumes ein Chromosom zuordnet *(Vollständigkeit)*. Umgekehrt muss sich aus dem Chromosom ein Individuum rekonstruieren lassen *(Valididät)*. Diese Eigenschaften sind zwingend notwendig, damit ein genetischer Algorithmus sinnvolle Lösungen finden kann. Zusätzlich wünschenswert wäre, wenn jedes Element des Lösungsraumes auf genau ein Chromosom abgebildet wird *(Redundanzfreiheit)*. Kann eine Lösung durch mehrere unterschiedliche Chromosome repräsentiert werden, so vergrößert sich der Suchraum des genetischen Algorithmus. Bei hoher Redundanz kann das zu Effizienzeinbußen des genetischen Algorithmus führen.

In unserer Aufgabe, den Abhängigkeitsgraphen $G = (V, E, \omega)$ eines Softwaresystems zu zerlegen, besteht der Lösungsraum aus der Menge aller möglichen Zerlegungen des Graphen. Wir suchen daher nach einer Codierung, mit der Zerlegungen von Graphen (bzw. der zugrundeliegenden Knotenmengen) in Chromosomenstränge abgebildet werden können.

Mit Hilfe der *Gruppennummer-Codierung* kommen wir der Urform der genetischen Algorithmen (HOLLAND, 1975) am nächsten. Dazu wird jedem Knoten des Abhängigkeitsgraphen durch eine bijektive Abbildung $p : V \to \mathbf{N}$ eine Nummer zugewiesen. Ebenso erhält jede Ballung eine Nummer. Für den Knoten mit der Nummer i wird dann an der i-ten Position im Chromosom (Locus) die Nummer der Ballung, zu der sie gehört, gespeichert (Allel). Diese Repräsentation hat den Vorteil, dass die klassischen Operatoren zu Rekombination und Mutation (siehe Abschnitte 7.4.3 bzw. 7.4.4) direkt verwendbar sind. Die Gruppennummer-Codierung ist allerdings nicht redundanzfrei. Die Chromosomen 11133222 und 33322111 repräsentieren beispielsweise dieselbe Zerlegung $C = \{\{1,2,3\},\{6,7,8\},\{4,5\}\}$. Die Anzahl der Chromosomen, die ein- und dieselbe Zerlegung darstellen, wächst sogar exponentiell mit der Anzahl der Ballungen.

Ballungen, die mehrere Elemente enthalten, können nur durch Schemata mit großer Länge δ beschrieben werden. Wollen wir beispielsweise ausdrücken, dass eine gute Lösung voraussetzt, dass Knoten 1 und 5 in einer Ballung sind, so benötigen wir dazu schon Schemata der Länge 5. Lange Schemata werden leichter durch Rekombination zerstört, so dass sich bei der Gruppennummer-Codierung gute Lösungen häufig nur schwer in einer Population halten können.

Bei der *Permutationscodierung* (COLE, 1998) wird wie bei der Gruppennummer-Codierung jedem Knoten des Abhängigkeitsgraphen G eine Nummer zugeordnet. Diese Nummern werden als Allele im Chromosom gespeichert. Aus der Position p des Allels bzw. der Knotennummer i ergibt sich über eine Dekodierungsfunktion die Zugehörigkeit

zu einer Ballung. JONES und BELTRAMO (1991) verwenden eine gierige Dekodierungsfunktion, die voraussetzt, dass die Anzahl k der Ballungen bereits im Vorfeld bekannt ist. Sie ordnet dann den Loci in den Chromosomen reihum Ballungen zu. Auf diese Weise kann die Zerlegung $C = \{\{1,2,3\},\{6,7,8\},\{4,5\}\}, k = 3$ durch das Chromosom 16427538 dargestellt werden. Ist k wie in unserem Fall nicht a priori bekannt, so kann ein spezielles Trennzeichen ($\bullet$) in das Alphabet der Chromosomen aufgenommen werden. In diesem Fall lässt sich die Zerlegung C durch folgendes Chromosom darstellen: $123 \bullet 678 \bullet 45$.

Die Permutationscodierung für Zerlegungen des Abhängigkeitsgraphen ist nicht redundanzfrei, sie führt wie die Gruppennummer-Codierung ebenfalls zu einer Vergrößerung des Suchraumes. In den Chromosomen werden einerseits die Positionen der Knoten innerhalb der Ballungen codiert, andererseits spielt die Anordnung der Ballungen eine Rolle. Sowohl Zerlegungen wie auch die einzelnen Ballungen sind jedoch Mengen im mathematischen Sinne, d.h. die Anordnung der Elemente spielt keine Rolle. Die Chromosomen $678 \bullet 123 \bullet 45$ (geänderte Anordnung der Ballungen der Zerlegung) und $132 \bullet 678 \bullet 54$ (geänderte Anordnung der Knoten in den einzelnen Ballungen) sind beispielsweise ebenfalls eine gültige Codierung der in $123 \bullet 678 \bullet 45$ codierten Zerlegung C.

Unser *HGGA* folgt dem *Genetic Grouping Algorithm (GGA)* von FALKENAUER (1998) und verwendet eine etwas komplexere Codierung für Zerlegungen des Abhängigkeitsgraphen G. Die Allele unserer Chromosomen speichern die Ballungen des Graphen in Form von Knotenmengen. Das den Chromosomen zugrundeliegende Alphabet $\mathscr{Z}$ ist daher die Potenzmenge $\mathscr{P}(V)$. Da jedes Gen eine Ballung repräsentiert und die Anzahl der Ballungen ebenfalls durch den *HGGA* bestimmt wird, haben die Chromosome unterschiedliche Länge. Abbildung 7.7 zeigt die Codierung der Zerlegung $C = \{\{1,2,3\},\{6,7,8\},\{4,5\}\}$.

Diese Codierung ist leider ebenfalls nicht redundanzfrei. Die Codierung unterscheidet zwischen unterschiedlichen Anordnungen der Ballungen innerhalb einer Zerlegung. Durch geschickte Wahl der Operationen für Rekombination und Mutation in Verbindung mit einer geeigneten Strategie zur Bildung der Startpopulation können wir allerdings sicherstellen, dass eine Population keine Redundanzen dieser Art enthält.

7.4.3 Rekombination

Zusammen mit der Mutation beeinflusst die Rekombination maßgeblich, wie die Fortentwicklung einer Population von Individuen in einem genetischen Algorithmus vonstatten geht. Die Rekombination erzeugt (in Verbindung mit passenden Selektionsstrategien) die Nachkommen zur jeweils aktuellen Generation von Individuen. Kennzeichen genetischer Algorithmen ist, dass die Rekombination ausschließlich das genetische Material einer Population manipuliert. Der zugrundeliegende Rekombinationsoperator formt

daher aus zwei Chromosomensträngen einen neuen Chromosomenstrang. Dieser reicht aus, um den Phänotyp des Nachkommens vollständig zu beschreiben. Die Rekombination muss dabei dafür sorgen, dass sich einzelne "gute" Bausteine der Chromosome allmählich zu guten Lösungen zusammensetzen. Aus diesem Grund muss der Rekombinationsoperator auf die gewählte Codierung der Individuen abgestimmt sein.

Der klassische genetische Rekombinationsoperator (HOLLAND, 1975) passt zur Gruppennummer-Codierung und funktioniert nach folgendem Prinzip: Per Zufall wird eine Schnittposition s bestimmt. An dieser Stelle werden die beiden Chromosomenstränge der Eltern in jeweils zwei Teile zerlegt. Diese Teile werden – wie in Abbildung 7.8(a) dargestellt – "über Kreuz" zu den neuen Chromosomensträngen der Kinder zusammengefügt. Aus den beiden Zerlegungen $C = \{\{1,2,3\},\{6,7,8\},\{4,5\}\}$ und $C' = \{\{1,2\},\{3,6\},\{7,8\},\{4,5\}\}$, codiert über die Chromosome 11133222 und 11244233, ergeben sich durch die Rekombination mit Schnittposition $s = 4$ die Chromosome 11134233 und 11243222 bzw. die Zerlegungen $\{\{1,2,3\},\{6\},\{4,7,8\},\{5\}\}$ und $\{\{1,2\},\{3,6,7,8\},\{5\},\{4\}\}$.

An diesem Beispiel zeigt sich eine Schwäche dieses Rekombinationsoperators in Verbindung mit der Gruppennummer-Codierung: Obwohl die Ballung {4, 5 } in beiden Zerlegungen C und C' enthalten war, ist sie in *keinem* der beiden Kinder mehr vorhanden.

Zur Permutationscodierung kennt die Literatur ebenfalls passende Rekombinationsoperatoren. Der *PMX*[13]-Operator (GOLDBERG und LINGLE, 1985) arbeitet mit zwei Schnittpunkten s_1 und s_2, die den sogenannten Schnittbereich definieren. Dieser Schnittbereich wird zwischen den beiden Eltern-Chromosomen ausgetauscht. Durch diesen Austausch entsteht eine Abbildung $m : \mathscr{Z} \to \mathscr{Z}$, welche die Allele in den Schnittbereichen *positionstreu* aufeinander abbildet, für alle anderen Werte ist m die Identität. Abbildung 7.8(b) verdeutlicht die Funktionsweise des PMX-Operators. Aus den Chromosomen $c_1 = 123 \bullet 678 \circ 45$ und $c_2 = 12 \bullet 36 \circ 78 \star 45$ der Zerlegungen C und C' berechnet sich m nach Wahl der Schnittpunkte $s_1 = 4$ und $s_2 = 7$ zu $m(3) = \bullet, m(\bullet) = 3, m(6) = 6, m(\circ) = 7, m(7) = \circ$ und $m(x) = x$ für alle $x \neq \bullet, \circ, 3, 6, 7$.[14] Wenden wir m auf die Gene der Chromosomen c_1 und c_2 an, so erhalten wir für die Chromosomen der Nachkommen $12 \bullet 36 \circ 8745$ und $123 \bullet 67 \circ 8 \star 45$.

In der Literatur finden sich weitere Rekombinationsoperatoren, die zu permutationscodierten Chromosomen passen – häufig verwendet wird noch der *OX*[15]-Operator (DAVIS, 1985).

Wir konzipieren nun passend zu unserer Codierung (siehe Abschnitt 7.4.2) einen Rekombinationsoperator. Dieser Operator soll möglichst viele Ballungen vollständig von

[13] *Partially Mapped Crossover*

[14] Damit der PMX-Operator korrekt funktioniert, müssen die Allele eines Chromosoms paarweise verschieden sein, wir haben daher die Menge der Trennzeichen entsprechend erweitert.

[15] *Order-based Crossover*

einer Generation in die nächste übertragen. Der Rekombinationoperator enthält folgende Arbeitsschritte:

1. *Bestimmen von Schnittbereichen:* In den Chromosomen der beiden Eltern werden zunächst unabhängig je zwei unterschiedliche Schnittbereiche zufällig bestimmt. Wir beachten, dass auf diese Weise die Schnittbereiche jeweils eine bestimmte Anzahl vollständiger Ballungen enthalten, da jedes Gen eine Ballung repräsentiert.

2. *Kreuzung:* Die zuvor bestimmten Schnittbereiche werden nun über Kreuz ausgetauscht. Dadurch entstehen neue Chromosome für zwei Nachkommen, die jeweils einen neuen, injizierten Bereich und maximal zwei alte Bereiche, die unverändert vom jeweiligen Elter[16]übernommen wurden, enthalten. Einige Knoten unseres Abhängigkeitsgraphen G sind nun mehreren Ballungen zugeordnet, so dass die neuen Chromosome zunächst noch keine gültigen Zerlegungen darstellen. Daher sind die folgenden beiden Schritte nötig, in denen die Chromosome "repariert" werden.

3. *Elimination ungültiger Ballungen:* Zunächst werden aus den beiden neuen Chromosomen alle Ballungen entfernt, die Elemente aus den neuen, injizierten Bereichen enthalten. Dabei bleiben die Elemente übrig, die nur in den zu entfernenden Ballungen enthalten waren, nicht aber in Ballungen in den Schnittbereichen. Diese wandern als "freie" Elemente in einen sogenannten *Gen-Pool*.

4. *Zuordnen freier Elemente zu gültigen Ballungen:* Die verbleibenden freien Elemente des Gen-Pools werden nun passenden, bereits existierenden Ballungen zugeordnet. Für jeden zuzuordnenden Knoten v wird dazu der *nächste Nachbar* $\overline{v}$ bezüglich der Kantengewichtung ω von G bestimmt und der Knoten dann der Ballung zugeorndet, die den nächsten Nachbarn enthält. Der nächste Nachbar $\overline{v}$ zu v ist derjenige Knoten, der über die am stärksten gewichtete Kante mit v verbunden ist. Für diesen gilt:

$$\omega(v,\overline{v}) = max_{v' \in V}(\omega(v,v')) \tag{7.23}$$

Abbildung 7.9 illustriert diese Schritte am Beispiel unserer Zerlegungen C und C'. Der Übersichtlichkeit halber zeigt die Abbildung nur eines der beiden Nachkommen.

Im Grundsatz folgt unser Operator der Vorgehensweise des *BPX*[17]-Operators, der dem *GGA* von FALKENAUER (1998) entstammt. Unser Operator weicht jedoch in den Schritten 3 und 4 vom PBX-Operator ab, in denen wir Wissen über die Problemdomäne ausnützen, um Chromosomen für hochwertige Nachkommen zu erzeugen.

[16] In der Genetik ist der geschlechtslose Begriff Elter anstelle der üblichen, geschlechtlichen Begriffe Vater und Mutter üblich.

[17] *Bin Packing Crossover*

7.4.4 Mutation

Die Mutation beeinflusst die Qualität der Lösungen, die ein genetischer Algorithmus entwickelt. Mutationsoperatoren injizieren neues genetisches Material in eine Population und tragen so dazu bei, dass der Lösungsraum angemessen durchkämmt wird.

Gute Mutationsoperatoren für genetische Algorithmen müssen so konzipiert sein, dass sie Bausteine für gute Lösungen nicht zerstören, aber dennoch einige Individuen einer Population so modifizieren, dass neue Bereiche des Lösungsraumes erschlossen werden. Beim Einführen neuen genetischen Materials in die Population muss gewährleistet sein, dass die Qualität (hinsichtlich der Qualitätsfunktion f) der mutierten Individuen mit einer gewissen Wahrscheinlichkeit überdurchschnittlich hoch ist. Mutanten mit geringer Qualität würden ansonsten durch die folgenden Selektionsvorgänge im Algorithmus unverzüglich wieder aus der Population entfernt.

Verwendet man einen klassischen genetischen Algorithmus (Gruppennummer-Codierung und Rekombination gemäß Abbildung 7.8(a)) zur Berechnung von Zerlegungen des Abhängigkeitsgraphen G, so kann eine einfache Mutationsoperation eingesetzt werden (DOVAL et al., 1999): das Allel eines zufällig gewählten Gens wird mit einem Zufallswert belegt. So könnte das Chromosom 1113322**2** zu unserer Beispielzerlegung C zum Chromosom 1113323**3**2 mutieren. Wir erkennen, dass dies auf Ebene der Phänotypen bedeutet, zufällig einen Knoten des Abhängigkeitsgraphen auszuwählen und einer anderen Ballung zuzuordnen.

Im Falle einer Permutationscodierung kann diese Mutationsoperation umgesetzt werden, indem ein Allel eines Chromosoms verschoben wird *(Permutationsmutation)*.

Eine solche Mutation führt mit sehr hoher Wahrscheinlichkeit zu Individuen niedriger Qualität, die rasch wieder eliminiert werden. Solche Mutationen kommen dem natürlichen Vorbild sehr nahe, allerdings führen sie in genetischen Algorithmen zu langen Laufzeiten. Diese entstehen dadurch, dass sehr viele Generationen vergehen können, bis die Mutation tatsächlich erhaltenswerte Individuen hervorbringt.

Wir verwenden daher Mutationsoperatoren, die Ballungen so modifizieren, dass sich die Chance auf hochwertige Individuen erhöht. Grundsätzlich sind in unserem *HGGA* folgende Mutationen vorgesehen, die mit unterschiedlichen Wahrscheinlichkeiten h ausgeführt werden:

- *Ballungsmutation – Zusammenfassen und Auftrennen von Ballungen:* Die Ballungsmutation manipuliert Ballungen, indem sie entweder zwei Ballungen zu einer neuen größeren Ballung zusammenfasst, oder indem sie eine Ballung in zwei kleinere Ballungen auftrennt. Damit die so erzeugten Nachkommen eine gewisse Überlebenschance beim Übergang in die nächste Generation haben, trennen wir eine Ballung nur an Kanten auf, die über geringe Kantengewichte ω verfügen.

- *Adoptionsmutation:* Bei der Adoptionsmutation werden einige Elemente von einer zufällig bestimmten Ballung C_i in eine beliebige Ballung C_j verschoben. Dabei werden solche Elemente v aus C_i ausgewählt, die über eine hohe Kopplung $\varepsilon_j(v) = \sum_{v' \in C_j} \omega(v, v')$ zu Elementen in C_j verfügen. Insbesondere werden Ballungen, die nur ein Element enthalten, nach diesem Prinzip mit anderen Ballungen zusammengeführt. In diesem Fall sprechen wir von einer *Waisenadoption*[18]. Diese kennen wir bereits als Bestandteil des *MMST* (vgl. Abschnitt 7.3).

- *Eliminationsmutation:* Bei der Eliminationsmutation wird eine zufällig ausgewählte Ballung entfernt. Dadurch entstehen ähnlich wie im Schritt 3 der Rekombination des *HGGA* freie Elemente, die wieder Ballungen zugeordnet werden müssen. Wir ordnen diese Elemente auch hier wieder den Ballungen zu, die die jeweils nächsten Nachbarn enthalten.

Im *HGGA* werden sowohl Ballungsmutation als auch Adoptionsmutation recht häufig ausgeführt. Im Rahmen der Ballungsmutation werden Auftrennen und Verschmelzen von Ballungen ungefähr gleich häufig ausgeführt. Die Eliminationsmutation wird sehr selten ausgeführt, da bei ihr die Gefahr recht groß ist, gute Bausteine innerhalb der Chromosome zu zerstören. Anhang B.3 enthält genauere Angaben über die Wahrscheinlichkeiten h, die der Anwendung all dieser Operationen zugrunde liegen.

Zusätzlich zu den Mutationsoperationen, die sowohl den Genotyp (in Form der Chromosome) als auch den Phänotyp (also die Zerlegungen) der Individuen manipulieren, kann auch die *Inversionsoperation* von Nutzen sein. Diese modifiziert ausschließlich den Genotyp der Individuen. Eine solche Operation kann es nur dann geben, wenn die dem Algorithmus zugrundeliegende Codierung der Individuen nicht redundanzfrei ist.

Ziel der Inversion ist es, dafür zu sorgen, dass sogenannte *co-adaptive* Gene an benachbarten Stellen im Chromosom gespeichert werden. Co-adaptive Gene bewirken in ihrer Gesamtheit eine positive, phänotypische Eigenschaft eines Individuums. In unserem Fall repräsentieren Gene Ballungen. Ballungen sind dann co-adaptiv, wenn ihre Knoten im Abhängigkeitsgraphen durch kurze Pfade verbunden sind. Können co-adaptive Gene benachbart im Chromosom gespeichert werden, so erhöht sich die Wahrscheinlichkeit, dass die Rekombination besonders gute Nachkommen hervorbringt. Der damit verbundene Austausch der Schnittbereiche in den Chromosomen wirkt sich auf im Graphen benachbarte Ballungen aus und erscheint damit besonders sinnvoll.

Der Inversionsoperator funktioniert nach folgendem Prinzip (GOLDBERG, 1989): Zunächst werden zwei zufällige Positionen s_1 und s_2, $s_1 < s_2$ im Chromosom bestimmt. Innerhalb des Bereichs zwischen s_1 und s_2 wird die Reihenfolge der Gene (bzw. Ballungen) invertiert.

Die Inversion wirkt indirekt – erst bei der nächsten Rekombination kann sich zeigen, ob die modifizierten Chromosome zu besseren Nachkommen führen und sich die neue

[18] engl. *orphan adoption*

Anordnung des genetischen Materials durchsetzt. Auf diese Weise suchen genetische Algorithmen mit Inversion sowohl nach einer guten phänotypischen Lösung als auch nach einer guten genotypischen Codierung.

In der Praxis benötigt unser *HGGA* die Inversion nicht, da wir bereits bei der Erzeugung der Startpopulation (siehe Abschnitt 7.4.7) für eine günstige Anordnung der Gene bzw. Ballungen in den Chromosomen sorgen und diese Anordnung bei der Rekombination weitestgehend erhalten bleibt.

7.4.5 Qualitätsfunktion

Ein genetischer Algorithmus benötigt eine Qualitätsfunktion f zur Bewertung der Individuen. In unserem Fall arbeitet die Qualitätsfunktion auf der phänotypischen Darstellung der Individuen; f ordnet also (unter Berücksichtigung der Gestalt des Abhängigkeitsgraphen $G(V,E,\omega)$) einer Zerlegung $C = \{C_1,\ldots,C_k\}$ einen Wert x auf einer Ordinalskala[19] zu, $f : G \times C \mapsto x, x \in \mathbf{R}$.

Einen Kandidaten für eine Qualitätsfunktion haben wir bereits in Formel 7.4 (vgl. CLARK et al. (2003)) eingeführt. Diese bewertet eine Zerlegung dann gut, wenn die externe Kopplung jeder einzelnen Ballung zu anderen Ballungen aus C im Verhältnis zur internen Kohäsion möglichst gering ausfällt.

DOVAL et al. (1999) verwenden für *Bunch* eine etwas andere Qualitätsfunktion: Wir nehmen zunächst an, dass der Graph G nicht gewichtet sei, also $\omega(v_1,v_2) = 1, v_1, v_2 \in V$. Dann berechnet sich die relative interne Kohäsion A_i der Ballung C_i durch

$$A_i = \frac{\mu_i}{|C_i|^2} \tag{7.24}$$

und die relative externe Kopplung $E_{i,j}$ von C_i zur Ballung C_j durch

$$E_{i,j} = \frac{\varepsilon_{i,j}}{2|C_i||C_j|} \tag{7.25}$$

A_i und $E_{i,j}$ stellen normierte Varianten für die interne Kohäsion μ_i bzw. die externe Kopplung $\varepsilon_{i,j}$ von Ballungen dar, wie wir sie in den Gleichungen 7.6 und 7.7 bereits definiert haben.

Damit definieren DOVAL et al. (1999) folgende Qualitätsfunktion:

[19] Eigentlich wäre wünschenswert, dass f eine Verhältnisskala zugrunde liegt, so dass Aussagen der Form Zerlegung *"C ist doppelt so gut wie Zerlegung C′"* möglich sind. Viele etablierte Qualitätsfunktionen zur Bewertung von Zerlegungen sind jedoch lediglich ordinale Maße, die nur unquantifizierbare Vergleiche *("C ist besser als C′")* zwischen Messwerten erlauben.

$$f(G,C) = \begin{cases} \frac{\sum_{i=1}^{k} A_i}{k} - \frac{\sum_{i,j=1,i\neq j}^{k} E_{i,j}}{\frac{k(k-1)}{2}} & , \quad k > 0 \\ A_i & , \quad k = 0 \end{cases} \tag{7.26}$$

Diese Funktion nimmt Werte im Intervall zwischen -1 (keine Kohäsion innerhalb der Ballungen) und 1 (keine Kopplung zwischen den Ballungen) an. Die Formel ist allerdings nur für ungewichtete Graphen geeignet. Um sie für gewichtete Graphen anzuwenden, können wir die Berechnung von relativer Kohäsion und Kopplung entsprechend anpassen:

$$\overline{A}_i = \frac{\mu_i}{|C_i|^2 \omega_{max}} \tag{7.27}$$

$$\overline{E}_{i,j} = \frac{\varepsilon_{i,j}}{2|C_i||C_j|\omega_{max}} \tag{7.28}$$

wobei $\omega_{max} = \max_{e \in E} \omega(e)$ das maximale Kantengewicht in G bezeichnet.

Wir verwenden im *HGGA* eine leicht modifizierte Form der Qualitätsfunktion aus Gleichung 7.4:

$$f(G,C) = \sum_{i=1}^{k} Q_i \tag{7.29}$$

mit

$$Q_i = \begin{cases} 0 & , \quad \mu_i = 0 \\ \frac{\mu_i}{\mu_i + \xi \sum_{j=1,i\neq j}^{k} \varepsilon_{i,j}} & , \quad \text{sonst} \end{cases} \tag{7.30}$$

Diese Qualitätfunktion besitzt lediglich eine Ordinalskala. Sie ist für ungerichtete, gewichtete Abhängigkeitsgraphen geeignet. Die Einzelmaße Q_i für die Ballungen nehmen Werte im Intervall von 0 (Ballung i hat keine interne Kohäsion) bis 1 (die Ballung i hat keine externe Kopplung zu anderen Ballungen) an. Um die Bildung sehr kleiner Ballungen (mit einem oder wenigen Knoten) zu verhindern, messen wir der externen Kopplung von Ballungen mit Hilfe des Faktors ξ eine verstärkte qualitätsmindernde Bedeutung bei. Im Rahmen einer Untersuchung mit Hilfe von Fallstudien (BIEHL, 2004) hat sich diese Qualitätsfunktion (mit $\xi = 10$) sehr gut bewährt, da sie auch leichte Unterschiede in der Qualität der Zerlegungen zuverlässig wiedergeben kann.

7.4.6 Selektion

Die Fortentwicklung einer Population P wird durch die Selektion maßgeblich bestimmt – zum einen müssen aus einer Generation vielversprechende Individuen als Eltern ausgewählt werden, zum anderen müssen aus der Gesamtmenge von Eltern und Nachkommen diejenigen Individuen ausgewählt werden, die in der Population verbleiben sollen. Entscheidend ist, dass die Selektion zumindest teilweise dem Zufall unterworfen wird *(Noisy Selection)*, um zu garantieren, dass der genetische Algorithmus den Lösungsraum breit durchkämmt.

Klassische genetische Algorithmen nach HOLLAND (1975), zu denen auch der in *Bunch* verwendete gehört, bedienen sich dabei der *Glücksrad*-Selektion[20]. Jedem Individuum wird dabei eine Wahrscheinlichkeit P_{sel} zugeordnet, mit der für die Reproduktion bzw. zum Verbleib in der Population ausgewählt wird. Diese Wahrscheinlichkeit wird dabei für ein Individuum C berechnet durch:

$$P_{sel}(C) = \frac{f(G,C)}{\sum_{C' \in P} f(G,C')} \tag{7.31}$$

Die Auswahl erfolgt somit zufallsgetrieben. Im Mittel werden dabei bessere Individuen mit höherer Wahrscheinlichkeit ausgewählt. Genauer: die Auswahlwahrscheinlichkeit $P_{sel}(C)$ ist proportional zur Fitness $f(G,C)$ des Individuums C. In *Bunch* wird die Glücksrad-Selektion mit einer $(\mu + \lambda)$-Strategie verwendet – eine neue Generation setzt sich aus ausgewählten Vertretern der Elterngeneration und deren Nachkommen zusammen.

Für unsere ordinale Qualitätsfunktion ist die Glücksrad-Selektion nur bedingt geeignet, da die Glücksrad-Selektion aufgrund der angestrebten Proportionalität zwischen Fitness und Auswahlwahrscheinlichkeit eine Bewertungsfunktion mit Verhältnisskala voraussetzt.

Für unseren *HGGA* verwenden wir ebenfalls die $(\mu + \lambda)$-Strategie, die wir mit Hilfe der *Turnier*-Selektion realisieren:

1. Zunächst ermitteln wir über ein Turnier eine *Rangliste* der μ Individuen der Population. Dazu werden zufällig je zwei Individuen der Population als Gegner ausgewählt. Der Sieger, das Individuum mit der höheren Fitness f, verbleibt in der Population, der Verlierer wird aus der Population entfernt und vorne in eine Rangliste eingefügt. Diese "Wettkämpfe" werden solange wiederholt, bis alle Individuen in die Rangliste eingetragen sind. Danach stehen bessere Kandidaten tendenziell weiter vorn in der Rangliste, schlechtere Kandidaten weiter hinten. Durch die zufällige Auswahl der Gegner ist allerdings nicht garantiert, dass die Rangliste streng nach der Fitness der Individuen geordnet ist.

[20] engl. *roulette wheel selection*

2. Aus der Rangliste wählen wir aus den λ ersten Plätzen je zwei aufeinander folgende Individuen als Elternpaare aus und wenden auf diese den in Abschnitt 7.4.3 beschriebenen Rekombinationsoperator an. Dadurch entstehen pro Elternpaar je zwei Nachkommen. Diese Nachkommen ersetzen die λ letzten Plätze in der Rangliste, so dass sich aus der Rangliste wieder eine Population mit μ Individuen ergibt, die dann der in Abschnitt 7.4.4 beschriebenen Mutation unterworfen werden, wobei die einzelnen Mutationsoperatoren mit den bestimmten Wahrscheinlichkeiten *h* (siehe Anhang B.3) angewendet werden.

7.4.7 Startpopulation

Damit genetische Algorithmen gute Ergebnisse erzielen können, sind sie darauf angewiesen, dass die Individuen der Population den Lösungsraum möglichst breit abdecken. Die Individuen der Startpopulation müssen daher sehr unterschiedliche phänotypische Eigenschaften aufweisen. Alle Individuen der Startpopulation müssen zudem gültige Lösungen darstellen – in unserem Fall müssen alle Individuen vollständige und konsistente Zerlegungen des Abhängigkeitsgraphen *G* darstellen (siehe Abschnitt 7.1).

Im Prinzip können wir beliebige Zerlegungen des Graphen zufällig erzeugen und als Startpopulation verwenden. In klassischen genetischen Algorithmen ist eine solche vollständig zufällige Erzeugung einer Startpopulation die Regel (HOLLAND, 1975; DOVAL et al., 1999).

Wünschenswert ist allerdings, dass die Individuen der Startpopulation möglichst *vielversprechende* Bereiche des Lösungsraums abdecken. Daher können problemspezifische Heuristiken verwendet werden, um die Startpopulation mit besonders geeigneten Individuen anzureichern und so eine frühzeitige Bildung guter Bausteine im genetischen Code der Population zu forcieren (FALKENAUER, 1998; HARMAN und MAHDAVI, 2003).

Wir nutzen dieses Prinzip für den *HGGA*, indem wir unseren MMST-Algorithmus (siehe Abschnitt 7.3) dazu verwenden, eine bestimmte Anzahl guter Individuen in die Startpopulation zu injizieren. Hierzu randomisieren wir den *MMST*-Algorithmus (Algorithmus 3), um eine Schar verschiedener Zerlegungen des Abhängigkeitsgraphen *G* erzeugen zu können. Der randomisierte *MMST* (kurz *RMMST*, siehe Algorithmus 5) unterscheidet sich vom *MMST* in folgenden Punkten:

- Der *MMST* berechnet eine Zerlegung des Abhängigkeitsgraphen *G*, indem er aus dem maximalen Spannbaum von *G* Kanten entfernt, sofern deren Gewicht eine gewissen Grenze unterschreitet. Aus den danach verbleibenden Teilgraphen ergibt sich dann eine Zerlegung des Abhängigkeitsgraphen. Ob eine Kante gelöscht wird, hängt unter anderem von einem fest vorgegeben Toleranzfaktor α ab. Zur Gewinnung einer Schar von Zerlegungen, führen wir den Algorithmus mit verschiedenen, zufällig aus einem sinnvollen Intervall $[\alpha_{min}, \alpha_{max}]$ gewählten Tole-

Algorithmus 5 RMMST

- - *Sei* $B = (V, E')$ *der maximaler Spannbaum von* $G = (V, E, \omega)$
- - E' *sei absteigend sortiert*
Wähle zufällig einen Toleranzfaktor $\alpha^* \in [\alpha_{min}, \alpha_{max}]$
for all $v \in V$ **do**
  $MakeSet(v)$
end for
$r := 1$
for all $(v_1, v_2) \in E'$ **do** - - *absteigend nach Kantengewicht*
  - - *Berücksichtige die Kanten nur mit einer gewissen Wahrscheinlichkeit:*
  $g := \frac{1}{2} + \frac{1}{2}\cos(\pi \frac{r}{|E'|})$
  Bestimme eine Zufallszahl $z \in [0, 1)$
  if $z \in [0, g)$ **then**
    if $\omega(v_1, v_2) < \frac{1}{2}\alpha^*(\phi(v_1) + \phi(v_2))$ **then**
      Ignoriere zu leichte Kante ($\hat{=}$ Entfernen der Kante aus dem Spannbaum)
    else
      $Union(v_1, v_2)$ - - *Verschmelze Ballungen*
    end if
  end if
  $r := r + 1$
end for
- - *"Adoptiere" einsame Knoten*
- - *. . . und schlage sie zufällig damit verknüpften Ballungen zu*
for all $v \in V$ **do**
  if $Size(v) = 1$ **then**
    Bestimme zufällig eine inzidente Kante $e = (v, v'), \omega(v, v') > 0$ zu v
    $Union(v, v')$ - - *Schlage v der Ballung zu, in der sich* v' *befindet*
  end if
end for
- - *Die* Union-Find-*Struktur enthält ein*
- - *Individuum der Startpopulation für den* HGGA

ranzfaktoren α^* aus. Dadurch entstehen für jeden Lauf des *RMMST* Zerlegungen unterschiedlicher Granularität.

- Wir unterwerfen die Entscheidung, ob wir eine Kante aus dem Spannbaum entfernen wollen, einer Wahrscheinlichkeit g. Kanten, die sich zu frühen Zeitpunkten für den gierig berechneten Spannbaum qualifiziert haben, tauchen mit größerer Wahrscheinlichkeit als interne Kanten innerhalb einer Ballung auf, als solche, die zu späteren Zeitpunkten hinzugefügt werden. Für die r-te zu betrachtende Kante berechnen wir die entsprechende Wahrscheinlichkeit $g(r)$ durch folgende Funktion:

$$g(r) := \frac{1}{2} + \frac{1}{2}\cos(\pi \frac{r}{|E'|}) \tag{7.32}$$

  Abbildung 7.10 zeigt, dass diese Funktion zu einem scharfen, nichtlinearen Abfall der Wahrscheinlichkeiten führt. Damit provozieren wir die Erzeugung kleiner Ballungen, die dann als hoffentlich geeignete Bausteine durch den *HGGA* ausgenutzt und über Rekombinationen und Verschmelzungsmutationen zu neuen Ballungen kombiniert werden können.

- Auch der *RMMST* enthält eine nachgelagerte Phase zur Adoption "einsamer" Knoten in Form von einelementigen Ballungen. Wir schlagen einen Einzelknoten v zufällig einer Ballung zu, die durch eine Kante mit v verbunden ist.

In der Praxis zeigt es sich, dass der *HGGA* die besten Resultate erzielt, wenn die Startpopulation aus einer Mischung von rein zufällig erzeugten Zerlegungen und per *RMMST* berechneten Zerlegungen besteht (siehe dazu auch 8.3.2).

7.5 Hierarchiebildung

Die oben aufgeführten Ballungsanalysen liefern gute, aber recht kleine Ballungen. Um größere Teilsysteme zu gewinnen, können diese kleinen Ballungen nochmals zu größeren Einheiten zusammengefasst werden. Dies kann über die iterative Anwendung der Ballungsanalyseverfahren geschehen.

Abbildung 7.11 zeigt, wie wir bei der hierarchischen Ballungsanalyse vorgehen können. Zunächst müssen wir mit Hilfe der Ballungen einer vorausgehenden Ballungsanalyse wieder einen Abhängigkeitsgraphen $G' = (V', E', \omega')$ konstruieren. Die Ballungen sind dabei dessen Knoten. Die Kanten des Graphen ergeben sich aus den externen Abhängigkeiten der Elemente der Ballungen.

Seien C_1 und C_2 solche Ballungen einer vorausgehenden Ballungsanalyse. Dann ergibt sich zwischen den beiden Ballungen folgendes Ähnlichkeitsmaß:

$$\omega'(C_1, C_2) = \sum_{(v_1, v_2) \in C_1 \times C_2} \omega(v_1, v_2) \tag{7.33}$$

Der neue Abhängigkeitsgraph kann dann einer Ballungsanalyse mit Hilfe des *MMST* oder des *HGGA* unterworfen werden. In der Praxis hat es sich bewährt, bei der iterativen Anwendung der Ballungsanalyseverfahren einelementige Ballungen zuzulassen. Hierfür kann bei *MMST* auf die zweite Phase verzichtet werden. Beim *HGGA* verzichten wir in diesem Fall auf die *Adoptionsmutation*.

Diese Vorgehensweise kann wiederholt werden, so dass sich insgesamt eine Hierarchie von Ballungen bzw. Teilsystemen ergibt. In dieser Hierarchie lässt sich jede Ballung einer *Ebene* zuordnen. Die Hierachieebene einer Ballung ergibt sich daraus, in welcher Iteration die Ballung entstanden ist.

7.6 Zusammenfassung

In diesem Kapitel haben wir gezeigt, wie sich Systeme auf der Grundlage ihrer Abhängigkeitsgraphen in Teilsysteme zerlegen lassen.

Grundvoraussetzung ist zunächst eine vernünftige Konstruktion des Abhängigkeitsgraphen unter Berücksichtigung der verschiedenen Rollen der Systembauteile und einer entsprechenden Gewichtung der Abhängigkeiten über sogenannte Ähnlichkeitsmaße. Wir haben erläutert, wie sich bei der Konstruktion des Abhängigkeitsgraphen die Ergebnisse einer vorangehenden Klassifikation der Systembauteile systematisch ausnutzen lassen.

Grundsätzlich gehört das Berechnen einer günstigen Zerlegung des Abhängigkeitsgraphen zur Klasse der NP-harten Probleme. Da das genaue Berechnen einer optimalen Zerlegung des Abhängigkeitsgraphen schon für kleinere Softwaresysteme nicht mehr praktikabel ist, müssen Näherungsverfahren zur Berechnung einer guten Zerlegung gefunden werden. Wir haben zwei solcher Verfahren konstruiert: ein deterministisches, graphbasiertes Verfahren in Form des *MMST*-Algorithmus und ein stochastisches Suchverfahren in Form des genetischen Algorithmus *HGGA*. Die Konstruktion des genetischen Algorithmus erfolgte anhand eines allgemeinen Schemas für genetische Algorithmen, dessen Freiheitsgrade wir mit Hilfe von problemorientiert ausgewählten Bausteinen passend gebunden haben.

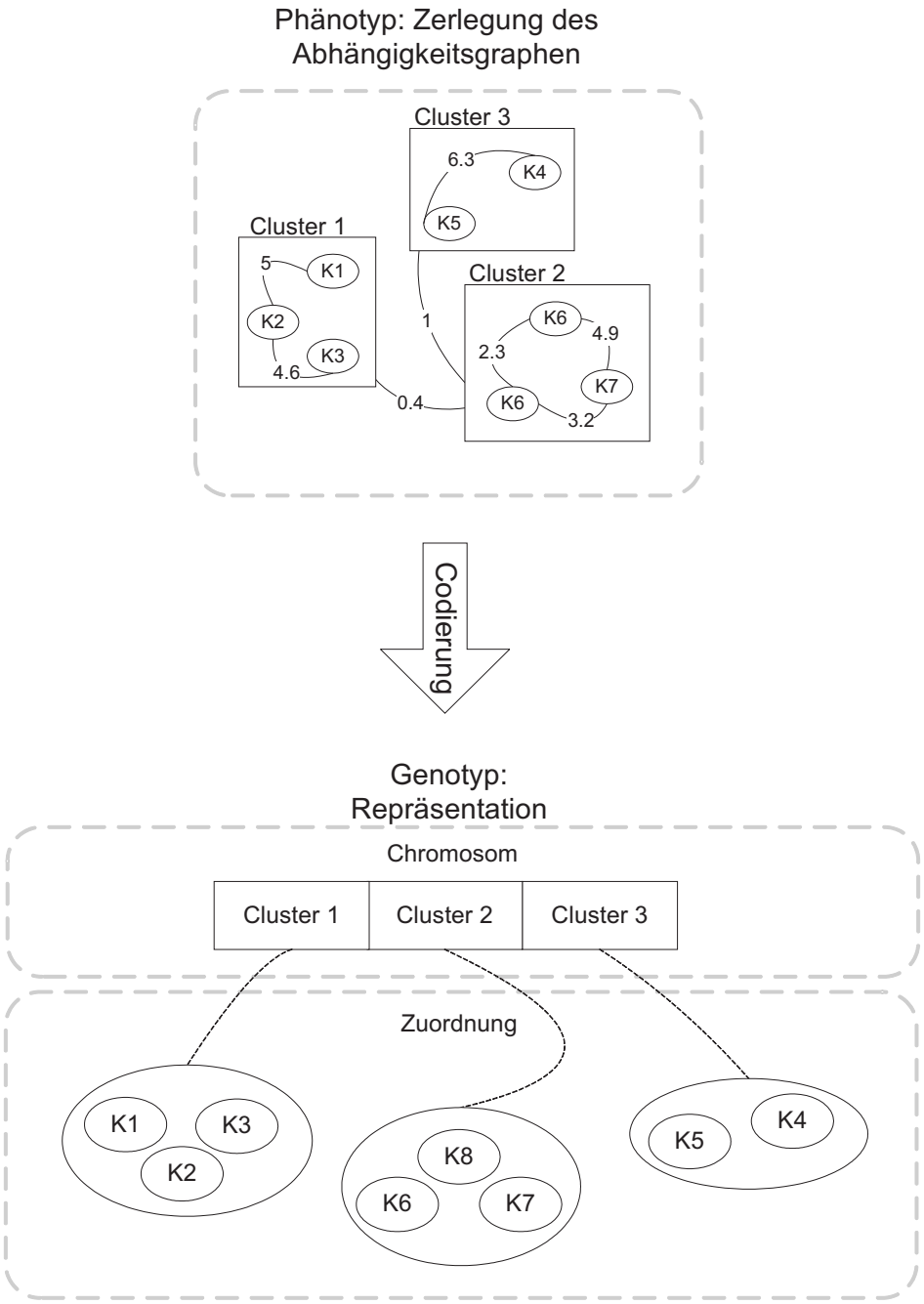
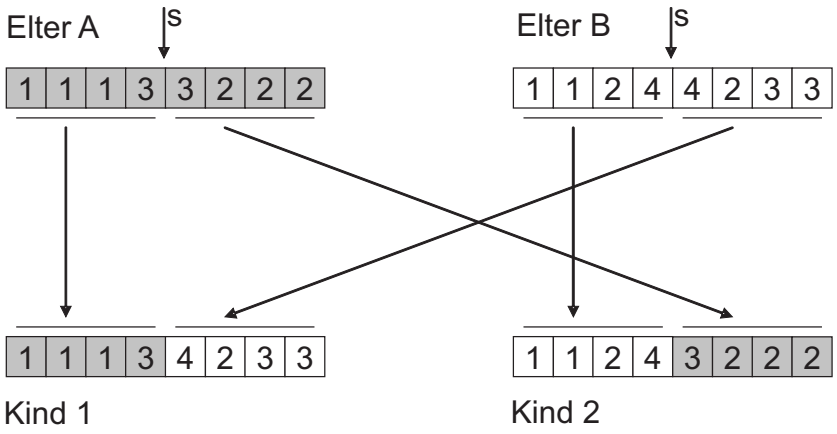
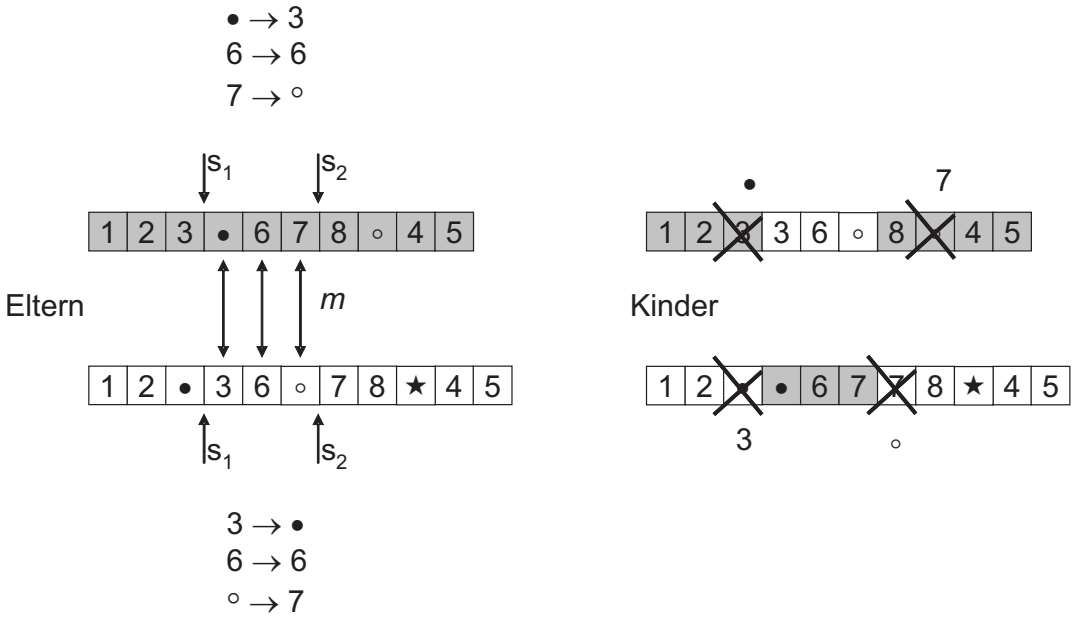


Abbildung 7.7: Repräsentation von Zerlegungen im *HGGA*



(a) Klassisch nach HOLLAND (1975)



(b) PMX nach GOLDBERG und LINGLE (1985)

Abbildung 7.8: Rekombinationsoperatoren für genetische Algorithmen

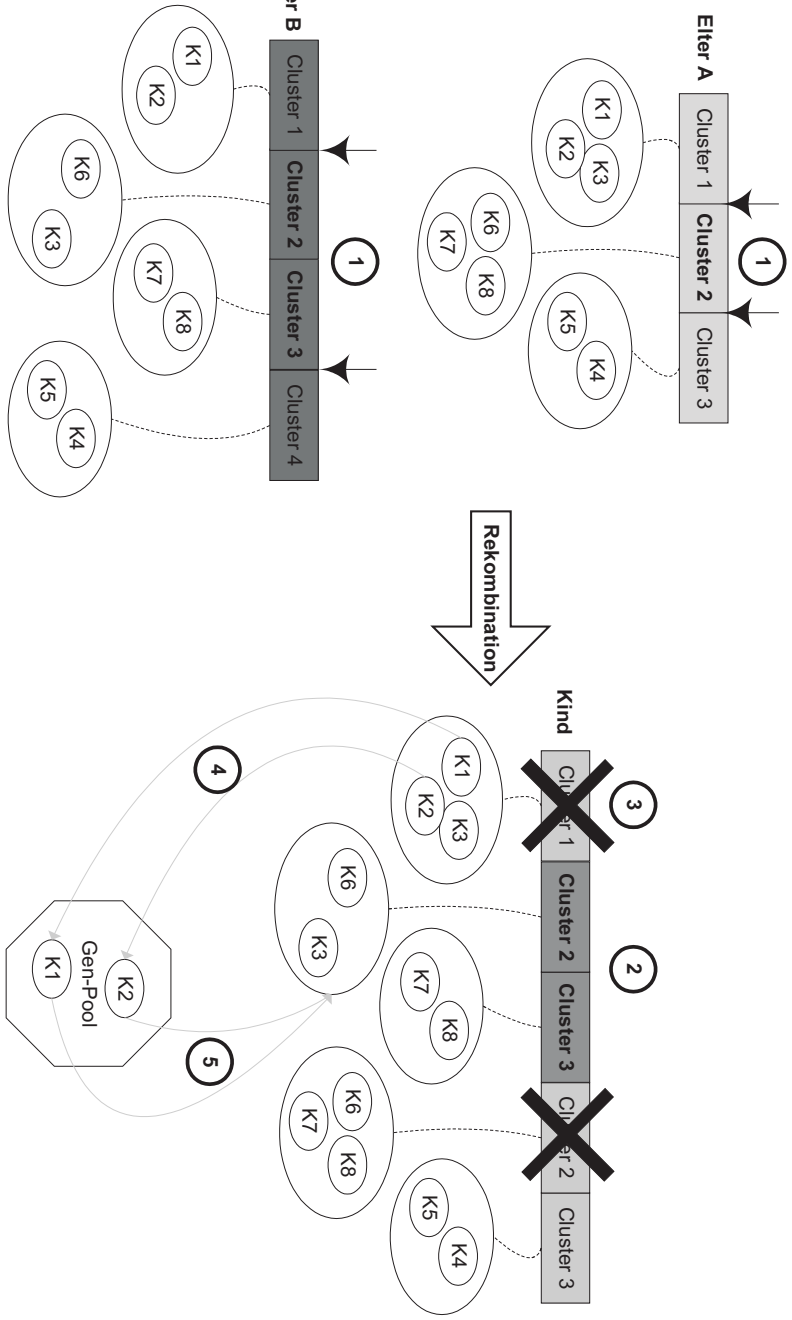


Abbildung 7.9: Rekombinationsoperator des *HGGA*

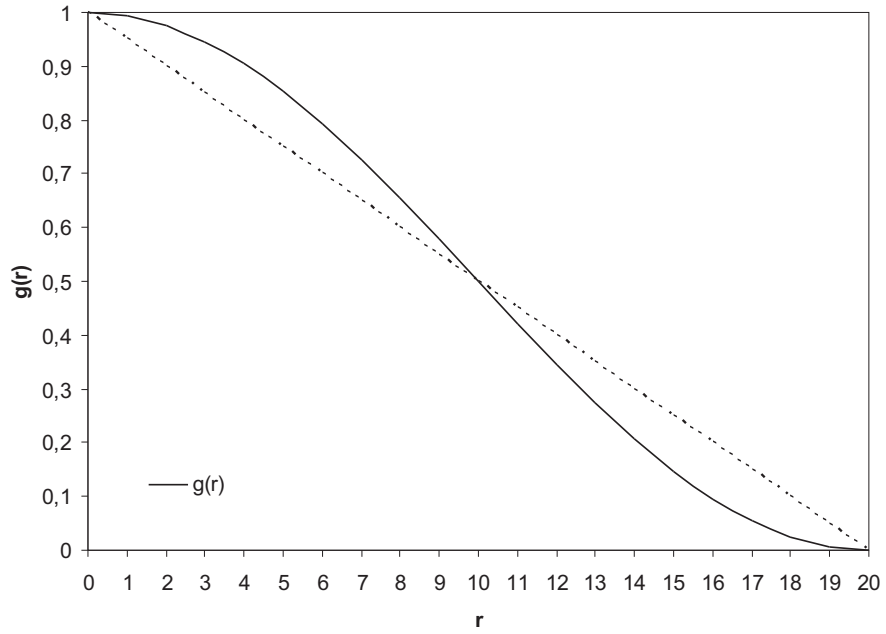


Abbildung 7.10: Randomisiertes Berücksichtigen von Kanten im *RMMST*

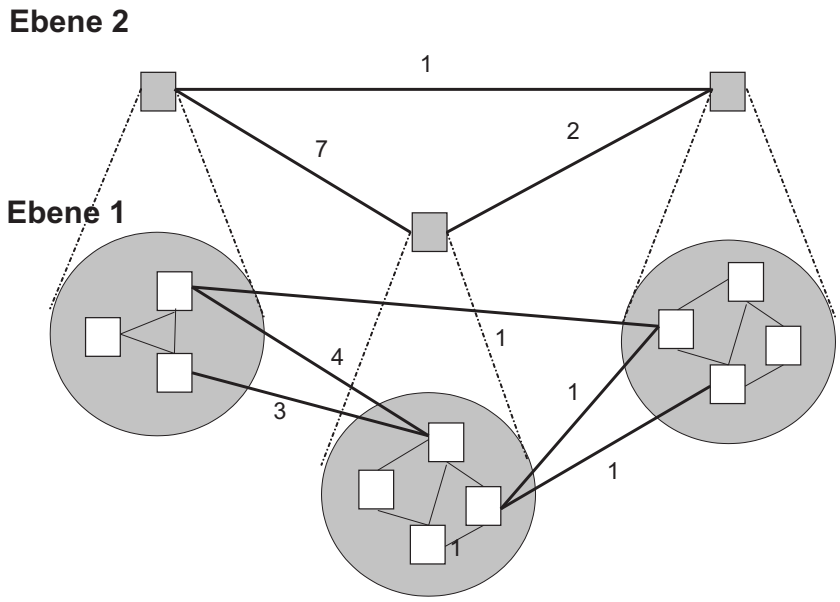


Abbildung 7.11: Hierarchische Ballungsanalyse

Kapitel 8

Evaluation

In diesem Kapitel untersuchen wir die praktische Anwendbarkeit unseres Verfahrens. Wir zeigen, wie es dazu eingesetzt werden kann, Softwareingenieuren das Verständnis eines Softwaresystems zu erleichtern.

Wir stellen dazu zunächst ein Werkzeug vor, das die in dieser Arbeit vorgestellten Techniken implementiert. Zudem eignet sich dieses Werkzeug als Experimentierplattform, die es erlaubt verschiedene Ausprägungen unseres Verfahrens – beispielsweise unterschiedliche Parametrisierungen unseres genetischen Algorithmus *HGGA* – zu erproben und miteinander zu vergleichen.

Wir stellen dann einige Fallstudien vor, an denen wir die die Anwendbarkeit unseres Verfahren erproben. Anhand dieser Fallstudien führen wir eine Reihe von Untersuchungen durch:

- Wir erproben unser Verfahren zur Klassifikation der Bauteile eines Softwaresystems auf Basis einer Mustersuche aus Kapitel 6.

- In Kapitel 7 haben wir unseren genetischen Algorithmus *HGGA* konstruiert, indem wir Freiheitsgrade eines Grundschemas für genetische Algorithmen durch geeignete Bausteine ausgestaltet haben. Um den Erfolg unserer Konstruktion zu belegen, vergleichen wir mit Hilfe von Messungen die von uns gewählte Ausprägung des *HGGA* mit alternativen Ausgestaltungen des Grundschemas.

- Wir bewerten die Qualität von Systemzerlegungen, die wir mit Hilfe unseres Verfahrens berechnen können, und vergleichen diese mit Zerlegungen, die wir mit vergleichbaren, bereits existierenden Verfahren bestimmt haben. Hierzu vergleichen wir alle berechneten Zerlegungen mit Referenzzerlegungen. Zudem untersuchen wir die Zerlegungen einiger Fallstudien noch im Detail und prüfen, in wie weit diese dem Systemverständnis dienlich sind.

- Anhand von Laufzeitmessungen zeigen wir, dass unser Verfahren auch zur Zerlegung großer Systeme mit mehr als einer Million Zeilen Quelltext eingesetzt werden kann.

8.1 Die Experimentierplattform *ACT*

Wir haben in den vorausgehenden Kapiteln eine Reihe von Techniken vorgestellt, mit denen Teilsystemstrukturen für objektorientierte Systeme berechnet werden können. Wir stellen in diesem Abschnitt die Experimentierplattform *ACT* vor, mit Hilfe derer diese Verfahren in der Praxis erprobt werden können.

Der Aufbau von *ACT* ist in Abbildung 8.1 dargestellt. Er orientiert sich an den grundlegenden Schritten unseres Ansatzes, die wir in Kapitel 4 definiert haben (vgl. dazu auch Abbildung 4.4):

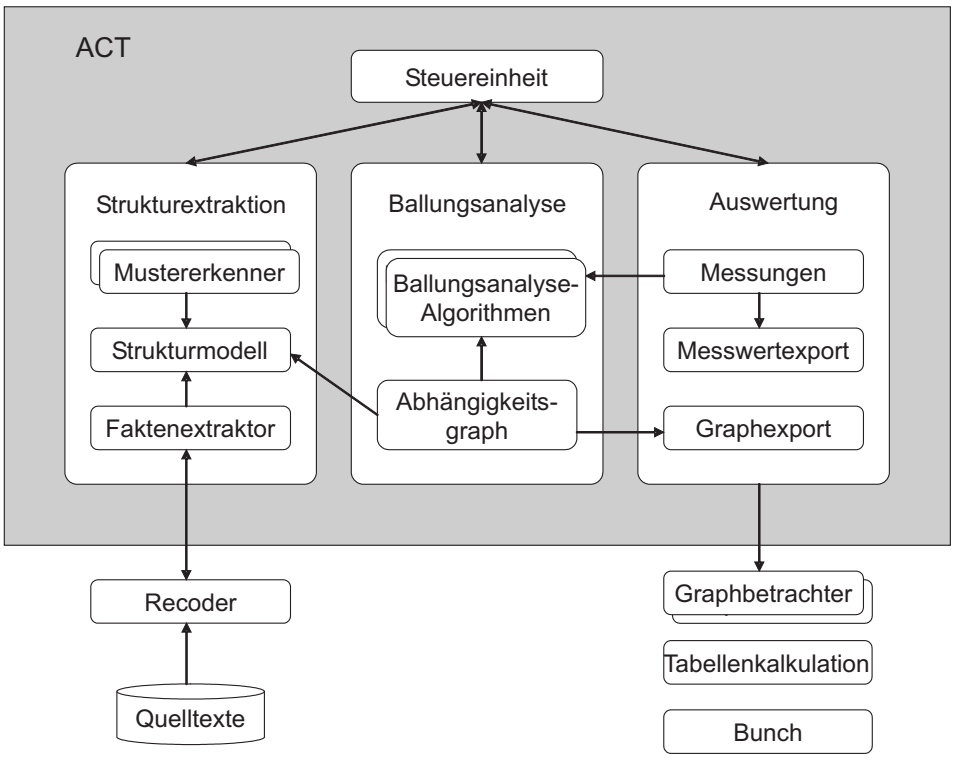


Abbildung 8.1: Architektur von *ACT*

Eine *Steuereinheit* steuert den Programmablauf von *ACT* und koordiniert das Zusammenspiel der einzelnen Teilsysteme.

Im Normalfall beginnt *ACT* damit, die Implementierungsstrukturen eines Softwaresystems aus dessen Quelltext zu extrahieren *(Strukturextraktion)*. Ergebnis dieses Schrittes ist ein Strukturmodell des Systems, wie wir es in Kapitel 5 beschrieben haben. Dieses hält *ACT* in Form eines Objektgraphen im Speicher vor. Zur Analyse der Quelltexte verwendet *ACT* die Metaprogrammier-Bibliothek *Recoder* (LUDWIG, 2002).

Anschließend erfolgt die Klassifikation der Bauteile des Systems mit Hilfe der Mustersuche. Die einzelnen Suchheuristiken, die wir in Kapitel 6 in Form prädikatenlogi-

scher Formeln spezifiziert hatten, sind in *ACT* in Form von Mustererkennern implementiert. Diese Mustererkenner sind *Java*-Klassen, die sich gemäß des *Besucher*-Musters (GAMMA et al., 1995) durch das Strukturmodell hangeln und Elemente mit Hilfe von Annotationen markieren, sobald sie auf eine passende Struktur gestoßen sind. *ACT* enthält Mustererkenner zur Klassifikation von Methoden (siehe Abschnitt 6.2), zur Identifikation der Entwurfsmuster *Fassade, Proxy, Adapter, Strategie, Fabrik* und *Kompositum* (siehe Abschnitt 6.4). Weitere Mustererkenner identifizieren Bibliotheken und bestimmte Bestandteile von Rahmenwerken (siehe Abschnitt 6.3).

Nach Abschluss der Mustersuche konstruiert *ACT* einen Abhängigkeitsgraphen. Je nach Konfiguration können dazu unterschiedliche Ähnlichkeitsmaße berücksichtigt werden, so dass neben den Maßen aus Abschnitt 7.2 auch andere Ähnlichkeitsmaße verwendet werden können. So kann *ACT* wahlweise auch die Ergebnisse der Mustersuche bei der Gewichtung der Kanten des Abhängigkeitsgraphen ignorieren und so existierende Verfahren des Standes der Technik – beispielsweise aus TRIFU (2003) oder RAYSIDE et al. (2000) – nachbilden.

Diese Abhängigkeitsgraphen können mit unterschiedlichen *Ballungsanalyseverfahren* in Teilsystemkandidaten zerlegt werden. Neben einer Implementierung des *MMST*-Algorithmus aus Abschnitt 7.3 enthält *ACT* eine auf vielfältige Weise parametrisierbare Implementierung unseres genetischen Algorithmus *HGGA*.

Die Implementierung des *HGGA* enthält austauschbare Komponenten, die es uns erlauben, die meisten der in Abbildung 7.6 dargestellten Varianten des Algorithmus abzudecken. Insbesondere können folgende Komponenten variiert werden: Qualitätsfunktionen zur Bewertung der Zerlegungen, Operatoren zur Entwicklung der Population (verschiedene Mutations- und Rekombinationsoperatoren), Selektions- und Aufbaustrategien, sowie Strategien zur Erstellung der Startpopulation. BIEHL (2004) enthält eine ausführliche Beschreibung dieser Implementierung des *HGGA*, die aufzeigt, wie die Entwurfsmuster *Befehl*, *Schablone* und *Strategie* gezielt eingesetzt werden können, um diese Variablität zu erreichen.

Die Abhängigkeitsgraphen von *ACT* können – mitsamt Informationen über berechnete Ballungen – in Dateien abgelegt werden. Dabei werden unterschiedliche Dateiformate für Graphen unterstützt, so dass die Abhängigkeitsgraphen mit externen Graphenbetrachtern, beispielsweise den kostenlos verfügbaren Graphbetrachtern *yEd*[1] und *dotty* (GANSNER et al., 1993), visualisiert oder mit anderen Werkzeugen, beispielsweise *Bunch*, bearbeitet werden können. Zudem können solche Graphen auch wieder in *ACT* eingelesen werden, so dass mit verschiedenen Ballungsanalysealgorithmen experimentiert werden kann, ohne dass eine erneute, zeitaufwendige Analyse des Quelltextes des zu untersuchenden Softwaresystems notwendig wird.

ACT enthält ein Modul *Auswertung*, welches zahlreiche Messungen durchführen kann. Neben der Aufzeichnung von Messwerten während des Ablaufs der Ballungsanalyseal-

[1] http://www.yworks.com/

Fallstudie	Kategorie	LOC	Klassen
Patterns	Programmierbeispiele	4200	389
Catalina	Webserver	64000	232
SSHTools	Netzwerksoftware	76000	507
JEdit	Texteditor	131000	805
Java AWT	GUI-Bibliothek	142000	482
Swing	GUI-Rahmenwerk	270000	2168
VVM 2000	Informationssystem (*Client-Server*)	1008000	5549

Tabelle 8.1: Übersicht der Fallstudien

gorithmen enthält das Modul auch Funktionen zum Vergleich verschiedener Zerlegungen eines Softwaresystems (siehe dazu auch Abschnitt 8.3.3). *ACT* speichert die Messwerte in Form von Tabellen, die komfortabel mit Tabellenkalkulationsprogrammen oder dergleichen ausgewertet werden können.

8.2 Fallstudien

Tabelle 8.1 enthält eine Übersicht über die Fallstudien, an denen wir unsere Verfahren erprobt haben.

Wie Tabelle 8.1 zeigt, entstammen die Fallstudien unterschiedlichsten Anwendungsdomänen. Die Größe der Fallstudien schließt sowohl kleinere Systeme, an denen wir die Qualität der berechneten Teilsystemstrukturen noch gut beurteilen können, als auch größere Systeme mit mehr als einer Million Zeilen Quelltext ein. Einige der Systeme wurden von einem oder wenigen Softwareingenieuren implementiert *(SSHTools)*, andere Systeme entstanden mit Hilfe größerer Entwicklungsteams *(Swing, VVM2000)* bzw. zahlreicher *OpenSource*-Entwickler *(jEdit)*. Die Fallstudien machen sich zudem unterschiedliche Implementierungstechniken zunutze. Die meisten Systeme sind konventionell implementierte Systeme, *VVM2000* wurde dagegen modellgetrieben entwickelt und setzt auf eine umfangreiche technische Infrastruktur auf Basis von *Enterprise JavaBeans* auf.

Gemeinsam ist allen Fallstudien ihre Implementierungssprache *Java*und ihr vergleichsweise geringes Alter. Alle Fallstudien verfügen über eine sorgfältig definierte Paketstruktur, mit der wir die Teilsystemstrukturen, die unser Verfahren berechnet, vergleichen können.

Wir schließen diesen Abschnitt mit einer Kurzbeschreibung der einzelnen Fallstudien ab:

- *Patterns:* Die Fallstudie *Patterns* besteht aus einer Menge kleinerer Programmbeispiele aus einem Lehrbuch über die Verwendung von Entwurfsmustern beim Entwickeln von *Java*-Programmen ECKEL (2003). Wir verwenden diese Fallstudie, um nachzuweisen, dass unser Ansatz zur Mustersuche aus Kapitel 6 trägt.

- *Catalina:* Die Fallstudie *Catalina* umfasst das Herzstück des Webservers *Tomcat 4*. *Tomcat* ist ein vollständig in *Java* implementierter Webserver des *Apache*-Projekts[2]. *Catalina* stellt die Referenzimplementierung der von SUN spezifizierten Ausführungsumgebung für *Java Servlets* dar. Wir erwarten aus diesem Grund, dass die Implementierung dieser Fallstudie sorgfältig durchgeführt wurde und sie über eine klare Struktur verfügt.

- *SHHTools:* Bei der Fallstudie *SSHTools*[3] handelt es sich um die *Java*-Implementierung von *Client* und *Server* der Kommunikationssoftware *SSH (Secure Socket Shell)*. *SSH* bietet dem Nutzer verschlüsselte Terminalverbindungen und Möglichkeiten zum Dateitransfer. Zudem lassen sich beliebige *IP*-Verbindungen über einen gesicherten *SSH-Kanal* abwickeln (*Forwarding)*.

- *jEdit:* Die Fallstudie *jEdit*[4] ist ein vollständig in *Java* implementierter Texteditor. *JEdit* ist ein typisches Beispiel für eine Anwendung, bei der die Benutzerschnittstelle eine dominierende Rolle einnimmt. Im Falle von *jEdit* ist diese mit Hilfe des Rahmenwerks *Swing* implementiert, welches im Lieferumfang von *Java* gehört.

- *Java AWT, Swing:* Die Fallstudien *Java AWT* und *Swing* sind Bibliotheken bzw. Rahmenwerke, die zum Lieferumfang der Standardausgabe *(J2SE)* von *Java* gehören. Beide stellen dem *Java*-Entwickler Konzepte zur Entwicklung von Anwendungen mit graphischer Benutzerschnittstelle zur Verfügung. Das umfangreichere *Swing* erweitert die Grundfunktionalität des *AWT (Abstract Window Toolkit)* durch zahlreiche zusätzliche Bauteile. Durchgehendes Konzept von *Swing* ist die Strukturierung seiner Bauteile nach dem *Model-View-Controller*-Muster. Wir analysieren *Swing* stets zusammen mit der mitgelieferten Beispielanwendung *Swingset*, die die Verwendung einer Vielzahl von Bauteilen aus *Swing* demonstriert.

- *VVM2000:* Bei der Fallstudie *VVM2000* handelt es sich um ein in *Java* implementiertes Informationssystem. *VVM2000* unterstützt ein Versicherungsunternehmen bei der Verwaltung von Versicherungsnehmern und Versicherungsverträgen. Die technologische Basis für *VVM2000* sind *Enterprise JavaBeans (EJB)*, eine Komponententechnologie für *Client-Server*-Systeme in *Java*. Die Konstruktion des Systems erfolgte nach Prinzipien des *modellzentrierten Entwurfs*. Dabei wurde zunächst die datenorientierte Sicht von *Geschäftsobjekten* (wie beispielsweise

[2] http://www.apache.org
[3] http://sourceforge.net/projects/sshtools/
[4] http://www.jedit.org

Kunde, Vertrag und dergleichen) modelliert. Mit Hilfe speziell angefertigter Codegeneratoren konnten aus diesem Modell Teile des Systems in Form von Implementierungsgerüsten automatisch erzeugt werden. In diese Gerüste fügten Programmierer dann in Handarbeit nachträglich die verbleibende Geschäftslogik ein. Charakteristikum von solchen Systemen ist – bedingt durch die Verwendung der Codegeneratoren – eine große Regelmäßigkeit der Systemaufbaus, insbesondere verfügen alle Geschäftsobjekte über dieselbe innere Struktur. Die Fallstudie *VVM2000* ist aufgrund ihres Umfangs von einer Million Zeilen Quelltext, ihrer Anwendungsdomäne und der verwendeten Technologien bei Konstruktion und Betrieb ein gutes Beispiel für moderne Systeme der industriellen Praxis.

8.3 Messungen

Zur Bearbeitung der Fallstudien und zur Durchführung der Messungen wurde ein handelsüblicher PC (Pentium-III 800 Mhz, 512 MB RAM) unter Microsoft Windows 2000 und dem SUN JDK 1.4 verwendet.

8.3.1 Mustersuche

Die Funktion der Mustersuchverfahren zur Klassifikation der Bauteile unseres Systems überprüfen wir an einer speziell dafür ausgewählten Fallstudie. Wir ziehen dazu die Programmierbeispiele eines Lehrbuchs über Entwurfsmuster heran (ECKEL, 2003). In diesen Beispielen sind alle Vorkommen von Entwurfsmustern sorgfältig dokumentiert, so dass sich leicht prüfen lässt, ob die Mustererkenner aus *ACT* zuverlässig arbeiten.

Es zeigt sich, dass *ACT* die in den Programmierbeispielen enthaltenen Muster *Fassade, Proxy, Adapter, Strategie, Fabrik* und *Kompositum* zuverlässig erkennt und markiert. Es treten dabei weder irrtümlich identifizierte Muster auf *(false positives)*, noch werden Muster übersehen *(false negatives)*.

Während der Bearbeitung der übrigen Fallstudien haben wir die Arbeitsweise der Mustersuche aus *ACT* stichprobenartig überprüft. Dabei ist allerdings nur eine Untersuchung auf *false positives* sinnvoll möglich, indem identifizierte Muster mit Hilfe des Quelltextes überprüft werden. Die Quote für *false positives* liegt für die betrachteten Fallstudien deutlich unter zehn Prozent.

8.3.2 Ausgestaltung des *HGGA*

In Abschnitt 7.4 haben wir die Konstruktion unseres genetischen Algorithmus *HGGA* zur Zerlegung von Softwaresystemen beschrieben. Dabei haben wir ein Grundschema

für genetische Algorithmen verwendet und mit passenden Bestandteilen bestückt, um ein gut funktionierendes Zerlegungsverfahren zu erhalten.

Wir untersuchen die von uns gewählte Ausgestaltung des *HGGA* im folgenden mit Hilfe von Experimenten und Messungen. Wir machen uns dabei die in Abschnitt 8.1 beschriebene Architektur von *ACT* zunutze, die es uns erlaubt, unterschiedliche Ausprägungen unseres genetischen Algorithmus zu erproben und dabei die Fortentwicklung der Individuen bzw. der Zerlegungen der Fallstudien zu begutachten. Dabei kann die Entwicklung der Qualität der Zerlegungen gemäß der Bewertungsfunktion f (siehe Gleichungen 7.29 und 7.30) in Diagrammen dargestellt werden. Für die einzelnen Ausprägungen ermitteln wir in jeder Generation die Qualität des jeweils besten Individuums und der durchschnittlichen Qualität der gesamten Population. Um Zufallseffekte zu reduzieren, ziehen wir für alle Qualitätswerte das Mittel aus je zehn unabhängigen Programmläufen heran.

Wir geben im folgenden aus Platzgründen lediglich Messungen für einige ausgewählte Fallstudien an. BIEHL (2004) belegt, dass sich die Beobachtungen, die wir in den folgenden Abschnitten anhand dieser Fallstudien machen, auf weitere Fallstudien in vergleichbarer Form übertragen lassen.

Operatoren. Eine der Problemstellungen in Abschnitt 7.4 bestand darin, geeignete Operatoren zur Entwicklung einer Population von Zerlegungen auszuwählen. Abbildung 8.2 zeigt die Entwicklung der Qualität der Zerlegungen der Fallstudie *Catalina* beim Einsatz unterschiedlicher Operatoren.

Catalina wurde dazu unter der Verwendung der Standardkonfiguration des *HGGA* (siehe Anhang B.3) zerlegt, dabei wurden allerdings jeweils unterschiedliche Operatorensätze verwendet: Die Messkurven *Mutation (Bestes)* bzw. *Mutation (Mittel)* geben die aus zehn Programmläufen gemittelte Qualität der jeweils besten Individuen bzw. die durchschnittliche Qualität der Population wieder, wobei die Population nur der Ballungsmutation unterworfen wurde. Es wurden weder Adoptionsmutationen noch Rekombinationen durchgeführt. Entsprechend geben die Kurven *Rekombination (Bestes)* bzw. *Rekombination (Mittel)* die entsprechenden Qualitätswerte für Populationen an, die nur mit Hilfe von Rekombinationen fortentwickelt wurden. Die Kurven *HGGA (Bestes)* und *HGGA (Mittel)* veranschaulichen die Fortentwicklung der Qualität der Populationen, wie sie entsteht, wenn Ballungsmutationen, Adoptionsmutationen und Rekombinationen verwendet werden.

Es zeigt sich, dass die Kombination aller Operatoren sowohl die beste Entwicklung der Qualität der Individuen hervorbringt als auch für das zügigste Konvergenzverhalten sorgt. Die Mutation trägt signifikant zur Ausbildung hochwertiger Individuen bei. Die Rekombination sorgt erwartungsgemäß dafür, dass sich bereits im genetischen Material vorhandene positive Eigenschaften einzelner Individuen rasch innerhalb der Population ausbreiten. Alle Operatoren arbeiten wunschgemäß so, dass sie im Mittel keinen zerstörerischen Einfluss auf das genetische Material der Population haben.

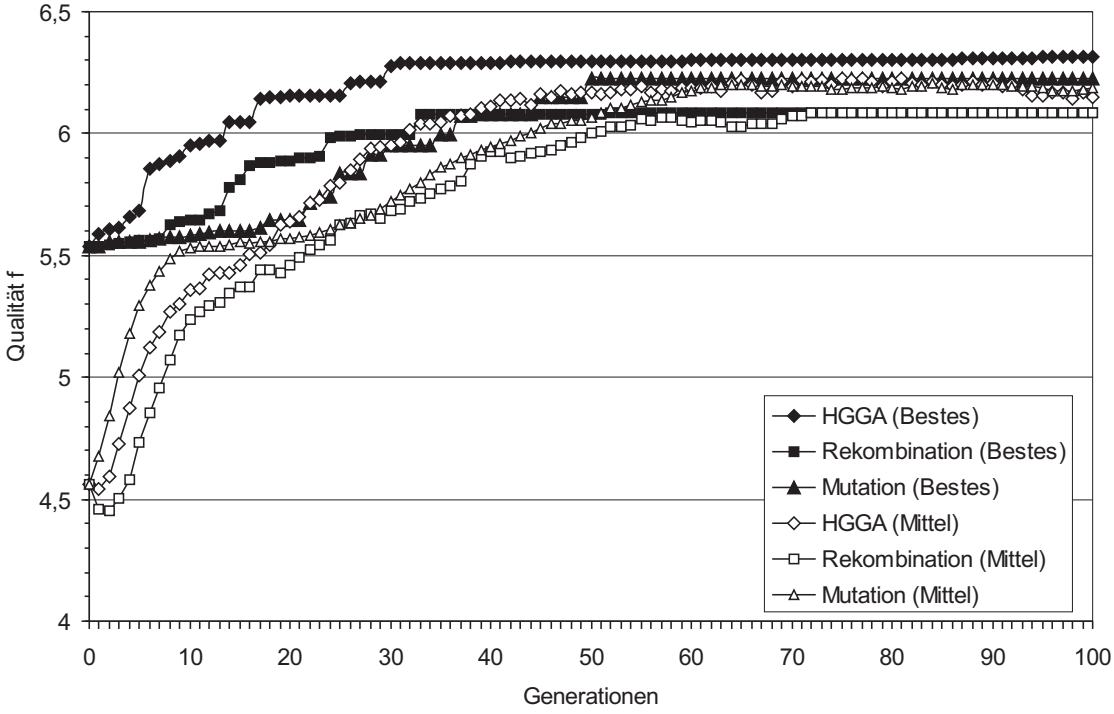


Abbildung 8.2: Auswirkungen der Operatoren des *HGGA (Catalina)*

Selektion. Die Selektionstrategie (siehe Abschnitt 7.4.6) eines genetischen Algorithmus hat ebenfalls Einfluss auf die Entwicklung der Qualität der Individuen. Abbildung 8.3 zeigt die Entwicklungen am Beispiel der Fallstudie *Catalina* für unterschiedliche Selektionsstrategien.

Die Messkurven zu *Glücksrad Plus* und *Turnier Plus* zeigen das Verhalten des *HGGA* bei Verwendung einer $(\mu+\lambda)$-Strategie, bei der die Individuen der jeweils nächsten Generation aus den Eltern *und* den Nachkommen durch Turniere bzw. mit der Glücksrad-Selektion ausgewählt werden.[5] Die Messkurven zu *Glücksrad Komma* entstanden durch Verwendung der (μ,λ)-Strategie, bei der die nächste Generation lediglich aus der Menge der per Rekombination erzeugten Nachkommen ausgewählt wird. Die Messkurven zeigen deutlich, dass bei der Verwendung der (μ,λ)-Strategie nicht nur eine Verschlechterung der mittleren Qualität einer Generation in Kauf genommen werden muss, sondern dass auch die jeweils Besten einer Folgegeneration durchaus hinter denen der Elterngeneration zurückbleiben können.

Die mit *Deterministisch* bezeichneten Messkurven zeigen die schlechteste aller Mög-

[5] Obwohl wir in Abschnitt 7.4.6 vorausgesetzt hatten, dass die Glücksrad-Selektion eigentlich ein Verhältnismaß als Bewertungsfunktion f voraussetzt, haben wir für diese Untersuchung einfach unsere Qualitätsfunktion aus Gleichung 7.29 eingesetzt, die lediglich über eine Ordinalskala verfügt.

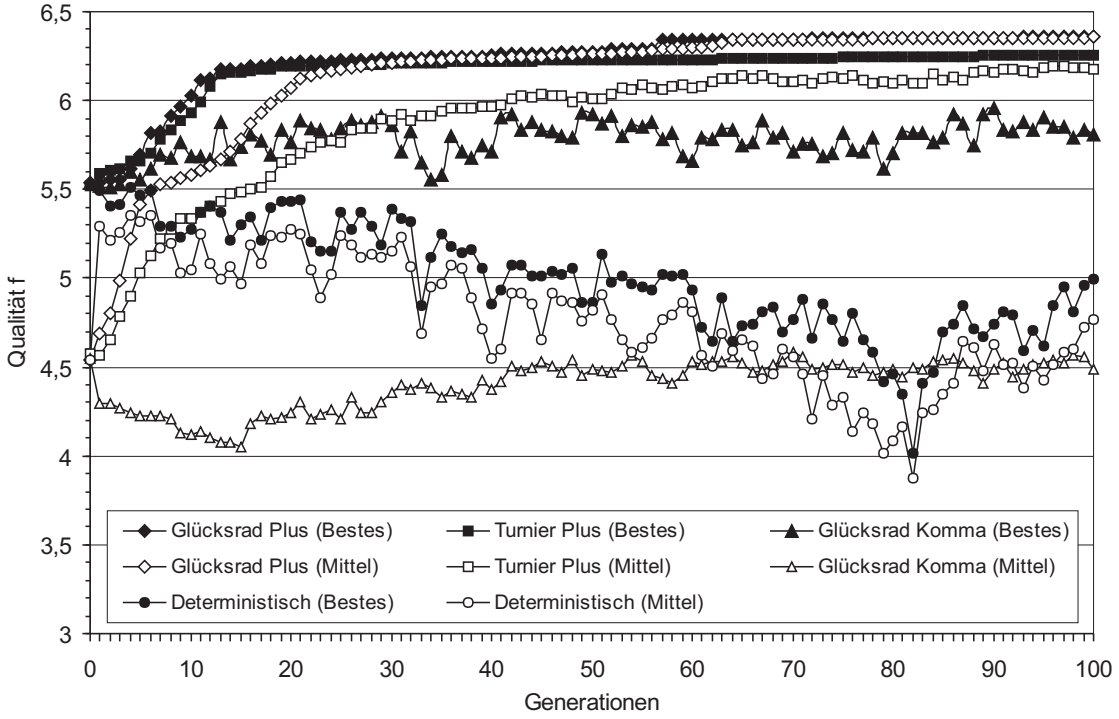


Abbildung 8.3: Auswirkungen unterschiedlicher Selektionsstrategien *(Catalina)*

lichkeiten: völlig deterministisch werden zunächst einige der besten Individuen einer Generation als Eltern ausgewählt. Durch Rekombination entstehen daraus die Individuen der Folgegeneration. Es zeigt sich, dass das Element des Zufalls eine wesentliche Rolle spielt. Es unterstützt, dass neue, gegebenenfalls vielversprechende Bereiche des Suchraums erschlossen werden.

Insgesamt weist die Glücksrad-Selektion mit $(\mu + \lambda)$-Strategie das beste Verhalten auf. Dicht dahinter folgt die Turnier-Selektion, ebenfalls in Verbindung mit einer $(\mu + \lambda)$-Strategie. Wir verwenden im *HGGA* die Turnier-Selektion, da wir dabei auf das Erstellen einer "Schattenpopulation", wie sie die Glücksrad-Selektion vorraussetzt, verzichten können. Eine solche Schattenpopulation ist bei der Glücksrad-Selektion erforderlich, da die Anordnung der Eltern vor der Rekombination "durchmischt" werden sollte, um den Suchraum möglich breit zu durchkämmen. Bei der Turnier-Selektion ergibt sich diese "durchmischte" Anordnung automatisch durch das Losverfahren bei der Besetzung der Turniere (FALKENAUER, 1998).

Startpopulation. Gestaltungsspielraum bestand auch hinsichtlich der Strategie zur Erzeugung der Startpopulation. Abbildung 8.4 zeigt, wie die Entwicklung des *HGGA* von der Wahl der Startpopulation beeinflusst wird. Die rein zufällige Erzeugung einer

Startpopulation durch verschieden große Zusammenhangskomponenten (Messkurven *Zufall*) führt ebenso zum Erfolg wie Startpopulationen, die mit Hilfe des *RMMST* bzw. aus einer Kombination beider Strategien (Messkurven *HGGA*) erzeugt wurden. Startpopulationen, die mit dem *RMMST* erzeugt wurden, sind anfangs von etwas homogenerer Qualität (d.h. die durchschnittliche Qualität der Individuen liegt etwas näher an der Qualität der jeweils Besten einer Generation). Startpopulationen mit einem guten Anteil rein zufällig erzeugter Individuen bergen dagegen das Potenzial für etwas bessere Lösungen in späteren Generationen. Insgesamt erweist sich als Startpopulation ein Mix aus rein zufällig erzeugten Individuen und *RMMST*-Lösungen als sehr gut geeignete Grundlage für den *HGGA*.

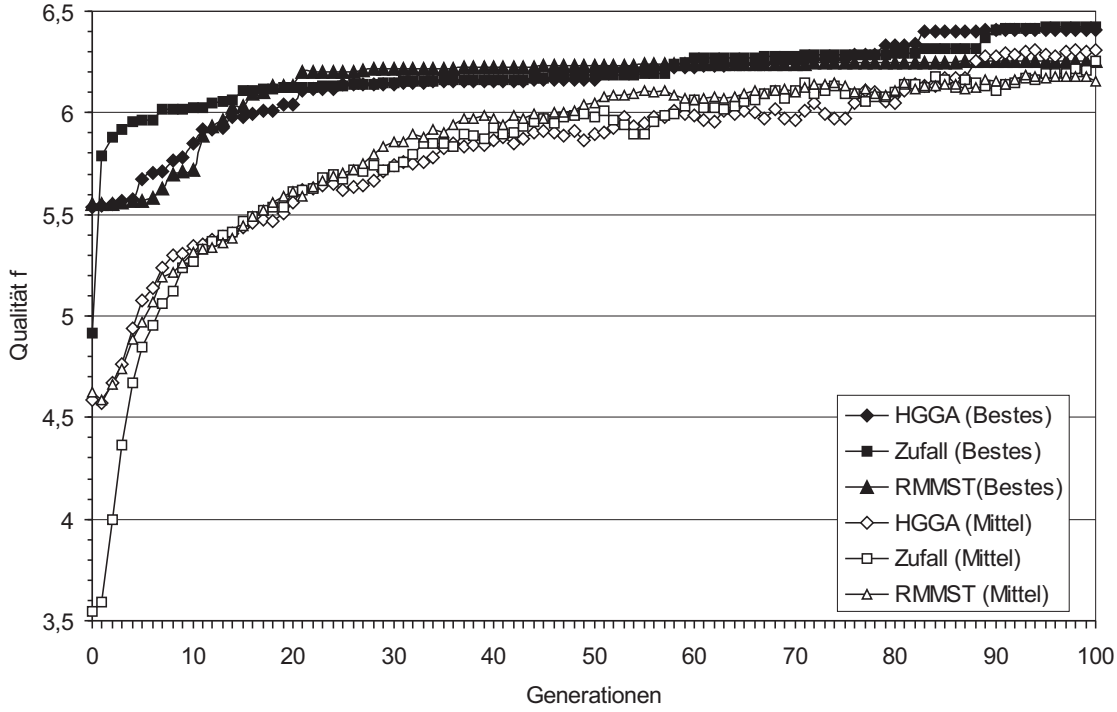


Abbildung 8.4: Auswirkungen unterschiedlicher Strategien für die Startpopulation *(Catalina)*

Populationsstärke und Anzahl der Generationen. Weiterer Gestaltungsspielraum ergibt sich aus zwei Parametern, die sowohl die Laufzeit als auch die Ergebnisqualität des *HGGA* maßgeblich beeinflussen. Dies sind die Anzahl der Individuen der Population μ und die Anzahl der Generationen g_{max}, die die Individuen der Population durchlaufen.

In jeder der g_{max} Generationen müssen μ Individuen den Rekombinations- und Mutationsoperatoren des *HGGA* unterworfen werden. Zudem muss im Verlauf der Selektion für jedes Individuum die Qualitätsfunktion f berechnet werden. Der Aufwand zur Berechnung unserer Qualitätsfunktion aus den Gleichungen 7.29 und 7.30 beträgt $O(n^2)$, wobei $n = |V|$. Der Aufwand für die Rekombination zweier Individuen bzw. für Mutation eines Individuums liegt ebenfalls in $O(n^2)$. Insgesamt liegt der Aufwand für den *HGGA* somit in $O(g_{max}\mu n^2)$.

Der Frage, wie die Ergebnisqualität des *HGGA* von μ und g_{max} abhängt, können wir nur experimentell nachgehen. Generell gilt: je mehr Generationen und je umfangreicher die Population des *HGGA* ausfällt, desto bessere Ergebnisse dürfen wir erwarten.

Abbildung 8.5 zeigt den Zusammenhang zwischen der Populationsstärke μ und der Qualität f des besten Individuums der Population nach $g_{max} = 1000$ Generationen am Beispiel der Fallstudie *SSHTools*.[6]

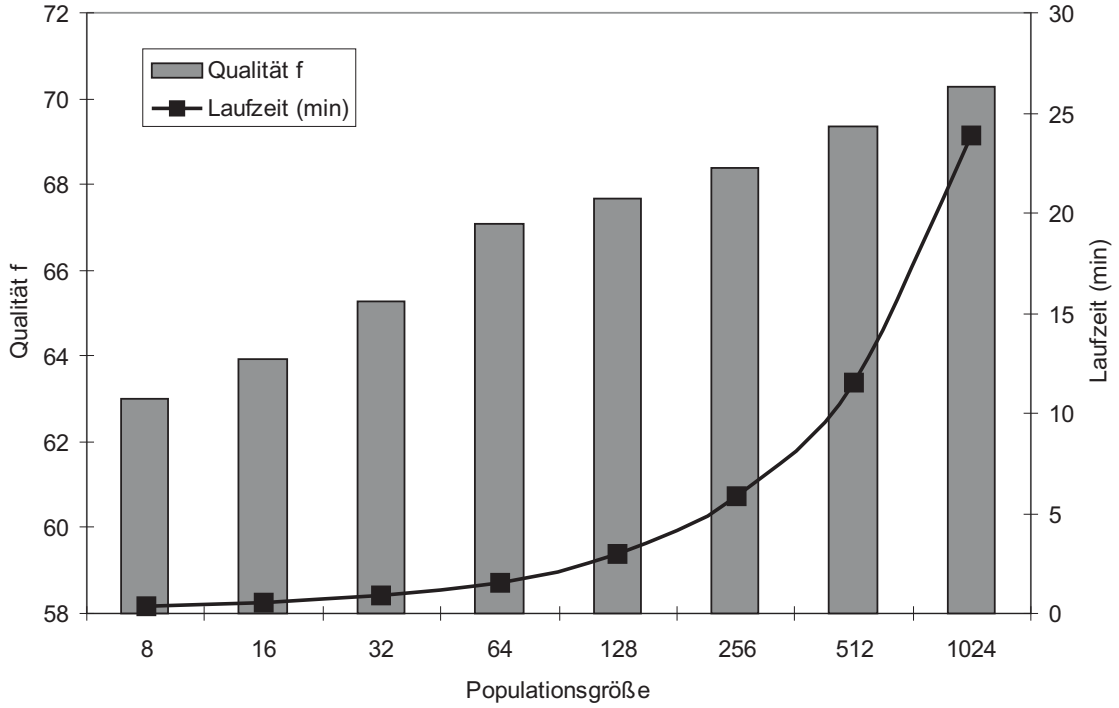


Abbildung 8.5: Auswirkungen der Populationsgröße auf Laufzeit und Qualität *(SSHTools)*

Es wird deutlich, dass die Qualität f des jeweils besten Individuums nur geringfügig zu-

[6]Die einzelnen Messwerte in Abbildung 8.5 stellen das Mittel aus jeweils 25 unabhängigen Programmläufen dar.

nimmt, obwohl die Populationsstärke – und damit auch die Laufzeit – jeweils verdoppelt werden.

Experimente zeigen, dass mit ansteigender Populationsstärke auch die Anzahl der Generationen erhöht werden muss, damit alle Individuen ausreichend Gelegenheit zu Mutation und Rekombination erhalten und sie so gegebenenfalls ihren Beitrag zur bestmöglichen Lösung leisten können.

Zusätzlich besteht ein Zusammenhang zwischen der Größe des Abhängigkeitsgraphen (hinsichtlich der Anzahl seiner Knoten $n = |V|$) und der Anzahl der Generationen, die zur Berechnung einer guten Lösung nötig sind. Wir führen dies darauf zurück, dass im Rahmen der Mutation und Rekombination möglichst viele Ballungen in den einzelnen Individuen berücksichtigt werden sollten.

Abbildung 8.6 zeigt die Entwicklung der Qualitätswerte für die Fallstudie *Swing* über $g_{max} = 2200$ Generationen hinweg. Im Vergleich zur Entwicklung der Qualitätswerte für die Fallstudie *Catalina* mit ca. 230 Klassen (Messkurve *HGGA* in Abbildung 8.2) flacht die Messkurve für *Swing* mit 2200 Klassen erst sehr spät, nämlich nach über 500 Generationen deutlich ab.

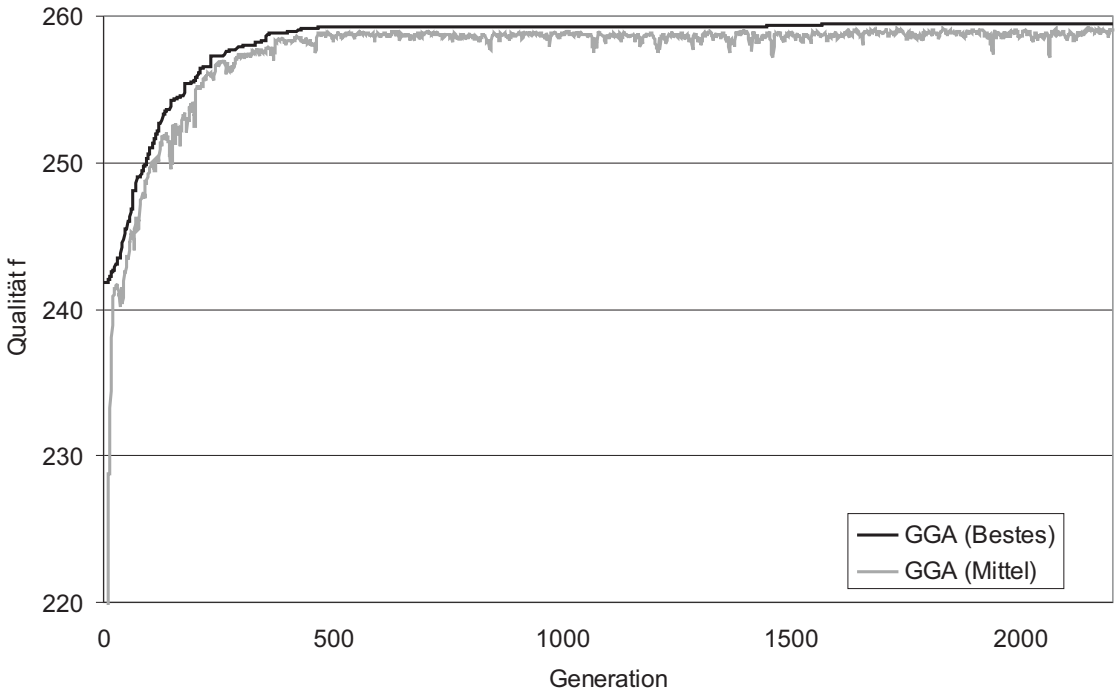


Abbildung 8.6: Konvergenz der Qualitätswerte bei *Swing*

Für die Fallstudien, die wir im Rahmen dieser Arbeit betrachtet haben, führt folgende Faustregel zu guten Zerlegungen, ohne dass die dafür erforderliche Rechenzeit die

Anwendbarkeit des Verfahrens gefährdet: Wir wählen $\mu = 100$ und $g_{max} = \max(\mu, |V|)$.

Im Zusammenhang mit den Parametern μ für die Populationsstärke und g_{max} für die Anzahl der Generationen ist auch die Frage nach der *Konsistenz* der Ergebnisse des *HGGA* von Interesse: Wie ähnlich sind sich Zerlegungen, die aus unterschiedlichen Programmläufen des *HGGA* gewonnen werden können?

Zur Untersuchung dieser Frage benötigen wir zunächst ein Maß, mit dessen Hilfe wir die Ähnlichkeit zwischen je zwei unterschiedlichen Zerlegungen ausdrücken können. Wir können hierfür beispielsweise die Anzahl von Verschmelzungs- bzw. Verschiebeoperationen verwenden, die notwendig sind, um die eine Zerlegung in die jeweils andere zu überführen. TZERPOS und R.C.HOLT (1999) definieren hierzu die so genannte *MoJo-Distanz*[7]: Sei $Op(C_1, C_2)$ die minimale Anzahl von Operationen, die notwendig sind, um die Zerlegung C_1 in die Zerlegung C_2 zu überführen, dann ergibt sich die *MoJo*-Distanz aus

$$MoJo(C_1, C_2) = MoJo(C_2, C_1) = \min(Op(C_1, C_2), Op(C_2, C_1)) \tag{8.1}$$

Als Operationen kommen das *Verschieben* eines Knotens von einer Teilmenge in eine andere, sowie das *Verschmelzen* von zwei Teilmengen in Frage. Beide Operationen werden dabei gleich gewichtet. Ein evtl. notwendiges *Zerteilen* von Teilmengen kann durch eine gewisse Anzahl von Verschiebeoperationen ausgedrückt werden. Zur Berechnung von *Op* geben TZERPOS und R.C.HOLT (1999) einen geeigneten heuristischen Algorithmus an.

Die MoJo-Distanz kann noch normiert werden, indem man sie auf die Anzahl der Knoten bezieht:

$$MoJo_N(C_1, C_2) = \frac{MoJo(C_1, C_2)}{|V|} \tag{8.2}$$

Wir beachten, dass große Werte von *MoJo* bzw. $MoJo_N$ bedeuten, dass sich die Zerlegungen unähnlich sind, kleine bedeuten, dass sie sich ähnlich sind.

Abbildung 8.7 zeigt die normierten *MoJo*-Distanzen zwischen je zwei aus 25 Zerlegungen, die aus unabhängigen Läufen des *HGGA* für die Fallstudie *SSHTools* mit $g_{max} = 100$ gewonnen wurden. Dunkle Färbungen stehen dabei für eine hohe Übereinstimmung der Zerlegungen des entsprechenden Gitterpunktes, helle Flächen deuten auf eine schlechte Übereinstimmung hin. Diagramme dieser Art lassen sich bequem heranziehen, um zu bewerten, wie zuverlässig die Ergebnisse des *HGGA* sind. Hohe Diskrepanzen zwischen einzelnen Zerlegungen aus unterschiedlichen Läufen deuten darauf hin, dass der *HGGA* nicht zuverlässig eine gute Zerlegung des Systems finden kann. In diesem Fall sollte eine

[7] engl. *Move-and-Join-Distance*

höhere Anzahl von Generationen g_{max} gewählt werden, ggf. kann auch die Populationsstärke μ erhöht werden. Eine ähnliche Betrachtung kann man auch für die Individuen innerhalb einer Population durchführen.

Abbildung 8.7 zeigt für *SSHTools* bereits eine gute Konsistenz der Zerlegungen. Die Literatur betrachtet Zerlegungen mit einer normierten MoJo-Distanz kleiner 0,33 bereits als sehr ähnlich (TZERPOS und R.C.HOLT, 1999; MITCHELL und MANCORIDIS, 2001).

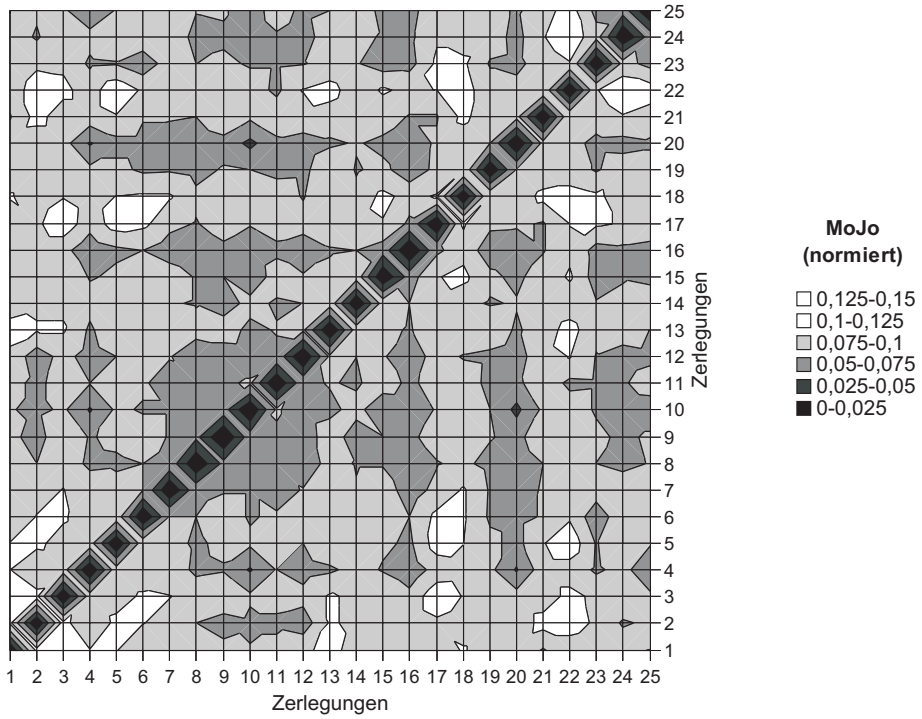


Abbildung 8.7: Konsistenz des *HGGA*: normierte MoJo-Distanz zwischen Zerlegungen verschiedener Läufe *(SSHTools)*

8.3.3 Vergleich unterschiedlicher Zerlegungsverfahren

Wir belegen in diesem Abschnitt die Tauglichkeit der in Kapitel 7 vorgestellten Verfahren zur Ballungsanalyse anhand verschiedener Messwerte für unsere Fallstudien. Neben Angaben zur Laufzeit der Algorithmen sind wir in erster Linie an objektiven Messdaten zur Beschaffenheit der berechneten Systemzerlegungen interessiert.

Vergleich mit Referenzzerlegungen. Um diese Messdaten besser beurteilen zu können, vergleichen wir sie mit Messdaten, wie wir sie mit Hilfe von Verfahren, die den bisherigen Stand der Technik widerspiegeln, gewinnen können. Einer der prominentesten Vertreter des aktuellen Standes der Technik ist das Werkzeug *Bunch*, welches wir in den Abschnitten 3.3 und 7.4 bereits ausführlich beschrieben haben. Die ebenfalls in Abschnitt 3.3 beschriebenen Ansätze von RAYSIDE et al. (2000) und TRIFU (2001) zur Zerlegung von *Java*-Systemen bieten ebenfalls interessante Vergleichsmöglichkeiten.

Eine in der Literatur verbreitete Möglichkeit, die Qualität von Systemzerlegungen objektiv zu bewerten, besteht darin, die berechneten Zerlegungen mit so genannten *Referenzzerlegungen* zu vergleichen. Unter einer Referenzzerlegung verstehen wir eine Zerlegung eines Softwaresystems, die von einem oder mehreren Softwareingenieuren durchgeführt wurde, die über weitgehende Kenntnisse über den Aufbau des betreffenden Systems verfügen. Leider sind allgemein etablierte Referenzzerlegungen, häufig auch *Expertenzerlegungen* genannt, für größere objektorientierte Systeme nicht erhältlich.

Eine mögliche Alternative besteht darin, die hierarchische Paketstruktur, wie sie in moderneren *Java*-Systemen in der Regel zu finden ist, als Referenzzerlegung zu verwenden. Dies ist jedoch nicht unproblematisch. Paketstrukturen werden häufig nicht nach strengen Regeln konzipiert, sondern häufig wird die Zuordnung von Klassen zu Paketen intuitiv durch die Entwickler des Systems vorgenommen. Hinzukommt, dass es sich bei der Paketstruktur um eine *gewachsene* Struktur handelt, welche Charakteristika aus der Entwicklungshistorie des Softwaresystemes widerspiegelt. So spiegelt sich in vielen Systemen die Aufgabenverteilung bei der Softwareentwicklung auf verschiedene Entwicklungsteams wider.

Trotz gut funktionierender Zerlegungsverfahren können so erhebliche Abweichungen zwischen berechneten Strukturen und den gewachsenen Referenzzerlegungen entstehen. Trotzdem hat sich der Vergleich mit gewachsenen Referenzzerlegungen in der Literatur als fester Bestandteil bei der Bewertung von Verfahren zur Zerlegung von Systemen etabliert, so dass wir auch bei der Bewertung unserer Verfahren davon Gebrauch machen wollen. Wir ergänzen diese Bewertung allerdings in Abschnitt 8.4 noch um eine detaillierte Begutachtung der Ergebnisse, die auch die manuelle Inspektion der Quelltexte der Systeme einschließt.

Für den Vergleich zwischen Referenzzerlegungen und berechneten Zerlegungen bieten sich so genannte *Distanzmaße* an. Ein solches haben wir mit der (normierten) *MoJo*-Distanz im vorangehenden Abschnitt bereits definiert. Da wir den Vergleich zwischen Referenzzerlegungen und berechneten Zerlegungen als Indikator für die Qualität eines Zerlegungsverfahren nutzen wollen, definieren wir mit Hilfe der normierten *MoJo*-Distanz ein passendes Qualitätsmaß: die *MoJo*-Qualität einer berechneten Zerlegung C_{ber} ergibt sich in Bezug auf eine Referenzzerlegung C_{ref} durch:

$$MoJo\_Q(C_{ber}) = (1 - Mojo\_N(C_{ber}, C_{ref})) \times 100 \tag{8.3}$$

Größere Werte für *MoJo_Q* bedeuten eine hohe Qualität der berechneten Zerlegung, insbesondere bedeutet $MoJo\_Q = 100$, dass die berechnete Zerlegung und die Referenzzerlegung genau übereinstimmen.

Eine weitere Möglichkeit, die Qualität einer berechneten Zerlegung zu bestimmen, ist in den Maßen *Zuverlässigkeit*[8] und *Vollständigkeit*[9] gegeben (ANQUETIL et al., 1999).

Die Zuverlässigkeit drückt aus, zu welchem Anteil Knotenpaare innerhalb von Ballungen in der berechneten Zerlegung auch Knotenpaare innerhalb von Ballungen in der Referenzzerlegung sind. Enthält die berechnete Zerlegung zahlreiche Ballungen, die Elemente gemeinsam gruppiert, obwohl diese in der Referenzzerlegung in getrennten Ballungen liegen, so verschlechtern sich die Zuverlässigkeitswerte des Ballungsanalyseverfahrens.

Die Vollständigkeit gibt an, zu welchem Anteil innere Knotenpaare von Ballungen in der Referenzzerlegung auch innere Knotenpaare in der berechneten Zerlegung sind. Ordnet das Ballungsanalyseverfahren in der berechneten Zerlegung Elemente unterschiedlichen Ballungen zu, obwohl diese in der Referenzzerlegung zusammen in einer Ballung liegen, so sinken die Vollständigkeitswerte.

Sei P_{intern} die Menge der inneren Knotenpaare einer Zerlegung C:

$$P_{intern} = \{(v_1, v_2) \quad | \quad v_1 \text{ und } v_2 \text{ sind Elemente derselben Ballung } C_i\} \tag{8.4}$$

Dann berechnet sich die Zuverlässigkeit einer berechneten Zerlegung C_{ber} als

$$Prec(C_{ber}) = \frac{|P_{intern}(C_{ber}) \cap P_{intern}(C_{ref})|}{|P_{intern}(C_{ber})|} \tag{8.5}$$

und ihre Vollständigkeit als

$$Rec(C_{ber}) = \frac{|P_{intern}(C_{ber}) \cap P_{intern}(C_{ref})|}{|P_{intern}(C_{ref})|} \tag{8.6}$$

Beiden Maßen ist gemein, dass höhere Werte als besser gelten können, da die berechnete Zerlegung C_{ber} für höhere Werte dichter an der Referenzzerlegung C_{ref} liegt. Gilt $Prec(C_{ber}) = Rec(C_{ber}) = 100$, dann sind berechnete Zerlegung und Referenzzerlegung identisch.

Tabelle 8.2 enthält die Maße *Prec*, *Rec*, *MoJo* und *MoJo_Q* für folgende Zerlegungen unserer Fallstudien:

[8] engl. *precision*
[9] engl. *recall*

- Die mit *MMST* bezeichneten Systemzerlegungen entstanden mit Hilfe des in Abschnitt 7.3 beschriebenen *MMST*-Algorithmus. Dieser erhielt als Eingabe jeweils einen Abhängigkeitsgraphen gemäß der in Kapitel 7 beschriebenen Konstruktion. Dies bedeutet auch, dass der Konstruktion des Abhängigkeitsgraphen eine Mustersuche vorausging, deren Ergebnisse in die Gewichtung der Abhängigkeiten einflossen.

- Die mit *HGGA* bezeichneten Systemzerlegungen entstanden nach dem selben Prinzip. Anstelle des *MMST*-Algorithmus kam jedoch unser in Abschnitt 7.4 beschriebener genetischer Algorithmus zur Ballungsanalyse zum Einsatz. Bei allen Fallstudien wählten wir dabei für die Populationsstärke $\mu = 100$, die Anzahl der Generationen g_{max} wählten wir gemäß unserer Faustregel: $g_{max} = \max(\mu, |V|)$.

- Die mit *Bunch* bezeichneten Zerlegungen des Systems wurden mit Hilfe des Werkzeuges *Bunch* gewonnen. Da *Bunch*, wie bereits in Abschnitt 3.3 erwähnt, kein Werkzeug zur Strukturextraktion enthält, erhält *Bunch* als Eingabe den Abhängigkeitsgraphen gemäß unserer Konstruktion aus Kapitel 7. Wir betrieben *Bunch* stets mit denselben Parametern für die Populationsstärke und die Generationenzahl, wie wir sie auch für den *HGGA* wählten.[10]

- Die mit *NA*[11] bezeichneten Zerlegungen wurden ebenfalls mit dem *MMST*-Algorithmus gewonnen. Dieser erhielt als Eingabe einen Anhängigkeitsgraphen, der ohne die spezielle Kantengewichtung auf Basis der vorangestellten Mustersuche konstruiert wird. Diese Vorgehensweise ist mit den Ansätzen von RAYSIDE et al. (2000) bzw. TRIFU (2001) vergleichbar.

Tabelle 8.2 und die Grafik in Abbildung 8.8 zeigen, dass unser hybrides Verfahren, bestehend aus der Kombination von Mustersuche und einer anschließenden Ballungsanalyse mittels des genetischen Algorithmus *HGGA*, Systemzerlegungen berechnet, die bei allen Fallstudien die größte Ähnlichkeit zur Paketstruktur aufweisen. Dies gilt zumindest hinsichtlich der Zuverlässigkeit *(Prec)* und der normierten *MoJo*-Qualität *(MoJo_Q)*. Diese Werte verschlechtern sich geringfügig – im Falle von *jEdit* und *VVM2000* sogar deutlich – wenn wir zur Ballungsanalyse statt des *HGGA* unseren *MMST*-Algorithmus einsetzen.

Verzichten wir – wie beim Verfahren *NA* – auf die Mustersuche und die entsprechende Anpassung der Gewichte im Abhängigkeitsgraphen, so verschlechtern sich die Zuverlässigkeitsmaße ebenfalls deutlich. Dies gilt – von den Werten für *VVM2000* abgesehen – ebenfalls für die normierte *MoJo*-Qualität.

[10] DOVAL et al. (1999) schlagen für die Populationsstärke die zehnfache und für die Anzahl der Generationen die hundertfache Knotenzahl vor. Versuche mit den kleineren Fallstudien *Catalina* und *SSHTools* führten jedoch auch bei den dann erforderlichen sehr langen Laufzeiten zu kaum besseren Messwerten.

[11] BAUER und TRIFU (2004) nennen dieses Verfahren *Non-Adaptive Clustering*, weil dabei auf die Anpassung der Kantengewichte zwischen den Knoten im Abhängigkeitsgraphen gemäß der durch die Mustersuche identifizierten Rollen für die Architektur des Systems verzichtet wird.

Verfahren	**Maß**	*Catalina*	*SSH*	*jEdit*	*AWT*	*Swing*	*VVM2000*
NA	Prec	19	10	44	28	23	6
	Rec	47	32	24	25	23	78
	MoJo	118	254	328	245	947	1300
	MoJo_Q	49	50	59	49	39	77
Bunch	Prec	14	7	11	21	16	6
	Rec	8	1	3	3	6	4
	MoJo	177	428	690	373	1243	4670
	MoJo_Q	24	16	14	23	20	16
MMST	Prec	19	52	38	70	56	12
	Rec	30	18	16	5	6	4
	MoJo	177	178	236	159	451	3218
	MoJo_Q	24	65	71	67	71	42
HGGA	Prec	48	53	85	75	70	57
	Rec	9	18	8	15	7	4
	MoJo	107	179	220	153	431	2555
	MoJo_Q	54	65	73	68	72	56

Tabelle 8.2: Messwerte zu den Ergebnissen der verschiedenen Zerlegungsverfahren

Die von *Bunch* berechneten Zerlegungen weisen durchweg nur wenige Übereinstimmungen mit den Paketstrukturen der Fallstudien auf. Sowohl die Zuverlässigkeits- und Vollständigkeitsmaße, als auch die normierte *MoJo*-Qualität bleiben weit hinter den Ergebnissen, wie sie mit den anderen Verfahren erzielt werden konnten, zurück.

Im Falle von *VVM2000* führt unser hybrides Verfahren hinsichtlich der normierten *MoJo*-Qualität zu deutlich schlechteren Ergebnissen als das Verfahren *NA*, welches auf die vorgeschaltete Mustersuche verzichtet. Dies liegt hauptsächlich daran, dass unser hybrides Verfahren unter anderem 802 Ballungen berechnet, die jeweils genau eine Klasse mit dem Namenssuffix *BeanInfo* enthalten. Wir untersuchen dieses Ergebnis in Abschnitt 8.4 im Rahmen einer Inspektion des Quelltextes von *VVM2000* noch genauer. Wäre unser Verfahren in der Lage, diese 802 Klassen korrekt anderen Ballungen zuzuordnen, hätten wir für *VVM2000* ebenfalls eine akzeptable *Mojo*-Qualität > 70 erhalten.

Offensichtlich weist unser hybrides Verfahren schlechtere Werte hinsichtlich der Vollständigkeit *(Rec)* der berechneten Zerlegungen auf, als dies beispielsweise bei der Anwendung des *MMST*-Algorithmus auf einen Abhängigkeitsgraphen der Fall ist, bei dessen Konstruktion auf die vorangehende Mustersuche verzichtet wurde (Verfahren *NA*).

Wir führen dies auf die unterschiedliche Granularität der Ballungen zurück. Tabelle 8.3 zeigt, dass die Paketstruktur eine Zerlegung des Systems in vergleichsweise wenige Pakete darstellt. Die einzelnen Pakete sind in der Regel recht umfangreich und enthalten zahlreiche Elemente. Unser *HGGA* führt dagegen zu zahlreichen recht kleinen Ballungen. Abbildung 8.9 untermauert diese Beobachtung durch ein Histogramm der Ballungs- bzw. Paketgrößen für die Fallstudie *SSHTools*. Das Verfahren *NA* führt hingegen zu grö-

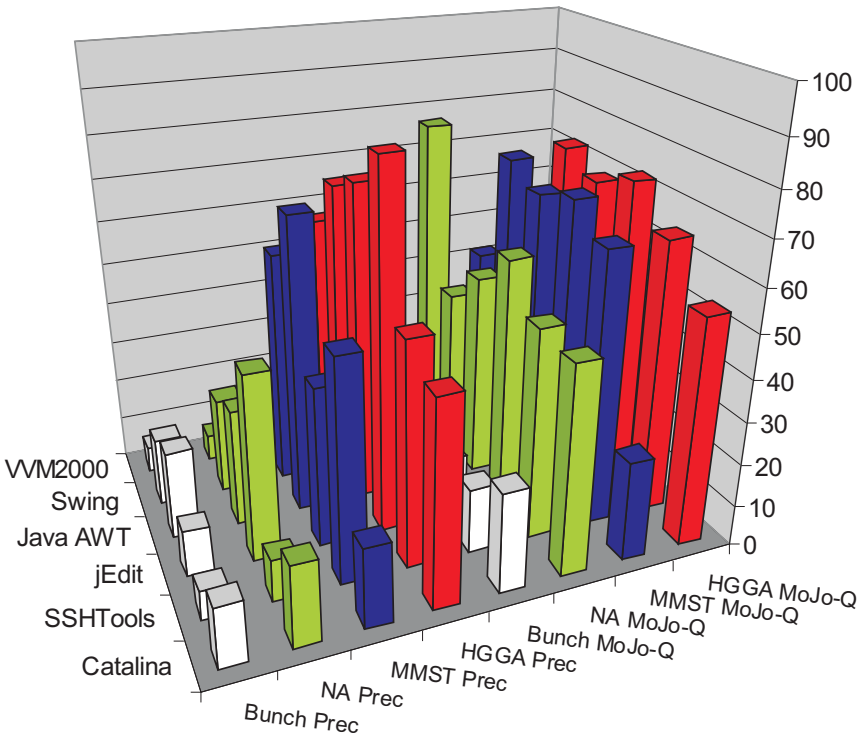


Abbildung 8.8: *MoJo*- und *Prec*-Qualitätsmaße für die verschiedenen Verfahren

ßeren Ballungen. Dadurch enthalten die Ballungen mehr Klassenpaare, die demselben Paket in der Paketstruktur entstammen – dies führt zu höheren Werten für die Vollständigkeit. Allerdings werden dadurch in der Regel auch zahlreiche Klassen zusammengruppiert, die in der Paketstruktur nicht in einem gemeinsamen Paket liegen. Dies ist an den niedrigen Werten für die Zuverlässigkeit deutlich zu ersehen.

Die hohen Zuverlässigkeitswerte der *HGGA*-Zerlegung zeigen, dass die kleinen Ballungen des *HGGA* in der Regel nur Elemente zusammen gruppieren, die auch in gemeinsamen Paketen liegen. Aufgrund der deutlich geringeren Größe der Ballungen des *HGGA* enthalten die einzelnen Pakete der Paketstruktur stets mehrere solche Ballungen. Dadurch kommen schlechtere Werte hinsichtlich des Vollständigkeitsmaßes zustande. Für die Praxis haben höhere Zuverlässigkeitswerte eine weitaus höhere Bedeutung als hohe Vollständigkeitswerte. Hohe Zuverlässigkeitswerte garantieren nämlich, dass nur Bauteile zusammengruppiert werden, die tatsächlich auch aus der Sicht eines Softwareingenieurs zusammen gehören. Zumindest ist dies dann der Fall, wenn die Ähnlichkeitsmaße bzw. die damit verbundene Gewichtung der Abhängigkeiten zwischen den Bauteilen die Denkweise von Softwareingenieuren widerspiegelt. Der nächste Abschnitt belegt dies mit einer Reihe konkreter Beispiele aus den Fallstudien.

	Catalina	*SSH*	*jEdit*	*AWT*	*Swing*	*VVM2000*
Knoten	232	507	805	482	2168	5549
Kanten	1501	5999	10599	5038	38222	78380
Dichte des Graphen	3,2	5,9	6,6	5,1	8,7	6,9
Anzahl Pakete	16	40	24	14	31	53
Paketgröße ϕ	15	13	34	34	50	104
Ballungen (HGGA)	86	76	143	112	483	1221[13]
Ballungsgröße ϕ	3	7	6	4	5	5

Tabelle 8.3: Daten zu den Abhängigkeitsgraphen der Fallstudien

Durch die Technik der hierarchischen Ballungsanalyse (siehe Abschnitt 7.5) lassen sich die kleinen Ballungen des *HGGA* zu größeren Ballungen zusammenführen, so dass diese eine Granularität erreichen, die der Paketstruktur der Fallstudien entspricht.

Anhand von Tabelle 8.3 können wir noch weitere interessante Beobachtungen machen:

- Es fällt auf, dass die "Dichte" der Abhängigkeitsgraphen (in Form der Kanten, die durchschnittlich auf einen Knoten kommen) aller Fallstudien – von *Catalina* abgesehen – auf relativ ähnlichem Niveau liegt. Dies ist von Interesse, da wir diese Dichte als ein Maß für die durchschnittliche Komplexität der Fallstudien verwenden können.[12] Obwohl die Fallstudien aus unterschiedlichen Domänen entstammen, sind sie hinsichtlich ihrer Durchschnittskomplexität demnach durchaus vergleichbar.

- Die durchschnittliche Größe der Ballungen, wie sie unser hybrides Verfahren berechnet, liegt zwischen vier und sieben Klassen. Obwohl die Zielfunktion des *HGGA* keine Komponente enthält, die Ballungen einer bestimmten Größe bevorzugt, liegen die Durchschnittsgrößen für alle Fallstudien recht dicht beieinander. Interessanterweise liegt die Durchschnittsgröße der berechneten Ballungen knapp unter der Anzahl von sieben Elementen, so dass die meisten Softwareentwickler diese noch für gut überschaubar halten (siehe auch Abschnitt 2.1.3).

Laufzeitmessungen. In Tabelle 8.4 haben wir zusammengestellt, wieviel Zeit unser hybrides Verfahren zur Zerlegung unserer Fallstudien im Rahmen unserer Qualitätsmessungen benötigt hat. Zudem geben wir die entsprechenden Vergleichswerte für *Bunch* an.

[12] BAUER und CIUPKE (2001) zeigen, wie dieses Maß zur Begutachtung eines Softwaresystems verwendet werden kann. Besonders "dichte" Stellen im Abhängigkeitsgraphen, also Knoten mit zahlreichen Abhängigkeiten sind dabei besonders kritisch, weil es sich dabei um Bauteile handelt, die eine wichtige Rolle im System spielen und häufig aufgrund ihrer hohen Kopplung auch nur sehr schwer zu verstehen sind.

[13] Hinzukommen 802 Ballungen, die jeweils genau eine Klasse mit dem Suffix *BeanInfo* enthalten. Wir gehen auf diese einelementigen Ballungen in Abschnitt 8.4 genauer ein.

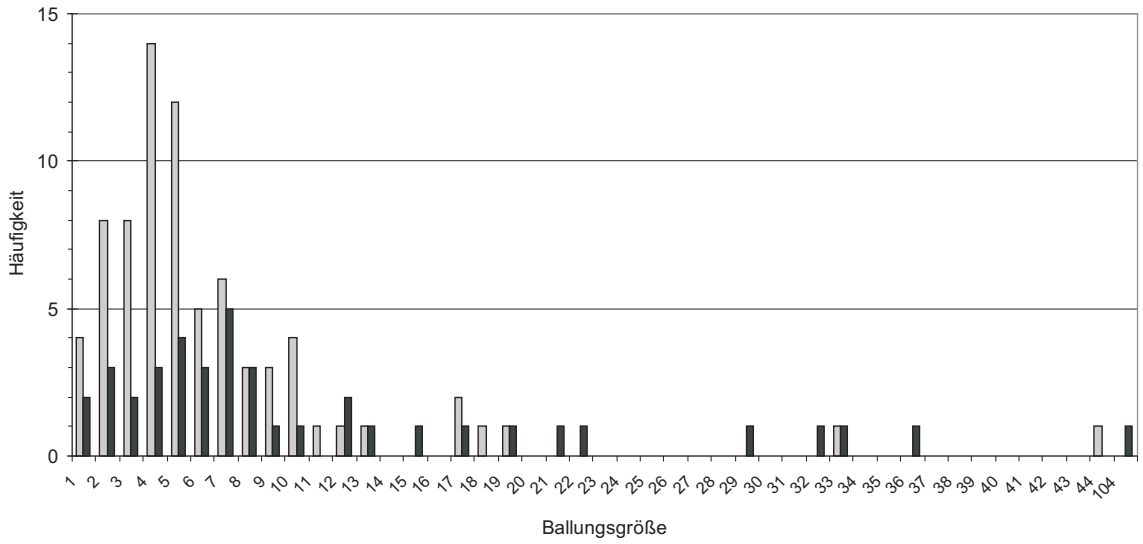


Abbildung 8.9: Histogramm zu den Größen der einzelnen Ballungen bei *SSHTools*

Es zeigt sich, dass der größte Zeitbedarf auf die Strukturextraktion, d.h. auf die Analyse der Quelltexte mit Hilfe von Techniken aus dem Übersetzerbau, und auf die Ballungsanalyse entfallen, zumindest, wenn für die Ballungsanalyse genetische Algorithmen eingesetzt werden.

Die gemessenen Zeiten zeigen, dass sich unser Verfahren auch zur Zerlegung großer Softwaresysteme praktikabel einsetzen lässt. Die Laufzeiten bleiben selbst für ein System mit knapp über einer Million Zeilen Code noch im Bereich weniger Stunden. Obwohl unser hybrides Verfahren in Verbindung mit dem genetischen Algorithmus *HGGA* deutlich bessere Ergebnisse erzielt, als der genetische Algorithmus aus *Bunch*, benötigt es im direkten Vergleich deutlich weniger Zeit für die eigentliche Ballungsanalyse.

8.4 Detaillierte Untersuchung der Ergebnisse

Im vorangehenden Abschnitt haben wir durch den Vergleich mit Referenzzerlegungen gesehen, dass unser hybrides Verfahren Systemzerlegungen berechnen kann, die – abgesehen von der kleineren Granularität der Teilsysteme – sehr viele Gemeinsamkeiten zu Zerlegungen aufweisen, die Softwareingenieure vornehmen, wenn sie Softwaresysteme konstruieren.

Wir untermauern in diesem Abschnitt die Leistungsfähigkeit unseres Verfahrens, indem wir die berechneten Zerlegungen für einige unserer Fallstudien im Detail untersuchen. Wir zeigen auf, dass die berechneten Zerlegungen sinnvolle Teilsystemstrukturen dar-

	Catalina	*SSH*	*jEdit*	*AWT*	*Swing*	*VVM2000*
Knoten	232	507	805	482	2168	5549
Strukturextraktion	226 s	341 s	770 s	520 s	24 min	137 min
Mustersuche	2 s	6 s	16 s	9 s	1 min	17 min
Graphkonstruktion	1 s	3 s	10 s	3 s	1 min	11 min
MMST	< 1 s	< 1 s	< 1 s	< 1 s	1 s	4 s
Populationsstärke	100	100	100	100	100	100
Generationen	250	500	800	500	2200	5500
HGGA	11 s	41 s	116 s	112 s	20 min	219 min
Bunch	86 s	221 s	383 s	222 s	40 min	302 min

Tabelle 8.4: Laufzeitmessungen

stellen, die dem Softwareingenieur beim Umgang und der Weiterentwicklung von Systemen nützlich sein können. Zudem diskutieren wir die Abweichungen zwischen den Referenzzerlegungen in Form der Paketstruktur und den berechneten Strukturen.

8.4.1 Fallstudie *SSHTools*

Abbildung 8.10 zeigt einen graphischen Vergleich zwischen der berechneten Zerlegung und der Paketstruktur für die Fallstudie *SSHTools*. Der abgebildete Graph enthält einen logischen Knoten (Ziffer 1), der das gesamte System repräsentiert. Dieses besteht aus den berechneten Teilsystemen. Dies ist im Graphen durch je einen Knoten und die zugehörige *enthält*-Kante zwischen Gesamtsystemknoten und Teilsystemknoten dargestellt. Jedes berechnete Teilsystem entspricht einer Ballung und enthält Klassen, die durch entsprechende Knoten und *enthält*-Kanten dargestellt sind. Die Farbe der Klassenknoten ergibt sich aus der Paketstruktur, dabei wurden Knoten aus jeweils gleichen Paketen gleich eingefärbt.

Teilsystem 2 beispielsweise enthält 33 Klassen, davon entstammen 26 dem Paket `j2ssh.sftp`[14] Jede dieser Klassen implementiert einen Basisbefehl des Protokolls *SFTP* zur verschlüsselten Übertragung von Dateien. So finden sich in diesem Teilsystem beispielsweise die Klassen `SshFxpOpen`, `SshFxpRead`, `SshFxpWrite` und dergleichen. Unser hybrides Verfahren hat – bei Verwendung des *HGGA* für die Ballungsanalyse – *alle* 26 dieser Basisbefehle erfolgreich in ein gemeinsames Teilsystem gruppiert. Hinzu kommen sieben weitere Klassen, vier davon sind Primitive des zugrundeliegenden *SSH*-Basisprotokolls. Die übrigen drei implementieren einen Basisdatentyp (`UnsignedInteger32`) und Datentypen für Ausnahmen.[15] *SSH-Primitive*,

[14] Im Quelltext der *SSHTools* eigentlich `com.sshtools.j2ssh.sftp`. Wir verzichten der Kürze halber im folgenden jedoch jeweils auf Präfixe, die nicht dem unmittelbaren Verständnis der Fallstudie dienlich sind.

[15] Ausnahmen (engl. *exceptions*) werden in *Java* durch Klassen implementiert.

Basisdatentypen und Ausnahmen wurden in der Paketstruktur jeweils separaten Paketen zugeordnet. Folglich weicht unser Verfahren hinsichtlich dieser sieben Klassen von der Paketstruktur ab. Die Zuordnung der *SSH-Primitive* müssen wir als fehlerhaft betrachten, diese hätten Teilsystem 3 zugeordnet werden müssen. Der Basisdatentyp `UnsignedInteger32` hätte einem Teilsystem mit Hilfsklassen zugeordnet werden können. Aus Sicht des Softwareingenieurs gibt es mehrere Möglichkeiten, die Klassen zur Implementierung von Ausnahmen zuzuordnen. Neben der Möglichkeit, die Ausnahmen in einem eigenen Paket zu organisieren, wie es die Paketstruktur tut, erscheint auch die Zuordnung der Ausnahmen zu den Teilsystemen, in denen die Ausnahmesituationen entstehen können, sinnvoll. Letztere Vorgehensweise deckt sich mit unserer berechneten Zerlegung.

Teilsystem 3 enthält Klassen dreier Pakete. Dies sind zunächst die Primitive des *SSH*-Basisprotokolls aus dem Paket `j2ssh.agent` (beispielsweise `SshAgentSuccess`, `SshAgentFailure`, `SshAgentPing`, `SshAgentData`), über die die Kommunikation mit den darüberliegenden Protokollschichten abläuft. Hinzu kommen sämtliche Klassen für die Datenpakete zur Kommunikation innerhalb der *SSH*-Protokollschicht. Einige davon dienen dem Errichten einer Verbindung (Paket `j2ssj.connection`: `MsgChannelOpen`, `MsgChannelClose` und dergleichen), zur Authentifizierung (Paket `j2ssh.authentication`: `SshMshUserAuthSucess`, `SshMsgUserAuthFailure`) und zum Transport von Daten (Paket `j2ssh.transport`: `SshMsgServiceRequest`, `SshMsgServiceAccept`, `SshMessage` und weitere).

Unser Zerlegungsverfahren gruppiert – abgesehen von den vier Klassen, die fälschlicherweise in Teilsystem 2 platziert wurden – alle diese Klassen in ein Paket. Aus der Sicht eines Softwareingenieurs ist diese Zerlegung sinnvoll, handelt es sich doch um Bestandteile des *SSH*-Basisprotokolls. Allerdings wäre zudem eine Trennung zwischen Protokollprimitiven und Paketen sinnvoll, wie sie die Paketstruktur vorsieht.

Die Ziffern 4 und 5 markieren weitere typische Beispiele für Teilsysteme, die sinnvolle Gruppierungen darstellen: Teilsystem 4 enthält Implementierungen zur Ansteuerung verschiedener Bildschirmterminals (*VT100, ANSI* und weitere), Teilsystem 5 enthält Klassen zur Implementierung von Kommunikationskanälen auf Basis der *TCP/IP*-Protokolle.

Teilsystem 6 enthält Klassen zur Definition von Menübefehlen für einen *SSHClient* mit graphischer Benutzerschnittstelle (Paket `common.ui`: `NewAction`, `SaveAction`, `CloseAction`, `EditAction OptionsAction`, und dergleichen). Weitere Bestandteile für die Benutzerschnittstelle finden sich in weiteren 6 Teilsystemen. Die Paketstruktur platziert alle diese Klassen in ein gemeinsames Teilsystem `common.ui`.

Diese Beobachtungen zeigen, dass unser *HGGA* auf Basis des Abhängigkeitsgraphen, wie wir ihn mit den Techniken aus den Kapiteln 6 und 7 konstruieren können, Teilsystemkandidaten berechnet, die eine sinnvolle, hilfreiche Strukturierung der Fallstudie *SSHTools* darstellen. In mehreren Punkten unterscheidet sich die berechnete Zerlegung

aber von der Paketstruktur. In einigen Fällen muss die berechnete Struktur noch überarbeitet werden – beispielsweise bei der Vermischung der Implementierungen der Primitive der Protokolle *SFTP* und *SSH* in Ballung 2. In anderen Fällen ist die durch den *HGGA* berechnete Zerlegung mindestens ebenso sinnvoll, wie die durch die Paketstruktur vorgesehene Gliederung des Systems. Ein Beispiel hierfür stellt die Zuordnung der Ausnahmeklassen dar.

Teilsystem 3 zeigt darüber hinaus, dass die Gliederung eines Systems in Pakete nicht ausschließlich aufgrund statischer Abhängigkeiten in den Implementierungsstrukturen – bzw. der Optimierung der durch die Abhängigkeiten verursachten Kohäsion und Kopplung der Pakete – erfolgt. Im Falle von Teilsystem 3 spielte bei der Gestaltung der Paketstruktur die Trennung der *SSH*-Protokollprimitive in die Belange Verbindungsaufbau und -abbau, Transport und Authentifizierung die primäre Rolle. Offensichtlich spiegelt sich diese konzeptionelle Trennung von Belangen kaum in Gestalt der Abhängigkeiten in der Implementierungsstruktur wieder. Interessant ist in diesem Zusammenhang die Beobachtung, dass die Bewertungsfuntkion f (siehe Gleichung 7.29) für die Paketstruktur deutlich schlechtere Werte annimmt, als dies für die berechnete Zerlegung der Fall ist.

Die anderen Verfahren liefern deutlich weniger leicht verständliche Zerlegungen der Fallstudie *SSHTools*. Der Verzicht auf die Mustersuche und die darauf aufbauende Gewichtung der Abhängigkeiten (Verfahren *NA*) führt beispielsweise zu einer Vermischung der Klassen für die Primitive des *SSH-Protokolls* mit Elementen aus der Benutzeroberfläche, so dass sich ein Teilsystemkandidat mit knapp 180 Klassen ergibt. Die Verwendung des genetischen Algorithmus aus *Bunch* führt zu vergleichbaren Vermischungen, allerdings fällt hier die Granularität der Teilsystemkandidaten deutlich kleiner aus. Dieser Effekt tritt auf, obwohl *Bunch* genau wie unser *HGGA* einen durch die Mustersuche verbesserten Abhängigkeitsgraphen mit angepassten Kantengewichten als Eingabe erhält. Wir schließen daraus, dass der genetische Algorithmus zur Ballungsanalyse aus *Bunch* deutlich schlechtere Zerlegungen hervorbringt. Unser *MMST* bleibt hinsichtlich der Nutzbarkeit der Zerlegung der Fallstudie *SSHTools* nur geringfügig hinter dem *HGGA* zurück. Er berechnet beispielsweise ebenfalls die Ballungen 2 und 3 für die Primitive der Protokolle *SFTP* und *SSH*, allerdings schlägt er drei Primitive des *SFTP*-Protokolls einer anderen Ballung zu.

8.4.2 Fallstudie *Java AWT*

Für die Fallstudie *Java AWT* haben wir eine ähnliche Inspektion der durch unser hybrides Verfahren aus Mustersuche und *HGGA* berechneten Zerlegung durchgeführt, wie wir dies im vorangehenden Abschnitt bereits für die Fallstudie *SSHTools* beschrieben haben. Wir fassen die wichtigsten Ergebnisse zusammen:[16]

[16] Eine ausführlichere Diskussion der Ergebnisse findet sich in TRIFU (2003).

Die berechneten Teilsystemkandidaten stellen in der Mehrzahl der Fälle kleinere Fragmente der recht großen Pakete der originalen Paketstruktur dar. Beispielsweise wurden das Paket `java.awt`, welches die meisten der Komponenten einer *AWT*-Benutzeroberfläche enthält, in mehrere Teilsysteme zerlegt. So entstanden beispielsweise jeweils Teilsysteme für die Klassen zur Erzeugung von Menüs (`Menu`, `MenuItem`, `MenuContainer`, `MenuShortCut`) und zur Texteingabe (`TextArea`, `TextField`, `TextComponent`).

Neben Klassen für solche Oberflächenelemente enthält *Java AWT* weitere Bestandteile. Zu den meisten Oberflächenelementen gibt es sogenannte *Peer*-Klassen, die eine betriebssystemabhängige Implementierung für die entsprechenden Oberflächenelemente bereitstellen. Für diese extern in *C* implementierten *Peer*-Klassen gibt es entsprechende in *Java* implementierte Schnittstellen im Paket `java.awt.peer`, so dass eine Schichtung (siehe auch Abschnitt 2.2) entsteht: Die unterste Schicht 0 besteht aus den in *C* implementierten Routinen. Die Klassen des Pakets `java.awt.peer` definieren eine Zugriffsschicht auf diese Routinen für *Java* (Schicht 1), die von den eigentlichen Oberflächenelementen im Paket `java.awt` (Schicht 2) benutzt werden (MIDDENDORF et al., 2002).

Unser hybrides Verfahren separiert die Schichten 1 und 2 erfolgreich: 25 der 27 *Peer*-Schnittstellen des `awt.peer`-Pakets wurden erfolgreich zusammen gruppiert. Bei Verzicht auf die vorausgehende Mustersuche konnten lediglich 15 der *Peer*-Schnittstellen erfolgreich von den Oberflächenelementen aus Schicht 2 separiert werden.

Ein ähnlicher Effekt tritt bei den Klassen des Paketes `java.awt.event` auf. Mit Hilfe von *Ereignissen*[17] wird in *Java AWT* die Kommunikation zwischen Oberflächenelementen und Anwendungslogik realisiert. Benutzeraktivitäten (wie Tastatureingaben oder Mausklicks auf bestimmte Oberflächenelemente) führen zur Erzeugung von Ereignissen unterschiedlichen Typs, die dann von der Anwendungslogik mit Hilfe sogenannter *Listener* aufgenommen und interpretiert werden können. Unterschiedliche, vordefinierte Ereignistypen werden im Paket `java.awt.event` in Form entsprechender Klassen bereitgestellt.

Unser hybrides Verfahren separiert diese *Event*-Klassen erfolgreich von den entsprechenden Oberflächenelementen: 42 der 44 Klassen aus dem Paket `java.awt.event` werden auf 14 Teilsysteme verteilt, die jeweils lediglich zusammengehörende Ereignistypen (Mausereignisse, Ereignisse auf Fenstern, Menüs und dergleichen) und entsprechende vordefinierte *Listener* enthalten. Bei Verzicht auf die Mustersuche (Verfahren *NA*) werden die 44 Klassen aus `java.awt.event` beliebig auf andere Teilsysteme verteilt.

Auch am Beispiel dieser Fallstudie wird deutlich, dass unser hybrides Verfahren aus Mustersuche und anschließender Ballungsanalyse mit dem *HGGA* Systemzerlegungen

[17] engl. *event*

berechnet, die den Softwareingenieuren beim Verstehen eines Softwaresystems von Nutzen sein können.

8.4.3 Fallstudie *VVM2000*

Die Fallstudie *VVM2000* stellt ein typisches Informationssystem dar, dessen Aufbau stark durch sogenannte *Geschäftsobjekte* geprägt ist. Diese Geschäftsobjekte werden häufig in Form von datenorientierten Modellen (beispielsweise in UML-Klassendiagrammen) spezifiziert, aus denen dann werkzeuggestützt Datenbank-Schemata und *Java*-Quelltextrahmen generiert werden können (*Modellgetriebene Entwicklung*, siehe KLEPPE et al. (2003) und FRANKEL (2003)). In die Quelltextrahmen wird dann nachträglich die Anwendungslogik in Form von handprogrammierten Methodenimplementierungen eingefügt. In *VVM2000* werden dabei für jedes Geschäftsobjekt eine Anzahl von Klassen generiert, deren Implementierung sich auf *Enterprise JavaBeans* (MONSON-HAEFEL, 2002) abstützt. Tabelle 8.5 zeigt solche Klassen für das fiktive[18] Geschäftsobjekt LVVertrag, welches einen Vertrag für eine Lebensversicherung modelliert.

Neben den *Java*-Implementierungen solcher Geschäftsobjekte enthält *VVM2000* noch weiteren Code. Eine Reihe von Klassen implementiert eine graphische Benutzerschnittstelle auf Basis von *Swing*. Andere Klassen stellen eine zusätzliche technische Infrastruktur bereit, mit Hilfe derer beispielsweise der Zugriff auf Datenbestände auf Großrechnern erfolgen kann.

Setzen wir unser hybrides Verfahren auf *VVM2000* an, so erhalten wir 2023 Ballungen. Eine genauere Untersuchung dieser Ballungen fördert folgendes Ergebnis zu Tage:

- 802 Ballungen enthalten je eine vollständige Implementierung eines Geschäftsobjektes. Unser Verfahren ist also in der Lage, *alle* zugehörigen Elemente eines Geschäftsobjektes gemäß Tabelle 8.5 zu identifizieren und gemeinsam zu einem Teilsystemkandidaten zu gruppieren. Dabei werden auch verschiedene Ausprägungen der Geschäftsobjekte korrekt behandelt. So stellt ein Teil der Geschäftsobjekte Entitäten dar, wie beispielsweise unser Geschäftsobjekt LVVertrag, andere Geschäftsobjekte beschreiben Aktivitäten, beispielsweise MedizinischeRisikoPruefung. Die Implementierung von Aktivitäten verzichtet auf die Primärschlüssel (Suffix *PK*), enthält allerdings häufig eine Klasse mit dem Suffix *Adapt*, die synchrone Methodenaufrufe in asychrone umsetzt.

- In allen 802 Ballungen für die Geschäftsobjekte fehlt allerdings die *BeanInfo*-Klasse. Diese wurde stets in eine eigene, einelementige Ballung platziert. Die

[18] *VVM2000* ist ein kommerzielles System, dessen Implementierung vertraulich bleiben muss. Namen, Bezeichner und dergleichen wurden daher modifiziert. Die Beispiele stimmen allerdings hinsichtlich ihres Aufbaus und ihrer Strukur mit dem Code aus *VVM2000* überein.

Klasse	Beschreibung
`LVVertrag`	*(Remote) Interface*. Definiert die Schnittstelle des Geschäftsobjektes (Methoden inkl. Zugriffsmethoden für Datenfelder (Attribute)).
`LVVertragBean`	*Bean*. Serverseitige Komponente. Implementiert die Funktionalität des Geschäftsobjektes in Form der in `LVVertrag` definierten Methoden.
`LVVertragBeanBase`	*Coderahmen für Bean*. Enthält den automatisch generierten Coderahmen für `LVVertragBean`. `LVVertragBean` enthält lediglich die von Hand implementierten Methoden und ist von `LVVetragBeanBase` abgeleitet. Auf diese Weise lassen sich Modelländerungen leichter beherrschen, weil eine klare Trennung zwischen generiertem Code und der Handimplementierung besteht.
`LVVertragBeanInfo`	*BeanInfo*-Klasse. Enthält Hilfsfunktionen zum Betrieb der Bean im *EJB-Container*
`LVVertragHome`	*Home Interface*. Definiert eine Schnittstelle zum Erzeugen und Auffinden von Instanzen der Bean.
`LVVertragHomeImpl`	Implementierung des *Home Interface*
`LVVertragPK`	*Primärschlüssel*. Implementiert den Primärschlüssel zum Auffinden von Bean-Instanzen
`LVVertragEdit`	*(Remote) Interface* zur kontrollierten Modifikation von LVertrag-Instanzen.
`LVVertragEditBean`	*Session Bean*. Implementiert `LVVertragEdit`.
`LVVertragEditBeanBase`	*Coderahmen* für `LVVetragEditBean` (s.o.)
`LVVertragEditHome`	*Home Interface* für *Session Bean*

Tabelle 8.5: Bauteile eines Geschäftsobjektes

BeanInfo-Klassen dienen dazu, dass der *EJB-Container* mit Hilfe der Introspektion[19] die Inbetriebnahme der Beans abwickeln kann.[20] Alle *BeanInfo*-Klassen enthalten keine statischen Abhängigkeiten zu ihren zugehörigen Beans, von dem Rückgabeparameter in einer einzigen Methode `getBeanClass()` abgesehen. Aus diesem Grund besteht nur eine sehr geringe Kopplung zwischen *BeanInfo*-Klassen und Beans. Da die *BeanInfo*-Klassen auch sonst nur statische Abhängigkeiten zu den Paketen `java.lang` und `java.bean` enthalten, führt dies dazu, dass diese in der Ballungsanalyse isoliert werden.

- Die verbleibenden Ballungen enthalten Infrastruktur-Code. Dabei fällt auf, dass Code für Benutzeroberfläche und Großrechnerzugriff jeweils durchgängig vom übrigen Infrastruktur-Code getrennt wurde.

Die Zerlegung von *VVM2000* in Geschäftsobjekte und Infrastrukturkomponenten gelang weder bei Verwendung des *MMST* noch bei Verwendung des genetischen Algorithmus aus *Bunch*. Auch bei Einsatz des Verfahrens *NA*, bei Verzicht auf Mustersuche und die entsprechende Anpassung der Kantengewichte des Abhängigkeitsgraphen, konnten weder Geschäftsobjekte noch Infrastrukturkomponenten korrekt identifiziert werden. Beim Verfahren *NA* wurde eine große Anzahl der *BeanInfo*-Klassen anderen Ballungen zugeschlagen.

Die Technik der hierarchischen Ballungsanalyse (siehe Abschnitt 7.5) führte bei *VVM2000* ebenfalls zu interessanten Einsichten. Auf der dritten Hierarchieebene entstanden knapp 100 Ballungen. Dabei konnten zum Beispiel die Geschäftsobjekte aus den Paketen `RechtlichePruefung`, `WirtschaftlichePruefung` und `MedizinischePruefung` zusammengeführt werden. Auch Komponenten aus dem Paket `Sicherheit`, in denen offensichtlich eine Verwaltung der Zugriffsrechte implementiert ist, wurden in einer gemeinsamen Ballung zusammengefasst.

8.5 Zusammenfassung

In den vorangehenden Abschnitten haben wir demonstriert, dass unser Verfahren den Anforderungen entspricht, die wir in Abschnitt 2.5 formuliert haben.

Unser Verfahren ist *allgemein* anwendbar – wir können beliebige objektorientierte Softwaresysteme in sinnvolle Teilsysteme zerlegen. Unsere sehr unterschiedlichen Fallstudien belegen, dass der gewählte Ansatz tragfähig ist, unabhängig von Anwendungsdomäne und Implementierungstechnik der zu untersuchenden Softwaresysteme.

Unser Verfahren ist zudem prinzipiell unabhängig von einer bestimmten Programmiersprache. Unser Werkzeug *ACT* benötigt lediglich geeignete Strukturextraktoren für die

[19] In *Java*-Terminologie: *Reflection*

[20] In modernen *EJB-Containern* werden diese Informationen in externen Spezifikationen, den *Deployment*-Beschreibungen, vorgehalten.

jeweiligen Programmiersprachen. Entsprechende Strukturextraktoren für die Programmiersprachen *C++* und *Delphi* sind in Arbeit (GAISSER, 2004; HALLER et al., 2003) und können in Zukunft auch von *ACT* verwendet werden.

Die Fallstudie *VVM2000* zeigt, dass unser Verfahren auch auf große Systeme mit mehr als einer Million Zeilen Quelltext anwendbar ist. Durch die Technik der hierarchischen Ballungsanalyse können Teilsystemstrukturen auf unterschiedlichen Abstraktionsebenen berechnet werden, so dass unser Verfahren die Anforderung *Skalierbarkeit für große Systeme* erfüllt.

Das Werkzeug *ACT* demonstriert, dass sich unser Verfahren *automatisieren* lässt. In den voranstehenden Abschnitten haben wir gezeigt, dass wir Teilsystemstrukturen berechnen können, die erheblich zum Verständnis des Aufbaus großer Softwaresysteme beitragen können. Erfahrene Softwareingenieure können die berechneten Strukturen auch als Ausgangsbasis zu einer Überarbeitung existierender Strukturen eines Softwaresystems nutzen, um dessen Wartbarkeit zu verbessern oder um es für weiterführende Entwicklungsarbeiten vorzubereiten.

Wir haben insbesondere gesehen, dass die berechneten Zerlegungen den Anforderungen entsprechen, die erfahrene Softwareingenieure an Teilsystemstrukturen stellen: Unser Verfahren kann Softwaresysteme *vollständig* in Teilsysteme zerlegen. Die berechneten Teilsysteme sind *aussagekräftig*, denn sie enthalten – von einigen wenigen Ausnahmen abgesehen – ausschließlich solche Bauteile des Systems, die zusammengehörende Konzepte beschreiben.

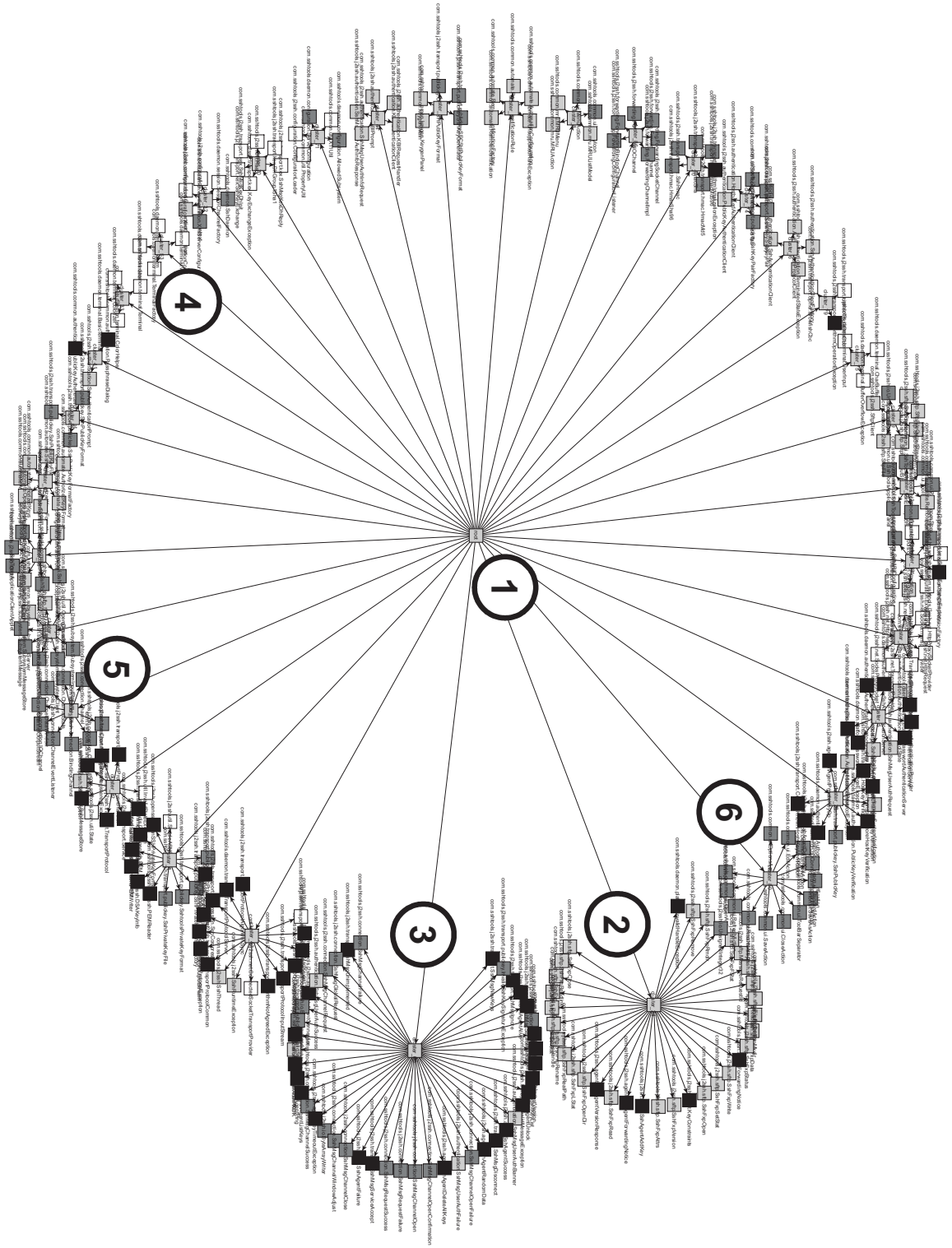


Abbildung 8.10: Zerlegung der Fallstudie *SSHTools*

Kapitel 9

Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Entwicklung eines praktikablen Verfahren zur Gewinnung von Teilsystemstrukturen aus der Implementierung objektorientierter Systeme. In diesem Kapitel stellen wir die wesentlichen Ergebnisse der Arbeit zusammen und diskutieren, wie wir dieses Ziel erreicht haben. Ferner geben wir einen Ausblick auf weiterführende Arbeiten.

9.1 Zusammenfassung

Wir haben in dieser Arbeit ein *hybrides Verfahren* zur Extraktion von Teilsystemstrukturen vorgestellt. Es beruht auf einer Kombination von Mustersuche und Ballungsanalyse.

Ausgangspunkt des Verfahrens ist ein Modell der Implementierungsstruktur eines Softwaresystems. Dieses Modell enthält die Bauteile des Systems – beispielsweise Klassen und Methoden – und die Beziehungen zwischen diesen Bauteilen. Es kann durch gängige Übersetzerbautechnik aus den Quelltexten des Softwaresystems gewonnen werden.

Mit Hilfe von Techniken zur Mustersuche klassifiziert unser Verfahren die Bauteile des Systems hinsichtlich ihrer Bedeutung für die Architektur des Softwaresystems.

Anschließend werden die Bauteile des Systems mit Hilfe von Algorithmen zur Ballungsanalyse so gruppiert, dass zusammengehörende Bauteile ein gemeinsames Teilsystem bilden. Das Verfahren stellt zwei Algorithmen zur Ballungsanalyse zur Verfügung. Der sehr effiziente graphbasierte Algorithmus *MMST* liefert mit wenig Berechnungsaufwand ordentliche Zerlegungen des Systems. Mit Hilfe des genetischen Algorithmus *HGGA* können diese Ergebnisse noch deutlich verbessert werden.

Zur Entscheidungsfindung, welche Bauteile gemeinsam in eine Gruppe platziert werden sollen, werten die Ballungsanalysealgorithmen Ähnlichkeitsmaße aus. Diese Maße haben wir daher so definiert, dass die Gruppen bzw. Ballungen, die als Ergebnis der Ballungsanalyse entstehen, Teilsystemkandidaten bilden, die den Prinzipien eines guten Softwareentwurfs genügen. Neben den üblichen Anforderungen, die an Teilsysteme gestellt werden – wie hohe innere Kohäsion und geringe Kopplung zu anderen Systemteilen – berücksichtigt unser Verfahren bei der Teilsystembildung auch die Rolle, welche

die einzelnen Bauteile in der Architektur eines Systems einnehmen, indem die Ergebnisse der vorangehenden Mustersuche ausgewertet werden.

Unsere Vorgehensweise führt – wie die Bearbeitung mehrerer umfangreicher Fallstudien mit Hilfe eines eigens konstruierten Werkzeugs zur automatisierten Durchführung unseres Verfahrens zeigt – zu Teilsystemzerlegungen, die das Systemverständnis fördern und zudem eine Basis für die systematische Pflege und Weiterentwicklung von Softwaresystemen bilden können.

9.2 Ergebnisse

Die Arbeit stellt ein *neuartiges* Verfahren zur Extraktion von Teilsystemstrukturen für existierende, objektorientierte Softwaresysteme vor. Es kombiniert erstmals die Stärken von Mustersuchverfahren, mit deren Hilfe den Bauteilen in gewisser Weise ihre Bedeutung im Entwurf des Systems zugeordnet werden kann, und von Ballungsanalyseverfahren, die Systeme *vollständig* in Teilsysteme zerlegen können.

Das Verfahren erfüllt folgende Anforderungen:

- *Allgemeinheit:* Es funktioniert für beliebige objektorientierte Systeme. Insbesondere liefert es auch gute Ergebnisse für Systemstrukturen, bei denen existierende Verfahren bisher versagt haben, beispielsweise wenn die Systeme geschichtete Architekturen, Client-Server-Architekturen, Rahmenwerke oder Bibliotheken aufweisen.

- *Automatisierbarkeit und Skalierbarkeit:* Wie das im Rahmen der Arbeit implementierte Werkzeug *ACT* zeigt, ist das Verfahren durchgängig automatisierbar. Wir konnten ferner nachweisen, dass das Verfahren auch für große Softwaresysteme mit mehr als einer Million Zeilen Code praktikabel ist.

- *Hohe Qualität der extrahierten logischen Strukturen:* Wir konnten in Kapitel 8 vorführen, dass unser Verfahren Zerlegungen von Softwaresystemen in Teilsysteme berechnet, die dem Systemverständnis dienlich sind und die als Grundlage für die Pflege, Sanierung und Weiterentwicklung von Softwaresystemen dienen können. Die Qualität der durch unser hybrides Verfahren berechneten Zerlegungen übertrifft die Ergebnisse deutlich, die sich mit herkömmlichen Verfahren erzielen lassen. Die höhere Qualität des Verfahrens ergibt sich im wesentlichen aus der Kombination der Ballungsanalyseverfahren mit der Mustersuche. Die Mustersuche schafft dabei die Voraussetzungen, indem sie die Ballungsanalyse mit semantischen Informationen über die Bedeutung der Bauteile des Systems für seine Architektur versorgt, so dass diese zusammen mit neu entwickelten Ähnlichkeitsmaßen zu besseren Ergebnissen führt.

	Software-maße	Muster-suche	Ballungs-analyse	Reflexions-modelle	**Hybrides Verfahren**
Skalierbarkeit	○	+	+	○	+
Automatisier-barkeit	○	+	+	−	+
Allgemeinheit	○	+	+	+	+
Qualität der Strukturen					
Vollständigkeit	−	−	+	○	+
Aussagekraft	○	○	○	○/+	+

Tabelle 9.1: Vergleichende Bewertung unseres hybriden Verfahrens

Wie Tabelle 9.1 zeigt, stellt unser hybrides Verfahren zur Gewinnung von Teilsystemstrukturen mit der Erfüllung dieser Anforderungen einen signifikanten Fortschritt gegenüber dem bisherigen Stand der Technik dar.

Auf technischer Ebene konnten wir Fortschritte in folgenden Bereichen erzielen:

- *Klassifikation von Systemteilen durch statische Mustersuche:* Wir haben im Rahmen unserer Arbeit eine Technik zur Mustersuche entwickelt, die es ermöglicht, Bauteile gemäß ihrer Bedeutung im Entwurf eines Softwaresystems zu klassifizieren. Die Mustersuche geht hinsichtlich Anzahl und Art der unterstützten Muster über existierende Arbeiten in diesem Bereich hinaus. Schon die Erkenntnisse, die sich mit Hilfe der Mustersuche gewinnen lassen, können erheblich zum Verständnis von Softwaresystemen beitragen.

- *Algorithmen zur Ballungsanalyse:* Unsere Arbeit stellt zwei Algorithmen zur Ballungsanalyse vor. Der graphbasierte *MMST* stellt eine Modifikation eines bereits existierenden Verfahrens dar, die sich für Zerlegung von Softwaresystemen besonders eignet. Der neu entwickelte genetische Algorithmus *HGGA* verwendet – im Anwendungsgebiet der Teilsystemzerlegung von Softwaresystemen erstmalig – problemspezifisch gestaltete Operationen zur Erzeugung und Fortentwicklung von Lösungspopulationen. Er hebt sich dadurch deutlich von vergleichbaren Algorithmen ab. Wie in Kapitel 8 gezeigt wurde, führt die problemspezifische Ausgestaltung der Operationen zu deutlich besseren Ergebnissen bei der Zerlegung von Softwaresystemen. Zusätzlich kann durch ihren Einsatz der Rechenaufwand signifikant verringert werden.

- *Definition von Ähnlichkeitsmaßen:* Unsere Arbeit verwertet erstmals erfolgreich neben direkten Abhängigkeiten auch *indirekte Abhängigkeiten* zwischen den Bauteilen des Systems, wie sie durch die gemeinsame Verwendung der Bauteile entstehen. Diese indirekten Abhängigkeiten stellen wertvolle Informationen dar, die

bei der Gliederung von Systemen berücksichtigt werden sollten, beispielsweise um Softwareingenieuren die Verwendung von Bibliotheken zu erleichtern. Unser Ähnlichkeitsmaß für die indirekten Abhängigkeiten ist daher auch über unser Verfahren hinaus in der Qualitätssicherung von Bibliotheken von Interesse.

Durch die erfolgreiche Kombination dieser Techniken versetzt unser Verfahren Softwareingenieure erstmals in die Lage, aussagekräftige logische Strukturen in Form von Teilsystemen automatisiert auf Basis der Implementierung von Softwaresystemen zu berechnen. Da diese Teilsystemstrukturen als solide Grundlage für das Verständnis, die Pflege, die Sanierung und die anschließende Weiterentwicklung von Softwaresystemen herangezogen werden können, liefert unsere Arbeit wesentliche Beiträge zur kostengünstigen und qualitätsbewussten Evolution von Softwaresystemen.

9.3 Ausblick

Unser hybrides Verfahren stellt eine gut funktionierende, grundsätzliche Vorgehensweise zur Berechnung von Teilsystemstrukturen dar. Es sieht an verschiedenen Stellen Erweiterungsmöglichkeiten vor, auf die wir im folgenden kurz eingehen wollen.

Die Mustersuche liefert bereits in der vorliegenden Form wertvolle Erkenntnisse über die Bedeutung der einzelnen Bauteile in der Architektur eines Softwaresystems. Es liegt nahe, die Mustersuche durch Erkennungsmöglichkeiten für zusätzliche Muster zu ergänzen. Für Praktiker sind hier Muster von besonderem Interesse, die sich aus der Verwendung technischer Infrastrukturen ergeben. So könnten Schnittstellen- und Komponentenbeschreibungen, wie sie gängige Komponenteninfrastruktursysteme verwenden, verwertet werden. Diese Beschreibungen liefern wertvolle Hinweise auf Teilsystemstrukturen. Beispiele hierfür sind *IDL*-Spezifikationen in *COM*- bzw. *CORBA*-Systemen oder *Deployment-Deskriptoren* in Systemen, die sich auf *Enterprise JavaBeans* abstützen.

Weitere Erweiterungsmöglichkeiten bestehen in der Verwendung zusätzlicher Ähnlichkeitsmaße. Unser Verfahren zeigt, dass indirekte Abhängigkeiten wertvolle neue Erkenntnisse über die Zusammenhänge zwischen den Bauteilen des Systems liefern. Unser Maß für die indirekten Abhängigkeiten zwischen Bauteilen, die stets gemeinsam verwendet werden, lässt sich durch ein Maß zur Untersuchung indirekter Abhängigkeiten ergänzen, die entstehen, wenn Bauteile häufig die selben Systemteile verwenden. Auf diese Weise könnten beispielsweise Bestandteile einer graphischen Benutzerschnittstelle oder einer Datenbankzugriffsschicht noch zuverlässiger identifiziert werden, da diese stets dieselben Funktionen einer GUI-Bibliothek oder eines Datenbanktreibers aufrufen.

Weitere Ähnlichkeitsmaße können gewonnen werden, indem wir die Historie von Softwaresystemen in Versionsverwaltungssystemen auswerten. Da sich Änderungen an einem Softwaresystem, die innerhalb eines Versionsschrittes vorgenommen wurden, mit großer Wahrscheinlichkeit auf bestimmte Belange der Funktionalität beziehen, können

wir davon ausgehen, dass zwischen den modifizierten Bauteilen ein logischer Zusammenhang besteht. Diesen können wir mit Hilfe eines Ähnlichkeitsmaßes erfassen und im Rahmen der Ballungsanalyse verwerten.

Von Interesse ist in diesem Zusammenhang die Frage, wie sich die zahlreichen Abhängigkeiten unterschiedlichen Typs auf die Zerlegung von Systemen auswirken. In wie weit decken sich die Zerlegungen, die unter ausschließlicher Verwendung von direkten Abhängigkeiten gewonnen werden können, mit den Zerlegungen, die die indirekten Abhängigkeiten widerspiegeln? Sind die Zerlegungen auf Basis direkter Abhängigkeiten deckungsgleich mit den Zerlegungen, die sich aus der skizzierten Betrachtung der Entwicklungshistorie ergeben? Die Beantwortung dieser Fragen dürfte spannende Erkenntnisse über die Konstruktion von Teilsystemhierarchien zu Tage fördern: Wie gut gelingt es Softwareingenieuren beispielsweise, Änderungen lokal (d.h. innerhalb) von Teilsystemen zu halten? In wie weit deckt sich die Gliederung von Softwaresystemen nach intuitiven Gesichtspunkten (“Platziere Elemente in ein gemeinsames Teilsystem, wenn sie entweder dieselben Bauteile verwenden oder stets gemeinsam verwendet werden sollen”) mit einer Systemgliederung nach dem Prinzip der Minimierung von Abhängigkeiten zwischen Teilsystemen?

Im Rahmen dieser Arbeit haben wir unser hybrides Verfahren vorwiegend daraufhin ausgelegt, Softwareingenieren das Verständnis der logischen Struktur zu erleichtern. Natürlich können die durch unser Verfahren berechneten Teilsystemstrukturen auch dazu herangezogen werden, die Qualität existierender Zerlegungen im Rahmen der *Qualitätssicherung* zu beurteilen. Hierzu kann der Vergleich der Zerlegungen wertvolle Hinweise liefern. Unser Verfahren berechnet Zerlegungen, die nach bestimmten objektiven Kriterien – beispielsweise der Vermeidung von Abhängigkeiten zwischen Teilsystemen – als gut gelten können. Unterschiede zwischen der existierenden Teilsystemstruktur und der berechneten Teilsystemstruktur können auf Bauteile hinweisen, die solchen Konstruktionskriterien widersprechen. Eine solche Vorgehensweise erfordert allerdings die genaue Betrachtung und Bewertung der Unterschiede zwischen den Zerlegungen. Eine entsprechende Unterstützung dieser Betrachtung durch Visualisierungswerkzeuge ist daher wünschenswert.

Unser hybrides Verfahren kann auch in ein Verfahren zur werkzeuggestützten Überarbeitung von unzureichend strukturierten Softwaresystemen ausgebaut werden. Eine mögliche Strategie hierfür könnte folgendermaßen aussehen: Wir berechnen zunächst mit Hilfe unseres Verfahrens verbesserte Teilsystemstrukturen und leiten dann Adaptionsprogramme (GENSSLER, 2004) zur Transformation der existierenden Teilsystemstrukturen in die verbesserten Teilsystemstrukturen ab. Nach Bereinigung der Teilsystemstrukturen wenden wir uns feingranulareren Strukturen zu. Zur Reorganisation von Vererbungsgraphen, Methoden- und Attributzuordnungen in Klassen könnte – wie bereits zur Optimierung der Teilsystemstrukturen – ein problemspezifisch angepasster genetischer Algorithmus zum Einsatz kommen, der die Qualität von Implementierungsstrukturen optimiert. Zur Bewertung der Implementierungsstrukturen im Rahmen der Zielfunktion für die Op-

timierung können die Arbeiten von CIUPKE (2001) und MARINESCU (2002) wertvolle Hinweise geben. Erste Ergebnisse der Verfolgung dieses Ziels (SENG und PACHE, 2004; PACHE, 2005) belegen die guten Erfolgsaussichten einer solchen Vorgehensweise.

Anhang A

Modellierung von Softwaresystemen mit Prädikatenlogik

Wir geben hier eine Übersicht über die in Kapitel 6 eingeführte Modellierung von Softwaresystemen durch Formeln der Prädikatenlogik erster Ordnung.

A.1 Entitäten, Beziehungen und Attribute

Die folgende Tabelle enthält alle Funktionen und Prädikate, die benötigt werden, um die Implementierungsstruktur eines Softwaresystems aus 5 gemäß der Konstruktion in Abschnitt 6.1 darzustellen.

Prädikat, Funktion	**Bedeutung**
Entitäten des Metamodells	
$Namensraum(n)$	n ist ein Namensraum
$Klasse(c)$	c ist eine Klasse
$Methode(m)$	m ist eine Methode
$Variable(x)$	x ist eine Variable
$Attribut(x)$	x ist ein Attribut (einer Klasse)
$LokaleVariable(x)$	x ist eine lokale Variable
$Parameter(x)$	x ist ein Parameter (einer Methode)
$Rumpf(r)$	r ist Rumpf (einer Methode)
Beziehungen	
$enthaeltKlasse(n,c)$	Die Klasse c gehört zum Namensraum n
$erbtVon(c_1,c_2)$	Die Klasse c_1 wurde von der Klasse c_2 abgeleitet
$hatAttribut(c,x)$	Die Klasse c hat das Attribut x
$hatMethode(c,m)$	Die Methode m gehört zur Klasse c
$hatParameter(c,x)$	Die Methode m hat einen Parameter x
$hatParameterTypen(m,P)$	Die Typen im Tupel P entsprechen den Typen der Parameter der Methode m
$hatRueckgabeTyp(m,c)$	Die Methode m gibt einen Typ c zurück

...

Prädikat, Funktion	Bedeutung
$hatRumpf(m,r)$	Zur Methode m gehört Rumpf r
$hatVariable(r,x)$	Die Methode m enthält eine Deklaration der Variablen x
$hatTyp(x,c)$	Die Variable x hat den (statischen) Typ c
$ruft(r,m)$	Der (Methoden-)Rumpf r enthält mindestens einen Aufruf an die Methode m
$i = \#Aufrufe(r,m)$	Der (Methoden-)Rumpf r enthält i Aufrufe an die Methode m
$greiftZu(r,x)$	Der (Methoden-)Rumpf r enthält mindestens einen Zugriff auf die Variable x
$i = \#Zugriffe(r,x)$	Der (Methoden-)Rumpf r enthält i Zugriffe auf die Variable x
Attribute für Elemente	
$s = name(x)$	Die Entität (Klasse, Methode oder Attribut) x hat den Namen n
$istOeffentlich(x)$	Die Entität (Klasse, Methode oder Attribut) x hat die Sichtbarkeit *öffentlich*
$istAbstrakt(x)$	Die Entität (Klasse, Methode oder Attribut) x ist abstrakt
$istSchnittstelle(c)$	Die Klasse c ist eine Schnittstelle
$istKonstruktur(m)$	Die Methode m ist ein Konstruktor
$istKlassenmethode(m)$	Die Methode m ist eine Klassenmethode
$istKlassenvariable(x)$	Das Attribut x ist eine Klassenvariable
$i = \#LOC(r)$	Der Rumpf r enthält i Zeilen Code
$i = \#Aufrufe(r)$	Der Rumpf r enthält insgesamt i Methodenaufrufe
$i = \#Anweisungen(r)$	Der Rumpf r enthält i Anweisungen
$i = \#Schleifen(r)$	Der Rumpf r enthält i Schleifen
$i = \#Verzweigungen(r)$	Der Rumpf r enthält i Verzweigungen

A.2 Prädikate zur Klassifikation der Bauteile eines Systems

Die folgende Tabelle enthält alle Prädikate, die wir im Rahmen der Mustersuche aus Kapitel 6.1 definiert haben, um die Bauteile des Systems gemäß ihrer Bedeutung für die Architektur zu klassifizieren.

Prädikat	Bedeutung
Klassifikation von Methoden	
$istAbstrakt(m)$	Die Methode m ist abstrakt

...

Prädikat	**Bedeutung**
$istKonstruktor(m)$	Die Methode m ist ein Konstruktor
$istLeer(m)$	Der Rumpf der Methode m ist leer
$istSchablonenMethode(m)$	m ist eine Schablonenmethode (GAMMA et al., 1995)
$istFabrikMethode(m)$	m ist eine Fabrikmethode (GAMMA et al., 1995)
$istDelegierend(m)$	Die Methode m erbringt ihre Funktionalität durch Delegation an eine andere Methode
$istAlias(m)$	Die Methode m delegiert an eine andere Methode aus derselben Klasse
$istZugriffsmethode(m)$	Die Methode m regelt den Zugriff auf ein Attribut der Klasse
$implementiert(m)$	Die Methode m implementiert eine abstrakte Methode einer Oberklasse
$erweitert(m)$	Die Methode m überschreibt eine Methodenimplementierung aus einer Oberklasse und ruft dabei die ürsprüngliche Implementierung auf
$erweitert(m)$	Die Methode m überschreibt eine Methodenimplementierung aus einer Oberklasse und ruft dabei die ürsprüngliche Implementierung *nicht* auf
$hinzugefuegt(m)$	Die Methode m ruft eine oder mehrere Methoden einer Oberklasse auf, ohne dabei eine Methode einer Oberklasse zu überschreiben
$bildetSchnittstelle(m)$	Die Methode m gehört zu Schnittstelle einer Klasse
$istInitialisierer(m)$	Die Methode m dient zur Initialisierung des Zustandes (in Form der Attribute) eines Objekts
Klassifikation von Klassen	
$istBibliotheksklasse(c)$	Die Klasse c gehört zu einer Bibliothek
$fassade(c,L)$	Die Klasse c stellt eine Fassadenklasse für die Klassen in L dar
$proxy(p,c,s)$	Die Klasse p implementiert Stellvertreterobjekte für die Objekte der Klasse c. s enthält die gemeinsame Schnittstelle von p und c
$adapter(a,c)$	Die Klasse a implementiert einen Adapter, der die Schnittstelle der Klasse c an neue Verwendungskontexte anpasst
$kompositum(k,c)$	Die Klasse k implementiert ein Kompositum zu einer Klasse c
$strategie(c,s)$	Die Klasse c implementiert im Rahmen des Entwurfsmusters *Strategie* eine konkrete Strategie für die abstrakte Schnittstelle s

...

Prädikat	**Bedeutung**
$abstrakteFabrik(a, L)$	Die Klasse *a* stellt eine *Abstrakte Fabrik* dar, *L* enthält die dazu gehörenden konkreten Fabriken
$konkreteProduke(c, L)$	Die Menge *L* enthält Produkte, die im Rahmen des Entwurfmusters *Abstrakte Fabrik* durch eine konkrete Fabrik *c* erzeugt werden können

Anhang B

Parametrisierung des Verfahrens

Unser Verfahren zur Zerlegung von Softwaresystemen kann an vielen Stellen parametrisiert werden. Wir stellen hier alle relevanten Parameter zusammen und geben die Werte an, die wir zur Bearbeitung der Fallstudien in Kapitel 8 verwendet haben.

B.1 Konstanten in Heuristiken zur Mustersuche

Konstante	Wert	Bedeutung
ξ_{OCode}	5	Anzahl von Anweisungen, die ein Methodenrumpf enthalten darf, damit er als einer gilt, der lediglich Organisationscode enthält. Solche Rümpfe sind ein Erkennungsmerkmal für Fabrikmethoden, delegierende Methoden und Zugriffsmethoden.
$\xi_{ModAttr}$	0,9	Mindestanteil der Attribute einer Klasse, die von einer Methode modifiziert werden müssen, damit die Methode als Initialisierungsmethode der Klasse gelten kann.
ξ_{Kunden}	8	Anzahl von Klassen, die als Nutzer bzw. Kunden einer Klasse C auftreten müssen, damit diese als Teil einer Bibliothek gelten kann.
$\xi_{Delegation}$	0,5	Mindestanteil der Delegationen unter den Methodenaufrufen aus einer Klasse, damit diese als Fassade gelten kann.
$\xi_{minKomplexitaet}$	3	Minimale Komplexität (Anzahl von Verzweigungen), die eine Methode haben muss, so dass sie als Implementierung einer konkreten Ausprägung eines Algorithmus im Rahmen des Entwurfsmusters *Strategie* gelten kann.

...

Konstante	Wert	Bedeutung
$\xi_{minFabrikMeth}$	4	Minimale Anzahl von Fabrikmethoden, die eine Klasse aufweisen muss, so dass sie als konkrete Fabrik im Rahmen des Entwurfsmusters *Abstrakte Fabrik* gelten kann.

B.2 Gewichtung der Abhängigkeiten

Die folgende Tabelle enthält die Belegungen für die Konstanten, die wir in Abschnitt 7.2 bei der Berechnung der Ähnlichkeitsmaße zur Erfassung der Abhängigkeiten zwischen der Klassen eines Softwaresystems verwendet haben.

Kontext	Konstante	Wert
Vererbung		
in *Proxy*	ω_{Ver_Proxy}	5
in *Kompositum*	$\omega_{Ver\_Kompositum}$	5
in *Strategie*	$\omega_{Ver\_Strategie}$	4
in *Abstrakter Fabrik*	$\omega_{Ver\_AbstrFabrik}$	0,5
Implementierung einer Schnittstelle	$\omega_{Ver\_Implementierung}$	1
Spezialisierung	$\omega_{Ver\_Spezialisierung}$	2
Anpassung einer Klasse eines Rahmenwerk	$\omega_{Ver\_Rahmenwerk}$	0,5
Aggregation		
in *Fassade*	$\omega_{Agg\_Fassade}$	5
in *Proxy*	ω_{Agg_Proxy}	4
in *Kompositum*	$\omega_{Agg\_Kompositum}$	5
zu *Fassade*	$\omega_{Agg\_zuFassade}$	0,5
zu *Proxy*	$\omega_{Agg\_zuProxy}$	0,5
in *Adapter*	$\omega_{Agg\_Adapter}$	0,5
Komposition	$\omega_{Agg\_Komposition}$	2
gewöhnliche Aggregation	$\omega_{Agg\_Standard}$	1
Methodenaufrufe		
innerhalb *Fassade*	$\omega_{Ruf\_Fassade}$	5
innerhalb *Proxy*	ω_{Ruf_Proxy}	5
innerhalb *Kompositum*	$\omega_{Ruf\_Kompositum}$	5
innerhalb *Adapter*	$\omega_{Ruf\_Adapter}$	0,5
an *Fassade*	$\omega_{Ruf\_anFassade}$	0,5
an Bibliotheksklasse	$\omega_{Ruf\_Bibliothek}$	0,5
an *Proxy*	$\omega_{Ruf\_anProxy}$	1
an Klassenmethode	$\omega_{Ruf\_Klassenmethode}$	1

...

Kontext	**Konstante**	**Wert**
von Delegationsmethode	$\omega_{Ruf\_Delegation}$	3
gewöhnlicher Aufruf	$\omega_{Ruf\_Standard}$	1
Attributzugriffe		
bei Klassenattributen	$\omega_{Att\_Klassenattr}$	1
bei Attributen aus Bibliotheksklassen	$\omega_{Att\_Bibliothek}$	1
gewöhnlicher Attributnutzung	$\omega_{Ruf\_Standard}$	2
Typdeklarationen		
Rückgabetypen in Fabrik	$\omega_{Typ\_AbstrFabrik}$	5
Rückgabetypen	$\omega_{Typ\_Rueckgabetyp}$	2
Formale Parameter	$\omega_{Typ\_Parameter}$	2
Lokale Variablen	$\omega_{Typ\_Variable}$	1

Die folgende Tabelle enthält die Gewichtung der Einzelmaße, die in die Berechnung des endgültigen Ähnlichkeitsmaßes zur Konstruktion des Abhängigkeitsgraphen eingehen (siehe Formel 7.20 in Abschnitt 7.2.5).

Maß	**Beschreibung**	**Gewichtung**
$\omega_{Ver}(c_1, c_2)$	Abhängigkeit durch eine *Vererbungsbeziehung* zwischen den Klassen c_1 und c_2	1
$\omega_{Agg}(c_1, c_2)$	Maß für die Abhängigkeit, die durch eine *Aggregation* zwischen den Klassen c_1 und c_2 entsteht	1
$\omega_{Typ}(c_1, c_2)$	Maß für die Abhängigkeiten, die durch *Typdeklarationen* des Typs c_2 (Parameter, lokale Variablen, Rückgabetypen) in Methoden der Klasse c_1 entstehen	1
$\omega_{Ruf}(c_1, c_2)$	Maß für die Abhängigkeiten, die durch *Aufrufe* von Methoden der Klasse c_2 in den Methodenrümpfen der Klasse c_1 entstehen	2
$\omega_{Att}(c_1, c_2)$	Maß für die Abhängigkeiten, die durch *Zugriffe auf Attribute* der Klasse c_2 in Methoden der Klasse c_1 entstehen	1
$\omega_{Ind}(c_1, c_2)$	Maß für die *indirekte* Abhängigkeit, die dadurch entsteht, dass c_1 und c_2 von Kunden gemeinsam verwendet wird	1

B.3 Konfiguration des *HGGA*

Die folgende Tabelle zeigt die Konfiguration des genetischen Algorithmus *HGGA*.

Freiheitsgrad bzw. Belegung	**Belegung**
Populationsgröße	$\mu = 100$
Anzahl der Generationen	$g_max = \max(\mu, \|V\|)$
Strategie zu Erzeugung der Startpopulation	
Individuen per *RMMST*	50 %
Zufällig erzeugte Individuen	50 %
Parametrisierung des *RMMST*	$\alpha^* \in [0,5 \ldots 0,9]$
Mutationen und Anwendungswahrscheinlichkeiten	
Ballungsmutation Zusammenfassen	$h_{Join} = 0,5$
Ballungsmutation Auftrennen	$h_{Split} = 0,5$
Adoptionmutation	$h_{Adopt} = 0,4$
Selektionstrategie	Turnierselektion $(\mu + \lambda)$ -Strategie, $\lambda = \frac{1}{2}\mu$
Bewertungsfunktion	Gleichungen 7.29 und 7.30, $\xi = 10$

Literaturverzeichnis

ANQUETIL, N., FOURRIER, C. und LETHBRIDGE, T. (1999): Experiments with Hierarchical Clustering Algorithms as Soft- ware Remodularization Methods. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE)*.

BALZERT, H. (1996): *Lehrbuch der Softwaretechnik – Software-Entwicklung*. Spektrum Akademischer Verlag. ISBN 3-8274-0042-2.

BÄR, H., BAUER, M., CIUPKE, O. et al. (1999): The FAMOOS Object-Oriented Reengineering Handbook. Techn. Ber., Forschungszentrum Informatik, Karlsruhe, Software Composition Group, University of Berne.

BASS, L., CLEMENTS, P. und KAZMAN, R. (1998): *Software Architecture in Practice*. Addison-Wesley.

BAUER, M. (1999a): Analyzing Software Systems Using Combinations of Metrics. In *Object-Oriented Technology, ECOOP'99 Workshop Reader, ECOOP'99 Workshops, Panels, and Posters, Lisbon, Portugal, June 14-18, 1999*, herausgegeben von Moreira, A. M. D. und Demeyer, S., Bd. 1743 von *Lecture Notes in Computer Science*. Springer. ISBN 3-540-66954-X.

BAUER, M. (1999b): Supporting Reengineering by Type Inference – A Reengineering Pattern. In *Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing*.

BAUER, M., BIEHL, M., PACHE, G. und SENG, O. (2005): Search-based improvement of subsystem decompositions. In *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2005)*. ACM Press.

BAUER, M. und CIUPKE, O. (2001): Objektorientierte Systeme unter der Lupe. In *Proceedings of the third German Workshop on Software-Reengineering*. Universität Koblenz-Landau, Institut für Informatik.

BAUER, M. und TRIFU, M. (2004): Architecture-aware Adaptive Clustering of OO Systems. In *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR 2004)*, S. 3–14. IEEE Computer Society.

BECK, K. und JOHNSON, R. (1994): Patterns Generate Architectures. In *Proceedings of the 8th Eurpean Conference on Object-Oriented Programming (ECOOP)*, Nr. 821 in Lecture Notes in Computer Science, S. 139–149. Springer.

BIEHL, M. (2004): Zerlegung von Softwaresystemen mit Genetischen Algorithmen. Studienarbeit, Institut für Programmstrukturen und Datenorganisation, Universtät Karlsruhe.

BOOCH, G. und RUMBAUGH, J. (1998): *The Unified Modeling Language Reference Manual*. Addison-Wesley. ISBN 0-201-30998-X.

BRUCKER, P. (1977): On the Complexity of Clustering Problems. In *Optimization and Operations Research*, herausgegeben von Henn, R., Korte, B. und Oettli, W., S. 45–54. Springer Verlag.

BUDD, T. (2002): *An Introduction to Object-Oriented Programming*. Addison-Wesley, dritte Aufl. ISBN 0-201-76031-2.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H. et al. (1996): *Pattern Oriented Software Architecture – A System of Patterns*. Wiley & Sons. ISBN 0-471-95869-7.

BÄR, H. (2004): *Statische Verifikation von Softwareprotokollen*. Dissertation, Universität Karlsruhe.

CHIDAMBER, S. R. und KEMERER, C. F. (1994): A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6): 476–493.

CIUPKE, O. (1999): Automatic Detection of Design Problems in Object-Oriented Reengineering. In *Proceedings of the 30st Conference on the Technology of Object-Oriented Languages and Systems (TOOLS)*. IEEE Computer Society.

CIUPKE, O. (2001): *Problemidentifikation in objektorientierten Softwarestrukturen*. Dissertation, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe.

CLARK, J., DOLADO, J., HARMAN, M. et al. (2003): Reformulating Software as a Search Problem. *IEE Proceedings*, 150(3): 161–175.

CLOCKSIN, W. und MELLISH, C. (1994): *Programmieren in Prolog*. Springer-Verlag, zweite Aufl.

COAD, P. und YOURDON, E. (1991): *Object-Oriented Design*. Prentice Hall.

COLE, R. M. (1998): *Clustering with Genetic Algorithms*. Dissertation, University of Western Australia.

DAVIS, L. (1985): Applying Adaptive Algorithms to Epistatic Domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, S. 162–164.

DEMEYER, S., DUCASSE, S. und LANZA, M. (1999): A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization. In *Proceedings on the 6th Working Conference on Reverse Engineering (WCRE)*, S. 175–186.

DEREMER, F. und KRON, H. H. (1976): Programming in the Large versus Programming in the Small. *IEEE Transactions on Software Engineering*.

DOVAL, D., MANCORIDIS, S. und MITCHELL, B. S. (1999): Automatic Clustering of Software Systems using a Genetic Algorithm. In *IEEE Proceedings of the 1999 Int. Conf. on Software Tools and Engineering Practice (STEP'99)*.

E ABREU, F. B., CALO PEREIRA, G. und SOUSA, P. (2000): A Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (CSMR)*, S. 13–22. IEEE Computer Society.

ECKEL, B. (2003): *Thinking in Patterns*. MindView, Inc.

FALKENAUER, E. (1994): A Hybrid Grouping Genetic Algorithm for Bin Packing. Techn. Ber., Research Centre for Belgian Metalworking Industry, Brussels, Belgium.

FALKENAUER, E. (1998): *Genetic Algorithms and Grouping Problems*. John Wiley & Sons.

FALKENAUER, E. und DELCHAMBRE, A. (1992): A Genetic Algorithm for Bin Packing and Line Balancing. In *Proceedings of the IEEE International Conference on Robotics and Automation*, S. 1186 – 1192.

FENTON, N. E. und PFLEEGER, S. L. (1997): *Software Metrics*. PWS Publishing Company, Boston, zweite Aufl.

FLANEGAN, D. (2005): *Java In A Nutshell*. O'Reilly, fünfte Aufl.

FRANKEL, D. S. (2003): *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley. ISBN 0-471-31920-1.

GAISSER, H. (2004): Ein Werkzeug zur Strukuruntersuchung von Delphi-Systemen. Studienarbeit, Institut für Programmstrukturen und Datenorganisation, Universtät Karlsruhe.

GAMMA, E., HELM, R., JOHNSON, R. und VLISSIDES, J. (1995): *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GANSNER, E., KOUTSOFIOS, E., NORTH, S. und VO, K. (1993): A Technique for Drawing Directed Graphs. In *IEEE Transactions of Software Engineering*, Bd. 19, S. 214–230.

GARLAN, D., ALLEN, R. und OCKERBLOOM, J. (1995): Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering (ICSE)*, S. 179–185.

GENSSLER, T. (2004): *Werkzeuggestützte Adaption objektorientierter Programme*. Dissertation, Universität Karlsruhe.

GOLDBERG, D. E. (1989): *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley.

GOLDBERG, D. E. und LINGLE, R. (1985): Alleles, Loci and the TSP. In *Proceedings of the first International Conference on Genetic Algorithms*, S. 154–159. Morgan Kaufman Publishing.

GOOS, G. (1996): *Vorlesungen über Informatik – Objektorientiertes Programmieren und Algorithmen*. Springer Verlag. ISBN 3-540-60351-2.

GOOS, G. (1997): *Vorlesungen über Informatik – Grundlagen und Funktionales Programmieren*. Springer Verlag. ISBN 3-540-62880-0.

GOOS, G. und WAITE, W. M. (1984): *Compiler Construction*. Springer Verlag.

GOSLING, J., JOY, B., STEELE, G. und BRACHA, G. (2000): *The Java Language Specification*. Addison-Wesley, zweite Aufl.

GRAHAM, R. L., KNUTH, D. E. und PATASHNIK, O. (1992): *Concrete Mathematics – a Foundation for Computer Science*. Addison-Wesley. ISBN 0-201-14236-8.

HALLER, P., KUTTRUFF, V. und SENG, O. (2003): Ein Faktenextraktor für C++ mit Unterstützung für Typrekonstruktion. *GI Softwaretechnik-Trends*.

HARMAN, M. und MAHDAVI, K. (2003): Finding Building Blocks for Software Clustering. In *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO) 2003*.

HARTIGAN, J. A. (1975): *Clustering Algorithms*. John Wiley & Sons.

HEUZEROTH, D., HOLL, T. und LÖWE, W. (2002): Combining Static and Dynamic Analyses to Detect Interaction Patterns. In *Proceedings of the Sixth International Conference on Integrated Design and Process Technology (IDPT)*.

HOLLAND, J. H. (1975): *Adaption in Natural and Artificial Systems*. University of Michigan Press.

JONES, D. E. und BELTRAMO, M. A. (1991): Solving partioning problems with genetic algorithms. In *Proceedings of the Forth International Conference on Genetic Algorithms*, herausgegeben von Belew, R. und Booker, L., S. 442–449. Morgan Kaufman Publishing.

KASTENS, U. (1990): *Übersetzerbau*. Handbuch Informatik. Oldenbourg Verlag, München.

KAUFMAN, L. und ROUSSEEUW, P. (1990): *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, New York.

KLEPPE, A., WARMER, J. und BAST, W. (2003): *MDA Explained: The Model Driven Architecture – Practice and Promise*. Addison-Wesley. ISBN 0-321-19442-X.

KOSCHKE, R. (2000): *Atomic Architectural Component Recovery for Program Understanding and Evolution*. Dissertation, Institut für Informatik, Universität Stuttgart.

KRUCHTEN, P. (1995): Architectural Blueprints – The 4+1 View Model of Software Architecture. *IEEE Software*, 12(2): 42–50.

KRUSKAL, J. (1956): On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society (AMS)*, 7: 48–50.

LANZA, M. und DUCASSE, S. (2001): A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, S. 300–311. ACM Press.

LORENZ, M. und KIDD, J. (1994): *Object-Oriented Software Metrics: A Practical Approach*. Prentice-Hall.

LUDWIG, A. (2002): *Automatische Transformation großer Softwaresysteme*. Dissertation, Universität Karlsruhe.

MANCORIDIS, S., MITCHELL, B., CHEN, Y. und GANSNER, E. (1999): Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM) 1999*, S. 50–59.

MANCORIDIS, S., MITCHELL, B., RORRES, C. et al. (1998): Using Automatic Clustering to Procude High-Level System Organisizations of Source Code. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC)*.

MARINESCU, R. (2001): Detecting Design Flaws via Metrics in Object-Oriented Systems. In *Proceedings of the 39th Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, S. 173–182.

MARINESCU, R. (2002): *Measurement and Quality in Object-Oriented Design*. Dissertation, Politehnica University of Timisoara.

MARTIN, R. C. (1996): Granularity. *C++ Report*.

MCCABE, T. (1976): A Complexity Measure. *IEEE Transactions on Software Engineering*, 2: 308–320.

MEYER, B. (1997): *Object-oriented Software Construction*. Prentice Hall, zweite Aufl. ISBN 0-13-629155-4.

MIDDENDORF, S., SINGER, R. und HEID, J. (2002): *Java – Programmierhandbuch und Referenz für die Java-2-Plattform, Standard Edition*. dpunkt-Verlag, dritte Aufl.

MITCHELL, B. (2002): *A Heuristic Search Approach to Solving the Software Clustering Problem*. Dissertation, Drexel University.

MITCHELL, B. S. und MANCORIDIS, S. (2001): Comparing the Decompositions Produced by Software Clustering Algorithms using Similarity Measurements. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM) 2001*, S. 744–753.

MONSON-HAEFEL, R. (2002): *Enterprise JavaBeans*. O'Reilly, zweite Aufl.

MURPHY, G., NOTKIN, D. und SULLIVAN, K. (2001): Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Transactions on Software Engineering*, 27(4): 364–380.

OTTMANN, T. und WIDMEYER, P. (1993): *Algorithmen und Datenstrukturen*. BI Wissenschaftsverlag, zweite Aufl. ISBN 3-411-16602-9.

PACHE, G. (2005): *Ein metrikbasiertes Suchverfahren zur automatischen Strukturverbesserung*. Diplomarbeit, Institut für Programmstrukturen und Datenorganisation, Universtät Karlsruhe.

PARNAS, D. L. (1972): On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12): 1053–1058.

PARNAS, D. L. (1974): On a Buzzword: Hierarchical Structure. In *Proceedings of the IFIP Congress '74*, S. 336–339. North Holland Publishing Company.

PARNAS, D. L. (1978): Some Software Engineering Principles. In *Infotech State of the Art Report on Structured Analysis and Design*. Infotech International.

PARNAS, D. L. (1994): Software Aging. In *Proceedings of the Sixteenth International Conference on Software Engineering (ICSE)*, S. 279–287. IEEE Computer Society.

PRECHELT, L. und KRÄMER, C. (1996): Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In *Proceedings of the Third Working Conference on Reverse Engineering (WCRE)*, S. 208–215. IEEE Computer Society.

PREE, W. (1997): *Komponentenbasierte Softwareentwicklung mit Frameworks*. dpunkt-Verlag, erste Aufl.

RAYSIDE, D., REUSS, S., HEDGES, E. und KONTOGIANNIS, K. (2000): The Effect of Call Graph Construction Algorithms for Object-Oriented Programs on Clustering. In *Proceedings of the 8th International Workshop on Program Comprehension (IWPC)*, S. 191–200.

RÜPING, A. (1997): *Softwareenwticklung mit objektorientierten Framworks*. Dissertation, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe.

SCHWEFEL, H.-P. (1995): *Evolution and Optimum Seeking*. John Wiley & Sons.

SCHÖNING, U. (1992): *Logik für Informatiker*. B.I. Wissenschaftsverlag, zweite Aufl.

SEEMANN, J. und VON GUDENBERG, J. W. (1998): Pattern-Based Design Recovery of Java Software. In *Proceedings of the Sixth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, S. 10–16. ACM Press.

SENG, O. und PACHE, G. (2004): Search-based Structure Improvement of Software Systems. In *Proceedings of the First International Workshop on Software Evolution Transformations (SET)*.

SHAW, M. und GARLAN, D. (1996): *Software Architecture*. Prentice-Hall. ISBN 0-13-182957-2.

SIMON, F. (2001): *Meßwertbasierte Qualitätssicherung – ein generisches Distanzmaß zur Erweiterung bisheriger Softwareproduktmaße*. Dissertation, Brandenburgische Technische Universität Cottbus.

TICHELAAR, S. (2001): *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. Dissertation, University of Berne, Switzerland.

TRIFU, A. (2001): *Using Cluster Analysis in the Architecture Recovery of Object-Oriented Systems*. Diplomarbeit, Institut für Programmstrukturen und Datenorganisation, Universtät Karlsruhe.

TRIFU, M. (2003): *Architecture-Aware Adaptive Clustering of Object-Oriented Systems*. Diplomarbeit, Institut für Programmstrukturen und Datenorganisation, Universtät Karlsruhe.

TZERPOS, V. und HOLT, R. (1998): Software Botryology: Automatic Clustering of Software Systems. In *Proceedings of the 9th International Workshop on Database and Expert Systems Applications*, S. 811–818. IEEE Computer Society.

TZERPOS, V. und R.C.HOLT (1999): MoJo – A Distance Metric for Software Clustering. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE)*, S. 187–193.

WEICKER, K. (2002): *Evolutionäre Algorithmen*. Teubner Verlag.

WIELEMAKER, J. (1997): *SWI-Prolog Reference Manual*. University of Amsterdam.

WIGGERTS, T. (1997): Using Clustering Algorithms in Legacy Systems Remodularization. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE)*, S. 33–43.

WIRTH, N. (1995): A Plea for Lean Software. *IEEE Computer*, 28(2).

ZAHN, C. T. (1971): Graph theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on Computers*, 20(1).

ÖSTERREICH, B. (1998): *Objektorientierte Softwareentwicklung*. Oldenbourg Verlag, vierte Aufl. ISBN 3-486-24787-5.